

C++ Primer, Fourth Edition

By Stanley B. Lippman, Josée Lajoie, Barbara E. Moon

Publisher: Addison Wesley Professional

Pub Date: **February 14, 2005**

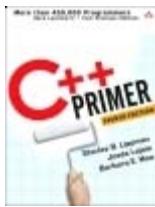
Print ISBN: 0-201-72148-1

Pages: 912

Table of Contents | Index

Overview

This popular tutorial introduction to standard C++ has been completely updated, reorganized, and rewritten to help programmers learn the language faster and use it in a more modern, effective way. Just as C++ has evolved since the last edition, so has the authors' approach to teaching it. They now introduce the C++ standard library from the beginning, giving readers the means to write useful programs without first having to master every language detail. Highlighting today's best practices, they show how to write programs that are safe, can be built quickly, and yet offer outstanding performance. Examples that take advantage of the library, and explain the features of C++, also show how to make the best use of the language. As in its previous editions, the book's authoritative discussion of fundamental C++ concepts and techniques makes it a valuable resource even for more experienced programmers. Program Faster and More Effectively with This Rewritten Classic Restructured for quicker learning, using the C++ standard library Updated to teach the most current programming styles and program design techniques Filled with new learning aids that emphasize important points, warn about common pitfalls, suggest good programming practices, and provide general usage tips Complete with exercises that reinforce skills learned Authoritative and comprehensive in its coverage The source code for the book's extended examples is available on the Web at the address below.



C++ Primer, Fourth Edition
 By Stanley B. Lippman, Josée Lajoie, Barbara E. Moo

 Publisher: **Addison Wesley Professional**
 Pub Date: **February 14, 2005**
 Print ISBN: **0-201-72148-1**
 Pages: **912**

[Table of Contents](#) | [Index](#)

[Copyright](#)

[Preface](#)

[Changes to the Fourth Edition](#)

[Structure of This Book](#)

[Acknowledgments](#)

[Chapter 1. Getting Started](#)

[Section 1.1. Writing a Simple C++ Program](#)

[Section 1.2. A First Look at Input/Output](#)

[Section 1.3. A Word About Comments](#)

[Section 1.4. Control Structures](#)

[Section 1.5. Introducing Classes](#)

[Section 1.6. The C++ Program](#)

[Chapter Summary](#)

[Keyterm Defined Terms](#)

[Part I: The Basics](#)

[Chapter 2. Variables and Basic Types](#)

[Section 2.1. Primitive Built-in Types](#)

[Section 2.2. Literal Constants](#)

[Section 2.3. Variables](#)

[Section 2.4. const Qualifier](#)

[Section 2.5. References](#)

[Section 2.6. Typedef Names](#)

[Section 2.7. Enumerations](#)

[Section 2.8. Class Types](#)

[Section 2.9. Writing Our Own Header Files](#)

[Chapter Summary](#)

[Keyterm Defined Terms](#)

[Chapter 3. Library Types](#)

[Section 3.1. Namespace using Declarations](#)

[Section 3.2. Library string Type](#)

[Section 3.3. Library vector Type](#)

[Section 3.4. Introducing Iterators](#)

[Section 3.5. Library bitset Type](#)

[Chapter Summary](#)

[Keyterm Defined Terms](#)

[Chapter 4. Arrays and Pointers](#)

[Section 4.1. Arrays](#)

[Section 4.2. Introducing Pointers](#)

[Section 4.3. C-Style Character Strings](#)

[Section 4.4. Multidimensioned Arrays](#)

[Chapter Summary](#)

[Keyterm Defined Terms](#)

Chapter 5. Expressions
Section 5.1. Arithmetic Operators
Section 5.2. Relational and Logical Operators
Section 5.3. The Bitwise Operators
Section 5.4. Assignment Operators
Section 5.5. Increment and Decrement Operators
Section 5.6. The Arrow Operator
Section 5.7. The Conditional Operator
Section 5.8. The sizeof Operator
Section 5.9. Comma Operator
Section 5.10. Evaluating Compound Expressions
Section 5.11. The new and delete Expressions
Section 5.12. Type Conversions
Chapter Summary
Keyterm Defined Terms
Chapter 6. Statements
Section 6.1. Simple Statements
Section 6.2. Declaration Statements
Section 6.3. Compound Statements (Blocks)
Section 6.4. Statement Scope
Section 6.5. The if Statement
Section 6.6. The switch Statement
Section 6.7. The while Statement
Section 6.8. The for Loop Statement
Section 6.9. The do while Statement
Section 6.10. The break Statement
Section 6.11. The continue Statement
Section 6.12. The goto Statement
Section 6.13. try Blocks and Exception Handling
Section 6.14. Using the Preprocessor for Debugging
Chapter Summary
Keyterm Defined Terms
Chapter 7. Functions
Section 7.1. Defining a Function
Section 7.2. Argument Passing
Section 7.3. The return Statement
Section 7.4. Function Declarations
Section 7.5. Local Objects
Section 7.6. Inline Functions
Section 7.7. Class Member Functions
Section 7.8. Overloaded Functions
Section 7.9. Pointers to Functions
Chapter Summary
Keyterm Defined Terms
Chapter 8. The IO Library
Section 8.1. An Object-Oriented Library
Section 8.2. Condition States
Section 8.3. Managing the Output Buffer
Section 8.4. File Input and Output
Section 8.5. String Streams
Chapter Summary
Keyterm Defined Terms
Part II: Containers and Algorithms
Chapter 9. Sequential Containers

- [Section 9.1. Defining a Sequential Container](#)
 - [Section 9.2. Iterators and Iterator Ranges](#)
 - [Section 9.3. Sequence Container Operations](#)
 - [Section 9.4. How a vector Grows](#)
 - [Section 9.5. Deciding Which Container to Use](#)
 - [Section 9.6. strings Revisited](#)
 - [Section 9.7. Container Adaptors](#)
 - [Chapter Summary](#)
 - [Keyterm Defined Terms](#)
 - [Chapter 10. Associative Containers](#)
 - [Section 10.1. Preliminaries: the pair Type](#)
 - [Section 10.2. Associative Containers](#)
 - [Section 10.3. The map Type](#)
 - [Section 10.4. The set Type](#)
 - [Section 10.5. The multimap and multiset Types](#)
 - [Section 10.6. Using Containers: Text-Query Program](#)
 - [Chapter Summary](#)
 - [Keyterm Defined Terms](#)
 - [Chapter 11. Generic Algorithms](#)
 - [Section 11.1. Overview](#)
 - [Section 11.2. A First Look at the Algorithms](#)
 - [Section 11.3. Revisiting Iterators](#)
 - [Section 11.4. Structure of Generic Algorithms](#)
 - [Section 11.5. Container-Specific Algorithms](#)
 - [Chapter Summary](#)
 - [Keyterm Defined Terms](#)
- [Part III: Classes and Data Abstraction](#)
- [Chapter 12. Classes](#)
 - [Section 12.1. Class Definitions and Declarations](#)
 - [Section 12.2. The Implicit this Pointer](#)
 - [Section 12.3. Class Scope](#)
 - [Section 12.4. Constructors](#)
 - [Section 12.5. Friends](#)
 - [Section 12.6. static Class Members](#)
 - [Chapter Summary](#)
 - [Keyterm Defined Terms](#)
 - [Chapter 13. Copy Control](#)
 - [Section 13.1. The Copy Constructor](#)
 - [Section 13.2. The Assignment Operator](#)
 - [Section 13.3. The Destructor](#)
 - [Section 13.4. A Message-Handling Example](#)
 - [Section 13.5. Managing Pointer Members](#)
 - [Chapter Summary](#)
 - [Keyterm Defined Terms](#)
 - [Chapter 14. Overloaded Operations and Conversions](#)
 - [Section 14.1. Defining an Overloaded Operator](#)
 - [Section 14.2. Input and Output Operators](#)
 - [Section 14.3. Arithmetic and Relational Operators](#)
 - [Section 14.4. Assignment Operators](#)
 - [Section 14.5. Subscript Operator](#)
 - [Section 14.6. Member Access Operators](#)
 - [Section 14.7. Increment and Decrement Operators](#)
 - [Section 14.8. Call Operator and Function Objects](#)
 - [Section 14.9. Conversions and Class Types](#)

[Chapter Summary](#)

[Keyterm Defined Terms](#)

[Part IV: Object-Oriented and Generic Programming](#)

[Chapter 15. Object-Oriented Programming](#)

[Section 15.1. OOP: An Overview](#)

[Section 15.2. Defining Base and Derived Classes](#)

[Section 15.3. Conversions and Inheritance](#)

[Section 15.4. Constructors and Copy Control](#)

[Section 15.5. Class Scope under Inheritance](#)

[Section 15.6. Pure Virtual Functions](#)

[Section 15.7. Containers and Inheritance](#)

[Section 15.8. Handle Classes and Inheritance](#)

[Section 15.9. Text Queries Revisited](#)

[Chapter Summary](#)

[Keyterm Defined Terms](#)

[Chapter 16. Templates and Generic Programming](#)

[Section 16.1. Template Definitions](#)

[Section 16.2. Instantiation](#)

[Section 16.3. Template Compilation Models](#)

[Section 16.4. Class Template Members](#)

[Section 16.5. A Generic Handle Class](#)

[Section 16.6. Template Specializations](#)

[Section 16.7. Overloading and Function Templates](#)

[Chapter Summary](#)

[Keyterm Defined Terms](#)

[Part V: Advanced Topics](#)

[Chapter 17. Tools for Large Programs](#)

[Section 17.1. Exception Handling](#)

[Section 17.2. Namespaces](#)

[Section 17.3. Multiple and Virtual Inheritance](#)

[Chapter Summary](#)

[Keyterm Defined Terms](#)

[Chapter 18. Specialized Tools and Techniques](#)

[Section 18.1. Optimizing Memory Allocation](#)

[Section 18.2. Run-Time Type Identification](#)

[Section 18.3. Pointer to Class Member](#)

[Section 18.4. Nested Classes](#)

[Section 18.5. Union: A Space-Saving Class](#)

[Section 18.6. Local Classes](#)

[Section 18.7. Inherently Nonportable Features](#)

[Chapter Summary](#)

[Keyterm Defined Terms](#)

[Appendix A. The Library](#)

[Section A.1. Library Names and Headers](#)

[Section A.2. A Brief Tour of the Algorithms](#)

[Section A.3. The IO Library Revisited](#)

[Index](#)

Copyright

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U. S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the U. S., please contact:

International Sales
international@pearsoned.com

Visit us on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data

Lippman, Stanley B.
C++ primer / Stanley B. Lippman, Josée Lajoie, Barbara E. Moo. 4th ed.
p. cm.
Includes index.
ISBN 0-201-72148-1 (pbk. : alk. paper)
1. C++ (Computer program language) I. Lajoie, Josée. II. Moo, Barbara E. III. Title
QA76.73.C153L57697 2005
005.13'3dc22 2004029301

Copyright © 2005 Objectwrite Inc., Josée Lajoie and Barbara E. Modell

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
One Lake Street
Upper Saddle River, NJ 07458

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts

First printing, February 2005

Dedication

To Bell, who makes this, and all things, possible

To Daniel and Anna, who contain virtually all possibilities

361

To Mark and Mom, for their unconditional love and support

JL

To Andy, who taught me to program and so much more

BEM

Preface

C++ Primer, Fourth Edition, provides a comprehensive introduction to the C++ language. As a primer, it provides a clear tutorial approach to the language, enhanced by numerous examples and other learning aids. Unlike most primers, it also provides a detailed description of the language, with particular emphasis on current and effective programming techniques.

本书全面介绍了 C++ 语言。作为一本入门书 (Primer) , 它以教程的形式对 C++ 语言进行清晰的讲解，并辅以丰富的示例和各种学习辅助手段。与大多数入门教程不同，本书对 C++ 语言本身进行了详尽的描述，并特别着重介绍了目前通行的、行之有效的程序设计技巧。

Countless programmers have used previous editions of *C++ Primer* to learn C++. In that time C++ has matured greatly. Over the years, the focus of the language and of C++ programmers has grown beyond a concentration on run-time efficiency to focus on ways of making programmers more efficient. With the widespread availability of the standard library, it is possible to use and learn C++ more effectively than in the past. This revision of the *C++ Primer* reflects these new possibilities.

无数程序员曾使用本书的前几个版本学习 C++ , 在此期间 C++ 也逐渐发展成熟。这些年来, C++ 语言的发展方向以及 C++ 程序员的关注点, 已经从以往注重运行时的效率, 转到千方百计地提高程序员的编程效率上。随着标准库的广泛可用, 我们现在能够比以往任何时候更高效地学习和使用 C++ 。本书这一版本充分体现了这一点。

Chapter 1. Getting Started

第一章 快速入门

CONTENTS

Section 1.1 Writing a Simple C++ Program	2
Section 1.2 A First Look at Input/Output	5
Section 1.3 A Word About Comments	10
Section 1.4 Control Structures	11
Section 1.5 Introducing Classes	20
Section 1.6 The C++ Program	25
Chapter Summary	28
Defined Terms	28

This chapter introduces most of the basic elements of C++: built-in, library, and class types; variables; expressions; statements; and functions. Along the way, we'll briefly explain how to compile and execute a program.

本章介绍 C++ 的大部分基本要素：内置类型、库类型、类类型、变量、表达式、语句和函数。在这一过程中还会简要说明如何编译和运行程序。

Having read this chapter and worked through the exercises, the reader should be able to write, compile, and execute simple programs. Subsequent chapters will explain in more detail the topics introduced here.

读者读完本章内容并做完练习，就应该可以编写、编译和执行简单的程序。后面的章节会进一步阐明本章所介绍的主题。

Learning a new programming language requires writing programs. In this chapter, we'll write a program to solve a simple problem that represents a common data-processing task: A bookstore keeps a file of transactions, each of which records the sale of a given book. Each transaction contains an ISBN (International Standard Book Number, a unique identifier assigned to most books published throughout the world), the number of copies sold, and the price at which each copy was sold. Each transaction looks like

要学会一门新的程序语言，必须实际动手编写程序。在这一章，我们将缩写程序解决一个简单的数据处理问题：某书店以文件形式保存其每一笔交易。每一笔交易记录某本书的销售情况，含有 ISBN（国际标准书号，世界上每种图书的唯一标识符）、销售册数和销售单价。每一笔交易形如：

0-201-70353-X 4 24.99

where the first element is the ISBN, the second is the number of books sold, and the last is the sales price. Periodically the bookstore owner reads this file and computes the number of copies of each title sold, the total revenue from that book, and the average sales price. We want to supply a program do these computations.

第一个元素是 ISBN，第二个元素是销售的册数，最后是销售单价。店主定期地查看这个文件，统计每本书的销售册数、总销售收入以及平均售价。我们要编写程序来进行这些计算。

Before we can write this program we need to know some basic features of C++. At a minimum we'll need to know how to write, compile, and execute a simple program. What must this program do? Although we have not yet designed our solution, we know that the program must

在编写这个程序之前，必须知道 C++ 的一些基本特征。至少我们要知道怎么样编写、编译和执行简单的程序。这个程序要做什么呢？虽然还没有设计解决方案，但是我们知道程序必须：

- Define variables
定义变量。
- Do input and output
实现输入和输出。
- Define a data structure to hold the data we're managing
定义数据结构来保存要处理的数据。

- Test whether two records have the same ISBN
测试是否两条记录具有相同的 ISBN。
- Write a loop that will process every record in the transaction file
编写循环。处理交易文件中的每一条记录。

We'll start by reviewing these parts of C++ and then write a solution to our bookstore problem.

我们将首先考察 C++ 的这些部分，然后编写书店问题的解决方案。

Team LiB

[◀ PREVIOUS](#) [NEXT ▶](#)

Part I: The Basics

第一部分 基本语言

Programming languages have distinctive features that determine the kinds of applications for which they are well suited. They also share many fundamental attributes. Essentially all languages provide:

各种程序设计语言都具有许多独具特色的特征，这些特征决定了用每种语言适合开发哪些类型的应用程序。程序设计语言也有一些共同的特征。基本上所有的语言都要提供下列特征：

- Built-in data types such as integers, characters, and so forth
内置数据类型，如整型、字符型等。
- Expressions and statements to manipulate values of these types
表达式和语句：表达式和语句用于操纵上述类型的值。
- Variables, which let us give names to the objects we use
变量：程序员可以使用变量对所用的对象命名。
- Control structures, such as `if` or `while`, that allow us to conditionally execute or repeat a set of actions
控制结构：如 `if` 或 `while`，程序员可以使用控制结构有条件地执行或重复执行一组动作。
- Functions that let us abstract actions into callable units of computation
函数：程序员可以使用函数把行为抽象成可调用的计算单元。

Most modern programming languages supplement this basic set of features in two ways: They let programmers extend the language by defining their own data types, and they provide a set of library routines that define useful functions and data types not otherwise built into the language.

大多数现代程序语言都采用两种方式扩充上述基本特征集：允许程序员通过自定义数据类型扩展该语言；提供一组库例程，这些例程定义了一些并非内置在语言中的实用函数和数据类型。

In C++, as in most programming languages, the type of an object determines what operations can be performed on it. Depending on the type of the objects involved, a statement might or might not be legal. Some languages, notably Smalltalk and Python, check the types involved in expressions at run time. In contrast, C++ is a statically typed language; type-checking is done at compile time. As a consequence, the compiler must be told the type of every name used in the program before that name can be used.

和大多数程序设计语言一样，C++ 中对象的类型决定了该对象可以执行的操作。语句正确与否取决于该语句中对象的类型。一些程序设计语言，特别是 Smalltalk 和 Python，在运行时才检查语句中对象的类型。相反，C++ 是静态类型 (statically typed) 语言，在编译时执行类型检查。结果是程序中使用某个名字之前，必须先告知编译器该名字的类型。

C++ provides a set of built-in data types, operators to manipulate those types, and a small set of statements for program flow control. These elements form an alphabet with which many large, complex real-world systems can and have been written. At this basic level, C++ is a simple language. Its expressive power arises from its support for mechanisms that allow the programmer to define new data structures.

C++ 提供了一组内置数据类型、操纵这些类型的操作符和一组少量的程序流控制语句。这些元素形成了一个“词汇表”，使用这个词汇表可以而且已经编写出许多大型、复杂的实际系统。从这个基本层面来看，C++ 是一门简单的语言。C++ 的表达能力是通过支持一些允许程序员定义新数据结构的机制来提升的。

Perhaps the most important feature in C++ is the class, which allows programmers to define their own data types. In C++ such types are sometimes called "class types" to distinguish them from the types that are built into the language. Some languages let programmers define data types that specify only what data make up the type. Others, like C++, allow programmers to define types that include operations as well as data. One of the primary design goals of C++ is to let programmers define their own types that are as easy to use as the built-in types. The Standard C++ library uses these features to implement a rich library of class types and associated functions.

可能 C++ 中最重要的特征是类 (class)，程序员可以使用类自定义数据类型。C++ 中这些类型有时也称为“类类型 (class type)”，以区别于语言的内置类型。有一些语言允许程序员定义的数据类型只能指定组成该类型的数据。包括 C++ 在内的其他语言允许程序员定义的类型不仅有数据还包括操作。C++ 主要设计目标之一就是允许程序员自定义类型，而且这些类型和内置类型一样易于使用。C++ 标准库复用这些特征，实现了一个具有丰富类型和相关函数的标准库。

The first step in mastering C++ learning the basics of the language and library is the topic of [Part I](#). [Chapter 2](#) covers the built-in data types and looks briefly at the mechanisms for defining our own new types. [Chapter 3](#) introduces two of the most fundamental library types: `string` and `vector`. Arrays, which are covered in [Chapter 4](#), are a lower-level data structure built into C++ and many other languages. Arrays are similar to `vectors` but harder to use. [Chapters 5](#) through [7](#) cover expressions, statements, and functions. This part concludes in [Chapter 8](#), which covers the most important facilities from the IO library.

掌握 C++ 的第一步是学习语言的基本知识和标准库，这正是[第一部分](#)介绍的内容。[第二章](#)介绍了内置数据类型，并简单探讨了自定义新类型的机制。[第三章](#)引入了两种最基本的标准库类型：`string` 和 `vector`。[第四章](#)介绍了数组，数组是一种低级的数据结构，内置于 C++ 和许多其他语言。数组类似于 `vector` 对象，但较难使用。[第五章](#)到[第七章](#)介绍了表达式、语句和函数。[第八章](#)是[第一部分](#)的最后一章，介绍了 IO 标准库中最重要的设施。

CONTENTS

[Chapter 2 Variables and Basic Types](#)

[Chapter 3 Library Types](#)

[Chapter 4 Arrays and Pointers](#)

[Chapter 5 Expressions](#)

[Chapter 6 Statements](#)

[Chapter 7 Functions](#)

[Chapter 8 The IO Library](#)

Team LiB

◀ PREVIOUS NEXT ▶

Part II: Containers and Algorithms

第二部分：容器和算法

We've said that C++ is about efficient programming with abstractions. The Standard Library is a good example: The library defines a number of container classes and a family of generic algorithms that let us write programs that are succinct, abstract, and efficient. The library worries about bookkeeping details in particular, taking care of memory management so that our programs can worry about the actual problems we need to solve.

C++ 提供了使用抽象进行高效率编程的方式。标准库就是一个很好的例子：标准库定义了许多容器类以及一系列泛型算法，使程序员可以更简洁、抽象和有效地编写程序。这样可以让标准库操心那些繁琐的细节，特别是内存管理，我们的程序只需要关注要解决的实际问题就行了。

In [Chapter 3](#) we introduced the `vector` container type. We'll learn more in [Chapter 9](#) about `vector` and the other sequential container types provided by the library. We'll also cover more operations provided by the `string` type. We can think of a `string` as a special kind of container that contains only characters. The `string` type supports many, but not all, of the container operations.

[第三章](#)介绍了 `vector` 容器类型。我们将会在[第九章](#)进一步探讨 `vector` 和其他顺序容器类型，而且还会学习 `string` 类型提供的更多操作，这些容器类型都是由标准库定义的。我们可将 `string` 视为仅包含字符的特殊容器，`string` 类型提供大量（但并不是全部）的容器操作。

The library also defines several associative containers. Elements in an associative container are ordered by key rather than sequentially. The associative containers share many operations with the sequential containers and also define operations that are specific to the associative containers. The associative containers are covered in [Chapter 10](#).

标准库还定义了几种关联容器。关联容器中的元素不是顺序排列，而是按键（key）排序的。关联容器共享了许多顺序容器提供的操作，此外，还定义了自己特殊的操作。我们将在[第十章](#)学习相关的内容。

[Chapter 11](#) introduces the generic algorithms. The algorithms typically operate on a range of elements from a container or other sequence. The algorithms library offers efficient implementations of various classical algorithms, such as searching, sorting, and other common tasks. For example, there is a `copy` algorithm, which copies elements from one sequence to another; `find`, which looks for a given element; and so on. The algorithms are generic in two ways: They can be applied to different kinds of containers, and those containers may contain elements of most types.

[第十一章](#)介绍了泛型算法，这些算法通常作用于容器或序列中某一范围的元素。算法库提供了各种各样经典算法的有效实现，像查找、排序及其他常见的算法任务。例如，[复制](#)算法将一个序列中所有元素复制到另一个序列中；[查找](#)算法则用于寻找一个指定元素，等等。泛型算法中，所谓“泛型（generic）”指的是两个方面：这些算法可作用于各种不同的容器类型，而这些容器又可以容纳多种不同类型的元素。

The library is designed so that the container types provide a common interface: If two containers offer a similar operation, then that operation will be defined identically for both containers. For example, all the containers have an operation to return the number of elements in the container. All the containers name that operation `size`, and they all define a type named `size_type` that is the type of the value returned by `size`. Similarly, the algorithms have a consistent interface. For example, most algorithms operate on a range of elements specified by a pair of iterators.

为容器类型提供通用接口是设计库的目的。如果两种容器提供相似的操作，则为它们定义的这个操作应该完全相同。例如，所有容器都有返回容器内元素个数的操作，于是所有容器都将操作命名为 `size`，并将 `size` 返回值的类型都指定为 `size_type` 类型。类似地，算法具有一致的接口。例如，大部分算法都作用在由一对迭代器指定的元素范围内。

Because the container operations and algorithms are defined consistently, learning the library becomes easier: Once you understand how an operation works, you can apply that same operation to other containers. More importantly, this commonality of interface leads to more flexible programs. It is often possible to take a program written to use one container type and change it to use a different container without having to rewrite code. As we'll see, the containers offer different performance tradeoffs, and the ability to change container types can be valuable when fine-tuning the performance of a system.

容器提供的操作和算法是一致定义的，这使得学习标准库更容易：只需理解一个操作如何工作，就能将该操作应用于其他的容器。更重要的是，接口的一致性使程序变得更灵活。通常不需要重新编写代码，就可以将一段使用某种容器类型的程序修改为使用不同容器实现。正如我们所看到的，容器提供了不同的性能折衷方案，可以改变容器类型对优化系统性能来说颇有价值。

CONTENTS

目录

[Chapter 9 Sequential Containers](#)

[Chapter 10 Associative Containers](#)

Part III: Classes and Data Abstraction

第三部分：类和数据抽象

Classes are central to most C++ programs: Classes let us define our own types that are customized for the problems we need to solve, resulting in applications that are easier to write and understand. Well-designed class types can be as easy to use as the built-in types.

在大多数 C++ 程序中，类都是至关重要的：我们能够使用类来定义为要解决的问题定制的数据类型，从而得到更加易于编写和理解的应用程序。设计良好的类类型可以像内置类型一样容易使用。

A class defines data and function members: The data members store the state associated with objects of the class type, and the functions perform operations that give meaning to the data. Classes let us separate implementation and interface. The interface specifies the operations that the class supports. Only the implementor of the class need know or care about the details of the implementation. This separation reduces the bookkeeping aspects that make programming tedious and error-prone.

类定义了数据成员和函数成员：数据成员用于存储与该类类型的对象相关联的状态，而函数成员则负责执行赋予数据意义的操作。通过类我们能够将实现和接口分离，用接口指定类所支持的操作，而实现的细节只需类的实现者了解或关心。这种分离可以减少使编程冗长乏味和容易出错的那些繁琐工作。

Class types often are referred to as *abstract data types*. An abstract data type treats the data (state) and operations on that state as a single unit. We can think abstractly about what the class does, rather than always having to be aware of how the class operates. Abstract data types are fundamental to both object-oriented and generic programming.

类类型常被称为抽象数据类型（*abstract data types*）。抽象数据类型将数据（即状态）和作用于状态的操作视为一个单元。我们可以抽象地考虑类该做什么，而无须知道类如何去完成这些操作。抽象数据类型是面向对象编程和泛型编程的基础。

[Chapter 12](#) begins our detailed coverage of how classes are defined. This chapter covers topics fundamental to any use of classes: class scope, data hiding, and constructors. It also introduces some new class features: friends, uses of the implicit `this` pointer, and the role of `static` and `mutable` members.

[第十二章](#)开始详细地介绍如何定义类，包括类的使用中非常基本的主题：类作用域、数据隐藏和构造函数。此外，还介绍了类的一些新特征：友元、使用隐含的 `this` 指针，以及静态（`static`）和可变（`mutable`）成员的作用。

Classes in C++ control what happens when objects are initialized, copied, assigned, and destroyed. In this respect, C++ differs from many other languages, many of which do not give class designers the ability to control these operations. [Chapter 13](#) covers these topics.

C++ 中的类能够控制在初始化、复制、赋值和销毁对象时发生的操作。在这方面，C++ 不同于许多其他语言，它们大多没有赋予类设计者控制这些操作的能力。[第十三章](#)讨论了这些主题。

[Chapter 14](#) looks at operator overloading, which allows operands of class types to be used with the built-in operators. Operator over-loading is one of the ways whereby C++ lets us create new types that are as intuitive to use as are the built-in types. This chapter also presents another special kind of class [member function](#)conversion functionswhich define implicit conversions from objects of class type. The compiler applies these conversions in the same contextsand for the same reasonsas it does with conversions among the built-in types.

[第十四章](#)考察了操作符重载，允许将类类型的操作数与内置操作符一起使用。利用操作符重载，在 C++ 中创建新的类型，就像创建内置类型一样。此外，还介绍了另一种特殊的类成员函数——转换函数，这种函数定义了类类型对象之间的隐式转换。编译器应用这些转换就像它们是在内置类型之间发生的转换一样。

CONTENTS

[Chapter 12 Classes](#)

[第十二章 类](#)

[Chapter 13 Copy Control](#)

[第十三章 复制控制](#)

[Chapter 14 Overloaded Operations and Conversions](#)

[第十四章 重载操作符与转换](#)

Part IV: Object-Oriented and Generic Programming

第四部分：面向对象编程与泛型编程

[Part IV](#) extends the discussion of [Part III](#) by covering how C++ supports object-oriented and generic programming.

[第四部分](#) 继续第三部分的讨论，涵盖 C++ 支持面向对象编程和泛型编程。[Part III](#)

[Chapter 15](#) covers inheritance and dynamic binding. Along with data abstraction, inheritance and dynamic binding are fundamental to [object-oriented programming](#).

[第十五章](#)讨论继承和动态绑定。继承和动态绑定与数据抽象一起成为[面向对象编程 \(object-oriented programming\)](#) 的基础。

[Chapter 16](#) covers function and class templates. Templates let us write generic classes and functions that are independent of type.

[第十六章](#)讨论函数模板和类模板。模板使我们能够编写独立于具体类型的泛型类和泛型函数。

Writing our own object-oriented or generic types requires a fairly good understanding of C++. Fortunately, we can use OO and generic types without understanding the details of how to build them. In fact, the standard library uses the facilities we'll study in [Chapters 15](#) and [16](#) extensively, and we've used the library types and algorithms without needing to know how they are implemented. Readers, therefore, should understand that [Part IV](#) covers advanced topics. Writing templates or object-oriented classes requires a good understanding of the basics of C++ and a good grasp of how to define more basic classes.

编写自己的面向对象类型或泛型类型需要对 C++ 的充分理解，幸运的是，我们可以使用面向对象和泛型类型而无需了解它们的构建细节。事实上，标准库广泛使用了将在[第十五章](#)和[第十六章](#)中介绍的设施，而且我们已经在不了解实现细节的情况下使用了标准库中的类型和算法。因此，读者应该理解[第四部分](#)涵盖的一些高级主题。编写模板或面向对象的类，需要充分理解 C++ 的基本原理并且很好地掌握怎样定义更基本的类。

CONTENTS

目录

[Chapter 15 Object-Oriented Programming](#)

[第十五章](#) 面向对象编程

[Chapter 16 Templates and Generic Programming](#)

[第十六章](#) 模板与泛型编程

Part V: Advanced Topics

第五部分 高级主题

[Part V](#) covers additional features that, although useful in the right context, are not needed by every C++ programmer. These features divide into two clusters: those that are useful for large-scale problems and those that are applicable to specialized problems rather than general ones.

[第五部分](#)涵盖了一些高级特征，虽然它们在适当情况下是有用的，但不是每个 C++ 程序员都需要。这些特征分为两类：用于规模问题，以及应用于特殊问题的。

[Chapter 17](#) covers exception handling, namespaces, and multiple inheritance. These features tend to be most useful in the context of large-scale problems.

[第十七章](#)涵盖了异常处理、命名空间和多重继承。这些特征在大规模问题的环境中最有用。

Even programs simple enough to be written by a single author can benefit from exception handling, which is why we introduced the basics of exception handling in [Chapter 6](#). However, the need to deal with unexpected run-time errors tends to be more important and harder to manage in problems that require large programming teams. In [Chapter 17](#) we review some additional useful exception-handling facilities. We also look in more detail at how exceptions are handled and the implications of exceptions on resource allocation and destruction. We also show how we can define and use our own exception classes.

即使是可由一个开发人员编写的简单程序也可以从异常处理中获益，这是我们之所以要在[第六章](#)介绍异常处理基础知识的原因。但是，在需要大的编程团队的问题中，处理不可预期的运行时错误变得更加重要，更难管理。[第十七章](#)将讨论另外一些有用的异常处理设施，还将更详细地介绍怎样处理异常、资源分配和回收异常的含义，以及怎样定义和使用自己的异常类。

Large-scale applications often use code from multiple independent vendors. Combining independently developed libraries would be difficult (if not impossible) if vendors had to put the names they define into a single namespace. Independently developed libraries would almost inevitably use names in common with one another; a name defined in one library would conflict with the use of that name in another library. To avoid name collisions, we can define names inside a `namespace`.

大规模应用程序经常使用来自多个独立供应商的代码，如果将供应商定义的名字都放在一个命名空间，组合独立开发的库即使不是完全不可能，将是很困难的。独立开发的库几乎不可避免地会使用彼此相同的名字，一个库中定义的名字可能会与其他库中的相同名冲突。为了避免名字冲突，可以将名字定义在 `namespace` 内。

Right from the beginning of this book we have used namespaces. Whenever we use a name from the standard library, we are using a name defined in the namespace named `std`. [Chapter 17](#) shows how we can define our own namespaces.

其实从本书一开始我们已经使用了命名空间。只要用到标准库中的名字，其实都是在使用名为 `std` 的命名空间中的定义的名字。[第十七章](#)将介绍怎样自定义命名空间。

[Chapter 17](#) closes by looking at an important but infrequently used language feature: multiple inheritance. Multiple inheritance is most useful for fairly complicated inheritance hierarchies.

[第十七章](#)的最后介绍了一个重要但不常用的语言特征——多重继承。多重继承对于相当复杂的继承层次最为有用。

[Chapter 18](#) covers several specialized tools and techniques. These tools and techniques are applicable to particular kinds of problems.

[第十八章](#)讨论了几个特殊的工具和技术，这些工具和技术可应用于某些特定类型的问题。

The first part of [Chapter 18](#) shows how classes can define their own optimized memory management. We next look at C++ support for run-time type identification (RTTI). These facilities let us determine the actual type of an object at run-time.

[第十八章](#)的第一节介绍了类怎样自定义优化内存管理，接着介绍 C++ 对运行时类型识别 (RTTI) 的支持，这种设施使我们能够在运行时确定对象的实际类型。

Next, we look at how we can define and use pointers to class members. Pointers to class members differ from pointers to ordinary data or functions. Ordinary pointers only vary based on the type of the object or function. Pointers to members must also reflect the class to which the member belongs.

接下来，介绍怎样定义和使用类成员的指针。类成员的指针不同于指向普通数据或函数的指针，普通指针只根据对象或函数的类型而变化，而成员的指针还必须反映成员所属的类。

We then look at three additional aggregate types: unions, nested classes, and local classes.

然后，介绍另外三种聚合类型：联合、嵌套类和局部类。

The chapter closes by looking briefly at a collection of features that are inherently nonportable: the `volatile` qualifier, bit-fields, and linkage directives.

[第十八章](#)最后简要介绍了一些固有的不可移植的特征: `volatile` 限定符、位域和链接指示。

CONTENTS

[Chapter 17 Tools for Large Programs](#)

[Chapter 18 Specialized Tools and Techniques](#)

Team LiB

◀ PREVIOUS NEXT ▶

Appendix A. The Library

附录 A. 标准库

CONTENTS

<u>Section A.1</u> Library Names and Headers	810
<u>Section A.2</u> A Brief Tour of the Algorithms	811
<u>Section A.3</u> The IO Library Revisited	825

This appendix presents additional useful details about the library. We'll start by collecting in one place the names we used from the standard library. [Table A.1](#) on the next page lists each name and the header that defines that name.

本附录提供了标准库更多有用的细节。首先将标准库中的名字集中在一起，表 A.1 列出每个名字以及定义该名字的头文件。

[Chapter 11](#) covered the library algorithms. That chapter illustrated how some of the more common algorithms are used, and described the architecture that underlies the algorithms library. In this Appendix, we list all the algorithms, organized by the kinds of operations they perform.

第十一章讨论标准库算法，那一章举例说明了怎样使用一些比较常用的算法，并描述了算法库的体系结构。本附录中将列出所有算法，按它们执行的操作种类组织。

We close by examining some additional IO library capabilities: format control, unformatted IO, and random access on files. Each IO type defines a collection of format states and associated functions to control those states. These format states give us finer control over how input and output works. The IO we've done has all been formatted—the input and output routines know about the types we use and format the data on input or output accordingly. There are also unformatted IO functions that deal with the stream at the `char` level, doing no interpretation of the data. In [Chapter 8](#) we saw that the `fstream` type can read and write the same file. In this Appendix, we'll see how to do so.

本附录的最后考察了另一些 IO 库的功能，包括格式控制、未格式化的 IO 和文件随机访问。每个 IO 类型定义了格式状态和控制这些状态的相关函数集合。使用这些格式状态我们能够更好地控制输入和输出的工作。我们已经做过的 IO 都是格式化的——输入和输出例程了解使用的类型，并据此格式化输入和输出数据。还有未格式化的 IO 函数，它们在 `char` 级别处理流，不解释数据。[第八章](#) 介绍过 `fstream` 类型能读写同一文件，本附录将介绍怎样做到这一点。

Team LiB[◀ PREVIOUS](#) [NEXT ▶](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

Team LiB[◀ PREVIOUS](#) [NEXT ▶](#)

Changes to the Fourth Edition

第四版的改动

In this edition, we have completely reorganized and rewritten the *C++ Primer* to highlight modern styles of C++ programming. This edition gives center stage to using the standard library while deemphasizing techniques for low-level programming. We introduce the standard library much earlier in the text and have reformulated the examples to take advantage of library facilities. We have also streamlined and reordered the presentation of language topics.

为了体现现代 C++ 编程风格，我们重新组织并重写了本书。书中不再强调低层编程技术，而把中心转向标准库的使用。书中很早就开始介绍标准库，示例也已经重新改写，充分利用了标准库设施。我们也对语言主题叙述的先后次序进行了重新编排，使讲解更加流畅。

In addition to restructuring the text, we have incorporated several new elements to enhance the reader's understanding. Each chapter concludes with a Chapter Summary and glossary of Defined Terms, which recap the chapter's most important points. Readers should use these sections as a personal checklist: If you do not understand a term, restudy the corresponding part of the chapter.

除重新组织内容外，为了便于读者理解，我们还增加了几个新的环节。每一章都新增了“小结”和“术语”，概括本章要点。读者可以利用这些部分进行自我检查；如果发现还有不理解的概念，可以重新学习该章中的相关部分。

We've also incorporated a number of other learning aids in the body of the text:

书中还加入了下述几种学习辅助手段：

- Important terms are indicated in **bold**; important terms that we assume are already familiar to the reader are indicated in ***bold italics***. Each term appears in the chapter's Defined Terms section.

重要术语用黑体表示，我们认为读者已经熟悉的重要术语则用楷体表示。这些术语都会出现在“术语”部分。

- Throughout the book, we highlight parts of the text to call attention to important aspects of the language, warn about common pitfalls, suggest good programming practices, and provide general usage tips. We hope that these notes will help readers more quickly digest important concepts and avoid common pitfalls.

书中用特殊版式突出标注的文字，是为了向读者提醒语言的重要特征，警示常见的错误，标明良好的编程实践，列出通用的使用技巧。希望这些标注可以帮助读者更快地消化重要概念，避免犯常见错误。

- To make it easier to follow the relationships among features and concepts, we provide extensive forward and backward cross-references.

为了更易于理解各种特征或概念间的关系，书中大量使用了前后交叉引用。

- We have provided sidebar discussions that focus on important concepts and supply additional explanations for topics that programmers new to C++ often find most difficult.

对于某些重要概念和 C++ 新手最头疼的问题，我们进行了额外的讨论和解释。这部分也以特殊版式标出。

- Learning any programming language requires writing programs. To that end, the primer provides extensive examples throughout the text. Source code for the extended examples is available on the Web at the following URL:

学习任何程序设计语言都需要编写程序。因此，本提供了大量的示例。所有示例的源代码可以从下列网址获得：

http://www.awprofessional.com/cpp_primer

What hasn't changed from earlier versions is that the book remains a comprehensive tutorial introduction to C++. Our intent is to provide a clear, complete and correct guide to the language. We teach the language by presenting a series of examples, which, in addition to explaining language features, show how to make the best use of C++. Although knowledge of C (the language on which C++ was originally based) is not assumed, we do assume the reader has programmed in a modern block-structured language.

万变不离其宗，本书保持了前几版的特色，仍然是一部全面介绍 C++ 的教程。我们的目标是提供一本清晰、全面、准确的指南性读物。我们通过讲解一系列示例来教授 C++ 语言，示例除了解释语言特征外，还展示了如何善用这门语言。虽然读者不需要事先学过 C 语言（C++ 最初的基础）的知识，但我们假定读者已经掌握了一种现代结构化语言。

Structure of This Book

本书结构

C++ Primer provides an introduction to the International Standard on C++, covering both the language proper and the extensive library that is part of that standard. Much of the power of C++ comes from its support for programming with abstractions. Learning to program effectively in C++ requires more than learning new syntax and semantics. Our focus is on how to use the features of C++ to write programs that are safe, that can be built quickly, and yet offer performance comparable to the sorts of low-level programs often written in C.

本介绍了 C++ 国际标准，既涵盖语言的特征，又讲述了也是标准组成部分的丰富标准库。C++ 的强大很大程度上来自它支持抽象程序设计。要学会用 C++ 高效地编程，只是掌握句法和语义是远远不够的。我们的重点在于，教会读者怎样利用 C++ 的特性，快速地写出安全的而且性能可与 C 语言低层程序相媲美的程序。

C++ is a large language and can be daunting to new users. Modern C++ can be thought of as comprising three parts:

C++ 是一种大型的编程语言，这可能会吓倒一些新手。现代 C++ 可以看成由以下三部分组成：

- The low-level language, largely inherited from C
低级语言，多半继承自 C。
- More advanced language features that allow us to define our own data types and to organize large-scale programs and systems
更高级的语言特征，用户可以借此定义自己的数据类型，组织大规模的程序和系统。
- The standard library, which uses these advanced features to provide a set of useful data structures and algorithms
标准库，使用上述高级特征提供一整套有用的数据结构和算法。

Most texts present C++ in this same order: They start by covering the low-level details and then introduce the more advanced language features. They explain the standard library only after having covered the entire language. The result, all too often, is that readers get bogged down in issues of low-level programming or the complexities of writing type definitions and never really understand the power of programming in a more abstract way. Needless to say, readers also often do not learn enough to build their own abstractions.

多数 C++ 教材按照下面的顺序展开：先讲低级细节，再介绍更高级的语言特征；在讲完整个语言后才开始解释标准库。结果往往使读者纠缠于低级的程序设计问题和复杂类型定义的编写等细节，而不能真正领会抽象编程的强大，更不用说学到足够的知识去创建自己的抽象了。

In this edition we take a completely different tack. We start by covering the basics of the language and the library together. Doing so allows you, the reader, to write significant programs. Only after a thorough grounding in using the library and writing the kinds of abstract programs that the library allows do we move on to those features of C++ that will enable you to write your own abstractions.

本版中我们独辟蹊径。一开始就讲述语言的基础知识和标准库，这样读者就可以写出比较大的有实际意义的程序来。透彻阐释了使用标准库（并且用标准库编写了各种抽象程序）的基础知识之后，我们才进入下一步，学习用 C++ 的其他高级特征来编写自己的抽象。

[Parts I](#) and [II](#) cover the basic language and library facilities. The focus of these parts is to learn how to write C++ programs and how to use the abstractions from the library. Most C++ programmers need to know essentially everything covered in this portion of the book.

[第一](#)和[第二](#)部分讨论语言的基础知识和标准库设施。其重点在于学会如何编写 C++ 程序，如何使用标准库提供的抽象设施。大部分 C++ 程序员需要了解本书这两部分的内容。

In addition to teaching the basics of C++, the material in [Parts I](#) and [II](#) serves another important purpose. The library facilities are themselves abstract data types written in C++. The library can be defined using the same class-construction features that are available to any C++ programmer. Our experience in teaching C++ is that by first using well-designed abstract types, readers find it easier to understand how to build their own types.

除了讲解基础知识以外，这两部分还有另外一个重要的意图。标准库设施本身是用 C++ 编写的抽象数据类型，定义标准库使用的是任何 C++ 程序员都能使用的构造类的语言特征。我们教授 C++ 的经验说明，一开始就使用设计良好的抽象类型，读者会更容易理解如何建立自己的类型。

[Parts III](#) through [V](#) focus on how we can write our own types. [Part III](#) introduces the heart of C++: its support for classes. The class mechanism provides the basis for writing our own abstractions. Classes are also the foundation for object-oriented and generic programming, which we cover in [Part IV](#). The *Primer* concludes with [Part V](#), which covers advanced features that are of most use in structuring large, complex systems.

[第三](#)到[第五](#)部分着重讨论如何编写自己的类型。第三部分介绍 C++ 的核心，即对类的支持。类机制提供了编写自定义抽象的基础。类也是第四部分中讨论的面向对象编程和泛型编程的基础。全书正文的最后是第五部分，这一部分讨论了一些高级特征，它们在构建大型复杂系统时最为常用。

Acknowledgments

致谢

As in previous editions of this *Primer*, we'd like to extend our thanks to Bjarne Stroustrup for his tireless work on C++ and for his friendship to these authors throughout most of that time. We'd also like to thank Alex Stepanov for his original insights that led to the containers and algorithms that form the core of the standard library. Finally, our thanks go to the C++ Standards committee members for their hard work in clarifying, refining, and improving C++ over many years.

与前几版一新，我们要感谢 Bjarne Stroustrup，他不知疲倦地从事着 C++ 方面的工作，他与我们的深厚友情由来已久。我们还要感谢 Alex Stepanov，正是他最初凭借敏锐的洞察力创造了容器和算法的概念，这些概念最终形成了标准库的核心。此外，我们要感谢 C++ 标准委员会的所有成员，他们多年来为 C++ 澄清概念、细化标准和改进功能付出了艰苦的努力。

We also extend our deep-felt thanks to our reviewers, whose helpful comments on multiple drafts led us to make improvements great and small throughout the book: Paul Abrahams, Michael Ball, Mary Dageforde, Paul DuBois, Matt Greenwood, Matthew P. Johnson, Andrew Koenig, Nevin Liber, Bill Locke, Robert Murray, Phil Romanik, Justin Shaw, Victor Shtern, Clovis Tondo, Daveed Vandevoorde, and Steve Vinoski.

我们要衷心地感谢本书审稿人，他们审阅了我们的多份书稿，帮助我们对本书进行了无数大大小小的修改。他们是 Paul Abrahams, Michael Ball, Mary Dageforde, Paul DuBois, Matt Greenwood, Matthew P. Johnson, Andrew Koenig, Nevin Liber, Bill Locke, Robert Murray, Phil Romanik, Justin Shaw, Victor Shtern, Clovis Tondo, Daveed Vandevoorde 和 Steve Vinoski。

This book was typeset using LATEX and the many packages that accompany the LATEX distribution. Our well-justified thanks go to the members of the LATEX community, who have made available such powerful typesetting tools.

The examples in this book have been compiled on the GNU and Microsoft compilers. Our thanks to their developers, and to those who have developed all the other C++ compilers, thereby making C++ a reality.

书中所有示例都已通过 GNU 和微软编译器的编译。感谢他们的开发者和所有开发其他 C++ 编译器的人，是他们使 C++ 变成现实。

Finally, we thank the fine folks at Addison-Wesley who have shepherded this edition through the publishing process: Debbie Lafferty, our original editor, who initiated this edition and who had been with the *Primer* from its very first edition; Peter Gordon, our new editor, whose insistence on updating and streamlining the text have, we hope, greatly improved the presentation; Kim Boedigheimer, who keeps us all on schedule; and Tyrrell Albaugh, Jim Markham, Elizabeth Ryan, and John Fuller, who saw us through the design and production process.

最后，感谢 的工作人员，他们引领了这一版的整个出版过程：——我们最初的编辑，是他提出出版本书的新版，他从本书最初版本起就一直致力于本书；——我们的新编辑，他坚持更新和精简本书内容，极大地改进了这一版本；——他保证了我们所有人能按进度工作；还有 、 和，他们和我们一起经历了整个设计和制作过程。

1.1. Writing a Simple C++ Program

1.1. 编写简单的 C++ 程序

Every C++ program contains one or more *functions*, one of which must be named `main`. A function consists of a sequence of *statements* that perform the work of the function. The operating system executes a program by calling the function named `main`. That function executes its constituent statements and returns a value to the operating system.

每个 C++ 程序都包含一个或多个函数，而且必须有一个命名为 `main`。函数由执行函数功能的语句序列组成。操作系统通过调用 `main` 函数来执行程序，`main` 函数则执行组成自己的语句并返回一个值给操作系统。

Here is a simple version of `main` does nothing but return a value:

下面是一个简单的 `main` 函数，它不执行任何功能，只是返回一个值：

```
int main()
{
    return 0;
}
```

The operating system uses the value returned by `main` to determine whether the program succeeded or failed. A return value of 0 indicates success.

操作系统通过 `main` 函数返回的值来确定程序是否成功执行完毕。返回 0 值表明程序成功执行完毕。

The `main` function is special in various ways, the most important of which are that the function must exist in every C++ program and it is the (only) function that the operating system explicitly calls.

`main` 函数在很多方面都比较特别，其中最重要的是每个 C++ 程序必须含有 `main` 函数，且 `main` 函数是（唯一）被操作系统显式调用的函数。

We define `main` the same way we define other functions. A function definition specifies four elements: the *return type*, the *function name*, a (possibly empty) *parameter list* enclosed in parentheses, and the *function body*. The `main` function may have only a restricted set of parameters. As defined here, the parameter list is empty; [Section 7.2.6](#) (p. 243) will cover the other parameters that can be defined for `main`.

定义 `main` 函数和定义其他函数一样。定义函数必须指定 4 个元素：返回类型、函数名、圆括号内的形参表（可能为空）和函数体。`main` 函数的形参数个数是有限的。本例中定义的 `main` 函数形参表为空。[第 7.2.6 节](#) 将介绍 `main` 函数中可以定义的其他形参。

The `main` function is required to have a return type of `int`, which is the type that represents integers. The `int` type is a *built-in type*, which means that the type is defined by the language.

`main` 函数的返回值必须是 `int` 型，该类型表示整数。`int` 类型是内置类型，即该类型是由 C++ 语言定义的。

The final part of a function definition, the function body, is a *block* of statements starting with an open *curly brace* and ending with a close curly:

函数体函数定义的最后部分，是以花括号开始并以花括号结束的语句块：

```
{     return 0;
}
```

The only statement in our program is a `return`, which is a statement that terminates a function.

例中唯一的语句就是 `return`，该语句终止函数。



Note the semicolon at the end of the `return` statement. Semicolons mark the end of most statements in C++. They are easy to overlook, but when forgotten can lead to mysterious compiler error messages.

注意 `return` 语句后面的分号。在 C++ 中多数语句以分号作为结束标记。分号很容易被忽略，而漏写分号将会导致莫名其妙的编译错误信息。

When the `return` includes a value such as `0`, that value is the return value of the function. The value returned must have the same type as the return type of the function or be a type that can be converted to that type. In the case of `main` the return type must be `int`, and the value `0` is an `int`.

当 `return` 带上一个值（如 `0`）时，这个值就是函数的返回值。返回值类型必须和函数的返回类型相同，或者可以转换成函数的返回类型。对于 `main` 函数，返回类型必须是 `int` 型，`0` 是 `int` 型的。

On most systems, the return value from `main` is a status indicator. A return value of `0` indicates the successful completion of `main`. Any other return value has a meaning that is defined by the operating system. Usually a nonzero return indicates that an error occurred. Each operating system has its own way of telling the user what `main` returned.

Section 1.1. Writing a Simple C++ Program

在大多数系统中，`main` 函数的返回值是一个状态指示器。返回值 `0` 往往表示 `main` 函数成功执行完毕。任何其他非零的返回值都有操作系统定义的含义。通常非零返回值表明有错误出现。每一种操作系统都有自己的方式告诉用户 `main` 函数返回什么内容。

1.1.1. Compiling and Executing Our Program

1.1.1. 编译与执行程序

Having written the program, we need to compile it. How you compile a program depends on your operating system and compiler. For details on how your particular compiler works, you'll need to check the reference manual or ask a knowledgeable colleague.

程序编写完后需要进行编译。如何进行编译，与具体操作系统和编译器有关。你需要查看有关参考手册或者询问有经验的同事，以了解所用的编译器的工作细节。

Many PC-based compilers are run from an integrated development environment (IDE) that bundles the compiler with associated build and analysis tools. These environments can be a great asset in developing complex programs but require a fair bit of time to learn how to use effectively. Most of these environments include a point-and-click interface that allows the programmer to write a program and use various menus to compile and execute the program. Learning how to use such environments is well beyond the scope of this book.

许多基于 PC 的编译器都在集成开发环境（IDE）中运行，IDE 将编译器与相关的构建和分析工具绑定在一起。这些环境在开发复杂程序时非常有用，但掌握起来需要花费一点时间。通常这些环境包含点击式界面，程序员在此界面下可以编写程序，并使用各种菜单来编译与执行程序本书不介绍怎样使用这些环境。

Most compilers, including those that come with an IDE, provide a command-line interface. Unless you are already familiar with using your compiler's IDE, it can be easier to start by using the simpler, command-line interface. Using the command-line interface lets you avoid the overhead of learning the IDE before learning the language.

大多数编译器，包括那些来自 IDE 的，都提供了命令行界面。除非你已经很熟悉你的 IDE，否则从使用简单的命令行界面开始可能更容易些。这样可以避免在学习语言之前得先去学习 IDE。

Program Source File Naming Convention

程序源文件命名规范

Whether we are using a command-line interface or an IDE, most compilers expect that the program we want to compile will be stored in a file. Program files are referred to as **source files**. On most systems, a source file has a name that consists of two parts: a file name for example, `prog1` and a file suffix. By convention, the suffix indicates that the file is a program. The suffix often also indicates what language the program is written in and selects which compiler to run. The system that we used to compile the examples in this book treats a file with a suffix of `.cc` as a C++ program and so we stored this program as

不管我们使用命令行界面还是 IDE，大多数编译器希望待编译的程序保存在文件中。程序文件称作**源文件**。大多数系统中，源文件的名字由文件名（如 `prog1`）和文件后缀两部分组成。依据惯例，文件后缀表明该文件是程序。文件后缀通常也表明程序是用什么语言编写的，以及选择哪一种编译器运行。我们用来编译本书实例的系统将带有后缀 `.cc` 的文件视为 C++ 程序，因此我们将该程序保存为：

```
prog1.cc
```

The suffix for C++ program files depends on which compiler you're running. Other conventions include

C++ 程序文件的后缀与运行的具体编译器有关。其他的形式还包括。

```
prog1.cxx  
prog1.cpp  
prog1.cp  
prog1.C
```

Invoking the GNU or Microsoft Compilers

调用 GNU 或微软编译器

The command used to invoke the C++ compiler varies across compilers and operating systems. The most common compilers are the GNU compiler and the Microsoft Visual Studio compilers. By default the command to invoke the GNU compiler is `g++`:

调用 C++ 编译器的命令因编译器和操作系统的不同而不同，常用的编译器是 GNU 编译器和微软 Visual Studio 编译器。调用 GNU 编译器的默认命令是 `g++`：

```
$ g++ prog1.cc -o prog1
```

where `$` is the system prompt. This command generates an executable file named `prog1` or `prog1.exe`, depending on the

Section 1.1. Writing a Simple C++ Program

operating system. On UNIX, executable files have no suffix; on Windows, the suffix is .exe. The -o prog1 is an argument to the compiler and names the file in which to put the executable file. If the -o prog1 is omitted, then the compiler generates an executable named a.out on UNIX systems and a.exe on Windows.

这里的 \$ 是系统提示符。这个命令产生一个为 prog1 或 prog1.exe 的可执行文件。在 UNIX 系统下，可执行文件没有后缀；而在 Windows 下，后缀为 .exe。-o prog1 是编译器参数以及用来存放可执行文件的文件名。如果省略 -o prog1，那么编译器在 UNIX 系统下产生名为 a.out 而在 Windows 下产生名为 a.exe 的可执行文件。

The Microsoft compilers are invoked using the command cl:

微软编译器采用命令 cl 来调用：

```
C:\directory> cl -GX prog1.cpp
```

where C:directory> is the system prompt and directory is the name of the current directory. The command to invoke the compiler is cl, and -GX is an option that is required for programs compiled using the command-line interface. The Microsoft compiler automatically generates an executable with a name that corresponds to the source file name. The executable has the suffix .exe and the same name as the source file name. In this case, the executable is named prog1.exe.

这里的 C:directory> 是系统提示符，directory 是当前目录名。cl 是调用编译器的命令。-GX 是一个选项，该选项在使用命令行界面编译器程序时是必需的。微软编译器自动产生与源文件同名的可执行文件，这个可执行文件具有 .exe 后缀且与源文件同名。本例中，可执行文件命名为 prog1.exe。

For further information consult your compiler's user's guide.

更多的信息请参考你的编译器用户指南。

Running the Compiler from the Command Line

从命令行编译器

If we are using a command-line interface, we will typically compile a program in a console window (such as a shell window on a UNIX system or a Command Prompt window on Windows). Assuming that our main program is in a file named prog1.cc, we might compile it by using a command such as:

如果使用命令行界面，一般在控制台窗口（例如 UNIX 的 shell 窗口或 Windows 的命令提示窗口）编译程序。假设 main 程序在名为 prog1.cc 的文件中，可以使用如下命令来编译：

```
$ CC prog1.cc
```

where CC names the compiler and \$ represents the system prompt. The output of the compiler is an executable file that we invoke by naming it. On our system, the compiler generates the executable in a file named a.exe. UNIX compilers tend to put their executables in a file named a.out. To run an executable we supply that name at the command-line prompt:

这里 CC 是编译器命令名，\$ 表示系统提示符。编译器输出一个可执行文件，我们可以按名调用这个可执行文件。在我们的系统中，编译器产生一个名为 a.exe 的可执行文件。UNIX 编译器则会将可执行文件放到一个名为 a.out 的文件中。要运行可执行文件，可在命令提示符处给出该文件名：

```
$ a.exe
```

executes the program we compiled. On UNIX systems you sometimes must also specify which directory the file is in, even if it is in the current directory. In such cases, we would write

执行编译过的程序。在 UNIX 系统中，即使在当前目录，有时还必须指定文件所在的目录。这种情况下，键入：

```
$ ./a.exe
```

The “.” followed by a slash indicates that the file is in the current directory.

“.”后面的斜杠表明文件处于当前目录下。

The value returned from main is accessed in a system-dependent manner. On both UNIX and Windows systems, after executing the program, you must issue an appropriate echo command. On UNIX systems, we obtain the status by writing

访问 main 函数的返回值的方式和系统有关。不论 UNIX 还是 Windows 系统，执行程序后，必须发出一个适当的 echo 命令。UNIX 系统中，通过键入如下命令获取状态：

```
$ echo $?
```

To see the status on a Windows system, we write

要在 Windows 系统下查看状态，键入

```
C:\directory> echo %ERRORLEVEL%
```

Exercises Section 1.1.1

Exercise 1.1: Review the documentation for your compiler and determine what file naming convention it uses. Compile and run the `main` program from page 2.

查看所用的编译器文档，了解它所用的文件命名规范。编译并运行本节的 `main` 程序。

Exercise 1.2: Change the program to return `-1`. A return value of `-1` is often treated as an indicator that the program failed. However, systems vary as to how (or even whether) they report a failure from `main`. Recompile and rerun your program to see how your system treats a failure indicator from `main`.

修改程序使其返回 `-1`。返回值 `-1` 通常作为程序运行失败的指示器。然而，系统不同，如何（甚至是否）报告 `main` 函数运行失败也不同。重新编译并再次运行程序，看看你的系统如何处理 `main` 函数的运行失败指示器。

1.2. A First Look at Input/Output

1.2. 初窥输入/输出

C++ does not directly define any statements to do input or output (IO). Instead, IO is provided by the [standard library](#). The IO library provides an extensive set of facilities. However, for many purposes, including the examples in this book, one needs to know only a few basic concepts and operations.

C++ 并没有直接定义进行输入或输出 (IO) 的任何语句，这种功能是由[标准库](#)提供的。IO 库提供了大量的设施。然而，对许多应用，包括本书的例子而言，编程者只需要了解一些基本概念和操作。

Most of the examples in this book use the [iostream library](#), which handles formatted input and output. Fundamental to the [iostream library](#) are two types named [istream](#) and [ostream](#), which represent input and output streams, respectively. A stream is a sequence of characters intended to be read from or written to an IO device of some kind. The term "stream" is intended to suggest that the characters are generated, or consumed, sequentially over time.

本书的大多数例子都使用了处理格式化输入和输出的 [iostream](#) 库。[iostream](#) 库的基础是两种命名为 [istream](#) 和 [ostream](#) 的类型，分别表示输入流和输出流。流是指要从某种 IO 设备上读入或写出的字符序列。术语“流”试图说明字符是随着时间顺序生成或消耗的。

1.2.1. Standard Input and Output Objects

1.2.1. 标准输入与输出对象

The library defines four IO objects. To handle input, we use an object of type [istream](#) named [cin](#) (pronounced "see-in"). This object is also referred to as the [standard input](#). For output, we use an [ostream](#) object named [cout](#) (pronounced "see-out"). It is often referred to as the [standard output](#). The library also defines two other [ostream](#) objects, named [cerr](#) and [clog](#) (pronounced "see-err" and "see-log," respectively). The [cerr](#) object, referred to as the [standard error](#), is typically used to generate warning and error messages to users of our programs. The [clog](#) object is used for general information about the execution of the program.

标准库定义了 4 个 IO 对象。处理输入时使用命名为 [cin](#) (读作 see-in) 的 [istream](#) 类型对象。这个对象也称为[标准输入](#)。处理输出时使用命名为 [cout](#) (读作 see-out) 的 [ostream](#) 类型对象，这个对象也称为[标准输出](#)。标准库还定义了另外两个 [ostream](#) 对象，分别命名为 [cerr](#) 和 [clog](#) (分别读作“see-err”和“see-log”)。[cerr](#) 对象又叫作[标准错误](#)，通常用来输出警告和错误信息给程序的使用者。而 [clog](#) 对象用于产生程序执行的一般信息。

Ordinarily, the system associates each of these objects with the window in which the program is executed. So, when we read from [cin](#), data is read from the window in which the program is executing, and when we write to [cout](#), [cerr](#), or [clog](#), the output is written to the same window. Most operating systems give us a way of redirecting the input or output streams when we run a program. Using redirection we can associate these streams with files of our choosing.

一般情况下，系统将这些对象与执行程序的窗口联系起来。这样，当我们从 [cin](#) 读入时，数据从执行程序的窗口读入，当写到 [cin](#)、[cerr](#) 或 [clog](#) 时，输出写至同一窗口。运行程序时，大部分操作系统都提供了重定向输入或输出流的方法。利用重定向可以将这些流与所选择的文件联系起来。

1.2.2. A Program that Uses the IO Library

1.2.2. 一个使用IO库的程序

So far, we have seen how to compile and execute a simple program, although that program did no work. In our overall problem, we'll have several records that refer to the same ISBN. We'll need to consolidate those records into a single total, implying that we'll need to know how to add the quantities of books sold.

到目前为止，我们已经明白如何编译与执行简单的程序，虽然那个程序什么也不做。在开篇的书店问题中，有一些记录含有相同的 ISBN，需要将这些记录进行汇总，也就是说需要弄清楚如何累加已售出书籍的数量。

To see how to solve part of that problem, let's start by looking at how we might add two numbers. Using the IO library, we can extend our [main](#) program to ask the user to give us two numbers and then print their sum:

为了弄清楚如何解决这个问题，我们先来看应如何把两数相加。我们可以使用 IO 库来扩充 [main](#) 程序，要求用户给出两个数，然后输出它们的和：

```
#include <iostream>
int main()
{
```

Section 1.2. A First Look at Input/Output

```
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2;
    std::cout << "The sum of " << v1 << " and " << v2
        << " is " << v1 + v2 << std::endl;
    return 0;
}
```

This program starts by printing

程序首先在用户屏幕上显示提示语:

Enter two numbers:

on the user's screen and then waits for input from the user. If the user enters

然后程序等待用户输入。如果用户输入

3 7

followed by a newline, then the program produces the following output:

跟着一个换行符，则程序产生下面的输出:

The sum of 3 and 7 is 10

The first line of our program is a [preprocessor directive](#):

程序的第一行是一个预处理指示:

```
#include <iostream>
```

which tells the compiler that we want to use the `iostream` library. The name inside angle brackets is a [header](#). Every program that uses a library facility must include its associated header. The `#include` directive must be written on a single line—the name of the header and the `#include` must appear on the same line. In general, `#include` directives should appear outside any function. Typically, all the `#include` directives for a program appear at the beginning of the file.

告诉编译器要使用 `iostream` 库。尖括号里的名字是一个。[头文件](#)。程序使用库工具时必须包含相关的头文件。`#include` 指示必须单独写成一行——头文件名和 `#include` 必须在同一行。通常，`#include` 指示应出现在任何函数的外部。而且习惯上，程序的所有 `#include` 指示都在文件开头部分出现。

Writing to a Stream

写入到流

The first statement in the body of `main` executes an [expression](#). In C++ an expression is composed of one or more operands and (usually) an operator. The expressions in this statement use the [output operator](#) (the `<<` operator) to print the prompt on the standard output:

`main` 函数体中第一条语句执行了一个[表达式](#)。C++ 中，一个表达式由一个或几个操作数和通常是一个操作符组成。该语句的表达式使用[输出操作符](#)（`<<` 操作符），在标准输出上输出提示语:

```
std::cout << "Enter two numbers:" << std::endl;
```

This statement uses the output operator twice. Each instance of the output operator takes two operands: The left-hand operand must be an `ostream` object; the right-hand operand is a value to print. The operator writes its right-hand operand to the `ostream` that is its left-hand operand.

这个语句用了两次输出操作符。每个输出操作符实例都接受两个操作数：左操作数必须是 `ostream` 对象；右操作数是要输出的值。操作符将其右操作数写到作为其左操作数的 `ostream` 对象。

In C++ every expression produces a result, which typically is the value generated by applying an operator to its operands. In the case of the output operator, the result is the value of its left-hand operand. That is, the value returned by an output operation is the output stream itself.

C++ 中，每个表达式都会产生一个结果，通常是将操作符作用到其操作数所产生的值。当操作符是输出操作符时，结果是左操作数的值。也就是说，输出操作返回的值是输出流本身。

The fact that the operator returns its left-hand operand allows us to chain together output requests. The statement that prints our prompt is equivalent to

既然输出操作符返回的是其左操作数，那么我们就可以将输出请求链接在一起。输出提示语的那条语句等价于

```
(std::cout << "Enter two numbers:") << std::endl;
```

Because `(std::cout << "Enter two numbers:")` returns its left operand, `std::cout`, this statement is equivalent to

因为 `((std::cout << "Enter two numbers:"))` 返回其左操作数 `std::cout`，这条语句等价于

```
std::cout << "Enter two numbers:";
std::cout << std::endl;
```

Section 1.2. A First Look at Input/Output

`endl` is a special value, called a **manipulator**, that when written to an output stream has the effect of writing a newline to the output and flushing the **buffer** associated with that device. By flushing the buffer, we ensure that the user will see the output written to the stream immediately.

`endl` 是一个特殊值，称为**操纵符**，将它写入输出流时，具有输出换行的效果，并刷新与设备相关联的**缓冲区**。通过刷新缓冲区，用户可立即看到写入到流中的输出。



Programmers often insert print statements during debugging. Such statements should always flush the stream. Forgetting to do so may cause output to be left in the buffer if the program crashes, leading to incorrect inferences about where the program crashed.

程序员经常在调试过程中插入输出语句，这些语句都应该刷新输出流。忘记刷新输出流可能会造成输出停留在缓冲区中，如果程序崩溃，将会导致程序错误推断崩溃位置。

Using Names from the Standard Library

使用标准库中的名字

Careful readers will note that this program uses `std::cout` and `std::endl` rather than just `cout` and `endl`. The prefix `std::` indicates that the names `cout` and `endl` are defined inside the **namespace** named `std`. Namespaces allow programmers to avoid inadvertent collisions with the same names defined by a library. Because the names that the standard library defines are defined in a namespace, we can use the same names for our own purposes.

细心的读者会注意到这个程序中使用的是 `std::cout` 和 `std::endl`，而不是 `cout` 和 `endl`。前缀 `std::` 表明 `cout` 和 `endl` 是定义在命名空间 `std` 中的。使用命名空间程序员可以避免与库中定义的名字相同而引起无意冲突。因为标准库定义的名字是定义在命名空间中，所以我们按自己的意图使用相同的名字。

One side effect of the library's use of a namespace is that when we use a name from the library, we must say explicitly that we want to use the name from the `std` namespace. Writing `std::cout` uses the **scope operator** (the `::` operator) to say that we want to use the name `cout` that is defined in the namespace `std`. We'll see in [Section 3.1](#) (p. 78) a way that programs often use to avoid this verbose syntax.

标准库使用命名空间的副作用是，当我们使用标准库中的名字时，必须显式地表达出使用的是命名空间 `std` 下的名字。`std::cout` 的写法使用了作用域操作符 (**scope operator**, `::` 操作符)，表示使用的是定义在命名空间 `std` 中的 `cout`。我们将在[第 3.1 节](#)学习到程序中经常使用的避免这种冗长句法的方法。

Reading From a Stream

读入流

Having written our prompt, we next want to read what the user writes. We start by defining two **variables** named `v1` and `v2` to hold the input: 在输出提示语后，将读入用户输入的数据。先定义两个名为 `v1` 和 `v2` 的**变量**来保存输入：

```
int v1, v2;
```

We define these variables as type `int`, which is the built-in type representing integral values. These variables are **uninitialized**, meaning that we gave them no initial value. Our first use of these variables will be to read a value into them, so the fact that they have no initial value is okay.

将这些变量定义为 `int` 类型，`int` 类型是一种代表整数值的内置类型。这些变量**未初始化**，表示没有赋给它们初始值。这些变量在首次使用时会读入一个值，因此可以没有初始值。

The next statement

下一条语句读取输入：

```
std::cin >> v1 >> v2;
```

reads the input. The **input operator** (the `>>` operator) behaves analogously to the output operator. It takes an `istream` as its left-hand operand and an object as its right-hand operand. It reads from its `istream` operand and stores the value it read in its right-hand operand. Like the output operator, the input operator returns its left-hand operand as its result. Because the operator returns its left-hand operand, we can combine a sequence of input requests into a single statement. In other words, this input operation is equivalent to

`输入操作符 (>> 操作符)` 行为与输出操作符相似。它接受一个 `istream` 对象作为其左操作数，接受一个对象作为其右操作数，它从 `istream` 操作数读取数据并保存到右操作数中。像输出操作符一样，输入操作符返回其左操作数作为结果。由于输入操作符返回其左操作数，我们可以将输入请求序列合并成单个语句。换句话说，这个输入操作等价于：

```
std::cin >> v1;
std::cin >> v2;
```

Section 1.2. A First Look at Input/Output

The effect of our input operation is to read two values from the standard input, storing the first in `v1` and the second in `v2`.

输入操作的效果是从标准输入读取两个值，将第一个存放在 `v1` 中，第二个存放在 `v2` 中。

Completing the Program

完成程序

What remains is to print our result:

剩下的就是要输出结果：

```
std::cout << "The sum of " << v1 << " and " << v2  
      << " is " << v1 + v2 << std::endl;
```

This statement, although it is longer than the statement that printed the prompt, is conceptually no different. It prints each of its operands to the standard output. What is interesting is that the operands are not all the same kinds of values. Some operands are [string literals](#), such as

这条语句虽然比输出提示语的语句长，但概念上没什么区别。它将每个操作数输出到标准输出。有趣的是操作数并不都是同一类型的值，有些操作数是[字符串字面值](#)。例如

"The sum of "

and others are various `int` values, such as `v1`, `v2`, and the result of evaluating the arithmetic expression:

其他是各种 `int` 值，如 `v1`、`v2` 以及对算术表达式

`v1 + v2`

The `iostream` library defines versions of the input and output operators that accept all of the built-in types.

求值的结果。`iostream` 库定义了接受全部内置类型的输入输出操作符版本。



When writing a C++ program, in most places that a space appears we could instead use a newline. One exception to this rule is that spaces inside a string literal cannot be replaced by a newline. Another exception is that spaces are not allowed inside preprocessor directives.

在写 C++ 程序时，大部分出现空格符的地方可用换行符代替。这条规则的一个例外是字符串字面值中的空格符不能用换行符代替。另一个例外是空格符不允许出现在预处理指示中。

Key Concept: Initialized and Uninitialized Variables

关键概念：已初始化变量和未初始化变量

Initialization is an important concept in C++ and one to which we will return throughout this book.

在 C++ 中，初始化是一个非常重要的概念，对它的讨论将贯穿本书始终。

Initialized variables are those that are given a value when they are defined. Uninitialized variables are not given an initial value:

已初始化变量是指变量在定义时就给定一个值。未初始化变量则未给定初始值：

```
int val1 = 0;      // initialized  
int val2;        // uninitialized
```

It is almost always right to give a variable an initial value, but we are not required to do so. When we are certain that the first use of a variable gives it a new value, then there is no need to invent an initial value. For example, our first nontrivial program on page 6 defined uninitialized variables into which we immediately read values.

给变量一个初始值几乎总是正确的，但不要求必须这样做。当我们确定变量在第一次使用时会赋一个新值，那就不需要创建初始值。例如，在本节开始我们的第一个有意义的程序中，定义了未初始化变量，并立即读取值给它们。

When we define a variable, we should give it an initial value unless we are certain that the initial value will be overwritten before the variable is used for any other purpose. If we cannot guarantee that the variable will be reset before being read, we should initialize it.

定义变量时，应该给变量赋初始值，除非确定将变量用于其他意图之前会覆盖这个初值。如果不能保证读取变量之前重置变量，就应该初始化变量。

Exercises Section 1.2.2

Exercise

1.3: Write a program to print "Hello, World" on the standard output.

编一个程序，在标准输出上打印“Hello, World”。

Exercise

1.4: Our program used the built-in addition operator, `+`, to generate the sum of two numbers. Write a program that uses the multiplication operator, `*`, to generate the product of two numbers.

我们的程序利用内置的加法操作符“`+`”来产生两个数的和。编写程序，使用乘法操作符“`*`”产生两个数的积。

Exercise

1.5: We wrote the output in one large statement. Rewrite the program to use a separate statement to print each operand.

我们的程序使用了一条较长的输出语句。重写程序，使用单独的语句打印每一个操作数。

Exercise

1.6: Explain what the following program fragment does:

解释下面的程序段：

```
std::cout << "The sum of " << v1;
<< " and " << v2;
<< " is " << v1 + v2
<< std::endl;
```

Is this code legal? If so, why? If not, why not?

这段代码合法吗？如果合法，为什么？如果不合法，又为什么？

1.3. A Word About Comments

1.3. 关于注释

Before our programs get much more complicated, we should see how C++ handles [comments](#). Comments help the human readers of our programs. They are typically used to summarize an algorithm, identify the purpose of a variable, or clarify an otherwise obscure segment of code. Comments do not increase the size of the executable program. The compiler ignores all comments.

在程序变得更复杂之前，我们应该明白C++如何处理[注释](#)。注释可以帮助其他人阅读程序，通常用于概括算法、确认变量的用途或者阐明难以理解的代码段。注释并不会增加可执行程序的大小，编译器会忽略所有注释。



In this book, we italicize comments to make them stand out from the normal program text. In actual programs, whether comment text is distinguished from the text used for program code depends on the sophistication of the programming environment.

本书中，注释排成斜体以区别于一般程序文本。实际程序中，注释文本是否区别于程序代码文本取决于编程环境是否完善。

There are two kinds of comments in C++: single-line and paired. A single-line comment starts with a double slash (//). Everything to the right of the slashes on the current line is a comment and ignored by the compiler.

C++ 中有单行注释和成对注释两种类型的注释。单行注释以双斜线 (//) 开头，行中处于双斜线右边的内容是注释，被编译器忽略。

The other delimiter, the comment pair /* */, is inherited from the C language. Such comments begin with a /* and end with the next */. The compiler treats everything that falls between the /* and */ as part of the comment:

另一种定界符，注释对 (/* */)，是从 C 语言继承过来的。这种注释以“/*”开头，以“*/”结尾。编译器把落入注释对“/* */”之间的内容作为注释：

```
#include <iostream>
/* Simple main function: Read two numbers and write their sum */
int main()
{
    // prompt user to enter two numbers
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;           // uninitialized
    std::cin >> v1 >> v2; // read input
    return 0;
}
```

A comment pair can be placed anywhere a tab, space, or newline is permitted. Comment pairs can span multiple lines of a program but are not required to do so. When a comment pair does span multiple lines, it is often a good idea to indicate visually that the inner lines are part of a multi-line comment. Our style is to begin each line in the comment with an asterisk, thus indicating that the entire range is part of a multi-line comment.

任何允许有制表符、空格或换行符的地方都允许放注释对。注释对可跨越程序的多行，但不是一定要如此。当注释跨多行时，最好能直观地指明每一行都是注释。我们的风格是在注释的每一行以星号开始，指明整个范围是多行注释的一部分。

Programs typically contain a mixture of both comment forms. Comment pairs generally are used for multi-line explanations, whereas double slash comments tend to be used for half-line and single-line remarks.

程序通常混用两种注释形式。注释对一般用于多行解释，而双斜线注释则常用于半行或单行的标记。

Too many comments intermixed with the program code can obscure the code. It is usually best to place a comment block above the code it explains.

太多的注释混入程序代码可能会使代码难以理解，通常最好是将一个注释块放在所解释代码的上方。

Comments should be kept up to date as the code itself changes. Programmers expect comments to remain accurate and so believe them, even when other forms of system documentation are known to be out of date. An incorrect comment is worse than no comment at all because it may mislead a subsequent reader.

代码改变时，注释应与代码保持一致。程序员即使知道系统其他形式的文档已经过期，还是会信任注释，认为它会是正确的。错误的注释比没有注释更糟，因为它会误导后来者。

Comment Pairs Do Not Nest

注释对不可嵌套

Section 1.3. A Word About Comments

A comment that begins with `/*` always ends with the next `*/`. As a result, one comment pair cannot occur within another. The compiler error message(s) that result from this kind of program mistake can be mysterious and confusing. As an example, compile the following program on your system:

```
#include <iostream>
/*
 * comment pairs /* */ cannot nest.
 * "cannot nest" is considered source code,
 * as is the rest of the program
 */
int main()
{
    return 0;
}
```

When commenting out a large section of a program, it can seem easiest to put a comment pair around a region that you want to omit temporarily. The trouble is that if that code already has a comment pair, then the newly inserted comment will terminate prematurely. A better way to temporarily ignore a section of code is to use your editor to insert single-line comment at the beginning of each line of code you want to ignore. That way, you need not worry about whether the code you are commenting out already contains a comment pair.

当注释掉程序的一大部分时，似乎最简单的办法就是在要临时忽略的区域前后放一个注释对。问题是如果那段代码已经有了注释对，那么新插入的注释对将提前终止。临时忽略一段代码更好的方法，是用编辑器在要忽略的每一行代码前面插入单行注释。这样，你就无需担心要注释的代码是否已包含注释对。

Exercises Section 1.3

Exercise Compile a program that has incorrectly nested comments.

1.7:

编译有不正确嵌套注释的程序。

Exercise Indicate which, if any, of the following output statements, are legal.

1.8:

指出下列输出语句哪些（如果有）是合法的。

```
std::cout << "/*";
std::cout << "*/";
std::cout << /* ** */ */;
```

After you've predicted what will happen, test your answer by compiling a program with these three statements. Correct any errors you encounter.

预测结果，然后编译包含上述三条语句的程序，检查你的答案。纠正所遇到的错误。

1.4. Control Structures

1.4. 控制结构

Statements execute sequentially: The first statement in a function is executed first, followed by the second, and so on. Of course, few programs including the one we'll need to write to solve our bookstore problem can be written using only sequential execution. Instead, programming languages provide various control structures that allow for more complicated execution paths. This section will take a brief look at some of the control structures provided by C++. [Chapter 6](#) covers statements in detail.

语句总是顺序执行的：函数的第一条语句首先执行，接着是第二条，依次类推。当然，少数程序——包括我们将要编写的解决书店问题的程序——可以仅用顺序执行语句编写。事实上，程序设计语言提供了多种控制结构支持更为复杂的执行路径。本节将简要地介绍 C++ 提供的控制结构，[第六章](#)再详细介绍各种语句。

1.4.1. The `while` Statement

1.4.1. `while` 语句

A [while statement](#) provides for iterative execution. We could use a `while` to write a program to sum the numbers from 1 through 10 inclusive as follows:

`while` 语句提供了迭代执行功能。可以用 `while` 语句编写一个如下所示的从 1 到 10（包括 10）的求和程序：

```
#include <iostream>
int main()
{
    int sum = 0, val = 1;
    // keep executing the while until val is greater than 10
    while (val <= 10) {
        sum += val; // assigns sum + val to sum
        ++val;       // add 1 to val
    }
    std::cout << "Sum of 1 to 10 inclusive is "
           << sum << std::endl;
    return 0;
}
```

This program when compiled and executed will print:

编译并执行后，将输出：

```
Sum of 1 to 10 inclusive is 55
```

As before, we begin by including the `iostream` header and define a `main` function. Inside `main` we define two `int` variables: `sum`, which will hold our summation, and `val`, which will represent each of the values from 1 through 10. We give `sum` an initial value of zero and start `val` off with the value one.

与前面一样，程序首先包含 `iostream` 头文件并定义 `main` 函数。在 `main` 函数中定义两个 `int` 型变量：`sum` 保存总和，`val` 表示从 1 到 10 之间的每一个值。我们给 `sum` 赋初值 0，而 `val` 则从 1 开始。

The important part is the `while` statement. A `while` has the form

重要的部分是 `while` 语句。`while` 结构有这样的形式：

```
while (condition) while_body_statement;
```

A `while` executes by (repeatedly) testing the `condition` and executing the associated `while_body_statement` until the `condition` is false.

`while` 通过测试 `condition` (条件) 和执行相关 `while_body_statement` 来重复执行，直到 `condition` 为假。

A [condition](#) is an expression that is evaluated so that its result can be tested. If the resulting value is nonzero, then the condition is true; if the value is zero then the condition is false.

[条件](#) 是一个可求值的表达式，所以可以测试其结果。如果结果值非零，那么条件为真；如果值为零，则条件为假。

If the `condition` is true (the expression evaluates to a value other than zero) then `while_body_statement` is executed. After executing `while_body_statement`, the `condition` is tested again. If `condition` remains true, then the `while_body_statement` is again executed. The `while` continues, alternatively testing the `condition` and executing `while_body_statement` until the `condition` is false.

如果 `condition` 为真 (表达式求值不为零)，则执行 `while_body_statement`。执行完后，再次测试 `condition`。如果 `condition` 仍为真，则再次执行

Section 1.4. Control Structures

`while_body_statement`。`while` 语句一直交替测试 `condition` 和执行 `while_body_statement`，直到 `condition` 为假为止。

In this program, the `while` statement is:

在这个程序中，`while` 语句是

```
// keep executing the while until val is greater than 10
while (val <= 10) {
    sum += val; // assigns sum + val to sum
    ++val; // add 1 to val
}
```

The condition in the `while` uses the [less-than-or-equal operator](#) (the `<=` operator) to compare the current value of `val` and `10`. As long as `val` is less than or equal to `10`, we execute the body of the `while`. In this case, the body of the `while` is a **block** containing two statements:

`while` 语句的条件用了[小于或等于操作符](#) (`<=` 操作符)，将 `val` 的当前值和 `10` 比较，只要 `val` 小于或等于 `10`，就执行 `while` 循环体。这种情况下，`while` 循环体是一个包含两个语句的块：

```
{
    sum += val; // assigns sum + val to sum
    ++val; // add 1 to val
}
```

A block is a sequence of statements enclosed by curly braces. In C++, a block may be used wherever a statement is expected. The first statement in the block uses the [compound assignment operator](#), (the `+=` operator). This operator adds its right-hand operand to its left-hand operand. It has the same effect as writing an addition and an [assignment](#):

块是被花括号括起来的语句序列。C++ 中，块可用于任何可以用一条语句的地方。块中第一条语句使用了[复合赋值操作符](#) (`+=` 操作符)，这个操作符把它的右操作数加至左操作数，这等效于编写含一个加法和一个赋值的语句：

```
sum = sum + val; // assign sum + val to sum
```

Thus, the first statement adds the value of `val` to the current value of `sum` and stores the result back into `sum`.

因此第一条语句是把 `val` 的值加到 `sum` 的当前值，并把结果存入 `sum`。

The next statement

第二条语句

```
++val; // add 1 to val
```

uses the [prefix increment operator](#) (the `++` operator). The increment operator adds one to its operand. Writing `++val` is the same as writing `val = val + 1`.

使用了[前自增操作符](#) (`++` 操作符)，自增操作符就是在它的操作数上加 `1`，`++val` 和 `val = val + 1` 是一样的。

After executing the `while` body we again execute the condition in the `while`. If the (now incremented) value of `val` is still less than or equal to `10`, then the body of the `while` is executed again. The loop continues, testing the condition and executing the body, until `val` is no longer less than or equal to `10`.

执行 `while` 的循环体后，再次执行 `while` 的条件。如果 `val` 的值（自增后）仍小于或等于 `10`，那么再次执行 `while` 的循环体。循环继续，测试条件并执行循环体，直到 `val` 的值不再小于或等于 `10` 为止。

Once `val` is greater than `10`, we fall out of the `while` loop and execute the statement following the `while`. In this case, that statement prints our output, followed by the `return`, which completes our `main` program.

一旦 `val` 的值大于 `10`，程序就跳出 `while` 循环并执行 `while` 后面的语句，此例中该语句打印输出，其后的 `return` 语句结束 `main` 程序。

Key Concept: Indentation and Formatting of C++ Programs

关键概念：C++ 程序的缩排和格式

C++ programs are largely free-format, meaning that the positioning of curly braces, indentation, comments, and newlines usually has no effect on the meaning of our programs. For example, the curly brace that denotes the beginning of the body of `main` could be on the same line as `main`, positioned as we have done, at the beginning of the next line, or placed anywhere we'd like. The only requirement is that it be the first nonblank, noncomment character that the compiler sees after the close parenthesis that concludes `main`'s parameter list.

C++ 程序的格式非常自由，花括号、缩排、注释和换行的位置通常对程序的语义没有影响。例如，表示 `main` 函数体开始的花括号可以放在与 `main` 同一行，也可以像我们那样，放在下一行的开始，或放在你喜欢的任何地方。唯一的要求是，它是编译器所看到在 `main` 的参数列表的右括号之后的第一个非空

格、非注释字符。

Although we are largely free to format programs as we wish, the choices we make affect the readability of our programs. We could, for example, have written `main` on a single, long line. Such a definition, although legal, would be hard to read.

虽然说我们可以很自由地编排程序的格式，但如果编排不当，会影响程序的可读性。例如，我们可以将 `main` 写成单独的一长行。这样的定义尽管合法，但很难阅读。

Endless debates occur as to the right way to format C or C++ programs. Our belief is that there is no single correct style but that there is value in consistency. We tend to put the curly braces that delimit functions on their own lines. We tend to indent compound input or output expressions so that the operators line up, as we did with the statement that wrote the output in the `main` function on page 6. Other indentation conventions will become clear as our programs become more complex.

关于什么是 C 或 C++ 程序的正确格式存在无休止的争论，我们相信没有唯一正确的风格，但一致性是有价值的。我们倾向于把确定函数边界的花括号自成一行，且缩进复合的输入或输出表达式从而使操作符排列整齐，正如第 1.2.2 节的 `main` 函数中的输出语句那样。随着程序的复杂化，其他缩排规范也会变得清晰。

The important thing to keep in mind is that other ways to format programs are possible. When choosing a formatting style, think about how it affects readability and comprehension. Once you've chosen a style, use it consistently.

可能存在其他格式化程序的方式，记住这一点很重要。在选择格式化风格时，要考虑提高程序的可读性，使其更易于理解。一旦选择了某种风格，就要始终如一地使用。

1.4.2. The `for` Statement

1.4.2. `for` 语句

In our `while` loop, we used the variable `val` to control how many times we iterated through the loop. On each pass through the `while`, the value of `val` was tested and then in the body the value of `val` was incremented.

在 `while` 循环中，我们使用变量 `val` 来控制循环执行次数。每次执行 `while` 语句，都要测试 `val` 的值，然后在循环体中增加 `val` 的值。

The use of a variable like `val` to control a loop happens so often that the language defines a second control structure, called a [for statement](#), that abbreviates the code that manages the loop variable. We could rewrite the program to sum the numbers from 1 through 10 using a `for` loop as follows:

由于需要频频使用像 `val` 这样的变量控制循环，因而 C++ 语言定义了第二种控制结构，称为 [for 语句](#)，简化管理循环变量的代码。使用 `for` 循环重新编写求 1 到 10 的和的程序，如下：

```
#include <iostream>
int main()
{
    int sum = 0;
    // sum values from 1 up to 10 inclusive
    for (int val = 1; val <= 10; ++val)
        sum += val; // equivalent to sum = sum + val
    std::cout << "Sum of 1 to 10 inclusive is "
    << sum << std::endl;
    return 0;
}
```

Prior to the `for` loop, we define `sum`, which we set to zero. The variable `val` is used only inside the iteration and is defined as part of the `for` statement itself. The `for` statement

在 `for` 循环之前，我们定义 `sum` 并赋 0 值。用于迭代的变量 `val` 被定义为 `for` 语句自身的一部分。`for` 语句

```
for (int val = 1; val <= 10; ++val)
    sum += val; // equivalent to sum = sum + val
```

has two parts: the `for` header and the `for` body. The header controls how often the body is executed. The header itself consists of three parts: an *init-statement*, a *condition*, and an *expression*. In this case, the *init-statement*

包含 `for` 语句头和 `for` 语句体两部分。`for` 语句头控制 `for` 语句体的执行次数。`for` 语句头由三部分组成：一个初始化语句，一个条件，一个表达式。在这个例子中，初始化语句

```
int val = 1;
```

defines an `int` object named `val` and gives it an initial value of one. The *initstatement* is performed only once, on entry to the `for`. The *condition*

定义一个名为 `val` 的 `int` 对象并给定初始值 1。初始化语句仅在进入 `for` 语句时执行一次。条件

Section 1.4. Control Structures

```
val <= 10
```

which compares the current value in `val` to 10, is tested each time through the loop. As long as `val` is less than or equal to 10, we execute the `for` body. Only after executing the body is the *expression* executed. In this `for`, the expression uses the prefix increment operator, which as we know adds one to the value of `val`. After executing the *expression*, the `for` retests the *condition*. If the new value of `val` is still less than or equal to 10, then the `for` loop body is executed and `val` is incremented again. Execution continues until the *condition* fails.

将 `val` 的当前值和 10 比较，每次经过循环都要测试。只要 `val` 小于或等于 10，就执行 `for` 语句体。仅当 `for` 语句体执行后才执行表达式。在这个 `for` 循环中，表达式使用前自增操作符，`val` 的值加 1，执行完表达式后，`for` 语句重新测试条件，如果 `val` 的新值仍小于或等于 10，则执行 `for` 语句体，`val` 再次自增，继续执行直到条件不成立。

In this loop, the `for` body performs the summation

在这个循环中，`for` 语句体执行求和

```
sum += val; // equivalent to sum = sum + val
```

The body uses the compound assignment operator to add the current value of `val` to `sum`, storing the result back into `sum`.

`for` 语句体使用复合赋值操作符，把 `val` 的当前值加到 `sum`，并将结果保存到 `sum` 中。

To recap, the overall execution flow of this `for` is:

摘要重述一下，`for` 循环总的执行流程为：

1. Create `val` and initialize it to 1.

创建 `val` 并初始化为 1。

2. Test whether `val` is less than or equal to 10.

测试 `val` 是否小于或等于 10。

3. If `val` is less than or equal to 10, execute the `for` body, which adds `val` to `sum`. If `val` is not less than or equal to 10, then break out of the loop and continue execution with the first statement following the `for` body.

如果 `val` 小于或等于 10，则执行 `for` 循环体，把 `val` 加到 `sum` 中。如果 `val` 大于 10，就退出循环，接着执行 `for` 语句体后的第一条语句。

4. Increment `val`.

`val` 递增。

5. Repeat the test in step 2, continuing with the remaining steps as long as the condition is true.

重复第 2 步的测试，只要条件为真，就继续执行其余步骤。



When we exit the `for` loop, the variable `val` is no longer accessible. It is not possible to use `val` after this loop terminates. However, not all compilers enforce this requirement.

退出 `for` 循环后，变量 `val` 不再可访问，循环终止后使用 `val` 是不可能的。然而，不是所有的编译器都有这一要求。

In pre-Standard C++ names defined in a `for` header were accessible outside the `for` itself. This change in the language definition can surprise people accustomed to using an older compiler when they instead use a compiler that adheres to the standard.

在标准化之前的 C++ 中，定义在 `for` 语句头的名字在 `for` 循环外是可访问的。语言定义中的这一改变，可能会使习惯于使用老式编译器的人，在使用遵循标准的新编译器时感到惊讶。

Compilation Revisited

再谈编译

Part of the compiler's job is to look for errors in the program text. A compiler cannot detect whether the meaning of a program is correct, but it can detect errors in the form of the program. The following are the most common kinds of errors a compiler will detect.

编译器的部分工作是寻找程序代码中的错误。编译器不能查出程序的意义是否正确，但它可以查出程序形式上的错误。下面是编译器能查出的最普遍的一些错误。

- 1. Syntax errors.** The programmer has made a grammatical error in the C++ language. The following program illustrates common syntax errors; each comment describes the error on the following line:

语法错误。程序员犯了 C++ 语言中的语法错误。下面代码段说明常见的语法错误；每个注释描述下一行的错误。

```
int main ( { // error: missing ')' in parameter list for main
    // error: used colon, not a semicolon after endl
    std::cout << "Read each file." << std::endl;
    // error: missing quotes around string literal
    std::cout << Update master. << std::endl;
    // ok: no errors on this line
    std::cout << "Write new master." << std::endl;
    // error: missing ';' on return statement
    return 0
}
```

- 2. Type errors.** Each item of data in C++ has an associated type. The value 10, for example, is an integer. The word "hello" surrounded by double quotation marks is a string literal. One example of a type error is passing a string literal to a function that expects an integer argument.

类型错误。C++ 中每个数据项都有其相关联的类型。例如，值 10 是一个整数。用双引号标注起来的单词“hello”是字符串字面值。类型错误的一个实例是传递了字符串字面值给应该得到整型参数的函数。

- 3. Declaration errors.** Every name used in a C++ program must be declared before it is used. Failure to declare a name usually results in an error message. The two most common declaration errors are to forget to use `std::` when accessing a name from the library or to inadvertently misspell the name of an identifier:

声明错误。C++ 程序中使用的每个名字必须在使用之前声明。没有声明名字通常会导致错误信息。最常见的两种声明错误，是从标准库中访问名字时忘记使用“`std::`”，以及由于疏忽而拼错标识符名：

```
#include <iostream>
int main()
{
    int v1, v2;
    std::cin >> v >> v2; // error: uses " v "not" v1"
    // cout not defined, should be std::cout
    cout << v1 + v2 << std::endl;
    return 0;
}
```

An error message contains a line number and a brief description of what the compiler believes we have done wrong. It is a good practice to correct errors in the sequence they are reported. Often a single error can have a cascading effect and cause a compiler to report more errors than actually are present. It is also a good idea to recompile the code after each fix or after making at most a small number of obvious fixes. This cycle is known as [edit-compile-debug](#).

错误信息包含行号和编译器对我们所犯错误的简要描述。按错误报告的顺序改正错误是个好习惯，通常一个错误可能会产生一连串的影响，并导致编译器报告比实际多得多的错误。最好在每次修改后或最多改正了一些显而易见的错误后，就重新编译代码。这个循环就是众所周知的[编辑—编译—调试](#)。

Exercises Section 1.4.2

Exercise

- 1.9:** What does the following `for` loop do? What is the final value of `sum`?

下列循环做什么？`sum` 的最终值是多少？

```
int sum = 0;
for (int i = -100; i <= 100; ++i)
    sum += i;
```

Exercise

- 1.10:** Write a program that uses a `for` loop to sum the numbers from 50 to 100. Now rewrite the program using a `while`.

用 `for` 循环编程，求从 50 到 100 的所有自然数的和。然后用 `while` 循环重写该程序。

Exercise

- 1.11:** Write a program using a `while` loop to print the numbers from 10 down to 0. Now rewrite the program using a `for`.

用 `while` 循环编程，输出 10 到 0 递减的自然数。然后用 `for` 循环重写该程序。

Exercise

- 1.12:** Compare and contrast the loops you wrote in the previous two exercises. Are there advantages or disadvantages to using either form?

对比前面两个习题中所写的循环。两种形式各有何优缺点？

Exercise Compilers vary as to how easy it is to understand their diagnostics. Write programs that contain the common errors discussed in the box on 16. Study the messages the compiler generates so that these messages will be familiar when you encounter them while compiling more complex programs.

编译器不同，理解其诊断内容的难易程度也不同。编写一些程序，包含本小节“再谈编译”部分讨论的那些常见错误。研究编译器产生的信息，这样你在编译更复杂的程序遇到这些信息时就不会陌生。

1.4.3. The `if` Statement

1.4.3. `if` 语句

A logical extension of summing the values between 1 and 10 is to sum the values between two numbers our user supplies. We might use the numbers directly in our `for` loop, using the first input as the lower bound for the range and the second as the upper bound. However, if the user gives us the higher number first, that strategy would fail: Our program would exit the `for` loop immediately. Instead, we should adjust the range so that the larger number is the upper bound and the smaller is the lower. To do so, we need a way to see which number is larger.

求 1 到 10 之间数的和，其逻辑延伸是求用户提供的两个数之间的数的和。可以直接在 `for` 循环中使用这两个数，使用第一个输入值作为下界而第二个输入值作为上界。然而，如果用户首先给定的数较大，这种策略将会失败：程序会立即退出 `for` 循环。因此，我们应该调整范围以便较大的数作上界而较小的数作下界。这样做，我们需要一种方式来判定哪个数更大一些。

Like most languages, C++ provides an [if statement](#) that supports conditional execution. We can use an `if` to write our revised sum program:

像大多数语言一样，C++ 提供支持条件执行的 [if 语句](#)。使用 `if` 语句来编写修订的求和程序如下：

```
#include <iostream>
int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1, v2;
    std::cin >> v1 >> v2; // read input
    // use smaller number as lower bound for summation
    // and larger number as upper bound
    int lower, upper;
    if (v1 <= v2) {
        lower = v1;
        upper = v2;
    } else {
        lower = v2;
        upper = v1;
    }
    int sum = 0;
    // sum values from lower up to and including upper
    for (int val = lower; val <= upper; ++val)
        sum += val; // sum = sum + val

    std::cout << "Sum of " << lower
    << " to " << upper
    << " inclusive is "
    << sum << std::endl;
}
return 0;
}
```

If we compile and execute this program and give it as input the numbers 7 and 3, then the output of our program will be

如果我们编译并执行这个程序给定输入数为 7 和 3，程序的输出结果将为：

```
Sum of 3 to 7 inclusive is 25
```

Most of the code in this program should already be familiar from our earlier examples. The program starts by writing a prompt to the user and defines four `int` variables. It then reads from the standard input into `v1` and `v2`. The only new code is the `if` statement

这个程序中大部分代码我们在之前的举例中已经熟悉了。程序首先向用户输出提示并定义 4 个 `int` 变量，然后从标准输入读入值到 `v1` 和 `v2` 中。仅有 `if` 条件语句是新增加的代码：

```
// use smaller number as lower bound for summation
// and larger number as upper bound
int lower, upper;
if (v1 <= v2) {
    lower = v1;
    upper = v2;
} else {
    lower = v2;
    upper = v1;
}
```

Section 1.4. Control Structures

The effect of this code is to set `upper` and `lower` appropriately. The `if` condition tests whether `v1` is less than or equal to `v2`. If so, we perform the block that immediately follows the condition. This block contains two statements, each of which does an assignment. The first statement assigns `v1` to `lower` and the second assigns `v2` to `upper`.

这段代码的效果是恰当地设置 `upper` 和 `lower`。`if` 的条件测试 `v1` 是否小于或等于 `v2`。如果是，则执行条件后面紧接着的语句块。这个语句块包含两条语句，每条语句都完成一次赋值，第一条语句将 `v1` 赋值给 `lower`，而第二条语句将 `v2` 赋值给 `upper`。

If the condition is false that is, if `v1` is larger than `v2` then we execute the statement following the `else`. Again, this statement is a block consisting of two assignments. We assign `v2` to `lower` and `v1` to `upper`.

如果这个条件为假（也就是说，如果 `v1` 大于 `v2`）那么执行 `else` 后面的语句。这个语句同样是一个由两个赋值语句组成的块，把 `v2` 赋值给 `lower` 而把 `v1` 赋值给 `upper`。

Exercises Section 1.4.3

Exercise

- 1.14:** What happens in the program presented in this section if the input values are equal?

如果输入值相等，本节展示的程序将产生什么问题？

Exercise

- 1.15:** Compile and run the program from this section with two equal values as input. Compare the output to what you predicted in the previous exercise. Explain any discrepancy between what happened and what you predicted.

用两个相等的值作为输入编译并运行本节中的程序。将实际输出与你在上一习题中所做的预测相比较，解释实际结果和你预计的结果间的不相符之处。

Exercise

- 1.16:** Write a program to print the larger of two inputs supplied by the user.

编写程序，输出用户输入的两个数中的较大者。

Exercise

- 1.17:** Write a program to ask the user to enter a series of numbers. Print a message saying how many of the numbers are negative numbers.

编写程序，要求用户输入一组数。输出信息说明其中有多少个负数。

1.4.4. Reading an Unknown Number of Inputs

1.4.4. 读入未知数目的输入

Another change we might make to our summation program on page 12 would be to allow the user to specify a set of numbers to sum. In this case we can't know how many numbers we'll be asked to add. Instead, we want to keep reading numbers until the program reaches the end of the input. When the input is finished, the program writes the total to the standard output:

对第 1.4.1 节的求和程序稍作改变，还可以允许用户指定一组数求和。这种情况下，我们不知道要对多少个数求和，而是要一直读数直到程序输入结束。输入结束时，程序将总和写到标准输出：

```
#include <iostream>
int main()
{
    int sum = 0, value;
    // read till end-of-file, calculating a running total of all values read
    while (std::cin >> value)
        sum += value; // equivalent to sum = sum + value
    std::cout << "Sum is: " << sum << std::endl;
    return 0;
}
```

If we give this program the input

如果我们给出本程序的输入：

3 4 5 6

then our output will be

那么输出是：

Section 1.4. Control Structures

```
Sum is: 18
```

As usual, we begin by including the necessary headers. The first line inside `main` defines two `int` variables, named `sum` and `value`. We'll use `value` to hold each number we read, which we do inside the condition in the `while`:

与平常一样，程序首先包含必要的头文件。`main` 中第一行定义了两个 `int` 变量，命名为 `sum` 和 `value`。在 `while` 条件中，用 `value` 保存读入的每一个数：

```
while (std::cin >> value)
```

What happens here is that to evaluate the condition, the input operation

这里所产生的是，为判断条件，先执行输入操作

```
std::cin >> value
```

is executed, which has the effect of reading the next number from the standard input, storing what was read in `value`. The input operator ([Section 1.2.2](#), p. 8) returns its left operand. The condition tests that result, meaning it tests `std::cin`.

它具有从标准输入读取下一个数并且将读入的值保存在 `value` 中的效果。输入操作符 ([第 1.2.2 节](#)) 返回其左操作数。`while` 条件测试输入操作符的返回结果，意味着测试 `std::cin`。

When we use an `istream` as a condition, the effect is to test the state of the stream. If the stream is valid that is, if it is still possible to read another input then the test succeeds. An `istream` becomes invalid when we hit `end-of-file` or encounter an invalid input, such as reading a value that is not an integer. An `istream` that is in an invalid state will cause the condition to fail.

当我们使用 `istream` 对象作为条件，结果是测试流的状态。如果流是有效的（也就是说，如果读入下一个输入是可能的）那么测试成功。遇到[文件结束符](#)或遇到无效输入时，如读取了一个不是整数的值，则 `istream` 对象是无效的。处于无效状态的 `istream` 对象将导致条件失败。

Until we do encounter end-of-file (or some other input error), the test will succeed and we'll execute the body of the `while`. That body is a single statement that uses the compound assignment operator. This operator adds its right-hand operand into the left hand operand.

在遇到文件结束符（或一些其他输入错误）之前，测试会成功并且执行 `while` 循环体。循环体是一条使用复合赋值操作符的语句，这个操作符将它的右操作数加到左操作数上。

Entering an End-of-file from the Keyboard

从键盘输入文件结束符

Operating systems use different values for end-of-file. On Windows systems we enter an end-of-file by typing a control-z simultaneously type the "ctrl" key and a "z." On UNIX systems, including Mac OS-X machines, it is usually control-d.

操作系统使用不同的值作为文件结束符。**Windows** 系统下我们通过键入 `control-z`——同时键入“`ctrl`”键和“`z`”键，来输入文件结束符。**Unix** 系统中，包括 **Mac OS-X** 机器，通常用 `control-d`。

Once the test fails, the `while` terminates and we fall through and execute the statement following the `while`. That statement prints `sum` followed by `endl`, which prints a newline and flushes the buffer associated with `cout`. Finally, we execute the `return`, which as usual returns zero to indicate success.

一旦测试失败，`while` 终止并退出循环体，执行 `while` 之后的语句。该语句在输出 `sum` 后输出 `endl`，`endl` 输出换行并刷新与 `cout` 相关联的缓冲区。最后，执行 `return`，通常返回零表示程序成功运行完毕。

Exercises Section 1.4.4

Exercise 1.18: Write a program that prompts the user for two numbers and writes each number in the range specified by the two numbers to the standard output.

编写程序，提示用户输入两个数并将这两个数范围内的每个数写到标准输出。

Exercise 1.19: What happens if you give the numbers 1000 and 2000 to the program written for the previous exercise? Revise the program so that it never prints more than 10 numbers per line.

如果上题给定数 1000 和 2000，程序将产生什么结果？修改程序，使每一行输出不超过 10 个数。

Exercise 1.20: Write a program to sum the numbers in a user-specified range, omitting the `if` test that sets the upper and lower bounds. Predict what happens if the input is the numbers 7 and 3, in that order. Now run the program giving it the numbers 7 and 3, and see if the results match your

Section 1.4. Control Structures

expectation. If not, restudy the discussion on the `for` and `while` loop until you understand what happened.

编写程序，求用户指定范围内的数的和，省略设置上界和下界的 `if` 测试。假定输入数是 7 和 3，按照这个顺序，预测程序运行结果。然后按照给定的数是 7 和 3 运行程序，看结果是否与你预测的相符。如果不相符，反复研究关于 `for` 和 `while` 循环的讨论直到弄清楚其中的原因。

1.5. Introducing Classes

1.5. 类的简介

The only remaining feature we need to understand before solving our bookstore problem is how to write a ***data structure*** to represent our transaction data. In C++ we define our own data structure by defining a ***class***. The class mechanism is one of the most important features in C++. In fact, a primary focus of the design of C++ is to make it possible to define ***class types*** that behave as naturally as the built-in types themselves. The library types that we've seen already, such as `istream` and `ostream`, are all defined as classes—that is, they are not strictly speaking part of the language.

解决书店问题之前，还需要弄明白如何编写**数据结构**来表示交易数据。C++ 中我们通过定义类来定义自己的数据结构。**类**机制是 C++ 中最重要的特征之一。事实上，C++ 设计的主要焦点就是使所定义的**类类型**的行为可以像内置类型一样自然。我们前面已看到的像 `istream` 和 `ostream` 这样的库类型，都是定义为类的，也就是说，它们严格说来不是语言的一部分。

Complete understanding of the class mechanism requires mastering a lot of information. Fortunately, it is possible to use a class that someone else has written without knowing how to define a class ourselves. In this section, we'll describe a simple class that we can use in solving our bookstore problem. We'll implement this class in the subsequent chapters as we learn more about types, expressions, statements, and functions—all of which are used in defining classes.

完全理解类机制需要掌握很多内容。所幸我们可以使用他人写的类而无需掌握如何定义自己的类。在这一节，我们将描述一个用于解决书店问题的简单类。当我们学习了更多关于类型、表达式、语句和函数的知识（所有这些在类定义中都将用到）后，将会在后面的章节实现这个类。

To use a class we need to know three things:

使用类时我们需要回答三个问题：

1. What is its name?

类的名字是什么？

2. Where is it defined?

它在哪里定义？

3. What operations does it support?

它支持什么操作？

For our bookstore problem, we'll assume that the class is named `Sales_item` and that it is defined in a header named `Sales_item.h`.

对于书店问题，我们假定类命名为 `Sales_item` 且类定义在命名为 `Sales_item.h` 的头文件中。

1.5.1. The `Sales_item` Class

1.5.1. `Sales_item` 类

The purpose of the `Sales_item` class is to store an ISBN and keep track of the number of copies sold, the revenue, and average sales price for that book. How these data are stored or computed is not our concern. To use a class, we need not know anything about how it is implemented. Instead, what we need to know is what operations the class provides.

`Sales_item` 类的目的是存储 ISBN 并保存该书的销售册数、销售收入和平均售价。我们不关心如何存储或计算这些数据。使用类时我们不需要知道这个类是怎样实现的，相反，我们需要知道的是该类提供什么操作。

As we've seen, when we use library facilities such as IO, we must include the associated headers. Similarly, for our own classes, we must make the definitions associated with the class available to the compiler. We do so in much the same way. Typically, we put the class definition into a file. Any program that wants to use our class must include that file.

正如我们所看到的，使用像 IO 一样的库工具，必须包含相关的头文件。类似地，对于自定义的类，必须使得编译器可以访问和类相关的定义。这几乎可以采用同样的方式。一般来说，我们将类定义放入一个文件中，要使用该类的任何程序都必须包含这个文件。

Conventionally, class types are stored in a file with a name that, like the name of a program source file, has two parts: a file name and a file suffix. Usually the file name is the same as the class defined in the header. The suffix usually is `.h`, but some programmers use `.H`, `.hpp`, or `.hxx`. Compilers usually aren't picky about header file names, but IDEs sometimes are. We'll assume that our class is defined in a file named `Sales_item.h`.

Section 1.5. Introducing Classes

依据惯例，类类型存储在一个文件中，其文件名如同程序的源文件名一样，由文件名和文件后缀两部分组成。通常文件名和定义在头文件中的类名是一样的。通常后缀是 `.h`，但也有一些程序员用 `.H`、`.hpp` 或 `.hxx`。编译器通常并不挑剔头文件名，但 IDE 有时会。假设我们的类定义在名为 `Sale_item.h` 的文件中。

Operations on `Sales_item` Objects

`Sales_item` 对象上的操作

Every class defines a type. The type name is the same as the name of the class. Hence, our `Sales_item` class defines a type named `Sales_item`. As with the built-in types, we can define a variable of a class type. When we write

每个类定义一种类型，类型名与类名相同。因此，我们的 `Sales_item` 类定义了一种命名为 `Sales_item` 的类型。像使用内置类型一样，可以定义类类型的变量。当写下

```
Sales_item item;
```

we are saying that `item` is an object of type `Sales_item`. We often contract the phrase "an object of type `Sales_item`" to "a `Sales_item` object" or even more simply to "a `Sales_item`".

就表示 `item` 是类型 `Sales_item` 的一个对象。通常将“类型 `Sales_item` 的一个对象”简称为“一个 `Sales_item` 对象”，或者更简单地简称为“一个 `Sales_item`”。

In addition to being able to define variables of type `Sales_item`, we can perform the following operations on `Sales_item` objects:

除了可以定义 `Sales_item` 类型的变量，我们还可以执行 `Sales_item` 对象的以下操作：

- Use the addition operator, `+`, to add two `Sales_items`
使用加法操作符，`+`，将两个 `Sales_item` 相加。
- Use the input operator, `<<` to read a `Sales_item` object,
使用输入操作符，`<<`，来读取一个 `Sales_item` 对象。
- Use the output operator, `>>` to write a `Sales_item` object
使用输出操作符，`>>`，来输出一个 `Sales_item` 对象。
- Use the assignment operator, `=`, to assign one `Sales_item` object to another
使用赋值操作符，`=`，将一个 `Sales_item` 对象赋值给另一个 `Sales_item` 对象。
- Call the `same_isbn` function to determine if two `Sales_items` refer to the same book
调用 `same_isbn` 函数确定两个 `Sales_item` 是否指同一本书。

Reading and Writing `Sales_item`s

读入和写出 `Sales_item` 对象

Now that we know the operations that the class provides, we can write some simple programs to use this class. For example, the following program reads data from the standard input, uses that data to build a `Sales_item` object, and writes that `Sales_item` object back onto the standard output:

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item book;
    // read ISBN, number of copies sold, and sales price
    std::cin >> book;
    // write ISBN, number of copies sold, total revenue, and average price
    std::cout << book << std::endl;
    return 0;
}
```

If the input to this program is

如果输入到程序的是

0-201-70353-X 4 24.99

then the output will be

则输出将是

0-201-70353-X 4 99.96 24.99

Our input said that we sold four copies of the book at \$24.99 each, and the output indicates that the total sold was four, the total revenue was \$99.96, and the average price per book was \$24.99.

输入表明销售了 4 本书，每本价格是 24.99 美元。输出表明卖出书的总数是 4 本，总收入是 99.96 美元，每本书的平均价格是 24.99 美元。

This program starts with two `#include` directives, one of which uses a new form. The `iostream` header is defined by the standard library; the `Sales_item` header is not. `Sales_item` is a type that we ourselves have defined. When we use our own headers, we use quotation marks (" ") to surround the header name.

这个程序以两个 `#include` 指示开始，其中之一使用了一种新格式。`iostream` 头文件由标准库定义，而 `Sales_item` 头文件则不是。`Sales_item` 是一种自定义类型。当使用自定义头文件时，我们采用双引号 (" ") 把头文件名括起来。



Headers for the standard library are enclosed in angle brackets (< >). Nonstandard headers are enclosed in double quotes (" ").

标准库的头文件用尖括号 < > 括起来，非标准库的头文件用双引号 " " 括起来。

Inside `main` we start by defining an object, named `book`, which we'll use to hold the data that we read from the standard input. The next statement reads into that object, and the third statement prints it to the standard output followed as usual by printing `endl` to flush the buffer.

在 `main` 函数中，首先定义一个对象，命名为 `book`，用它保存从标准输入读取的数据。下一条语句读入数据到此对象，第三条语句将它打印到标准输出，像平常一样紧接着打印 `endl` 来刷新缓冲区。

Key Concept: Classes Define Behavior

关键概念：类定义行为

As we go through these programs that use `Sales_item`s, the important thing to keep in mind is that the author of the `Sales_item` class defined all the actions that can be performed by objects of this class. That is, the author of the `Sales_item` data structure defines what happens when a `Sales_item` object is created and what happens when the addition or the input and output operators are applied to `Sales_item` objects, and so on.

在编写使用 `Sales_item` 的程序时，重要的是记住类 `Sales_item` 的创建者定义该类对象可以执行的所有操作。也就是说，`Sales_item` 数据结构的创建者定义创建 `Sales_item` 对象时会发生什么，以及加操作符或输入输出操作符应用到 `Sales_item` 对象时又会发生什么，等等。

In general, only the operations defined by a class can be used on objects of the class type. For now, the only operations we know we can perform on `Sales_item` objects are the ones listed on page 21.

通常，只有由类定义的操作可被用于该类类型的对象。此时，我们知道的可以在 `Sales_item` 对象上执行的操作只是前面列出的那些。

We'll see how these operations are defined in [Sections 7.7.3 and 14.2](#).

我们将在[第 7.7.3 节](#)和[第 14.2 节](#)看到如何定义这些操作。

Adding `Sales_item`s

将 `Sales_item` 对象相加

A slightly more interesting example adds two `Sales_item` objects:

更有趣的例子是将两个 `Sales_item` 对象相加：

```
#include <iostream>
#include "Sales_item.h"
int main()
{
```

Section 1.5. Introducing Classes

```
Sales_item item1, item2;
std::cin >> item1 >> item2; // read a pair of transactions
std::cout << item1 + item2 << std::endl; // print their sum
return 0;
}
```

If we give this program the following input

如果我们给这个程序下面的输入：

```
0-201-78345-x 3 20.00
0-201-78345-x 2 25.00
```

our output is

则输出为

```
0-201-78345-x 5 110.00
```

This program starts by including the `Sales_item` and `iostream` headers. Next we define two `Sales_item` objects to hold the two transactions that we wish to sum. The output expression does the addition and prints the result. We know from the list of operations on page 21 that adding two `Sales_items` together creates a new object whose ISBN is that of its operands and whose number sold and revenue reflect the sum of the corresponding values in its operands. We also know that the items we add must represent the same ISBN.

程序首先包含两个头文件 `Sales_item` 和 `iostream`。接下来定义两个 `Sales_item` 对象来存放要求和的两笔交易。输出表达式做加法运算并输出结果。从前面列出的操作，可以得知将两个 `Sales_item` 相加将创建一个新对象，新对象的 ISBN 是其操作数的 ISBN，销售的数量和收入反映其操作数中相应值的和。我们也知道相加的项必须具有同样的 ISBN。

It's worth noting how similar this program looks to the one on page 6: We read two inputs and write their sum. What makes it interesting is that instead of reading and printing the sum of two integers, we're reading and printing the sum of two `Sales_item` objects. Moreover, the whole idea of "sum" is different. In the case of `ints` we are generating a conventional sum—the result of adding two numeric values. In the case of `Sales_item` objects we use a conceptually new meaning for sum—the result of adding the components of two `Sales_item` objects.

值得注意的是这个程序是如何类似于第 1.2.2 节中的程序：读入两个输入并输出它们的和。令人感兴趣的是，本例并不是读入两个整数并输出两个整数的和，而是读入两个 `Sales_item` 对象并输出两个 `Sales_item` 对象的和。此外，“和”的意义也不同。在整数的实例中我们产生的是传统求和——两个数值相加后的结果。在 `Sales_item` 对象的实例上我们使用了在概念上有新意义的求和——两个 `Sales_item` 对象的成分相加后的结果。

Exercises Section 1.5.1

Exercise

- 1.21:** The Web site (http://www.awprofessional.com/cpp_primer) contains a copy of `Sales_item.h` in the `Chapter_1` code directory. Copy that file to your working directory. Write a program that loops through a set of book sales transactions, reading each transaction and writing that transaction to the standard output.

本书配套网站的第一章的代码目录下有 `Sales_item.h` 源文件。复制该文件到你的工作目录。编写程序，循环遍历一组书的销售交易，读入每笔交易并将交易写至标准输出。

Exercise

- 1.22:** Write a program that reads two `Sales_item` objects that have the same ISBN and produces their sum.

编写程序，读入两个具有相同 ISBN 的 `Sales_item` 对象并产生它们的和。

Exercise

- 1.23:** Write a program that reads several transactions for the same ISBN. Write the sum of all the transactions that were read.

编写程序，读入几个具有相同 ISBN 的交易，输出所有读入交易的和。

1.5.2. A First Look at Member Functions

1.5.2. 初窥成员函数

Unfortunately, there is a problem with the program that adds `Sales_items`. What should happen if the input referred to two different ISBNs? It doesn't make sense to add the data for two different ISBNs together. To solve this problem, we'll first check whether the `Sales_item` operands refer to the same ISBNs:

不幸的是，将 `Sales_item` 相加的程序有一个问题。如果输入指向了两个不同的 ISBN 将发生什么？将两个不同 ISBN 的数据相加没有意义。为解决这个问题，首先检查 `Sales_item` 操作数是否都具有相同的 ISBN。

Section 1.5. Introducing Classes

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item item1, item2;
    std::cin >> item1 >> item2;
    // first check that item1 and item2 represent the same book
    if (item1.same_isbn(item2)) {
        std::cout << item1 + item2 << std::endl;
        return 0; // indicate success
    } else {
        std::cerr << "Data must refer to same ISBN"
            << std::endl;
        return -1; // indicate failure
    }
}
```

The difference between this program and the previous one is the `if` test and its associated `else` branch. Before explaining the `if` condition, we know that what this program does depends on the condition in the `if`. If the test succeeds, then we write the same output as the previous program and return `0` indicating success. If the test fails, we execute the block following the `else`, which prints a message and returns an error indicator.

这个程序和前一个程序不同之处在于 `if` 测试语句以及与它相关联的 `else` 分支。在解释 `if` 语句的条件之前，我们明白程序的行为取决于 `if` 语句中的条件。如果测试成功，那么产生与前一程序相同的输出，并返回 `0` 表示程序成功运行完毕。如果测试失败，执行 `else` 后面的语句块，输出信息并返回错误提示。

What Is a Member Function?

什么是成员函数

The `if` condition

上述 `if` 语句的条件

```
// first check that item1 and item2 represent the same book
if (item1.same_isbn(item2)) {
```

calls a member function of the `Sales_item` object named `item1`. A member function is a function that is defined by a class. Member functions are sometimes referred to as the methods of the class.

调用命名为 `item1` 的 `Sales_item` 对象的成员函数。成员函数是由类定义的函数，有时称为类方法。

Member functions are defined once for the class but are treated as members of each object. We refer to these operations as member functions because they (usually) operate on a specific object. In this sense, they are members of the object, even though a single definition is shared by all objects of the same type.

成员函数只定义一次，但被视为每个对象的成员。我们将这些操作称为成员函数，是因为它们（通常）在特定对象上操作。在这个意义上，它们是对象的成员，即使同一类型的所有对象共享同一个定义也是如此。

When we call a member function, we (usually) specify the object on which the function will operate. This syntax uses the dot operator (the `.` operator):

当调用成员函数时，（通常）指定函数要操作的对象。语法是使用点操作符 `(.)`：

```
item1.same_isbn
```

means "the `same_isbn` member of the object named `item1`." The dot operator fetches its right-hand operand from its left. The dot operator applies only to objects of class type: The left-hand operand must be an object of class type; the right-hand operand must name a member of that type.

意思是“命名为 `item1` 的对象的 `same_isbn` 成员”。点操作符通过它的左操作数取得右操作数。点操作符仅应用于类类型的对象：左操作数必须是类类型的对象，右操作数必须指定该类型的成员。



Unlike most other operators, the right operand of the dot `(".")` operator is not an object or value; it is the name of a member.

与大多数其他操作符不同，点操作符 `(".")` 的右操作数不是对象或值，而是成员的名字。

When we use a member function as the right-hand operand of the dot operator, we usually do so to call that function. We execute a member

Section 1.5. Introducing Classes

function in much the same way as we do any function: To call a function, we follow the function name by the **call operator** (the "`()`" operator). The call operator is a pair of parentheses that encloses a (possibly empty) list of **arguments** that we pass to the function.

通常使用成员函数作为点操作符的右操作数来调用成员函数。执行成员函数和执行其他函数相似：要调用函数，可将调用操作符 (`()`) 放在函数名之后。调用操作符是一对圆括号，括住传递给函数的实参列表（可能为空）。

The `same_isbn` function takes a single argument, and that argument is another `Sales_item` object. The call

`same_isbn` 函数接受单个参数，且该参数是另一个 `Sales_item` 对象。函数调用

```
item1.same_isbn(item2)
```

passes `item2` as an argument to the function named `same_isbn` that is a member of the object named `item1`. This function compares the ISBN part of its argument, `item2`, to the ISBN in `item1`, the object on which `same_isbn` is called. Thus, the effect is to test whether the two objects refer to the same ISBN.

将 `item2` 作为参数传递给名为 `same_isbn` 的函数，该函数是名为 `item1` 的对象的成员。它将比较参数 `item2` 的 ISBN 与函数 `same_isbn` 要操作的对象 `item1` 的 ISBN。效果是测试两个对象是否具有相同的 ISBN。

If the objects refer to the same ISBN, we execute the statement following the `if`, which prints the result of adding the two `Sales_item` objects together. Otherwise, if they refer to different ISBNs, we execute the `else` branch, which is a block of statements. The block prints an appropriate error message and exits the program, returning `-1`. Recall that the return from `main` is treated as a status indicator. In this case, we return a nonzero value to indicate that the program failed to produce the expected result.

如果对象具有相同的 ISBN，执行 `if` 后面的语句，输出两个 `Sales_item` 对象的和；否则，如果对象具有不同的 ISBN，则执行 `else` 分支的语句块。该块输出适当的错误信息并退出程序，返回 `-1`。回想 `main` 函数的返回值被视为状态指示器；本例中，返回一个非零值表示程序未能产生期望的结果。

Exercises Section 1.5.2

Exercise

1.24:

Write a program that reads several transactions. For each new transaction that you read, determine if it is the same ISBN as the previous transaction, keeping a count of how many transactions there are for each ISBN. Test the program by giving multiple transactions. These transactions should represent multiple ISBNs but the records for each ISBN should be grouped together.

编写程序，读入几笔不同的交易。对于每笔新读入的交易，要确定它的 ISBN 是否和以前的交易的 ISBN 一样，并且记下每一个 ISBN 的交易的总数。通过给定多笔不同的交易来测试程序。这些交易必须代表多个不同的 ISBN，但是每个 ISBN 的记录应分在同一组。

1.6. The C++ Program

1.6. C++ 程序

Now we are ready to solve our original bookstore problem: We need to read a file of sales transactions and produce a report that shows for each book the total revenue, average sales price, and the number of copies sold.

现在我们已经做好准备，可以着手解决最初的书店问题了：我们需要读入销售交易文件，并产生报告显示每本书的总销售收入、平均销售价格和销售册数。

We'll assume that all of the transactions for a given ISBN appear together. Our program will combine the data for each ISBN in a `Sales_item` object named `total`. Each transaction we read from the standard input will be stored in a second `Sales_item` object named `trans`. Each time we read a new transaction we'll compare it to the `Sales_item` object in `total`. If the objects refer to the same ISBN, we'll update `total`. Otherwise we'll print the value in `total` and reset it using the transaction we just read.

假定给定ISBN的所有交易出现在一起。程序将把每个 ISBN 的数据组合至命名为 `total` 的 `Sales_item` 对象中。从标准输入中读取的每一笔交易将被存储到命名为 `trans` 的第二个 `Sales_item` 对象中。每读取一笔新的交易，就将它与 `total` 中的 `Sales_item` 对象相比较，如果对象含有相同的 ISBN，就更新 `total`；否则就输出 `total` 的值，并使用刚读入的交易重置 `total`。

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    // declare variables to hold running sum and data for the next record
    Sales_item total, trans;
    // is there data to process?
    if (std::cin >> total) {
        // if so, read the transaction records
        while (std::cin >> trans)
            if (total.same_isbn(trans))
                // match: update the running total
                total = total + trans;
            else {
                // no match: print & assign to total
                std::cout << total << std::endl;
                total = trans;
            }
        // remember to print last record
        std::cout << total << std::endl;
    } else {
        // no input!, warn the user
        std::cout << "No data?!" << std::endl;
        return -1; // indicate failure
    }
    return 0;
}
```

This program is the most complicated one we've seen so far, but it uses only facilities that we have already encountered. As usual, we begin by including the headers that we use: `iostream` from the library and `Sales_item.h`, which is our own header.

这个程序是到目前我们见到的程序中最为复杂的一个，但它仅使用了我们已遇到过的工具。和平常一样，我们从包含所使用的头文件开始：标准库中的 `iostream` 和自定义的头文件 `Sales_item.h`。

Inside `main` we define the objects we need: `total`, which we'll use to sum the data for a given ISBN, and `trans`, which will hold our transactions as we read them. We start by reading a transaction into `total` and testing whether the read was successful. If the read fails, then there are no records and we fall through to the outermost `else` branch, which prints a message to warn the user that there was no input.

在 `main` 中我们定义了所需要的对象 `total` 用来计算给定的 ISBN 的交易的总数，`trans` 用来存储读取的交易。我们首先将交易读入 `total` 并测试是否读取成功；如果读取失败，表示没有记录，程序进入最外层的 `else` 分支，输出信息警告用户没有输入。

Assuming we have successfully read a record, we execute the code in the `if` branch. The first statement is a `while` that will loop through all the remaining records. Just as we did in the program on page 18, our `while` condition reads a value from the standard input and then tests that valid data was actually read. In this case, we read a `Sales_item` object into `trans`. As long as the read succeeds, we execute the body of the `while`.

假如我们成功读取了一个记录，则执行 `if` 分支里的代码。首先执行 `while` 语句，循环遍历剩余的所有记录。就像第 1.4.3 节的程序一样，`while` 循环的条件从标准输入中读取值并测试实际读取的是是否是合法数据。本例中，我们将一个 `Sales_item` 对象读至 `trans`。只要读取成功，就执行 `while` 循环体。

The body of the `while` is a single `if` statement. We test whether the ISBNs are equal, and if so we add the two objects and store the result in `total`. If the ISBNs are not equal, we print the value stored in `total` and reset `total` by assigning `trans` to it. After execution of the `if`, we return to the condition in the `while`, reading the next transaction and so on until we run out of records.

`while` 循环体只是一条 `if` 语句。我们测试 ISBN 是否相等。如果相等，我们将这两个对象相加并将结果存储到 `total` 中。否则，我们就输出存储在 `total` 中的值，并将 `trans` 赋值给 `total` 来重置 `total`。执行完 `if` 语句之后，将返回到 `while` 语句中的条件，读入下一个交易，直到执行完所有记录。

Once the `while` completes, we still must write the data associated with the last ISBN. When the `while` terminates, `total` contains the data for the

Section 1.6. The C++ Program

last ISBN in the file, but we had no chance to print it. We do so in the last statement of the block that concludes the outermost `if` statement.

一旦 `while` 完成，我们仍须写出与最后一个 ISBN 相关联的数据。当 `while` 语句结束时，`total` 包含文件中最后一条 ISBN 数据，但是我们没有机会输出这条数据。我们在结束最外层 `if` 语句的语句块的最后一条语句中进行输出。

Exercises Section 1.6

Exercise 1.25: Using the `Sales_item.h` header from the Web site, compile and execute the bookstore program presented in this section.

使用源自本书配套网站的 `Sales_item.h` 头文件，编译并执行本节给出的书店程序。

Exercise 1.26: In the bookstore program we used the addition operator and not the compound assignment operator to add `trans` to `total`. Why didn't we use the compound assignment operator?

在书店程序中，我们使用了加法操作符而不是复合赋值操作符将 `trans` 加到 `total` 中，为什么我们不使用复合赋值操作符？

Team LiB

◀ PREVIOUS NEXT ▶

Chapter Summary

小结

This chapter introduced enough of C++ to let the reader compile and execute simple C++ programs. We saw how to define a `main` function, which is the function that is executed first in any C++ program. We also saw how to define variables, how to do input and output, and how to write `if`, `for`, and `while` statements. The chapter closed by introducing the most fundamental facility in C++: the class. In this chapter we saw how to create and use objects of a given `class`. Later chapters show how to define our own classes.

本章介绍了足够多的 C++ 知识，让读者能够编译和执行简单 C++ 程序。我们看到了如何定义 `main` 函数，这是任何 C++ 程序首先执行的函数。我们也看到了如何定义变量，如何进行输入和输出，以及如何编写 `if`、`for` 和 `while` 语句。本章最后介绍 C++ 最基本的工具：类。在这一章中，我们看到了如何创建和使用给定类的对象。后面的章节中将介绍如何自定义类。

Defined Terms

术语

argument (实参)

A value passed to a function when it is called.

传递给被调用函数的值。

block (块)

Sequence of statements enclosed in curly braces.

花括号括起来的语句序列。

buffer (缓冲区)

A region of storage used to hold data. IO facilities often store input (or output) in a buffer and read or write the buffer independently of actions in the program. Output buffers usually must be explicitly flushed to force the buffer to be written. By default, reading `cin` flushes `cout`; `cout` is also flushed when the program ends normally.

一段用来存放数据的存储区域。IO 设备常存储输入（或输出）到缓冲区，并独立于程序动作对缓冲区进行读写。输出缓冲区通常必须显式刷新以强制输出缓冲区内容。默认情况下，读 `cin` 会刷新 `cout`；当程序正常结束时，`cout` 也被刷新。

built-in type (内置类型)

A type, such as `int`, defined by the language.

C++ 语言本身定义的类型，如 `int`。

cerr

`ostream` object tied to the standard error, which is often the same stream as the standard output. By default, writes to `cerr` are not buffered. Usually used for error messages or other output that is not part of the normal logic of the program.

绑定到标准错误的 `ostream` 对象，这通常是与标准输出相同的流。默认情况下，输出 `cerr` 不缓冲，通常用于不是程序正常逻辑部分的错误信息或其他输出。

cin

`istream` object used to read from the standard input.

用于从标准输入中读入的 `istream` 对象。

class

C++ mechanism for defining our own data structures. The class is one of the most fundamental features in C++. Library types, such as `istream` and `ostream`, are classes.

用于自定义数据结构的 C++ 机制。类是 C++ 中最基本的特征。标准库类型，如 `istream` 和 `ostream`，都是类。

class type

A type defined by a class. The name of the type is the class name.

由类所定义的类型，类型名就是类名。

clog

`ostream` object tied to the standard error. By default, writes to `clog` are buffered. Usually used to report information about program execution to a log file.

绑定到标准错误的 `ostream` 对象。默认情况下，写到 `clog` 时是带缓冲的。通常用于将程序执行信息写入到日志文件中。

comments (注释)

Program text that is ignored by the compiler. C++ has two kinds of comments: single-line and paired. Single-line comments start with a `//`. Everything from the `//` to the end of the line is a comment. Paired comments begin with a `/*` and include all text up to the next `*/`.

编译器会忽略的程序文本。C++ 有单行注释和成对注释两种类型的注释。单行注释以 `//` 开头，从 `//` 到行的结尾是一条注释。成对注释以 `/*` 开始包括到下一个 `*/` 为止的所有文本。

condition (条件)

An expression that is evaluated as true or false. An arithmetic expression that evaluates to zero is false; any other value yields true.

求值为真或假的表达式。值为 0 的算术表达式是假，其他所有非 0 值都是真。

cout

`ostream` object used to write to the standard output. Ordinarily used to write the output of a program.

用于写入到标准输出的 `ostream` 对象，一般情况下用于程序的输出。

curly brace (花括号)

Curly braces delimit blocks. An open curly `{}` starts a block; a close curly `}` ends one.

花括号对语句块定界。左花括号`{`开始一个块，右花括号`}`结束块。

data structure (数据结构)

A logical grouping of data and operations on that data.

数据及数据上操作的逻辑组合。

edit-compile-debug (编辑—编译—调试)

The process of getting a program to execute properly.

使得程序正确执行的过程。

end-of-file (文件结束符)

System-specific marker in a file that indicates that there is no more input in the file.

文件中与特定系统有关的标记，表示这个文件中不再有其他输入。

expression (表达式)

an

The smallest unit of computation. An expression consists of one or more operands and usually an operator. Expressions are evaluated to produce a result. For example, assuming `i` and `j` are `ints`, then `i + j` is an arithmetic addition expression that yields the sum of the two `int` values. Expressions are covered in more detail in [Chapter 5](#).

最小的计算单元。表达式包含一个或多个操作数并经常含有一个操作符。表达式被求值并产生一个结果。例如，假定 `i` 和 `j` 都为 `int` 型，则 `i + j` 是一个算术加法表达式并求这两个 `int` 值的和。表达式将在[第五章](#)详细介绍。

for statement (for 语句)

Control statement that provides iterative execution. Often used to step through a data structure or to repeat a calculation a fixed number of times.

提供迭代执行的控制语句，通常用于步进遍历数据结构或对一个计算重复固定次数。

function (函数)

A named unit of computation.

有名字的计算单元。

function body (函数体)

Statement block that defines the actions performed by a function.

定义函数所执行的动作的语句块。

function name (函数名)

Name by which a function is known and can be called.

函数的名字标识，函数通过函数名调用。

header (头文件)

A mechanism whereby the definitions of a class or other names may be made available to multiple programs. A header is included in a program through a `#include` directive.

使得类或其他名字的定义在多个程序中可用的一种机制。程序中通过 `#include` 指示包含头文件。

if statement (if 语句)

Conditional execution based on the value of a specified condition. If the condition is true, the `if` body is executed. If not, control flows to the statement following the `else` if there is one or to the statement following the `if` if there is no `else`.

根据指定条件的值执行的语句。如果条件为真，则执行 `if` 语句体；否则控制流执行 `else` 后面的语句，如果没有 `else` 将执行 `if` 后面的语句。

iostream (输入输出流)

library type providing stream-oriented input and output.

提供面向流的输入和输出的标准库类型。

istream (输入流)

Library type providing stream-oriented input.

提供面向流的输入的标准库类型。

library type (标准库类型)

A type, such as `istream`, defined by the standard library.

标准库所定义的类型，如 `istream`。

main function (主函数)

Function called by the operating system when executing a C++ program. Each program must have one and only one function named `main`.

执行 C++ 程序时，操作系统调用的函数。每一个程序有且仅有一个主函数 `main`。

manipulator (操纵符)

Object, such as `std::endl`, that when read or written "manipulates" the stream itself. [Section A.3.1](#) (p. 825) covers manipulators in more detail.

在读或写时“操纵”流本身的对象，如 `std::endl`。[A.3.1 节](#)详细讲述操纵符。

member function (成员函数)

Operation defined by a class. Member functions ordinarily are called to operate on a specific object.

类定义的操作。成员函数通常在特定的对象上进行操作。

method (方法)

Synonym for member function.

成员函数的同义词。

namespace (命名空间)

Mechanism for putting names defined by a library into a single place. Namespaces help avoid inadvertent name clashes. The names defined by the C++ library are in the namespace `std`.

将库所定义的名字放至单独一个地方的机制。命名空间有助于避免无意的命名冲突。C++ 标准库所定义的名字在命名空间 `std` 中。

ostream (输出流)

Keyterm Defined Terms

Library type providing stream-oriented output.

提供面向流的输出的库类型。

parameter list (形参表)

Part of the definition of a function. Possibly empty list that specifies what arguments can be used to call the function.

函数定义的组成部分。指明可以用什么参数来调用函数，可能为空。

preprocessor directive (预处理器指示)

An instruction to the C++ preprocessor. `#include` is a preprocessor directive. Preprocessor directives must appear on a single line. We'll learn more about the preprocessor in [Section 2.9.2](#).

C++ 预处理器的指示。`#include` 是一个预处理器指示。预处理器指示必须出现在单独的行中。[第 2.9.2 节](#)将对预处理器作详细的介绍。

return type (返回类型)

Type of the value returned by a function.

函数返回值的类型。

source file (源文件)

Term used to describe a file that contains a C++ program.

用来描述包含在 C++ 程序中的文件的术语。

standard error (标准错误)

An output stream intended for use for error reporting. Ordinarily, on a windowing operating system, the standard output and the standard error are tied to the window in which the program is executed.

用于错误报告的输出流。通常，在视窗操作系统中，将标准输出和标准错误绑定到程序的执行窗口。

standard input (标准输入)

The input stream that ordinarily is associated by the operating system with the window in which the program executes.

和程序执行窗口相关联的输入流，通常这种关联由操作系统设定。

standard library (标准库)

Collection of types and functions that every C++ compiler must support. The library provides a rich set of capabilities including the types that support IO. C++ programmers tend to talk about "the library," meaning the entire standard library or about particular parts of the library by referring to a library type. For example, programmers also refer to the "`iostream` library," meaning the part of the standard library defined by the `iostream` classes.

每个 C++ 编译器必须支持的类型和函数的集合。标准库提供了强大的功能，包括支持 IO 的类型。C++ 程序员谈到的“标准库”，是指整个标准库，当提到某个标准库类型时也指标准库中某个特定的部分。例如，程序员提到的“`iostream` 库”，专指标准库中由 `iostream` 类定义的那部分。

standard output (标准输出)

The output stream that ordinarily is associated by the operating system with the window in which the program executes.

和程序执行窗口相关联的输出流，通常这种关联由操作系统设定。

statement (语句)

The smallest independent unit in a C++ program. It is analogous to a sentence in a natural language. Statements in C++ generally end in semicolons.

C++ 程序中最小的独立单元，类似于自然语言中的句子。C++ 中的语句一般以分号结束。

std

Name of the namespace used by the standard library. `std::cout` indicates that we're using the name `cout` defined in the `std` namespace.

标准库命名空间的名字，`std::cout` 表明正在使用定义在 `std` 命名空间中的名字 `cout`。

string literal (字符串字面值)

Sequence of characters enclosed in double quotes.

以双引号括起来的字符序列。

uninitialized variable (未初始化变量)

Variable that has no initial value specified. There are no uninitialized variables of class type. Variables of class type for which no initial value is specified are initialized as specified by the class definition. You must give a value to an uninitialized variable before attempting to use the variable's value. *Uninitialized variables can be a rich source of bugs.*

没有指定初始值的变量。类类型没有未初始化变量。没有指定初始值的类类型变量由类定义初始化。在使用变量值之前必须给未初始化的变量赋值。未初始化变量是造成bug的主要原因之一。

variable (变量)

A named object.

有名字的对象。

while statement (while语句)

An iterative control statement that executes the statement that is the `while` body as long as a specified condition is true. The body is executed zero or more times, depending on the truth value of the condition.

一种迭代控制语句，只要指定的条件为真就执行 `while` 循环体。`while` 循环体执行0次还是多次，依赖于条件的真值。

() operator [(操作符)]

The call operator: A pair of parentheses "()" following a function name. The operator causes a function to be invoked. Arguments to the function may be passed inside the parentheses.

调用操作符。跟在函数名后且成对出现的圆括号。该操作符导致函数被调用，给函数的实参可在括号里传递。

++ operator (++操作符)

Increment operator. Adds one to the operand; `++i` is equivalent to `i = i + 1`.

自增操作符。将操作数加 1，`++i` 等价于 `i = i + 1`。

+= operator (+=操作符)

A compound assignment operator. Adds right-hand operand to the left and stores the result back into the left-hand operand; `a += b` is equivalent to `a = a + b`.

复合赋值操作符，将右操作数和左操作数相加，并将结果存储到左操作数中；`a += b` 等价于 `a = a + b`。

. operator (. 操作符)

Dot operator. Takes two operands: the left-hand operand is an object and the right is the name of a member of that object. The operator fetches that member from the named object.

点操作符。接受两个操作数：左操作数是一个对象，而右边是该对象的一个成员的名字。这个操作符从指定对象中取得成员。

:: operator (:: 操作符)

Scope operator. We'll see more about scope in [Chapter 2](#). Among other uses, the scope operator is used to access names in a namespace. For example, `std::cout` says to use the name `cout` from the namespace `std`.

作用域操作符。在[第二章](#)中，我们将看到更多关于作用域的介绍。在其他的使用过程中，`::` 操作符用于在命名空间中访问名字。例如，`std::cout` 表示使用命名空间 `std` 中的名字 `cout`。

= operator (= 操作符)

Assigns the value of the right-hand operand to the object denoted by the left-hand operand.

表示把右操作数的值赋给左操作数表示的对象。

<< operator (<< 操作符)

Output operator. Writes the right-hand operand to the output stream indicated by the left-hand operand: `cout << "hi"` writes `hi` to the standard output. Output operations can be chained together: `cout << "hi" << "bye"` writes `hibye`.

输出操作符。把右操作数写到左操作数指定的输出流: `cout << "hi"` 把 `hi` 写入到标准输出流。输出操作可以链接在一起使用: `cout << "hi << "bye"` 输出 `hi` 和 `bye`。

>> operator (>> 操作符)

Input operator. Reads from the input stream specified by the left-hand operand into the right-hand operand: `cin >> i` reads the next value on the standard input into `i`. Input operations can be chained together: `cin >> i >> j` reads first into `i` and then into `j`.

输入操作符。从左操作数指定的输入流读入数据到右操作数: `cin >> i` 把标准输入流中的下一个值读入到 `i` 中。输入操作能够链接在一起使用: `cin >> i >> j` 先读入 `i` 然后再读入 `j`。

== operator (== 操作符)

The equality operator. Tests whether the left-hand operand is equal to the right-hand.

等于操作符，测试左右两边的操作数是否相等。

!= operator (!= 操作符)

Assignment operator. Tests whether the left-hand operand is not equal to the right-hand.

不等于操作符。测试左右两边的操作数是否不等。

<= operator (<= 操作符)

The less-than-or-equal operator. Tests whether the left-hand operand is less than or equal to the right-hand.

小于或等于操作符。测试左操作数是否小于或等于右操作数。

< operator (< 操作符)

The less-than operator. Tests whether the left-hand operand is less than the right-hand.

小于操作符。测试左操作数是否小于右操作数。

>= operator (>= 操作符)

Greater-than-or-equal operator. Tests whether the left-hand operand is greater than or equal to the right-hand.

大于或等于操作符。测试左操作数是否大于或等于右操作数。

> operator (> 操作符)

Greater-than operator. Tests whether the left-hand operand is greater than the right-hand.

大于操作符。测试左操作数是否大于右操作数。

Chapter 2. Variables and Basic Types

第二章 变量和基本类型

CONTENTS

Section 2.1 Primitive Built-in Types	34
Section 2.2 Literal Constants	37
Section 2.3 Variables	43
Section 2.4 <code>const</code> Qualifier	56
Section 2.5 References	58
Section 2.6 Typedef Names	61
Section 2.7 Enumerations	62
Section 2.8 Class Types	63
Section 2.9 Writing Our Own Header Files	67
Chapter Summary	73
Defined Terms	73

Types are fundamental to any program. They tell us what our data mean and what operations we can perform on our data.

类型是所有程序的基础。类型告诉我们数据代表什么意思以及可以对数据执行哪些操作。

C++ defines several primitive types: characters, integers, floating-point numbers, and so on. The language also provides mechanisms that let us define our own data types. The library uses these mechanisms to define more complex types such as variable-length character `strings`, `vectors`, and so on. Finally, we can modify existing types to form compound types. This chapter covers the built-in types and begins our coverage of how C++ supports more complicated types.

C++ 语言定义了几种基本类型：字符型、整型、浮点型等。C++ 还提供了可用于自定义数据类型的机制，标准库正是利用这些机制定义了许多更复杂的类型，比如可变长字符串 `string`、`vector` 等。此外，我们还能修改已有的类型以形成复合类型。本章介绍内置类型，并开始介绍 C++ 如何支持更复杂的类型。

Types determine what the data and operations in our programs mean. As we saw in [Chapter 1](#), the same statement

类型确定了数据和操作在程序中的意义。我们在[第一章](#)已经看到，如下的语句

```
i = i + j;
```

can mean different things depending on the types of `i` and `j`. If `i` and `j` are integers, then this statement has the ordinary, arithmetic meaning of `+`. However, if `i` and `j` are `Sales_item` objects, then this statement adds the components of these two objects.

有不同的含义，具体含义取决于 `i` 和 `j` 的类型。如果 `i` 和 `j` 都是整型，则这条语句表示一般的算术“`+`”运算；如果 `i` 和 `j` 都是 `Sales_item` 对象，则这条语句是将这两个对象的组成成分分别加起来。

In C++ the support for types is extensive: The language itself defines a set of primitive types and ways in which we can modify existing types. It also provides a set of features that allow us to define our own types. This chapter begins our exploration of types in C++ by covering the built-in types and showing how we associate a type with an object. It also introduces ways we can both modify types and can build our own types.

C++ 中对类型的支持是非常广泛的：语言本身定义了一组基本类型和修改已有类型的方法，还提供了一组特征用于自定义类型。本章通过介绍内置类型和如何关联类型与对象来探讨 C++ 中的类型。本章还将介绍更改类型和建立自定义类型的方法。

2.1. Primitive Built-in Types

2.1. 基本内置类型

C++ defines a set of **arithmetic types**, which represent integers, floating-point numbers, and individual characters and boolean values. In addition, there is a special type named **void**. The **void** type has no associated values and can be used in only a limited set of circumstances. The **void** type is most often used as the return type for a function that has no return value.

C++ 定义了一组表示整数、浮点数、单个字符和布尔值的**算术类型**，另外还定义了一种称为 **void** 的特殊类型。**void** 类型没有对应的值，仅用在有限的一些情况下，通常用作无返回值函数的返回类型。

The size of the arithmetic types varies across machines. By size, we mean the number of bits used to represent the type. The standard guarantees a minimum size for each of the arithmetic types, but it does not prevent compilers from using larger sizes. Indeed, almost all compilers use a larger size for **int** than is strictly required. [Table 2.1](#) (p. 36) lists the built-in arithmetic types and the associated minimum sizes.

算术类型的存储空间依机器而定。这里的存储空间是指用来表示该类型的位 (bit) 数。C++ 标准规定了每个算术类型的最小存储空间，但它并不阻止编译器使用更大的存储空间。事实上，对于 **int** 类型，几乎所有的编译器使用的存储空间都比所要求的大。[int 表 2.1](#) 列出了内置算术类型及其对应的最小存储空间。

Table 2.1. C++: Arithmetic Types

表 2.1. C++ 算术类型

Type	Meaning	Minimum Size
类型	含义	最小存储空间
bool	boolean	NA
char	character	8 bits
wchar_t	wide character	16 bits
short	short integer	16 bits
int	integer	16 bits
long	long integer	32 bits
float	single-precision floating-point	6 significant digits
double	double-precision floating-point	10 significant digits
long double	extended-precision floating-point	10 significant digits

 Because the number of bits varies, the maximum (or minimum) values that these types can represent also vary by machine.

因为位数的不同，这些类型所能表示的最大（最小）值也因机器的不同而有所不同。

2.1.1. Integral Types

2.1.1. 整型

The arithmetic types that represent integers, characters, and boolean values are collectively referred to as the **integral types**.

Section 2.1. Primitive Built-in Types

表示整数、字符和布尔值的算术类型合称为[整型](#)。

There are two character types: `char` and `wchar_t`. The `char` type is guaranteed to be big enough to hold numeric values that correspond to any character in the machine's basic character set. As a result, `chars` are usually a single machine byte. The `wchar_t` type is used for extended character sets, such as those used for Chinese and Japanese, in which some characters cannot be represented within a single `char`.

字符类型有两种：`char` 和 `wchar_t`。`char` 类型保证了有足够的空间，能够存储机器基本字符集中任何字符相应的数值，因此，`char` 类型通常是单个机器字节（byte）。`wchar_t` 类型用于扩展字符集，比如汉字和日语，这些字符集中的一些字符不能用单个 `char` 表示。

The types `short`, `int`, and `long` represent integer values of potentially different sizes. Typically, `shorts` are represented in half a machine [word](#), `ints` in a machine word, and `longs` in either one or two machine words (on 32-bit machines, `ints` and `longs` are usually the same size).

`short`、`int` 和 `long` 类型都表示整型值，存储空间的大小不同。一般，`short` 类型为半个[机器字长](#)，`int` 类型为一个机器字长，而 `long` 类型为一个或两个机器字长（在 32 位机器中 `int` 类型和 `long` 类型通常字长是相同的）。

Machine-Level Representation of The Built-in Types

内置类型的机器级表示

The C++ built-in types are closely tied to their representation in the computer's memory. Computers store data as a sequence of bits, each of which holds either 0 or 1. A segment of memory might hold

C++ 的内置类型与其在计算机的存储器中的表示方式紧密相关。计算机以位序列存储数据，每一位存储 **0** 或 **1**。一段内存可能存储着

00011011011100010110010000111011 ...

At the bit level, memory has no structure and no meaning.

在位这一级上，存储器是没有结构和意义的。

The most primitive way we impose structure on memory is by processing it in chunks. Most computers deal with memory as chunks of bits of particular sizes, usually powers of 2. They usually make it easy to process 8, 16, or 32 bits at a time, and chunks of 64 and 128 bits are becoming more common. Although the exact sizes can vary from one machine to another, we usually refer to a chunk of 8 bits as a "byte" and 32 bits, or 4 bytes, as a "word."

让存储具有结构的最基本方法是用块（chunk）处理存储。大部分计算机都使用特定位数的块来处理存储，块的位数一般是 2 的幂，因为这样可以一次处理 **8**、**16** 或 **32** 位。**64** 和 **128** 位的块如今也变得更为普遍。虽然确切的大小因机器不同而不同，但是通常将 **8** 位的块作为一个字节，**32** 位或 **4** 个字节作为一个“字”（word）”。

Most computers associate a number called an [address](#) with each [byte](#) in memory. Given a machine that has 8-bit bytes and 32-bit words, we might represent a word of memory as follows:

大多数计算机将存储器中的每一个字节和一个称为[地址](#)的数关联起来。对于一个 **8 位字节** 和 **32 位字** 的机器，我们可以将存储器的字表示如下：

736424	0	0	0	1	1	0	1	1
736425	0	1	1	1	0	0	0	1
736426	0	1	1	0	0	1	0	0
736427	0	0	1	1	1	0	1	1

In this illustration, each byte's address is shown on the left, with the 8 bits of the byte following the address.

在这个图中，左边是字节的地址，地址后面为字节的 **8 位**。

We can use an address to refer to any of several variously sized collections of bits starting at that address. It is possible to speak of the word at address 736424 or the byte at address 736426. We can say, for example, that the byte at address 736425 is not equal to the byte at address 736427.

可以用地址表示从该地址开始的任何几个不同大小的位集合。可以说地址为 **736424** 的字，也可以说地址为 **736426** 的字节。例如，可以说地址为**736425** 的字节和地址为 **736427** 的字节不相等。

To give meaning to the byte at address 736425, we must know the type of the value stored there. Once we know the type, we know how many bits are needed to represent a value of that type and how to interpret those bits.

要让地址为 **736425** 的字节具有意义，必须要知道存储在该地址的值的类型。一旦知道了该地址的值的类型，就知道了表示该类型的值需要多少位和如何解释这些位。

If we know that the byte at location 736425 has type "unsigned 8-bit integer," then we know that the byte represents

Section 2.1. Primitive Built-in Types

the number 112. On the other hand, if that byte is a character in the ISO-Latin-1 character set, then it represents the lower-case letter q. The bits are the same in both cases, but by ascribing different types to them, we interpret them differently.

如果知道地址为 736425 的字节的类型是 8 位无符号整数，那么就可以知道该字节表示整数 112。另外，如果这个字节是 ISO-Latin-1 字符集中的一个字符，那它就表示小写字母 q。虽然两种情况的位相同，但归属于不同类型，解释也就不同。

The type `bool` represents the truth values, `true` and `false`. We can assign any of the arithmetic types to a `bool`. An arithmetic type with value 0 yields a `bool` that holds `false`. Any nonzero value is treated as `true`.

`bool` 类型表示真值 `true` 和 `false`。可以将算术类型的任何值赋给 `bool` 对象。0 值算术类型代表 `false`，任何非 0 的值都代表 `true`。

Signed and Unsigned Types

带符号和无符号类型

The integral types, except the boolean type, may be either `signed` or `unsigned`. As its name suggests, a signed type can represent both negative and positive numbers (including zero), whereas an `unsigned` type represents only values greater than or equal to zero.

除 `bool` 类型外，整型可以是带符号的 (`signed`) 也可以是无符号的 (`unsigned`)。顾名思义，带符号类型可以表示正数也可以表示负数（包括 0），而无符号型只能表示大于或等于 0 的数。

The integers, `int`, `short`, and `long`, are all signed by default. To get an unsigned type, the type must be specified as `unsigned`, such as `unsigned long`. The `unsigned int` type may be abbreviated as `unsigned`. That is, `unsigned` with no other type implies `unsigned int`.

整型 `int`、`short` 和 `long` 都默认为带符号型。要获得无符号型则必须指定该类型为 `unsigned`，比如 `unsigned long`。`unsigned int` 类型可以简写为 `unsigned`，也就是说，`unsigned` 后不加其他类型说明符意味着是 `unsigned int`。

Unlike the other integral types, there are three distinct types for `char`: plain `char`, `signed char`, and `unsigned char`. Although there are three distinct types, there are only two ways a `char` can be represented. The `char` type is represented using either the `signed char` or `unsigned char` version. Which representation is used for `char` varies by compiler.

和其他整型不同，`char` 有三种不同的类型：`plain char`、`unsigned char` 和 `signed char`。虽然 `char` 有三种不同的类型，但只有两种表示方式。可以使用 `unsigned char` 或 `signed char` 表示 `char` 类型。使用哪种 `char` 表示方式由编译器而定。

How Integral Values Are Represented

整型值的表示

In an `unsigned` type, all the bits represent the value. If a type is defined for a particular machine to use 8 bits, then the `unsigned` version of this type could hold the values 0 through 255.

无符号型中，所有的位都表示数值。如果在某种机器中，定义一种类型使用 8 位表示，那么这种类型的 `unsigned` 型可以取值 0 到 255。

The C++ standard does not define how `signed` types are represented at the bit level. Instead, each compiler is free to decide how it will represent `signed` types. These representations can affect the range of values that a `signed` type can hold. We are guaranteed that an 8-bit `signed` type will hold at least the values from 127 through 127; many implementations allow values from 128 through 127.

C++ 标准并未定义 `signed` 类型如何用位来表示，而是由每个编译器自由决定如何表示 `signed` 类型。这些表示方式会影响 `signed` 类型的取值范围。8 位 `signed` 类型的取值肯定至少是从 -127 到 127，但也有许多实现允许取值从 -128 到 127。

Under the most common strategy for representing `signed` integral types, we can view one of the bits as a sign bit. Whenever the sign bit is 1, the value is negative; when it is 0, the value is either 0 or a positive number. An 8-bit integral `signed` type represented using a sign-bit can hold values from 128 through 127.

表示 `signed` 整型类型最常见的策略是用其中一个位作为符号位。符号位为 1，值就为负数；符号位为 0，值就为 0 或正数。一个 `signed` 整型取值是从 -128 到 127。

Assignment to Integral Types

整型的赋值

The type of an object determines the values that the object can hold. This fact raises the question of what happens when one tries to assign a value outside the allowable range to an object of a given type. The answer depends on whether the type is `signed` or `unsigned`.

Section 2.1. Primitive Built-in Types

对象的类型决定对象的取值。这会引起一个疑问：当我们试着把一个超出其取值范围的值赋给一个指定类型的对象时，结果会怎样呢？答案取决于这种类型是 `signed` 还是 `unsigned` 的。

For `unsigned` types, the compiler *must* adjust the out-of-range value so that it will fit. The compiler does so by taking the remainder of the value modulo the number of distinct values the `unsigned` target type can hold. An object that is an 8-bit `unsigned char`, for example, can hold values from 0 through 255 inclusive. If we assign a value outside this range, the compiler actually assigns the remainder of the value modulo 256. For example, we might attempt to assign the value 336 to an 8-bit `signed char`. If we try to store 336 in our 8-bit `unsigned char`, the actual value assigned will be 80, because 80 is equal to 336 modulo 256.

对于 `unsigned` 类型来说，编译器必须调整越界值使其满足要求。编译器会将该值对 `unsigned` 类型的可能取值数目求模，然后取所得值。比如 8 位的 `unsigned char`，其取值范围从 0 到 255（包括 255）。如果赋给超出这个范围的值，那么编译器将会取该值对 256 求模后的值。例如，如果试图将 336 存储到 8 位的 `unsigned char` 中，则实际赋值为 80，因为 80 是 336 对 256 求模后的值。

For the `unsigned` types, a negative value is always out of range. An object of `unsigned` type may never hold a negative value. Some languages make it illegal to assign a negative value to an `unsigned` type, but C++ does not.

对于 `unsigned` 类型来说，负数总是超出其取值范围。`unsigned` 类型的对象可能永远不会保存负数。有些语言中将负数赋给 `unsigned` 类型是非法的，但在 C++ 中这是合法的。



In C++ it is perfectly legal to assign a negative number to an object with `unsigned` type. The result is the negative value modulo the size of the type. So, if we assign 1 to an 8-bit `unsigned char`, the resulting value will be 255, which is 1 modulo 256.

C++ 中，把负值赋给 `unsigned` 对象是完全合法的，其结果是该负数对该类型的取值个数求模后的值。所以，如果把 -1 赋给 8 位的 `unsigned char`，那么结果是 255，因为 255 是 -1 对 256 求模后的值。

When assigning an out-of-range value to a `signed` type, it is up to the compiler to decide what value to assign. In practice, many compilers treat `signed` types similarly to how they are required to treat `unsigned` types. That is, they do the assignment as the remainder modulo the size of the type. However, we are not guaranteed that the compiler will do so for the `signed` types.

当将超过取值范围的值赋给 `signed` 类型时，由编译器决定实际赋的值。在实际操作中，很多的编译器处理 `signed` 类型的方式和 `unsigned` 类型类似。也就是说，赋值时是取该值对该类型取值数目求模后的值。然而我们不能保证编译器都会这样处理 `signed` 类型。

2.1.2. Floating-Point Types

2.1.2. 浮点型

The types `float`, `double`, and `long double` represent floating-point single-, double-, and extended-precision values. Typically, `floats` are represented in one word (32 bits), `doubles` in two words (64 bits), and `long double` in either three or four words (96 or 128 bits). The size of the type determines the number of significant digits a floating-point value might contain.

类型 `float`、`double` 和 `long double` 分别表示单精度浮点数、双精度浮点数和扩展精度浮点数。一般 `float` 类型用一个字（32 位）来表示，`double` 类型用两个字（64 位）来表示，`long double` 类型用三个或四个字（96 或 128 位）来表示。类型的取值范围决定了浮点数所含的有效数位数。



The `float` type is usually not precise enough for real programs. `float` is guaranteed to offer only 6 significant digits. The `double` type guarantees at least 10 significant digits, which is sufficient for most calculations.

对于实际的程序来说，`float` 类型精度通常是不够的——`float` 型只能保证 6 位有效数字，而 `double` 型至少可以保证 10 位有效数字，能满足大多数计算的需要。

Advice: Using the Built-in Arithmetic Types

建议：使用内置算术类型

Section 2.1. Primitive Built-in Types

The number of integral types in C++ can be bewildering. C++, like C, is designed to let programs get close to the hardware when necessary, and the integral types are defined to cater to the peculiarities of various kinds of hardware. Most programmers can (and should) ignore these complexities by restricting the types they actually use.

C++ 中整型数有点令人迷惑不解。就像 C 语言一样，C++ 被设计成允许程序在必要时直接处理硬件，因此整型被定义成满足各种各样硬件的特性。大多数程序员可以（应该）通过限制实际使用的类型来忽略这些复杂性。

In practice, many uses of integers involve counting. For example, programs often count the number of elements in a data structure such as a `vector` or an array. We'll see in [Chapters 3](#) and [4](#) that the library defines a set of types to use when dealing with the size of an object. When counting such elements it is always right to use the library-defined type intended for this purpose. When counting in other circumstances, it is usually right to use an `unsigned` value. Doing so avoids the possibility that a value is too large to fit results in a (seemingly) negative result.

实际上，许多人用整型进行计数。例如：程序经常计算像 `vector` 或数组这种数据结构的元素个数。在[第三章](#)和[第四章](#)中，我们将看到标准库定义了一组类型用于统计对象的大小。因此，当计数这些元素时使用标准库定义的类型总是正确的。其他情况下，使用 `unsigned` 类型比较明智，可以避免值越界导致结果为负数的可能性。

When performing integer arithmetic, it is rarely right to use `shorts`. In most programs, using `shorts` leads to mysterious bugs when a value is assigned to a `short` that is bigger than the largest number it can hold. What happens depends on the machine, but typically the value "wraps around" so that a number too large to fit turns into a large negative number. For the same reason, even though `char` is an integral type, the `char` type should be used to hold characters and not for computation. The fact that `char` is `signed` on some implementations and `unsigned` on others makes it problematic to use it as a computational type.

当执行整型算术运算时，很少使用 `short` 类型。大多数程序中，使用 `short` 类型可能会隐含赋值越界的错误。这个错误会产生什么后果将取决于所使用的机器。比较典型的情况是值“截断（wrap around）”以至于因越界而变成很大的负数。同样的道理，虽然 `char` 类型是整型，但是 `char` 类型通常用来存储字符而不用于计算。事实上，在某些应用中 `char` 类型被当作 `signed` 类型，在另外一些应用中则被当作 `unsigned` 类型，因此把 `char` 类型作为计算类型使用时容易出问题。

On most machines, integer calculations can safely use `int`. Technically speaking, an `int` can be as small as 16 bits too small for most purposes. In practice, almost all general-purpose machines use 32-bits for `ints`, which is often the same size used for `long`. The difficulty in deciding whether to use `int` or `long` occurs on machines that have 32-bit `ints` and 64-bit `longs`. On such machines, the run-time cost of doing arithmetic with `longs` can be considerably greater than doing the same calculation using a 32-bit `int`. Deciding whether to use `int` or `long` requires detailed understanding of the program and the actual run-time performance cost of using `long` versus `int`.

在大多数机器上，使用 `int` 类型进行整型计算不易出错。就技术上而言，`int` 类型用 16 位表示——这对大多数应用来说太小了。实际应用中，大多数通用机器都是使用和 `long` 类型一样长的 32 位来表示 `int` 类型。整型运算时，用 32 位表示 `int` 类型和用 64 位表示 `long` 类型的机器会出现应该选择 `int` 类型还是 `long` 类型的难题。在这些机器上，用 `long` 类型进行计算所付出的运行代价远远高于用 `int` 类型进行同样计算的代价，所以选择类型前要先了解程序的细节并且比较 `long` 类型与 `int` 类型的实际运行时性能代价。

Determining which floating-point type to use is easier: It is almost always right to use `double`. The loss of precision implicit in `float` is significant, whereas the cost of double precision calculations versus single precision is negligible. In fact, on some machines, double precision is faster than single. The precision offered by `long double` usually is unnecessary and often entails considerable extra run-time cost.

决定使用哪种浮点型就容易多了：使用 `double` 类型基本上不会有错。在 `float` 类型中隐式的精度损失是不能忽视的，而 `double` 类型精度代价相对于 `float` 类型精度代价可以忽略。事实上，有些机器上，`double` 类型比 `float` 类型的计算要快得多。`long double` 类型提供的精度通常没有必要，而且还需要承担额外的运行代价。

Exercises Section 2.1.2

Exercise What is the difference between an `int`, a `long`, and a `short` value?

2.1:

`int`、`long` 和 `short` 类型之间有什么差别？

Exercise What is the difference between an `unsigned` and a `signed` type?

2.2:

`unsigned` 和 `signed` 类型有什么差别？

Exercise If a `short` on a given machine has 16 bits then what is the largest number that can be assigned to a `short`? To an `unsigned short`?

如果在某机器上 `short` 类型占 16 位，那么可以赋给 `short` 类型的最大数是什么？`unsigned short` 类型的最大数又是什么？

Exercise What value is assigned if we assign 100,000 to a 16-bit `unsigned short`? What value is assigned if we assign 100,000 to a plain 16-bit `short`?

当给 16 位的 `unsigned short` 对象赋值 100 000 时，赋的值是什么？

Section 2.1. Primitive Built-in Types

Exercise What is the difference between a `float` and a `double`?
2.5: `float` 类型和 `double` 类型有什么差别?

Exercise To calculate a mortgage payment, what types would you use for the rate, principal, and payment? Explain why you selected each type.

要计算抵押贷款的偿还金额，利率、本金和付款额应分别选用哪种类型？解释你选择的理由。

2.2. Literal Constants

2.2. 字面值常量

A value, such as `42`, in a program is known as a **literal constant**: literal because we can speak of it only in terms of its value; constant because its value cannot be changed. Every literal has an associated type. For example, `0` is an `int` and `3.14159` is a `double`. Literals exist only for the built-in types. There are no literals of class types. Hence, there are no literals of any of the library types.

像 `42` 这样的值，在程序中被当作字面值常量。称之为字面值是因为只能用它的值称呼它，称之为常量是因为它的值不能修改。每个字面值都有相应的类型，例如：`0` 是 `int` 型，`3.14159` 是 `double` 型。只有内置类型存在字面值，没有类类型的字面值。因此，也没有任何标准库类型的字面值。

Rules for Integer Literals

整型字面值规则

We can write a literal integer constant using one of three notations: decimal, octal, or hexadecimal. These notations, of course, do not change the bit representation of the value, which is always binary. For example, we can write the value `20` in any of the following three ways:

定义字面值整数常量可以使用以下三种进制中的任一种：十进制、八进制和十六进制。当然这些进制不会改变其二进制位的表示形式。例如，我们能将值 `20` 定义成下列三种形式中的任意一种：

```
20      // decimal
024     // octal
0x14    // hexadecimal
```

Literal integer constants that begin with a leading `0` (zero) are interpreted as octal; those that begin with either `0x` or `0X` are interpreted as hexadecimal.

以 `0` (零) 开头的字面值整数常量表示八进制，以 `0x` 或 `0X` 开头的表示十六进制。

By default, the type of a literal integer constant is either `int` or `long`. The precise type depends on the value of the literal values that fit in an `int` are type `int` and larger values are type `long`. By adding a suffix, we can force the type of a literal integer constant to be type `long` or `unsigned` or `unsigned long`. We specify that a constant is a `long` by immediately following the value with either `L` or `l` (the letter "ell" in either uppercase or lowercase).

字面值整数常量的类型默认为 `int` 或 `long` 类型。其精度类型决定于字面值——其值适合 `int` 就是 `int` 类型，比 `int` 大的值就是 `long` 类型。通过增加后缀，能够强制将字面值整数常量转换为 `long`、`unsigned` 或 `unsigned long` 类型。通过在数值后面加 `L` 或者 `l` (字母“l”大写或小写) 指定常量为 `long` 类型。



When specifying a long, use the uppercase `L`: the lowercase letter `l` is too easily mistaken for the digit 1.

定义长整型时，应该使用大写字母 `L`。小写字母 `l` 很容易和数值 `1` 混淆。

In a similar manner, we can specify `unsigned` by following the literal with either `u` or `U`. We can obtain an `unsigned long` literal constant by following the value by both `L` and `u`. The suffix must appear with no intervening space:

类似地，可通过在数值后面加 `U` 或 `u` 定义 `unsigned` 类型。同时加 `L` 和 `U` 就能够得到 `unsigned long` 类型的字面值常量。但其后缀不能有空格：

```
128u      /* unsigned */          1024UL    /* unsigned long */
1L       /* long */           8Lu      /* unsigned long */
```

There are no literals of type `short`.

没有 `short` 类型的字面值常量。

Rules for Floating-Point Literals

浮点字面值规则

We can use either common decimal notation or scientific notation to write floating-point literal constants. Using scientific notation, the exponent is indicated either by `E` or `e`. By default, floating-point literals are type `double`. We indicate single precision by following the value with either `F` or `f`. Similarly, we specify extended precision by following the value with either `L` or `l` (again, use of the lowercase `l` is discouraged). Each pair of literals below denote the same underlying value:

通常可以用十进制或者科学计数法来表示浮点字面值常量。使用科学计数法时，指数用 `E` 或者 `e` 表示。默认的浮点字面值常量为 `double` 类型。在数值的后面加上 `F` 或 `f` 表示单精度。同样加上 `L` 或者 `l` 表示扩展精度（再次提醒，不提倡使用小写字母`l`）。下面每一组字面值表示相同的值：

<code>3.14159F</code>	<code>.001f</code>	<code>12.345L</code>	<code>0.</code>
<code>3.14159E0f</code>	<code>1E-3F</code>	<code>1.2345E1L</code>	<code>0e0</code>

Boolean and Character Literals

布尔字面值和字符字面值

The words `true` and `false` are literals of type `bool`:

单词 `true` 和 `false` 是布尔型的字面值：

```
bool test = false;
```

Printable character literals are written by enclosing the character within single quotation marks:

可打印的字符型字面值通常用一对单引号来定义：

```
'a'      '2'      ','      ' ' // blank
```

Such literals are of type `char`. We can obtain a wide-character literal of type `wchar_t` by immediately preceding the character literal with an `L`, as in

这些字面值都是 `char` 类型的。在字符字面值前加 `L` 就能够得到 `wchar_t` 类型的宽字符字面值。如：

```
L'a'
```

Escape Sequences for Nonprintable Characters

非打印字符的转义序列

Some characters are nonprintable. A nonprintable character is a character for which there is no visible image, such as backspace or a control character. Other characters have special meaning in the language, such as the single and double quotation marks, and the backslash. Nonprintable characters and special characters are written using an escape sequence. An escape sequence begins with a backslash. The language defines the following escape sequences:

有些字符是不可打印的。不可打印字符实际上是不可显示的字符，比如退格或者控制符。还有一些在语言中有特殊意义的字符，例如单引号、双引号和反斜线符号。不可打印字符和特殊字符都用转义字符书写。转义字符都以反斜线符号开始，C++ 语言中定义了如下转义字符：

newline	<code>\n</code>	horizontal tab	<code>\t</code>
换行符		水平制表符	
vertical tab	<code>\v</code>	backspace	<code>\b</code>
纵向制表符		退格符	
carriage return	<code>\r</code>	formfeed	<code>\f</code>
回车符		进纸符	
alert (bell) 符	<code>\a</code>	backslash	<code>\\\</code>
报警 (响铃) 符		反斜线	
question mark	<code>\?</code>	single quote	<code>\'</code>
疑问号		单引号	

Section 2.2. Literal Constants

double quote \"

双引号

We can write any character as a generalized escape sequence of the form

我们可以将任何字符表示为以下形式的通用转义字符:

\ooo

where ooo represents a sequence of as many as three octal digits. The value of the octal digits represents the numerical value of the character. The following examples are representations of literal constants using the ASCII character set:

这里 ooo 表示三个八进制数字，这三个数字表示字符的数字值。下面的例子是用 ASCII 码字符集表示字面值常量:

\7 (bell) \12 (newline) \40 (blank)
\0 (null) \062 ('2') \115 ('M')

The character represented by '\0' is often called a "null character," and has special significance, as we shall soon see.

字符'\0'通常表示"空字符 (null character)"，我们将会看到它有着非常特殊的意义。

We can also write a character using a hexadecimal escape sequence

同样也可以用十六进制转义字符来定义字符:

\xddd

consisting of a backslash, an x, and one or more hexadecimal digits.

它由一个反斜线符、一个 x 和一个或者多个十六进制数字组成。

Character String Literals

字符串字面值

All of the literals we've seen so far have primitive built-in types. There is one additional literal string literal that is more complicated. String literals are [arrays](#) of constant characters, a type that we'll discuss in more detail in [Section 4.3](#) (p. 130).

之前见过的所有字面值都有基本内置类型。还有一种字面值（字符串字面值）更加复杂。字符串字面值是一串常量字符，这种类型将在[第 4.3 节](#)详细说明。

String literal constants are written as zero or more characters enclosed in double quotation marks. Nonprintable characters are represented by their underlying escape sequence.

字符串字面值常量用双引号括起来的零个或者多个字符表示。不可打印字符表示成相应的转义字符。

```
"Hello World!"           // simple string literal  
""                      // empty string literal  
"\nCC\toptions\tfile.[cc]\n" // string literal using newlines and tabs
```

For compatibility with C, string literals in C++ have one character in addition to those typed in by the programmer. Every string literal ends with a null character added by the compiler. A character literal

为了兼容 C 语言，C++ 中所有的字符串字面值都由编译器自动在末尾添加一个空字符。字符字面值

'A' // single quote: character literal

represents the single character A, whereas

表示单个字符 A，然而

"A" // double quote: character string literal

represents an array of two characters: the letter A and the null character.

表示包含字母 A 和空字符两个字符的字符串。

Just as there is a wide character literal, such as

正如存在宽字符字面值，如

L'a'

there is a wide string literal, again preceded by L, such as

Section 2.2. Literal Constants

也存在宽字符串字面值，一样在前面加“`L`”，如

```
L"a wide string literal"
```

The type of a wide string literal is an array of constant wide characters. It is also terminated by a wide null character.

宽字符串字面值是一串常量宽字符，同样以一个宽空字符结束。

Concatenated String Literals

字符串字面值的连接

Two string literals (or two wide string literals) that appear adjacent to one another and separated only by spaces, tabs, or newlines are concatenated into a single new string literal. This usage makes it easy to write long literals across separate lines:

两个相邻的仅由空格、制表符或换行符分开的字符串字面值（或宽字符串字面值），可连接成一个新字符串字面值。这使得多行书写长字符串字面值变得简单：

```
// concatenated long string literal
std::cout << "a multi-line "
           "string literal "
           "using concatenation"
           << std::endl;
```

When executed this statement would print:

执行这条语句将会输出：

```
a multi-line string literal using concatenation
```

What happens if you attempt to concatenate a string literal and a wide string literal? For example:

如果连接字符串字面值和宽字符串字面值，将会出现什么结果呢？例如：

```
// Concatenating plain and wide character strings is undefined
std::cout << "multi-line " L"literal " << std::endl;
```

The result is undefined that is, there is no standard behavior defined for concatenating the two different types. The program might appear to work, but it also might crash or produce garbage values. Moreover, the program might behave differently under one compiler than under another.

其结果是未定义的，也就是说，连接不同类型的行为标准没有定义。这个程序可能会执行，也可能会崩溃或者产生没有用的值，而且在不同的编译器下程序的动作可能不同。

Multi-Line Literals

多行字面值

There is a more primitive (and less useful) way to handle long strings that depends on an infrequently used program formatting feature: Putting a backslash as the last character on a line causes that line and the next to be treated as a single line.

处理长字符串有一个更基本的（但不常使用）方法，这个方法依赖于很少使用的程序格式化特性：在一行为的末尾加一反斜线符号可将此行和下一行当作同一行处理。

As noted on page 14, C++ programs are largely free-format. In particular, there are only a few places that we may not insert whitespace. One of these is in the middle of a word. In particular, we may not break a line in the middle of a word. We can circumvent this rule by using a backslash:

正如第 1.4.1 节提到的，C++ 的格式非常自由。特别是有一些地方不能插入空格，其中之一是在单词中间。特别是不能在单词中间断开一行。但可以通过使用反斜线符号巧妙实现：

```
// ok: A \ before a newline ignores the line break
std::cout \
t << "Hi" << st\
d::endl;
```

is equivalent to

等价于

```
std::cout << "Hi" << std::endl;
```

We could use this feature to write a long string literal:

可以使用这个特性来编写长字符串字面值：

```
// multiline string literal
std::cout << "a multi-line \
string literal \
using a backslash"
```

```

        << std::endl;
    return 0;
}

```

Note that the backslash must be the last thing on the line no comments or trailing blanks are allowed. Also, any leading spaces or tabs on the subsequent lines are part of the literal. For this reason, the continuation lines of the long literal do not have the normal indentation.

注意反斜线符号必须是该行的尾字符——不允许有注释或空格符。同样，后继行首的任何空格和制表符都是字符串字面值的一部分。正因如此，长字符串字面值的后继行才不会有正常的缩进。

Advice: Don't Rely on Undefined Behavior

建议：不要依赖未定义行为

Programs that use undefined behavior are in error. If they work, it is only by coincidence. Undefined behavior results from a program error that the compiler cannot detect or from an error that would be too much trouble to detect.

使用了未定义行为的程序都是错误的，即使程序能够运行，也只是巧合。未定义行为源于编译器不能检测到的程序错误或太麻烦以至无法检测的错误。

Unfortunately, programs that contain undefined behavior can appear to execute correctly in some circumstances and/or on one compiler. There is no guarantee that the same program, compiled under a different compiler or even a subsequent release of the current compiler, will continue to run correctly. Nor is there any guarantee that what works with one set of inputs will work with another.

不幸的是，含有未定义行为的程序在有些环境或编译器中可以正确执行，但并不能保证同一程序在不同编译器中甚至在当前编译器的后继版本中会继续正确运行，也不能保证程序在一组输入上可以正确运行且在另一组输入上也能够正确运行。

Programs should not (knowingly) rely on undefined behavior. Similarly, programs usually should not rely on machine-dependent behavior, such as assuming that the size of an `int` is a fixed and known value. Such programs are said to be *nonportable*. When the program is moved to another machine, any code that relies on machine-dependent behavior may have to be found and corrected. Tracking down these sorts of problems in previously working programs is, mildly put, a profoundly unpleasant task.

程序不应该依赖未定义行为。同样地，通常程序不应该依赖机器相关的行为，比如假定 `int` 的位数是个固定且已知的值。我们称这样的程序是不可移植的。当程序移植到另一台机器上时，要寻找并更改任何依赖机器相关操作的代码。在本来可以运行的程序中寻找这类问题是一项非常不愉快的任务。

Exercises Section 2.2

Exercise 2.7: Explain the difference between the following sets of literal constants:

解释下列字面常量的不同之处。

- (a) 'a', L'a', "a", L"a"
- (b) 10, 10u, 10L, 10uL, 012, 0xC
- (c) 3.14, 3.14f, 3.14L

Exercise 2.8: Determine the type of each of these literal constants:

确定下列字面常量的类型：

- (a) -10 (b) -10u (c) -10. (d) -10e-2

Exercise 2.9: Which, if any, of the following are illegal?

下列哪些（如果有）是非法的？

- (a) "Who goes with F\145rgus?\012"
- (b) 3.14elL (c) "two" L"some"
- (d) 1024f (e) 3.14UL
- (f) "multiple line
comment"

Exercise 2.10: Using escape sequences, write a program to print `2M` followed by a newline. Modify the program to print `2`, then a tab, then an `M`, followed by a newline.

使用转义字符编写一段程序，输出 `2M`，然后换行。修改程序，输出 `2`，跟着一个制表符，然后是 `M`，最后是换行符。

2.3. Variables

2.3. 变量

Imagine that we are given the problem of computing 2 to the power of 10. Our first attempt might be something like

如果要计算 2 的 10 次方，我们首先想到的可能是：

```
#include <iostream>
int main()
{
    // a first, not very good, solution
    std::cout << "2 raised to the power of 10: ";
    std::cout << 2*2*2*2*2*2*2*2*2*2;
    std::cout << std::endl;
    return 0;
}
```

This program solves the problem, although we might double- or triple-check to make sure that exactly 10 literal instances of 2 are being multiplied. Otherwise, we're satisfied. Our program correctly generates the answer 1,024.

这个程序确实解决了问题，尽管我们可能要一而再、再而三地检查确保恰好有 10 个字面值常量 2 相乘。这个程序产生正确的答案 1024。

We're next asked to compute 2 raised to the power of 17 and then to the power of 23. Changing our program each time is a nuisance. Worse, it proves to be remarkably error-prone. Too often, the modified program produces an answer with one too few or too many instances of 2.

接下来要计算 2 的 17 次方，然后是 23 次方。而每次都要改变程序是很麻烦的事。更糟的是，这样做还容易引起错误。修改后的程序常常会产生多乘或少乘 2 的结果。

An alternative to the explicit brute force power-of-2 computation is twofold:

替代这种蛮力型计算的方法包括两部分内容：

1. Use named objects to perform and print each computation.

使用已命名对象执行运算并输出每次计算。

2. Use flow-of-control constructs to provide for the repeated execution of a sequence of program statements while a condition is true.

使用控制流结构，当某个条件为真时重复执行一系列程序语句。

Here, then, is an alternative way to compute 2 raised to the power of 10:

以下是计算 2 的 10 次方的替代方法：

```
#include <iostream>
int main()
{
    // local objects of type int
    int value = 2;
    int pow = 10;
    int result = 1;
    // repeat calculation of result until cnt is equal to pow
    for (int cnt = 0; cnt != pow; ++cnt)
        result *= value;    // result = result * value;
    std::cout
        << value
        << " raised to the power of "
        << pow << ": \t"
        << result << std::endl;
    return 0;
}
```

`value`, `pow`, `result`, and `cnt` are variables that allow for the storage, modification, and retrieval of values. The `for` loop allows for the repeated execution of our calculation until it's been executed `pow` times.

`value`、`pow`、`result` 和 `cnt` 都是变量，可以对数值进行存储、修改和查询。`for` 循环使得计算过程重复执行 `pow` 次。

Exercises Section 2.3

Exercise 2.11: Write a program that prompts the user to input two numbers, the base and exponent. Print the result of raising the base to the power of the exponent.

编写程序，要求用户输入两个数——底数 (base) 和指数 (exponent)，输出底数的指数次方的结果。

Key Concept: Strong Static Typing

关键概念：强静态类型

C++ is a statically typed language, which means that types are checked at compile time. The process by which types are checked is referred to as type-checking.

C++ 是一门静态类型语言，在编译时会作类型检查。

In most languages, the type of an object constrains the operations that the object can perform. If the type does not support a given operation, then an object of that type cannot perform that operation.

在大多数语言中，对象的类型限制了对象可以执行的操作。如果某种类型不支持某种操作，那么这种类型的对象也就不能执行该操作。

In C++, whether an operation is legal or not is checked at compile time. When we write an expression, the compiler checks that the objects used in the expression are used in ways that are defined by the type of the objects. If not, the compiler generates an error message; an executable file is not produced.

在 C++ 中，操作是否合法是在编译时检查的。当编写表达式时，编译器检查表达式中的对象是否按该对象的类型定义的使用方式使用。如果不是的话，那么编译器会提示错误，而不产生可执行文件。

As our programs, and the types we use, get more complicated, we'll see that static [type checking](#) helps find bugs in our programs earlier. A consequence of static checking is that the type of every entity used in our programs must be known to the compiler. Hence, we must define the type of a variable before we can use that variable in our programs.

随着程序和使用的类型变得越来越复杂，我们将看到静态类型检查能帮助我们更早地发现错误。静态类型检查使得编译器必须能识别程序中的每个实体的类型。因此，程序中使用变量前必须先定义变量的类型

2.3.1. What Is a Variable?

2.3.1. 什么是变量

A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable. C++ programmers tend to refer to variables as "variables" or as "objects" interchangeably.

变量提供了程序可以操作的有名字的存储区。C++ 中的每一个变量都有特定的类型，该类型决定了变量的内存大小和布局、能够存储于该内存中的值的取值范围以及可应用在该变量上的操作集。C++ 程序员常常把变量称为“变量”或“对象 (object) ”。

Lvalues and Rvalues

左值和右值

We'll have more to say about expressions in [Chapter 5](#), but for now it is useful to know that there are two kinds of expressions in C++:

我们在[第五章](#)再详细探讨表达式，现在先介绍 C++ 的两种表达式：

1. [lvalue](#) (pronounced "ell-value"): An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.

[左值](#) (发音为 ell-value) : 左值可以出现在赋值语句的左边或右边。

2. [rvalue](#) (pronounced "are-value"): An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

[右值](#) (发音为 are-value) : 右值只能出现在赋值的右边，不能出现在赋值语句的左边。

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned. Given the variables:

变量是左值，因此可以出现在赋值语句的左边。数字字面值是右值，因此不能被赋值。给定以下变量：

```
int units_sold = 0;
double sales_price = 0, total_revenue = 0;
```

it is a compile-time error to write either of the following:

下列两条语句都会产生编译错误：

```
// error: arithmetic expression is not an lvalue
units_sold * sales_price = total_revenue;
// error: literal constant is not an lvalue
0 = 1;
```

Some operators, such as assignment, require that one of their operands be an lvalue. As a result, lvalues can be used in more contexts than can rvalues. The context in which an lvalue appears determines how it is used. For example, in the expression

有些操作符，比如赋值，要求其中的一个操作数必须是左值。结果，可以使用左值的上下文比右值更广。左值出现的上下文决定了左值是如何使用的。例如，表达式

```
units_sold = units_sold + 1;
```

the variable `units_sold` is used as the operand to two different operators. The `+` operator cares only about the values of its operands. The value of a variable is the value currently stored in the memory associated with that variable. The effect of the addition is to fetch that value and add one to it.

中，`units_sold` 变量被用作两种不同操作符的操作数。`+` 操作符仅关心其操作数的值。变量的值是当前存储在和该变量相关联的内存中的值。加法操作符的作用是取得变量的值并加 1。

The variable `units_sold` is also used as the left-hand side of the `=` operator. The `=` operator reads its right-hand side and writes to its left-hand side. In this expression, the result of the addition is stored in the storage associated with `units_sold`; the previous value in `units_sold` is overwritten.

变量 `units_sold` 也被用作 `=` 操作符的左操作数。`=` 操作符读取右操作数并写到左操作数。在这个表达式中，加法运算的结果被保存到与 `units_sold` 相关联的存储单元中，而 `units_sold` 之前的值则被覆盖。



In the course of the text, we'll see a number of situations in which the use of an rvalue or lvalue impacts the behavior and/or the performance of our programs in particular when passing and returning values from a function.

在本书中，我们将看到在许多情形中左值或右值的使用影响程序的操作和/或性能——特别是在向函数传递值或从函数中返回值的时候。

Exercises Section 2.3.1

Exercise

2.12: Distinguish between an lvalue and an rvalue; show examples of each.

区分左值和右值，并举例说明。

Exercise

2.13: Name one case where an lvalue is required.

举出一个需要左值的例子。

Terminology: What Is an object?

术语：什么是对象？

C++ programmers tend to be cavalier in their use of the term *object*. Most generally, an object is a region of memory that has a type. More specifically, evaluating an expression that is an lvalue yields an object.

C++ 程序员经常随意地使用术语对象。一般而言，对象就是内存中具有类型的区域。说得更具体一些，计算左值表达式就会产生对象。

Strictly speaking, some might reserve the term *object* to describe only variables or values of class types. Others might distinguish between named and unnamed objects, always referring to variables when discussing named objects. Still others distinguish between objects and values, using the term *object* for data that can be changed by the program and using the term *value* for those that are read-only.

严格地说，有些人只把术语对象用于描述变量或类类型的值。有些人还区别有名字的对象和没名字的对象，当谈到有名字的对象时一般指变量。还有一些人

Section 2.3. Variables

区分对象和值，用术语对象描述可被程序改变的数据，用术语值描述只读数据。

In this book, we'll follow the more colloquial usage that an object is a region of memory that has a type. We will freely use *object* to refer to most of the data manipulated by our programs regardless of whether those data have built-in or class type, are named or unnamed, or are data that can be read or written.

在本书中，我们遵循更为通用的用法，即对象是内存中具有类型的区域。我们可以自由地使用对象描述程序中可操作的大部分数据，而不管这些数据是内置类型还是类类型，是有名字的还是没名字的，是可读的还是可写的。

2.3.2. The Name of a Variable

2.3.2. 变量名

The name of a variable, its **identifier**, can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper- and lowercase letters are distinct: Identifiers in C++ are case-sensitive. The following defines four distinct identifiers:

变量名，即变量的**标识符**，可以由字母、数字和下划线组成。变量名必须以字母或下划线开头，并且区分大小写字母：C++ 中的标识符都是大小写敏感的。下面定义了 4 个不同的标识符：

```
// declares four different int variables
int somename, someName, SomeName, SOMENAME;
```



There is no language-imposed limit on the permissible length of a name, but out of consideration for others that will read and/or modify our code, it should not be too long.

语言本身并没有限制变量名的长度，但考虑到将会阅读和/或修改我们的代码的其他人，变量名不应太长。

For example,

例如：

```
gosh_this_is_an_impossibly_long_name_to_type
```

is a really bad identifier name.

就是一个糟糕的标识符名。

C++ Keywords

C++ 关键字

C++ reserves a set of words for use within the language as keywords. Keywords may not be used as program identifiers. [Table 2.2](#) on the next page lists the complete set of C++ keywords.

C++ 保留了一组词用作该语言的关键字。关键字不能用作程序的标识符。[表 2.2](#) 列出了 C++ 所有的关键字。

Table 2.2. C++ Keywords

表 2.2. C++ 关键字

asm	do	if	return	try
auto	double	inline	short	typedef
bool	dynamic_cast	int	signed	typeid
break	else	long	sizeof	typename
case	enum	mutable	static	union
catch	explicit	namespace	static_cast	unsigned
char	export	new	struct	using
class	extern	operator	switch	virtual

Section 2.3. Variables

const	false	private	template	void
const_cast	float	protected	this	volatile
continue	for	public	throw	wchar_t
default	friend	register	true	while
delete	goto	reinterpret_cast		

C++ also reserves a number of words that can be used as alternative names for various operators. These alternative names are provided to support character sets that do not support the standard set of C++ operator symbols. These names, listed in [Table 2.3](#), also may not be used as identifiers:

C++ 还保留了一些词用作各种操作符的替代名。这些替代名用于支持某些不支持标准C++操作符号集的字符集。它们也不能用作标识符。[表 2.3](#)列出了这些替代名。

Table 2.3. C++ Operator Alternative Names

表 2.3. C++ 操作符替代名

and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bitor	not	or	xor	

In addition to the keywords, the standard also reserves a set of identifiers for use in the library. Identifiers cannot contain two consecutive underscores, nor can an identifier begin with an underscore followed immediately by an upper-case letter. Certain identifiers those that are defined outside a function may not begin with an underscore.

除了关键字，C++ 标准还保留了一组标识符用于标准库。标识符不能包含两个连续的下划线，也不能以下划线开头后面紧跟一个大写字母。有些标识符（在函数外定义的标识符）不能以下划线开头。

Conventions for Variable Names

变量命名习惯

There are a number of generally accepted conventions for naming variables. Following these conventions can improve the readability of a program.

变量命名有许多被普遍接受的习惯，遵循这些习惯可以提高程序的可读性。

- A variable name is normally written in lowercase letters. For example, one writes `index`, not `Index` or `INDEX`.
变量名一般用小写字母。例如，通常会写成 `index`，而不写成 `Index` 或 `INDEX`。
- An identifier is given a mnemonic name that is, a name that gives some indication of its use in a program, such as `on_loan` or `salary`.
标识符应使用能帮助记忆的名字，也就是说，能够提示其在程序中的用法的名字，如 `on_loan` 或 `salary`。
- An identifier containing multiple words is written either with an underscore between each word or by capitalizing the first letter of each embedded word. For example, one generally writes `student_loan` or `studentLoan`, not `studentloan`.
包含多个词的标识符书写为在每个词之间添加一个下划线，或者每个内嵌的词的第一个字母都大写。例如通常会写成 `student_loan` 或 `studentLoan`，而不写成 `studentloan`。



The most important aspect of a naming convention is that it be applied consistently.

命名习惯最重要的是保持一致。

Exercises Section 2.3.2

Exercise Which, if any, of the following names are invalid? Correct each identified invalid name.

2.14:

下面哪些（如果有）名字是非法的？更正每个非法的标识符名字。

- | | |
|---------------------------|-----------------------|
| (a) int double = 3.14159; | (b) char _; |
| (c) bool catch-22; | (d) char 1_or_2 ='1'; |
| (e) float Float = 3.14f; | |

2.3.3. Defining Objects

2.3.3. 定义对象

The following statements define five variables:

下列语句定义了 5 个变量：

```
int units_sold;
double sales_price, avg_price;
std::string title;
Sales_item curr_book;
```

Each definition starts with a **type specifier**, followed by a comma-separated list of one or more names. A semicolon terminates the definition. The type specifier names the type associated with the object: `int`, `double`, `std::string`, and `Sales_item` are all names of types. The types `int` and `double` are built-in types, `std::string` is a type defined by the library, and `Sales_item` is a type that we used in [Section 1.5](#) (p. 20) and will define in subsequent chapters. The type determines the amount of storage that is allocated for the variable and the set of operations that can be performed on it.

每个定义都是以**类型说明符**开始，后面紧跟着以逗号分开的含有一个或多个说明符的列表。分号结束定义。类型说明符指定与对象相关联的类型：`int`、`double`、`std::string` 和 `Sales_item` 都是类型名。其中 `int` 和 `double` 是内置类型，`std::string` 是标准库定义的类型，`Sales_item` 是我们在[第 1.5 节](#)使用的类型，将会在后面章节定义。类型决定了分配给变量的存储空间的大小和可以在其上执行的操作。

Multiple variables may be defined in a single statement:

多个变量可以定义在同一条语句中：

```
double salary, wage;      // defines two variables of type double
int month,
    day, year;        // defines three variables of type int
std::string address;     // defines one variable of type std::string
```

Initialization

初始化

A definition specifies a variable's type and identifier. A definition may also provide an initial value for the object. An object defined with a specified first value is spoken of as **initialized**. C++ supports two forms of **variable initialization**: **copy-initialization** and **direct-initialization**. The copy-initialization syntax uses the equal (`=`) symbol; direct-initialization places the initializer in parentheses:

变量定义指定了变量的类型和标识符，也可以为对象提供初始值。定义时指定了初始值的对象被称为是**已初始化的**。C++ 支持两种**初始化变量**的形式：**复制初始化**和**直接初始化**。复制初始化语法用等号 (`=`)，直接初始化则是把初始化式放在括号中：

```
int ival(1024);      // direct-initialization
int ival = 1024;     // copy-initialization
```

In both cases, `ival` is initialized to `1024`.

这两种情形中，`ival` 都被初始化为 `1024`。



Although, at this point in the book, it may seem obscure to the reader, in C++ it is essential to understand that initialization is not assignment. Initialization happens when a variable is created and gives that variable its initial value. Assignment involves obliterating an object's current value and replacing that value with a new one.

虽然在本书到目前为止还没有清楚说明，但是在 C++ 中理解“初始化不是赋值”是必要的。初始化指创建变量并给它赋初始值，而赋值则是擦除对象的当前值并用新值代替。

Section 2.3. Variables

Many new C++ programmers are confused by the use of the `=` symbol to initialize a variable. It is tempting to think of initialization as a form of assignment. But initialization and assignment are different operations in C++. This concept is particularly confusing because in many other languages the distinction is irrelevant and can be ignored. Moreover, even in C++ the distinction rarely matters until one attempts to write fairly complex classes. Nonetheless, it is a crucial concept and one that we will reiterate throughout the text.

使用 `=` 来初始化变量使得许多 C++ 编程新手感到迷惑，他们很容易把初始化当成是赋值的一种形式。但是在 C++ 中初始化和赋值是两种不同的操作。这个概念特别容易误导人，因为在许多其他的语言中这两者的差别不过是枝节问题因而可以被忽略。即使在 C++ 中也只有在编写非常复杂的类时才会凸显这两者之间的区别。无论如何，这是一个关键的概念，也是我们将会在整本书中反复强调的概念。



There are subtle differences between copy- and direct-initialization when initializing objects of a class type. We won't completely explain these differences until [Chapter 13](#). For now, it's worth knowing that the direct syntax is more flexible and can be slightly more efficient.

当初始化类类型对象时，复制初始化和直接初始化之间的差别是很微妙的。我们在[第十三章](#)再详细解释它们之间的差别。现在我们只需知道，直接初始化语法更灵活且效率更高。

Using Multiple Initializers

使用多个初始化式

When we initialize an `object` of a built-in type, there is only one way to do so: We supply a value, and that value is copied into the newly defined object. For built-in types, there is little difference between the direct and the copy forms of initialization.

初始化内置类型的[对象](#)只有一种方法：提供一个值，并且把这个值复制到新定义的对象中。对内置类型来说，复制初始化和直接初始化几乎没有差别。

For objects of a class type, there are initializations that can be done only using direct-initialization. To understand why, we need to know a bit about how classes control initialization.

对类类型的对象来说，有些初始化仅能用直接初始化完成。要想理解其中缘由，需要初步了解类是如何控制初始化的。

Each class may define one or more special member functions ([Section 1.5.2](#), p. 24) that say how we can initialize variables of the class type. The member functions that define how initialization works are known as `constructors`. Like any function, a constructor can take multiple arguments. A class may define several constructors, each of which must take a different number or type of arguments.

每个类都可能会定义一个或几个特殊的成员函数（[第 1.5.2 节](#)）来告诉我们如何初始化类类型的变量。定义如何进行初始化的成员函数称为[构造函数](#)。和其他函数一样，构造函数能接受多个参数。一个类可以定义几个构造函数，每个构造函数必须接受不同数目或者不同类型的参数。

As an example, we'll look a bit at the `string` class, which we'll cover in more detail in [Chapter 3](#). The `string` type is defined by the library and holds character strings of varying sizes. To use `strings`, we must include the `string` header. Like the IO types, `string` is defined in the `std` namespace.

我们以 `string` 类为例（`string` 类将在[第三章](#)详细讨论）。`string` 类型在标准库中定义，用于存储不同长度的字符串。使用 `string` 时必须包含 `string` 头文件。和 IO 类型一样，`string` 定义在 `std` 命名空间中。

The `string` class defines several constructors, giving us various ways to initialize a `string`. One way we can initialize a `string` is as a copy of a character string literal:

`string` 类定义了几个构造函数，使得我们可以用不同的方式初始化 `string` 对象。其中一种初始化 `string` 对象的方式是作为字符串字面值的副本：

```
#include <string>
// alternative ways to initialize string from a character string literal
std::string titleA = "C++ Primer, 4th Ed.";
std::string titleB("C++ Primer, 4th Ed.");
```

In this case, either initialization form can be used. Both definitions create a `string` object whose initial value is a copy of the specified string literal.

本例中，两种初始化方式都可以使用。两种定义都创建了一个 `string` 对象，其初始值都是指定的字符串字面值的副本。

However, we can also initialize a `string` from a count and a character. Doing so creates a `string` containing the specified character repeated as many times as indicated by the count:

也可以通过一个计数器和一个字符初始化 `string` 对象。这样创建的对象包含重复多次的指定字符，重复次数由计数器指定：

```
std::string all_nines(10, '9'); // all_nines= "9999999999"
```

In this case, the only way to initialize `all_nines` is by using the direct form of initialization. It is not possible to use copy-initialization with multiple initializers.

本例中，初始化 `all_nines` 的唯一方法是直接初始化。有多个初始化式时不能使用复制初始化。

Initializing Multiple Variables

初始化多个变量

When a definition defines two or more variables, each variable may have its own initializer. The name of an object becomes visible immediately, and so it is possible to initialize a subsequent variable to the value of one defined earlier in the same definition. Initialized and uninitialized variables may be defined in the same definition. Both forms of initialization syntax may be intermixed:

当一个定义中定义了两个以上变量的时候，每个变量都可能有自己的初始化式。对象的名字立即变成可见，所以可以用同一个定义中前面已定义变量的值初始化后面的变量。已初始化变量和未初始化变量可以在同一个定义中定义。两种形式的初始化文法可以相互混合。

```
#include <string>
// ok: salary defined and initialized before it is used to initialize wage
double salary = 9999.99,
    wage(salary + 0.01);
// ok: mix of initialized and uninitialized
int interval,
    month = 8, day = 7, year = 1955;
// ok: both forms of initialization syntax used
std::string title("C++ Primer, 4th Ed."),
    publisher = "A-W";
```

An object can be initialized with an arbitrarily complex expression, including the return value of a function:

对象可以用任意复杂的表达式（包括函数的返回值）来初始化：

```
double price = 109.99, discount = 0.16;
double sale_price = apply_discount(price, discount);
```

In this example, `apply_discount` is a function that takes two values of type `double` and returns a value of type `double`. We pass the variables `price` and `discount` to that function and use its return value to initialize `sale_price`.

本例中，函数 `apply_discount` 接受两个 `double` 类型的值并返回一个 `double` 类型的值。将变量 `price` 和 `discount` 传递给函数，并且用它的返回值来初始化 `sale_price`。

Exercises Section 2.3.3

Exercise What, if any, are the differences between the following definitions:

2.15: 下面两个定义是否不同？有何不同？

```
int month = 9, day = 7;
int month = 09, day = 07;
```

If either definition contains an error, how might you correct the problem?

如果上述定义有错的话，那么应该怎样改正呢？

Exercise Assuming `calc` is a function that returns a `double`, which, if any, of the following are illegal definitions? Correct any that are identified as illegal.

假设 `calc` 是一个返回 `double` 对象的函数。下面哪些是非法定义？改正所有的非法定义。

- (a) int car = 1024, auto = 2048;
- (b) int ival = ival;
- (c) std::cin >> int input_value;
- (d) double salary = wage = 9999.99;
- (e) double calc = calc();

2.3.4. Variable Initialization Rules

2.3.4. 变量初始化规则

When we define a variable without an initializer, the system sometimes initializes the variable for us. What value, if any, is supplied depends on the

Section 2.3. Variables

type of the variable and may depend on where it is defined.

当定义没有初始化式的变量时，系统有时候会帮我们初始化变量。这时，系统提供什么样的值取决于变量的类型，也取决于变量定义的位置。

Initialization of Variables of Built-in Type

内置类型变量的初始化

Whether a variable of built-in type is automatically initialized depends on where it is defined. Variables defined outside any function body are initialized to zero. Variables of built-in type defined inside the body of a function are **uninitialized**. Using an uninitialized variable for anything other than as the left-hand operand of an assignment is undefined. Bugs due to uninitialized variables can be hard to find. As we cautioned on page 42, you should never rely on undefined behavior.

内置类型变量是否自动初始化取决于变量定义的位置。在函数体外定义的变量都初始化成 0，在函数体里定义的内置类型变量不进行自动初始化。除了用作赋值操作符的左操作数，未初始化变量用作任何其他用途都是没有定义的。未初始化变量引起的错误难于发现。正如我们在第 2.2 节劝告的，永远不要依赖未定义行为。

Caution: Uninitialized Variables Cause Run-Time Problems

警告：未初始化的变量引起运行问题

Using an uninitialized object is a common program error, and one that is often difficult to uncover. The compiler is not required to detect a use of an uninitialized variable, although many will warn about at least some uses of uninitialized variables. However, no compiler can detect all uses of uninitialized variables.

使用未初始化的变量是常见的程序错误，通常也是难以发现的错误。虽然许多编译器都至少会提醒不要使用未初始化变量，但是编译器并未被要求去检测未初始化变量的使用。而且，没有一个编译器能检测出所有未初始化变量的使用。

Sometimes, we're lucky and using an uninitialized variable results in an immediate crash at run time. Once we track down the location of the crash, it is usually pretty easy to see that the variable was not properly initialized.

有时我们很幸运，使用未初始化的变量导致程序在运行时突然崩溃。一旦跟踪到程序崩溃的位置，就可以轻易地发现没有正确地初始化变量。

Other times, the program completes but produces erroneous results. Even worse, the results can appear correct when we run our program on one machine but fail on another. Adding code to the program in an unrelated location can cause what we thought was a correct program to suddenly start to produce incorrect results.

但有时，程序运行完毕却产生错误的结果。更糟糕的是，程序运行在一部机器上时能产生正确的结果，但在另外一部机器上却不能得到正确的结果。添加代码到程序的一些不相关的位置，会导致我们认为是正确的程序产生错误的结果。

The problem is that uninitialized variables actually do have a value. The compiler puts the variable somewhere in memory and treats whatever bit pattern was in that memory as the variable's initial state. When interpreted as an integral value, any bit pattern is a legitimate value although the value is unlikely to be one that the programmer intended. Because the value is legal, using it is unlikely to lead to a crash. What it is likely to do is lead to incorrect execution and/or incorrect calculation.

问题出在未初始化的变量事实上都有一个值。编译器把该变量放到内存中的某个位置，而把这个位置的无论哪种位模式都当成是变量初始的状态。当被解释成整型值时，任何位模式都是合法的值——虽然这个值不可能是程序员想要的。因为这个值合法，所以使用它也不可能导致程序崩溃。可能的结果是导致程序错误执行和/或错误计算。



We recommend that every object of built-in type be initialized. It is not always necessary to initialize such variables, but it is easier and safer to do so until you can be certain it is safe to omit an initializer.

建议每个内置类型的对象都要初始化。虽然这样做并不总是必需的，但是会更加容易和安全，除非你确定忽略初始化式不会带来风险。

Initialization of Variables of Class Type

类类型变量的初始化

Each class defines how objects of its type can be initialized. Classes control object initialization by defining one or more constructors (Section 2.3.3, p. 49). As an example, we know that the `string` class provides at least two constructors. One of these constructors lets us initialize a `string` from a character string literal and another lets us initialize a `string` from a character and a count.

Section 2.3. Variables

每个类都定义了该类型的对象可以怎样初始化。类通过定义一个或多个构造函数来控制类对象的初始化（[第 2.3.3 节](#)）。例如：我们知道 `string` 类至少提供了两个构造函数，其中一个允许我们通过字符串字面值初始化 `string` 对象，另外一个允许我们通过字符和计数器初始化 `string` 对象。

Each class may also define what happens if a variable of the type is defined but an initializer is not provided. A class does so by defining a special constructor, known as the **default constructor**. This constructor is called the default constructor because it is run "by default;" if there is no initializer, then this constructor is used. The default constructor is used regardless of where a variable is defined.

如果定义某个类的变量时没有提供初始化式，这个类也可以定义初始化时的操作。它是通过定义一个特殊的构造函数即**默认构造函数**来实现的。这个构造函数之所以被称作默认构造函数，是因为它是“默认”运行的。如果没有提供初始化式，那么就会使用默认构造函数。不管变量在哪里定义，默认构造函数都会被使用。

Most classes provide a default constructor. If the class has a default constructor, then we can define variables of that class without explicitly initializing them. For example, the `string` type defines its default constructor to initialize the `string` as an empty string that is, a string with no characters:

大多数类都提供了默认构造函数。如果类具有默认构造函数，那么就可以在定义该类的变量时不用显式地初始化变量。例如，`string` 类定义了默认构造函数来初始化 `string` 变量为空字符串，即没有字符的字符串：

```
std::string empty; // empty is the empty string; empty = ""
```

Some class types do not have a default constructor. For these types, every definition must provide explicit initializer(s). It is not possible to define variables of such types without giving an initial value.

有些类型没有默认构造函数。对于这些类型来说，每个定义都必须提供显式的初始化式。没有初始值是根本不可能定义这种类型的变量的。

Exercises Section 2.3.4

Exercise What are the initial values, if any, of each of the following variables?

2.17:

下列变量的初始值（如果有）是什么？

```
std::string global_str;
int global_int;
int main()
{
    int local_int;
    std::string local_str;
    // ...
    return 0;
}
```

2.3.5. Declarations and Definitions

2.3.5. 声明和定义

As we'll see in [Section 2.9](#) (p. 67), C++ programs typically are composed of many files. In order for multiple files to access the same variable, C++ distinguishes between declarations and definitions.

正如将在[第 2.9 节](#)所看到的那样，C++ 程序通常由许多文件组成。为了让多个文件访问相同的变量，C++ 区分了声明和定义。

A **definition** of a variable allocates storage for the variable and may also specify an initial value for the variable. There must be one and only one definition of a variable in a program.

变量的**定义**用于为变量分配存储空间，还可以为变量指定初始值。在一个程序中，变量有且仅有一个定义。

A **declaration** makes known the type and name of the variable to the program. A definition is also a declaration: When we define a variable, we declare its name and type. We can declare a name without defining it by using the `extern` keyword. A declaration that is not also a definition consists of the object's name and its type preceded by the keyword `extern`:

声明用于向程序表明变量的类型和名字。定义也是声明：当定义变量时我们声明了它的类型和名字。可以通过使用`extern`关键字声明变量名而不定义它。不定义变量的声明包括对象名、对象类型和对象类型前的关键字`extern`：

```
extern int i; // declares but does not define i
int i;         // declares and defines i
```

Section 2.3. Variables

An `extern` declaration is *not* a definition and does not allocate storage. In effect, it claims that a definition of the variable exists elsewhere in the program. A variable can be declared multiple times in a program, but it must be defined only once.

`extern` 声明不是定义，也不分配存储空间。事实上，它只是说明变量定义在程序的其他地方。程序中变量可以声明多次，但只能定义一次。

A declaration may have an initializer only if it is also a definition because only a definition allocates storage. The initializer must have storage to initialize. If an initializer is present, the declaration is treated as a definition even if the declaration is labeled `extern`:

只有当声明也是定义时，声明才可以有初始化式，因为只有定义才分配存储空间。初始化式必须要有存储空间来进行初始化。如果声明有初始化式，那么它可被当作是定义，即使声明标记为 `extern`:

```
extern double pi = 3.1416; // definition
```

Despite the use of `extern`, this statement defines `pi`. Storage is allocated and initialized. An `extern` declaration may include an initializer only if it appears outside a function.

虽然使用了 `extern`，但是这条语句还是定义了 `pi`，分配并初始化了存储空间。只有当 `extern` 声明位于函数外部时，才可以含有初始化式。

Because an `extern` that is initialized is treated as a definition, any subsequent definition of that variable is an error:

因为已初始化的 `extern` 声明被当作是定义，所以该变量任何随后的定义都是错误的：

```
extern double pi = 3.1416; // definition
double pi;                // error: redefinition of pi
```

Similarly, a subsequent `extern` declaration that has an initializer is also an error:

同样，随后的含有初始化式的 `extern` 声明也是错误的：

```
extern double pi = 3.1416; // definition
extern double pi;          // ok: declaration not definition
extern double pi = 3.1416; // error: redefinition of pi
```

The distinction between a declaration and a definition may seem pedantic but in fact is quite important.

声明和定义之间的区别可能看起来微不足道，但事实上却是举足轻重的。



In C++ a variable must be defined exactly once and must be defined or declared before it is used.

在 C++ 语言中，变量必须且仅能定义一次，而且在使用变量之前必须定义或声明变量。

Any variable that is used in more than one file requires declarations that are separate from the variable's definition. In such cases, one file will contain the definition for the variable. Other files that use that same variable will contain declarations for but not a definition of that same variable.

任何在多个文件中使用的变量都需要有与定义分离的声明。在这种情况下，一个文件含有变量的定义，使用该变量的其他文件则包含该变量的声明（而不是定义）。

Exercises Section 2.3.5

Exercise Explain the meaning of each of these instances of `name`:

2.18:

解释下列例子中 `name` 的意义

```
extern std::string name;
std::string name("exercise 3.5a");
extern std::string name("exercise 3.5a");
```

2.3.6. Scope of a Name

2.3.6. 名字的作用域

Every name in a C++ program must refer to a unique entity (such as a variable, function, type, etc.). Despite this requirement, names can be used more than once in a program: A name can be reused as long as it is used in different contexts, from which the different meanings of the name can be distinguished. The context used to distinguish the meanings of names is a `scope`. A scope is a region of the program. A name can refer to different entities in different scopes.

C++ 程序中，每个名字都与唯一的实体（比如变量、函数和类型等）相关联。尽管有这样的要求，还是可以在程序中多次使用同一个名字，只要它用在不同的上下文中，且通

Section 2.3. Variables

通过这些上下文可以区分该名字的不同意义。用来区分名字的不同意义的上下文称为作用域。作用域是程序的一段区域。一个名称可以和不同作用域中的不同实体相关联。

Most scopes in C++ are delimited by curly braces. Generally, names are visible from their point of declaration until the end the scope in which the declaration appears. As an example, consider this program, which we first encountered in [Section 1.4.2](#) (p. 14):

C++ 语言中，大多数作用域是用花括号来界定的。一般来说，名字从其声明点开始直到其声明所在的作用域结束处都是可见的。例如，思考[第 1.4.2 节](#)中的程序：

```
#include <iostream>
int main()
{
    int sum = 0;
    // sum values from 1 up to 10 inclusive
    for (int val = 1; val <= 10; ++val)
        sum += val;    // equivalent to sum = sum + val
    std::cout << "Sum of 1 to 10 inclusive is "
            << sum << std::endl;
    return 0;
}
```

This program defines three names and uses two names from the standard library. It defines a function named `main` and two variables named `sum` and `val`. The name `main` is defined outside any curly braces and is visible throughout the program. Names defined outside any function have **global scope**: they are accessible from anywhere in the program. The name `sum` is defined within the scope of the `main` function. It is accessible throughout the `main` function but not outside of it. The variable `sum` has **local scope**. The name `val` is more interesting. It is defined in the scope of the `for` statement ([Section 1.4.2](#), p. 14). It can be used in that statement but not elsewhere in `main`. It has **statement scope**.

这个程序定义了三个名字，使用了两个标准库的名字。程序定义了一个名为 `main` 的函数，以及两个名为 `sum` 和 `val` 的变量。名字 `main` 定义在所有花括号之外，在整个程序都可见。定义在所有函数外部的名字具有全局作用域，可以在程序中的任何地方访问。名字 `sum` 定义在 `main` 函数的作用域中，在整个 `main` 函数中都可以访问，但在 `main` 函数外则不能。变量 `sum` 有局部作用域。名字 `val` 更有意思，它定义在 `for` 语句的作用域中，只能在 `for` 语句中使用，而不能用在 `main` 函数的其他地方。它具有语句作用域。

Scopes in C++ Nest

C++ 中作用域可嵌套

Names defined in the global scope can be used in a local scope; global names and those defined local to a function can be used inside a statement scope, and so on. Names can also be redefined in an inner scope. Understanding what entity a name refers to requires unwinding the scopes in which the names are defined:

定义在全局作用域中的名字可以在局部作用域中使用，定义在全局作用域中的名字和定义在函数的局部作用域中的名字可以在语句作用域中使用，等等。名字还可以在内部作用域中重新定义。理解和名字相关联的实体需要明白定义名字的作用域：

```
#include <iostream>
#include <string>
/* Program for illustration purposes only:
 * It is bad style for a function to use a global variable and then
 * define a local variable with the same name
 */
std::string s1 = "hello"; // s1 has global scope
int main()
{
    std::string s2 = "world"; // s2 has local scope
    // uses global s1; prints "hello world"
    std::cout << s1 << " " << s2 << std::endl;
    int s1 = 42; // s1 is local and hides global s1
    // uses local s1; prints "42 world"
    std::cout << s1 << " " << s2 << std::endl;
    return 0;
}
```

This program defines three variables: a global `string` named `s1`, a local `string` named `s2`, and a local `int` named `s1`. The definition of the local `s1` hides the global `s1`.

这个程序中定义了三个变量：`string` 类型的全局变量 `s1`、`string` 类型的局部变量 `s2` 和 `int` 类型的局部变量 `s1`。局部变量 `s1` 的定义屏蔽了全局变量 `s1`。

Variables are visible from their point of declaration. Thus, the local definition of `s1` is not visible when the first output is performed. The name `s1` in that output expression refers to the global `s1`. The output printed is `hello world`. The second statement that does output follows the local definition of `s1`. The local `s1` is now in scope. The second output uses the local rather than the global `s1`. It writes `42 world`.

变量从声明开始才可见，因此执行第一次输出时局部变量 `s1` 不可见，输出表达式中的 `s1` 是全局变量 `s1`，输出“`hello world`”。第二条输出语句跟在 `s1` 的局部定义后，现在局部变量 `s1` 在作用域中。第二条输出语句使用的是局部变量 `s1` 而不是全局变量 `s1`，输出“`42 world`”。

Programs such as the preceding are likely to be confusing. It is almost always a bad idea to define a local variable with the same name as a global variable that the function uses or might use. It is much better to use a distinct name for the local.

像上面这样的程序很可能让人大惑不解。在函数内定义一个与函数可能会用到的全局变量同名的局部变量总是不好的。局部变量最



好使用不同的名字。

We'll have more to say about local and global scope in [Chapter 7](#) and about statement scope in [Chapter 6](#). C++ has two other levels of scope: **class scope**, which we'll cover in [Chapter 12](#) and **namespace scope**, which we'll see in [Section 17.2](#).

[第七章](#)将详细讨论局部作用域和全局作用域, [第六章](#)将讨论语句作用域。C++ 还有另外两种不同级别的作用域: 类作用域 ([第十二章](#)将介绍) 和命名空间作用域 ([第 17.2 节](#)将介绍)。

2.3.7. Define Variables Where They Are Used

2.3.7. 在变量使用处定义变量

In general, variable definitions or declarations can be placed anywhere within the program that a statement is allowed. A variable must be declared or defined before it is used.

一般来说, 变量的定义或声明可以放在程序中能摆放语句的任何位置。变量在使用前必须先声明或定义。



It is usually a good idea to define an object near the point at which the object is first used.

通常把一个对象定义在它首次使用的地方是一个很好的办法。

Defining an object where the object is first used improves readability. The reader does not have to go back to the beginning of a section of code to find the definition of a particular variable. Moreover, it is often easier to give the variable a useful initial value when the variable is defined close to where it is first used.

在对象第一次被使用的地方定义对象可以提高程序的可读性。读者不需要返回到代码段的开始位置去寻找某一特殊变量的定义, 而且, 在此处定义变量, 更容易给它赋以有意义的初始值。

One constraint on placing declarations is that variables are accessible from the point of their definition until the end of the enclosing block. A variable must be defined in or before the outermost scope in which the variable will be used.

放置声明的一个约束是, 变量只在从其定义处开始到该声明所在的作用域的结束处才可以访问。必须在使用该变量的最外层作用域里面或之前定义变量。

Exercises Section 2.3.6

Exercise

2.19: What is the value of `j` in the following program?

下列程序中 `j` 的值是多少?

```
int i = 42;
int main()
{
    int i = 100;
    int j = i;
    // ...
}
```

Exercise

2.20: Given the following program fragment, what values are printed?

下列程序段将会输出什么?

```
int i = 100, sum = 0;
for (int i = 0; i != 10; ++i)
    sum += i;
std::cout << i << " " << sum << std::endl;
```

Exercise

2.21: Is the following program legal?

下列程序合法吗?

Section 2.3. Variables

```
int sum = 0;
for (int i = 0; i != 10; ++i)
    sum += i;
std::cout << "Sum from 0 to " << i
        << " is " << sum << std::endl;
```

Team LiB

◀ PREVIOUS NEXT ▶

2.4. **const** Qualifier

2.4. **const** 限定符

There are two problems with the following `for` loop, both concerning the use of 512 as an upper bound.

下列 `for` 循环语句有两个问题，两个都和使用 512 作为循环上界有关。

```
for (int index = 0; index != 512; ++index) {
    // ...
}
```

The first problem is readability. What does it mean to compare `index` with 512? What is the loop doing that is, what makes 512 matter? (In this example, 512 is known as a [magic number](#), one whose significance is not evident within the context of its use. It is as if the number had been plucked by magic from thin air.)

第一个问题是程序的可读性。比较 `index` 与 512 有什么意思呢？循环在做什么呢？也就是说 512 作用何在？[本例中，512 被称为魔数 ([magic number](#))，它的意义在上下文中没有体现出来。好像这个数是魔术般地从空中出现的。]

The second problem is maintainability. Imagine that we have a large program in which the number 512 occurs 100 times. Let's further assume that 80 of these references use 512 to indicate the size of a particular buffer but the other 20 use 512 for different purposes. Now we discover that we need to increase the buffer size to 1024. To make this change, we must examine every one of the places that the number 512 appears. We must determine correctly in every case which of those uses of 512 refer to the buffer size and which do not. Getting even one instance wrong breaks the program and requires us to go back and reexamine each use.

第二个问题是程序的可维护性。假设这个程序非常庞大，512 出现了 100 次。进一步假设在这 100 次中，有 80 次是表示某一特殊缓冲区的大小，剩余 20 次用于其他目的。现在我们需要把缓冲区的大小增大到 1024。要实现这一改变，必须检查每个 512 出现的位置。我们必须确定（在每种情况下都准确地确定）哪些 512 表示缓冲区大小，而哪些不是。改错一个都会使程序崩溃，又得回过头来重新检查。

The solution to both problems is to use an object initialized to 512:

解决这两个问题的方法是使用一个初始化为 512 的对象：

```
int bufSize = 512;      // input buffer size
for (int index = 0; index != bufSize; ++index) {
    // ...
}
```

By choosing a mnemonic name, such as `bufSize`, we make the program more readable. The test is now against the object rather than the literal constant:

通过使用好记的名字如 `bufSize`，增强了程序的可读性。现在是对对象 `bufSize` 测试而不是字面值常量 512 测试：

```
index != bufSize
```

If we need to change this size, the 80 occurrences no longer need to be found and corrected. Rather, only the one line that initializes `bufSize` requires change. Not only does this approach require significantly less work, but also the likelihood of making a mistake is greatly reduced.

现在如果想要改变缓冲区大小，就不再需要查找和改正 80 次出现的地方。而只有初始化 `bufSize` 那行需要修改。这种方法不但明显减少了工作量，而且还大大减少了出错的可能性。

Defining a **const** Object

定义 **const** 对象

There is still a serious problem with defining a variable to represent a constant value. The problem is that `bufSize` is modifiable. It is possible for `bufSize` to be changed accidentally or otherwise. The `const` type qualifier provides a solution: It transforms an object into a constant.

定义一个变量代表某一常数的方法仍然有一个严重的问题。即 `bufSize` 是可以被修改的。`bufSize` 可能被有意或无意地修改。`const` 限定符提供了一个解决办法，它把一个对象转换成一个常量。

```
const int bufSize = 512;      // input buffer size
```

defines `bufSize` to be a constant initialized with the value 512. The variable `bufSize` is still an lvalue ([Section 2.3.1](#), p. 45), but now the lvalue is unmodifiable. Any attempt to write to `bufSize` results in a compile-time error.

定义 `bufSize` 为常量并初始化为 512。变量 `bufSize` 仍然是一个左值（[第 2.3.1 节](#)），但是现在这个左值是不可修改的。任何修改 `bufSize` 的尝试都会导致编译错误：

```
bufSize = 0; // error: attempt to write to const object
```



Because we cannot subsequently change the value of an object declared to be `const`, we must initialize it when it is defined:

因为常量在定义后就不能被修改，所以定义时必须初始化：

```
const std::string hi = "hello!"; // ok: initialized
const int i, j = 0; // error: i is uninitialized const
```

`const` Objects Are Local to a File By Default

`const` 对象默认为文件的局部变量

When we define a non`const` variable at global scope ([Section 2.3.6](#), p. 54), it is accessible throughout the program. We can define a non`const` variable in one file and assuming an appropriate declaration has been made can use that variable in another file:

在全局作用域（[第 2.3.6 节](#)）里定义非 `const` 变量时，它在整个程序中都可以访问。我们可以把一个非 `const` 变更定义在一个文件中，假设已经做了合适的声明，就可在另外的文件中使用这个变量：

```
// file_1.cc
int counter; // definition
// file_2.cc
extern int counter; // uses counter from file_1
++counter; // increments counter defined in file_1
```

Unlike other variables, unless otherwise specified, `const` variables declared at global scope are local to the file in which the object is defined. The variable exists in that file only and cannot be accessed by other files.

与其他变量不同，除非特别说明，在全局作用域声明的 `const` 变量是定义该对象的文件的局部变量。此变量只存在于那个文件中，不能被其他文件访问。

We can make a `const` object accessible throughout the program by specifying that it is `extern`:

通过指定 `const` 变更为 `extern`，就可以在整个程序中访问 `const` 对象：

```
// file_1.cc
// defines and initializes a const that is accessible to other files
extern const int bufSize = fcn();
// file_2.cc
extern const int bufSize; // uses bufSize from file_1
// uses bufSize defined in file_1
for (int index = 0; index != bufSize; ++index)
    // ...
```

In this program, `file_1.cc` defines and initializes `bufSize` to the result returned from calling the function named `fcn`. The definition of `bufSize` is `extern`, meaning that `bufSize` can be used in other files. The declaration in `file_2.cc` is also made `extern`. In this case, the `extern` signifies that `bufSize` is a declaration and hence no initializer is provided.

本程序中，`file_1.cc` 通过函数 `fcn` 的返回值来定义和初始化 `bufSize`。而 `bufSize` 定义为 `extern`，也就意味着 `bufSize` 可以在其他的文件中使用。`file_2.cc` 中 `extern` 的声明同样是 `extern`；这种情况下，`extern` 标志着 `bufSize` 是一个声明，所以没有初始化式。

We'll see in [Section 2.9.1](#) (p. 69) why `const` objects are made local to a file.

我们将会在[第 2.9.1 节](#)看到为何 `const` 对象局部于文件创建。



Non`const` variables are `extern` by default. To make a `const` variable accessible to other files we must explicitly specify that it is `extern`.

非 `const` 变量默认为 `extern`。要使 `const` 变量能够在其他的文件中访问，必须地指定它为 `extern`。

Exercises Section 2.4

Exercise 2.22: The following program fragment, while legal, is an example of poor style. What problem(s) does it contain? How would you improve it?

下种段虽然合法，但是风格很糟糕。有什么问题呢？怎样改善？

```
for (int i = 0; i < 100; ++i)
    // process i
```

Exercise 2.23: Which of the following are legal? For those usages that are illegal, explain why.

下列哪些语句合法？对于那些不合法的，请解释为什么不合法。

- (a) `const int buf;`
- (b) `int cnt = 0;`
`const int sz = cnt;`
- (c) `cnt++; sz++;`

2.5. References

2.5. 引用

A **reference** serves as an alternative name for an object. In real-world programs, references are primarily used as formal parameters to functions. We'll have more to say about reference parameters in [Section 7.2.2](#) (p. 232). In this section we introduce and illustrate the use of references as independent objects.

引用就是对象的另一个名字。在实际程序中，引用主要用作函数的形式参数。我们将在[第 7.2.2 节](#)再详细介绍引用参数。在这一节，我们用独立的对象来介绍并举例说明引用的用法。

A reference is a **compound type** that is defined by preceding a variable name by the `&` symbol. A compound type is a type that is defined in terms of another type. In the case of references, each reference type "refers to" some other type. We cannot define a reference to a reference type, but can make a reference to any other data type.

引用是一种**复合类型**，通过在变量名前添加“`&`”符号来定义。复合类型是指用其他类型定义的类型。在引用的情况下，每一种引用类型都“关联到”某一其他类型。不能定义引用类型的引用，但可以定义任何其他类型的引用。

A reference *must* be initialized using an object of the same type as the reference:

引用必须用与该引用同类型的对象初始化：

```
int ival = 1024;
int &refVal = ival; // ok: refVal refers to ival
int &refVal2;      // error: a reference must be initialized
int &refVal3 = 10; // error: initializer must be an object
```

A Reference Is an Alias

引用是别名

Because a reference is just another name for the object to which it is bound, *all* operations on a reference are actually operations on the underlying object to which the reference is bound:

因为引用只是它绑定的对象的另一名字，作用在引用上的所有操作事实上都是作用在该引用绑定的对象上：

```
refVal += 2;
```

adds 2 to `ival`, the object referred to by `refVal`. Similarly,

将 `refVal` 指向的对象 `ival` 加 2。类似地，

```
int ii = refVal;
```

assigns to `ii` the value currently associated with `ival`.

把和 `ival` 相关联的值赋给 `ii`。



When a reference is initialized, it remains bound to that object as long as the reference exists. There is no way to rebind a reference to a different object.

当引用初始化后，只要该引用存在，它就保持绑定到初始化时指向的对象。不可能将引用绑定到另一个对象。

The important concept to understand is that a reference is *just another name for an object*. Effectively, we can access `ival` either through its actual name or through its alias, `refVal`. Assignment is just another operation, so that when we write

要理解的重要概念是引用只是对象的另一名字。事实上，我们可以通过 `ival` 的原名访问 `ival`，也可以通过它的别名 `refVal` 访问。赋值只是另外一种操作，因此我们编写

```
refVal = 5;
```

the effect is to change the value of `ival` to 5. A consequence of this rule is that you must initialize a reference when you define it; initialization is the only way to say to which object a reference refers.

的效果是把 `ival` 的值修改为5。这一规则的结果是必须在定义引用时进行初始化。初始化是指明引用指向哪个对象的唯一方法。

Defining Multiple References

定义多个引用

We can define multiple references in a single type definition. Each identifier that is a reference must be preceded by the `&` symbol:

可以在一个类型定义行中定义多个引用。必须在每个引用标识符前添加“`&`”符号：

```
int i = 1024, i2 = 2048;
int &r = i, r2 = i2;      // r is a reference, r2 is an int
int i3 = 1024, &ri = i3;  // defines one object, and one reference
int &r3 = i3, &r4 = i2;   // defines two references
```

const References

const 引用

A `const reference` is a reference that may refer to a `const` object:

`const` 引用是指向 `const` 对象的引用：

```
const int ival = 1024;
const int &refVal = ival;      // ok: both reference and object are const
int &ref2 = ival;            // error: non const reference to a const object
```

We can read from but not write to `refVal`. Thus, any assignment to `refVal` is illegal. This restriction should make sense: We cannot assign directly to `ival` and so it should not be possible to use `refVal` to change `ival`.

可以读取但不能修改 `refVal`，因此，任何对 `refVal` 的赋值都是不合法的。这个限制有其意义：不能直接对 `ival` 赋值，因此不能通过使用 `refVal` 来修改 `ival`。

For the same reason, the initialization of `ref2` by `ival` is an error: `ref2` is a plain, `nonconst reference` and so could be used to change the value of the object to which `ref2` refers. Assigning to `ival` through `ref2` would result in changing the value of a `const` object. To prevent such changes, it is illegal to bind a plain reference to a `const` object.

同理，用 `ival` 初始化 `ref2` 也是不合法的：`ref2` 是普通的非 `const` 引用，因此可以用来修改 `ref2` 指向的对象的值。通过 `ref2` 对 `ival` 赋值会导致修改 `const` 对象的值。为阻止这样的修改，需要规定将普通的引用绑定到 `const` 对象是不合法的。

Terminology: const Reference is a Reference to const

术语：`const` 引用是指向 `const` 的引用

C++ programmers tend to be cavalier in their use of the term `const` reference. Strictly speaking, what is meant by "const reference" is "reference to const." Similarly, programmers use the term "nonconst reference" when speaking of reference to a nonconst type. This usage is so common that we will follow it in this book as well.

C++ 程序员常常随意地使用术语 `const` 引用。严格来说，“`const` 引用”的意思是“指向 `const` 对象的引用”。类似地，程序员使用术语“非 `const` 引用”表示指向非 `const` 类型的引用。这种用法非常普遍，我们在本书中也遵循这种用法。

A `const` reference can be initialized to an object of a different type or to an rvalue (Section 2.3.1, p. 45), such as a literal constant:

`const` 引用可以初始化为不同类型的对象或者初始化为右值（第 2.3.1 节），如字面值常量：

```
int i = 42;
// legal for const references only
const int &r = 42;
const int &r2 = r + i;
```

The same initializations are not legal for `nonconst` references. Rather, they result in compile-time errors. The reason is subtle and warrants an

Section 2.5. References

explanation.

同样的初始化对于非 `const` 引用却是不合法的，而且会导致编译时错误。其原因非常微妙，值得解释一下。

This behavior is easiest to understand when we look at what happens when we bind a reference to an object of a different type. If we write
观察将引用绑定到不同的类型时所发生的事情，最容易理解上述行为。假如我们编写

```
double dval = 3.14;  
const int &ri = dval;
```

the compiler transforms this code into something like this:

```
int temp = dval;           // create temporary int from the double  
const int &ri = temp;     // bind ri to that temporary
```

If `ri` were not `const`, then we could assign a new value to `ri`. Doing so would not change `dval` but would instead change `temp`. To the programmer expecting that assignments to `ri` would change `dval`, it would appear that the change did not work. Allowing only `const` references to be bound to values requiring temporaries avoids the problem entirely because a `const` reference is read-only.

如果 `ri` 不是 `const`，那么可以给 `ri` 赋一新值。这样做不会修改 `dval`，而是修改了 `temp`。期望对 `ri` 的赋值会修改 `dval` 的程序员会发现 `dval` 并没有被修改。仅允许 `const` 引用绑定到需要临时使用的值完全避免了这个问题，因为 `const` 引用是只读的。



A non`const` reference may be attached only to an object of the same type as the reference itself.

非 `const` 引用只能绑定到与该引用同类型的对象。

A `const` reference may be bound to an object of a different but related type or to an rvalue.

`const` 引用则可以绑定到不同但相关的类型的对象或绑定到右值。

Exercises Section 2.5

Exercise

2.24: Which of the following definitions, if any, are invalid? Why? How would you correct them?

下列哪些定义是非法的？为什么？如何改正？

- (a) `int ival = 1.01;`
- (b) `int &rval1 = 1.01;`
- (c) `int &rval2 = ival;`
- (d) `const int &rval3 = 1;`

Exercise

2.25: Given the preceding definitions, which, if any, of the following assignments are invalid? If they are valid, explain what they do.

在上题给出的定义下，下列哪些赋值是非法的？如果赋值合法，解释赋值的作用。

- (a) `rval2 = 3.14159;`
- (b) `rval2 = rval3;`
- (c) `ival = rval3;`
- (d) `rval3 = ival;`

Exercise

2.26: What are the differences among the definitions in (a) and the assignments in (b)? Which, if any, are illegal?

(a) 中的定义和 (b) 中的赋值存在哪些不同？哪些是非法的？

- (a) `int ival = 0;`
- (b) `ival = ri;`
- `const int &ri = 0;`
- `ri = ival;`

Exercise

2.27: What does the following code print?

下列代码输出什么？

```
int i, &ri = i;  
i = 5; ri = 10;  
std::cout << i << " " << ri << std::endl;
```

Team LiB

◀ PREVIOUS NEXT ▶

2.6. Typedef Names

2.6. `typedef` 名字

A **`typedef`** lets us define a synonym for a type:

`typedef` 可以用来定义类型的同义词:

```
typedef double wages;      // wages is a synonym for double
typedef int exam_score;    // exam_score is a synonym for int
typedef wages salary;     // indirect synonym for double
```

A `typedef` name can be used as a type specifier:

`typedef` 名字可以用作类型说明符:

```
wages hourly, weekly;   // double hourly, weekly;
exam_score test_result; // int test_result;
```

A `typedef` definition begins with the keyword **`typedef`**, followed by the data type and identifier. The identifier, or `typedef` name, does not introduce a new type but rather a synonym for the existing data type. A `typedef` name can appear anywhere in a program that a type name can appear.

`typedef` 定义以关键字 **`typedef`** 开始，后面是数据类型和标识符。标识符或类型名并没有引入新的类型，而只是现有数据类型的同义词。**`typedef`** 名字可出现在程序中类型名可出现的任何位置。

Typedefs are commonly used for one of three purposes:

`typedef` 通常被用于以下三种目的:

- To hide the implementation of a given type and emphasize instead the purpose for which the type is used
为了隐藏特定类型的实现，强调使用类型的目的。
- To streamline complex type definitions, making them easier to understand
简化复杂的类型定义，使其更易理解。
- To allow a single type to be used for more than one purpose while making the purpose clear each time the type is used
允许一种类型用于多个目的，同时使得每次使用该类型的目的明确。

2.7. Enumerations

2.7. 枚举

Often we need to define a set of alternative values for some attribute. A file, for example, might be open in one of three states: input, output, and append. One way to keep track of these state values might be to associate a unique constant number with each. We might write the following:

我们经常需要为某些属性定义一组可选择的值。例如，文件打开的状态可能会有三种：输入、输出和追加。记录这些状态值的一种方法是使每种状态都与一个唯一的常数值相关联。我们可能会这样编写代码：

```
const int input = 0;
const int output = 1;
const int append = 2;
```

Although this approach works, it has a significant weakness: There is no indication that these values are related in any way. [Enumerations](#) provide an alternative method of not only defining but also grouping sets of integral constants.

虽然这种方法也能奏效，但是它有个明显的缺点：没有指出这些值是相关联的。[枚举](#)提供了一种替代的方法，不但定义了整数常量集，而且还把它们聚集成组。

Defining and Initializing Enumerations

定义和初始化枚举

An enumeration is defined using the `enum` keyword, followed by an optional enumeration name, and a comma-separated list of [enumerators](#) enclosed in braces.

枚举的定义包括关键字 `enum`，其后是一个可选的枚举类型名，和一个用花括号括起来、用逗号分开的[枚举成员](#)列表。

```
// input is 0, output is 1, and append is 2
enum open_modes {input, output, append};
```

By default, the first enumerator is assigned the value zero. Each subsequent enumerator is assigned a value one greater than the value of the enumerator that immediately precedes it.

默认地，第一个枚举成员赋值为 0，后面的每个枚举成员赋的值比前面的大 1。

Enumerators Are `const` Values

枚举成员是常量

We may supply an initial value for one or more enumerators. The value used to initialize an enumerator must be a [constant expression](#). A constant expression is an expression of integral type that the compiler can evaluate at compile time. An integral literal constant is a constant expression, as is a `const` object ([Section 2.4](#), p. 56) that is itself initialized from a constant expression.

可以为一个或多个枚举成员提供初始值，用来初始化枚举成员的值必须是一个[常量表达式](#)。常量表达式是编译器在编译时就能够计算出结果的整型表达式。整型字面值常量是常量表达式，正如一个通过常量表达式自我初始化的 `const` 对象（[第 2.4 节](#)）也是常量表达式一样。

For example, we might define the following enumeration:

例如，可以定义下列枚举类型：

```
// shape is 1, sphere is 2, cylinder is 3, polygon is 4
enum Forms {shape = 1, sphere, cylinder, polygon};
```

In the `enum Forms` we explicitly assigned `shape` the value 1. The other enumerators are implicitly initialized: `sphere` is initialized to 2, `cylinder` to 3, and `polygon` to 4.

在 `枚举类型 Forms` 中，显式将 `shape` 赋值为 1。其他枚举成员隐式初始化：`sphere` 初始化为 2，`cylinder` 初始化为 3，`polygon` 初始化为 4。

Section 2.7. Enumerations

An enumerator value need not be unique.

枚举成员值可以是不唯一的。

```
// point2d is 2, point2w is 3, point3d is 3, point3w is 4
enum Points { point2d = 2, point2w,
               point3d = 3, point3w };
```

In this example, the enumerator `point2d` is explicitly initialized to `2`. The next enumerator, `point2w`, is initialized by default, meaning that its value is one more than the value of the previous enumerator. Thus, `point2w` is initialized to `3`. The enumerator `point3d` is explicitly initialized to `3`, and `point3w`, again is initialized by default, in this case to `4`.

本例中，枚举成员 `point2d` 显式初始化为 `2`。下一个枚举成员 `point2w` 默认初始化，即它的值比前一枚举成员的值大 `1`。因此 `point2w` 初始化为 `3`。枚举成员 `point3d` 显式初始化为 `3`。一样，`point3w` 默认初始化，结果为 `4`。

It is not possible to change the value of an enumerator. As a consequence an enumerator is itself a constant expression and so can be used where a constant expression is required.

不能改变枚举成员的值。枚举成员本身就是一个常量表达式，所以也可用于需要常量表达式的任何地方。

Each `enum` Defines a Unique Type

每个 `enum` 都定义一种唯一的类型

Each `enum` defines a new type. As with any type, we can define and initialize objects of type `Points` and can use those objects in various ways. An object of enumeration type may be initialized or assigned only by one of its enumerators or by another object of the same enumeration type:

每个 `enum` 都定义了一种新的类型。和其他类型一样，可以定义和初始化 `Points` 类型的对象，也可以以不同的方式使用这些对象。枚举类型的对象的初始化或赋值，只能通过其枚举成员或同一枚举类型的其他对象来进行：

```
Points pt3d = point3d; // ok: point3d is a Points enumerator
Points pt2w = 3;         // error: pt2w initialized with int
pt2w = polygon;        // error: polygon is not a Points enumerator
pt2w = pt3d;            // ok: both are objects of Points enum type
```

Note that it is illegal to assign the value `3` to a `Points` object even though `3` is a value associated with one of the `Points` enumerators.

注意把 `3` 赋给 `Points` 对象是非法的，即使 `3` 与一个 `Points` 枚举成员相关联。

2.8. Class Types

2.8. 类类型

In C++ we define our own data types by defining a **class**. A class defines the data that an object of its type contains and the operations that can be executed by objects of that type. The library types `string`, `istream`, and `ostream` are all defined as classes.

C++ 中，通过定义类来自定义数据类型。类定义了该类型的对象包含的数据和该类型的对象可以执行的操作。标准库类型`string`、`istream`和`ostream`都定义成类。

C++ support for classes is extensive in fact, defining classes is so important that we shall devote [Parts III](#) through [V](#) to describing C++ support for classes and operations using class types.

C++ 对类的支持非常丰富——事实上，定义类是如此重要，我们把[第三到第五部分](#)全部用来描述 C++ 对类及类操作的支持。

In [Chapter 1](#) we used the `Sales_item` type to solve our bookstore problem. We used objects of type `Sales_item` to keep track of sales data associated with a particular ISBN. In this section, we'll take a first look at how a simple class, such as `Sales_item`, might be defined.

在[第一章](#)中，我们使用 `Sales_item` 类型来解决书店问题。使用 `Sales_item` 类型的对象来记录对应于特定 ISBN 的销售数据。在这节中，我们先了解如何定义简单的类，如 `Sales_item` 类。

Class Design Starts with the Operations

从操作开始设计类

Each class defines an **interface** and **implementation**. The interface consists of the operations that we expect code that uses the class to execute. The implementation typically includes the data needed by the class. The implementation also includes any functions needed to define the class but that are not intended for general use.

每个类都定义了一个接口和一个实现。接口由使用该类的代码需要执行的操作组成。实现一般包括该类所需要的数据。实现还包括定义该类需要的但又不供一般性使用的函数。

When we define a class, we usually begin by defining its interface—the operations that the class will provide. From those operations we can then determine what data the class will require to accomplish its tasks and whether it will need to define any functions to support the implementation.

定义类时，通常先定义该类的接口，即该类所提供的操作。通过这些操作，可以决定该类完成其功能所需要的数据，以及是否需要定义一些函数来支持该类的实现。

The operations our type will support are the operations we used in [Chapter 1](#). These operations were outlined in [Section 1.5.1](#) (p. 21):

我们将要定义的类型所支持的操作，就是我们在[第一章](#)中所用到的操作。这些操作如下（参见[第 1.5.1 节](#)）：

- The addition operator to add two `Sales_items`
加法操作符，将两个 `Sales_item` 相加。
- The input and output operators to read and write `Sales_item` objects
输入和输出操作符，读和写 `Sales_item` 对象。
- The assignment operator to assign one `Sales_item` object to another
赋值操作符，把 `Sales_item` 对象赋给另一个 `Sales_item` 对象。
- The `same_isbn` function to determine if two objects refer to the same book
`same_isbn` 函数，检测两个对象是否指同一本书。

We'll see how to define these operations in [Chapters 7](#) and [14](#) after we learn how to define functions and operators. Even though we can't yet implement these functions, we can figure out what data they'll need by thinking a bit about what these operations must do. Our `Sales_item` class must

在学完怎样定义函数和操作符后，我们将会在[第七章](#)和[第十四章](#)看到该怎样来定义这些操作。虽然现在不能实现这些函数，但通过思考这些操作必须要实现的功能，我们可以看出该类需要什么样的数据。`Sales_item` 类必须

Section 2.8. Class Types

1. Keep track of how many copies of a particular book were sold

记录特定书的销售册数。

2. Report the total revenue for that book

记录该书的总销售收入。

3. Calculate the average sales price for that book

计算该书的平均售价。

Looking at this list of tasks, we can see that we'll need an `unsigned` to keep track of how many books are sold and a `double` to keep track of the total revenue. From these data we can calculate the average sales price as total revenue divided by number sold. Because we also want to know which book we're reporting on, we'll also need a `string` to keep track of the ISBN.

查看以上所列出的任务，可以知道需要一个 `unsigned` 类型的对象来记录书的销售册数，一个 `double` 类型的对象来记录总销售收入，然后可以用总收入除以销售册数计算出平均售价。因为我们还想知道是在记录哪本书，所以还需要定义一个 `string` 类型的对象来记录书的 ISBN。

Defining the `Sales_item` Class

定义 `Sales_item` 类

Evidently what we need is the ability to define a data type that will have these three data elements and the operations we used in [Chapter 1](#). In C++, the way we define such a data type is to define a class:

很明显，我们需要能够定义一种包含这三个数据元素和在[第一章](#)所用到的操作的数据类型。在 C++ 语言中，定义这种数据类型的方法就是定义类：

```
class Sales_item {
public:
    // operations on Sales_item objects will go here
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};
```

A class definition starts with the keyword `class` followed by an identifier that names the class. The body of the class appears inside curly braces. The close curly must be followed by a semicolon.

类定义以关键字 `class` 开始，其后是该类的名字标识符。类体位于花括号里面。花括号后面必须要跟一个分号。



It is a common mistake among new programmers to forget the semicolon at the end of a class definition.

编程新手经常会忘记类定义后面的分号，这是个很普遍的错误！

The class body, which can be empty, defines the data and operations that make up the type. The operations and data that are part of a class are referred to as its **members**. The operations are referred to as the member functions ([Section 1.5.2](#), p. 24) and the data as **data members**.

类体可以为空。类体定义了组成该类型的数据和操作。这些操作和数据是类的一部分，也称为类的成员。操作称为成员函数（[第 1.5.2 节](#)），而数据则称为**数据成员**。

The class also may contain zero or more `public` or `private` **access labels**. An access label controls whether a member is accessible outside the class. Code that uses the class may access only the `public` members.

类也可以包含 0 个到多个 `private` 或 `public` 访问标号。访问标号控制类的成员在类外部是否可访问。使用该类的代码可能只能访问 `public` 成员。

When we define a class, we define a new type. The class name is the name of that type. By naming our class `Sales_item` we are saying that `Sales_item` is a new type and that programs may define variables of this type.

定义了类，也就定义了一种新的类型。类名就是该类型的名字。通过命名 `Sales_item` 类，表示 `Sales_item` 是一种新的类型，而且程序也可以定义该类型的变量。

Each class defines its own scope ([Section 2.3.6](#), p. 54). That is, the names given to the data and operations inside the class body must be unique within the class but can reuse names defined outside the class.

每一个类都定义了它自己的作用域（[第 2.3.6 节](#)）。也就是说，数据和操作的名字在类的内部必须唯一，但可以重用定义在类外的名字。

Class Data Members

类的数据成员

The data [members of a class](#) are defined in somewhat the same way that normal variables are defined. We specify a type and give the member a name just as we do when defining a simple variable:

定义类的数据成员和定义普通变量有些相似。我们同样是指定一种类型并给该成员一个名字：

```
std::string isbn;
unsigned units_sold;
double revenue;
```

Our class has three data members: a member of type `string` named `isbn`, an `unsigned` member named `units_sold`, and a member of type `double` named `revenue`. The data members of a class define the contents of the objects of that class type. When we define objects of type `Sales_item`, those objects will contain a `string`, an `unsigned`, and a `double`.

这个类含有三个数据成员：一个名为 `isbn` 的 `string` 类型成员，一个名为 `units_sold` 的 `unsigned` 类型成员，一个名为 `revenue` 的 `double` 类型成员。类的数据成员定义了该类类型对象的内容。当定义 `Sales_item` 类型的对象时，这些对象将包含一个 `string` 型变量，一个 `unsigned` 型变量和一个 `double` 型变量。

There is one crucially important difference between how we define variables and class data members: We ordinarily cannot initialize the members of a class as part of their definition. When we define the data members, we can only name them and say what types they have. Rather than initializing data members when they are defined inside the class definition, classes control initialization through special member functions called constructors ([Section 2.3.3](#), p. 49). We will define the `Sales_item` constructors in [Section 7.7.3](#) (p. 262).

定义变量和定义数据成员存在非常重要的区别：一般不能把类成员的初始化作为其定义的一部分。当定义数据成员时，只能指定该数据成员的名字和类型。类不是在类定义里定义数据成员时初始化数据成员，而是通过称为构造函数（[第 2.3.3 节](#)）的特殊成员函数控制初始化。我们将在[第 7.7.3 节](#)定义 `Sales_item` 的构造函数。

Access Labels

访问标号

Access labels control whether code that uses the class may use a given member. Member functions of the class may use any member of their own class, regardless of the access level. The access labels, `public` and `private`, may appear multiple times in a class definition. A given label applies until the next access label is seen.

访问标号负责控制使用该类的代码是否可以使用给定的成员。类的成员函数可以使用类的任何成员，而不管其访问级别。访问标号 `public`、`private` 可以多次出现在类定义中。给定的访问标号应用到下一个访问标号出现时为止。

The `public` section of a class defines members that can be accessed by any part of the program. Ordinarily we put the operations in the `public` section so that any code in the program may execute these operations.

类中 `public` 部分定义的成员在程序的任何部分都可以访问。一般把操作放在 `public` 部分，这样程序的任何代码都可以执行这些操作。

Code that is not part of the class does not have access to the `private` members. By making the `Sales_item` data members `private`, we ensure that code that operates on `Sales_item` objects cannot directly manipulate the data members. Programs, such as the one we wrote in [Chapter 1](#), may not access the `private` members of the class. Objects of type `Sales_item` may execute the operations but not change the data directly.

不是类的组成部分的代码不能访问 `private` 成员。通过设定 `Sales_item` 的数据成员为 `private`，可以保证对 `Sales_item` 对象进行操作的代码不能直接操纵其数据成员。就像我们在[第一章](#)编写的程序那样，程序不能访问类中的 `private` 成员。`Sales_item` 类型的对象可以执行那些操作，但是不能直接修改这些数据。

Using the `struct` Keyword

使用 `struct` 关键字

C++ supports a second keyword, `struct`, that can be used to define class types. The `struct` keyword is inherited from C.

C++ 支持另一个关键字 `struct`，它也可以定义类类型。`struct` 关键字是从 C 语言中继承过来的。

If we define a class using the `class` keyword, then any members defined before the first access label are implicitly `private`; if we use the `struct` keyword, then those members are `public`. Whether we define a class using the `class` keyword or the `struct` keyword affects only the default initial access level.

如果使用 `class` 关键字来定义类，那么定义在第一个访问标号前的任何成员都隐式指定为 `private`；如果使用 `struct` 关键字，那么这些成员都是 `public`。使用 `class` 还是 `struct` 关键字来定义类，仅仅影响默认的初始访问级别。

Section 2.8. Class Types

We could have defined our `sales_item` equivalently by writing

可以等效地定义 `Sales_item` 类为:

```
struct Sales_item {  
    // no need for public label, members are public by default  
    // operations on Sales_item objects  
private:  
    std::string isbn;  
    unsigned units_sold;  
    double revenue;  
};
```

There are only two differences between this class definition and our initial class definition: Here we use the `struct` keyword, and we eliminate the use of `public` keyword immediately following the opening curly brace. Members of a `struct` are public, unless otherwise specified, so there is no need for the `public` label.

本例的类定义和前面的类定义只有两个区别: 这里使用了关键字 `struct`, 并且没有在花括号后使用关键字 `public`。`struct` 的成员都是 `public`, 除非有其他特殊的声明, 所以就没有必要添加 `public` 标号。



The only difference between a class defined with the `class` keyword or the `struct` keyword is the default access level: By default, members in a `struct` are `public`; those in a `class` are `private`.

用 `class` 和 `struct` 关键字定义类的唯一差别在于默认访问级别: 默认情况下, `struct` 的成员为 `public`, 而 `class` 的成员为 `private`。

Exercises Section 2.8

Exercise 2.28: Compile the following program to determine whether your compiler warns about a missing semicolon after a class definition:

编译以下程序, 确定你的编译器是否会警告遗漏了类定义后面的分号。

```
class Foo {  
    // empty  
} // Note: no semicolon  
int main()  
{  
    return 0;  
}
```

If the diagnostic is confusing, remember the message for future reference.

如果编译器的诊断结果难以理解, 记住这些信息以备后用。

Exercise 2.29: Distinguish between the `public` and `private` sections of a class.

区分类中的 `public` 部分和 `private` 部分。

Exercise 2.30: Define the data members of classes to represent the following types:

定义表示下列类型的类的数据成员:

- (a) a phone number
- (b) an address
- (c) an employee or a company
- (d) a student at a university

2.9. Writing Our Own Header Files

2.9. 编写自己的头文件

We know from [Section 1.5](#)(p. 20)that ordinarily class definitions go into a **header file**. In this section we'll see how to define a header file for the `Sales_item` class.

我们已经从[第 1.5 节](#)了解到，一般类定义都会放入**头文件**。在本节中我们将看到怎样为 `Sales_item` 类定义头文件。

In fact, C++ programs use headers to contain more than class definitions. Recall that every name must be declared or defined before it is used. The programs we've written so far handle this requirement by putting all their code into a single file. As long as each entity precedes the code that uses it, this strategy works. However, few programs are so simple that they can be written in a single file. Programs made up of multiple files need a way to link the use of a name and its declaration. In C++ that is done through header files.

事实上，C++ 程序使用头文件包含的不仅仅是类定义。回想一下，名字在使用前必须先声明或定义。到目前为止，我们编写的程序是把代码放到一个文件里来处理这个要求。只要每个实体位于使用它的代码之前，这个策略就有效。但是，很少有程序简单到可以放置在一个文件中。由多个文件组成的程序需要一种方法连接名字的使用和声明，在 C++ 中这是通过头文件实现的。

To allow programs to be broken up into logical parts, C++ supports what is commonly known as **separate compilation**. Separate compilation lets us compose a program from several files. To support separate compilation, we'll put the definition of `Sales_item` in a header file. The member functions for `Sales_item`, which we'll define in [Section 7.7](#) (p. 258), will go in a separate source file. Functions such as `main` that use `Sales_item` objects are in other source files. Each of the source files that use `Sales_item` must include our `Sales_item.h` header file.

为了允许把程序分成独立的逻辑块，C++ 支持所谓的**分别编译**。这样程序可以由多个文件组成。为了支持分别编译，我们把 `Sales_item` 的定义放在一个头文件里面。我们后面在[第 7.7 节](#)中定义的 `Sales_item` 成员函数将放在单独的源文件中。像 `main` 这样使用 `Sales_item` 对象的函数放在其他的源文件中，任何使用 `Sales_item` 的源文件都必须包含 `Sales_item.h` 头文件。

2.9.1. Designing Our Own Headers

2.9.1. 设计自己的头文件

A header provides a centralized location for related declarations. Headers normally contain class definitions, `extern` variable declarations, and function declarations, about which we'll learn in [Section 7.4](#) (p. 251). Files that use or define these entities include the appropriate header(s).

头文件为相关声明提供了一个集中存放的位置。头文件一般包含类的定义、`extern` 变量的声明和函数的声明。函数的声明将在[第 7.4 节](#)介绍。使用或定义这些实体的文件要包含适当的头文件。

Proper use of header files can provide two benefits: All files are guaranteed to use the same declaration for a given entity; and should a declaration require change, only the header needs to be updated.

头文件的正确使用能够带来两个好处：保证所有文件使用给定实体的同一声明；当声明需要修改时，只有头文件需要更新。

Some care should be taken in designing headers. The declarations in a header should logically belong together. A header takes time to compile. If it is too large programmers may be reluctant to incur the compile-time cost of including it.

设计头文件还需要注意以下几点：头文件中的声明在逻辑上应该是统一的。编译头文件需要一定的时间。如果头文件太大，程序员可能不愿意承受包含该头文件所带来的编译时间代价。



To reduce the compile time needed to process headers, some C++ implementations support precompiled header files. For more details, consult the reference manual of your C++ implementation.

为了减少处理头文件的编译时间，有些 C++ 的实现支持预编译头文件。欲进一步了解详细情况，请参考你的 C++ 实现的手册。

Compiling and Linking Multiple Source Files

编译和链接多个源文件

To produce an executable file, we must tell the compiler not only where to find our `main` function but also where to find

Section 2.9. Writing Our Own Header Files

the definition of the member functions defined by the `Sales_item` class. Let's assume that we have two files: `main.cc`, which contains the definition of `main`, and `Sales_item.cc`, which contains the `Sales_item` member functions. We might compile these files as follows:

要产生可执行文件，我们不但要告诉编译器到哪里去查找 `main` 函数，而且还要告诉编译器到哪里去查找 `Sales_item` 类所定义的成员函数的定义。假设我们有两个文件：`main.cc` 含有 `main` 函数的定义，`Sales_item.cc` 含有 `Sales_item` 的成员函数。我们可以按以下方式编译这两个文件：

```
$ CC -c main.cc Sales_item.cc # by default generates a.exe  
# some compilers generate a.out  
  
# puts the executable in main.exe  
$ CC -c main.cc Sales_item.cc -o main
```

where `$` is our system prompt and `#` begins a command-line comment. We can now run the executable file, which will run our `main` program.

其中 `$` 是我们的系统提示符，`#` 开始命令行注释。现在我们可以运行可执行文件，它将运行我们的 `main` 程序。

If we have only changed one of our `.cc` source files, it is more efficient to recompile only the file that actually changed. Most compilers provide a way to separately compile each file. This process usually yields a `.o` file, where the `.o` extension implies that the file contains object code.

如果我们只是修改了一个 `.cc` 源文件，较有效的方法是只重新编译修改过的文件。大多数编译器都提供了分别编译每一个文件的方法。通常这个过程产生 `.o` 文件，`.o` 扩展名暗示该文件含有目标代码。

The compiler lets us link object files together to form an executable. On the system we use, in which the compiler is invoked by a command named `cc`, we would compile our program as follows:

编译器允许我们把目标文件链接在一起以形成可执行文件。我们所使用的系统可以通过命令名 `cc` 调用编译。因此可以按以下方式编译程序：

```
$ CC -c main.cc          # generates main.o  
$ CC -c Sales_item.cc    # generates Sales_item.o  
$ CC main.o Sales_item.o # by default generates a.exe;  
# some compilers generate a.out  
  
# puts the executable in main.exe  
$ CC main.o Sales_item.o -o main
```

You'll need to check with your compiler's user's guide to understand how to compile and execute programs made up of multiple source files.

你需要检查所用编译器的用户手册，了解如何编译和执行由多个源文件组成的程序。



Many compilers offer an option to enhance the error detection of the compiler. Check your compiler's user's guide to see what additional checks are available.

许多编译器提供了增强其错误检测能力的选项。查看所用编译器的用户指南，了解有哪些额外的检测方法。

Headers Are for Declarations, Not Definitions

头文件用于声明而不是用于定义

When designing a header it is essential to remember the difference between definitions, which may only occur once, and declarations, which may occur multiple times ([Section 2.3.5](#), p. 52). The following statements are definitions and therefore should not appear in a header:

当设计头文件时，记住定义和声明的区别是很重要的。定义只可以出现一次，而声明则可以出现多次 ([第 2.3.5 节](#))。下列语句是一些定义，所以不应该放在头文件里：

```
extern int ival = 10;      // initializer, so it's a definition  
double fica_rate;         // no extern, so it's a definition
```

Although `ival` is declared `extern`, it has an initializer, which means this statement is a definition. Similarly, the declaration of `fica_rate`, although it does not have an initializer, is a definition because the `extern` keyword is absent. Including either of these definitions in two or more files of the same program will result in a linker error complaining about multiple definitions.

虽然 `ival` 声明为 `extern`，但是它有初始化式，代表这条语句是一个定义。类似地，`fica_rate` 的声明虽然没有初始化式，但也是一个定义，因为没有关键字 `extern`。同一个程序中有两个以上文件含有上述任一个定义都会导致多重定义链接错误。

Because headers are included in multiple source files, they should not contain definitions of variables or functions.

因为头文件包含在多个源文件中，所以不应该含有变量或函数的定义。



There are three exceptions to the rule that headers should not contain definitions: classes, `const` objects whose value is known at compile time, and `inline` functions ([Section 7.6](#) (p. 256) covers `inline` functions) are all defined in headers. These entities may be defined in more than one source file as long as the definitions in each file are exactly the same.

对于头文件不应该含有定义这一规则，有三个例外。头文件可以定义类、值在编译时就已知道的 `const` 对象和 `inline` 函数（[第 7.6 节](#)介绍 `inline` 函数）。这些实体可在多个源文件中定义，只要每个源文件中的定义是相同的。

These entities are defined in headers because the compiler needs their definitions (not just declarations) to generate code. For example, to generate code that defines or uses objects of a class type, the compiler needs to know what data members make up that type. It also needs to know what operations can be performed on these objects. The class definition provides the needed information. That `const` objects are defined in a header may require a bit more explanation.

在头文件中定义这些实体，是因为编译器需要它们的定义（不只是声明）来产生代码。例如：为了产生能定义或使用类的对象的代码，编译器需要知道组成该类型的数据成员。同样还需要知道能够在这些对象上执行的操作。类定义提供所需要的信息。在头文件中定义 `const` 对象则需要更多的解释。

Some `const` Objects Are Defined in Headers

一些 `const` 对象定义在头文件中

Recall that by default a `const` variable ([Section 2.4](#), p. 57) is local to the file in which it is defined. As we shall now see, the reason for this default is to allow `const` variables to be defined in header files.

回想一下，`const` 变量（[第 2.4 节](#)）默认时是定义该变量的文件的局部变量。正如我们现在所看到的，这样设置默认情况的原因在于允许 `const` 变量定义在头文件中。

In C++ there are places where constant expression ([Section 2.7](#), p. 62) is required. For example, the initializer of an enumerator must be a constant expression. We'll see other cases that require constant expressions in later chapters.

在 C++ 中，有些地方需要放置常量表达式（[第 2.7 节](#)）。例如，枚举成员的初始化式必须是常量表达式。在以后的章节中将会看到其他需要常量表达式的例子。

Generally speaking, a constant expression is an expression that the compiler can evaluate at compile-time. A `const` variable of integral type may be a constant expression when it is itself initialized from a constant expression. However, for the `const` to be a constant expression, the initializer must be visible to the compiler. To allow multiple files to use the same constant value, the `const` and its initializer must be visible in each file. To make the initializer visible, we normally define such `const`s inside a header file. That way the compiler can see the initializer whenever the `const` is used.

一般来说，常量表达式是编译器在编译时就能够计算出结果的表达式。当 `const` 整型变量通过常量表达式自我初始化时，这个 `const` 整型变量就可能是常量表达式。而 `const` 变量要成为常量表达式，初始化式必须为编译器可见。为了能够让多个文件使用相同的常量值，`const` 变量和它的初始化式必须是每个文件都可见的。而要使初始化式可见，一般都把这样的 `const` 变量定义在头文件中。那样的话，无论该 `const` 变量何时使用，编译器都能够看见其初始化式。

However, there can be only one definition ([Section 2.3.5](#), p. 52) for any variable in a C++ program. A definition allocates storage; all uses of the variable must refer to the same storage. Because, by default, `const` objects are local to the file in which they are defined, it is legal to put their definition in a header file.

但是，C++ 中的任何变量都只能定义一次（[第 2.3.5 节](#)）。定义会分配存储空间，而所有对该变量的使用都关联到同一存储空间。因为 `const` 对象默认为定义它的文件的局部变量，所以把它们的定义放在头文件中是合法的。

There is one important implication of this behavior. When we define a `const` in a header file, every source file that includes that header has its own `const` variable with the same name and value.

这种行为有一个很重要的含义：当我们在头文件中定义了 `const` 变量后，每个包含该头文件的源文件都有了自己的 `const` 变量，其名称和值都一样。

When the `const` is initialized by a constant expression, then we are guaranteed that all the variables will have the same value. Moreover, in practice, most compilers will replace any use of such `const` variables by their corresponding constant expression at compile time. So, in practice, there won't be any storage used to hold `const` variables that are initialized by constant expressions.

当该 `const` 变量是用常量表达式初始化时，可以保证所有的变量都有相同的值。但是在实践中，大部分的编译器在编译时都会用相应的常量表达式替换这些 `const` 变量的任何使用。所以，在实践中不会有存储空间用于存储用常量表达式初始化的 `const` 变量。

When a `const` is initialized by a value that is not a constant expression, then it should not be defined in header file. Instead, as with any other variable, the `const` should be defined and initialized in a source file. An `extern` declaration for that `const` should be made in the header, enabling multiple files to share that variable.

如果 `const` 变量不是用常量表达式初始化，那么它就不应该在头文件中定义。相反，和其他的变量一样，该 `const` 变量应该在一个源文件中定义并初始化。应在头文件中为它

Section 2.9. Writing Our Own Header Files

添加 `extern` 声明，以使其能被多个文件共享。

Exercises Section 2.9.1

Exercise Identify which of the following statements are declarations and which ones are definitions.
2.31: Explain why they are declarations or definitions.

判别下列语句哪些是声明，哪些是定义，请解释原因。

- (a) `extern int ix = 1024;`
- (b) `int iy;`
- (c) `extern int iz;`
- (d) `extern const int &ri;`

Exercise Which of the following declarations and definitions would you put in a header? In a source file?
2.32: Explain why.

下列声明和定义哪些应该放在头文件中？哪些应该放在源文件中？请解释原因。

- (a) `int var;`
- (b) `const double pi = 3.1416;`
- (c) `extern int total = 255;`
- (d) `const double sq2 = sqrt(2.0);`

Exercise Determine what options your compiler offers for increasing the warning level. Recompile selected earlier programs using this option to see whether additional problems are reported.
2.33:

确定你的编译器提供了哪些提高警告级别的选项。使用这些选项重新编译以前选择的程序，查看是否会报告新的问题。

2.9.2. A Brief Introduction to the Preprocessor

2.9.2. 预处理器的简单介绍

Now that we know what we want to put in our headers, our next problem is to actually write a header. We know that to use a header we have to `#include` it in our source file. In order to write our own headers, we need to understand a bit more about how a `#include` directive works. The `#include` facility is a part of the C++ [preprocessor](#). The preprocessor manipulates the source text of our programs and runs before the compiler. C++ inherits a fairly elaborate preprocessor from C. Modern C++ programs use the preprocessor in a very restricted fashion.

既然已经知道了什么应该放在头文件中，那么我们下一个问题是真正地编写头文件。我们知道要使用头文件，必须在源文件中`#include`该头文件。为了编写头文件，我们需要进一步理解 `#include` 指示是怎样工作的。`#include` 设施是C++ [预处理器](#)的一部分。预处理器处理程序的源代码，在编译器之前运行。C++ 继承了 C 的非常精细的预处理器。现在的 C++ 程序以高度受限的方式使用预处理器。

A `#include` directive takes a single argument: the name of a header. The pre-processor replaces each `#include` by the contents of the specified header. Our own headers are stored in files. System headers may be stored in a compiler-specific format that is more efficient. Regardless of the form in which a header is stored, it ordinarily contains class definitions and declarations of the variables and functions needed to support separate compilation.

`#include` 指示只接受一个参数：头文件名。预处理器用指定的头文件的内容替代每个 `#include`。我们自己的头文件存储在文件中。系统的头文件可能用特定于编译器的更高效的格式保存。无论头文件以何种格式保存，一般都含有支持分别编译所需的类定义及变量和函数的声明。

Headers Often Need Other Headers

头文件经常需要其他头文件

Headers often `#include` other headers. The entities that a header defines often use facilities from other headers. For example, the header that defines our `Sales_item` class must include the `string` library. The `Sales_item` class has a `string` data member and so must have access to the `string` header.

头文件经常 `#include` 其他头文件。头文件定义的实体经常使用其他头文件的设施。例如，定义 `Sales_item` 类的头文件必须包含 `string` 库。`Sales_item` 类含有一个

Section 2.9. Writing Our Own Header Files

`string` 类型的数据成员，因此必须可以访问 `string` 头文件。

Including other headers is so common that it is not unusual for a header to be included more than once in the same source file. For example, a program that used the `Sales_item` header might also use the `string` library. That program wouldn't know that our `Sales_item` header uses the `string` library. In this case, the `string` header would be included twice: once by the program itself and once as a side-effect of including our `Sales_item` header.

包含其他头文件是如此司空见惯，甚至一个头文件被多次包含进同一源文件也不稀奇。例如，使用 `Sales_item` 头文件的程序也可能使用 `string` 库。该程序不会（也不应该）知道 `Sales_item` 头文件使用了 `string` 库。在这种情况下，`string` 头文件被包含了两次：一次是通过程序本身直接包含，另一次是通过包含 `Sales_item` 头文件而间接包含。

Accordingly, it is important to design header files so that they can be included more than once in a single source file. We must ensure that including a header file more than once does not cause multiple definitions of the classes and objects that the header file defines. A common way to make headers safe uses the preprocessor to define a header guard. The guard is used to avoid reprocessing the contents of a header file if the header has already been seen.

因此，设计头文件时，应使其可以多次包含在同一源文件中，这一点很重要。我们必须保证多次包含同一头文件不会引起该头文件定义的类和对象被多次定义。使得头文件安全的通用做法，是使用预处理器定义头文件保护符。头文件保护符用于避免在已经见到头文件的情况下重新处理该头文件的内容。

Avoiding Multiple Inclusions

避免多重包含

Before we write our own header, we need to introduce some additional preprocessor facilities. The preprocessor lets us define our own variables.

在编写头文件之前，我们需要引入一些额外的预处理器设施。预处理器允许我们自定义变量。



Names used for preprocessor variables must be unique within the program. Any uses of a name that matches a preprocessor variable is assumed to refer to the preprocessor variable.

预处理器变量的名字在程序中必须是唯一的。任何与预处理器变量相匹配的名字的使用都关联到该预处理器变量。

To help avoid name clashes, preprocessor variables usually are written in all uppercase letters.

为了避免名字冲突，预处理器变量通常用全大写字母表示。

A preprocessor variable has two states: defined or not yet defined. Various preprocessor directives define and test the state of preprocessor variables. The `#define` directive takes a name and defines that name as a preprocessor variable. The `#ifndef` directive tests whether the specified preprocessor variable has not yet been defined. If it hasn't, then everything following the `#ifndef` is processed up to the next `#endif`.

预处理器变量有两种状态：已定义或未定义。定义预处理器变量和检测其状态所用的预处理器指示不同。`#define` 指示接受一个名字并定义该名字为预处理器变量。`#ifndef` 指示检测指定的预处理器变量是否未定义。如果预处理器变量未定义，那么跟在其后的所有指示都被处理，直到出现 `#endif`。

We can use these facilities to guard against including a header more than once:

可以使用这些设施来预防多次包含同一头文件：

```
#ifndef SALESITEM_H
#define SALESITEM_H
// Definition of Sales_item class and related functions goes here
#endif
```

The conditional directive

条件指示

```
#ifndef SALESITEM_H
```

tests whether the `SALESITEM_H` preprocessor variable has *not* been defined. If `SALESITEM_H` has not been defined, the `#ifndef` succeeds and all the lines following `#ifndef` until the `#endif` is found are processed. Conversely, if the variable `SALESITEM_H` has been defined, then the `#ifndef` directive is false. The lines between it and the `#endif` directive are ignored.

测试 `SALESITEM_H` 预处理器变量是否未定义。如果 `SALESITEM_H` 未定义，那么 `#ifndef` 测试成功，跟在 `#ifndef` 后面的所有行都被执行，直到发现 `#endif`。相反，如果 `SALESITEM_H` 已定义，那么 `#ifndef` 指示测试为假，该指示和 `#endif` 指示间的代码都被忽略。

To guarantee that the header is processed only once in a given source file, we start by testing the `#ifndef`. The first time the header is processed, this test will succeed, because `SALESITEM_H` will not yet have been defined. The next statement defines `SALESITEM_H`. That way, if the file we are compiling happens to include this header a second time, the `#ifndef` directive will discover that `SALESITEM_H` is defined and skip the rest of the header file.

Section 2.9. Writing Our Own Header Files

为了保证头文件在给定的源文件中只处理过一次，我们首先检测 `#ifndef`。第一次处理头文件时，测试会成功，因为 `SALESITEM_H` 还未定义。下一条语句定义了 `SALESITEM_H`。那样的话，如果我们编译的文件恰好又一次包含了该头文件。`#ifndef` 指示会发现 `SALESITEM_H` 已经定义，并且忽略该头文件的剩余部分。



Headers should have guards, even if they aren't included by another header. Header guards are trivial to write and can avoid mysterious compiler errors if the header subsequently is included more than once.

头文件应该含有保护符，即使这些头文件不会被其他头文件包含。编写头文件保护符并不困难，而且如果头文件被包含多次，它可以避免难以理解的编译错误。

This strategy works well provided that no two headers define and use a pre-processor constant with the same name. We can avoid problems with duplicate preprocessor variables by naming them for an entity, such as a class, that is defined inside the header. A program can have only one class named `Sales_item`. By using the class name to compose the name of the header file and the preprocessor variable, we make it pretty likely that only one file will use this preprocessor variable.

当没有两个头文件定义和使用同名的预处理器常量时，这个策略相当有效。我们可以为定义在头文件里的实体（如类）命名预处理器变量来避免预处理器变量重名的问题。一个程序只能含有一个名为 `Sales_item` 的类。通过使用类名来组成头文件和预处理器变量的名字，可以使得很可能只有一个文件将会使用该预处理器变量。

Using Our Own Headers

使用自定义的头文件

The `#include` directive takes one of two forms:

`#include` 指示接受以下两种形式：

```
#include <standard_header>
#include "my_file.h"
```

If the header name is enclosed by angle brackets (`< >`)，it is presumed to be a standard header. The compiler will look for it in a predefined set of locations, which can be modified by setting a search path environment variable or through a command line option. The search methods used vary significantly across compilers. We recommend you ask a colleague or consult your compiler's user's guide for further information. If the header name is enclosed by a pair of quotation marks, the header is presumed to be a nonsystem header. The search for nonsystem headers usually begins in the directory in which the source file is located.

如果头文件名括在尖括号 (`< >`) 里，那么认为该头文件是标准头文件。编译器将会在预定义的位置集查找该头文件，这些预定义的位置可以通过设置查找路径环境变量或者通过命令行选项来修改。使用的查找方法因编译器的不同而差别迥异。建议你咨询同事或者查阅编译器用户指南来获得更多的信息。如果头文件名括在一对引号里，那么认为它是非系统头文件，非系统头文件的查找通常开始于源文件所在的路径。

Team LiB

◀ PREVIOUS NEXT ▶

Chapter Summary

小结

Types are fundamental to all programming in C++.

类型是 C++ 程序设计的基础。

Each type defines the storage requirements and the operations that may be performed on all objects of that type. The language provides a set of fundamental built-in types such as `int` and `char`. These types are closely tied to their representation on the machine's hardware.

每种类型都定义了其存储空间要求和可以在该类型的所有对象上执行的操作。C++ 提供了一组基本内置类型，如 `int`、`char` 等。这些类型与它们在机器硬件上的表示方式紧密相关。

Types can be `nonconst` or `const`; a `const` object must be initialized and its value may not be changed. In addition, we can define compound types, such as references. A reference provides another name for an object. A compound type is a type that is defined in terms of another type.

类型可以为 `const` 或非 `const`；`const` 对象必须要初始化，且其值不能被修改。另外，我们还可以定义复合类型，如引用。引用为对象提供了另一个名字。复合类型是用其他类型定义的类型。

The language lets us define our own types by defining a class. The library uses the class facility to provide a set of higher-level abstractions such as the `IO` and `string` types.

C++ 语言支持通过定义类来自定义类型。标准库使用类设施来提供一组高级的抽象概念，如 `IO` 和 `string` 类型。

C++ is a statically typed language: Variables and functions must be declared before they are used. A variable can be declared many times but defined only once. It is almost always a good idea to initialize variables when you define them.

C++ 是一种静态类型语言：变量和函数在使用前必须先声明。变量可以声明多次但是只能定义一次。定义变量时就进行初始化几乎总是个好主意。

Defined Terms

术语

access labels (访问标号)

Members in a class may be defined to be `private`, which protects them from access from code that uses the type. Members may also be defined as `public`, which makes them accessible code throughout the program.

类的成员可以定义为 `private`，这能够防止使用该类型的代码访问该成员。成员还可以定义为 `public`，这将使该整个程序中都可访问成员。

address (地址)

Number by which a byte in memory can be found.

一个数字，通过该数字可在存储器上找到一个字节。

arithmetic types (算术类型)

The arithmetic types represent numbers: integers and floating point. There are three types of floating point values: `long double`, `double`, and `float`. These represent extended, double, and single precision values. It is almost always right to use `double`. In particular, `float` is guaranteed only six significant digits too small for most calculations. The integral types include `bool`, `char`, `wchar_t`, `short`, `int`, and `long`. Integer types can be signed or unsigned. It is almost always right to avoid `short` and `char` for arithmetic. Use `unsigned` for counting. The `bool` type may hold only two values: `true` or `false`. The `wchar_t` type is intended for characters from an extended character set; `char` type is used for characters that fit in 8 bits, such as Latin-1 or ASCII.

表示数值即整数和浮点数的类型。浮点型值有三种类型：`long double`、`double` 和 `float`，分别表示扩展精度值、双精度值和单精度值。一般总是使用 `double` 型。特别地，`float` 只能保证六位有效数字，这对于大多数的计算来说都不够。整型包括 `bool`、`char`、`wchar_t`、`short`、`int` 和 `long`。整型可以是带符号或无符号的。一般在算术计算中总是避免使用 `short` 和 `char`。`unsigned` 可用于计数。`bool` 类型只有 `true` 和 `false` 两个值。`wchar_t` 类型用于扩展字符集的字符；`char` 类型用于适合 8 个位的字符，比如 Latin-1 或者 ASCII。

array (数组)

Data structure that holds a collection of unnamed objects that can be accessed by an index. This chapter introduced the use of character arrays to hold string literals. [Chapter 4](#) will discuss arrays in much more detail.

存储一组可通过下标访问的未命名对象的数据结构。本章介绍了存储字符串字面值的字符数组。[第四章](#)将会更加详细地介绍数组。

byte (字节)

Typically the smallest addressable unit of memory. On most machines a byte is 8 bits.

最小的可寻址存储单元。大多数的机器上一个字节有 8 个位 (bit)。

class (类)

C++ mechanism for defining data types. Classes are defined using either the `class` or `struct` keyword. Classes may have data and function members. Members may be `public` or `private`. Ordinarily, function members that define the operations on the type are made `public`; data members and functions used in the implementation of the class are made `private`. By default, members in a class defined using the `class` keyword are private; members in a class defined using the `struct` keyword are public.

C++ 中定义数据类型的机制。类可以用 `class` 或 `struct` 关键字定义。类可以有数据和函数成员。成员可以是 `public` 或 `private`。一般来说，定义该类型的操作的函数成员设为 `public`；用于实现该类的数据成员和函数设为 `private`。默认情况下，用 `class` 关键字定义的类其成员为 `private`，而用 `struct` 关键字定义的类其成员为 `public`。

class member (类成员)

A part of a class. Members are either data or operations.

类的一部分，可以是数据或操作。

compound type (复合类型)

A type, such as a reference, that is defined in terms of another type. [Chapter 4](#) covers two additional compound types: pointers and arrays.

用其他类型定义的类型，如引用。[第四章](#)将介绍另外两种复合类型：指针和数组。

const reference (const 引用)

A reference that may be bound to a `const` object, a non`const` object, or to an rvalue. A `const` reference may not change the object to which it refers.

可以绑定到 `const` 对象、非 `const` 对象或右值的引用。`const` 引用不能改变与其相关联的对象。

constant expression (常量表达式)

An integral expression whose value can be evaluated at compile-time.

值可以在编译时计算出来的整型表达式。

constructor (构造函数)

Special member function that is used to initialize newly created objects. The job of a constructor is to ensure that the data members of an object have safe, sensible initial values.

用来初始化新建对象的特殊成员函数。构造函数的任务是保证对象的数据成员拥有可靠且合理的初始值。

copy-initialization (复制初始化)

Form of initialization that uses the `=` symbol to indicate that variable should be initialized as a copy of the initializer.

一种初始化形式，用“=”表明变量应初始化为初始化式的副本。

data member (数据成员)

The data elements that constitute an object. Data members ordinarily should be private.

组成对象的数据元素。数据成员一般应设为私有的。

declaration (声明)

Asserts the existence of a variable, function, or type defined elsewhere in the program. Some declarations are also definitions; only definitions allocate storage for variables. A variable may be declared by preceding its type with the keyword `extern`. Names may not be used until they are defined or declared.

表明在程序中其他地方定义的变量、函数或类型的存在性。有些声明也是定义。只有定义才为变量分配存储空间。可以通过在类型前添加关键字 `extern` 来声明变量。名字直到定义或声明后才能使用。

default constructor (默认构造函数)

The constructor that is used when no explicit values are given for an initializer of a class type object. For example, the default constructor for `string` initializes the new `string` as the empty `string`. Other `string` constructors initialize the `string` with characters specified when the `string` is created.

在没有为类类型对象的初始化式提供显式值时所使用的构造函数。例如，`string` 类的默认构造函数将新建的 `string` 对象初始化为空 `string`，而其他构造函数都是在创建 `string` 对象时用指定的字符去初始化 `string` 对象。

definition (定义)

Allocates storage for a variable of a specified type and optionally initializes the variable. Names may not be used until they are defined or declared.

为指定类型的变量分配存储空间，也可能可选地初始化该变量。名字直到定义或声明后才能使用。

direct-initialization (直接初始化)

Form of initialization that places a comma-separated list of initializers inside a pair of parentheses.

一种初始化形式，将逗号分隔的初始化式列表放在圆括号内。

enumeration (枚举)

A type that groups a set of named integral constants.

将一些命名整型常量聚组成的一种类型。

[enumerator \(枚举成员\)](#)

The named members of an enumeration. Each enumerator is initialized to an integral value and the value of the enumerator is `const`. Enumerators may be used where integral constant expressions are required, such as the dimension of an array definition.

枚举类型的有名字的成员。每个枚举成员都初始化为整型值且值为 `const`。枚举成员可用在需要整型常量表达式的地方，比如数组定义的维度。

[escape sequence \(转义字符\)](#)

Alternative mechanism for representing characters. Usually used to represent nonprintable characters such as newline or tab. An escape sequence is a backslash followed by a character, a three-digit octal number, or a hexadecimal number. The escape sequences defined by the language are listed on page [40](#). Escape sequences can be used as a literal character (enclosed in single quotes) or as part of a literal string (enclosed in double quotes).

一种表示字符的可选机制。通常用于表示不可打印字符如换行符或制表符。转义字符是反斜线后面跟着一个字符、一个 3 位八进制数或一个十六进制的数。`C++` 语言定义的转义字符列在[第 2.2 节](#)。转义字符还可用作字符字面值（括在单引号里）或用作字符串字面值的一部分（括在双引号里）。

[global scope \(全局作用域\)](#)

Scope that is outside all other scopes.

位于任何其他作用域外的作用域。

[header \(头文件\)](#)

A mechanism for making class definitions and other declarations available in multiple source files. User-defined headers are stored as files. System headers may be stored as files or in some other system-specific format.

使得类的定义和其他声明在多个源文件中可见的一种机制。用户定义的头文件以文件方式保存。系统头文件可能以文件方式保存，也可能以系统特有的其他格式保存。

[header guard \(头文件保护符\)](#)

The preprocessor variable defined to prevent a header from being included more than once in a single source file.

为防止头文件被同一源文件多次包含而定义的预处理器变量。

[identifier \(标识符\)](#)

A name. Each identifier is a nonempty sequence of letters, digits, and underscores that must not begin with a digit. Identifiers are case-sensitive: Upper- and lowercase letters are distinct. Identifiers may not use `C++` keywords. Identifiers may not contain two adjacent underscores nor may they begin with an underscore followed by an uppercase letter.

名字。每个标识符都是字母、数字和下划线的非空序列，且序列不能以数字开头。标识符是大小写敏感的：大写字母和小写字母含义不同。标识符不能使用`C++`中的关键字，不能包含相邻的下划线，也不能以下划线后跟一个大写字母开始。

[implementation \(实现\)](#)

The (usually `private`) members of a class that define the data and any operations that are not intended for use by code that uses the type. The `istream` and `ostream` classes, for example, manage an IO buffer that is part of their implementation and not directly accessible to users of those classes.

定义数据和操作的类成员（通常为 `private`），这些数据和操作并非为使用该类型的代码所用。例如，`istream` 和 `ostream` 类管理的 IO 缓冲区是它们的实现的一部分，但并不允许这些类的使用者直接访问。

[initialized \(已初始化的\)](#)

A variable that has an initial value. An initial value may be specified when defining a variable. Variables usually should be initialized.

含有初始值的变量。当定义变量时，可指定初始值。变量通常要初始化。

[integral types \(整型\)](#)

See arithmetic type.

见 arithmetic type。

[interface \(接口\)](#)

The operations supported by a type. Well-designed classes separate their interface and implementation, defining the interface in the `public` part of the class and the implementation in the `private` parts. Data members ordinarily are part of the implementation. Function members are part of the interface (and hence `public`) when they are operations that users of the type are expected to use and part of the implementation when they perform operations needed by the class but not defined for general use.

Keyterm Defined Terms

由某种类型支持的操作。设计良好的类分离了接口和实现，在类的 `public` 部分定义接口，`private` 部分定义实现。数据成员一般是实现的一部分。当函数成员是期望该类型的使用者使用的操作时，函数成员就是接口的一部分（因此为 `public`）；当函数成员执行类所需要的、非一般性使用的操作时，函数成员就是实现的一部分。

link (链接)

Compilation step in which multiple object files are put together to form an executable program. The link step resolves interfile dependencies, such as linking a function call in one file to a function definition contained in a second file.

一个编译步骤，此时多个目标文件放置在一起以形成可执行程序。链接步骤解决了文件间的依赖，如将一个文件中的函数调用链接到另一个文件中的函数定义。

literal constant (字面值常量)

A value such as a number, a character, or a string of characters. The value cannot be changed. Literal characters are enclosed in single quotes, literal strings in double quotes.

诸如数、字符或字符串的值，该值不能修改。字面值字符用单引号括住，而字面值字符串则用双引号括住。

local scope (局部作用域)

Term used to describe function scope and the scopes nested inside a function.

用于描述函数作用域和函数内嵌套的作用域的术语。

lvalue (左值)

A value that may appear on the left-hand of an assignment. A non`const` lvalue may be read and written.

可以出现在赋值操作左边的值。非 `const` 左值可以读也可以写。

magic number (魔数)

A literal number in a program whose meaning is important but not obvious. It appears as if by magic.

程序中意义重要但又不明显的字面值数字。它的出现好像变魔术一般。

nonconst reference (非 const 引用)

A reference that may be bound only to a non`const` lvalue of the same type as the reference. A non`const` reference may change the value of the underlying object to which it refers.

只能绑定到与该引用同类型的非 `const` 左值的引用。非 `const` 引用可以修改与其相关联的对象的值。

nonprintable character (非打印字符)

A character with no visible representation, such as a control character, a backspace, newline, and so on.

不可见字符。如控制符、回退删除符、换行符等。

object (对象)

A region of memory that has a type. A variable is an object that has a name.

具有类型的一段内存区域。变量就是一个有名字的对象。

preprocessor (预处理器)

The preprocessor is a program that runs as part of compilation of a C++ program. The preprocessor is inherited from C, and its uses are largely obviated by features in C++. One essential use of the preprocessor remains: the `#include` facility, which is used to incorporate headers into a program.

预处理器是作为 C++ 程序编译的一部分运行的程序。预处理器继承于 C 语言，C++ 的特征大量减少了它的使用，但仍保存了一个很重要的用法：`#include` 设施，用来把头文件并入程序。

private member (私有成员)

Member that is inaccessible to code that uses the class.

使用该类的代码不可访问的成员。

public member (公用成员)

Keyterm Defined Terms

Member of a class that can be used by any part of the program.

可被程序的任何部分使用的类成员。

reference (引用)

An alias for another object. Defined as follows:

对象的别名。定义如下：

```
type &id = object;
```

Defines *id* to be another name for *object*. Any operation on *id* is translated as an operation on *object*.

定义 *id* 为 *object* 的另一名字。任何对 *id* 的操作都会转变为对 *object* 的操作。

run time (运行时)

Refers to the time during which the program is executing.

指程序正执行的那段时间。

rvalue (右值)

A value that can be used as the right-hand, but not left-hand side of an assignment. An rvalue may be read but not written.

可用于赋值操作的右边但不能用于左边的值。右值只能读而不能写。

scope (作用域)

A portion of a program in which names have meaning. C++ has several levels of scope:

程序的一部分，在其中名字有意义。C++ 含有下列几种作用域：

global names defined outside any other scope.

全局——名字定义在任何其他作用域外。

class names defined by a class.

类——名字由类定义。

namespace names defined within a namespace.

命名空间——名字在命名空间中定义。

local names defined within a function.

局部——名字在函数内定义。

block names defined within a block of statements, that is, within a pair of curly braces.

块——名字定义在语句块中，也就是说，定义在一对花括号里。

statement names defined within the condition of a statement, such as an **if**, **for**, or **while**.

语句——名字在语句（如 **if**、**while** 和 **for** 语句）的条件内定义。

Scopes nest. For example, names declared at global scope are accessible in function and statement scope.

作用域可嵌套。例如，在全局作用域中声明的名字在函数作用域和语句作用域中都可以访问。

separate compilation (分别编译)

Ability to split a program into multiple separate source files.

将程序分成多个分离的源文件进行编译。

signed (带符号型)

Keyterm Defined Terms

Integer type that holds negative or positive numbers, including zero.

保存负数、正数或零的整型。

statically typed (静态类型的)

Term used to refer to languages such as C++ that do compile-time type checking. C++ verifies at compile-time that the types used in expressions are capable of performing the operations required by the expression.

描述进行编译时类型检查的语言（如 C++）的术语。C++ 在编译时验证表达式使用的类型可以执行该表达式需要的操作。

struct

Keyword that can be used to define a class. By default, members of a **struct** are public until specified otherwise.

用来定义类的关键字。除非有特殊的声明，默认情况下 **struct** 的成员都为公用的。

type-checking (类型检查)

Term used to describe the process by which the compiler verifies that the way objects of a given type are used is consistent with the definition of that type.

编译器验证给定类型的对象的使用方式是否与该类型的定义一致，描述这一过程的术语。

type specifier (类型说明符)

The part of a definition or declaration that names the type of the variables that follow.

定义或声明中命名其后变量的类型的部分。

typedef

Introduces a synonym for some other type. Form:

为某种类型引入同义词。格式：

```
typedef type synonym;
```

defines *synonym* as another name for the type named *type*.

定义 *synonym* 为名为 *type* 的类型的另一名字。

undefined behavior (未定义行为)

A usage for which the language does not specify a meaning. The compiler is free to do whatever it wants. Knowingly or unknowingly relying on undefined behavior is a great source of hard-to-track run-time errors and portability problems.

语言没有规定其意义的用法。编译器可以自由地做它想做的事。有意或无意地依赖未定义行为将产生大量难于跟踪的运行时错误和可移植性问题。

uninitialized (未初始化的)

Variable with no specified initial value. An uninitialized variable is not zero or "empty;" instead, it holds whatever bits happen to be in the memory in which it was allocated. Uninitialized variables are a great source of bugs.

没有指定初始值的变量。未初始化变量不是0也不是“空”，相反，它会保存碰巧遗留在分配给它的内存里的任何位。未初始化变量会产生很多错误。

unsigned (无符号型)

Integer type that holds values greater than or equal to zero.

保存大于等于零的值的整型。

variable initialization (变量初始化)

Term used to describe the rules for initializing variables and array elements when no explicit initializer is given. For class types, objects are initialized by running the class's default constructor. If there is no default constructor, then there is a compile-time error: The object must be given an explicit initializer. For built-in types, initialization depends on scope. Objects defined at global scope are initialized to 0; those defined at local scope are uninitialized and have undefined values.

描述当没有给出显式初始化式时初始化变量或数组元素的规则的术语。对类类型来说，通过运行类的默认构造函数来初始化对象。如果没有默认构造函数，那么将会出现编译时错误：必须给对象指定显式的初始化式。对于内置类型来说，初始化取决于作用域。定义在全局作用域的对象初始化为 0，而定义在局部作用域的对象则未初始化，拥有未定义值。

[void type \(空类型\)](#)

Special-purpose type that has no operations and no value. It is not possible to define a variable of type `void`. Most commonly used as the return type of a function that does not return a result.

用于特殊目的的没有操作也没有值的类型。不可能定义一个 `void` 类型的变量。最经常用作不返回结果的函数的返回类型。

[word \(字\)](#)

The natural unit of integer computation on a given machine. Usually a word is large enough to hold an address. Typically on a 32-bit machine a word is 4 bytes.

机器上的自然的整型计算单元。通常一个字足以容纳一个地址。一般在 32 位的机器上，机器字长为 4 个字节。

Team LiB

[PREVIOUS](#) [NEXT](#)

Chapter 3. Library Types

第三章 标准库类型

CONTENTS

Section 3.1 Namespace <code>using</code> Declarations	78
Section 3.2 Library <code>string</code> Type	80
Section 3.3 Library <code>vector</code> Type	90
Section 3.4 Introducing Iterators	95
Section 3.5 Library <code>bitset</code> Type	101
Chapter Summary	107
Defined Terms	107

In addition to the primitive types covered in [Chapter 2](#), C++ defines a rich library of abstract data types. Among the most important library types are `string` and `vector`, which define variable-sized character strings and collections, respectively. Associated with `string` and `vector` are companion types known as iterators, which are used to access the characters in a `string` or the elements in a `vector`. These library types are abstractions of more primitive types arrays and pointers that are part of the language.

除[第二章](#)介绍的基本数据类型外，C++ 还定义了一个内容丰富的抽象数据类型标准库。其中最重要的标准库类型是 `string` 和 `vector`，它们分别定义了大小可变的字符串和集合。`string` 和 `vector` 往往将迭代器用作配套类型（companion type），用于访问 `string` 中的字符，或者 `vector` 中的元素。这些标准库类型是语言组成部分中更基本的那些数据类型（如数组和指针）的抽象。

Another library type, `bitset`, provides an abstract way to manipulate a collection of bits. This class provides a more convenient way of dealing with bits than is offered by the built-in bitwise operators on values of integral type.

另一种标准库类型 `bitset`，提供了一种抽象方法来操作位的集合。与整型值上的内置位操作符相比，`bitset` 类类型提供了一种更方便的处理位的方式。

This chapter introduces the library `vector`, `string`, and `bitset` types. The next chapter covers arrays and pointers, and [Chapter 5](#) looks at built-in bitwise operators.

本章将介绍标准库中的 `vector`、`string` 和 `bitset` 类型。[第四章](#)将讨论数组和指针，[第五章](#)将讲述内置位操作符。

The types that we covered in [Chapter 2](#) are all low-level types: They represent abstractions such as numbers or characters and are defined in terms of how they are represented on the machine.

[第二章](#)所涉及的类型都是低层数据类型：这些类型表示数值或字符的抽象，并根据其具体机器表示来定义。

In addition to the types defined in the language, the standard library defines a number of higher level **abstract data types**. These library types are higher-level in that they mirror more complex concepts. They are abstract because when we use them we don't need to care about how the types are represented. We need to know only what operations they support.

除了这些在语言中定义的类型外，C++ 标准库还定义了许多更高级的抽象数据类型之所以说这些标准库类型是更高级的，是因为其中反映了更复杂的概念；之所以说它们是抽象的，是因为我们在使用时不需要关心它们是如何表示的，只需知道这些抽象数据类型支持哪些操作就可以了。

Two of the most important library types are `string` and `vector`. The `string` type supports variable-length character strings. The `vector` type holds a sequence of objects of a specified type. These types are important because they offer improvements over more primitive types defined by the language. [Chapter 4](#) looks at the language-level constructs that are similar to, but less flexible and more error-prone than, the library `string` and `vector` types.

两种最重要的标准库类型是 `string` 和 `vector`。`string` 类型支持长度可变的字符串，`vector` 可用于保存一组指定类型的对象。说它们重要，是因为它们在 C++ 定义的基本类型基础上作了一些改进。[第四章](#)还将学习类似于标准库中 `string` 和 `vector` 类型的语言级构造，但标准库的 `string` 和 `vector` 类型可能更灵活，且不易出错。

Another library type that offers a more convenient and reasonably efficient abstraction of a language level facility is the `bitset` class. This class lets us treat a value as a collection of bits. It provides a more direct way of operating on bits than do the bitwise operators that we cover in [Section 5.3](#) (p. 154).

另一种标准库类型提供了更方便和合理有效的语言级的抽象设施，它就是 `bitset` 类。通过这个类可以把某个值当作们的集合来处理。与 [第 5.3 节](#)介绍的位操作符相比，`bitset` 类提供操作位更直接的方法。

Before continuing our exploration of the library types, we'll look at a mechanism for simplifying access to the names defined in the library.

在继续探究标准库类型之前，我们先看一种机制，这种机制能够简化对标准库中所定义名字的访问。

3.1. Namespace `using` Declarations

3.1. 命名空间的 `using` 声明

The programs we've seen so far have referred to names from the library by explicitly noting that the name comes from the `std` namespace. For example, when we want to read from the standard input, we write `std::cin`. Such names use the `::` operator, which is the scope operator ([Section 1.2.2, p. 8](#)). This operator says that we should look for the name of the right-hand operand in the scope of the left-hand operand. Thus, `std::cin` says that we want the name `cin` that is defined in the namespace `std`. Referring to library names through this notation can be cumbersome.

在本章之前看到的程序，都是通过直接说明名字来自 `std` 命名空间，来引用标准库中的名字。例如，需要从标准输入读取数据时，就用 `std::cin`。这些名字都用了`::`操作符，该操作符是作用域操作符（[第 1.2.2 节](#)）。它的含义是右操作数的名字可以在左操作数的作用域中找到。因此，`std::cin` 的意思是说所需要名字 `cin` 是在命名空间 `std` 中定义的。显然，通过这种符号引用标准库名字的方式是非常麻烦的。

Fortunately, there are easier ways to use namespace members. In this section we'll cover the safest mechanism: [using declarations](#). We will see other ways to simplify the use of names from a namespace in [Section 17.2](#) (p. 712).

幸运的是，C++ 提供了更简洁的方式来使用命名空间成员。本节将介绍一种最安全的机制：[using 声明](#)。关于其他简化使用命名空间中名字的方法将在[第 17.2 节](#)中介绍

A `using` declaration allows us to access a name from a namespace without the prefix `namespace_name::`. A `using` declaration has the following form:

使用 `using` 声明可以在不需要加前缀 `namespace_name::` 的情况下访问命名空间中的名字。`using` 声明的形式如下：

```
using namespace::name;
```

Once the `using` declaration has been made, we can access `name` directly without reference to its namespace:

一旦使用了 `using` 声明，我们就可以直接引用名字，而不需要再引用该名字的命名空间。

```
#include <string>
#include <iostream>
// using declarations states our intent to use these names from the namespace std
using std::cin;
using std::string;
int main()
{
    string s;      // ok: string is now a synonym for std::string
    cin >> s;      // ok: cin is now a synonym for std::cin
    cout << s;      // error: no using declaration; we must use full name
    std::cout << s; // ok: explicitly use cout from namespace std
}
```

Using the unqualified version of a namespace name without a `using` declaration is an error, although some compilers may fail to detect this error.

没有 `using` 声明，而直接使用命名空间中名字的未限定版本是错误的，尽管有些编译器也许无法检测出这种错误。

A Separate Using Declaration Is Required for Each Name

每个名字都需要一个 `using` 声明

A `using` declaration introduces only one namespace member at a time. It allows us to be very specific regarding which names are used in our programs. If we want to use several names from `std` or any other namespace then we must issue a `using` declaration for each name that we intend to use. For example, we could rewrite the addition program from page 6 as follows:

一个 `using` 声明一次只能作用于一个命名空间成员。`using` 声明可用来明确指定在程序中用到的命名空间中的名字，如果希望使用 `std`（或其他的命名空间）中的几个名字，则必须为要用到的每个名字都提供一个 `using` 声明。例如，利用 `using` 声明可以这样重新编写[第 1.2.2 节](#)中的加法程序：

```
#include <iostream>
// using declarations for names from the standard library
using std::cin;
using std::cout;
using std::endl;
int main()
{
    cout << "Enter two numbers:" << endl;
    int v1, v2;
    cin >> v1 >> v2;
    cout << "The sum of " << v1
        << " and " << v2
        << " is " << v1 + v2 << endl;
    return 0;
}
```

The `using` declarations for `cin`, `cout`, and `endl` mean that we can use those names without the `std::` prefix, making the code easier to read.

对 `cin`, `cout` 和 `endl` 进行 `using` 声明，就意味着以后可以省前缀 `std::`，直接使用命名空间中的名字，这样代码可以更易读。

From this point on, our examples will assume that `using` declarations have been provided for the names we use from the standard library. Thus, we will refer to `cin`, not `std::cin`, in the text and in code examples. To keep the code examples short, we won't show the `using` declarations that are needed to compile the examples. Similarly, our code examples will not show the necessary `#include` directives. [Table A.1](#) (p. 810) in [Appendix A](#) lists the library names and corresponding headers for standard-library names we use in this primer.

从这里开始，假定本书所有例子中所用到的标准库中的名字都已提供了 `using` 声明。这样，无论是在文档还是在代码实例中引用 `cin`，我们都不再写为前缀形式 `std::cin`，为了使代码实例简短，我们还省略了编译时所必需的 `using` 声明。同样的，程序实例也会省略必需的 `#include` 指示。本书附录 A 中的表 A.1 列出了本书中用到的标准名字的库名和相应的头文件。



Readers should be aware that they must add appropriate `#include` and `using` declarations to our examples before compiling them.

在编译我们提供的实例程序前，读者一定要注意在程序中添加适当的 `#include` 和 `using` 声明。

Class Definitions that Use Standard Library Types

使用标准库类型的类定义

There is one case in which we should *always* use the fully qualified library names: inside header files. The reason is that the contents of a header are copied into our program text by the preprocessor. When we `#include` a file, it is as if the exact text of the header is part of our file. If we place a `using` declaration within a header, it is equivalent to placing the same `using` declaration in every program that includes the header *whether that program wants the using declaration or not*.

有一种情况下，必须总是使用完全限定的标准库名字：在头文件中。理由是头文件的内容会被预处理器复制到程序中。用 `#include` 包含文件时，相当于头文件中的文本将成为我们编写的文件的一部分。如果在头文件中放置 `using` 声明，就相当于在包含该头文件 `using` 的每个程序中都放置了同一 `using`，不论该程序是否需要 `using` 声明。



In general, it is good practice for headers to define only what is strictly necessary.

通常，头文件中应该只定义确实必要的东西。请养成这个好习惯。

Exercises Section 3.1

- Exercise 3.1:** Rewrite the program from [Section 2.3](#) (p. 43) that calculated the result of raising a given number to a given power to use appropriate `using` declarations rather than accessing library names through a `std::` prefix.

用适当的 `using` 声明，而不用 `std::`，访问标准库中名字的方法，重新编写[第 2.3 节](#)的程序，计算一给定数的给定次幂的结果。

3.2. Library `string` Type

3.2. 标准库 `string` 类型

The `string` type supports variable-length character strings. The library takes care of managing the memory associated with storing the characters and provides various useful operations. The library `string` type is intended to be efficient enough for general use.

`string` 类型支持长度可变的字符串，C++ 标准库将负责管理与存储字符相关的内存，以及提供各种有用的操作。标准库 `string` 类型的目的就是满足对字符串的一般应用。

As with any library type, programs that use `strings` must first include the associated header. Our programs will be shorter if we also provide an appropriate `using` declaration:

与其他的标准库类型一样，用户程序要使用 `string` 类型对象，必须包含相关头文件。如果提供了合适的 `using` 声明，那么编写出来的程序将会变得简短些：

```
#include <string>
using std::string;
```

3.2.1. Defining and Initializing `string`s

3.2.1. `string` 对象的定义和初始化

The `string` library provides several constructors ([Section 2.3.3](#), p. 49). A constructor is a special member function that defines how objects of that type can be initialized. [Table 3.1](#) on the facing page lists the most commonly used `string` constructors. The default constructor ([Section 2.3.4](#), p. 52) is used "by default" when no initializer is specified.

`string` 标准库支持几个构造函数 ([第 2.3.3 节](#))。构造函数是一个特殊成员函数，定义如何初始化该类型的对象。[表 3.1](#) 列出了几个 `string` 类型常用的构造函数。当没有明确指定对象初始化式时，系统将使用默认构造函数 ([第 2.3.4 节](#))。

Table 3.1. Ways to Initialize a `string`

表 3.1. 几种初始化 `string` 对象的方式

<code>string s1;</code>	Default constructor; <code>s1</code> is the empty string 默认构造函数 <code>s1</code> 为空串
<code>string s2(s1);</code>	Initialize <code>s2</code> as a copy of <code>s1</code> 将 <code>s2</code> 初始化为 <code>s1</code> 的一个副本
<code>string s3("value");</code>	Initialize <code>s3</code> as a copy of the string literal 将 <code>s3</code> 初始化为一个字符串字面值副本
<code>string s4(n, 'c');</code>	Initialize <code>s4</code> with <code>n</code> copies of the character ' <code>c</code> ' 将 <code>s4</code> 初始化为字符 ' <code>c</code> ' 的 <code>n</code> 个副本

Caution: Library `string` Type and String Literals

警告：标准库 `string` 类型和字符串字面值

For historical reasons, and for compatibility with C, character string literals are *not* the same type as the standard library `string` type. This fact can cause confusion and is important to keep in mind when using a string literal or the `string` data type.

因为历史原因以及为了与 C 语言兼容，字符串字面值与标准库 `string` 类型不是同一种类型。这一点很容易引起混乱，编程时一定要注意区分字符串字面值和 `string` 数据类型的使用，这很重要。

Exercises Section 3.2.1

Exercise 3.2: What is a default constructor?

什么是默认构造函数?

Exercise 3.3: Name the three ways to initialize a `string`.

列举出三种初始化 `string` 对象的方法。

Exercise 3.4: What are the values of `s` and `s2`?

`s` 和 `s2` 的值分别是什么?

```
string s;
int main() {
    string s2;
}
```

3.2.2. Reading and Writing `string`s

3.2.2. `string` 对象的读写

我们已在第一章学习了用 `iostream` 标准库来读写内置类型的值，如 `int double` 等。同样地，也可以用 `iostream` 和 `string` 标准库，使用标准输入输出操作符来读写 `string` 对象：

```
// Note: #include and using declarations must be added to compile this code
int main()
{
    string s;           // empty string
    cin >> s;          // read whitespace-separated string into s
    cout << s << endl; // write s to the output
    return 0;
}
```

This program begins by defining a `string` named `s`. The next line,

以上程序首先定义命名为 `s` 的 `string` 第二行代码：

```
cin >> s;           // read whitespace-separated string into s
```

reads the standard input storing what is read into `s`. The `string` input operator:

从标准输入读取 `string` 并将读入的串存储在 `s` 中。`string` 类型的输入操作符：

- Reads and discards any leading whitespace (e.g., spaces, newlines, tabs)
读取并忽略开头所有的空白字符（如空格，换行符，制表符）。
- It then reads characters until the next whitespace character is encountered
读取字符直至再次遇到空白字符，读取终止。

So, if the input to this program is "Hello World!" , (note leading and trailing spaces) then the output will be "Hello" with no extra spaces.

如果给定和上一个程序同样的输入，则输出的结果是"Hello World!"（注意到开头和结尾的空格），则屏幕上将输出"Hello"，而不含任何空格。

The input and output operations behave similarly to the operators on the builtin types. In particular, the operators return their left-hand operand as their result. Thus, we can chain together multiple reads or writes:

输入和输出操作的行为与内置类型操作符基本类似。尤其是，这些操作符返回左操作数作为运算结果。因此，我们可以把多个读操作或多个写操作放在一起：

```
string s1, s2;
cin >> s1 >> s2; // read first input into s1, second into s2
cout << s1 << s2 << endl; // write both strings
```

Section 3.2. Library string Type

If we give this version of the program the same input as in the previous paragraph, our output would be

如果给定和上一个程序同样的输入，则输出的结果将是：

HelloWorld!



To compile this program, you must add `#include` directives for both the `iostream` and `string` libraries and must issue `using` declarations for all the names used from the library: `string`, `cin`, `cout`, and `endl`.

对于上例，编译时必须加上 `#include` 来标示 `iostream` 和 `string` 标准库，以及给出用到的所有标准库中的名字（如 `string`, `cin`, `cout`, `endl`）的 `using` 声明。

The programs presented from this point on will assume that the needed `#include` and `using` declarations have been made.

从本例开始的程序均假设程序中所有必须 `#include` 和 `using` 声明已给出。

Reading an Unknown Number of `string`s

读入未知数目的 `string` 对象

Like the input operators that read built-in types, the `string` input operator returns the stream from which it read. Therefore, we can use a `string` input operation as a condition, just as we did when reading `ints` in the program on page 18. The following program reads a set of `strings` from the standard input and writes what it has read, one `string` per line, to the standard output:

和内置类型的输入操作一样，`string` 的输入操作符也会返回所读的数据流。因此，可以把输入操作作为判断条件，这与我们在 1.4.4 节读取整型数据的程序做法是一样的。下面的程序将从标准输入读取一组 `string` 对象，然后在标准输出上逐行输出：

```
int main()
{
    string word;
    // read until end-of-file, writing each word to a new line
    while (cin >> word)
        cout << word << endl;
    return 0;
}
```

In this case, we read into a `string` using the input operator. That operator returns the `istream` from which it read, and the `while` condition tests the stream after the read completes. If the stream is valid it hasn't hit end-of-file or encountered an invalid input then the body of the `while` is executed and the value we read is printed to the standard output. Once we hit end-of-file, we fall out of the `while`.

上例中，用输入操作符来读取 `string` 对象。该操作符返回所读的 `istream` 对象，并在读取结束后，作为 `while` 的判断条件。如果输入流是有效的，即还未到达文件尾且未遇到无效输入，则执行 `while` 循环体，并将读取到的字符串输出到标准输出。如果到达了文件尾，则跳出 `while` 循环。

Using `getline` to Read an Entire Line

使用 `getline` 读取整行文本

There is an additional useful `string` IO operation: `getline`. This is a function that takes both an input stream and a `string`. The `getline` function reads the next line of input from the stream and stores what it read, *not including* the newline, in its `string` argument. Unlike the input operator, `getline` does not ignore leading newlines. Whenever `getline` encounters a newline, even if it is the first character in the input, it stops reading the input and returns. The effect of encountering a newline as the first character in the input is that the `string` argument is set to the empty `string`.

另外还有一个有用的 `string` IO 操作： `getline`。这个函数接受两个参数：一个输入流对象和一个 `string` 对象。`getline` 函数从输入流的下一行读取，并保存读取的内容到不包括换行符。和输入操作符不一样的是，`getline` 并不忽略行开头的换行符。只要 `getline` 遇到换行符，即便它是输入的第一个字符，`getline` 也将停止读入并返回。如果第一个字符就是换行符，则 `string` 参数将被置为空 `string`。

The `getline` function returns its `istream` argument so that, like the input operator, it can be used as a condition. For example, we could rewrite the previous program that wrote one word per line to write a line at a time instead:

`getline` 函数将 `istream` 参数作为返回值，和输入操作符一样也把它用作判断条件。例如，重写前面那段程序，把每行输出一个单词改为每次输出一行文本：

```
int main()
{
    string line;
    // read line at time until end-of-file
    while (getline(cin, line))
        cout << line << endl;
    return 0;
}
```

Section 3.2. Library string Type

Because `line` does not contain a newline, we must write our own if we want the `strings` written one to a line. As usual, we use `endl` to write a newline and flush the output buffer.

由于 `line` 不含换行符，若要逐行输出需要自行添加。照常，我们用 `endl` 来输出一个换行符并刷新输出缓冲区。



The newline that causes `getline` to return is discarded; it does *not* get stored in the `string`.

由于 `getline` 函数返回时丢弃换行符，换行符将不会存储在 `string` 对象中。

Exercises Section 3.2.2

Exercise 3.5: Write a program to read the standard input a line at a time. Modify your program to read a word at a time.

编写程序实现从标准输入每次读入一行文本。然后改写程序，每次读入一个单词。

Exercise 3.6: Explain how whitespace characters are handled in the `string` input operator and in the `getline` function.

解释 `string` 类型的输入操作符和 `getline` 函数分别如何处理空白字符。

3.2.3. Operations on `string`s

3.2.3. `string` 对象的操作

Table 3.2 on the next page lists the most commonly used `string` operations.

表 3.2 列出了常用的 `string` 操作。

Table 3.2. `string` Operations

<code>s.empty()</code>	Returns <code>true</code> if <code>s</code> is empty; otherwise returns <code>false</code> 如果 <code>s</code> 为空串，则返回 <code>true</code> ，否则返回 <code>false</code> 。
<code>s.size()</code>	Returns number of characters in <code>s</code> 返回 <code>s</code> 中字符的个数
<code>s[n]</code>	Returns the character at position <code>n</code> in <code>s</code> ; positions start at 0. 返回 <code>s</code> 中位置为 <code>n</code> 的字符，位置从 0 开始计数
<code>s1 + s2</code>	Returns a <code>string</code> equal to the concatenation of <code>s1</code> and <code>s2</code> 把 <code>s1</code> 和 <code>s2</code> 连接成一个新字符串，返回新生成的字符串
<code>s1 = s2</code>	Replaces characters in <code>s1</code> by a copy of <code>s2</code> 把 <code>s1</code> 内容替换为 <code>s2</code> 的副本
<code>v1 == v2</code>	Returns <code>true</code> if <code>v1</code> and <code>v2</code> are equal; <code>false</code> otherwise 比较 <code>v1</code> 与 <code>v2</code> 的内容，相等则返回 <code>true</code> ，否则返回 <code>false</code>
<code>!=, <, <=, >, and >=</code>	Have their normal meanings 保持这些操作符惯有的含义

The `string size` and `empty` Operations

Section 3.2. Library string Type

`string` 的 `size` 和 `empty` 操作

The length of a `string` is the number of characters in the `string`. It is returned by the `size` operation:

`string` 对象的长度指的是 `string` 对象中字符的个数，可以通过 `size` 操作获取：

```
int main()
{
    string st("The expense of spirit\n");
    cout << "The size of " << st << "is " << st.size()
        << " characters, including the newline" << endl;
    return 0;
}
```

If we compile and execute this program it yields

编译并运行这个程序，得到的结果为：

```
The size of The expense of spirit
is 22 characters, including the newline
```

Often it is useful to know whether a `string` is empty. One way we could do so would be to compare `size` with 0:

了解 `string` 对象是否空是有用的。一种方法是将 `size` 与 0 进行比较：

```
if (st.size() == 0)
    // ok: empty
```

In this case, we don't really need to know how many characters are in the `string`; we are only interested in whether the `size` is zero. We can more directly answer this question by using the `empty` member:

本例中，程序员并不需要知道 `string` 对象中有多少个字符，只想知道 `size` 是否为 0。用 `string` 的成员函数 `empty()` 可以更直接地回答这个问题：

```
if (st.empty())
    // ok: empty
```

The `empty` function returns the `bool` ([Section 2.1](#), p. 34) value `true` if the `string` contains no characters; otherwise, it returns `false`.

`empty()` 成员函数将返回 `bool` ([2.1 节](#))，如果 `string` 对象为空则返回 `true` 否则返回 `false`。

`string::size_type`

`string::size_type` 类型

It might be logical to expect that `size` returns an `int`, or, thinking back to the note on page 38, an `unsigned`. Instead, the `size` operation returns a value of type `string::size_type`. This type requires a bit of explanation.

从逻辑上来讲，`size()` 成员函数似乎应该返回整形数值，或如 [2.2 节](#)“建议”中所述的无符号整数。但事实上，`size` 操作返回的是 `string::size_type` 类型的值。我们需要对这种类型做一些解释。

The `string` class and many other library types defines several companion types. These companion types make it possible to use the library types in a machine-independent manner. The type `size_type` is one of these companion types. It is defined as a synonym for an `unsigned` type either `unsigned int` or `unsigned long` that is guaranteed to be big enough to hold the size of any `string`. To use the `size_type` defined by `string`, we use the scope operator to say that the name `size_type` is defined in the `string` class.

`string` 类类型和许多其他库类型都定义了一些配套类型（companion type）。通过这些配套类型，库类型的使用就能与机器无关（machine-independent）。`size_type` 就是这些配套类型中的一种。它定义为与 `unsigned` 型（`unsigned int` 或 `unsigned long`）具有相同的含义，而且可以保证足够大能够存储任意 `string` 对象的长度。为了使用由 `string` 类型定义的 `size_type` 类型是由 `string` 类定义。



Any variable used to store the result from the `string size` operation ought to be of type `string::size_type`. It is particularly important *not* to assign the return from `size` to an `int`.

任何存储 `string` 的 `size` 操作结果的变量必须为 `string::size_type` 类型。特别重要的是，还要把 `size` 的返回值赋给一个 `int` 变量。

Although we don't know the precise type of `string::size_type`, we do know that it is an `unsigned` type ([Section 2.1.1](#), p. 34). We also know that for a given type, the `unsigned` version can hold a positive value twice as large as the corresponding `signed` type can hold. This fact implies that the largest `string` could be twice as large as the size an `int` can hold.

虽然我们不知道 `string::size_type` 的确切类型，但可以知道它是 `unsigned` 型 ([2.1.1 节](#))。对于任意一种给定的数据类型，它的 `unsigned` 型所能表示的最大正数值比对应的 `signed` 型要大倍。这个事实表明 `size_type` 存储的 `string` 长度是 `int` 所能存储的两倍。

Another problem with using an `int` is that on some machines the size of an `int` is too small to hold the size of even plausibly large `strings`. For

Section 3.2. Library string Type

example, if a machine has 16-bit `ints`, then the largest `string` an `int` could represent would have 32,767 characters. A `string` that held the contents of a file could easily exceed this size. The safest way to hold the `size` of a `string` is to use the type the library defines for this purpose, which is `string::size_type`.

使用 `int` 变量的另一个问题是，有些机器上 `int` 变量的表示范围太小，甚至无法存储实际并不长的 `string` 对象。如在有 16 位 `int` 型的机器上，`int` 类型变量最大只能表示 32767 个字符的 `string` 个字符的 `string` 对象。而能容纳一个文件内容的 `string` 对象轻易就会超过这个数字。因此，为了避免溢出，保存一个 `string` 对象 `size` 的最安全的方法就是使用标准库类型 `string::size_type`。

The `string` Relational Operators

`string` 关系操作符

The `string` class defines several operators that compare two `string` values. Each of these operators works by comparing the characters from each `string`.

`string` 类定义了几种关系操作符用来比较两个 `string` 值的大小。这些操作符实际上是每个 `string`



`string` comparisons are case-sensitive—the upper- and lowercase versions of a letter are different characters. On most computers, the uppercase letters come first: Every uppercase letter is less than any lowercase letter.

`string` 对象比较操作是区分大小写的，即同一个字符的大小写形式被认为是两个不同的字符。在多数计算机上，大写的字母位于小写之前：任何一个大写之母都小于任意的小写字母。

The equality operator compares two `strings`, returning `true` if they are equal. Two `strings` are equal if they are the same length and contain the same characters. The library also defines `!=` to test whether two `strings` are unequal.

`==` 操作符比较两个 `string` 对象，如果它们相等，则返回 `true`。两个 `string` 对象相等是指它们的长度相同，且含有相同的字符。标准库还定义了 `!=` 操作符来测试两个 `string` 对象是否不等。

The relational operators `<`, `<=`, `>`, `>=` test whether one `string` is less than, less than or equal, greater than, or greater than or equal to another:

关系操作符 `<`, `<=`, `>`, `>=` 分别用于测试一个 `string` 对象是否小于、小于或等于、大于、大于或等于另一个 `string` 对象：

```
string big = "big", small = "small";
string s1 = big;    // s1 is a copy of big
if (big == small) // false
// ...
if (big <= s1)    // true, they're equal, so big is less than or equal to s1
// ...
```

The relational operators compare `strings` using the same strategy as in a (case-sensitive) dictionary:

关系操作符比较两个 `string` 对象时采用了和（大小写敏感的）字典排序相同的策略：

- If two `strings` have different lengths and if every character in the shorter `string` is equal to the corresponding character of the longer `string`, then the shorter `string` is less than the longer one.
- 如果两个 `string` 对象长度不同，且短的 `string` 对象与长的 `string` 对象的前面部分相匹配，则短的 `string` 对象小于长的 `string` 对象。
- If the characters in two `strings` differ, then we compare them by comparing the first character at which the `strings` differ.
- 如果 `string` 对象的字符不同，则比较第一个不匹配的字符。`string`

As an example, given the `strings`

举例来说，给定 `string` 对象；

```
string substr = "Hello";
string phrase = "Hello World";
string slang = "Hiya";
```

then `substr` is less than `phrase`, and `slang` is greater than either `substr` or `phrase`.

则 `substr` 小于 `phrase`，而 `slang` 则大于 `substr` 或 `phrase`

Assignment for `strings`

Section 3.2. Library string Type

string 对象的赋值

In general the library types strive to make it as easy to use a library type as it is to use a built-in type. To this end, most of the library types support assignment. In the case of `strings`, we can assign one `string` object to another:

总体上说，标准库类型尽量设计得和基本数据类型一样方便易用。因此，大多数库类型支持赋值操作。对 `string` 对象来说，可以把一个 `string` 对象赋值给另一个 `string` 对象；

```
// st1 is an empty string, st2 is a copy of the literal
string st1, st2 = "The expense of spirit";
st1 = st2; // replace st1 by a copy of st2
```

After the assignment, `st1` contains a copy of the characters in `st2`.

赋值操作后，`st1` 就包含了 `st2` 串所有字符的一个副本。

Most `string` library implementations go to some trouble to provide efficient implementations of operations such as assignment, but it is worth noting that conceptually, assignment requires a fair bit of work. It must delete the storage containing the characters associated with `st1`, allocate the storage needed to contain a copy of the characters associated with `st2`, and then copy those characters from `st2` into this new storage.

大多数 `string` 库类型的赋值等操作的实现都会遇到一些效率上的问题，但值得注意的是，从概念上讲，赋值操作确实需要做一些工作。它必须先把 `st1` 占用的相关内存释放掉，然后再分配给 `st2` 足够存放 `st2` 副本的内存空间，最后把 `st2` 中的所有字符复制到新分配的内存空间。

Adding Two strings

两个 string 对象相加

Addition on `strings` is defined as concatenation. That is, it is possible to concatenate two or more `strings` through the use of either the plus operator (+) or the compound assignment operator (+=) ([Section 1.4.1](#), p. 13). Given the two `strings`

`string` 对象的加法被定义为连接（concatenation）。也就是说，两个（或多个）`string` 对象可以通过使用加操作符 + 或者复合赋值操作符 += ([1.4.1 节](#)) 连接起来。给定两个 `string` 对象：

```
string s1("hello, ");
string s2("world\n");
```

we can concatenate the two `strings` to create a third `string` as follows:

下面把两个 `string` 对象连接起来产生第三个 `string` 对象：

```
string s3 = s1 + s2; // s3 is hello, world\n
```

If we wanted to append `s2` to `s1` directly, then we would use +=:

如果要把 `s2` 直接追加到 `s1` 的末尾，可以使用 += 操作符：

```
s1 += s2; // equivalent to s1 = s1 + s2
```

Adding Character String Literals and strings

和字符串字面值的连接

The `strings` `s1` and `s2` included punctuation directly. We could achieve the same result by mixing `string` objects and string literals as follows:

上面的字符串对象 `s1` 和 `s2` 直接包含了标点符号。也可以通过将 `string` 对象和字符串字面值混合连接得到同样的结果：

```
string s1("hello");
string s2("world");

string s3 = s1 + ", " + s2 + "\n";
```

When mixing `strings` and string literals, at least one operand to each + operator must be of `string` type:

当进行 `string` 对象和字符串字面值混合连接操作时，+ 操作符的左右操作数必须至少有一个是 `string` 类型的：

```
string s1 = "hello"; // no punctuation
string s2 = "world";
```

Section 3.2. Library string Type

```
string s3 = s1 + ", " ; // ok: adding a string and a literal
string s4 = "hello" + ", " ; // error: no string operand
string s5 = s1 + ", " + "world" ; // ok: each + has string operand
string s6 = "hello" + ", " + s2; // error: can't add string literals
```

The initializations of `s3` and `s4` involve only a single operation. In these cases, it is easy to determine that the initialization of `s3` is legal: We initialize `s3` by adding a `string` and a string literal. The initialization of `s4` attempts to add two string literals and is illegal.

`s3` 和 `s4` 的初始化只用了一个单独的操作。在这些例子中，很容易判断 `s3` 的初始化是合法的：把一个 `string` 对象和一个字符串字面值连接起来。而 `s4` 的初始化试图将两个字符串字面值相加，因此是非法的。

The initialization of `s5` may appear surprising, but it works in much the same way as when we chain together input or output expressions ([Section 1.2](#), p. 5). In this case, the `string` library defines addition to return a `string`. Thus, when we initialize `s5`, the subexpression `s1 + ", "` returns a `string`, which can be concatenated with the literal `"world\n"`. It is as if we had written

`s5` 的初始化方法显得有点不可思议，但这种用法和标准输入输出的串联效果是一样的（[1.2 节](#)）。本例中，`string` 标准库定义加操作返回一个 `string` 对象。这样，在对 `s5` 进行初始化时，子表达式 `s1 + ", "` 将返回一个新 `string` 对象，后者再和字面值 `"world\n"` 连接。整个初始化过程可以改写为：

```
string tmp = s1 + ", " ; // ok: + has a string operand
s5 = tmp + "world" ; // ok: + has a string operand
```

On the other hand, the initialization of `s6` is illegal. Looking at each subexpression in turn, we see that the first subexpression adds two string literals. There is no way to do so, and so the statement is in error.

而 `s6` 的初始化是非法的。依次来看每个子表达式，则第一个子表达式试图把两个字符串字面值连接起来。这是不允许的，因此这个语句是错误的。

Fetching a Character from a `string`

从 `string` 对象获取字符

The `string` type uses the `subscript` (`[]`) operator to access the individual characters in the `string`. The subscript operator takes a `size_type` value that denotes the character position we wish to fetch. The value in the subscript is often called "the subscript" or "an [index](#)."

`string` 类型通过下标操作符 (`[]`) 来访问 `string` 对象中的单个字符。下标操作符需要取一个 `size_type` 类型的值，来标明要访问字符的位置。这个下标中的值通常被称为“下标”或“索引”（[index](#)）



Subscripts for `strings` start at zero; if `s` is a `string`, then if `s` isn't empty, `s[0]` is the first character in the `string`, `s[1]` is the second if there is one, and the last character is in `s[s.size() - 1]`.

`string` 对象的下标从 0 开始。如果 `s` 是一个 `string` 对象且 `s` 不空，则 `s[0]` 就是字符串的第一个字符，`s[1]` 就表示第二个字符（如果有的话），而 `s[s.size() - 1]` 则表示 `s` 的最后一个字符。

It is an error to use an index outside this range.

引用下标时如果超出下标作用范围就会引起溢出错误。

We could use the subscript operator to print each character in a `string` on a separate line:

可用下标操作符分别取出 `string` 对象的每个字符，分行输出：

```
string str("some string");
for (string::size_type ix = 0; ix != str.size(); ++ix)
    cout << str[ix] << endl;
```

On each trip through the loop we fetch the next character from `str`, printing it followed by a newline.

每次通过循环，就从 `str` 对象中读取下一个字符，输出该字符并换行。

Subscripting Yields an Lvalue

下标操作可用作左值

Recall that a variable is an lvalue ([Section 2.3.1](#), p. 45), and that the left-hand side of an assignment must be an lvalue. Like a variable, the value returned by the subscript operator is an lvalue. Hence, a subscript can be used on either side of an assignment. The following loop sets each character in `str` to an asterisk:

Section 3.2. Library string Type

前面说过，变量是左值（[2.3.1 节](#)），且赋值操作的左操作的必须是左值。和变量一样，`string` 对象的下标操作返回值也是左值。因此，下标操作可以放于赋值操作符的左边或右边。通过下面循环把 `str` 对象的每一个字符置为 '*'：

```
for (string::size_type ix = 0; ix != str.size(); ++ix)
    str[ix] = '*';
```

Computing Subscript Values

计算下标值

Any expression that results in an integral value can be used as the index to the subscript operator. For example, assuming `someval` and `someotherval` are integral objects, we could write

任何可产生整型值的表达式可用作下标操作符的索引。例如，假设 `someval` 和 `someotherval` 是两个整形对象，可以这样写：

```
str[someotherval * someval] = someval;
```

Although any integral type can be used as an index, the actual type of the index is `string::size_type`, which is an `unsigned` type.

虽然任何整型数值都可作为索引，但索引的实际数据类型却是类型 `unsigned` 类型 `string::size_type`。



The same reasons to use `string::size_type` as the type for a variable that holds the return from `size` apply when defining a variable to serve as an index. A variable used to index a `string` should have type `string::size_type`.

前面讲过，应该用 `string::size_type` 类型的变量接受 `size` 函数的返回值。在定义用作索引的变量时，出于同样的道理，`string` 对象的索引变量最好也用 `string::size_type` 类型。

When we subscript a `string`, we are responsible for ensuring that the index is "in range." By in range, we mean that the index is a number that, when assigned to a `size_type`, is a value in the range from 0 through the size of the `string` minus one. By using a `string::size_type` or another `unsigned` type as the index, we ensure that the subscript cannot be less than zero. As long as our index is an `unsigned` type, we need only check that it is less than the size of the `string`.

在使用下标索引 `string` 对象时，必须保证索引值“在上下界范围内”。“在上下界范围内”就是指索引值是一个赋值为 `size_type` 类型的值，其取值范围在 0 到 `string` 对象长度减 1 之间。使用 `string::size_type` 类型或其他 `unsigned` 类型，就只需要检测它是否小于 `string` 对象的长度。



The library is not required to check the value of the index. Using an index that is out of range is undefined and usually results in a serious run-time error.

标准库不要求检查索引值，所用索引的下标越界是没有定义的，这样往往会导致严重的运行时错误。

3.2.4. Dealing with the Characters of a `string`

3.2.4. `string` 对象中字符的处理

Often we want to process the individual characters of a `string`. For example, we might want to know if a particular character is a whitespace character or whether the character is alphabetic or numeric. [Table 3.3](#) on the facing page lists the functions that can be used on the characters in a `string` (or on any other `char` value). These functions are defined in the [ctype header](#).

我们经常要对 `string` 对象中的单个字符进行处理，例如，通常需要知道某个特殊字符是否为空白字符、字母或数字。[表 3.3](#) 列出了各种字符操作函数，适用于 `string` 对象的字符（或其他任何 `char` 值）。这些函数都在 [ctype 头文件](#) 中定义。

Table 3.3. ctype Functions

<code>isalnum(c)</code>	<code>True</code> if <code>c</code> is a letter or a digit. 如果 <code>c</code> 是字母或数字，则为 <code>True</code> 。
<code>isalpha(c)</code>	<code>true</code> if <code>c</code> is a letter. 如果 <code>c</code> 是字母，则为 <code>true</code> 。

Section 3.2. Library string Type

iscntrl(c)	true if c is a control character. 如果 c 是控制字符，则为 true。
isdigit(c)	true if c is a digit. 如果 c 是数字，则为 true。
isgraph(c)	true if c is not a space but is printable. 如果 c 不是空格，但可打印，则为 true。
islower(c)	true if c is a lowercase letter. 如果 c 是小写字母，则为 true。
isprint(c)	True if c is a printable character. 如果 c 是可打印的字符，则为 true。
ispunct(c)	True if c is a punctuation character. 如果 c 是标点符号，则 true。
isspace(c)	true if c is whitespace. 如果 c 是空白字符，则为 true。
isupper(c)	True if c is an uppercase letter. 如果 c 是大写字母，则 true。
isxdigit(c)	true if c is a hexadecimal digit. 如果是 c 十六进制数，则为 true。
tolower(c)	If c is an uppercase letter, returns its lowercase equivalent; otherwise returns c unchanged. 如果 c 大写字母，返回其小写字母形式，否则直接返回 c。
toupper(c)	If c is a lowercase letter, returns its uppercase equivalent; otherwise returns c unchanged. 如果 c 是小写字母，则返回其大写字母形式，否则直接返回 c。

These functions mostly test the given character and return an `int`, which acts as a truth value. Each function returns zero if the test fails; otherwise, they return a (meaningless) nonzero value indicating that the character is of the requested kind.

表中的大部分函数是测试一个给定的字符是否符合条件，并返回一个 `int` 作为真值。如果测试失败，则该函数返回 0，否则返回一个（无意义的）非 0，表示被测字符符合条件。

For these functions, a printable character is a character with a visible representation; whitespace is one of space, tab, vertical tab, return, newline, and formfeed; and punctuation is a printable character that is not a digit, a letter, or (printable) whitespace character such as space.

表中的这些函数，可打印的字符是指那些可以表示的字符，空白字符则是空格、制表符、垂直制表符、回车符、换行符和进纸符中的任意一种；标点符号则是除了数字、字母或（可打印的）空白字符（如空格）以外的其他可打印字符。

As an example, we could use these functions to print the number of punctuation characters in a given `string`:

这里给出一个例子，运用这些函数输出一给定 `string` 对象中标点符号的个数：

```
string s("Hello World!!!");  
string::size_type punct_cnt = 0;  
// count number of punctuation characters in s  
for (string::size_type index = 0; index != s.size(); ++index)  
    if (ispunct(s[index]))  
        ++punct_cnt;  
cout << punct_cnt  
    << " punctuation characters in " << s << endl;
```

The output of this program is

这个程序的输出结果是：

```
3 punctuation characters in Hello World!!!
```

Rather than returning a truth value, the `tolower` and `toupper` functions return a character either the argument unchanged or the lower- or uppercase version of the character. We could use `tolower` to change `s` to lowercase as follows:

和返回真值的函数不同的是，`tolower` 和 `toupper` 函数返回的是字符，返回实参字符本身或返回该字符相应的大小写字符。我们可以用 `tolower` 函数把 `string` 对象 `s` 中的字母改为小写字母，程序如下：

Section 3.2. Library string Type

```
// convert s to lowercase
for (string::size_type index = 0; index != s.size(); ++index)
    s[index] = tolower(s[index]);
cout << s << endl;
```

which generates

得到的结果为：

```
hello world!!!
```

Advice: Use the C++ Versions of C Library Headers

建议：采用 C 标准库头文件的 C++ 版本

In addition to facilities defined specifically for C++, the C++ library incorporates the C library. The `cctype` header makes available the C library functions defined in the C header file named `ctype.h`.

C++ 标准库除了定义了一些选定于 C++ 的设施外，还包括 C 标准库。C++ 中的头文件 `cctype` 其实就是利用了 C 标准库函数，这些库函数就定义在 C 标准库的 `ctype.h` 头文件中。

The standard C headers names use the form `name.h`. The C++ versions of these headers are named `cname`. The C++ versions remove the `.h` suffix and precede the `name` by the letter `c`. The `c` indicates that the header originally comes from the C library. Hence, `cctype` has the same contents as `ctype.h`, but in a form that is appropriate for C++ programs. In particular, the names defined in the `cname` headers are defined inside the `std` namespace, whereas those defined in the `.h` versions are not.

C 标准库头文件命名形式为 `name` 而 C++ 版本则命名为 `cname`，少了后缀 `.h` 而在头文件名前加了 `c` 表示这个头文件源自 C 标准库。因此，`cctype` 与 `ctype.h` 文件的内容是一样的，只是采用了更适合 C++ 程序的形式。特别地，`cname` 头文件中定义的名字都定义在命名空间 `std` 内，而 `.h` 版本中的名字却不是这样。

Ordinarily, C++ programs should use the `cname` versions of headers and not the `name.h` versions. That way names from the standard library are consistently found in the `std` namespace. Using the `.h` headers puts the burden on the programmer to remember which library names are inherited from C and which are unique to C++.

通常，C++ 程序中应采用 `cname` 这种头文件的版本，而不采用 `name.h` 版本，这样，标准库中的名字在命名空间 `std` 中保持一致。使用 `.h` 版本会给程序员带来负担，因为他们必须记得哪些标准库名字是从 C 继承来的，而哪些是 C++ 所特有的。

Exercises Section 3.2.4

Exercise 3.7: Write a program to read two `strings` and report whether the `strings` are equal. If not, report which of the two is the larger. Now, change the program to report whether the `strings` have the same length and if not report which is longer.

编一个程序读入两个 `string` 对象，测试它们是否相等。若不相等，则指出两个中哪个较大。接着，改写程序测试它们的长度是否相等，若不相等指出哪个较长。

Exercise 3.8: Write a program to read `strings` from the standard input, concatenating what is read into one large `string`. Print the concatenated `string`. Next, change the program to separate adjacent input `strings` by a space.

编一个程序，从标准输入读取多个 `string` 对象，把它们连接起来存放到一个更大的 `string` 对象中。并输出连接后的 `string` 对象。接着，改写程序，将连接后相邻 `string` 对象以空格隔开。

Exercise 3.9: What does the following program do? Is it valid? If not, why not?
下列程序实现什么功能？实现合法？如果不合法，说明理由。

```
string s;
cout << s[0] << endl;
```

Exercise 3.10: Write a program to strip the punctuation from a `string`. The input to the program should be a `string` of characters including punctuation; the output should be a `string` in which the punctuation is removed.

编一个程序，从 `string` 对象中去掉标点符号。要求输入到程序的字符串必须含有标点符号，输出结果则是去掉标点符号后的 `string` 对象。

Team LiB

◀ PREVIOUS NEXT ▶

3.3. Library `vector` Type

3.3. 标准库 `vector` 类型

A `vector` is a collection of objects of a single type, each of which has an associated integer index. As with `strings`, the library takes care of managing the memory associated with storing the elements. We speak of a `vector` as a container because it contains other objects. All of the objects in a container must have the same type. We'll have much more to say about containers in [Chapter 9](#).

`vector` 是同一种类型的对象的集合，每个对象都有一个对应的整数索引值。和 `string` 对象一样，标准库将负责管理与存储元素相关的内存。我们把 `vector` 称为容器，是因为它可以包含其他对象。一个容器中的所有对象都必须是同一种类型的。我们将在[第九章](#)更详细地介绍容器。

To use a `vector`, we must include the appropriate header. In our examples, we also assume an appropriate `using` declaration is made:

使用 `vector` 之前，必须包含相应的头文件。本书给出的例子，都是假设已作了相应的 `using` 声明：

```
#include <vector>
using std::vector;
```

A `vector` is a class template. Templates let us write a single class or function definition that can be used on a variety of types. Thus, we can define a `vector` that holds `strings`, or a `vector` to hold `ints`, or one to hold objects of our own class types, such as `Sales_items`. We'll see how to define our own class templates in [Chapter 16](#). Fortunately, we need to know very little about how templates are defined in order to use them.

`vector` 是一个类模板 (class template)。使用模板可以编写一个类定义或函数定义，而用于多个不同的数据类型。因此，我们可以定义保存 `string` 对象的 `vector`，或保存 `int` 值的 `vector`，又或是保存自定义的类类型对象（如 `Sales_items` 对象）的 `vector`。将在[第十六章](#)介绍如何定义程序员自己的类模板。幸运的是，使用类模板时只需要简单了解类模板是如何定义的就可以了。

To declare objects of a type generated from a class template, we must supply additional information. The nature of this information depends on the template. In the case of `vector`, we must say what type of objects the `vector` will contain. We specify the type by putting it between a pair of angle brackets following the template's name:

声明从类模板产生的某种类型的对象，需要提供附加信息，信息的种类取决于模板。以 `vector` 为例，必须说明 `vector` 保存何种对象的类型，通过将类型放在类型放在类模板名称后面的尖括号中来指定类型：

```
vector<int> ivec;           // ivec holds objects of type int
vector<Sales_item> Sales_vec; // holds Sales_item
```

As in any variable definition, we specify a type and a list of one or more variables. In the first of these definitions, the type is `vector<int>`, which is a `vector` that holds objects of type `int`. The name of the variable is `ivec`. In the second, we define `Sales_vec` to hold `Sales_item` objects.

和其他变量定义一样，定义 `vector` 对象要指定类型和一个变量的列表。上面的第一个定义，类型是 `vector<int>`，该类型即是含有若干 `int` 类型对象的 `vector`，变量名为 `ivec`。第二个定义的变量名是 `Sales_vec`，它所保存的元素是 `Sales_item` 类型的对象。



`vector` is not a type; it is a template that we can use to define any number of types. Each of `vector` type specifies an element type. Hence, `vector<int>` and `vector<string>` are types.

`vector` 不是一种数据类型，而只是一个类模板，可用来定义任意多种数据类型。`vector` 类型的每一种都指定了其保存元素的类型。因此，`vector<int>` 和 `vector<string>` 都是数据类型。

3.3.1. Defining and Initializing `vector`s

3.3.1. `vector` 对象的定义和初始化

The `vector` class defines several constructors ([Section 2.3.3](#), p. 49), which we use to define and initialize `vector` objects. The constructors are listed in [Table 3.4](#).

`vector` 类定义了好几种构造函数 ([2.3.3 节](#))，用来定义和初始化 `vector` 对象。[表 3.4](#) 列出了这些构造函数。

Table 3.4. Ways to Initialize a `vector`

<code>vector<T> v1;</code>	<code>vector</code> that holds objects of type <code>T</code> ; Default constructor <code>v1</code> is empty <code>vector</code> 保存类型为 <code>T</code> 对象。 默认构造函数 <code>v1</code> 为空。
<code>vector<T> v2(v1);</code>	<code>v2</code> is a copy of <code>v1</code> <code>v2</code> 是 <code>v1</code> 的一个副本。
<code>vector<T> v3(n, i);</code>	<code>v3</code> has <code>n</code> elements with value <code>i</code> <code>v3</code> 包含 <code>n</code> 个值为 <code>i</code> 的元素。
<code>vector<T> v4(n);</code>	<code>v4</code> has <code>n</code> copies of a value-initialized object <code>v4</code> 含有值初始化的元素的 <code>n</code> 个副本。

Creating a Specified Number of Elements

创建确定个数的元素

When we create a `vector` that is not empty, we must supply value(s) to use to initialize the elements. When we copy one `vector` to another, each element in the new `vector` is initialized as a copy of the corresponding element in the original `vector`. The two `vectors` must hold the same element type:

若要创建非空的 `vector` 对象，必须给出初始化元素的值。当把一个 `vector` 对象复制到另一个 `vector` 对象时，新复制的 `vector` 中每一个元素都初始化为原 `vectors` 中相应元素的副本。但这两个 `vector` 对象必须保存同一种元素类型：

```
vector<int> ivec1;           // ivec1 holds objects of type int
vector<int> ivec2(ivec1);    // ok: copy elements of ivec1 into ivec2
vector<string> svec(ivec1); // error: svec holds strings, not ints
```

We can initialize a `vector` from a count and an element value. The constructor uses the count to determine how many elements the `vector` should have and uses the value to specify the value each of those elements will have:

可以用元素个数和元素值对 `vector` 对象进行初始化。构造函数用元素个数来决定 `vector` 对象保存元素的个数，元素值指定每个元素的初始值：

```
vector<int> ivec4(10, -1);      // 10 elements, each initialized to -1
vector<string> svec(10, "hi!"); // 10 strings, each initialized to "hi!"
```

Key Concept: `vector`s Grow Dynamically

关键概念：`vector` 对象动态增长

A central property of `vectors` (and the other library containers) is that they are required to be implemented so that it is efficient to add elements to them at run time. Because `vectors` grow efficiently, it is usually best to let the `vector` grow by adding elements to it dynamically as the element values are known.

`vector` 对象（以及其他标准库容器对象）的重要属性就在于可以在运行时高效地添加元素。因为 `vector` 增长的效率高，在元素值已知的情况下，最好是动态地添加元素。

As we'll see in [Chapter 4](#), this behavior is distinctly different from that of built-in arrays in C and for that matter in most other languages. In particular, readers accustomed to using C or Java might expect that because `vector` elements are stored contiguously, it would be best to preallocate the `vector` at its expected size. In fact, the contrary is the case, for reasons we'll explore in [Chapter 9](#).

Section 3.3. Library vector Type

正如[第四章](#)将介绍的，这种增长方式不同于 C 语言中的内置数据类型，也不同于大多数其他编程语言的数据类型。具体而言，如果读者习惯了 C 或 Java 的风格，由于 `vector` 元素连续存储，可能希望最好是预先分配合适的空间。但事实上，为了达到连续性，C++ 的做法恰好相反，具体原因将在[第九章](#)探讨。



Although we can preallocate a given number of elements in a `vector`, it is usually more efficient to define an empty `vector` and add elements to it (as we'll learn how to do shortly).

虽然可以对给定元素个数的 `vector` 对象预先分配内存，但更有效的方法是先初始化一个空 `vector` 对象，然后再动态地增加元素（我们随后将学习如何进行这样的操作）。

Value Initialization

值初始化

When we do not specify an element initializer, then the library creates a [value initialized](#) element initializer for us. This library-generated value is used to initialize each element in the container. The value of the element initializer depends on the type of the elements stored in the `vector`.

如果没有指定元素的初始化式，那么标准库将自行提供一个元素初始值进行[值初始化 \(value initialization\)](#)。这个由库生成的初始值将用来初始化容器中的每个元素，具体值为何，取决于存储在 `vector` 中元素的数据类型。

If the `vector` holds elements of a built-in type, such as `int`, then the library creates an element initializer with a value of 0:

如果 `vector` 保存内置类型（如 `int` 类型）的元素，那么标准库将用 0 值创建元素初始化式：

```
vector<string> fvec(10); // 10 elements, each initialized to 0
```

If the `vector` holds elements of a class type, such as `string`, that defines its own constructors, then the library uses the value type's default constructor to create the element initializer:

如果 `vector` 保存的是含有构造函数的类类型（如 `string`）的元素，标准库将用该类型的默认构造函数创建元素初始化式：

```
vector<string> svec(10); // 10 elements, each an empty string
```



As we'll see in [Chapter 12](#), some classes that define their own constructors do not define a default constructor. We cannot initialize a `vector` of such a type by specifying only a size; we must also specify an initial element value.

[第十二章](#)将介绍一些有自定义构造函数但没有默认构造函数的类，在初始化这种类型的 `vector` 对象时，程序员就不能仅提供元素个数，还需要提供元素初始值。

There is a third possibility: The element type might be of a class type that does not define any constructors. In this case, the library still creates a value-initialized object. It does so by value-initializing each member of that object.

还有第三种可能性：元素类型可能是没有定义任何构造函数的类类型。这种情况下，标准库仍产生一个带初始值的对象，这个对象的每个成员进行了值初始化。

Exercises Section 3.3.1

Exercise

3.11: Which, if any, of the following `vector` definitions are in error?

下面哪些 `vector` 定义不正确？

(a) `vector< vector<int> > ivec;`

Section 3.3. Library vector Type

```
(b) vector<string> svec = ivec;
(c) vector<string> svec(10, "null");
```

Exercise 3.12: How many elements are there in each of the following `vector`s? What are the values of the elements?

下列每个 `vector` 对象中元素个数是多少? 各元素的值是什么?

```
(a) vector<int> ivec1;
(b) vector<int> ivec2(10);
(c) vector<int> ivec3(10, 42);
(d) vector<string> svec1;
(e) vector<string> svec2(10);
(f) vector<string> svec3(10, "hello");
```

3.3.2. Operations on `vector`s

3.3.2. `vector` 对象的操作

The `vector` library provides various operations, many of which are similar to operations on `strings`. [Table 3.5](#) lists the most important `vector` operations.

`vector` 标准库提供了许多类似于 `string` 对象的操作, [表 3.5](#) 列出了几种最重要的 `vector` 操作。

Table 3.5. `vector` Operations

<code>v.empty()</code>	Returns <code>true</code> if <code>v</code> is empty; otherwise returns <code>false</code> 如果 <code>v</code> 为空, 则返回 <code>true</code> , 否则返回 <code>false</code> 。
<code>v.size()</code>	Returns number of elements in <code>v</code> 返回 <code>v</code> 中元素的个数。
<code>v.push_back(t)</code>	Adds element with value <code>t</code> to end of <code>v</code> 在 <code>v</code> 的末尾增加一个值为 <code>t</code> 的元素。
<code>v[n]</code>	Returns element at position <code>n</code> in <code>v</code> 返回 <code>v</code> 中位置为 <code>n</code> 的元素。
<code>v1 = v2</code>	Replaces elements in <code>v1</code> by a copy of elements in <code>v2</code> 把 <code>v1</code> 的元素替换为 <code>v2</code> 中元素的副本。
<code>v1 == v2</code>	Returns <code>true</code> if <code>v1</code> and <code>v2</code> are equal 如果 <code>v1</code> 与 <code>v2</code> 相等, 则返回 <code>true</code> 。
<code>!=, <, <=,</code> <code>>, and >=</code>	Have their normal meanings 保持这些操作符惯有的含义。

The `size` of a `vector`

`vector` 对象的 `size`

The `empty` and `size` operations are similar to the corresponding `string` operations ([Section 3.2.3](#), p. 83). The `size` member returns a value of the `size_type` defined by the corresponding `vector` type.

Section 3.3. Library vector Type

`empty` 和 `size` 操作类似于 `string` 的相关操作 ([3.2.3 节](#))。成员函数 `size` 返回相应 `vector` 类定义的 `size_type` 的值。



To use `size_type`, we must name the type in which it is defined. A `vector` type *always* includes the element type of the `vector`:

```
vector<int>::size_type      // ok  
vector::size_type           // error
```

Adding Elements to a `vector`

向 `vector` 添加元素

The `push_back` Operation

`push_back` operation takes an element value and adds that value as a new element at the back of a `vector`. In effect it "pushes" an element onto the "back" of the `vector`:

`push_back` 操作接受一个元素值，并将它作为一个新的元素添加到 `vector` 对象的后面，也就是“插入 (push) ”到 `vector` 对象的“后面 (back) ”：

```
// read words from the standard input and store them as elements in a vector  
string word;  
vector<string> text;    // empty vector  
while (cin >> word) {  
    text.push_back(word);    // append word to text  
}
```

This loop reads a sequence of `strings` from the standard input, appending them one at a time onto the back of the `vector`. We start by defining `text` as an initially empty `vector`. Each trip through the loop adds a new element to the `vector` and gives that element the value of whatever word was read from the input. When the loop completes, `text` will have as many elements as were read.

该循环从标准输入读取一系列 `string` 对象，逐一追加到 `vector` 对象的后面。首先定义一个空的 `vector` 对象 `text`。每循环一次就添加一个新元素到 `vector` 对象，并将从输入读取的 `word` 值赋予该元素。当循环结束时，`text` 就包含了所有读入的元素。

Subscripting a `vector`

`vector` 的下标操作

Objects in the `vector` are not named. Instead, they can be accessed by their position in the `vector`. We can fetch an element using the subscript operator. Subscripting a `vector` is similar to subscripting a `string` ([Section 3.2.3, p. 87](#)).

`vector` 中的对象是没有命名的，可以按 `vector` 中对象的位置来访问它们。通常使用下标操作符来获取元素。`vector` 的下标操作类似于 `string` 类型的下标操作 ([3.2.3 节](#))。.

The `vector` subscript operator takes a value and returns the element at that position in the `vector`. Elements in a `vector` are numbered beginning with 0. The following example uses a `for` loop to reset each element in the `vector` to 0:

`vector` 的下标操作符接受一个值，并返回 `vector` 中该对应位置的元素。`vector` 元素的位置从 0 开始。下例使用 `for` 循环把 `vector` 中的每个元素值都重置为 0：

```
// reset the elements in the vector to zero  
for (vector<int>::size_type ix = 0; ix != ivec.size(); ++ix)  
    ivec[ix] = 0;
```

Like the `string` subscript operator, the `vector` subscript yields an lvalue so that we may write to it, which we do in the body of the loop. Also, as we do for `strings`, we use the `size_type` of the `vector` as the type for the subscript.

和 `string` 类型的下标操作符一样，`vector` 下标操作的结果为左值，因此可以像循环体中所做的那样实现写入。另外，和 `string` 对象的下标操作类似，这里用 `size_type`

Section 3.3. Library vector Type

类型作为 `vector` 下标的类型。



Even if `ivec` is empty, this `for` loop executes correctly. If `ivec` is empty, the call to `size` returns 0 and the test in the `for` compares `ix` to 0. Because `ix` is itself 0 on the first trip, the test would fail and the loop body would not be executed even once.

在上例中，即使 `ivec` 为空，`for` 循环也会正确执行。`ivec` 为空则调用 `size` 返回 0，并且 `for` 中的测试比较 `ix` 和 0。第一次循环时，由于 `ix` 本身就是 0 就是 0，则条件测试失败，`for` 循环体一次也不执行。

Subscripting Does Not Add Elements

下标操作不添加元素

Programmers new to C++ sometimes think that subscripting a `vector` adds elements; it does not:

初学 C++ 的程序员可能会认为 `vector` 的下标操作可以添加元素，其实不然：

```
vector<int> ivec; // empty vector
for (vector<int>::size_type ix = 0; ix != 10; ++ix)
    ivec[ix] = ix; // disaster: ivec has no elements
```

Key Concept: Safe, Generic Programming

关键概念：安全的泛型编程

Programmers coming to C++ from C or Java might be surprised that our loop used `!=` rather than `<` to test the index against the `size` of the `vector`. C programmers are probably also surprised that we call the `size` member in the `for` rather than calling it once before the loop and remembering its value.

习惯于 C 或 Java 编程的 C++ 程序员可能会觉得难以理解，`for` 循环的判断条件用 `!=` 而不是用 `<` 来测试 `vector` 下标值是否越界。C 程序员难以理解的还有，上例中没有在 `for` 循环之前就调用 `size` 成员函数并保存其返回的值，而是在 `for` 语句头中调用 `size` 成员函数。

C++ programmers tend to write loops using `!=` in preference to `<` as a matter of habit. In this case, there is no particular reason to choose one operator or the other. We'll understand the rationale for this habit once we cover generic programming in [Part II](#).

C++ 程序员习惯于优先选用 `!=` 而不是 `<` 来编写循环判断条件。在上例中，选用或不用某种操作符并没有特别的取舍理由。学习完本书[第二部分](#)的泛型编程后，你将会明白这种习惯的合理性。

Calling `size` rather than remembering its value is similarly unnecessary in this case but again reflects a good habit. In C++, data structures such as `vector` can grow dynamically. Our loop only reads elements; it does not add them. However, a loop could easily add new elements. If the loop did add elements, then testing a saved value of `size` would fail our loop would not account for the newly added elements. Because a loop might add elements, we tend to write our loops to test the current `size` on each pass rather than store a copy of what the `size` was when we entered the loop.

调用 `size` 成员函数而不保存它返回的值，在这个例子中同样不是必需的，但这反映了一种良好的编程习惯。在 C++ 中，有些数据结构（如 `vector`）可以动态增长。上例中循环仅需要读取元素，而不需要增加新的元素。但是，循环可以容易地增加新元素，如果确实增加了新元素的话，那么测试已保存的 `size` 值作为循环的结束条件就会有问题，因为没有将新加入的元素计算在内。所以我们倾向于在每次循环中测试 `size` 的当前值，而不是在进入循环前，存储 `size` 值的副本。

As we'll see in [Chapter 7](#), in C++ functions can be declared to be `inline`. When it can do so, the compiler will expand the code for an `inline` function directly rather than actually making a function call. Tiny library functions such as `size` are almost surely defined to be `inline`, so we expect that there is little run-time cost in making this call on each trip through the loop.

我们将在[第七章](#)学习到，C++ 中有些函数可以声明为内联（`inline`）函数。编译器遇到内联函数时就会直接扩展相应代码，而不是进行实际的函数调用。像 `size` 这样的小库函数几乎都定义为内联函数，所以每次循环过程中调用它的运行时代价是比较小的。

This code intended to insert new 10 elements into `ivec`, giving the elements the values from 0 through 9. However, `ivec` is an empty `vector` and

Section 3.3. Library vector Type

subscripts can only be used to fetch existing elements.

上述程序试图在 `ivec` 中插入 10 个新元素，元素值依次为 0 到 9 的整数。但是，这里 `ivec` 是空的 `vector` 对象，而且下标只能用于获取已存在的元素。

The right way to write this loop would be

这个循环的正确写法应该是：

```
for (vector<int>::size_type ix = 0; ix != 10; ++ix)
    ivec.push_back(ix); // ok: adds new element with value ix
```



An element must exist in order to subscript it; elements are *not* added when we assign through a subscript.

必须是已存在的元素才能用下标操作符进行索引。通过下标操作进行赋值时，不会添加任何元素。

Caution: Only Subscript Elements that Are Known to Exist!

警告：仅能对确知已存在的元素进行下标操作

It is crucially important to understand that we may use the subscript operator, (the `[]` operator), to fetch only elements that actually exist. For example,

对于下标操作符（`[]` 操作符）的使用有一点非常重要，就是仅能提取确实已存在的元素，例如：

```
vector<int> ivec;           // empty vector
cout << ivec[0];           // Error: ivec has no elements!

vector<int> ivec2(10);     // vector with 10 elements
cout << ivec2[10];         // Error: ivec has elements 0...9
```

Attempting to fetch an element that doesn't exist is a run-time error. As with most such errors, there is no assurance that the implementation will detect it. The result of executing the program is uncertain. The effect of fetching a nonexistent element is undefinedwhat happens will vary by implementation, but the program will almost surely fail in some interesting way at run time.

试图获取不存在的元素必须产生运行时错误。和大多数同类错误一样，不能确保执行过程可以捕捉到这类错误，运行程序的结果是不确定的。由于取不存在的元素的结果标准没有定义，因而不同的编译器实现会导致不同的结果，但程序运行时几乎肯定会以某种有趣的方式失败。

This caution applies any time we use a subscript, such as when subscripting a `string` and, as we'll see shortly, when subscripting a built-in array.

本警告适用于任何使用下标操作的时候，如 `string` 类型的下标操作，以及将要简要介绍的内置数组的下标操作。

Attempting to subscript elements that do not exist is, unfortunately, an extremely common and pernicious programming error. So-called "buffer overflow" errors are the result of subscripting elements that don't exist. Such bugs are the most common cause of security problems in PC and other applications.

不幸的是，试图对不存在的元素进行下标操作是程序设计过程中经常会犯的严重错误。所谓的“缓冲区溢出”错误就是对不存在的元素进行下标操作的结果。这样的缺陷往往导致 **PC** 机和其他应用中最常见的安全问题。

Exercises Section 3.3.2

Section 3.3. Library vector Type

Exercise Read a set of integers into a `vector`. Calculate and print the sum of each pair of adjacent elements in the `vector`. If there is an odd number, tell the user and print the value of the last element without summing it. Now change your program so that it prints the sum of the first and last elements, followed by the sum of the second and second-to-last and so on.

读一组整数到 `vector` 对象，计算并输出每对相邻元素的和。如果读入元素个数为奇数，则提示用户最后一个元素没有求和，并输出其值。然后修改程序：头尾元素两两配对（第一个和最后一个，第二个和倒数第二个，以此类推），计算每对元素的和，并输出。

Exercise Read some text into a `vector`, storing each word in the input as an element in the `vector`.
3.14: transform each word into uppercase letters. Print the transformed elements from the `vector`, printing eight words to a line.

读入一段文本到 `vector` 对象，每个单词存储为 `vector` 中的一个元素。把 `vector` 对象中每个单词转化为大写字母。输出 `vector` 对象中转化后的元素，每八个单词为一行输出。

Exercise Is the following program legal? If not, how might you fix it?
3.15: 下面程序合法吗？如果不合法，如何更正？

```
vector<int> ivec;
ivec[0] = 42;
```

Exercise List three ways to define a `vector` and give it 10 elements, each with the value 42. Indicate
3.16: whether there is a preferred way to do so and why.

列出三种定义 `vector` 对象的方法，给定 10 个元素，每个元素值为 42。指出是否还有更好的实现方法，并说明为什么。

3.4. Introducing Iterators

3.4. 迭代器简介

While we can use subscripts to access the elements in a `vector`, the library also gives us another way to examine elements: We can use an **iterator**. An iterator is a type that lets us examine the elements in a container and navigate from one element to another.

除了使用下标来访问 `vector` 对象的元素外，标准库还提供了另一种访问元素的方法：使用[迭代器 \(iterator\)](#)。迭代器是一种检查容器内元素并遍历元素的数据类型。

The library defines an iterator type for each of the standard containers, including `vector`. Iterators are more general than subscripts: All of the library containers define iterator types, but only a few of them support subscripting. Because iterators are common to all containers, modern C++ programs tend to use iterators rather than subscripts to access container elements, even on types such as `vector` that support subscripting.

标准库为每一种标准容器（包括 `vector`）定义了一种迭代器类型。迭代器类型提供了比下标操作更通用化的方法：所有的标准库容器都定义了相应的迭代器类型，而只有少数的容器支持下标操作。因为迭代器对所有的容器都适用，现代 C++ 程序更倾向于使用迭代器而不是下标操作访问容器元素，即使对支持下标操作的 `vector` 类型也是这样。

The details of how iterators work are discussed in [Chapter 11](#), but we can use them without understanding them in their full complexity.

[第十一章](#)将详细讨论迭代器的工作原理，但使用迭代器并不需要完全了解它复杂的实现细节。

Container **iterator** Type

容器的 **iterator** 类型

Each of the container types, such as `vector`, defines its own iterator type:

每种容器类型都定义了自己的迭代器类型，如 `vector`:

```
vector<int>::iterator iter;
```

This statement defines a variable named `iter`, whose type is the type named `iterator` defined by `vector<int>`. Each of the library container types defines a member named `iterator` that is a synonym for the actual type of its iterator.

这行语句定义了一个名为 `iter` 的变量，它的数据类型是 `vector<int>` 定义的 `iterator` 类型。每个标准库容器类型都定义了一个名为 `iterator` 的成员，这里的 `iterator` 与迭代器实际类型的含义相同。

Terminology: Iterators and Iterator Types

术语：迭代器和迭代器类型

When first encountered, the nomenclature around iterators can be confusing. In part the confusion arises because the same term, `iterator`, is used to refer to two things. We speak generally of the concept of an iterator, and we speak specifically of a concrete `iterator` type defined by a container, such as `vector<int>`.

程序员首次遇到有关迭代器的术语时可能会困惑不解，原因之一是由于同一个术语 `iterator` 往往表示两个不同的事物。一般意义上指的是迭代器的概念；而具体而言时指的则是由容器定义的具体的 `iterator` 类型，如 `vector<int>`。

What's important to understand is that there is a collection of types that serve as iterators. These types are related conceptually. We refer to a type as an iterator if it supports a certain set of actions. Those actions let us navigate among the elements of a container and let us access the value of those elements.

重点要理解的是，有许多用作迭代器的类型，这些类型在概念上是相关的。若一种类型支持一组确定的操作（这些操作可用来遍历容器内的元素，并访问这些元素的值），我们就称这种类型为迭代器。

Each container class defines its own `iterator` type that can be used to access the elements in the container. That is, each container defines a type named `iterator`, and that type supports the actions of an (conceptual) iterator.

各容器类都定义了自己的 `iterator` 类型，用于访问容器内的元素。换句话说，每个容器都定义了一个名为 `iterator` 的类型，而这种类型支持（概念上的）迭代器的各种操作。

The `begin` and `end` Operations

`begin` 和 `end` 操作

Each container defines a pair of functions named `begin` and `end` that return iterators. The iterator returned by `begin` refers to the first element, if any, in the container:

每种容器都定义了一对命名为 `begin` 和 `end` 的函数，用于返回迭代器。如果容器中有元素的话，由 `begin` 返回的迭代器指向第一个元素：

```
vector<int>::iterator iter = ivec.begin();
```

This statement initializes `iter` to the value returned by the `vector` operation named `begin`. Assuming the `vector` is not empty, after this initialization, `iter` refers to the same element as `ivec[0]`.

上述语句把 `iter` 初始化为由名为 `vector` 操作返回的值。假设 `vector` 不空，初始化后，`iter` 即指该元素为 `ivec[0]`。

The iterator returned by the `end` operation is an iterator positioned "one past the end" of the `vector`. It is often referred to as the [off-the-end iterator](#) indicating that it refers to a nonexistent element "off the end" of the `vector`. If the `vector` is empty, the iterator returned by `begin` is the same as the iterator returned by `end`.

由 `end` 操作返回的迭代器指向 `vector` 的“末端元素的下一个”。[“超出末端迭代器”（off-the-end iterator）](#)。表明它指向了一个不存在的元素。如果 `vector` 为空，`begin` 返回的迭代器与 `end` 返回的迭代器相同。



The iterator returned by the `end` operation does not denote an actual element in the `vector`. Instead, it is used as a [sentinel](#) indicating when we have processed all the elements in the `vector`.

由 `end` 操作返回的迭代器并不指向 `vector` 中任何实际的元素，相反，它只是起一个[哨兵（sentinel）](#)的作用，表示我们已处理完 `vector` 中所有元素。

Dereference and Increment on `vector` Iterators

`vector` 迭代器的自增和解引用运算

The operations on iterator types let us retrieve the element to which an iterator refers and let us move an iterator from one element to another. 迭代器类型定义了一些操作来获取迭代器所指向的元素，并允许程序员将迭代器从一个元素移动到另一个元素。

Iterator types use the **dereference operator** (the `*` operator) to access the element to which the iterator refers:

迭代器类型可使用解引用操作符 (**dereference operator**) (`*`) 来访问迭代器所指向的元素：

```
*iter = 0;
```

The dereference operator returns the element that the iterator currently denotes. Assuming `iter` refers to the first element of the `vector`, then `*iter` is the same element as `ivec[0]`. The effect of this statement is to assign 0 to that element.

解引用操作符返回迭代器当前所指向的元素。假设 `iter` 指向 `vector` 对象 `ivec` 的第一元素，那么 `*iter` 和 `ivec[0]` 就是指向同一个元素。上面这个语句的效果就是把这个元素的值赋为 0。

Iterators use the increment operator (`++`) ([Section 1.4.1](#), p. 13) to advance an iterator to the next element in the container. Incrementing an iterator is a logically similar operation to the increment operator when applied to `int` objects. In the case of `ints`, the effect is to "add one" to the `int`'s value. In the case of iterators, the effect is to "advance the iterator by one position" in the container. So, if `iter` refers to the first element, then `++iter` denotes the second element.

迭代器使用自增操作符 ([1.4.1 节](#)) 向前移动迭代器指向容器中下一个元素。从逻辑上说，迭代器的自增操作和 `int` 型对象的自增操作类似。对 `int` 对象来说，操作结果就是把 `int` 型值“加 1”，而对迭代器对象则是把容器中的迭代器“向前移动一个位置”。因此，如果 `iter` 指向第一个元素，则 `++iter` 指向第二个元素。

Section 3.4. Introducing Iterators



Because the iterator returned from `end` does not denote an element, it may not be incremented or dereferenced.

由于 `end` 操作返回的迭代器不指向任何元素，因此不能对它进行解引用或自增操作。

Other Iterator Operations

迭代器的其他操作

Another pair of useful operations that we can perform on iterators is comparison: Two iterators can be compared using either `==` or `!=`. Iterators are equal if they refer to the same element; they are unequal otherwise.

另一对可执行于迭代器的操作就是比较：用 `==` 或 `!=` 操作符来比较两个迭代器，如果两个迭代器对象指向同一个元素，则它们相等，否则就不相等。

A Program that Uses Iterators

迭代器应用的程序示例

Assume we had a `vector<int>` named `ivec` and we wanted to reset each of its elements to zero. We might do so by using a subscript:

假设已声明了一个 `vector<int>` 型的 `ivec` 变量，要把它所有元素值重置为 0，可以用下标操作来完成：

```
// reset all the elements in ivec to 0
for (vector<int>::size_type ix = 0; ix != ivec.size(); ++ix)
    ivec[ix] = 0;
```

This program uses a `for` loop to iterate through the elements in `ivec`. The `for` defines an index, which it increments on each iteration. The body of the `for` sets each element in `ivec` to zero.

上述程序用 `for` 循环遍历 `ivec` 的元素，`for` 循环定义了一个索引 `ix`，每循环迭代一次 `ix` 就自增 1。`for` 循环体将 `ivec` 的每个元素赋值为 0。

A more typical way to write this loop would use iterators:

更典型的做法是用迭代器来编写循环：

```
// equivalent loop using iterators to reset all the elements in ivec to 0
for (vector<int>::iterator iter = ivec.begin();
     iter != ivec.end(); ++iter)
    *iter = 0; // set element to which iter refers to 0
```

The `for` loop starts by defining `iter` and initializing it to refer to the first element in `ivec`. The condition in the `for` tests whether `iter` is unequal to the iterator returned by the `end` operation. Each iteration increments `iter`. The effect of this `for` is to start with the first element in `ivec` and process in sequence each element in the `vector`. Eventually, `iter` will refer to the last element in `ivec`. After we process the last element and increment `iter`, it will become equal to the value returned by `end`. At that point, the loop stops.

`for` 循环首先定义了 `iter`，并将它初始化为指向 `ivec` 的第一个元素。`for` 循环的条件测试 `iter` 是否与 `end` 操作返回的迭代器不等。每次迭代 `iter` 都自增 1，这个 `for` 循环的效果是从 `ivec` 第一个元素开始，顺序处理 `vector` 中的每一元素。最后，`iter` 将指向 `ivec` 中的最后一个元素，处理完最后一个元素后，`iter` 再增加 1，就会与 `end` 操作的返回值相等，在这种情况下，循环终止。

The statement in the `for` body uses the dereference operator to access the value of the current element. As with the subscript operator, the value returned by the dereference operator is an lvalue. We can assign to this element to change its value. The effect of this loop is to assign the value zero to each element in `ivec`.

`for` 循环体内的语句用解引用操作符来访问当前元素的值。和下标操作符一样，解引用操作符的返回值是一个左值，因此可以对它进行赋值来改变它的值。上述循环的效果就是把 `ivec` 中所有元素都赋值为 0。

Having walked through the code in detail, we can see that this program has exactly the same effect as the version that used subscripts: We start at the first element in the `vector` and set each element in the `vector` to zero.

通过上述对代码的详细分析，可以看出这段程序与用下标操作符的版本达到相同的操作效果：从 `vector` 的第一个元素开始，把 `vector` 中每个元素都置为 0。

Section 3.4. Introducing Iterators



This program, like the one on page 94, is safe if the `vector` is empty. If `ivec` is empty, then the iterator returned from `begin` does not denote any element; it can't, because there are no elements. In this case, the iterator returned from `begin` is the same as the one returned from `end`, so the test in the `for` fails immediately.

本节给出的例子程序和 3.3.2 节 `vector` 的下标操作的程序一样，如果 `vector` 为空，程序是安全的。如果 `ivec` 为空，则 `begin` 返回的迭代器不指向任何元素——由于没有元素，所以它不能指向任何元素。在这种情况下，从 `begin` 操作返回的迭代器与从 `end` 操作返回的迭代器的值相同，因此 `for` 语句中的测试条件立即失败。

const_iterator

The previous program used a `vector::iterator` to change the values in the `vector`. Each container type also defines a type named `const_iterator`, which should be used when reading, but not writing to, the container elements.

前面的程序用 `vector::iterator` 改变 `vector` 中的元素值。每种容器类型还定义了一种名为 `const_iterator` 的类型，该类型只能用于读取容器内元素，但不能改变其值。

When we dereference a plain `iterator`, we get a `nonconst` reference (Section 2.5, p. 59) to the element. When we dereference a `const_iterator`, the value returned is a reference to a `const` (Section 2.4, p. 56) object. Just as with any `const` variable, we may not write to the value of this element.

当我们对普通 `iterator` 类型解引用时，得到对某个元素的非 `const` (2.5 节)。而如果我们对 `const_iterator` 类型解引用时，则可以得到一个指向 `const` 对象的引用 (2.4 节)，如同任何常量一样，该对象不能进行重写。

For example, if `text` is a `vector<string>`, we might want to traverse it, printing each element. We could do so as follows:

例如，如果 `text` 是 `vector<string>` 类型，程序员想要遍历它，输出每个元素，可以这样编写程序：

```
// use const_iterator because we won't change the elements
for (vector<string>::const_iterator iter = text.begin();
     iter != text.end(); ++iter)
    cout << *iter << endl; // print each element in text
```

This loop is similar to the previous one, except that we are reading the value from the iterator, not assigning to it. Because we read, but do not write, through the iterator, we define `iter` to be a `const_iterator`. When we dereference a `const_iterator`, the value returned is `const`. We may not assign to an element using a `const_iterator`:

除了是从迭代器读取元素值而不是对它进行赋值之外，这个循环与前一个相似。由于这里只需要借助迭代器进行读，不需要写，这里把 `iter` 定义为 `const_iterator` 类型。当对 `const_iterator` 类型解引用时，返回的是一个 `const` 值。不允许用 `const_iterator` 进行赋值

```
for (vector<string>::const_iterator iter = text.begin();
     iter != text.end(); ++ iter)
    *iter = " "; // error: *iter is const
```

When we use the `const_iterator` type, we get an iterator whose own value can be changed but that cannot be used to change the underlying element value. We can increment the iterator and use the dereference operator to read a value but not to assign to that value.

使用 `const_iterator` 类型时，我们可以得到一个迭代器，它自身的值可以改变，但不能用来改变其所指向的元素的值。可以对迭代器进行自增以及使用解引用操作符来读取值，但不能对该元素赋值。

A `const_iterator` should not be confused with an `iterator` that is `const`. When we declare an iterator as `const` we must initialize the iterator. Once it is initialized, we may not change its value:

不要把 `const_iterator` 对象与 `const` 的 `iterator` 对象混淆起来。声明一个 `const` 迭代器时，必须初始化迭代器。一旦被初始化后，就不能改变它的值：

```
vector<int> nums(10); // nums is nonconst
const vector<int>::iterator cit = nums.begin();
*cit = 1; // ok: cit can change its underlying element
++cit; // error: can't change the value of cit
```

A `const_iterator` may be used with either a `const` or `nonconst` `vector`, because it cannot write an element. An iterator that is `const` is largely useless: Once it is initialized, we can use it to write the element it refers to, but cannot make it refer to any other element.

`const_iterator` 对象可以用于 `const vector` 或非 `const vector`，因为不能改写元素值。`const` 迭代器这种类型几乎没什么用处：一旦它被初始化后，只能用它来改写其指向的元素，但不能使它指向任何其他元素。

Section 3.4. Introducing Iterators

```
const vector<int> nines(10, 9); // cannot change elements in nines
// error: citz could change the element it refers to and nines is const
const vector<int>::iterator citz = nines.begin();
// ok: it can't change an element value, so it can be used with a const vector<int>
vector<int>::const_iterator it = nines.begin();
*it = 10; // error: *it is const
++it; // ok: it isn't const so we can change its value
```



```
// an iterator that cannot write elements
vector<int>::const_iterator
// an iterator whose value cannot change
const vector<int>::iterator
```

Exercises Section 3.4

Exercise 3.17: Redo the exercises from [Section 3.3.2](#) (p. 96), using iterators rather than subscripts to access the elements in the `vector`.

重做 [3.3.2 节](#) 的习题，用迭代器而不是下标操作来访问 `vector` 中的元素。

Exercise 3.18: Write a program to create a `vector` with 10 elements. Using an iterator, assign each element a value that is twice its current value.

编写程序来创建有 10 个元素的 `vector` 对象。用迭代器把每个元素值改为当前值的 2 倍。

Exercise 3.19: Test your previous program by printing the `vector`.

验证习题 3.18 的程序，输出 `vector` 的所有元素。

Exercise 3.20: Explain which iterator you used in the previous programs, and why.

解释一下在上几个习题的程序实现中你用了哪种迭代器，并说明原因。

Exercise 3.21: When would you use an iterator that is `const`? When would you use a `const_iterator`. Explain the difference between them.

何时使用 `const` 迭代器的？又在何时使用 `const_iterator`？解释两者的区别。

3.4.1. Iterator Arithmetic

3.4.1. 迭代器的算术操作

In addition to the increment operator, which moves an iterator one element at a time, `vector` iterators (but few of the other library container iterators) also support other arithmetic operations. These operations are referred to as [iterator arithmetic](#), and include:

除了一次移动迭代器的一个元素的增量操作符外，`vector` 迭代器（其他标准库容器迭代器很少）也支持其他的算术操作。这些操作称为[迭代器算术操作 \(iterator arithmetic\)](#)，包括：

- `iter + n`

```
iter - n
```

We can add or subtract an integral value to an iterator. Doing so yields a new iterator positioned `n` elements ahead of (addition) or behind (subtraction) the element to which `iter` refers. The result of the addition or subtraction must refer to an element in the `vector` to which `iter` refers or to one past the end of that `vector`. The type of the value added or subtracted ought ordinarily to be the `vector`'s `size_type` or `difference_type` (see below).

可以对迭代器对象加上或减去一个整形值。这样做将产生一个新的迭代器，其位置在 `iter` 所指元素之前（加）或之后（减）`n` 个元素的位置。加或减之后的结果必须指向 `iter` 所指 `vector` 中的某个元素，或者是 `vector` 末端的下一个元素。加上或减去的值的类型应该是 `vector` 的 `size_type` 或 `difference_type` 类型（参考下面的解释）。

- `iter1 - iter2`

Computes the difference between two iterators as a value of a `signed` integral type named `difference_type`, which, like `size_type`, is defined by `vector`. The type is `signed` because subtraction might have a negative result. This type is guaranteed to be large enough to hold the distance between any two iterators. Both `iter1` and `iter2` must refer to elements in the same `vector` or the element one past the end of that `vector`.

该表达式用来计算两个迭代器对象的距离，该距离是名为 `difference_type` 的 `signed` 类型 `size_type` 的值，这里的 `difference_type` 是 `signed` 类型，因为减法运算可能产生负数的结果。该类型可以保证足够大以存储任何两个迭代器对象间的距离。`iter1` 与 `iter2` 两者必须都指向同一 `vector` 中的元素，或者指向 `vector` 末端之后的下一个元素。

We can use iterator arithmetic to move an iterator to an element directly. For example, we could locate the middle of a `vector` as follows:

可以用迭代器算术操作来移动迭代器直接指向某个元素，例如，下面语句直接定位 `vector` 中间元素：

```
vector<int>::iterator mid = vi.begin() + vi.size() / 2;
```

This code initializes `mid` to refer to the element nearest to the middle of `ivec`. It is more efficient to calculate this iterator directly than to write an equivalent program that increments the iterator one by one until it reaches the middle element.

上述代码用来初始化 `mid` 使其指向 `vi` 中最靠近正中间的元素。这种直接计算迭代器的方法，与用迭代器逐个元素自增操作到达中间元素的方法是等价的，但前者的效率要高得多。



Any operation that changes the size of a `vector` makes existing iterators invalid. For example, after calling `push_back`, you should not rely on the value of an iterator into the `vector`.

任何改变 `vector` 长度的操作都会使已存在的迭代器失效。例如，在调用 `push_back` 之后，就不能再信赖指向 `vector` 的迭代器的值了。

Exercises Section 3.4.1

Exercise What happens if we compute `mid` as follows:

3.22:

如果采用下面的方法来计算 `mid` 会产生什么结果？

```
vector<int>::iterator mid = (vi.begin() + vi.end()) / 2;
```

3.5. Library `bitset` Type

3.5. 标准库 `bitset`

Some programs deal with ordered collections of bits. Each bit can contain either a 0 (off) or a 1 (on) value. Using bits is a compact way to keep yes/no information (sometimes called flags) about a set of items or conditions. The standard library makes it easy to deal with bits through the `bitset` class. To use a `bitset` we must include its associated header file. In our examples, we also assume an appropriate `using` declaration for `std::bitset` is made:

有些程序要处理二进制位的有序集，每个位可能包含 0（关）1（开）值。位是用来保存一组项或条件的 yes/no 信息（有时也称标志）的简洁方法。标准库提供的 `bitset` 类简化了位集的处理。要使用 `bitset` 类就必须包含相关的头文件。在本书提供的例子中，假设都使用 `std::bitset` 的 `using` 声明：

```
#include <bitset>
using std::bitset;
```

3.5.1. Defining and Initializing `bitset`s

3.5.1. `bitset` 对象的定义和初始化

[Table 3.6](#) lists the constructors for `bitset`. Like `vector`, the `bitset` class is a class template. Unlike `vector`, objects of type `bitset` differ by size rather than by type. When we define a `bitset`, we say how many bits the `bitset` will contain, which we do by specifying the size between a pair of angle brackets.

[表 3.6](#) 列出了 `bitset` 的构造函数。类似于 `vector`, `bitset` 类是一种类模板；而与 `vector` 不一样的是 `bitset` 类型对象的区别仅在其长度而不在其类型。在定义 `bitset` 时，要明确 `bitset` 含有多少位，须在尖括号内给出它的长度值：

Table 3.6. Ways to Initialize a `bitset`

<code>bitset<n> b;</code>	<code>b</code> has <code>n</code> bits, each bit is 0 <code>b</code> 有 <code>n</code> 位，每位都 0
<code>bitset<n> b(u);</code>	<code>b</code> is a copy of the <code>unsigned long</code> value <code>u</code> <code>b</code> 是 <code>unsigned long</code> 型 <code>u</code> 的一个副本
<code>bitset<n> b(s);</code>	<code>b</code> is a copy of the bits contained in <code>string s</code> <code>b</code> 是 <code>string</code> 对象 <code>s</code> 中含有的位串的副本
<code>bitset<n> b(s, pos, n);</code>	<code>b</code> is a copy of the bits in <code>n</code> characters from <code>s</code> starting from position <code>pos</code> <code>b</code> 是 <code>s</code> 中从位置 <code>pos</code> 开始的 <code>n</code> 个位的副本。

```
bitset<32> bitvec; // 32 bits, all zero
```

The size must be a constant expression ([Section 2.7](#), p. 62). It might be defined, as we did here, using an integral literal constant or using a `const` object of integral type that is initialized from a constant.

给出的长度值必须是常量表达式 ([2.7 节](#))。正如这里给出的，长度值必须定义为整型字面值常量或是已用常量值初始化的整型的 `const` 对象。

This statement defines `bitvec` as a `bitset` that holds 32 bits. Just as with the elements of a `vector`, the bits in a `bitset` are not named. Instead, we refer to them positionally. The bits are numbered starting at 0. Thus, `bitvec` has bits numbered 0 through 31. The bits starting at 0 are referred to as the [low-order](#) bits, and those ending at 31 are referred to as [high-order](#) bits.

这条语句把 `bitvec` 定义为含有 32 个位的 `bitset` 对象。和 `vector` 的元素一样，`bitset` 中的位是没有命名的，程序员只能按位置来访问。位集合的位置编号从 0 开始，因此，`bitvec` 的位序是从 0 到 31。以 0 位开始的位串是低阶位 ([low-order](#))，以 31 位结束的位串是高阶位 ([high-order](#))。

Section 3.5. Library bitset Type

Initializing a `bitset` from an `unsigned` Value

用 `unsigned` 值初始化 `bitset` 对象

When we use an `unsigned long` value as an initializer for a `bitset`, that value is treated as a bit pattern. The bits in the `bitset` are a copy of that pattern. If the size of the `bitset` is greater than the number of bits in an `unsigned long`, then the remaining high-order bits are set to zero. If the size of the `bitset` is less than that number of bits, then only the low-order bits from the `unsigned` value are used; the high-order bits beyond the size of the `bitset` object are discarded.

当用 `unsigned long` 值作为 `bitset` 对象的初始值时，该值将转化为二进制的位模式。而 `bitset` 对象中的位集作为这种位模式的副本。如果 `bitset` 类型长度大于 `unsigned long` 值的二进制位数，则其余的高阶位将置为 0；如果 `bitset` 类型长度小于 `unsigned long` 值的二进制位数，则只使用 `unsigned` 值中的低阶位，超过 `bitset` 类型长度的高阶位将被丢弃。

On a machine with 32-bit `unsigned longs`, the hexadecimal value `0xffff` is represented in bits as a sequence of 16 ones followed by 16 zeroes. (Each `0xf` digit is represented as `1111`.) We can initialize a `bitset` from `0xffff`:

在 32 位 `unsigned long` 的机器上，十六进制值 `0xffff` 表示为二进制位就是十六个 1 和十六个 0（每个 `0xf` 可表示为 `1111`）。可以用 `0xffff` 初始化 `bitset` 对象：

```
// bitvec1 is smaller than the initializer
bitset<16> bitvec1(0xffff);           // bits 0 ... 15 are set to 1
// bitvec2 same size as initializer
bitset<32> bitvec2(0xffff);           // bits 0 ... 15 are set to 1; 16 ... 31 are 0
// on a 32-bit machine, bits 0 to 31 initialized from 0xffff
bitset<128> bitvec3(0xffff);          // bits 32 through 127 initialized to zero
```

In all three cases, the bits 0 to 15 are set to one. For `bitvec1`, the high-order bits in the initializer are discarded; `bitvec1` has fewer bits than an `unsigned long`. `bitvec2` is the same size as an `unsigned long`, so all the bits are used to initialize that object. `bitvec3` is larger than an `unsigned long`, so its high-order bits above 31 are initialized to zero.

上面的三个例子中，0 到 15 位都置为 1。由于 `bitvec1` 位数少于 `unsigned long` 的位数，因此 `bitvec1` 的初始值的高阶被丢弃。`bitvec2` 和 `unsigned long` 长度相同，因此所有位正好放置了初始值。`bitvec3` 长度大于 32, 31 位以上的高阶位就被置为 0。

Initializing a `bitset` from a `string`

用 `string` 对象初始化 `bitset` 对象

When we initialize a `bitset` from a `string`, the `string` represents the bit pattern directly. The bits are read from the `string` from right to left:

当用 `string` 对象初始化 `bitset` 对象时，`string` 对象直接表示为位模式。从 `string` 对象读入位集的顺序是从右向左（from right to left）：

```
string strval("1100");
bitset<32> bitvec4(strval);
```

The bit pattern in `bitvec4` has bit positions 2 and 3 set to 1, while the remaining bit positions are 0. If the `string` contains fewer characters than the size of the `bitset`, the high-order bits are set to zero.

`bitvec4` 的位模式中第 2 和 3 的位置为 1，其余位置都为 0。如果 `string` 对象的字符个数小于 `bitset` 类型的长度，则高阶位置为 0。



The numbering conventions of `strings` and `bitsets` are inversely related: The rightmost character in the `string` the one with the highest subscript is used to initialize the low-order bit in the `bitset` the bit with subscript 0. When initializing a `bitset` from a `string`, it is essential to remember this difference.

`string` 对象和 `bitsets` 对象之间是反向转化的：`string` 对象的最右边字符（即下标最大的那个字符）用来初始化 `bitset` 对象的低阶位（即下标为 0 的位）。当用 `string` 对象初始化 `bitset` 对象时，记住这一差别很重要。

We need not use the entire `string` as the initial value for the `bitset`. Instead, we can use a substring as the initializer:

不一定要把整个 `string` 对象都作为 `bitset` 对象的初始值。相反，可以只用某个子串作为初始值：

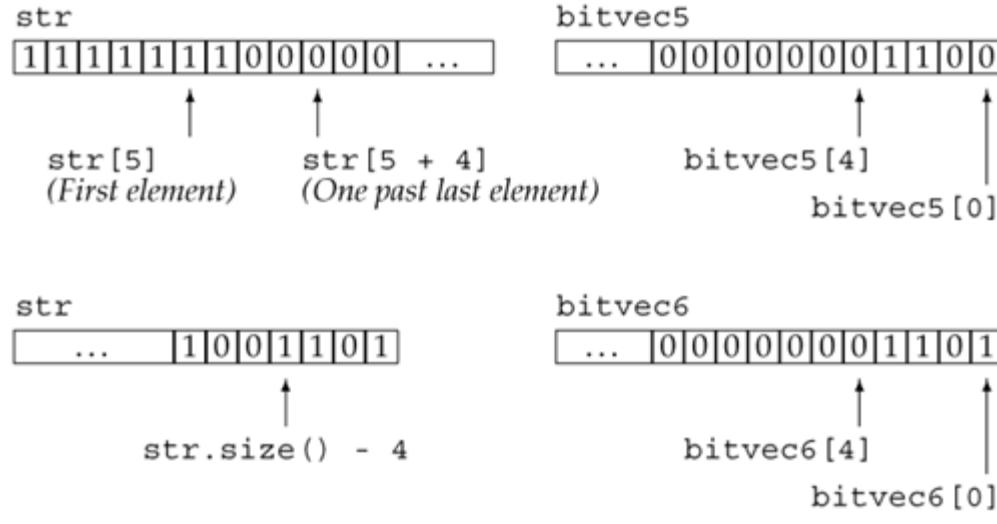
```
string str("11111100000011001101");
bitset<32> bitvec5(str, 5, 4); // 4 bits starting at str[5], 1100
```

Section 3.5. Library bitset Type

```
bitset<32> bitvec6(str, str.size() - 4); // use last 4 characters
```

Here `bitvec5` is initialized by a substring of `str` starting at `str[5]` and continuing for four positions. As usual, we start at the rightmost end of this substring when initializing the `bitset`, meaning that `bitvec5` is initialized with bit positions 0 through 3 set to `1100` while its remaining bit positions are set to 0. Leaving off the third parameter says to use characters from the starting position to the end of the `string`. In this case, the characters starting four from the end of `str` are used to initialize the lower four bits of `bitvec6`. The remainder of the bits in `bitvec6` are initialized to zero. We can view these initializations as

这里用 `str` 从 `str[5]` 开始包含四个字符的子串来初始化 `bitvec5`。照常，初始化 `bitset` 对象时总是从子串最右边结尾字符开始的，`bitvec5` 的从 3 到 0 的二进制位置为 `1100`，其他二进制位都置为 0。如果省略第三个参数则意味着取从开始位置一直到 `string` 末尾的所有字符。本例中，取出 `str` 末尾的四位来对 `bitvec6` 的低四位进行初始化。`bitvec6` 其余的位初始化为 0。这些初始化过程的图示如下：



3.5.2. Operations on `bitset`s

3.5.2. `bitset` 对象上的操作

The `bitset` operations ([Table 3.7](#)) define various operations to test or set one or more bits in the `bitset`.

多种 `bitset` 操作 ([表 3.7](#)) 用来测试或设置 `bitset` 对象中的单个或多个二进制位。

把 `b` 中所有二进制位都置为 1

Table 3.7. `bitset` Operations

<code>b.any()</code>	Is any bit in <code>b</code> on? <code>b</code> 中是否存在置为 1 的二进制位?
<code>b.none()</code>	Are no bits in <code>b</code> on? <code>b</code> 中不存在置为 1 的二进制位吗?
<code>b.count()</code>	Number of bits in <code>b</code> that are on <code>b</code> 中置为 1 的二进制位的个数
<code>b.size()</code>	Number of bits in <code>b</code> <code>b</code> 中置为 1 的二进制位的个数
<code>b[pos]</code>	Access bit in <code>b</code> at position <code>pos</code> 访问 <code>b</code> 中在 <code>pos</code> 处二进制位
<code>b.test(pos)</code>	Is bit in <code>b</code> in position <code>pos</code> on? <code>b</code> 中在 <code>pos</code> 处的二进制位置为 1
<code>b.set()</code>	Turn on all bits in <code>b</code>
<code>b.set(pos)</code>	Turn on the bit in <code>b</code> at position <code>pos</code> 把 <code>b</code> 中在 <code>pos</code> 处的二进制位置为 1

Section 3.5. Library bitset Type

<code>b.reset()</code>	Turn off all bits in <code>b</code> 把 <code>b</code> 中所有二进制位都置为 0
<code>b.reset(pos)</code>	Turn off the bit in <code>b</code> at position <code>pos</code> 把 <code>b</code> 中在 <code>pos</code> 处的二进制位置为 0
<code>b.flip()</code>	Change the state of each bit in <code>b</code> 把 <code>b</code> 中所有二进制位逐位取反
<code>b.flip(pos)</code>	Reverse value of the bit in <code>b</code> in position <code>pos</code> 把 <code>b</code> 中在 <code>pos</code> 处的二进制位取反
<code>b.to_ulong()</code>	Returns an <code>unsigned long</code> with the same bits as in <code>b</code> 用 <code>b</code> 中同样的二进制位返回一个 <code>unsigned long</code> 值
<code>os << b</code>	Prints the bits in <code>b</code> to the stream <code>os</code> 把 <code>b</code> 中的位集输出到 <code>os</code> 流

Testing the Entire `bitset`

测试整个 `bitset` 对象

The `any` operation returns `true` if one or more bits of the `bitset` object are turned on that is, are equal to 1. Conversely, the operation `none` returns `true` if all the bits of the object are set to zero.

如果 `bitset` 对象中有一个或几个二进制位置为 1，则 `any` 操作返回 `true`，也就是说，其返回值等于 1；相反，如果 `bitset` 对象中二进制位全为 0，则 `none` 操作返回 `true`。

```
bitset<32> bitvec; // 32 bits, all zero
bool is_set = bitvec.any();           // false, all bits are zero
bool is_not_set = bitvec.none();      // true, all bits are zero
```

If we need to know how many bits are set, we can use the `count` operation, which returns the number of bits that are set:

如果需要知道置为 1 的二进制位的个数，可以使用 `count` 操作，该操作返回置为 1 的二进制位的个数：

```
size_t bits_set = bitvec.count(); // returns number of bits that are on
```

The return type of the `count` operation is a library type named `size_t`. The `size_t` type is defined in the `cstdint` header, which is the C++ version of the `stddef.h` header from the C library. It is a machine-specific `unsigned` type that is guaranteed to be large enough to hold the size of an object in memory.

`count` 操作的返回类型是标准库中命名为 `size_t` 类型。`size_t` 类型定义在 `cstdint` 头文件中，该文件是 C 标准库的头文件 `stddef.h` 的 C++ 版本。它是一个与机器相关的 `unsigned` 类型，其大小足以保证存储内存中对象的大小。

The `size` operation, like the one in `vector` and `string`, returns the total number of bits in the `bitset`. The value returned has type `size_t`:

与 `vector` 和 `string` 中的 `size` 操作一样，`bitset` 的 `size` 操作返回 `bitset` 对象中二进制位的个数，返回值的类型是 `size_t`:

```
size_t sz = bitvec.size(); // returns 32
```

Accessing the Bits in a `bitset`

访问 `bitset` 对象中的位

The subscript operator lets us read or write the bit at the indexed position. As such, we can use it to test the value of a given bit or to set that

Section 3.5. Library bitset Type

value:

可以用下标操作符来读或写某个索引位置的二进制位，同样地，也可以用下标操作符测试给定二进制位的值或设置某个二进制位的值：

```
// assign 1 to even numbered bits
for (int index = 0; index != 32; index += 2)
    bitvec[index] = 1;
```

This loop turns on the even-numbered bits of `bitvec`.

上面的循环把 `bitvec` 中的偶数下标的位都置为 1。

As with the subscript operator, we can use the `set`, `test`, and `reset` operations to test or set a given bit value:

除了用下标操作符，还可以用 `set`、`test` 和 `reset` 操作来测试或设置给定二进制位的值：

```
// equivalent loop using set operation
for (int index = 0; index != 32; index += 2)
    bitvec.set(index);
```

To test whether a bit is on, we can use the `test` operation or test the value returned from the subscript operator:

为了测试某个二进制位是否为 1，可以用 `test` 操作或者测试下标操作符的返回值：

```
if (bitvec.test(i))
    // bitvec[i] is on
// equivalent test using subscript
if (bitvec[i])
    // bitvec[i] is on
```

The result of testing the value returned from a subscript is `true` if the bit is 1 or `false` if the bit is 0.

如果下标操作符测试的二进制位为 1，则返回的测试值的结果为 `true`，否则返回 `false`。

Setting the Entire `bitset`

对整个 `bitset` 对象进行设置

The `set` and `reset` operations can also be used to turn on or turn off the entire `bitset` object, respectively:

`set` 和 `reset` 操作分别用来对整个 `bitset` 对象的所有二进制位全置 1 和全置 0：

```
bitvec.reset(); // set all the bits to 0.
bitvec.set();   // set all the bits to 1
```

The `flip` operation reverses the value of an individual bit or the entire `bitset`:

`flip` 操作可以对 `bitset` 对象的所有位或个别位取反：

```
bitvec.flip(0); // reverses value of first bit
bitvec[0].flip(); // also reverses the first bit
bitvec.flip();   // reverses value of all bits
```

Retrieving the Value of a `bitset`

获取 `bitset` 对象的值

The `to_ulong` operation returns an `unsigned long` that holds the same bit pattern as the `bitset` object. We can use this operation only if the size of the `bitset` is less than or equal to the size of an `unsigned long`:

`to_ulong` 操作返回一个 `unsigned long` 值，该值与 `bitset` 对象的位模式存储值相同。仅当 `bitset` 类型的长度小于或等于 `unsigned long` 的长度时，才可以使用 `to_ulong` 操作：

Section 3.5. Library bitset Type

```
unsigned long ulong = bitvec3.to_ulong();
cout << "ulong = " << ulong << endl;
```

The `to_ulong` operation is intended to be used when we need to pass a `bitset` to a C or pre-Standard C++ program. If the `bitset` contains more bits than the size of an `unsigned long`, a run-time exception is signaled. We'll introduce exceptions in [Section 6.13](#) (p. 215) and look at them in more detail in [Section 17.1](#) (p. 688).

`to_ulong` 操作主要用于把 `bitset` 对象转到 C 风格或标准 C++ 之前风格的程序上。如果 `bitset` 对象包含的二进制位数超过 `unsigned long` 长度，将会产生运行时异常。本书将在 [6.13 节](#)介绍异常 (exception)，并在 [17.1 节](#)中详细地讨论它。

Printing the Bits

输出二进制位

We can use the output operator to print the bit pattern in a `bitset` object:

可以用输出操作符输出 `bitset` 对象中的位模式：

```
bitset<32> bitvec2(0xffff); // bits 0 ... 15 are set to 1; 16 ... 31 are 0
cout << "bitvec2: " << bitvec2 << endl;
```

will print

输出结果为：

```
bitvec2: 00000000000000001111111111111111
```

Using the Bitwise Operators

使用位操作符

The `bitset` class also supports the built-in bitwise operators. As defined by the language, these operators apply to integral operands. They perform operations similar to the `bitset` operations described in this section. [Section 5.3](#) (p. 154) describes these operators.

`bitset` 类也支持内置的位操作符。C++ 定义的这些操作符都只适用于整型操作数，它们所提供的操作类似于本节所介绍的 `bitset` 操作。[5.3 节](#)将介绍这些操作符。

Exercises Section 3.5.2

Exercise Explain the bit pattern each of the following `bitset` objects contains:

3.23:

解释下面每个 `bitset` 对象包含的位模式：

```
(a) bitset<64> bitvec(32);
(b) bitset<32> bv(1010101);
(c) string bstr; cin >> bstr; bitset<8>bv(bstr);
```

Exercise Consider the sequence 1,2,3,5,8,13,21. Initialize a `bitset<32>` object that has a one bit in each position corresponding to a number in this sequence. Alternatively, given an empty `bitset`, write a small program to turn on each of the appropriate bits.

考虑这样的序列 1, 2, 3, 5, 8, 13, 21，并初始化一个将该序列数字所对应的位置置为 1 的 `bitset<32>` 对象。然后换个方法，给定一个空的 `bitset`，编写一小段程序把相应的数位设置为 1。

Chapter Summary

The library defines several higher-level abstract data types, including `strings` and `vectors`. The `string` class provides variable-length character strings, and the `vector` type manages a collection of objects of a single type.

C++ 标准库定义了几种更高级的抽象数据类型，包括 `string` 和 `vector` 类型。`string` 类型提供了变长的字符串，而 `vector` 类型则可用于管理同一类型的对象集合。

Iterators allow indirect access to objects stored in a container. Iterators are used to access and navigate between the elements in `strings` and `vectors`.

迭代器实现了对存储于容器中对象的间接访问。迭代器可以用于访问和遍历 `string` 类型和 `vectors` 类型的元素。

In the next chapter we'll cover arrays and pointers, which are types built into the language. These types provide low-level analogs to the `vector` and `string` libraries. In general, the library classes should be used in preference to low-level array and pointer alternatives built into the language.

下一章将介绍 C++ 的内置数据类型：数组和指针。这两种类型提供了类似于 `vector` 和 `string` 标准库类型的低级抽象类型。总的来说，相对于 C++ 内置数据类型的数组和指针而言，程序员应优先使用标准库类类型。

Defined Terms

术语

abstract data type (抽象数据类型)

A type whose representation is hidden. To use an abstract type, we need know only what operations the type supports.
隐藏其实现的数据类型。使用抽象数据类型时，只需要了解该类型所支持的操作。

bitset

Standard library class that holds a collection of bits and provides operations to test and set the bits in the collection.
一种标准库类型，用于保存位置，并提供地各个位的测试和置位操作。

cctype header (cctype 头文件)

Header inherited from C that contains routines to test character values. See page [88](#) for a listing of the most common routines.
从 C 标准库继承而来的头文件，包含一组测试字符值的例程。第 8.3.4 节的表 3.3 列出了常用的例程。

class template (类模板)

A blueprint from which many potential class types can be created. To use a class template, we must specify what actual type(s) or value(s) to use. For example, a `vector` is a template that holds objects of a given type. When we create a `vector`, we must say what type *this vector* will hold. `vector<int>` holds `ints`, `vector<string>` holds `strings`, and so on.

一个可创建许多潜在类类型的蓝图。使用类模板时，必须给出实际的类型和值。例如，`vector` 类型是保存给定类型对象的模板。创建一个 `vector` 对象时，必须指出这个 `vector` 对象所保存的元素的类型。`vector<int>` 保存 `int` 的对象，而 `vector<string>` 则保存 `string` 对象，以此类推。

container (容器)

A type whose objects hold a collection of objects of a given type.
一种类型，其对象保存一组给定类型的对象的集合。

difference type

A `signed` integral type defined by `vector` that is capable of holding the distance between any two iterators.
一种由 `vector` 类型定义的 `signed` 整型，用于存储任意两个迭代器间的距离。

empty

Function defined by the `string` and `vector` types. `empty` returns a `bool` indicating whether the `string` has any characters or whether the `vector` has any elements. Returns `True` if `size` is zero; `false` otherwise.

由`string`类型和`vector`类型定义的成员函数。`empty`返回布尔值，用于检测`string`是否有字符或`vector`是否有元素。如果`string`或`vector`的`size`为0，则返回`true`，否则返回`false`。

getline

Function defined in the `string` header that takes an `istream` and a `string`. The function reads the stream up to the next newline, storing what it read into the `string`, and returns the `istream`. The newline is read and discarded.

`string`头文件中定义的函数，该函数接受一个`istream`对象和一个`string`对象，读取输入流直到下一个换行符，存储读入的输入流到`string`对象中，并返回`istream`对象。换行符被读入并丢弃。

high-order (高阶)

Bits in a `bitset` with the largest indices.

`bitset`对象中索引值最大的位。

[index \(索引\)](#)

Value used in the subscript operator to denote the element to retrieve from a `string` or `vector`.

下标操作符所使用的值，用于表示从`string`对象或`vector`对象中获取的元素。也称“下标”。

[iterator \(迭代器\)](#)

A type that can be used to examine the elements of a container and to navigate between them.

用于对容器类型的元素进行检查和遍历的数据类型。

[iterator arithmetic \(迭代器的算术操作\)](#)

The arithmetic operations that can be applied to some, but not all, iterator types. An integral type can be added to or subtracted from an iterator, resulting in an iterator positioned that many elements ahead of or behind the original iterator. Two iterators can be subtracted, yielding the distance between the iterators. Iterator arithmetic is valid only on iterators that refer to elements in the same container or the off-the-end iterator of the container.

应用于一些（并非全部）迭代器类型的算术操作。迭代器对象可以加上或减去一个整型数值，结果迭代器指向处于原迭代器之前或之后若干个元素的位置。两个迭代器对象可以相减，得到的结果是它们之间的距离。迭代器算术操作只适用于指向同一容器中的元素或指向容器末端的下一元素迭代器。

[low-order \(低阶\)](#)

Bits in a `bitset` with the lowest indices.

`bitset`对象中索引值最小的位。

[off-the-end iterator \(超出末端的迭代器\)](#)

The iterator returned by `end`. It is an iterator that refers to a nonexistent element one past the end of a container.

由`end`操作返回的迭代器，是一种指向容器末端之后的不存在元素的迭代器。

[push_back](#)

Function defined by `vector` that appends elements to the back of a `vector`.

由`vector`类型定义的成员函数，用于把元素追加到`vector`对象的尾部。

[sentinel \(哨兵\)](#)

Programming technique that uses a value as a guard to control processing. In this chapter, we showed the use of the iterator returned by `end` as a guard to stop processing elements in a `vector` once we had processed every element in the `vector`.

一种程序设计技术，使用一个值来控制处理过程。在本章中使用由`end`操作返回的迭代器作为保护符，当处理完`vector`对象中的所有元素后，用它来停止处理`vector`中的元素。

[size](#)

Function defined by the library types `string`, `vector`, and `bitset` that returns the number of characters, elements, or bits respectively. The `string` and `vector` functions return a value of the `size_type` for the type. For example, `size` of a `string` returns a `string::size_type`. The `bitset` operation returns a value of type `size_t`.

由库类型`string`、`vector`和`bitset`定义的函数，分别用于返回此三个类型的字符个数、元素个数、二进制位的个数。`string`和`vector`类的`size`成员函数返回`size_type`类型的值（例如，`string`对象的`size`操作返回`string::size_type`类型值）。`bitset`对象的`size`操作返回`size_t`类型值。

[size_t](#)

Machine-dependent unsigned integral type defined in `cstdint` header that is large enough to hold the size of the largest possible array.

在`cstdint`头文件中定义的机器相关的无符号整型，该类型足以保存最大数组的长度。

在`cstdint`头文件中定义的机器相关的无符号整型，该类型足以保存最大数组的长度。

[size_type](#)

Type defined by the `string` and `vector` classes that is capable of containing the size of any `string` or `vector`, respectively. Library classes that define `size_type` define it as an `unsigned` type.

由`string`类类型和`vector`类类型定义的类型，用以保存任意`string`对象或`vector`对象的长度。标准库类型将`size_type`定义为`unsigned`类型。

[using declarations \(using 声明\)](#)

Make a name from a namespace accessible directly.

使命名空间的名字可以直接引用。比如：

```
using namespace::name;
```

makes *name* accessible without the *namespace::* prefix.

可以直接访问*name*而无须前缀*namespace::*。

[value initialization \(值初始化\)](#)

Initialization that happens for container elements when the container size is specified but there is no explicit element initializer. The elements are initialized as a copy of a compiler-generated value. If the container holds built-in types, then the value copied into the elements is zero. For class types, the value is generated by running the class's default constructor. Container elements that are of class type can be value-initialized only if the class has a default constructor.

当给定容器的长度，但没有显式提供元素的初始值时，对容器元素进行的初始化。元素被初始化为一个编译器产生的值的副本。如果容器保存内置类型变量，则元素的初始值将置为0。如果容器用于保存类对象，则元素的初始值由类的默认构造函数产生。只有类提供了默认构造函数时，类类型的容器元素才能进行值初始化。

[++ operator \(++ 操作符\)](#)

The iterator types define the increment operator to "add one" by moving the iterator to refer to the next element.

迭代器类型定义的自增操作符，通过“加1”移动迭代器指向下一个元素。

[:: operator \(::操作符\)](#)

The scope operator. It finds the name of its right-hand operand in the scope of its left-hand operand. Used to access names in a namespace, such as `std::cout`, which represents the name `cout` from the namespace `std`. Similarly, used to obtain names from a class, such as `string::size_type`, which is the `size_type` defined by the `string` class.

作用域操作符。::操作符在其左操作数的作用域内找到其右操作数的名字。用于访问某个命名空间中的名字，如`std::cout`，表明名字`cout`来自命名空间`std`。同样地，可用来从某个类取名字，如`string::size_type`，表明`size_type`是由`string`类定义的。

[\[\] operator \(\[\]操作符\)](#)

An overloaded operator defined by the `string`, `vector`, and `bitset` types. It takes two operands: The left-hand operand is the name of the object and the right-hand operand is an index. It fetches the element whose position matches the index. Indices count from zero—the first element is element 0 and the last is element indexed by `obj.size() - 1`. Subscript returns an lvalue, so we may use a subscript as the left-hand operand of an assignment. Assigning to the result of a subscript assigns a new value to the indexed element.

由`string`, `vector`和`bitset`类型定义的重载操作符。它接受两个操作数：左操作数是对象名字，右操作数是一个索引。该操作符用于取出位置与索引相符的元素，索引计数从0开始，即第一个元素的索引为0，最后一个元素的索引为`obj.size() - 1`。下标操作返回左值，因此可将下标操作作为赋值操作的左操作数。对下标操作的结果赋值是赋一个新值到相应的元素。

[* operator \(*操作符\)](#)

The iterator types define the dereference operator to return the object to which the iterator refers. Dereference returns an lvalue, so we may use a dereference operator as the left-hand operand of an assignment. Assigning to the result of a dereference assigns a new value to the indexed element.

迭代器类型定义解引用操作符来返回迭代器所指向的对象。解引用返回左值，因此可将解引用操作符用作赋值操作的左操作数。对解引用操作的结果赋值是赋一个新值到相应的元素。

[<< operator \(<< 操作符\)](#)

The `string` and `bitset` library types define an output operator. The `string` operator prints the characters in a `string`. The `bitset` operator prints the bit pattern in the `bitset`.

标准库类型`string`和`bitset`定义了输出操作符。`string`类型的输出操作符将输出`string`对象中的字符。`bitset`类型的输出操作符则输出`bitset`对象的位模式。

[>> operator \(>> 操作符\)](#)

The `string` and `bitset` library types define an input operator. The `string` operator reads whitespace-delimited chunks of characters, storing what is read into the right-hand (`string`) operand. The `bitset` operator reads a bit sequence into its `bitset` operand.

标准库类型`string`和`bitset`定义了输入操作符。`string`类型的输入操作符读入以空白字符为分隔符的字符串，并把读入的内容存储在右操作数(`string`对象)中。`bitset`类型的输入操作符则读入一个位序列到其`bitset`操作数中。

Chapter 4. Arrays and Pointers

第四章 数组和指针

CONTENTS

Section 4.1 Arrays	110
Section 4.2 Introducing Pointers	114
Section 4.3 C-Style Character Strings	130
Section 4.4 Multidimensioned Arrays	141
Chapter Summary	145
Defined Terms	145

The language defines two lower-level compound types arrays and pointers that are similar to `vectors` and iterators. Like a `vector`, an array holds a collection of objects of some type. Unlike `vectors`, arrays are fixed size; once an array is created, new elements cannot be added. Like iterators, pointers can be used to navigate among and examine the elements in an array.

C++ 语言提供了两种类似于 `vector` 和迭代器类型的低级复合类型——数组和指针。与 `vector` 类型相似，数组也可以保存某种类型的一组对象；而它们的区别在于，数组的长度是固定的。数组一经创建，就不允许添加新的元素。指针则可以像迭代器一样用于遍历和检查数组中的元素。

Modern C++ programs should almost always use `vectors` and iterators in preference to the lower-level arrays and pointers. Well-designed programs use arrays and pointers only in the internals of class implementations where speed is essential.

现代 C++ 程序应尽量使用 `vector` 和迭代器类型，而避免使用低级的数组和指针。设计良好的程序只有在强调速度时才在类实现的内部使用数组和指针。

Arrays are data structures that are similar to library `vectors` but are built into the language. Like a `vector`, an array is a container of objects of a single data type. The individual objects are not named; rather, each one is accessed by its position in the array.

数组是 C++ 语言中类似于标准库 `vector` 类型的内置数据结构。与 `vector` 类似，数组也是一种存储单一数据类型对象的容器，其中每个对象都没有单独的名字，而是通过它在数组中的位置对它进行访问。

Arrays have significant drawbacks compared to `vectors`: They are fixed size, and they offer no help to the programmer in keeping track of how big a given array is. There is no `size` operation on arrays. Similarly, there is no `push_back` to automatically add elements. If the array size needs to change, then the programmer must allocate a new, larger array and copy the elements into that new space.

与 `vector` 类型相比，数组的显著缺陷在于：数组的长度是固定的，而且程序员无法知道一个给定数组的长度。数组没有获取其容量大小的 `size` 操作，也不提供 `push_back` 操作在其中自动添加元素。如果需要更改数组的长度，程序员只能创建一个更大的新数组，然后把原数组的所有元素复制到新数组空间中去。



Programs that rely on built-in arrays rather than using the standard `vector` are more error-prone and harder to debug.

与使用标准 `vector` 类型的程序相比，依赖于内置数组的程序更容易出错而且难于调试。

Prior to the advent of the standard library, C++ programs made heavy use of arrays to hold collections of objects. Modern C++ programs should almost always use `vectors` instead of arrays. Arrays should be restricted to the internals of programs and used only where performance testing indicates that `vectors` cannot provide the necessary speed. However, there will be a large body of existing C++ code that relies on arrays for some time to come. Hence, all C++ programmers must know a bit about how arrays work.

在出现标准库之前，C++ 程序大量使用数组保存一组对象。而现代的 C++ 程序则更多地使用 `vector` 来取代数组，数组被严格限制于程序内部使用，只有当性能测试表明使用 `vector` 无法达到必要的速度要求时，才使用数组。然而，在将来一段时间之内，原来依赖于数组的程序仍大量存在，因此，C++ 程序员还是必须掌握数组的使用方法。

4.1. Arrays

4.1. 数组

An array is a compound type ([Section 2.5](#), p. 58) that consists of a type specifier, an identifier, and a **dimension**. The type specifier indicates what type the elements stored in the array will have. The dimension specifies how many elements the array will contain.

数组是由类型名、标识符和维数组成的复合数据类型 ([第 2.5 节](#))，类型名规定了存放在数组中的元素的类型，而维数则指定数组中包含的元素个数。



The type specifier can denote a built-in data or class type. With the exception of references, the element type can also be any compound type. There are no arrays of references.

数组定义中的类型名可以是内置数据类型或类类型；除引用之外，数组元素的类型还可以是任意的复合类型。没有所有元素都是引用的数组。

4.1.1. Defining and Initializing Arrays

4.1.1. 数组的定义和初始化

The dimension must be a constant expression ([Section 2.7](#), p. 62) whose value is greater than or equal to one. A constant expression is any expression that involves *only* integral literal constants, enumerators ([Section 2.7](#), p. 62), or `const` objects of integral type that are themselves initialized from constant expressions. A non`const` variable, or a `const` variable whose value is not known until run time, cannot be used to specify the dimension of an array.

数组的维数必须用值大于等于1的常量表达式定义 ([第 2.7 节](#))。此常量表达式只能包含整型字面值常量、枚举常量 ([第 2.7 节](#)) 或者用常量表达式初始化的整型 `const` 对象。非 `const` 变量以及要到运行阶段才知道其值的 `const` 变量都不能用于定义数组的维数。

The dimension is specified inside a `[]` bracket pair:

数组的维数必须在一对方括号 `[]` 内指定：

```
// both buf_size and max_files are const
const unsigned buf_size = 512, max_files = 20;
int staff_size = 27;           // nonconst
const unsigned sz = get_size(); // const value not known until run time
char input_buffer[buf_size];   // ok: const variable
string fileTable[max_files + 1]; // ok: constant expression
double salaries[staff_size];   // error: non const variable
int test_scores[get_size()];    // error: non const expression
int vals[sz];                 // error: size not known until run time
```

Although `staff_size` is initialized with a literal constant, `staff_size` itself is a non`const` object. Its value can be known only at run time, so it is illegal as an array dimension. Even though `size` is a `const` object, its value is not known until `get_size` is called at run time. Therefore, it may not be used as a dimension. On the other hand, the expression

虽然 `staff_size` 是用字面值常量进行初始化，但 `staff_size` 本身是一个非 `const` 对象，只有在运行时才能获得它的值，因此，使用该变量来定义数组维数是非法的。而对于 `sz`，尽管它是一个 `const` 对象，但它的值要到运行时调用 `get_size` 函数后才知道，因此，它也不能用于定义数组维数。

```
max_files + 1
```

is a constant expression because `max_files` is a `const` variable. The expression can be and is evaluated at compile time to a value of 21.

另一方面，由于 `max_files` 是 `const` 变量，因此表达式是常量表达式，编译时即可计算出该表达式的值为21。

Section 4.1. Arrays

Explicitly Initializing Array Elements

显式初始化数组元素

When we define an array, we can provide a comma-separated list of initializers for its elements. The initializer list must be enclosed in braces:

在定义数组时，可为其元素提供一组用逗号分隔的初值，这些初值用花括号{}括起来，称为初始化列表：

```
const unsigned array_size = 3;  
int ia[array_size] = {0, 1, 2};
```

If we do not supply element initializers, then the elements are initialized in the same way that variables are initialized ([Section 2.3.4](#), p. 50).

如果没有显式提供元素初值，则数组元素会像普通变量一样初始化（[第 2.3.4 节](#)）：

- Elements of an array of built-in type defined outside the body of a function are initialized to zero.
在函数体外定义的内置数组，其元素均初始化为 0。
- Elements of an array of built-in type defined inside the body of a function are uninitialized.
在函数体内定义的内置数组，其元素无初始化。
- Regardless of where the array is defined, if it holds elements of a class type, then the elements are initialized by the default constructor for that class if it has one. If the class does not have a default constructor, then the elements must be explicitly initialized.
不管数组在哪里定义，如果其元素为类类型，则自动调用该类的默认构造函数进行初始化；如果该类没有默认构造函数，则必须为该数组的元素提供显式初始化。



Unless we explicitly supply element initializers, the elements of a local array of built-in type are uninitialized.
Using these elements for any purpose other than to assign a new value is undefined.

除非显式地提供元素初值，否则内置类型的局部数组的元素没有初始化。此时，除了给元素赋值外，其他使用这些元素的操作没有定义。

An explicitly initialized array need not specify a dimension value. The compiler will infer the array size from the number of elements listed:

显式初始化的数组不需要指定数组的维数值，编译器会根据列出的元素个数来确定数组的长度：

```
int ia[] = {0, 1, 2}; // an array of dimension 3
```

If the dimension size is specified, the number of elements provided must not exceed that size. If the dimension size is greater than the number of listed elements, the initializers are used for the first elements. The remaining elements are initialized to zero if the elements are of built-in type or by running the default constructor if they are of class type:

如果指定了数组维数，那么初始化列表提供的元素个数不能超过维数值。如果维数大于列出的元素初值个数，则只初始化前面的数组元素；剩下的其他元素，若是内置类型则初始化为0，若是类类型则调用该类的默认构造函数进行初始化：

```
const unsigned array_size = 5;  
// Equivalent to ia = {0, 1, 2, 0, 0}  
// ia[3] and ia[4] default initialized to 0  
int ia[array_size] = {0, 1, 2};  
// Equivalent to str_arr = {"hi", "bye", "", "", ""}  
// str_arr[2] through str_arr[4] default initialized to the empty string  
string str_arr[array_size] = {"hi", "bye"};
```

Character Arrays Are Special

特殊的字符数组

A character array can be initialized with either a list of comma-separated character literals enclosed in braces or a string literal. Note, however, that the two forms are not equivalent. Recall that a string literal ([Section 2.2](#), p. 40) contains an additional terminating null character. When we create a character array from a string literal, the null is also inserted into the array:

字符数组既可以用一组由花括号括起来、逗号隔开的字符字面值进行初始化，也可以用一个字符串字面值进行初始化。然而，要注意这两种初始化形式并不完全相同，字符串字面值（[第 2.2 节](#)）包含一个额外的空字符（null）用于结束字符串。当使用字符串字面值来初始化创建的新数组时，将在新数组中加入空字符：

```
char ca1[] = {'C', '+', '+'}; // no null  
char ca2[] = {'C', '+', '+', '\0'}; // explicit null  
char ca3[] = "C++"; // null terminator added automatically
```

Section 4.1. Arrays

The dimension of `ca1` is 3; the dimension of `ca2` and `ca3` is 4. It is important to remember the null-terminator when initializing an array of characters to a literal. For example, the following is a compile-time error:

`ca1` 的维数是 3, 而 `ca2` 和 `ca3` 的维数则是 4。使用一组字符字面值初始化字符数组时, 一定要记得添加结束字符串的空字符。例如, 下面的初始化将导致编译时的错误:

```
const char ch3[6] = "Daniel"; // error: Daniel is 7 elements
```

While the literal contains only six explicit characters, the required array size is seven to hold the literal and one for the null.

上述字符串字面值包含了 6 个显式字符, 存放该字符串的数组则必须有 7 个元素——6 个用于存储字符字面值, 而 1 个用于存放空字符 null。

No Array Copy or Assignment

不允许数组直接复制和赋值

Unlike a `vector`, it is not possible to initialize an array as a copy of another array. Nor is it legal to assign one array to another:

与 `vector` 不同, 一个数组不能用另外一个数组初始化, 也不能将一个数组赋值给另一个数组, 这些操作都是非法的:

```
int ia[] = {0, 1, 2}; // ok: array of ints
int ia2[](ia);        // error: cannot initialize one array with another

int main()
{
    const unsigned array_size = 3;
    int ia3[array_size]; // ok: but elements are uninitialized!

    ia3 = ia;           // error: cannot assign one array to another
    return 0;
}
```



Some compilers allow array assignment as a [compiler extension](#). If you intend to run a given program on more than one compiler, it is usually a good idea to avoid using nonstandard compiler-specific features such as array assignment.

一些编译器允许将数组赋值作为[编译器扩展](#)。但是如果希望编写的程序能在不同的编译器上运行, 则应该避免使用像数组赋值这类依赖于编译器的非标准功能。

Caution: Arrays Are Fixed Size

警告: 数组的长度是固定的

Unlike the `vector` type, there is no `push_back` or other operation to add elements to the array. Once we define an array, we cannot add elements to it.

与 `vector` 类型不同, 数组不提供 `push_back` 或者其他的操作在数组中添加新元素, 数组一经定义, 就不允许再添加新元素。

If we must add elements to the array, then we must manage the memory ourselves. We have to ask the system for new storage to hold the larger array and copy the existing elements into that new storage. We'll see how to do so in [Section 4.3.1 \(p. 134\)](#).

如果必须在数组中添加新元素, 程序员就必须自己管理内存: 要求系统重新分配一个新的内存空间用于存放更大的数组, 然后把原数组的所有元素复制到新分配的内存空间中。我们将会在第 4.3.1 节学习如何去实现。

Exercises Section 4.1.1

Exercise 4.1: Assuming `get_size` is a function that takes no arguments and returns an `int` value, which of the following definitions are illegal? Explain why.

假设 `get_size` 是一个没有参数并返回 `int` 值的函数, 下列哪些定义是非法的? 为什么?

```
unsigned buf_size = 1024;
(a) int ia[buf_size];
(b) int ia[get_size()];
(c) int ia[4 * 7 - 14];
```

```
(d) char st[11] = "fundamental";
```

Exercise What are the values in the following arrays?

4.2:

下列数组的值是什么?

```
string sa[10];
int ia[10];
int main() {
    string sa2[10];
    int ia2[10];
}
```

Exercise Which, if any, of the following definitions are in error?

4.3:

下列哪些定义是错误的?

- (a) int ia[7] = { 0, 1, 1, 2, 3, 5, 8 };
- (b) vector<int> ivec = { 0, 1, 1, 2, 3, 5, 8 };
- (c) int ia2[] = ia1;
- (d) int ia3[] = ivec;

Exercise How can you initialize some or all the elements of an array?

4.4:

如何初始化数组的一部分或全部元素?

Exercise List some of the drawbacks of using an array instead of a `vector`.

4.5:

列出使用数组而不是 `vector` 的缺点。

4.1.2. Operations on Arrays

4.1.2. 数组操作

Array elements, like `vector` elements, may be accessed using the subscript operator ([Section 3.3.2](#), p. [94](#)). Like the elements of a `vector`, the elements of an array are numbered beginning with 0. For an array of ten elements, the correct index values are 0 through 9, not 1 through 10.

与 `vector` 元素一样，数组元素可用下标操作符 ([第 3.3.2 节](#)) 来访问，数组元素也是从 0 开始计数。对于一个包含 10 个元素的数组，正确的下标值是从 0 到 9，而不是从 1 到 10。

When we subscript a `vector`, we use `vector::size_type` as the type for the index. When we subscript an array, the right type to use for the index is `size_t` ([Section 3.5.2](#), p. [104](#)).

在用下标访问元素时，`vector` 使用 `vector::size_type` 作为下标的类型，而数组下标的正确类型则是 `size_t` ([第 3.5.2 节](#))。

In the following example, a `for` loop steps through the 10 elements of an array, assigning to each the value of its index:

在下面的例子中，`for` 循环遍历数组的 10 个元素，并以其下标值作为各个元素的初始值:

```
int main()
{
    const size_t array_size = 10;
    int ia[array_size]; // 10 ints, elements are uninitialized

    // loop through array, assigning value of its index to each element
    for (size_t ix = 0; ix != array_size; ++ix)
        ia[ix] = ix;
    return 0;
}
```

Using a similar loop, we can copy one array into another:

使用类似的循环，可以实现把一个数组复制给另一个数组:

```
int main()
{
```

Section 4.1. Arrays

```
const size_t array_size = 7;
int ia1[] = { 0, 1, 2, 3, 4, 5, 6 };

// copy elements from ia1 into ia2
for (size_t ix = 0; ix != array_size; ++ix)
    ia2[ix] = ia1[ix];
return 0;
}
```

Checking Subscript Values

检查数组下标值

As with both `strings` and `vectors`, the programmer must guarantee that the subscript value is in range that the array has an element at the index value.

正如 `string` 和 `vector` 类型，程序员在使用数组时，也必须保证其下标值在正确范围之内，即数组在该下标位置应对应一个元素。

Nothing stops a programmer from stepping across an array boundary except attention to detail and thorough testing of the code. It is not inconceivable for a program to compile and execute and still be fatally wrong.

除了程序员自己注意细节，并彻底测试自己的程序之外，没有别的办法可防止数组越界。通过编译并执行的程序仍然存在致命的错误，这并不是不可能的。



By far, the most common causes of security problems are so-called "buffer overflow" bugs. These bugs occur when a subscript is not checked and reference is made to an element outside the bounds of an array or other similar data structure.

导致安全问题的最常见原因是所谓“缓冲区溢出（buffer overflow）”错误。当我们在编程时没有检查下标，并且引用了超出数组或其他类似数据结构边界的元素时，就会导致这类错误。

Exercises Section 4.1.2

Exercise 4.6: This code fragment intends to assign the value of its index to each array element. It contains a number of indexing errors. Identify them.

下面的程序段企图将下标值赋给数组的每个元素，其中在下标操作上有一些错误，请指出这些错误。

```
const size_t array_size = 10;
int ia[array_size];
for (size_t ix = 1; ix <= array_size; ++ix)
    ia[ix] = ix;
```

Exercise 4.7: Write the code necessary to assign one array to another. Now, change the code to use `vectors`. How might you assign one `vector` to another?

编写必要的代码将一个数组赋给另一个数组，然后把这段代码改用 `vector` 实现。考虑如何将一个 `vector` 赋给另一个 `vector`。

Exercise 4.8: Write a program to compare two arrays for equality. Write a similar program to compare two `vectors`.

编写程序判断两个数组是否相等，然后编写一段类似的程序比较两个 `vector`。

Exercise 4.9: Write a program to define an array of 10 `ints`. Give each element the same value as its position in the array.

编写程序定义一个有 10 个 `int` 型元素的数组，并以其在数组中的位置作为各元素的初值。

Team LiB

◀ PREVIOUS NEXT ▶

4.2. Introducing Pointers

4.2. 指针的引入

Just as we can traverse a `vector` either by using a subscript or an iterator, we can also traverse an array by using either a subscript or a `pointer`. A pointer is a compound type; a pointer points to an object of some other type. Pointers are iterators for arrays: A pointer can point to an element in an array. The dereference and increment operators, when applied to a pointer that points to an array element, have similar behavior as when applied to an iterator. When we dereference a pointer, we obtain the object to which the pointer points. When we increment a pointer, we advance the pointer to denote the next element in the array. Before we write programs using pointers, we need to know a bit more about them.

`vector` 的遍历可使用下标或迭代器实现，同理，也可用下标或指针来遍历数组。指针是指向某种类型对象的复合数据类型，是用于数组的迭代器：指向数组中的一个元素。在指向数组元素的指针上使用解引用操作符 `*` (dereference operator) 和自增操作符 `++` (increment operator)，与在迭代器上的用法类似。对指针进行解引用操作，可获得该指针所指对象的值。而当指针做自增操作时，则移动指针使其指向数组中的下一个元素。在使用指针编写程序之前，我们需进一步了解一下指针。

4.2.1. What Is a Pointer?

4.2.1. 什么是指针

For newcomers, pointers are often hard to understand. Debugging problems due to pointer errors bedevil even experienced programmers. However, pointers are an important part of most C programs and to a much lesser extent remain important in many C++ programs.

对初学者来说，指针通常比较难理解。而由指针错误引起的调试问题连富有经验的程序员都感到头疼。然而，指针是大多数C程序的重要部分，而且在许多 C++ 程序中仍然受到重用。

Conceptually, pointers are simple: A pointer points at an object. Like an iterator, a pointer offers indirect access to the object to which it points. However, pointers are a much more general construct. Unlike iterators, pointers can be used to point at single objects. Iterators are used only to access elements in a container.

指针的概念很简单：指针用于指向对象。与迭代器一样，指针提供对其所指对象的间接访问，只是指针结构更通用一些。与迭代器不同的是，指针用于指向单个对象，而迭代器只能用于访问容器内的元素。

Specifically, a pointer holds the address of another object:

具体来说，指针保存的是另一个对象的地址：

```
string s("hello world");
string *sp = &s; // sp holds the address of s
```

The second statement defines `sp` as a pointer to `string` and initializes `sp` to point to the `string` object named `s`. The `*` in `*sp` indicates that `sp` is a pointer. The `&` operator in `&s` is the **address-of** operator. It returns a value that when dereferenced yields the original object. The address-of operator may be applied only to an lvalue (Section 2.3.1, p. 45). Because a variable is an lvalue, we may take its address. Similarly, the subscript and dereference operators, when applied to a `vector`, `string`, or built-in array, yield lvalues. Because these operators yield lvalues, we may apply the address-of to the result of the subscript or dereference operator. Doing so gives us the address of a particular element.

第二条语句定义了一个指向 `string` 类型的指针 `sp`，并初始化 `sp` 使其指向 `string` 类型的对象 `s`。`*sp` 中的 `*` 操作符表明 `sp` 是一个指针变量，`&s` 中的 `&` 符号是取地址操作符，当此操作符用于一个对象上时，返回的是该对象的存储地址。取地址操作符只能用于左值（第 2.3.1 节），因为只有当变量用作左值时，才能取其地址。同样地，由于用于 `vector` 类型、`string` 类型或内置数组的下标操作和解引用操作生成左值，因此可对这两种操作的结果做取地址操作，这样即可获取某一特定对象的存储地址。

Advice: Avoid Pointers and Arrays

建议：尽量避免使用指针和数组

Pointers and arrays are surprisingly error-prone. Part of the problem is conceptual: Pointers are used for low-level manipulations and it is easy to make bookkeeping mistakes. Other problems arise because of the syntax, particularly the declaration syntax used with pointers.

指针和数组容易产生不可预料的错误。其中一部分是概念上的问题：指针用于低级操作，容易产生与繁琐细节相关的 (**bookkeeping**) 错误。其他错误则源于使用指针的语法规则，特别是声明指针的语法。

Many useful programs can be written without needing to use arrays or pointers. Instead, modern C++ programs should use `vectors` and iterators to replace general arrays and `strings` to replace C-style array-based character strings.

许多有用的程序都可不使用数组或指针实现，现代C++程序采用`vector`类型和迭代器取代一般的数组、采用`string`类型取代C风格字符串。

4.2.2. Defining and Initializing Pointers

4.2.2. 指针的定义和初始化

Every pointer has an associated type. The type of a pointer determines the type of the objects to which the pointer may point. A pointer to `int`, for example, may only point to an object of type `int`.

每个指针都有一个与之关联的数据类型，该数据类型决定了指针所指向的对象的类型。例如，一个 `int` 型指针只能指向 `int` 型对象。

Defining Pointer Variables

指针变量的定义

We use the `*` symbol in a declaration to indicate that an identifier is a pointer:

C++ 语言使用 `*` 符号把一个标识符声明为指针：

```
vector<int> *pvec;      // pvec can point to a vector<int>
int *ip1, *ip2; // ip1 and ip2 can point to an int
string *pstring; // pstring can point to a string
double *dp; // dp can point to a double
```



When attempting to understand pointer declarations, read them from right to left.

理解指针声明语句时，请从右向左阅读。

Reading the definition of `pstring` from right to left, we see that

从右向左阅读 `pstring` 变量的定义，可以看到

```
string *pstring;
```

defines `pstring` as a pointer that can point to `string` objects. Similarly,

语句把 `pstring` 定义为一个指向 `string` 类型对象的指针变量。类似地，语句

```
int *ip1, *ip2; // ip1 and ip2 can point to an int
```

defines `ip2` as a pointer and `ip1` as a pointer. Both pointers point to `int`s.

把 `ip1` 和 `ip2` 都定义为指向 `int` 型对象的指针。

The `*` can come anywhere in a list of objects of a given type:

在声明语句中，符号 `*` 可用在指定类型的对象列表的任何位置：

```
double dp, *dp2; // dp2 is a pointer, dp is an object: both type double
```

defines `dp2` as a pointer and `dp` as an object, both of type `double`.

该语句定义了一个 `double` 类型的 `dp` 对象以及一个指向 `double` 类型对象的指针 `dp2`。

Section 4.2. Introducing Pointers

A Different Pointer Declaration Style

另一种声明指针的风格

The `*` symbol may be separated from its identifier by a space. It is legal to write:

在定义指针变量时，可用空格将符号 `*` 与其后的标识符分隔开来。下面的写法是合法的：

```
string* ps; // legal but can be misleading
```

which says that `ps` is a pointer to `string`.

也就是说，该语句把 `ps` 定义为一个指向 `string` 类型对象的指针。

We say that this definition can be misleading because it encourages the belief that `string*` is the type and any variable defined in the same definition is a pointer to `string`. However,

这种指针声明风格容易引起这样的误解：把 `string*` 理解为一种数据类型，认为在同一声明语句中定义的其他变量也是指向 `string` 类型对象的指针。然而，语句

```
string* ps1, ps2; // ps1 is a pointer to string, ps2 is a string
```

defines `ps1` as a pointer, but `ps2` is a plain `string`. If we want to define two pointers in a single definition, we must repeat the `*` on each identifier:

实际上只把 `ps1` 定义为指针，而 `ps2` 并非指针，只是一个普通的 `string` 对象而已。如果需要在一个声明语句中定义两个指针，必须在每个变量标识符前再加符号 `*` 声明：

```
string* ps1, *ps2; // both ps1 and ps2 are pointers to string
```

Multiple Pointer Declarations Can Be Confusing

连续声明多个指针易导致混淆

There are two common styles for declaring multiple pointers of the same type. One style requires that a declaration introduce only a single name. In this style, the `*` is placed with the type to emphasize that the declaration is declaring a pointer:

连续声明同一类型的多个指针有两种通用的声明风格。其中一种风格是一个声明语句只声明一个变量，此时，符号 `*` 紧挨着类型名放置，强调这个声明语句定义的是一个指针：

```
string* ps1;
string* ps2;
```

The other style permits multiple declarations in a single statement but places the `*` adjacent to the identifier. This style emphasizes that the object is a pointer:

另一种风格则允许在一条声明语句中声明多个指针，声明时把符号 `*` 靠近标识符放置。这种风格强调对象是一个指针：

```
string *ps1, *ps2;
```



As with all questions of style, there is no single right way to declare pointers. The important thing is to choose a style and stick with it.

关于指针的声明，不能说哪种声明风格是唯一正确的方式，重要的是选择一种风格并持续使用。

In this book we use the second style and place the `*` with the pointer variable name.

在本书中，我们将采用第二种声明风格：将符号 `*` 紧贴着指针变量名放置。

Possible Pointer Values

指针可能的取值

A valid pointer has one of three states: It can hold the address of a specific object, it can point one past the end of an object, or it can be zero. A zero-valued pointer points to no object. An uninitialized pointer is invalid until it is assigned a value. The following definitions and assignments are all legal:

Section 4.2. Introducing Pointers

一个有效的指针必然是以下三种状态之一：保存一个特定对象的地址；指向某个对象后面的另一对象；或者是0值。若指针保存0值，表明它不指向任何对象。未初始化的指针是无效的，直到给该指针赋值后，才可使用它。下列定义和赋值都是合法的：

```
int ival = 1024;
int *pi = 0;           // pi initialized to address no object
int *pi2 = & ival; // pi2 initialized to address of ival
int *pi3;             // ok, but dangerous, pi3 is uninitialized
pi = pi2;             // pi and pi2 address the same object, e.g. ival
pi2 = 0;              // pi2 now addresses no object
```

Avoid Uninitialized Pointers

避免使用未初始化的指针



Uninitialized pointers are a common source of run-time errors.

很多运行时错误都源于使用了未初始化的指针。

As with any other uninitialized variable, what happens when we use an uninitialized pointer is undefined. Using an uninitialized pointer almost always results in a run-time crash. However, the fact that the crash results from using an uninitialized pointer can be quite hard to track down.

就像使用其他没有初始化的变量一样，使用未初始化的指针时的行为C++标准中并没有定义使用未初始化的指针，它几乎总会导致运行时崩溃。然而，导致崩溃的这一原因很难发现。

Under most compilers, if we use an uninitialized pointer the effect will be to use whatever bits are in the memory in which the pointer resides as if it were an address. Using an uninitialized pointer uses this supposed address to manipulate the underlying data at that supposed location. Doing so usually leads to a crash as soon as we attempt to dereference the uninitialized pointer.

对大多数的编译器来说，如果使用未初始化的指针，会将指针中存放的不确定值视为地址，然后操纵该内存地址中存放的位内容。使用未初始化的指针相当于操纵这个不确定地址中存储的基础数据。因此，在对未初始化的指针进行解引用时，通常会导致程序崩溃。

It is not possible to detect whether a pointer is uninitialized. There is no way to distinguish a valid address from an address formed from the bits that are in the memory in which the pointer was allocated. Our recommendation to initialize all variables is particularly important for pointers.

C++ 语言无法检测指针是否未被初始化，也无法区分有效地址和由指针分配到的存储空间中存放的二进制位形成的地址。建议程序员在使用之前初始化所有的变量，尤其是指针。



If possible, do not define a pointer until the object to which it should point has been defined. That way, there is no need to define an uninitialized pointer.

如果可能的话，除非所指向的对象已经存在，否则不要先定义指针，这样可避免定义一个未初始化的指针。

If you must define a pointer separately from pointing it at an object, then initialize the pointer to zero. The reason is that a zero-valued pointer can be tested and the program can detect that the pointer does not point to an object.

如果必须分开定义指针和其所指向的对象，则将指针初始化为0。因为编译器可检测出0值的指针，程序可判断该指针并未指向一个对象。

Constraints on Initialization of and Assignment to Pointers

指针初始化和赋值操作的约束

There are only four kinds of values that may be used to initialize or assign to a pointer:

Section 4.2. Introducing Pointers

对指针进行初始化或赋值只能使用以下四种类型的值：

1. A constant expression ([Section 2.7](#), p. 62) with value 0 (e.g., a `const` integral object whose value is zero at compile time or a literal constant 0)
0 值常量表达式 ([第 2.7 节](#))，例如，在编译时可获得 0 值的整型 `const` 对象或字面值常量 0。
2. An address of an object of an appropriate type
类型匹配的对象的地址。
3. The address one past the end of another object
另一对象末的下一地址。
4. Another valid pointer of the same type
同类型的另一个有效指针。

It is illegal to assign an `int` to a pointer, even if the value of the `int` happens to be 0. It is okay to assign the literal 0 or a `const` whose value is known to be 0 at compile time:

把 `int` 型变量赋给指针是非法的，尽管此 `int` 型变量的值可能为 0。但允许把数值 0 或在编译时可获得 0 值的 `const` 量赋给指针：

```
int ival;
int zero = 0;
const int c_ival = 0;
int *pi = ival; // error: pi initialized from int value of ival
pi = zero;      // error: pi assigned int value of zero
pi = c_ival;    // ok: c_ival is a const with compile-time value of 0
pi = 0;         // ok: directly initialize to literal constant 0
```

In addition to using a literal 0 or a `const` with a compile-time value of 0, we can also use a facility that C++ inherits from C. The `cstdlib` header defines a preprocessor variable ([Section 2.9.2](#), p. 69) named `NULL`, which is defined as 0. When we use a preprocessor variable in our code, it is automatically replaced by its value. Hence, initializing a pointer to `NULL` is equivalent to initializing it to 0:

除了使用数值 0 或在编译时值为 0 的 `const` 量外，还可以使用 C++ 语言从 C 语言中继承下来的预处理器变量 `NULL` ([第 2.9.2 节](#))，该变量在 `cstdlib` 头文件中定义，其值为 0。如果在代码中使用了这个预处理器变量，则编译时会自动被数值 0 替换。因此，把指针初始化为 `NULL` 等效于初始化为 0 值：

```
// cstdlib #defines NULL to 0
int *pi = NULL; // ok: equivalent to int *pi = 0;
```

As with any preprocessor variable ([Section 2.9.2](#), p. 71) we should not use the name `NULL` for our own variables.

正如其他的预处理器变量一样 ([第 2.9.2 节](#))，不可以使用 `NULL` 这个标识符给自定义的变量命名。



Preprocessor variables are not defined in the `std` namespace and hence the name is `NULL`, not `std::NULL`.

预处理器变量不是在 `std` 命名空间中定义的，因此其名字应为 `NULL`，而非 `std::NULL`。

With two exceptions, which we cover in [Sections 4.2.5](#) and [15.3](#), we may only initialize or assign a pointer from an address or another pointer that has the same type as the target pointer:

除了将在[第 4.2.5 节](#)和[第 15.3 节](#)介绍的两种例外情况之外，指针只能初始化或赋值为同类型的变量地址或另一指针：

```
double dval;
double *pd = &dval; // ok: initializer is address of a double
double *pd2 = pd; // ok: initializer is a pointer to double

int *pi = pd; // error: types of pi and pd differ
pi = &dval; // error: attempt to assign address of a double to int *
```

The reason the types must match is that the type of the pointer is used to determine the type of the object that it addresses. Pointers are used to indirectly access an object. The operations that the pointer can perform are based on the type of the pointer: A pointer to `int` treats the underlying object as if it were an `int`. If that pointer actually addressed an object of some other type, such as `double`, then any operations performed by the pointer would be in error.

由于指针的类型用于确定指针所指对象的类型，因此初始化或赋值时必须保证类型匹配。指针用于间接访问对象，并基于指针的类型提供可执行的操作，例如，`int` 型指针只能把其指向的对象当作 `int` 型数据来处理，如果该指针确实指向了其他类型（如 `double` 类型）的对象，则在指针上执行的任何操作都有可能出错。

void* Pointers**void*** 指针

The type **void*** is a special pointer type that can hold an address of any object:

C++ 提供了一种特殊的指针类型 **void***, 它可以保存任何类型对象的地址:

```
double obj = 3.14;
double *pd = &obj;
// ok: void* can hold the address value of any data pointer type
void *pv = &obj;           // obj can be an object of any type
pv = pd;                  // pd can be a pointer to any type
```

A **void*** indicates that the associated value is an address but that the type of the object at that address is unknown.

void* 表明该指针与一地址值相关, 但不清楚存储在此地址上的对象的类型。

There are only a limited number of actions we can perform on a **void*** pointer: We can compare it to another pointer, we can pass or return it from a function, and we can assign it to another **void*** pointer. We cannot use the pointer to operate on the object it addresses. We'll see in [Section 5.12.4](#) (p. 183) how we can retrieve the address stored in a **void*** pointer.

void* 指针只支持几种有限的操作: 与另一个指针进行比较; 向函数传递 **void*** 指针或从函数返回 **void*** 指针; 给另一个 **void*** 指针赋值。不允许使用 **void*** 指针操纵它所指向的对象。我们将在[第 5.12.4 节](#)讨论如何重新获取存储在 **void*** 指针中的地址。

Exercises Section 4.2.2

Exercise

4.10: Explain the rationale for preferring the first form of pointer declaration:

下面提供了两种指针声明的形式, 解释宁愿使用第一种形式的原因:

```
int *ip; // good practice
int* ip; // legal but misleading
```

Exercise

4.11: Explain each of the following definitions. Indicate whether any are illegal and if so why.

解释下列声明语句, 并指出哪些是非法的, 为什么?

- (a) int* ip;
- (b) string s, *sp = 0;
- (c) int i; double* dp = &i;
- (d) int* ip, ip2;
- (e) const int i = 0, *p = i;
- (f) string *p = NULL;

Exercise

4.12: Given a pointer, **p**, can you determine whether **p** points to a valid object? If so, how? If not, why not?

已知一指针 **p**, 你可以确定该指针是否指向一个有效的对象吗? 如果可以, 如何确定? 如果不可以, 请说明原因。

Exercise

4.13: Why is the first pointer initialization legal and the second illegal?

下列代码中, 为什么第一个指针的初始化是合法的, 而第二个则不合法?

```
int i = 42;
void *p = &i;
long *lp = &i;
```

4.2.3. Operations on Pointers

4.2.3. 指针操作

Pointers allow indirect manipulation of the object to which the pointer points. We can access the object by dereferencing the pointer. Dereferencing

Section 4.2. Introducing Pointers

a pointer is similar to dereferencing an iterator ([Section 3.4](#), p. 98). The `*` operator (the dereference operator) returns the object to which the pointer points:

指针提供间接操纵其所指对象的功能。与对迭代器进行解引用操作 ([第 3.4 节](#)) 一样，对指针进行解引用可访问它所指的对象，`*` 操作符 (解引用操作符) 将获取指针所指的对象：

```
string s("hello world");
string *sp = &s; // sp holds the address of s
cout << *sp; // prints hello world
```

When we dereference `sp`, we fetch the value of `s`. We hand that value to the output operator. The last statement, therefore, prints the contents of `s` that is, `hello world`.

对 `sp` 进行解引用将获得 `s` 的值，然后用输出操作符输出该值，于是最后一条语句输出了 `s` 的内容 `hello world`。

Dereference Yields an Lvalue

生成左值的解引用操作

The dereference operator returns the lvalue of the underlying object, so we can use it to change the value of the object to which the pointer points:

解引用操作符返回指定对象的左值，利用这个功能可修改指针所指对象的值：

```
*sp = "goodbye"; // contents of s now changed
```

Because we assign to `*sp`, this statement leaves `sp` pointing to `s` and changes the value of `s`.

因为 `sp` 指向 `s`，所以给 `*sp` 赋值也就修改了 `s` 的值。

We can also assign a new value to `sp` itself. Assigning to `sp` causes `sp` to point to a different object:

也可以修改指针 `sp` 本身的价值，使 `sp` 指向另外一个新对象：

```
string s2 = "some value";
sp = &s2; // sp now points to s2
```

We change the value of a pointer by assigning to it directly without dereferencing the pointer.

给指针直接赋值即可修改指针的值——不需要对指针进行解引用。

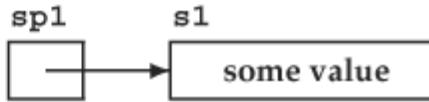
Key Concept: Assigning **TO** or **THROUGH** a Pointer

关键概念：给指针赋值或通过指针进行赋值

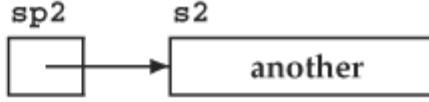
When first using pointers, the difference in whether an assignment is to the pointer or through the pointer to the value pointed to can be confusing. The important thing to keep in mind is that if the left-hand operand is dereferenced, then the value pointed to is changed. If there is no dereference, then the pointer itself is being changed. A picture can sometimes help:

对于初学指针者，给指针赋值和通过指针进行赋值这两种操作的差别确实让人费解。谨记区分的重要方法是：如果对左操作数进行解引用，则修改的是指针所指对象的值；如果没有使用解引用操作，则修改的是指针本身的价值。如图所示，帮助理解下列例子：

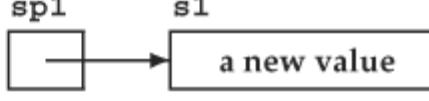
```
string s1("some value");
string *sp1 = &s1;
```



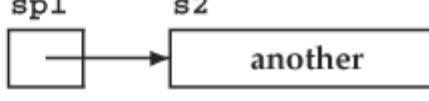
```
string s2("another");
string *sp2 = &s2;
```



```
// assign through sp1
// value in s1 changed
*sp1 = "a new value";
```



```
// assign to sp1
// sp1 points to a different object
sp1 = sp2;
```



Comparing Pointers and References

指针和引用的比较

While both references and pointers are used to indirectly access another value, there are two important differences between references and pointers. The first is that a reference always refers to an object: It is an error to define a reference without initializing it. The behavior of assignment is the second important difference: Assigning to a reference changes the object to which the reference is bound; it does not rebind the reference to another object. Once initialized, a reference *always* refers to the same underlying object.

虽然使用引用 (reference) 和指针都可间接访问另一个值，但它们之间有两个重要区别。第一个区别在于引用总是指向某个对象：定义引用时没有初始化是错误的。第二个重要区别则是赋值行为的差异：给引用赋值修改的是该引用所关联的对象的值，而并不是使引用与另一个对象关联。引用一经初始化，就始终指向同一个特定对象（这就是为什么引用必须在定义时初始化的原因）。

Consider these two program fragments. In the first, we assign one pointer to another:

考虑以下两个程序段。第一个程序段将一个指针赋给另一指针：

```
int ival = 1024, ival2 = 2048;
int *pi = &ival, *pi2 = &ival2;
pi = pi2; // pi now points to ival2
```

After the assignment, `ival`, the object addressed by `pi` remains unchanged. The assignment changes the value of `pi`, making it point to a different object. Now consider a similar program that assigns two references:

赋值结束后，`pi` 所指向的 `ival` 对象值保持不变，赋值操作修改了 `pi` 指针的值，使其指向另一个不同的对象。现在考虑另一段相似的程序，使用两个引用赋值：

```
int &ri = ival, &ri2 = ival2;
ri = ri2; // assigns ival2 to ival
```

This assignment changes `ival`, the value referenced by `ri`, and not the reference itself. After the assignment, the two references still refer to their original objects, and the value of those objects is now the same as well.

这个赋值操作修改了 `ri` 引用的值 `ival` 对象，而并非引用本身。赋值后，这两个引用还是分别指向原来关联的对象，此时这两个对象的值相等。

Pointers to Pointers

指向指针的指针

Pointers are themselves objects in memory. They, therefore, have addresses that we can store in a pointer:

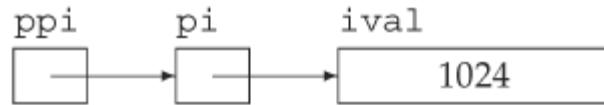
指针本身也是可用指针指向的内存对象。指针占用内存空间存放其值，因此指针的存储地址可存放在指针中。下面程序段：

```
int ival = 1024;
int *pi = &ival; // pi points to an int
int **ppi = &pi; // ppi points to a pointer to int
```

which yields a pointer to a pointer. We designate a pointer to a pointer by using `**`. We might represent these objects as

Section 4.2. Introducing Pointers

定义了指向指针的指针。C++ 使用 `**` 操作符指派一个指针指向另一指针。这些对象可表示为：



As usual, dereferencing `ppi` yields the object to which `ppi` points. In this case, that object is a pointer to an `int`:

对 `ppi` 进行解引用照常获得 `ppi` 所指的对象，在本例中，所获得的对象是指向 `int` 型变量的指针 `pi`：

```
int *pi2 = *ppi; // ppi points to a pointer
```

To actually access `ival`, we need to dereference `ppi` twice:

为了真正地访问到 `ival` 对象，必须对 `ppi` 进行两次解引用：

```
cout << "The value of ival\n"
     << "direct value: " << ival << "\n"
     << "indirect value: " << *pi << "\n"
     << "doubly indirect value: " << **ppi
     << endl;
```

This program prints the value of `ival` three different ways. First, by direct reference to the variable. Then, through the pointer to `int` in `pi`, and finally, by dereferencing `ppi` twice to get to the underlying value in `ival`.

这段程序用三种不同的方式输出 `ival` 的值。首先，采用直接引用变量的方式输出；然后使用指向 `int` 型对象的指针 `pi` 输出；最后，通过对 `ppi` 进行两次解引用获得 `ival` 的特定值。

Exercises Section 4.2.3

Exercise 4.14: Write code to change the value of a pointer. Write code to change the value to which the pointer points.

编写代码修改指针的值；然后再编写代码修改指针所指对象的值。

Exercise 4.15: Explain the key differences between pointers and references.

解释指针和引用的主要区别。

Exercise 4.16: What does the following program do?

下列程序段实现什么功能？

```
int i = 42, j = 1024;
int *p1 = &i, *p2 = &j;
*p2 = *p1 * *p2;
*p1 *= *p1;
```

4.2.4. Using Pointers to Access Array Elements

4.2.4. 使用指针访问数组元素

Pointers and arrays are closely intertwined in C++. In particular, when we use the name of an array in an expression, that name is automatically converted into a pointer to the first element of the array:

C++ 语言中，指针和数组密切相关。特别是在表达式中使用数组名时，该名字会自动转换为指向数组第一个元素的指针：

```
int ia[] = {0,2,4,6,8};
int *ip = ia; // ip points to ia[0]
```

If we want to point to another element in the array, we could do so by using the subscript operator to locate the element and then applying the address-of operator to find its location:

如果希望使指针指向数组中的另一个元素，则可使用下标操作符给某个元素定位，然后用取地址操作符 `&` 获取该元素的存储地址：

Section 4.2. Introducing Pointers

```
ip = &ia[4]; // ip points to last element in ia
```

Pointer Arithmetic

指针的算术操作

Rather than taking the address of the value returned by subscripting, we could use **pointer arithmetic**. Pointer arithmetic works the same way (and has the same constraints) as iterator arithmetic ([Section 3.4.1](#), p. 100). Using pointer arithmetic, we can compute a pointer to an element by adding (or subtracting) an integral value to (or from) a pointer to another element in the array:

与其使用下标操作，倒不如通过**指针的算术操作**来获取指定内容的存储地址。指针的算术操作和迭代器的算术操作（[第 3.4.1 节](#)）以相同的方式实现（也具有相同的约束）。使用指针的算术操作在指向数组某个元素的指针上加上（或减去）一个整型数值，就可以计算出指向数组另一元素的指针值：

```
ip = ia; // ok: ip points to ia[0]
int *ip2 = ip + 4; // ok: ip2 points to ia[4], the last element in ia
```

When we add 4 to the pointer `ip`, we are computing a new pointer. That new pointer points to the element four elements further on in the array from the one to which `ip` currently points.

在指针 `ip` 上加 4 得到一个新的指针，指向数组中 `ip` 当前指向的元素后的第 4 个元素。

More generally, when we add (or subtract) an integral value to a pointer, the effect is to compute a new pointer. The new pointer points to the element as many elements as that integral value ahead of (or behind) the original pointer.

通常，在指针上加上（或减去）一个整型数值 `n` 等效于获得一个新指针，该新指针指向指针原来指向的元素之后（或之前）的第 `n` 个元素。



Pointer arithmetic is legal only if the original pointer and the newly calculated pointer address elements of the same array or an element one past the end of that array. If we have a pointer to an object, we can also compute a pointer that points just after that object by adding one to the pointer.

指针的算术操作只有在原指针和计算出来的新指针都指向同一个数组的元素，或指向该数组存储空间的下一单元时才是合法的。如果指针指向一对象，我们还可以在指针上加1从而获取指向相邻的下一个对象的指针。

Given that `ia` has 4 elements, adding 10 to `ia` would be an error:

假设数组 `ia` 只有 4 个元素，则在 `ia` 上加 10 是错误的：

```
// error: ia has only 4 elements, ia + 10 is an invalid address
int *ip3 = ia + 10;
```

We can also subtract two pointers as long as they point into the same array or to an element one past the end of the array:

只要两个指针指向同一数组或有一个指向该数组末端的下一单元，C++ 还支持对这两个指针做减法操作：

```
ptrdiff_t n = ip2 - ip; // ok: distance between the pointers
```

The result is four, the distance between the two pointers, measured in objects. The result of subtracting two pointers is a library type named **ptrdiff_t**. Like `size_t`, the `ptrdiff_t` type is a machine-specific type and is defined in the `cstdint` header. The `size_t` type is an `unsigned` type, whereas `ptrdiff_t` is a `signed` integral type.

结果是 4，这两个指针所指向的元素间隔为 4 个对象。两个指针减法操作的结果是标准库类型（library type）**ptrdiff_t** 的数据。与 `size_t` 类型一样，`ptrdiff_t` 也是一种与机器相关的类型，在 `cstdint` 头文件中定义。`size_t` 是 `unsigned` 类型，而 `ptrdiff_t` 则是 `signed` 整型。

The difference in type reflects how these two types are used: `size_t` is used to hold the size of an array, which must be a positive value. The `ptrdiff_t` type is guaranteed to be large enough to hold the difference between any two pointers into the same array, which might be a negative value. For example, had we subtracted `ip2` from `ip`, the result would be `-4`.

这两种类型的差别体现了它们各自的用途：`size_t` 类型用于指明数组长度，它必须是一个正数；`ptrdiff_t` 类型则应保证足以存放同一数组中两个指针之间的差距，它有可能是负数。例如，`ip` 减去 `ip2`，结果为 `-4`。

It is always possible to add or subtract zero to a pointer, which leaves the pointer unchanged. More interestingly, given a pointer that has a value of zero, it is also legal to add zero to that pointer. The result is another zero-valued pointer. We can also subtract two pointers that have a value of zero. The result of subtracting two zero-valued pointers is zero.

允许在指针上加减 0，使指针保持不变。更有趣的是，如果一指针具有 0 值（空指针），则在该指针上加 0 仍然是合法的，结果得到另一个值为 0 的指针。也可以对两个空指针做减法操作，得到的结果仍是 0。

Interaction between Dereference and Pointer Arithmetic

解引用和指针算术操作之间的相互作用

The result of adding an integral value to a pointer is itself a pointer. We can dereference the resulting pointer directly without first assigning it to another pointer:

在指针上加一个整型数值，其结果仍然是指针。允许在这个结果上直接进行解引用操作，而不必先把它赋给一个新指针：

```
int last = *(ia + 4); // ok: initializes last to 8, the value of ia[4]
```

This expression calculates the address four elements past `ia` and dereferences that pointer. It is equivalent to writing `ia[4]`.

这个表达式计算出 `ia` 所指向元素后面的第 4 个元素的地址，然后对该地址进行解引用操作，等价于 `ia[4]`。



The parentheses around the addition are essential. Writing

加法操作两边用圆括号括起来是必要的。如果写为：

```
last = *ia + 4; // ok: last = 4, equivalent to ia[0]+4
```

means dereference `ia` and add four to the dereferenced value.

意味着对 `ia` 进行解引用，获得 `ia` 所指元素的值 `ia[0]`，然后加 4。

The parentheses are required due to the [precedence](#) of the addition and dereference operators. We'll learn more about precedence in [Section 5.10.1](#) (p. 168). Simply put, precedence stipulates how operands are grouped in expressions with multiple operators. The dereference operator has a higher precedence than the addition operator.

由于加法操作和解引用操作的[优先级](#)不同，上述表达式中的圆括号是必要的。我们将在[第 5.10.1 节](#)讨论操作符的优先级。简单地说，优先级决定了有多个操作符的表达式如何对操作数分组。解引用操作符的优先级比加法操作符高。

The operands to operators with higher precedence are grouped more tightly than those of lower precedence. Without the parentheses, the dereference operator would use `ia` as its operand. The expression would be evaluated by dereferencing `ia` and adding four to the value of the element at the beginning of `ia`.

与低优先级的操作符相比，优先级高的操作符的操作数先被组合起来操作。如果没有圆括号，解引用操作符的操作数是 `ia`，该表达式先对 `ia` 解引用，获得 `ia` 数组中的第一个元素，并将该值与 4 相加。

By parenthesizing the expression, we override the normal precedence rules and effectively treat `(ia + 4)` as a single operand. That operand is an address of an element four past the one to which `ia` points. That new address is dereferenced.

如果表达式加上圆括号，则不管一般的优先级规则，将 `(ia + 4)` 作为单个操作数，这是 `ia` 所指向的元素后面第4个元素的地址，然后对这个新地址进行解引用。

Subscripts and Pointers

下标和指针

We have already seen that when we use an array name in an expression, we are actually using a pointer to the first element in the array. This fact has a number of implications, which we shall point out as they arise.

我们已经看到，在表达式中使用数组名时，实际上使用的是指向数组第一个元素的指针。这种用法涉及很多方面，当它们出现时我们会逐一指出来。

One important implication is that when we subscript an array, we are really subscripting a pointer:

其中一个重要的应用是使用下标访问数组时，实际上是使用下标访问指针：

```
int ia[] = {0,2,4,6,8};
int i = ia[0]; // ia points to the first element in ia
```

When we write `ia[0]`, that is an expression that uses the name of an array. When we subscript an array, we are really subscripting a pointer to an element in that array. We can use the subscript operator on any pointer, as long as that pointer points to an element in an array:

Section 4.2. Introducing Pointers

`ia[0]` 是一个使用数组名的表达式。在使用下标访问数组时，实际上是对指向数组元素的指针做下标操作。只要指针指向数组元素，就可以对它进行下标操作：

```
int *p = &ia[2];      // ok: p points to the element indexed by 2
int j = p[1];        // ok: p[1] equivalent to *(p + 1),
                     //     p[1] is the same element as ia[3]
int k = p[-2];       // ok: p[-2] is the same element as ia[0]
```

Computing an Off-the-End Pointer

计算数组的超出末端指针

When we use a `vector`, the `end` operation returns an iterator that refers just past the end of the `vector`. We often use this iterator as a sentinel to control loops that process the elements in the `vector`. Similarly, we can compute an off-the-end pointer value:

`vector` 类型提供的 `end` 操作将返回指向超出 `vector` 末端位置的一个迭代器。这个迭代器常用作哨兵，来控制处理 `vector` 中元素的循环。类似地，可以计算数组的超出末端指针的值：

```
const size_t arr_size = 5;
int arr[arr_size] = {1,2,3,4,5};
int *p = arr;           // ok: p points to arr[0]
int *p2 = p + arr_size; // ok: p2 points one past the end of arr
                       // use caution -- do not dereference!
```

In this case, we set `p` to point to the first element in `arr`. We then calculate a pointer one past the end of `arr` by adding the size of `arr` to the pointer value in `p`. When we add 5 to `p`, the effect is to calculate the address of that is five `ints` away from `p` in other words, `p + 5` points just past the end of `arr`.

本例中，`p` 指向数组 `arr` 的第一个元素，在指针 `p` 上加数组长度即可计算出数组 `arr` 的超出末端指针。`p` 加 5 即得 `p` 所指向的元素后面的第五个 `int` 元素的地址——换句话说，`p + 5` 指向数组的超出末端的位置。



It is legal to compute an address one past the end of an array or object. It is not legal to dereference a pointer that holds such an address. Nor is it legal to compute an address more than one past the end of an array or an address before the beginning of an array.

C++ 允许计算数组或对象的超出末端的地址，但不允许对此地址进行解引用操作。而计算数组超出末端位置之后或数组首地址之前的地址都是不合法的。

The address we calculated and stored in `p2` acts much like the iterator returned from the `end` operation on `vectors`. The iterator we obtain from `end` denotes "one past the end" of the `vector`. We may not dereference that iterator, but we may compare it to another iterator value to see whether we have processed all the elements in the `vector`. Similarly, the value we calculated for `p2` can be used *only* to compare to another pointer value or as an operand in a pointer arithmetic expression. If we attempt to dereference `p2`, the most likely result is that it would yield some garbage value. Most compilers, would treat the result of dereferencing `p2` as an `int`, using whatever bits happened to be in memory at the location just after the last element in `arr`.

计算并存储在 `p2` 中的地址，与在 `vector` 上做 `end` 操作所返回的迭代器具有相同的功能。由 `end` 返回的迭代器标志了该 `vector` 对象的“超出末端位置”，不能进行解引用运算，但是可将它与别的迭代器比较，从而判断是否已经处理完 `vector` 中所有的元素。同理，`p2` 也只能用来与其他指针比较，或者用做指针算术操作表达式的操作数。对 `p2` 进行解引用将得到无效值。对大多数的编译器来说，会把对 `p2` 进行解引用的结果（恰好存储在 `arr` 数组的最后一个元素后面的内存中的二进制位）视为一个 `int` 型数据。

Printing the Elements of an Array

输出数组元素

Now we are ready to write a program that uses pointers:

用指针编写以下程序：

```
const size_t arr_sz = 5;
int int_arr[arr_sz] = { 0, 1, 2, 3, 4 };
// pbegin points to first element, pend points just after the last
for (int *pbegin = int_arr, *pend = int_arr + arr_sz;
     pbegin != pend; ++pbegin)
    cout << *pbegin << ' '; // print the current element
```

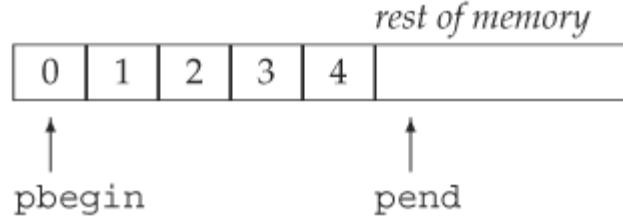
Section 4.2. Introducing Pointers

This program uses a feature of the `for` loop that we have not yet used: We may define multiple variables inside the `init-statement` ([Section 1.4.2, p. 14](#)) of a `for` as long as the variables are defined using the same type. In this case, we're defining two `int` pointers named `pbegin` and `pend`.

这段程序使用了一个我们以前没有用过的 `for` 循环性质：只要定义的多个变量具有相同的类型，就可以在 `for` 循环的初始化语句（[第 1.4.2 节](#)）中同时定义它们。本例在初始化语句中定义了两个 `int` 型指针 `pbegin` 和 `pend`。

We use these pointers to traverse the array. Like other built-in types, arrays have no member functions. Hence, there are no `begin` and `end` operations on arrays. Instead, we must position pointers to denote the first and one past the last elements ourselves. We do so in the initialization of our two pointers. We initialize `pbegin` to address the first element of `int_arr` and `pend` to one past the last element in the array:

C++ 允许使用指针遍历数组。和其他内置类型一样，数组也没有成员函数。因此，数组不提供 `begin` 和 `end` 操作，程序员只能自己给指针定位，使之分别标志数组的起始位置和超出末端位置。可在初始化中实现这两个指针的定位：初始化指针 `pbegin` 指向 `int_arr` 数组的第一个元素，而指针 `pend` 则指向该数组的超出末端的位置：



The pointer `pend` serves as a sentinel, allowing the `for` loop to know when to stop. Each iteration of the `for` loop increments `pbegin` to address the next element. On the first trip through the loop, `pbegin` denotes the first element, on the second iteration, the second element, and so on. After processing the last element in the array, `pbegin` will be incremented once more and will then equal `pend`. At that point we know that we have iterated across the entire array.

指针 `pend` 是标志 `for` 循环结束的哨兵。`for` 循环的每次迭代都会使 `pbegin` 递增 1 以指向数组的下一个元素。第一次执行 `for` 循环时，`pbegin` 指向数组中的第一个元素；第二次循环，指向第二个元素；这样依次类推。当处理完数组的最后一个元素后，`pbegin` 再加 1 则与 `pend` 值相等，表示整个数组已遍历完毕。

Pointers Are Iterators for Arrays

指针是数组的迭代器

Astute readers will note that this program is remarkably similar to the program on page [99](#), which traversed and printed the contents of a `vector` of `strings`. The loop in that program

聪明的读者可能已经注意到这段程序与[第 3.4 节](#)的一段程序非常相像，该程序使用下面的循环遍历并输出一个 `string` 类型的 `vector` 的内容：

```
// equivalent loop using iterators to reset all the elements in ivec to 0
for (vector<int>::iterator iter = ivec.begin();
     iter != ivec.end(); ++iter)
    *iter = 0; // set element to which iter refers to 0
```

used iterators in much the same way that pointers are used in the program to print the contents of the array. This similarity is not a coincidence. In fact, the built-in array type has many of the properties of a library container, and pointers, when we use them in conjunction with arrays, are themselves iterators. We'll have much more to say about containers and iterators in [Part II](#).

这段程序使用迭代器的方式就像上个程序使用指针实现输出数组内容一样。指针和迭代器的这个相似之处并不是巧合。实际上，内置数组类型具有标准库容器的许多性质，与数组联合使用的指针本身就是迭代器。在第二部分中，我们还会详细介绍容器和迭代器类型。

Exercises Section 4.2.4

Exercise 4.17: Given that `p1` and `p2` point to elements in the same array, what does the following statement do?

已知 `p1` 和 `p2` 指向同一个数组中的元素，下面语句实现什么功能？

```
p1 += p2 - p1;
```

Are there any values of `p1` or `p2` that could make this code illegal?

当 `p1` 和 `p2` 具有什么值时这个语句是非法的？

Exercise 4.18: Write a program that uses pointers to set the elements in an array of `ints` to zero.

编写程序，使用指针把一个 `int` 型数组的所有元素设置为0。

4.2.5. Pointers and the `const` Qualifier

4.2.5. 指针和 `const` 限定符

There are two kinds of interactions between pointers and the `const` qualifier discussed in [Section 2.4](#) (p. 56): We can have pointers to `const` objects and pointers that are themselves `const`. This section discusses both kinds of pointers.

[第 2.4 节](#)介绍了指针和 `const` 限定符之间的两种交互类型：指向 `const` 对象的指针和 `const` 指针。我们在本节中详细讨论这两类指针。

Pointers to `const` Objects

指向 `const` 对象的指针

The pointers we've seen so far can be used to change the value of the objects to which they point. But if we have a pointer to a `const` object, we do not want to allow that pointer to change the underlying, `const` value. The language enforces this property by requiring that pointers to `const` objects must take the `constness` of their target into account:

到目前为止，我们使用指针来修改其所指对象的值。但是如果指针指向 `const` 对象，则不允许用指针来改变其所指的 `const` 值。为了保证这个特性，C++ 语言强制要求指向 `const` 对象的指针也必须具有 `const` 特性：

```
const double *cptr; // cptr may point to a double that is const
```

Here `cptr` is a pointer to an object of type `const double`. The `const` qualifies the type of the object to which `cptr` points, not `cptr` itself. That is, `cptr` itself is not `const`. We need not initialize it and can assign a new value to it if we so desire. What we cannot do is use `cptr` to change the value to which it points:

这里的 `cptr` 是一个指向 `double` 类型 `const` 对象的指针，`const` 限定了 `cptr` 指针所指向的对象类型，而并非 `cptr` 本身。也就是说，`cptr` 本身并不是 `const`。在定义时不需要对它进行初始化，如果需要的话，允许给 `cptr` 重新赋值，使其指向另一个 `const` 对象。但不能通过 `cptr` 修改其所指对象的值：

```
*cptr = 42; // error: *cptr might be const
```

It is also a compile-time error to assign the address of a `const` object to a plain, non`const` pointer:

把一个 `const` 对象的地址赋给一个普通的、非 `const` 对象的指针也会导致编译时的错误：

```
const double pi = 3.14;
double *ptr = &pi;           // error: ptr is a plain pointer
const double *cptr = &pi; // ok: cptr is a pointer to const
```

We cannot use a `void*` pointer ([Section 4.2.2](#), p. 119) to hold the address of a `const` object. Instead, we must use the type `const void*` to hold the address of a `const` object:

不能使用 `void*` 指针 ([第 4.2.2 节](#)) 保存 `const` 对象的地址，而必须使用 `const void*` 类型的指针保存 `const` 对象的地址：

```
const int universe = 42;
const void *cpv = &universe; // ok: cpv is const
void *pv = &universe;      // error: universe is const
```

A pointer to a `const` object can be assigned the address of a non`const` object, such as

允许把非 `const` 对象的地址赋给指向 `const` 对象的指针，例如：

```
double dval = 3.14; // dval is a double; its value can be changed
cptr = &dval;       // ok: but can't change dval through cptr
```

Although `dval` is not a `const`, any attempt to modify its value through `cptr` results in a compile-time error. When we declared `cptr`, we said that it would not change the value to which it points. The fact that it happens to point to a non`const` object is irrelevant.

尽管 `dval` 不是 `const` 对象，但任何企图通过指针 `cptr` 修改其值的行为都会导致编译时的错误。`cptr` 一经定义，就不允许修改其所指对象的值。如果该指针恰好指向非 `const` 对象时，同样必须遵循这个规则。



We cannot use a pointer to `const` to change the underlying object. However, if the pointer addresses a non`const` object, it is possible that some other action will change the object to which the pointer points.

Section 4.2. Introducing Pointers

不能使用指向 `const` 对象的指针修改基础对象，然而如果该指针指向的是一个非 `const` 对象，可用其他方法修改其所指的对象。

The fact that values to which a `const` pointer points can be changed is subtle and can be confusing. Consider:

事实是，可以修改 `const` 指针所指向的值，这一点常常容易引起误会。考虑：

```
dval = 3.14159;      // dval is not const
*cptr = 3.14159;    // error: cptr is a pointer to const
double *ptr = &dval; // ok: ptr points at non-const double
*ptr = 2.72;         // ok: ptr is plain pointer
cout << *cptr;      // ok: prints 2.72
```

In this case, `cptr` is defined as a pointer to `const` but it actually points at a non`const` object. Even though the object to which it points is non`const`, we cannot use `cptr` to change the object's value. Essentially, there is no way for `cptr` to know whether the object it points to is `const`, and so it treats all objects to which it might point as `const`.

在此例题中，指向 `const` 的指针 `cptr` 实际上指向了一个非 `const` 对象。尽管它所指的对象并非 `const`，但仍不能使用 `cptr` 修改该对象的值。本质上来说，由于没有方法分辨 `cptr` 所指的对象是否为 `const`，系统会把它所指的所有对象都视为 `const`。

When a pointer to `const` does point to a non`const`, it is possible that the value of the object might change: After all, that value is not `const`. We could either assign to it directly or, as here, indirectly through another, plain non`const` pointer. It is important to remember that there is no guarantee that an object pointed to by a pointer to `const` won't change.

如果指向 `const` 的指针所指的对象并非 `const`，则可直接给该对象赋值或间接地利用普通的非 `const` 指针修改其值：毕竟这个值不是 `const`。重要的是要记住：不能保证指向 `const` 的指针所指对象的值一定不可修改。



It may be helpful to think of pointers to `const` as "pointers that *think* they point to `const`."

如果把指向 `const` 的指针理解为“自以为指向 `const` 的指针”，这可能会对理解有所帮助。

In real-world programs, pointers to `const` occur most often as formal parameters of functions. Defining a parameter as a pointer to `const` serves as a contract guaranteeing that the actual object being passed into the function will not be modified through that parameter.

在实际的程序中，指向 `const` 的指针常用作函数的形参。将形参定义为指向 `const` 的指针，以此确保传递给函数的实际对象在函数中不因为形参而被修改。

`const` Pointers

`const` 指针

In addition to pointers to `const`, we can also have `const` pointers that is, pointers whose own value we may not change:

除指向 `const` 对象的指针外，C++ 语言还提供了 `const` 指针——本身的值不能修改：

```
int errNumb = 0;
int *const curErr = &errNumb; // curErr is a constant pointer
```

Reading this definition from right to left, we see that "curErr is a constant pointer to an object of type `int`." As with any `const`, we may not change the value of the pointer that is, we may not make it point to any other object. Any attempt to assign to a constant pointer even assigning the same value back to `curErr` is flagged as an error during compilation:

我们可以从右向左把上述定义语句读作“`curErr` 是指向 `int` 型对象的 `const` 指针”。与其他 `const` 量一样，`const` 指针的值不能修改，这就意味着不能使 `curErr` 指向其他对象。任何企图给 `const` 指针赋值的行为（即使给 `curErr` 赋回同样的值）都会导致编译时的错误：

```
curErr = curErr; // error: curErr is const
```

As with any `const`, we must initialize a `const` pointer when we create it.

与任何 `const` 量一样，`const` 指针也必须在定义时初始化。

The fact that a pointer is itself `const` says nothing about whether we can use the pointer to change the value to which it points. Whether we can change the value pointed to depends entirely on the type to which the pointer points. For example, `curErr` addresses a plain, non`const` `int`. We can use `curErr` to change the value of `errNumb`:

Section 4.2. Introducing Pointers

指针本身是 `const` 的事实并没有说明是否能使用该指针修改它所指向对象的值。指针所指对象的值能否修改完全取决于该对象的类型。例如，`curErr` 指向一个普通的非常量 `int` 型对象 `errNumb`，则可使用 `curErr` 修改该对象的值：

```
if (*curErr) {
    errorHandler();
    *curErr = 0; // ok: reset value of the object to which curErr is bound
}
```

`const` Pointer to a `const` Object

指向 `const` 对象的 `const` 指针

We can also define a constant pointer to a constant object as follows:

还可以如下定义指向 `const` 对象的 `const` 指针：

```
const double pi = 3.14159;
// pi_ptr is const and points to a const object
const double *const pi_ptr = &pi;
```

In this case, neither the value of the object addressed by `pi_ptr` nor the address itself can be changed. We can read its definition from right to left as "`pi_ptr` is a constant pointer to an object of type `double` defined as `const`."

本例中，既不能修改 `pi_ptr` 所指向对象的值，也不允许修改该指针的指向（即 `pi_ptr` 中存放的地址值）。可从右向左阅读上述声明语句：“`pi_ptr` 首先是一个 `const` 指针，指向 `double` 类型的 `const` 对象”。

Pointers and Typedefs

指针和 `typedef`

The use of pointers in `typedefs` ([Section 2.6](#), p. 61) often leads to surprising results. Here is a question almost everyone answers incorrectly at least once. Given the following,

在 `typedef` ([第 2.6 节](#)) 中使用指针往往会带来意外的结果。下面是一个几乎所有人刚开始时都会答错的问题。假设给出以下语句：

```
typedef string *pstring;
const pstring cstr;
```

what is the type of `cstr`? The simple answer is that it is a pointer to `const pstring`. The deeper question is: what underlying type does a pointer to `const pstring` represent? Many think that the actual type is

请问 `cstr` 变量是什么类型？简单的回答是 `const pstring` 类型的指针。进一步问：`const pstring` 指针所表示的真实类型是什么？很多人都认为真正的类型是：

```
const string *cstr; // wrong interpretation of const pstring cstr
```

That is, that a `const pstring` would be a pointer to a constant `string`. But that is incorrect.

也就是说，`const pstring` 是一种指针，指向 `string` 类型的 `const` 对象，但这是错误的。

The mistake is in thinking of a `typedef` as a textual expansion. When we declare a `const pstring`, the `const` modifies the type of `pstring`, which is a pointer. Therefore, this definition declares `cstr` to be a `const` pointer to `string`. The definition is equivalent to

错误的原因在于将 `typedef` 当做文本扩展了。声明 `const pstring` 时，`const` 修饰的是 `pstring` 的类型，这是一个指针。因此，该声明语句应该是把 `cstr` 定义为指向 `string` 类型对象的 `const` 指针，这个定义等价于：

```
// cstr is a const pointer to string
string *const cstr; // equivalent to const pstring cstr
```

Advice: Understanding Complicated `const` Type Declarations

建议：理解复杂的 `const` 类型的声明

Part of the problem in reading `const` declarations arises because the `const` can go either before or after the type:

阅读 `const` 声明语句产生的部分问题，源于 `const` 限定符既可以放在类型前也可以放在类型后：

```
string const s1; // s1 and s2 have same type,
```

Section 4.2. Introducing Pointers

```
const string s2; // they're both strings that are const
```

When writing `const` definitions using `typedefs`, the fact that the `const` can precede the type can lead to confusion as to the actual type being defined:

用 `typedef` 写 `const` 类型定义时, `const` 限定符加在类型名前面容易引起对所定义的真正类型的误解:

[View full width]

```
string s;
typedef string *pstring;
const pstring cstr1 = &s; // written this way the type
→ is obscured
pstring const cstr2 = &s; // all three declarations are
→ the same type
string *const cstr3 = &s; // they're all const pointers
→ to string
```

Putting the `const` after `pstring` and reading the declaration from right to left makes it clearer that `cstr2` is a `const pstring`, which in turn is a `const` pointer to `string`.

把 `const` 放在类型 `pstring` 之后, 然后从右向左阅读该声明语句就会非常清楚地知道 `cstr2` 是 `const pstring` 类型, 即指向 `string` 对象的 `const` 指针。

Unfortunately, most readers of C++ programs expect to see the `const` before the type. As a result, it is probably a good idea to put the `const` first, respecting common practice. But it can be helpful in understanding declarations to rewrite them to put the `const` after the type.

不幸的是, 大多数人在阅读 C++ 程序时都习惯看到 `const` 放在类型前面。于是为了遵照惯例, 只好建议编程时把 `const` 放在类型前面。但是, 把声明语句重写为置 `const` 于类型之后更便于理解。

Exercises Section 4.3

Exercise

4.19: Explain the meaning of the following five definitions. Identify any illegal definitions.

解释下列 5 个定义的含义, 指出其中哪些定义是非法的:

- (a) int i;
- (b) const int ic;
- (c) const int *pic;
- (d) int *const cpi;
- (e) const int *const cpic;

Exercise

4.20: Which of the following initializations are legal? Explain why.

下列哪些初始化是合法的? 为什么?

- (a) int i = -1;
- (b) const int ic = i;
- (c) const int *pic = ⁣
- (d) int *const cpi = ⁣
- (e) const int *const cpic = ⁣

Exercise

4.21: Based on the definitions in the previous exercise, which of the following assignments are legal?

Explain why.

根据上述定义, 下列哪些赋值运算是合法的? 为什么?

- (a) i = ic;
- (b) pic = ⁣
- (c) cpi = pic;
- (d) pic = cpic;
- (e) cpic = ⁣
- (f) ic = *cpic;

Team LiB

◀ PREVIOUS NEXT ▶

4.3. C-Style Character Strings

4.3. C 风格字符串



Although C++ supports C-style strings, they should not be used by C++ programs. C-style strings are a surprisingly rich source of bugs and are the root cause of many, many security problems.

尽管 C++ 支持 C 风格字符串，但不应该在 C++ 程序中使用这个类型。C 风格字符串常常带来许多错误，是导致大量安全问题的根源。

In [Section 2.2](#) (p. 40) we first used string literals and learned that the type of a string literal is array of constant characters. We can now be more explicit and note that the type of a string literal is an array of `const char`. A string literal is an instance of a more general construct that C++ inherits from C: **C-style character strings**. C-style strings are not actually a type in either C or C++. Instead, C-style strings are null-terminated arrays of characters:

在第 2.2 节中我们第一次使用了字符串字面值，并了解字符串字面值的类型是字符常量的数组，现在可以更明确地认识到：字符串字面值的类型就是 `const char` 类型的数组。C++ 从 C 语言继承下来的一种通用结构是 C 风格字符串，而字符串字面值就是该类型的实例。实际上，C 风格字符串既不能确切地归结为 C 语言的类型，也不能归结为 C++ 语言的类型，而是以空字符 `null` 结束的字符数组：

```
char ca1[] = {'C', '+', '+'};           // no null, not C-style string
char ca2[] = {'C', '+', '+', '\0'};      // explicit null
char ca3[] = "C++";                    // null terminator added automatically
const char *cp = "C++";                // null terminator added automatically
char *cp1 = ca1;                      // points to first element of a array, but not C-style string
char *cp2 = ca2;                      // points to first element of a null-terminated char array
```

Neither `ca1` nor `cp1` are C-style strings: `ca1` is a character array, but the array is not null-terminated. `cp1`, which points to `ca1`, therefore, does not point to a null-terminated array. The other declarations are all C-style strings, remembering that the name of an array is treated as a pointer to the first element of the array. Thus, `ca2` and `ca3` are pointers to the first elements of their respective arrays.

`ca1` 和 `cp1` 都不是 C 风格字符串：`ca1` 是一个不带结束符 `null` 的字符数组，而指针 `cp1` 指向 `ca1`，因此，它指向的并不是以 `null` 结束的数组。其他的声明则都是 C 风格字符串，数组的名字即是指向该数组第一个元素的指针。于是，`ca2` 和 `ca3` 分别是指向各自数组第一个元素的指针。

Using C-style Strings

C 风格字符串的使用

C-style strings are manipulated through (`const`) `char*` pointers. One frequent usage pattern uses pointer arithmetic to traverse the C-style string. The traversal tests and increments the pointer until we reach the terminating null character:

C++ 语言通过(`const`)`char*` 类型的指针来操纵 C 风格字符串。一般来说，我们使用指针的算术操作来遍历 C 风格字符串，每次对指针进行测试并递增 1，直到到达结束符 `null` 为止：

```
const char *cp = "some value";
while (*cp) {
    // do something to *cp
    ++cp;
}
```

The condition in the `while` dereferences the `const char*` pointer `cp` and the resulting character is tested for its `true` or `false` value. A true value is any character other than the null. So, the loop continues until it encounters the null character that terminates the array to which `cp` points. The body of the `while` does whatever processing is needed and concludes by incrementing `cp` to advance the pointer to address the next character in the array.

`while` 语句的循环条件是对 `const char*` 类型的指针 `cp` 进行解引用，并判断 `cp` 当前指向的字符是 `true` 值还是 `false` 值。真值表明这是除 `null` 外的任意字符，则继续循环直到 `cp` 指向结束字符数组的 `null` 时，循环结束。`while` 循环体做完必要的处理后，`cp` 加1，向下移动指针指向数组中的下一个字符。

This loop will fail if the array that `cp` addresses is not null-terminated. If this case, the loop is apt to read characters starting at `cp` until it encounters a null character somewhere in memory.

如果 `cp` 所指向的字符数组没有 `null` 结束符，则此循环将会失败。这时，循环会从 `cp` 指向的位置开始读数，直到遇到内存中某处 `null` 结束符为止。



C Library String Functions

C 风格字符串的标准库函数

The Standard C library provides a set of functions, listed in [Table 4.1](#), that operate on C-style strings. To use these functions, we must include the associated C header file

表4-1列出了 C 语言标准库提供的一系列处理 C 风格字符串的库函数。要使用这些标准库函数，必须包含相应的 C 头文件：

which is the C++ version of the `string.h` header from the C library.

`cstring` 是 `string.h` 头文件的 C++ 版本，而 `string.h` 则是 C 语言提供的标准库。



These functions do no checking on their string parameters.

这些标准库函数不会检查其字符串参数。

Table 4.1. C-Style Character String Functions

表 4.1. 操纵 C 风格字符串的标准库函数

<code>strlen(s)</code>	Returns the length of <code>s</code> , not counting the null. 返回 <code>s</code> 的长度，不包括字符串结束符 null
<code>strcmp(s1, s2)</code>	Compares <code>s1</code> and <code>s2</code> for equality. Returns 0 if <code>s1 == s2</code> , positive value if <code>s1 > s2</code> , negative value if <code>s1 < s2</code> . 比较两个字符串 <code>s1</code> 和 <code>s2</code> 是否相同。若 <code>s1</code> 与 <code>s2</code> 相等，返回 0；若 <code>s1</code> 大于 <code>s2</code> ，返回正数；若 <code>s1</code> 小于 <code>s2</code> ，则返回负数
<code>strcat(s1, s2)</code>	Appends <code>s2</code> to <code>s1</code> . Returns <code>s1</code> . 将字符串 <code>s2</code> 连接到 <code>s1</code> 后，并返回 <code>s1</code>
<code>strcpy(s1, s2)</code>	Copies <code>s2</code> into <code>s1</code> . Returns <code>s1</code> . 将 <code>s2</code> 复制给 <code>s1</code> ，并返回 <code>s1</code>
<code>strncat(s1, s2, n)</code>	Appends <code>n</code> characters from <code>s2</code> onto <code>s1</code> . Returns <code>s1</code> . 将 <code>s2</code> 的前 <code>n</code> 个字符连接到 <code>s1</code> 后面，并返回 <code>s1</code>
<code>strncpy(s1, s2, n)</code>	Copies <code>n</code> characters from <code>s2</code> into <code>s1</code> . Returns <code>s1</code> . 将 <code>s2</code> 的前 <code>n</code> 个字符复制给 <code>s1</code> ，并返回 <code>s1</code>

```
#include <cstring>
```

The pointer(s) passed to these routines must be nonzero and each pointer must point to the initial character in a null-terminated array. Some of these functions write to a string they are passed. These functions assume that the array to which they write is large enough to hold whatever characters the function generates. It is up to the programmer to ensure that the target string is big enough.

传递给这些标准库函数例程的指针必须具有非零值，并且指向以 `null` 结束的字符数组中的第一个元素。其中一些标准库函数会修改传递给它的字符串，这些函数将假定它们所修改的字符串具有足够大的空间接收本函数新生成的字符，程序员必须确保目标字符串必须足够大。

When we compare library `strings`, we do so using the normal relational operators. We can use these operators to compare pointers to C-style strings, but the effect is quite different; what we're actually comparing is the pointer values, not the strings to which they point:

C++ 语言提供普通的关系操作符实现标准库类型 `string` 的对象的比较。这些操作符也可用于比较指向C风格字符串的指针，但效果却很不相同：实际上，此时比较的是指针上存放的地址值，而并非它们所指向的字符串：

```
if (cp1 < cp2) // compares addresses, not the values pointed to
```

Assuming `cp1` and `cp2` point to elements in the same array (or one past that array), then the effect of this comparison is to compare the address in `cp1` with the address in `cp2`. If the pointers do not address the same array, then the comparison is undefined.

如果 `cp1` 和 `cp2` 指向同一数组中的元素（或该数组的溢出位置），上述表达式等效于比较在 `cp1` 和 `cp2` 中存放的地址；如果这两个指针指向不同的数组，则该表达式实现的比较没有定义。

To compare the strings, we must use `strcmp` and interpret the result:

字符串的比较和比较结果的解释都须使用标准库函数 `strcmp` 进行：

```
const char *cp1 = "A string example";
const char *cp2 = "A different string";
int i = strcmp(cp1, cp2);      // i is positive
i = strcmp(cp2, cp1);        // i is negative
i = strcmp(cp1, cp1);        // i is zero
```

The `strcmp` function returns three possible values: 0 if the strings are equal; or a positive or negative value, depending on whether the first string is larger or smaller than the second.

标准库函数 `strcmp` 有 3 种可能的返回值：若两个字符串相等，则返回 0 值；若第一个字符串大于第二个字符串，则返回正数，否则返回负数。

Never Forget About the Null-Terminator

永远不要忘记字符串结束符 `null`

When using the C library string functions it is essential to remember the strings must be null-terminated:

在使用处理 C 风格字符串的标准库函数时，牢记字符串必须以结束符 `null` 结束：

```
char ca[] = {'C', '+', '+'}; // not null-terminated
cout << strlen(ca) << endl; // disaster: ca isn't null-terminated
```

In this case, `ca` is an array of characters but is not null-terminated. What happens is undefined. The `strlen` function assumes that it can rely on finding a null character at the end of its argument. The most likely effect of this call is that `strlen` will keep looking through the memory that follows wherever `ca` happens to reside until it encounters a null character. In any event, the return from `strlen` will not be the correct value.

在这个例题中，`ca` 是一个没有 `null` 结束符的字符数组，则计算的结果不可预料。标准库函数 `strlen` 总是假定其参数字符串以 `null` 字符结束，当调用该标准库函数时，系统将会从实参 `ca` 指向的内存空间开始一直搜索结束符，直到恰好遇到 `null` 为止。`strlen` 返回这一段内存空间中总共有多少个字符，无论如何这个数值不可能是正确的。

Caller Is Responsible for Size of a Destination String

调用者必须确保目标字符串具有足够的大小

The array that we pass as the first argument to `strcat` and `strcpy` must be large enough to hold the generated string. The code we show here, although a common usage pattern, is fraught with the potential for serious error:

传递给标准库函数 `strcat` 和 `strcpy` 的第一个实参数组必须具有足够大的空间存放新生成的字符串。以下代码虽然演示了一种通常的用法，但是却有潜在的严重错误：

```
// Dangerous: What happens if we miscalculate the size of largeStr?
char largeStr[16 + 18 + 2];           // will hold cp1 a space and cp2
strcpy(largeStr, cp1);                // copies cp1 into largeStr
strcat(largeStr, " ");               // adds a space at end of largeStr
```

Section 4.3. C-Style Character Strings

```
strcat(largeStr, cp2);           // concatenates cp2 to largeStr
// prints A string example A different string
cout << largeStr << endl;
```

The problem is that we could easily miscalculate the size needed in `largeStr`. Similarly, if we later change the sizes of the strings to which either `cp1` or `cp2` point, then the calculated size of `largeStr` will be wrong. Unfortunately, programs similar to this code are widely distributed. Programs with such code are error-prone and often lead to serious security leaks.

问题在于我们经常会算错 `largeStr` 需要的大小。同样地，如果 `cp1` 或 `cp2` 所指向的字符串大小发生了变化，`largeStr` 所需要的大小则会计算错误。不幸的是，类似于上述代码的程序应用非常广泛，这类程序往往容易出错，并导致严重的安全漏洞。

When Using C-Style Strings, Use the `strn` Functions

使用 `strn` 函数处理C风格字符串

If you must use C-style strings, it is usually safer to use the `strncat` and `strncpy` functions instead of `strcat` and `strcpy`:

如果必须使用 C 风格字符串，则使用标准库函数 `strncat` 和 `strncpy` 比 `strcat` 和 `strcpy` 函数更安全：

```
char largeStr[16 + 18 + 2]; // to hold cp1 a space and cp2
strncpy(largeStr, cp1, 17); // size to copy includes the null
strncat(largeStr, " ", 2); // pedantic, but a good habit
strncat(largeStr, cp2, 19); // adds at most 18 characters, plus a null
```

The trick to using these versions is to properly calculate the value to control how many characters get copied. In particular, we must *always* remember to account for the null when copying or concatenating characters. We must allocate space for the null because that is the character that terminates `largeStr` after each call. Let's walk through these calls in detail:

使用标准库函数 `strncat` 和 `strncpy` 的诀窍在于可以适当地控制复制字符的个数。特别是在复制和串连字符串时，一定要时刻记住算上结束符 `null`。在定义字符串时要切记预留存放 `null` 字符的空间，因为每次调用标准库函数后都必须以此结束字符串 `largeStr`。让我们详细分析一下这些标准库函数的调用：

- On the call to `strncpy`, we ask to copy 17 characters: all the characters in `cp1` plus the null. Leaving room for the null is necessary so that `largeStr` is properly terminated. After the `strncpy` call, `largeStr` has a `strlen` value of 16. Remember, `strlen` counts the characters in a C-style string, not including the null.

调用 `strncpy` 时，要求复制 17 个字符：字符串 `cp1` 中所有字符，加上结束符 `null`。留下存储结束符 `null` 的空间是必要的，这样 `largeStr` 才可以正确地结束。调用 `strncpy` 后，字符串 `largeStr` 的长度 `strlen` 值是 16。记住：标准库函数 `strlen` 用于计算 C 风格字符串中的字符个数，不包括 `null` 结束符。

- When we call `strncat`, we ask to copy two characters: the space and the null that terminates the string literal. After this call, `largeStr` has a `strlen` of 17. The null that had ended `largeStr` is overwritten by the space that we appended. A new null is written after that space.

调用 `strncat` 时，要求复制 2 个字符：一个空格和结束该字符串字面值的 `null`。调用结束后，字符串 `largeStr` 的长度是 17，原来用于结束 `largeStr` 的 `null` 被新添加的空格覆盖了，然后在空格后面写入新的结束符 `null`。

- When we append `cp2` in the second call, we again ask to copy all the characters from `cp2`, *including* the null. After this call, the `strlen` of `largeStr` would be 35: 16 characters from `cp1`, 18 from `cp2`, and 1 for the space that separates the two strings.

第二次调用 `strncat` 串接 `cp2` 时，要求复制 `cp2` 中所有字符，包括字符串结束符 `null`。调用结束后，字符串 `largeStr` 的长度是 35：`cp1` 的 16 个字符和 `cp2` 的 18 个字符，再加上分隔这两个字符串的一个空格。

The array size of `largeStr` remains 36 throughout.

整个过程中，存储 `largeStr` 的数组大小始终保持为 36（包括结束符）。

These operations are safer than the simpler versions that do not take a size argument as long as we calculate the size argument correctly. If we ask to copy or concatenate more characters than the size of the target array, we will still overrun that array. If the string we're copying from or concatenating is bigger than the requested size, then we'll inadvertently truncate the new version. Truncating is safer than overrunning the array, but it is still an error.

只要可以正确计算出 `size` 实参的值，使用 `strn` 版本要比没有 `size` 参数的简化版本更安全。但是，如果要向目标数组复制或串接比其 `size` 更多的字符，数组溢出的现象仍然会发生。如果要复制或串接的字符串比实际要复制或串接的 `size` 大，我们会不经意地把新生成的字符串截短了。截短字符串比数组溢出要安全，但这仍是错误的。

Whenever Possible, Use Library `strings`

尽可能使用标准库类型 `string`

Section 4.3. C-Style Character Strings

None of these issues matter if we use C++ library `strings`:

如果使用 C++ 标准库类型 `string`, 则不存在上述问题:

```
string largeStr = cp1; // initialize largeStr as a copy of cp1
largeStr += " ";      // add space at end of largeStr
largeStr += cp2;       // concatenate cp2 onto end of largeStr
```

Now the library handles all memory management, and we need no longer worry if the size of either string changes.

此时, 标准库负责处理所有的内存管理问题, 我们不必再担心每一次修改字符串时涉及到的大小问题。



For most applications, in addition to being safer, it is also more efficient to use library `strings` rather than C-style strings.

对大部分的应用而言, 使用标准库类型 `string`, 除了增强安全性外, 效率也提高了, 因此应该尽量避免使用 C 风格字符串。

Exercises Section 4.3

Exercise Explain the difference between the following two `while` loops:

4.22:

解释下列两个 `while` 循环的差别:

```
const char *cp = "hello";
int cnt;
while (*cp) { ++cnt; ++cp; }
while (*cp) { ++cnt; ++cp; }
```

Exercise What does the following program do?

4.23:

下列程序实现什么功能?

```
const char ca[] = {'h', 'e', 'l', 'l', 'o'};
const char *cp = ca;
while (*cp) {
    cout << *cp << endl;
    ++cp;
}
```

Exercise Explain the differences between `strcpy` and `strncpy`. What are the advantages of each? The disadvantages?

4.24:

解释 `strcpy` 和 `strncpy` 的差别在哪里, 各自的优缺点是什么?

Exercise Write a program to compare two `strings`. Now write a program to compare the value of two C-style character strings.

4.25:

编写程序比较两个 `string` 类型的字符串, 然后编写另一个程序比较两个C风格字符串的值。

Exercise Write a program to read a `string` from the standard input. How might you write a program to read from the standard input into a C-style character string?

4.26:

编写程序从标准输入设备读入一个 `string` 类型的字符串。考虑如何编程实现从标准输入设备读入一个 C 风格字符串。

4.3.1. Dynamically Allocating Arrays

4.3.1. 创建动态数组

A variable of array type has three important limitations: Its size is fixed, the size must be known at compile time, and the array exists only until the end of the block in which it was defined. Real-world programs usually cannot live with these restrictions—they need a way to allocate an array

Section 4.3. C-Style Character Strings

dynamically at run time. Although all arrays have fixed size, the size of a dynamically allocated array need not be fixed at compile time. It can be (and usually is) determined at run time. Unlike an array variable, a dynamically allocated array continues to exist until it is explicitly freed by the program.

数组类型的变量有三个重要的限制：数组长度固定不变，在编译时必须知道其长度，数组只在定义它的块语句内存在。实际的程序往往不能忍受这样的限制——它们需要在运行时动态地分配数组。虽然数组长度是固定的，但动态分配的数组不必在编译时知道其长度，可以（通常也是）在运行时才确定数组长度。与数组变量不同，动态分配的数组将一直存在，直到程序显式释放它为止。

Every program has a pool of available memory it can use during program execution to hold dynamically allocated objects. This pool of available memory is referred to as the program's **free store** or **heap**. C programs use a pair of functions named `malloc` and `free` to allocate space from the free store. In C++ we use `new` and `delete` expressions.

每一个程序在执行时都占用一块可用的内存空间，用于存放动态分配的对象，此内存空间称为程序的**自由存储区或堆**。C 语言程序使用一对标准库函数 `malloc` 和 `free` 在自由存储区中分配存储空间，而 C++ 语言则使用 `new` 和 `delete` 表达式实现相同的功能。

Defining a Dynamic Array

动态数组的定义

When we define an array variable, we specify a type, a name, and a dimension. When we dynamically allocate an array, we specify the type and size but do not name the object. Instead, the `new` expression returns a pointer to the first element in the newly allocated array:

数组变量通过指定类型、数组名和维数来定义。而动态分配数组时，只需指定类型和数组长度，不必为数组对象命名，`new` 表达式返回指向新分配数组的第一个元素的指针：

```
int *pia = new int[10]; // array of 10 uninitialized ints
```

This `new` expression allocates an array of ten `ints` and returns a pointer to the first element in that array, which we use to initialize `pia`.

此 `new` 表达式分配了一个含有 10 个 `int` 型元素的数组，并返回指向该数组第一个元素的指针，此返回值初始化了指针 `pia`。

A `new` expression takes a type and optionally an array dimension specified inside a bracket-pair. The dimension can be an arbitrarily complex expression. When we allocate an array, `new` returns a pointer to the first element in the array. Objects allocated on the free store are unnamed. We use objects on the heap only indirectly through their address.

`new` 表达式需要指定指针类型以及在方括号中给出的数组维数，该维数可以是任意的复杂表达式。创建数组后，`new` 将返回指向数组第一个元素的指针。在自由存储区中创建的数组对象是没有名字的，程序员只能通过其地址间接地访问堆中的对象。

Initializing a Dynamically Allocated Array

初始化动态分配的数组

When we allocate an array of objects of a class type, then that type's default constructor ([Section 2.3.4](#), p. 50) is used to initialize each element. If the array holds elements of built-in type, then the elements are uninitialized:

动态分配数组时，如果数组元素具有类类型，将使用该类的默认构造函数（[第 2.3.4 节](#)）实现初始化；如果数组元素是内置类型，则无初始化：

```
string *psa = new string[10]; // array of 10 empty strings
int *pia = new int[10]; // array of 10 uninitialized ints
```

Each of these `new` expressions allocates an array of ten objects. In the first case, those objects are `strings`. After allocating memory to hold the objects, the default `string` constructor is run on each element of the array in turn. In the second case, the objects are a built-in type; memory to hold ten `ints` is allocated, but the elements are uninitialized.

这两个 `new` 表达式都分配了含有 10 个对象的数组。其中第一个数组是 `string` 类型，分配了保存对象的内存空间后，将调用 `string` 类型的默认构造函数依次初始化数组中的每个元素。第二个数组则具有内置类型的元素，分配了存储 10 个 `int` 对象的内存空间，但这些元素没有初始化。

Alternatively, we can value-initialize ([Section 3.3.1](#), p. 92) the elements by following the array size by an empty pair of parentheses:

也可使用跟在数组长度后面的一对空圆括号，对数组元素做值初始化（[第 3.3.1 节](#)）：

```
int *pia2 = new int[10](); // array of 10 uninitialized ints
```

The parentheses are effectively a request to the compiler to value-initialize the array, which in this case sets its elements to 0.

圆括号要求编译器对数组做值初始化，在本例中即把数组元素都设置为0。

The elements of a dynamically allocated array can be initialized only to the default value of the element type. The elements cannot be initialized to separate values as can be done for elements of an array variable.



对于动态分配的数组，其元素只能初始化为元素类型的默认值，而不能像数组变量一样，用初始化列表为数组元素提供各不相同的初值。

Dynamic Arrays of `const` Objects

`const` 对象的动态数组

If we create an array of `const` objects of built-in type on the free store, we must initialize that array: The elements are `const`, there is no way to assign values to the elements. The only way to initialize the elements is to value-initialize the array:

如果我们在自由存储区中创建的数组存储了内置类型的 `const` 对象，则必须为这个数组提供初始化：因为数组元素都是 `const` 对象，无法赋值。实现这个要求的唯一方法是对数组做值初始化：

```
// error: uninitialized const array
const int *pci_bad = new const int[100];
// ok: value-initialized const array
const int *pci_ok = new const int[100]();
```

It is possible to have a `const` array of elements of a class type that provides a default constructor:

C++ 允许定义类类型的 `const` 数组，但该类类型必须提供默认构造函数：

```
// ok: array of 100 empty strings
const string *pcs = new const string[100];
```

In this case, the default constructor is used to initialize the elements of the array.

在这里，将使用 `string` 类的默认构造函数初始化数组元素。

Of course, once the elements are created, they may not be changed which means that such arrays usually are not very useful.

当然，已创建的常量元素不允许修改——因此这样的数组实际上用处不大。

It Is Legal to Dynamically Allocate an Empty Array

允许动态分配空数组

When we dynamically allocate an array, we often do so because we don't know the size of the array at compile time. We might write code such as之所以要动态分配数组，往往是由编译时并不知道数组的长度。我们可以编写如下代码

```
size_t n = get_size(); // get_size returns number of elements needed
int* p = new int[n];
for (int* q = p; q != p + n; ++q)
    /* process the array */;
```

to figure out the size of the array and then allocate and process the array.

计算数组长度，然后创建和处理该数组。

An interesting question is: What happens if `get_size` returns 0? The answer is that our code works fine. The language specifies that a call to `new` to create an array of size zero is legal. It is legal even though we could not create an array variable of size 0:

有趣的是，如果 `get_size` 返回 0 则会怎么样？答案是：代码仍然正确执行。C++ 虽然不允许定义长度为 0 的数组变量，但明确指出，调用 `new` 动态创建长度为 0 的数组是合法的：

```
char arr[0];           // error: cannot define zero-length array
char *cp = new char[0]; // ok: but cp can't be dereferenced
```

When we use `new` to allocate an array of zero size, `new` returns a valid, nonzero pointer. This pointer will be distinct from any other pointer returned by `new`. The pointer cannot be dereferenced after all, it points to no element. The pointer can be compared and so can be used in a loop such as the preceding one. It is also legal to add (or subtract) zero to such a pointer and to subtract the pointer from itself, yielding zero.

用 `new` 动态创建长度为 0 的数组时，`new` 返回有效的非零指针。该指针与 `new` 返回的其他指针不同，不能进行解引用操作，因为它毕竟没有指向任何元素。而允许的操作包括：比较运算，因此该指针能在循环中使用；在该指针上加（减）0；或者减去本身，得 0 值。

In our hypothetical loop, if the call to `get_size` returned 0, then the call to `new` would still succeed. However, `p` would not address any element; the

Section 4.3. C-Style Character Strings

array is empty. Because `n` is zero, the `for` loop effectively compares `q` to `p`. These pointers are equal; `q` was initialized to `p`, so the condition in the `for` fails and the loop body is not executed.

在上述例题中，如果 `get_size` 返回 0，则仍然可以成功调用 `new`，但是 `p` 并没有指向任何对象，数组是空的。因为 `n` 为 0，所以 `for` 循环实际比较的是 `p` 和 `q`，而 `q` 是用 `p` 初始化的，两者具有相等的值，因此 `for` 循环条件不成立，循环体一次都没有执行。

Freeing Dynamic Memory

动态空间的释放

When we allocate memory, we must eventually free it. Otherwise, memory is gradually used up and may be exhausted. When we no longer need the array, we must *explicitly* return its memory to the free store. We do so by applying the `delete []` expression to a pointer that addresses the array we want to release:

```
delete [] pia;
```

deallocates the array pointed to by `pia`, returning the associated memory to the free store. The empty bracket pair between the `delete` keyword and the pointer is necessary: It indicates to the compiler that the pointer addresses an array of elements on the free store and not simply a single object.

该语句回收了 `pia` 所指向的数组，把相应的内存返还给自由存储区。在关键字 `delete` 和指针之间的空方括号对是必不可少的：它告诉编译器该指针指向的是自由存储区中的数组，而并非单个对象。



If the empty bracket pair is omitted, it is an error, but an error that the compiler is unlikely to catch; the program may fail at run time.

如果遗漏了空方括号对，这是一个编译器无法发现的错误，将导致程序在运行时出错。

The least serious run-time consequence of omitting brackets when freeing an array is that too little memory will be freed, leading to a memory leak. On some systems and/or for some element types, more serious run-time problems are possible. It is essential to remember the bracket-pair when deleting pointers to arrays.

理论上，回收数组时缺少空方括号对，至少会导致运行时少释放了内存空间，从而产生内存泄漏 (memory leak)。对于某些系统和/或元素类型，有可能会带来更严重的运行时错误。因此，在释放动态数组时千万别忘了方括号对。

Contrasting C-Style Strings and C++ Library `string`s

C 风格字符串与 C++ 的标准库类型 `string` 的比较

The following two programs illustrate the differences in using C-style character strings versus using the C++ library `string` type. The `string` version is shorter, easier to understand, and less error-prone:

以下两段程序反映了使用 C 风格字符串与 C++ 的标准库类型 `string` 的不同之处。使用 `string` 类型的版本更短、更容易理解，而且出错的可能性更小：

[\[View full width\]](#)

```
// C-style character string implementation
const char *pc = "a very long literal string";
const size_t len = strlen(pc +1);           // space to
→   allocate
// performance test on string allocation and copy
for (size_t ix = 0; ix != 1000000; ++ix) {
    char *pc2 = new char[len + 1]; // allocate the space
    strcpy(pc2, pc);             // do the copy
    if (strcmp(pc2, pc))         // use the new string
```

Section 4.3. C-Style Character Strings

```
    ; // do nothing
    delete [] pc2;           // free the memory
}
// string implementation
string str("a very long literal string");
// performance test on string allocation and copy
for (int ix = 0; ix != 1000000; ++ix) {
    string str2 = str; // do the copy, automatically
→ allocated
    if (str != str2)      // use the new string
        ; // do nothing
}
                                         // str2 is
→ automatically freed
```

These programs are further explored in the exercises to [Section 4.3.1 \(p. 139\)](#).
这些程序将在4.3.1节的习题中做进一步探讨。

Using Dynamically Allocated Arrays

动态数组的使用

A common reason to allocate an array dynamically is if its dimension cannot be known at compile time. For example, `char*` pointers are often used to refer to multiple C-style strings during the execution of a program. The memory used to hold the various strings typically is allocated dynamically during program execution based on the length of the string to be stored. This technique is considerably safer than allocating a fixed-size array. Assuming we correctly calculate the size needed at run time, we no longer need to worry that a given string will overflow the fixed size of an array variable.

通常是因为在编译时无法知道数组的维数，所以才需要动态创建该数组。例如，在程序执行过程中，常常使用`char*`指针指向多个C风格字符串，于是必须根据每个字符串的长度实时地动态分配存储空间。采用这种技术要比建立固定大小的数组安全。如果程序员能够准确计算出运行时需要的数组长度，就不必再担心因数组变量具有固定的长度而造成的溢出问题。

Suppose we have the following C-style strings:

假设有以下C风格字符串：

```
const char *noerr = "success";
// ...
const char *err189 = "Error: a function declaration must "
    "specify a function return type!";
```

We might want to copy one or the other of these strings at run time to a new character array. We could calculate the dimension at run time, as follows:

我们想在运行时把这两个字符串中的一个复制给新的字符数组，于是可以用以下程序在运行时计算维数：

```
const char *errorTxt;
if (errorFound)
    errorTxt = err189;
else
    errorTxt = noerr;
// remember the 1 for the terminating null
int dimension = strlen(errorTxt) + 1;
char *errMsg = new char[dimension];
// copy the text for the error into errMsg
strncpy (errMsg, errorTxt, dimension);
```

Recall that `strlen` returns the length of the string *not* including the null. It is essential to remember to add 1 to the length returned from `strlen` to accommodate the trailing null.

别忘记标准库函数 `strlen` 返回的是字符串的长度，并不包括字符串结束符，在获得的字符串长度上必须加 1 以便在动态分配时预留结束符的存储空间。

Exercises Section 4.3.1

Exercise Given the following `new` expression, how would you `delete` `pa`?

4.27: 假设有下面的 `new` 表达式，请问如何释放 `pa`？

```
int *pa = new int[10];
```

Exercise Write a program to read the standard input and build a `vector` of `ints` from values that are read. Allocate an array of the same size as the `vector` and copy the elements from the `vector`

Section 4.3. C-Style Character Strings

into the array.

编写程序由从标准输入设备读入的元素数据建立一个 `int` 型 `vector` 对象，然后动态创建一个与该 `vector` 对象大小一致的数组，把 `vector` 对象的所有元素复制给新数组。

Exercise Given the two program fragments in the highlighted box on page [138](#).

4.29:

对本小节第 5 条框中的两段程序：

- Explain what the programs do.

解释这两个程序实现什么功能？

- As it happens, on average, the `string` class implementation executes considerably faster than the C-style string functions. The relative average execution times on our more than five-year-old PC are as follows:

平均来说，使用 `string` 类型的程序执行速度要比用 C 风格字符串的快很多，在我们用了五年的 PC 机上其平均执行速度分别是：

user	0.47	# string class
user	2.55	# C-style character string

Did you expect that? How would you account for it?

你预计的也一样吗？请说明原因。

Exercise Write a program to concatenate two C-style string literals, putting the result in a C-style string.

4.30:

Write a program to concatenate two library `strings` that have the same value as the literals used in the first program.

编写程序连接两个C风格字符串字面值，把结果存储在一个C风格字符串中。然后再编写程序连接两个 `string` 类型字符串，这两个 `string` 类型字符串与前面的C风格字符串字面值具有相同的内容。

4.3.2. Interfacing to Older Code

4.3.2. 新旧代码的兼容

Many C++ programs exist that predate the standard library and so do not yet use the `string` and `vector` types. Moreover, many C++ programs interface to existing C programs that cannot use the C++ library. Hence, it is not infrequent to encounter situations where a program written in modern C++ must interface to code that uses arrays and/or C-style character strings. The library offers facilities to make the interface easier to manage.

许多 C++ 程序在有标准类之前就已经存在了，因此既没有使用标准库类型 `string` 也没有使用 `vector`。而且，许多 C++ 程序为了兼容现存的 C 程序，也不能使用 C++ 标准库。因此，现代的 C++ 程序经常必须兼容使用数组和／或 C 风格字符串的代码，标准库提供了使兼容界面更容易管理的手段。

Mixing Library `string`s and C-Style Strings

混合使用标准库类 `string` 和 C 风格字符串

As we saw on page [80](#) we can initialize a `string` from a string literal:

正如第 3.2.1 节中显示的，可用字符串字面值初始化 `string` 类对象：

```
string st3("Hello World"); // st3 holds Hello World
```

More generally, because a C-style string has the same type as a string literal and is null-terminated in the same way, we can use a C-style string anywhere that a string literal can be used:

通常，由于 C 风格字符串与字符串字面值具有相同的数据类型，而且都是以空字符 `null` 结束，因此可以把 C 风格字符串用在任何可以使用字符串字面值的地方：

- We can initialize or assign to a `string` from a C-style string.

可以使用 C 风格字符串对 `string` 对象进行初始化或赋值。

Section 4.3. C-Style Character Strings

- We can use a C-style string as one of the two operands to the `string` addition or as the right-hand operand to the compound assignment operators.

`string` 类型的加法操作需要两个操作数，可以使用 C 风格字符串作为其中的一个操作数，也允许将 C 风格字符串用作复合赋值操作的右操作数。

The reverse functionality is not provided: there is no direct way to use a library `string` when a C-style string is required. For example, there is no way to initialize a character pointer from a `string`:

反之则不成立：在要求C风格字符串的地方不可直接使用标准库 `string` 类型对象。例如，无法使用 `string` 对象初始化字符指针：

```
char *str = st2; // compile-time type error
```

There is, however, a `string` member function named `c_str` that we can often use to accomplish what we want:

但是，`string` 类提供了一个名为 `c_str` 的成员函数，以实现我们的要求：

```
char *str = st2.c_str(); // almost ok, but not quite
```

The name `c_str` indicates that the function returns a C-style character string. Literally, it says, "Get me the C-style string representation" that is, a pointer to the beginning of a null-terminated character array that holds the same data as the characters in the `string`.

`c_str` 函数返回 C 风格字符串，其字面意思是：“返回 C 风格字符串的表示方法”，即返回指向字符数组首地址的指针，该数组存放了与 `string` 对象相同的内容，并且以结束符 `null` 结束。

This initialization fails because `c_str` returns a pointer to an array of `const char`. It does so to prevent changes to the array. The correct initialization is:

如果 `c_str` 返回的指针指向 `const char` 类型的数组，则上述初始化失败，这样做是为了避免修改该数组。正确的初始化应为：

```
const char *str = st2.c_str(); // ok
```



The array returned by `c_str` is not guaranteed to be valid indefinitely. Any subsequent use of `st2` that might change the value of `st2` can invalidate the array. If a program needs continuing access to the data, then the program must copy the array returned by `c_str`.

`c_str` 返回的数组并不保证一定是有效的，接下来对 `st2` 的操作有可能会改变 `st2` 的值，使刚才返回的数组失效。如果程序需要持续访问该数据，则应该复制 `c_str` 函数返回的数组。

Using an Array to Initialize a `vector`

使用数组初始化 `vector` 对象

On page 112 we noted that it is not possible to initialize an array from another array. Instead, we have to create the array and then explicitly copy the elements from one array into the other. It turns out that we can use an array to initialize a `vector`, although the form of the initialization may seem strange at first. To initialize a `vector` from an array, we specify the address of the first element and one past the last element that we wish to use as initializers:

第 4.1.1 节提到不能用一个数组直接初始化另一数组，程序员只能创建新数组，然后显式地把源数组的元素逐个复制给新数组。这反映 C++ 允许使用数组初始化 `vector` 对象，尽管这种初始化形式起初看起来有点陌生。使用数组初始化 `vector` 对象，必须指出用于初始化式的第一个元素以及数组最后一个元素的下一位置的地址：

```
const size_t arr_size = 6;
int int_arr[arr_size] = {0, 1, 2, 3, 4, 5};
// ivec has 6 elements: each a copy of the corresponding element in int_arr
vector<int> ivec(int_arr, int_arr + arr_size);
```

The two pointers passed to `ivec` mark the range of values with which to initialize the `vector`. The second pointer points one past the last element to be copied. The range of elements marked can also represent a subset of the array:

传递给 `ivec` 的两个指针标出了 `vector` 初值的范围。第二个指针指向被复制的最后一个元素之后的地址空间。被标出的元素范围可以是数组的子集：

```
// copies 3 elements: int_arr[1], int_arr[2], int_arr[3]
vector<int> ivec(int_arr + 1, int_arr + 4);
```

This initialization creates `ivec` with three elements. The values of these elements are copies of the values in `int_arr[1]` through `int_arr[3]`.

这个初始化创建了含有三个元素的 `ivec`，三个元素的值分别是 `int_arr[1]` 到 `int_arr[3]` 的副本。

Exercises Section 4.3.2

Exercise Write a program that reads a string into a character array from the standard input. Describe

- 4.31:** how your program handles varying size inputs. Test your program by giving it a string of data that is longer than the array size you've allocated.

编写程序从标准输入设备读入字符串，并把该串存放在字符数组中。描述你的程序如何处理可变长的输入。提供比你分配的数组长度长的字符串数据测试你的程序。

- Exercise** Write a program to initialize a `vector` from an array of `ints`.

- 4.32:** 编写程序用 `int` 型数组初始化 `vector` 对象。

- Exercise** Write a program to copy a `vector` of `ints` into an array of `ints`.

- 4.33:** 编写程序把 `int` 型 `vector` 复制给 `int` 型数组。

- Exercise** Write a program to read `strings` into a `vector`. Now, copy that `vector` into an array of character pointers.

- 4.34:** For each element in the `vector`, allocate a new character array and copy the data from the `vector` element into that character array. Then insert a pointer to the character array into the array of character pointers.

编写程序读入一组 `string` 类型的数据，并将它们存储在 `vector` 中。接着，把该 `vector` 对象复制给一个字符指针数组。为 `vector` 中的每个元素创建一个新的字符数组，并把该 `vector` 元素的数据复制到相应的字符数组中，最后把指向该数组的指针插入字符指针数组。

- Exercise** Print the contents of the `vector` and the array created in the previous exercise. After printing the array, remember to delete the character arrays.

输出习题 4.34 中建立的 `vector` 对象和数组的内容。输出数组后，记得释放字符数组。

4.4. Multidimensioned Arrays

4.4. 多维数组



Strictly speaking, there are no multidimensioned arrays in C++. What is commonly referred to as a multidimensioned array is actually an array of arrays:

严格地说，C++ 中没有多维数组，通常所指的多维数组其实就是数组的数组：

```
// array of size 3, each element is an array of ints of size 4
int ia[3][4];
```

It can be helpful to keep this fact in mind when using what appears to be a multidimensioned array.

在使用多维数组时，记住这一点有利于理解其应用。

An array whose elements are an array is said to have two dimensions. Each dimension is referred to by its own subscript:

如果数组的元素又是数组，则称为二维数组，其每一维对应一个下标：

```
ia[2][3] // fetches last element from the array in the last row
```

The first dimension is often referred to as the row and the second as the column. In C++ there is no limit on how many subscripts are used. That is, we could have an array whose elements are arrays of elements that are arrays, and so on.

第一维通常称为行 (row)，第二维则称为列 (column)。C++ 中并未限制可用的下标个数，也就是说，我们可以定义元素是数组（其元素又是数组，如此类推）的数组。

Initializing the Elements of a Multidimensioned Array

多维数组的初始化

As with any array, we can initialize the elements by providing a bracketed list of initializers. Multidimensioned arrays may be initialized by specifying bracketed values for each row:

和处理一维数组一样，程序员可以使用由花括号括起来的初始化式列表来初始化多维数组的元素。对于多维数组的每一行，可以再用花括号指定其元素的初始化式：

```
int ia[3][4] = { /* 3 elements, each element is an array of size 4 */
    {0, 1, 2, 3}, /* initializers for row indexed by 0 */
    {4, 5, 6, 7}, /* initializers for row indexed by 1 */
    {8, 9, 10, 11} /* initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent, although considerably less clear.

其中用来标志每一行的内嵌的花括号是可选的。下面的初始化尽管有点不清楚，但与前面的声明完全等价：

```
// equivalent initialization without the optional nested braces for each row
int ia[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

As is the case for single-dimension arrays, elements may be left out of the initializer list. We could initialize only the first element of each row as follows:

与一维数组一样，有些元素将不使用初始化列表提供的初始化式进行初始化。下面的声明只初始化了每行的第一个元素：

```
// explicitly initialize only element 0 in each row
int ia[3][4] = {{0}, {4}, {8}};
```

The values of the remaining elements depend on the element type and follow the rules described on page [112](#).

Section 4.4. Multidimensioned Arrays

其余元素根据其元素类型用第 4.1.1 节描述的规则初始化。

If the nested braces were omitted, the results would be very different:

如果省略内嵌的花括号，结果会完全不同：

```
// explicitly initialize row 0
int ia[3][4] = {0, 3, 6, 9};
```

initializes the elements of the first row. The remaining elements are initialized to 0.

该声明初始化了第一行的元素，其余元素都被初始化为 0。

Subscripting a Multidimensioned Array

多维数组的下标引用

Indexing a multidimensioned array requires a subscript for each dimension. As an example, the following pair of nested `for` loops initializes a two-dimensional array:

为了对多维数组进行索引，每一维都需要一个下标。例如，下面的嵌套 `for` 循环初始化了一个二维数组：

```
const size_t rowSize = 3;
const size_t colSize = 4;
int ia [rowSize][colSize]; // 12 uninitialized elements
// for each row
for (size_t i = 0; i != rowSize; ++i)
    // for each column within the row
    for (size_t j = 0; j != colSize; ++j)
        // initialize to its positional index
        ia[i][j] = i * colSize + j;
```

When we want to access a particular element of the array, we must supply both a row and column index. The row index specifies which of the inner arrays we intend to access. The column index selects an element from that inner array. Remembering this fact can help in calculating proper subscript values and in understanding how multidimensioned arrays are initialized.

当需要访问数组中的特定元素时，必须提供其行下标和列下标。行下标指出需要哪个内部数组，列下标则选取该内部数组的指定元素。了解多维数组下标引用策略有助于正确计算其下标值，以及理解多维数组如何初始化。

If an expression provides only a single index, then the result is the inner-array element at that row index. Thus, `ia[2]` fetches the array that is the last row in `ia`. It does not fetch any element from that array; it fetches the array itself.

如果表达式只提供了一个下标，则结果获取的元素是该行下标索引的内层数组。如 `ia[2]` 将获得 `ia` 数组的最后一行，即这一行的内层数组本身，而非该数组中的任何元素。

4.4.1. Pointers and Multidimensioned Arrays

4.4.1. 指针和多维数组

As with any array, when we use the name of a multidimensioned array, it is automatically converted to a pointer to the first element in the array.

与普通数组一样，使用多维数组名时，实际上将其自动转换为指向该数组第一个元素的指针。



When defining a pointer to a multidimensioned array, it is essential to remember that what we refer to as a multidimensioned array is really an array of arrays.

定义指向多维数组的指针时，千万别忘了该指针所指向的多维数组其实是数组的数组。

Because a multidimensioned array is really an array of arrays, the pointer type to which the array converts is a pointer to the first inner array. Although conceptually straightforward, the syntax for declaring such a pointer can be confusing:

因为多维数组其实就是数组的数组，所以由多维数组转换而成的指针类型应是指向第一个内层数组的指针。尽管这个概念非常明了，但声明这种指针的语法还是不容易理解：

```
int ia[3][4]; // array of size 3, each element is an array of ints of size 4
int (*ip)[4] = ia; // ip points to an array of 4 ints
ip = &ia[2]; // ia[2] is an array of 4 ints
```

We define a pointer to an array similarly to how we would define the array itself: We start by declaring the element type followed by a name and

Section 4.4. Multidimensioned Arrays

a dimension. The trick is that the name is a pointer, so we must prepend `*` to the name. We can read the definition of `ip` from the inside out as saying that `*ip` has type `int[4]` that is, `ip` is a pointer to an `int` array of four elements.

定义指向数组的指针与如何定义数组本身类似：首先声明元素类型，后接（数组）变量名字和维数。窍门在于（数组）变量的名字其实是指针，因此需在标识符前加上`*`。如果从内向外阅读 `ip` 的声明，则可理解为：`*ip` 是 `int[4]` 类型——即 `ip` 是一个指向含有 4 个元素的数组的指针。



The parentheses in this declaration are essential:

在下面的声明中，圆括号是必不可少的：

```
int *ip[4]; // array of pointers to int
int (*ip)[4]; // pointer to an array of 4 ints
```

TypeDefs Simplify Pointers to Multidimensioned Arrays

用 `typedef` 简化指向多维数组的指针

TypeDefs ([Section 2.6](#), p. 61) can help make pointers to elements in multidimensioned arrays easier to write, read, and understand. We might write a `typedef` for the element type of `ia` as

`typedef` 类型定义 ([第 2.6 节](#)) 可使指向多维数组元素的指针更容易读、写和理解。以下程序用 `typedef` 为 `ia` 的元素类型定义新的类型名：

```
typedef int int_array[4];
int_array *ip = ia;
```

We might use this `typedef` to print the elements of `ia`:

可使用 `typedef` 类型输出 `ia` 的元素：

```
for (int_array *p = ia; p != ia + 3; ++p)
    for (int *q = *p; q != *p + 4; ++q)
        cout << *q << endl;
```

The outer `for` loop starts by initializing `p` to point to the first array in `ia`. That loop continues until we've processed all three rows in `ia`. The increment, `++p`, has the effect of moving `p` to point to the next row (e.g., the next element) in `ia`.

外层的 `for` 循环首先初始化 `p` 指向 `ia` 的第一个内部数组，然后一直循环到 `ia` 的三行数据都处理完为止。`++p` 使 `p` 加 1，等效于移动指针使其指向 `ia` 的下一行（例如：下一个元素）。

The inner `for` loop actually fetches the `int` values stored in the inner arrays. It starts by making `q` point to the first element in the array to which `p` points. When we dereference `p`, we get an array of four `ints`. As usual, when we use an array, it is converted automatically to a pointer to its first element. In this case, that first element is an `int`, and we point `q` at that `int`. The inner `for` loop runs until we've processed every element in the inner array. To obtain a pointer just off the end of the inner array, we again dereference `p` to get a pointer to the first element in that array. We then add 4 to that pointer to process the four elements in each inner array.

内层的 `for` 循环实际上处理的是存储在内部数组中的 `int` 型元素值。首先让 `q` 指向 `p` 所指向的数组的第一个元素。对 `p` 进行解引用获得一个有 4 个 `int` 型元素的数组，通常，使用这个数组时，系统会自动将它转换为指向该数组第一个元素的指针。在本例中，第一个元素是 `int` 型数据，`q` 指向这个整数。系统执行内层的 `for` 循环直到处理完当前 `p` 指向的内部数组中所有的元素为止。当 `q` 指针刚达到该内部数组的超出末端位置时，再次对 `p` 进行解引用以获得指向下一个内部数组第一个元素的指针。在 `p` 指向的地址上加 4 使得系统可循环处理每一个内部数组的 4 个元素。

Exercises Section 4.4.1

Exercise 4.36: Rewrite the program to print the contents of the array `ia` without using a `typedef` for the type of the pointer in the outer loop.

重写程序输出 `ia` 数组的内容，要求在外层循环中不能使用 `typedef` 定义的类型。

Chapter Summary

小结

This chapter covered arrays and pointers. These facilities provide functionality similar to that provided by the library `vector` and `string` types and their companion iterators. The `vector` type can be thought of as a more flexible, easier to manage array. Similarly, `strings` are a great improvement on C-style strings that are implemented as null-terminated character arrays.

本章介绍了数组和指针。数组和指针所提供的功能类似于标准库的 `vector` 类与 `string` 类和相关的迭代器所提供。我们可以把 `vector` 类型理解为更灵活、更容易管理的数据，同样，`string` 是 C 风格字符串的改进类型，而 C 风格字符串是以空字符结束的字符数组。

Iterators and pointers allow indirect access to objects. Iterators are used to examine elements and navigate between the elements in `vectors`. Pointers provide similar access to array elements. Although conceptually simple, pointers are notoriously hard to use in practice.

迭代器和指针都能用于间接地访问所指向的对象。`vector` 类型所包含的元素通过迭代器来操纵，类似地，指针则用于访问数组元素。尽管道理都很简单，但在实际应用中，指针的难用是出了名的。

Pointers and arrays can be necessary for certain low-level tasks, but they should be avoided because they are error-prone and hard to debug. In general, the library abstractions should be used in preference to low-level array and pointer alternatives built into the language. This advice is especially applicable to using `strings` instead of C-style null-terminated character arrays. Modern C++ programs should not use C-style strings.

某些低级任务必须使用指针和数组，但由于使用指针和数组容易出错而且难以调试，应尽量避免使用。一般而言，应该优先使用标准库抽象类而少用语言内置的低级数组和指针。尤其是应该使用 `string` 类型取代 C 风格以空字符结束的字符数组。现代 C++ 程序不应使用 C 风格字符串。

Defined Terms

术语

C-style strings (C 风格字符串)

C programs treat pointers to null-terminated character arrays as strings. In C++, string literals are C-style strings. The C library defines a set of functions that operate on such strings, which C++ makes available in the `cstring` header. C++ programs should use C++ library strings in preference to C-style strings, which are inherently error-prone. A sizeable majority of security holes in networked programs are due to bugs related to using C-style strings and arrays.

C 程序把指向以空字符结束的字符数组的指针视为字符串。在 C++ 中，字符串字面值就是 C 风格字符串。C 标准库定义了一系列处理这种字符串的库函数，C++ 中将这些标准库函数放在 `cstring` 头文件中。由于 C 风格字符串本质上容易出错，C++ 程序应该优先使用 C++ 标准库类 `string` 而少用 C 风格字符串。网络程序中大量的安全漏洞都源于与使用 C 风格字符串和数组相关的缺陷。

compiler extension (编译器扩展)

Feature that is added to the language by a particular compiler. Programs that rely on compiler extensions cannot be moved easily to other compilers.

特定编译器为语言添加的特性。依赖于编译器扩展的程序很难移植到其他的编译器。

compound type (复合类型)

Type that is defined in terms of another type. Arrays, pointers, and references are compound types.

使用其他类型定义的类型。数组、指针和引用都是复合类型。

const void*

A pointer type that can point to any `const` type. See `void*`.

可以指向任意 `const` 类型的指针类型，参见 `void *`。

delete expression (delete 表达式)

A `delete` expression frees memory that was allocated by `new`:

`delete` 表达式用于释放由 `new` 动态分配的内存：

```
delete [] p;
```

where `p` must be a pointer to the first element in a dynamically allocated array. The bracket pair is essential: It indicates to the compiler that the pointer points at an array, not at a single object. In C++ programs, `delete` replaces the use of the C library `free` function.

在此表达式中，`p` 必须是指向动态创建的数组中第一个元素的指针，其中方括号必不可少：它告诉编译器该指针指向数组，而非单个对象。C++ 程序使用 `delete` 取代 C 语言的标准库函数 `free`。

dimension (维数)

The size of an array.

数组大小。

dynamically allocated (动态分配的)

An object that is allocated on the program's free store. Objects allocated on the free store exist until they are explicitly deleted.

在程序自由存储区中建立的对象。该对象一经创建就一直存在，直到显式释放为止。

free store (自由存储区)

Memory pool available to a program to hold dynamically allocated objects.

程序用来存储动态创建对象的内存区域。

[heap \(堆\)](#)

Synonym for free store.

自由存储区的同义词。

[new expression \(new表达式\)](#)

Allocates dynamic memory. We allocate an array of `n` elements as follows:

用于分配动态内存的表达式。下面的语句分配了一个有 `n` 个元素的数组：

```
new type[n];
```

The array holds elements of the indicated `type`. `new` returns a pointer to the first element in the array. C++ programs use `new` in place of the C library `malloc` function.

该数组存放 `type` 类型的元素。`new` 返回指向该数组第一个元素的指针。C++ 程序使用 `new` 取代 C 语言的标准库函数 `malloc`。

[pointer \(指针\)](#)

An object that holds the address of an object.

存放对象地址的对象。

[pointer arithmetic \(指针算术操作\)](#)

The arithmetic operations that can be applied to pointers. An integral type can be added to or subtracted from a pointer, resulting in a pointer positioned that many elements ahead or behind the original pointer. Two pointers can be subtracted, yielding the difference between the pointers. Pointer arithmetic is valid only on pointers that denote elements in the same array or an element one past the end of that array.

可用于指针的算术操作。允许在指针上做加上或减去整型值的操作，以获得当前指针之前或之后若干个元素处的地址。两个指针可做减法操作，得到它们之间的差值。只有当指针指向同一个数组或其超出末端的位置时，指针的算术操作才有意义。

[precedence \(优先级\)](#)

Defines the order in which operands are grouped with operators in a compound expression.

在复杂的表达式中，优先级确定了操作数分组的次序。

[ptrdiff_t](#)

Machine-dependent signed integral type defined in `cstdint` header that is large enough to hold the difference between two pointers into the largest possible array.

在 `cstdint` 头文件中定义的与机器相关的有符号整型，该类型具有足够的大小存储两个指针的差值，这两个指针指向同一个可能的最大数组。

[size_t](#)

Machine-dependent unsigned integral type defined in `cstdint` header that is large enough to hold the size of the largest possible array.

在 `cstdint` 头文件中定义的与机器相关的无符号整型，它具有足够的大小存储一个可能的最大数组。

[* operator \(* 操作符\)](#)

Dereferencing a pointer yields the object to which the pointer points. The dereference operator returns an lvalue; we may assign to the value returned from the dereference operator, which has the effect of assigning a new value to the underlying element.

对指针进行解引用操作获得该指针所指向的对象。解引用操作符返回左值，因此可为其结果赋值，等效于为该指针所指向的特定对象赋新值。

[++ operator \(++ 操作符\)](#)

When used with a pointer, the increment operator "adds one" by moving the pointer to refer to the next element in an array.

用于指针时，自增操作符给指针“加 1”，移动指针使其指向数组的下一个元素。

[\[\] operator \(\[\] 操作符\)](#)

The subscript operator takes two operands: a pointer to an element of an array and an index. Its result is the element that is offset from the pointer by the index. Indices count from zero the first element in an array is element 0, and the last is element size of the array minus

1. The subscript operator returns an lvalue; we may use a subscript as the left-hand operand of an assignment, which has the effect of assigning a new value to the indexed element.

下标操作符接受两个操作数：一个是指向数组元素的指针，一个是下标 `n`。该操作返回偏离指针当前指向 `n` 个位置的元素值。数组下标从 0 开始计数——数组第一个元素的下标为 0，最后一个元素的下标是数组长度减 1。下标操作返回左值，可用做赋值操作的左操作数，等效于为该下标引用的元素赋新值。

& operator (& 操作符)

The address-of operator. Takes a single argument that must be an lvalue. Yields the address in memory of that object.

取地址操作符需要一个操作数，其唯一的操作数必须是左值对象，该操作返回操作数对象在内存中的存储地址。

void*

A pointer type that can point to any non`const` type. Only limited operations are permitted on `void*` pointers. They can be passed or returned from functions and they can be compared with other pointers. They may not be dereferenced.

可以指向任何非 `const` 对象的指针类型。`void*` 指针只提供有限的几种操作：可用作函数形参类型或返回类型，也可与其他指针做比较操作，但是不能进行解引用操作。

Chapter 5. Expressions

第五章 表达式

CONTENTS

Section 5.1 Arithmetic Operators	149
Section 5.2 Relational and Logical Operators	152
Section 5.3 The Bitwise Operators	154
Section 5.4 Assignment Operators	159
Section 5.5 Increment and Decrement Operators	162
Section 5.6 The Arrow Operator	164
Section 5.7 The Conditional Operator	165
Section 5.8 The <code>sizeof</code> Operator	167
Section 5.9 Comma Operator	168
Section 5.10 Evaluating Compound Expressions	168
Section 5.11 The <code>new</code> and <code>delete</code> Expressions	174
Section 5.12 Type Conversions	178
Chapter Summary	188
Defined Terms	188

C++ provides a rich set of operators and defines what these operators do when applied to operands of built-in type. It also allows us to define meanings for the operators when applied to class types. This facility, known as [operator overloading](#), is used by the library to define the operators that apply to the library types.

C++ 提供了丰富的操作符，并定义操作数为内置类型时，这些操作符的含义。除此之外，C++ 还支持[操作符重载](#)，允许程序员自定义用于类类型时操作符的含义。标准库正是使用这种功能定义用于库类型的操作符。

In this chapter our focus is on the operators as defined in the language and applied to operands of built-in type. We will also look at some of the operators defined by the library. [Chapter 14](#) shows how we can define our own overloaded operators.

本章重点介绍 C++ 语言定义的操作符，它们使用内置类型的操作数；本章还会介绍一些标准库定义的操作符。[第十四章](#)将学习如何定义自己的重载操作符。

An expression is composed of one or more [operands](#) that are combined by [operators](#). The simplest form of an [expression](#) consists of a single literal constant or variable. More complicated expressions are formed from an operator and one or more operands.

表达式由一个或多个[操作数](#)通过[操作符](#)组合而成。最简单的表达式仅包含一个字面值常量或变量。较复杂的表达式则由操作符以及一个或多个操作数构成。

Every expression yields a [result](#). In the case of an expression with no operator, the result is the operand itself, e.g., a literal constant or a variable. When an object is used in a context that requires a value, then the object is evaluated by fetching the object's value. For example, assuming `ival` is an `int` object,

每个表达式都会产生一个[结果](#)。如果表达式中没有操作符，则其结果就是操作数本身（例如，字面值常量或变量）的值。当一个对象用在需要使用其值的地方，则计算该对象的值。例如，假设`ival`是一个`int`型对象：

```
if (ival)           // evaluate ival as a condition
// ...
```

we could use `ival` as an expression in the condition of an `if`. The condition succeeds if the value of `ival` is not zero and fails otherwise.

上述语句将 `ival` 作为 `if` 语句的条件表达式。当 `ival` 为非零值时，`if` 条件成立；否则条件不成立。

The result of expressions that involve operators is determined by applying each operator to its operand(s). Except when noted otherwise, the result

Chapter 5. Expressions

of an expression is an **rvalue** ([Section 2.3.1](#), p. 45). We can read the result but cannot assign to it.

对于含有操作符的表达式，它的值通过对操作数做指定操作获得。除了特殊用法外，表达式的结果是右值（[第 2.3.1 节](#)），可以读取该结果值，但是不允许对它进行赋值。



The meaning of an operator what operation is performed and the type of the result depends on the types of its operands.

操作符的含义——该操作符执行什么操作以及操作结果的类型——取决于操作数的类型。

Until one knows the type of the operand(s), it is not possible to know what a particular expression means. The expression

除非已知道操作数的类型，否则无法确定一个特定表达式的含义。下面的表达式

i + j

might mean integer addition, concatenation of **strings**, floating-point addition, or something else entirely. How the expression is evaluated depends on the types of i and j.

既可能是整数的加法操作、字符串的串接或者浮点数的加法操作，也完全可能是其他的操作。如何计算该表达式的值，完全取决于 i 和 j 的数据类型。

There are both **unary** operators and **binary** operators. [Unary operators](#), such as address-of (&) and dereference (*), act on one operand. Binary operators, such as addition (+) and subtraction (-), act on two operands. There is also one ternary operator that takes three operands. We'll look at this operator in [Section 5.7](#) (p. 165).

C++ 提供了一元操作符和二元操作符两种操作符。作用在一个操作数上的操作符称为一元操作符，如取地址操作符 (&) 和解引用操作符 (*)；而二元操作符则作用于两个操作数上，如加法操作符 (+) 和减法操作符 (-)。除此之外，C++ 还提供了一个使用三个操作数的三元操作符 (ternary operator)，我们将在[第 5.7 节](#)介绍它。

Some **symbols**, such as *, are used to represent both a unary and a binary operator. The * symbol is used as the (unary) dereference operator and as the (binary) multiplication operator. The uses of the symbol are independent; it can be helpful to think of them as two different symbols. The context in which an operator symbol is used always determines whether the symbol represents a unary or binary operator.

有些符号 (symbols) 既可表示一元操作也可表示二元操作。例如，符号 * 既可以作为（一元）解引用操作符，也可以作为（二元）乘法操作符，这两种用法相互独立、各不相关，如果将其视为两个不同的符号可能会更容易理解些。对于这类操作符，需要根据该符号所处的上下文来确定它代表一元操作还是二元操作。

Operators impose requirements on the type(s) of their operand(s). The language defines the type requirements for the operators when applied to built-in or compound types. For example, the dereference operator, when applied to an object of built-in type, requires that its operand be a pointer type. Attempting to dereference an object of any other built-in or compound type is an error.

操作符对其操作数的类型有要求，如果操作符应用于内置或复合类型的操作数，则由 C++ 语言定义其类型要求。例如，用于内置类型对象的解引用操作符要求其操作数必须是指针类型，对任何其他内置类型或复合类型对象进行解引用将导致错误的产生。

The binary operators, when applied to operands of built-in or compound type, usually require that the operands be the same type, or types that can be converted to a common type. We'll look at conversions in [Section 5.12](#) (p. 178). Although the rules can be complex, for the most part conversions happen in expected ways. For example, we can convert an integer to floating-point, and vice versa, but we cannot convert a pointer type to floating-point.

对于操作数为内置或复合类型的二元操作符，通常要求它的两个操作数具有相同的数据类型，或者其类型可以转换为同一种数据类型。关于类型转换，我们将在[第 5.12 节](#)学习。尽管规则可能比较复杂，但大部分的类型转换都可按预期的方式进行。例如，整型可转换为浮点类型，反之亦然，但不能将指针类型转换为浮点类型。

Understanding expressions with multiple operators requires understanding operator [precedence](#), [associativity](#), and the [order of evaluation](#) of the operands. For example, the expression

要理解由多个操作符组成的表达式，必须先理解操作符的[优先级](#)、[结合性](#)和操作数的[求值顺序](#)。例如，表达式

5 + 10 * 20 / 2;

uses addition, multiplication, and division. The result of this expression depends on how the operands are grouped to the operators. For example, the operands to the * operator could be 10 and 20, or 10 and 20/2, or 15 and 20 or 15 and 20/2. Associativity and precedence rules specify the grouping of operators and their operands. In C++ this expression evaluates to 105, which is the result of multiplying 10 and 20, dividing that result by 2, and then adding 5.

使用了加法、乘法和除法操作。该表达式的值取决于操作数与操作符如何结合。例如，乘法操作符 * 的操作数可以是 10 和 20，也可以是 10 和 20 / 2，或者 15 和 20 或 15 和 20 / 2。结合性和优先级规则规定了操作数与操作符的结合方式。在 C++ 语言中，该表达式的值应是 105，10 和 20 先做乘法操作，然后其结果除以 2，再加 5 即为最后结果。

Knowing how operands and operators are grouped is not always sufficient to determine the result. It may also be necessary to know in what order the operands to each operator are evaluated. Each operator controls what assumptions, if any, can be made as to the order in which the operands will be evaluated that is, whether we can assume that the left-hand operand is always evaluated before the right or not. Most operators do not guarantee a particular order of evaluation. We will cover these topics in [Section 5.10](#) (p. 168).

求解表达式时，仅了解操作数和操作符如何结合是不够的，还必须清楚操作符上每一个操作数的求值顺序。每个操作符都控制了其假定的求值顺序，即，我们是否可以假定左操作数总是先于右操作数求值。大部分的操作符无法保证某种特定的求值次序，我们将于第 5.10 节讨论这个问题。

Team LiB

◀ PREVIOUS NEXT ▶

5.1. Arithmetic Operators

5.1. 算术操作符

Unless noted otherwise, these operators may be applied to any of the arithmetic types ([Section 2.1](#), p. 34), or any type that can be converted to an arithmetic type.

除非特别说明，表5-1所示操作符可用于任意算术类型（[第 2.1 节](#)）或者任何可转换为算术类型的数据类型。

The table groups the operators by their precedence—the unary operators have the highest precedence, then the multiplication and division operators, and then the binary addition and subtraction operators. Operators of higher precedence group more tightly than do operators with lower precedence. These operators are all left associative, meaning that they group left to right when the precedence levels are the same.

表 5.1 按优先级来对操作符进行分组——一元操作符优先级最高，其次是乘、除操作，接着是二元的加、减法操作。高优先级的操作符要比低优先级的结合得更紧密。这些算术操作符都是左结合，这就意味着当操作符的优先级相同时，这些操作符从左向右依次与操作数结合。

Table 5.1. Arithmetic Operators

表 5.1. 算术操作符

Operator	Function	Use
操作符	功能	用法
+	unary plus (一元正号)	+ expr
-	unary minus (一元负号)	- expr
*	multiplication (乘法)	expr * expr
/	division (除法)	expr / expr
%	remainder (求余)	expr % expr
+	addition (加法)	expr + expr
-	subtraction (减法)	expr - expr

Applying precedence and associativity to the previous expression:

对于前述表达式

5 + 10 * 20 / 2;

we can see that the operands to the multiplication operator (*) are 10 and 20. The result of that expression and 2 are the operands to the division operator (/). The result of that division and 5 are the operands to the addition operator (+).

考虑优先级与结合性，可知该表达式先做乘法（*）操作，其操作数为 10 和 20，然后以该操作的结果和 2 为操作数做除法（/）操作，其结果最后与操作数 5 做加法（+）操作。

The unary minus operator has the obvious meaning. It negates its operand:

一元负号操作符具有直观的含义，它对其操作数取负：

```
int i = 1024;
int k = -i; // negates the value of its operand
```

Unary plus returns the operand itself. It makes no change to its operand.

一元正号操作符则返回操作数本身，对操作数不作任何修改。

Caution: Overflow and Other Arithmetic Exceptions

警告：溢出和其他算术异常

The result of evaluating some arithmetic expressions is undefined. Some expressions are undefined due to the nature of mathematics for example, division by zero. Others are undefined due to the nature of computers such as overflow, in which a value is computed that is too large for its type.

某些算术表达式的求解结果未定义，其中一部分由数学特性引起，例如除零操作；其他则归咎于计算机特性，如溢出：计算出的数值超出了其类型的表示范围。

Consider a machine on which `shorts` are 16 bits. In that case, the maximum `short` is 32767. Given only 16 bits, the following compound assignment overflows:

考虑某台机器，其 `short` 类型为 16 位，能表示的最大值是 32767。假设 `short` 类型只有 16 位，下面的复合赋值操作将会溢出：

```
// max value if shorts are 8 bits
short short_value = 32767;
short ival = 1;
// this calculation overflows
short_value += ival;
cout << "short_value: " << short_value << endl;
```

Representing a signed value of 32768 requires 17 bits, but only 16 are available. On many systems, there is no compile-time or run-time warning when an overflow might occur. The actual value put into `short_value` varies across different machines. On our system the program completes and writes

表示 32768 这个有符号数需 17 位的存储空间，但是这里仅有 16 位，于是导致溢出现象的发生，此时，许多系统都不会给出编译时或运行时的警告。对于不同的机器，上述例子的 `short_value` 变量真正获得的值不尽相同。在我们的系统上执行该程序后将得到：

```
short_value: -32768
```

The value "wrapped around:" The sign bit, which had been 0, was set to 1, resulting in a negative value. Because the arithmetic types have limited size, it is always possible for some calculations to overflow. Adhering to the recommendations from the "Advice" box on page 38 can help avoid such problems.

其值“截断（wrapped around）”，将符号位的值由 0 设为 1，于是结果变为负数。因为算术类型具有有限的长度，因此计算后溢出的现象常常发生。遵循第 2.2 节建议框中给出的建议将有助于避免此类问题。

The binary `+` and `-` operators may also be applied to pointer values. The use of these operators with pointers was described in [Section 4.2.4](#) (p. 123).

二元 `+`、`-` 操作符也可用于指针值，对指针使用这些操作符的用法将在[第 4.2.4 节](#)介绍。

The arithmetic operators, `+`, `-`, `*`, and `/` have their obvious meanings: addition, subtraction, multiplication, and division. Division between integers results in an integer. If the quotient contains a fractional part, it is truncated:

算术操作符 `+`、`-`、`*` 和 `/` 具有直观的含义：加法、减法、乘法和除法。对两个整数做除法，结果仍为整数，如果它的商包含小数部分，则小数部分会被截除：

```
int ival1 = 21/6; // integral result obtained by truncating the remainder
int ival2 = 21/7; // no remainder, result is an integral value
```

Both `ival1` and `ival2` are initialized with a value of 3.

`ival1` 和 `ival2` 均被初始化为 3。

The `%` operator is known as the "remainder" or the "modulus" operator. It computes the remainder of dividing the left-hand operand by the right-hand operand. This operator can be applied only to operands of the integral types: `bool`, `char`, `short`, `int`, `long`, and their associated `unsigned` types:

操作符 `%` 称为“求余（remainder）”或“求模（modulus）”操作符，用于计算左操作数除以右操作数的余数。该操作符的操作数只能为整型，包括 `bool`、`char`、`short`、`int` 和 `long` 类型，以及对应的 `unsigned` 类型：

```
int ival = 42;
double dval = 3.14;
ival % 12; // ok: returns 6
ival % dval; // error: floating point operand
```

For both division (`/`) and modulus(`%`)，when both operands are positive, the result is positive (or zero). If both operands are negative, the result of division is positive (or zero) and the result of modulus is negative (or zero). If only one operand is negative, then the value of the result is machine-dependent for both operators. The sign is also machine-dependent for modulus; the sign is negative (or zero) for division:

如果两个操作数为正，除法 (`/`) 和求模 (`%`) 操作的结果也是正数（或零）；如果两个操作数都是负数，除法操作的结果为正数（或零），而求模操作的结果则为负数（或零）；如果只有一个操作数为负数，这两种操作的结果取决于机器；求模结果的符号也取决于机器，而除法操作的值则是负数（或零）：

```
21 % 6; // ok: result is 3
21 % 7; // ok: result is 0
-21 % -8; // ok: result is -5
```

Section 5.1. Arithmetic Operators

```
21 % -5; // machine-dependent: result is 1 or -4  
21 / 6; // ok: result is 3  
21 / 7; // ok: result is 3  
-21 / -8; // ok: result is 2  
21 / -5; // machine-dependent: result -4 or -5
```

When only one operand is negative, the sign and value of the result for the modulus operator can follow either the sign of the numerator or of the denominator. On a machine where modulus follows the sign of the numerator then the value of division truncates toward zero. If modulus matches the sign of the denominator, then the result of division truncates toward minus infinity.

当只有一个操作数为负数时，求模操作结果值的符号可依据分子（被除数）或分母（除数）的符号而定。如果求模的结果随分子的符号，则除出来的值向零一侧取整；如果求模与分母的符号匹配，则除出来的值向负无穷一侧取整。

Exercises Section 5.1

Exercise 5.1: Parenthesize the following expression to indicate how it is evaluated. Test your answer by compiling the expression and printing its result.

在下列表达式中，加入适当的圆括号以标明其计算顺序。编译该表达式并输出其值，从而检查你的回答是否正确。

`12 / 3 * 4 + 5 * 15 + 24 % 4 / 2`

Exercise 5.2: Determine the result of the following expressions and indicate which results, if any, are machine-dependent.

计算下列表达式的值，并指出哪些结果值依赖于机器？

`-30 * 3 + 21 / 5`
`-30 + 3 * 21 / 5`
`30 / 3 * 21 % 5`
`-30 / 3 * 21 % 4`

Exercise 5.3: Write an expression to determine whether an `int` value is even or odd.

编写一个表达式判断一个 `int` 型数值是偶数还是奇数。

Exercise 5.4: Define the term overflow. Show three expressions that will overflow.

定义术语“溢出”的含义，并给出导致溢出的三个表达式。

5.2. Relational and Logical Operators

5.2. 关系操作符和逻辑操作符

The relational and logical operators take operands of arithmetic or pointer type and return values of type `bool`.

关系操作符和逻辑操作符（表 5.2）使用算术或指针类型的操作数，并返回 `bool` 类型的值。

Table 5.2. Relational and Logical Operators

表 5.2. 关系操作符和逻辑操作符

<u>Each of these operators yields <code>bool</code></u>		
下列操作符都产生 <code>bool</code> 值		
<u>Operator</u> 操作符	<u>Function</u> 功能	<u>Use</u> 用法
!	logical NOT (逻辑非)	<code>!expr</code>
<	less than (小于)	<code>expr < expr</code>
<=	less than or equal (小于等于)	<code>expr <= expr</code>
>	greater than (大于)	<code>expr > expr</code>
>=	greater than or equal (大于等于)	<code>expr >= expr</code>
==	equality (相等)	<code>expr == expr</code>
!=	inequality (不等)	<code>expr != expr</code>
&&	logical AND (逻辑与)	<code>expr && expr</code>
	logical OR (逻辑或)	<code>expr expr</code>

Logical AND and OR Operators

逻辑与、逻辑或操作符

The logical operators treat their operands as conditions (Section 1.4.1, p. 12). The operand is evaluated; if the result is zero the condition is `false`, otherwise it is `true`. The overall result of the AND operator is `TRue` if and only if both its operands evaluate to `TRue`. The logical OR (`||`) operator evaluates to `true` if either of its operands evaluates to `true`. Given the forms

逻辑操作符将其操作数视为条件表达式（第 1.4.1 节）：首先对操作数求值；若结果为 0，则条件为假（`false`），否则为真（`true`）。仅当逻辑与（`&&`）操作符的两个操作数都为 `true`，其结果才得 `true`。对于逻辑或（`||`）操作符，只要两个操作数之一为 `true`，它的值就为 `true`。给定以下形式：

```
expr1 && expr2 // logical AND
expr1 || expr2 // logical OR
```

`expr2` is evaluated if and only if `expr1` does not by itself determine the result. In other words, we're guaranteed that `expr2` will be evaluated if and only if

仅当由 `expr1` 不能确定表达式的值时，才会求解 `expr2`。也就是说，当且仅当下列情况出现时，必须确保 `expr2` 是可以计算的：

Section 5.2. Relational and Logical Operators

- In a logical AND expression, `expr1` evaluates to `TRUE`. If `expr1` is `false`, then the expression will be `false` regardless of the value of `expr2`. When `expr1` is `true`, it is possible for the expression to be `true` if `expr2` is also `TRUE`.

在逻辑与表达式中, `expr1` 的计算结果为 `true`。如果 `expr1` 的值为 `false`, 则无论 `expr2` 的值是什么, 逻辑与表达式的值都为 `false`。当 `expr1` 的值为 `true` 时, 只有 `expr2` 的值也是 `true`, 逻辑与表达式的值才为 `true`。

- In a logical OR expression, `expr1` evaluates to `false`; if `expr1` is `false`, then the expression depends on whether `expr2` is `true`.

在逻辑或表达式中, `expr1` 的计算结果为 `false`。如果 `expr1` 的值为 `false`, 则逻辑或表达式的值取决于 `expr2` 的值是否为 `true`。



The logical AND and OR operators always evaluate their left operand before the right. The right operand is evaluated only if the left operand does not determine the result. This evaluation strategy is often referred to as "short-circuit evaluation."

逻辑与和逻辑或操作符总是先计算其左操作数, 然后再计算其右操作数。只有在仅靠左操作数的值无法确定该逻辑表达式的结果时, 才会求解其右操作数。我们常常称这种求值策略为“短路求值 (short-circuit evaluation) ”。

A valuable use of the logical AND operator is to have `expr1` evaluate to `false` in the presence of a boundary condition that would make the evaluation of `expr2` dangerous. As an example, we might have a `string` that contains the characters in a sentence and we might want to make the first word in the sentence all uppercase. We could do so as follows:

对于逻辑与操作符, 一个很有价值的用法是: 如果某边界条件使 `expr2` 的计算变得危险, 则应在该条件出现之前, 先让 `expr1` 的计算结果为 `false`。例如, 编写程序使用一个 `string` 类型的对象存储一个句子, 然后将该句子的第一个单词的各字符全部变成大写, 可如下实现:

```
string s("Expressions in C++ are composed...");  
string::iterator it = s.begin();  
// convert first word in s to uppercase  
while (it != s.end() && !isspace(*it)) {  
    *it = toupper(*it); // toupper covered in section 3.2.4 (p. 88)  
    ++it;  
}
```

In this case, we combine our two tests in the condition in the `while`. First we test whether `it` has reached the end of the `string`. If not, `it` refers to a character in `s`. Only if that test succeeds is the right-hand operand evaluated. We're guaranteed that `it` refers to an actual character before we test to see whether the character is a space or not. The loop ends either when a space is encountered or, if there are no spaces in `s`, when we reach the end of `s`.

在这个例子中, `while` 循环判断了两个条件。首先检查 `it` 是否已经到达 `string` 类型对象的结尾, 如果不是, 则 `it` 指向 `s` 中的一个字符。只有当该检验条件成立时, 系统才会计算逻辑与操作符的右操作数, 即在保证 `it` 确实指向一个真正的字符之后, 才检查该字符是否为空格。如果遇到空格, 或者 `s` 中没有空格而已经到达 `s` 的结尾时, 循环结束。

Logical NOT Operator

逻辑非操作符

The logical NOT operator (`!`) treats its operand as a condition. It yields a result that has the opposite truth value from its operand. If the operand evaluates as nonzero, then `!` returns `false`. For example, we might determine that a `vector` has elements by applying the logical NOT operator to the value returned by `empty`:

逻辑非操作符 (`!`) 将其操作数视为条件表达式, 产生与其操作数值相反的条件值。如果其操作数为非零值, 则做 `!` 操作后的结果为 `false`。例如, 可如下在 `vector` 类型对象的 `empty` 成员函数上使用逻辑非操作符, 根据函数返回值判断该对象是否为空:

```
// assign value of first element in vec to x if there is one  
int x = 0;  
if (!vec.empty())  
    x = *vec.begin();
```

The subexpression

如果调用 `empty` 函数返回 `false`, 则子表达式

```
!vec.empty()
```

evaluates to `true` if the call to `empty` returns `false`.

的值为 `true`。

The Relational Operators Do Not Chain Together

不应该串接使用关系操作符

The relational operators (`<`, `<=`, `>`, `>=`) are left associative. The fact that they are left associative is rarely of any use because the relational operators return `bool` results. If we do chain these operators together, the result is likely to be surprising:

关系操作符 (`<`, `<=`, `>`, `>=`) 具有左结合特性。事实上，由于关系操作符返回`bool`类型的结果，因此很少使用其左结合特性。如果把多个关系操作符串接起来使用，结果往往出乎预料：

```
// oops! this condition does not determine if the 3 values are unequal
if (i < j < k) { /* ... */ }
```

As written, this expression will evaluate as `true` if `k` is greater than one! The reason is that the left operand of the second less-than operator is the `True/false` result of the first—that is, the condition compares `k` to the integer values of 0 or 1. To accomplish the test we intended, we must rewrite the expression as follows:

这种写法只要 `k` 大于 1，上述表达式的值就为 `true`。这是因为第二个小于操作符的左操作数是第一个小于操作符的结果：`true` 或 `false`。也就是，该条件将 `k` 与整数 0 或 1 做比较。为了实现我们想要的条件检验，应重写上述表达式如下：

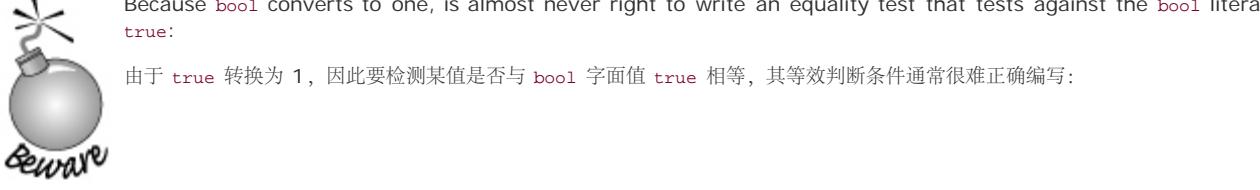
```
if (i < j && j < k) { /* ... */ }
```

Equality Tests and the `bool` Literals

相等测试与 `bool` 字面值

As we'll see in [Section 5.12.2](#) (p. 180) a `bool` can be converted to any arithmetic type—the `bool` value `false` converts to zero and `true` converts to one.

正如[第 5.12.2 节](#)将介绍的，`bool` 类型可转换为任何算术类型——`bool` 值 `false` 用 0 表示，而 `true` 则为 1。



```
if (val == true) { /* ... */ }
```

Either `val` is itself a `bool` or it is a type to which a `bool` can be converted. If `val` is a `bool`, then this test is equivalent to writing

`val` 本身是 `bool` 类型，或者 `val` 具有可转换为 `bool` 类型的数据类型。如果 `val` 是 `bool` 类型，则该判断条件等效于：

```
if (val) { /* ... */ }
```

which is shorter and more direct (although admittedly when first learning the language this kind of abbreviation can be perplexing).

这样的代码更短而且更直接（尽管对初学者来说，这样的缩写可能会令人费解）。

More importantly, if `val` is not a `bool`, then comparing `val` with `true` is equivalent to writing

更重要的是，如果 `val` 不是 `bool` 值，`val` 和 `true` 的比较等效于：

```
if (val == 1) { /* ... */ }
```

which is very different from

这与下面的条件判断完全不同：

```
// condition succeeds if val is any nonzero value
if (val) { /* ... */ }
```

in which any nonzero value in `val` is true. If we write the comparison explicitly, then we are saying that the condition will succeed only for the specific value 1.

此时，只要 `val` 为任意非零值，条件判断都得 `true`。如果显式地书写条件比较，则只有当 `val` 等于指定的 1 值时，条件才成立。

Exercises Section 5.2

Exercise 5.5: Explain when operands are evaluated in the logical AND operator, logical OR operator, and equality operator.

解释逻辑与操作符、逻辑或操作符以及相等操作符的操作数在什么时候计算。

Exercise 5.6: Explain the behavior of the following `while` condition:

解释下列 `while` 循环条件的行为:

```
char *cp = "Hello World";
while (*cp && *cp)
```

Exercise 5.7: Write the condition for a `while` loop that would read `ints` from the standard input and stop when the value read is equal to `42`.

编写 `while` 循环条件从标准输入设备读入整型(`int`)数据, 当读入值为 42 时循环结束。

Exercise 5.8: Write an expression that tests four values, `a`, `b`, `c`, and `d`, and ensures that `a` is greater than `b`, which is greater than `c`, which is greater than `d`.

编写表达式判断四个值 `a`、`b`、`c` 和 `d` 是否满足 `a` 大于 `b`、`b` 大于 `c` 而且 `c` 大于 `d` 的条件。

5.3. The Bitwise Operators

5.3. 位操作符

The bitwise operators take operands of integral type. These operators treat their integral operands as a collection of bits, providing operations to test and set individual bits. In addition, these operators may be applied to `bitset` (Section 3.5, p. 101) operands with the behavior as described here for integral operands.

位操作符（表5-3）使用整型的操作数。位操作符将其整型操作数视为二进制位的集合，为每一位提供检验和设置的功能。另外，这类操作符还可用于 `bitset` 类型（第 3.5 节）的操作数，该类型具有这里所描述的整型操作数的行为。

Table 5.3. Bitwise Operators

表 5.3. 位操作符

Operator	Function	Use
操作符	功能	用法
<code>~</code>	bitwise NOT (位求反)	<code>~expr</code>
<code><<</code>	left shift (左移)	<code>expr1 << expr2</code>
<code>>></code>	right shift (右移)	<code>expr1 >> expr2</code>
<code>&</code>	bitwise AND (位与)	<code>expr1 & expr2</code>
<code>^</code>	bitwise XOR (位异或)	<code>expr1 ^ expr2</code>
<code> </code>	bitwise OR (位或)	<code>expr1 expr2</code>

The type of an integer manipulated by the bitwise operators can be either signed or unsigned. If the value is negative, then the way that the "sign bit" is handled in a number of the bitwise operations is machine-dependent. It is, therefore, likely to differ across implementations; programs that work under one implementation may fail under another.

位操作符操纵的整数的类型可以是有符号的也可以是无符号的。如果操作数为负数，则位操作符如何处理其操作数的符号位依赖于机器。于是它们的应用可能不同：在一个应用环境中实现的程序可能无法用于另一应用环境。



Because there are no guarantees for how the sign bit is handled, we strongly recommend using an `unsigned` type when using an integral value with the bitwise operators.

对于位操作符，由于系统不能确保如何处理其操作数的符号位，所以强烈建议使用`unsigned`整型操作数。

In the following examples we assume that an `unsigned char` has 8 bits. The bitwise NOT operator (`~`) is similar in behavior to the `bitset flip` (Section 3.5.2, p. 105) operation: It generates a new value with the bits of its operand inverted. Each 1 bit is set to 0; each 0 bit is set to 1:

在下面的例子中，假设 `unsigned char` 类型有 8 位。位求反操作符 (`~`) 的功能类似于 `bitset` 的 `flip` 操作（第 3.5.2 节）：将操作数的每一个二进制位取反：将 1 设置为 0、0 设置为 1，生成一个新值：

```
unsigned char bits = 0227;
```

```
bits = ~bits;
```

1	0	0	1	0	1	1	1
---	---	---	---	---	---	---	---

0	1	1	0	1	0	0	0
---	---	---	---	---	---	---	---

The `<<`, `>>` operators are the bitwise shift operators. These operators use their right-hand operand to indicate by how many bits to shift. They yield a value that is a copy of the left-hand operand with the bits shifted as directed by the right-hand operand. The bits are shifted left (`<<`) or right (`>>`), discarding the bits that are shifted off the end.

Section 5.3. The Bitwise Operators

`<<` 和 `>>` 操作符提供移位操作，其右操作数标志要移动的位数。这两种操作符将其左操作数的各个位向左 (`<<`) 或向右 (`>>`) 移动若干个位 (移动的位数由其右操作数指定)，从而产生新的值，并丢弃移出去的位。

```
unsigned char bits = 1;  
  
bits << 1; // left shift  
  
bits << 2; // left shift  
  
bits >> 3; // right shift
```

1	0	0	1	1	0	1	1
0	0	1	1	0	1	1	0
0	1	1	0	1	1	0	0
0	0	0	1	0	0	1	1

The left shift operator (`<<`) inserts 0-valued bits from the right. The right shift operator (`>>`) inserts 0-valued bits from the left if the operand is unsigned. If the operand is signed, it can either insert copies of the sign bit or insert 0-valued bits; which one it uses is implementation defined. The right-hand operand must not be negative and must be a value that is strictly less than the number of bits in the left-hand operand. Otherwise, the effect of the operation is undefined.

左移操作符 (`<<`) 在右边插入 0 以补充空位。对于右移操作符 (`>>`)，如果其操作数是无符号数，则从左边开始插入 0；如果操作数是有符号数，则插入符号位的副本或者 0 值，如何选择需依据具体的实现而定。移位操作的右操作数不可以是负数，而且必须是严格小于左操作数位数的值。否则，操作的效果未定义。

The bitwise AND operator (`&`) takes two integral operands. For each bit position, the result is 1 if both operands contain 1; otherwise, the result is 0.

位与操作 (`&`) 需要两个整型操作数，在每个位的位置，如果两个操作数对应的位都为 1，则操作结果中该位为 1，否则为 0。



It is a common error to confuse the bitwise AND operator (`&`) with the logical AND operator (`&&`) ([Section 5.2](#), p. [152](#)). Similarly, it is common to confuse the bitwise OR operator (`|`) and the logical OR operator (`||`).

常犯的错误是把位与操作 (`&`) 和逻辑与操作 (`&&`) ([第 5.2 节](#)) 混淆了。同样地，位或操作 (`|`) 和逻辑或操作 (`||`) 也很容易搞混。

Here we illustrate the result of bitwise AND of two `unsigned char` values, each of which is initialized by an octal literal:

下面我们用图解的方法说明两个 `unsigned char` 类型值的位与操作，这两个操作数均用八进制字面常量初始化：

```
unsigned char b1 = 0145;  
  
unsigned char b2 = 0257;  
  
unsigned char result = b1 & b2;
```

0	1	1	0	0	1	0	1
1	0	1	0	1	1	1	1
0	0	1	0	0	1	0	1

The bitwise XOR (exclusive or) operator (`^`) also takes two integral operands. For each bit position, the result is 1 if either but not both operands contain 1; otherwise, the result is 0.

位异或 (互斥或, exclusive or) 操作符 (`^`) 也需要两个整型操作数。在每个位的位置，如果两个操作数对应的位只有一个 (不是两个) 为 1，则操作结果中该位为 1，否则为 0。

```
result = b1 ^ b2;
```

1	1	0	0	1	0	1	0
---	---	---	---	---	---	---	---

The bitwise OR (inclusive or) operator (`|`) takes two integral operands. For each bit position, the result is 1 if either or both operands contain 1; otherwise, the result is 0.

位或 (包含或, inclusive or) 操作符 (`|`) 需要两个整型操作数。在每个位的位置，如果两个操作数对应的位有一个或者两个都为 1，则操作结果中该位为 1，否则为 0。

```
result = b1 | b2;
```

1	1	1	0	1	1	1	1
---	---	---	---	---	---	---	---

5.3.1. Using `bitset` Objects or Integral Values

5.3.1. `bitset` 对象或整型值的使用

We said that the `bitset` class was easier to use than the lower-level bitwise operations on integral values. Let's look at a simple example and show how we might solve a problem using either a `bitset` or the bitwise operators. Assume that a teacher has 30 students in a class. Each week the class is given a pass/fail quiz. We'll track the results of each quiz using one bit per student to represent the pass or fail grade on a given test. We might represent each quiz in either a `bitset` or as an integral value:

`bitset` 类比整型值上的低级位操作更容易使用。观察下面简单的例子，了解如何使用 `bitset` 类型或者位操作来解决问题。假设某老师带了一个班，班中有 30 个学生，每个星期在班上做一次测验，只有及格和不及格两种测验成绩，对每个学生用一个二进制位来记录一次测试及格或不及格，以方便我们跟踪每次测验的结果，这样就可以用一个`bitset`对象或整数值来代表一次测验：

```
bitset<30> bitset_quiz1;      // bitset solution
unsigned long int_quiz1 = 0; // simulated collection of bits
```

In the `bitset` case we can define `bitset_quiz1` to be exactly the size we need. By default each of the bits is set to zero. In the case where we use a built-in type to hold our quiz results, we define `int_quiz1` as an `unsigned long`, meaning that it will have at least 32 bits on any machine. Finally, we explicitly initialize `int_quiz1` to ensure that the bits start out with well-defined values.

使用 `bitset` 类型时，可根据所需要的大小明确地定义 `bitset_quiz1`，它的每一个位都默认设置为 0 值。如果使用内置类型来存放测验成绩，则应将变量 `int_quiz1` 定义为 `unsigned long` 类型，这种数据类型在所有机器上都至少拥有32位的长度。最后，显式地初始化 `int_quiz1` 以保证该变量在使用前具有明确定义的值。

The teacher must be able to set and test individual bits. For example, assuming that the student represented by position 27 passed, we'd like to be able to set that bit appropriately:

老师可以设置和检查每个位。例如，假设第27位所表示的学生及格了，则可以使用下面的语句适当地设置对应的位：

```
bitset_quiz1.set(27); // indicate student number 27 passed
int_quiz1 |= 1UL<<27; // indicate student number 27 passed
```

In the `bitset` case we do so directly by passing the bit we want turned on to `set`. The `unsigned long` case will take a bit more explanation. The way we'll set a specific bit is to OR our quiz data with another integer that has only one bit the one we want turned on. That is, we need an `unsigned long` where bit 27 is a one and all the other bits are zero. We can obtain such a value by using the left shift operator and the integer constant 1:

如果使用 `bitset` 实现，可直接传递要置位的位给 `set` 函数。而用 `unsigned long` 实现时，实现的方法则比较复杂。设置指定位的方法是：将测验数据与一个整数做位或操作，该整数只有一个指定的位为 1。也就是说，我们需要一个只有第 27 位为 1 其他位都为0的无符号长整数 (`unsigned long`)，这样的整数可用左移操作符和整型常量 1 生成：

```
1UL << 27; // generate a value with only bit number 27 set
```

Now when we bitwise OR this value with `int_quiz1`, all the bits except bit 27 will remain unchanged. That bit will be turned on. We use a compound assignment ([Section 1.4.1](#), p. 13) to OR this value into `int_quiz1`. This operator, `|=`, executes in the same way that `+=` does. It is equivalent to the more verbose:

然后让这个整数与 `int_quiz1` 做位或操作，操作后，除了第 27 位外其他所有位的值都保持不变，而第 27 位则被设置为 1。这里，使用复合赋值操作（[第 1.4.1 节](#)）将位或操作的结果赋给 `int_quiz1`，该操作符 `|=` 操作的方法与 `+=` 相同。于是，上述功能等效于下面更详细的形式：

```
// following assignment is equivalent to int_quiz1 |= 1UL << 27;
int_quiz1 = int_quiz1 | 1UL << 27;
```

Imagine that the teacher reexamined the quiz and discovered that student 27 actually had failed the test. The teacher must now turn off bit 27:

如果老师重新复核测验成绩，发现第 27 个学生实际上在该次测验中不及格，这时老师应把第 27 位设置为 0：

```
bitset_quiz1.reset(27); // student number 27 failed
int_quiz1 &= ~(1UL<<27); // student number 27 failed
```

Again, the `bitset` version is direct. We `reset` the indicated bit. For the simulated case, we need to do the inverse of what we did to set the bit: This time we'll need an integer that has bit 27 turned off and all the other bits turned on. We'll bitwise AND this value with our quiz data to turn off just that bit. We can obtain a value with all but bit 27 turned on by inverting our previous value. Applying the bitwise NOT to the previous integer will turn on every bit except the 27th. When we bitwise AND this value with `int_quiz1`, all except bit 27 will remain unchanged.

使用 `bitset` 的版本可直接实现该功能，只要复位 (`reset`) 指定的位即可。而对于另一种情况，则需通过反转左移操作后的结果来实现设置：此时，我们需要一个只有第 27 位为 0 而其他位都为 1 的整数。然后将这个整数与测验数据做位与操作，把指定的位设置为 0。位求反操作使得除了第 27 位外其他位都设置为 1，然后此值和 `int_quiz1` 做位与操作，保证了除第 27 位外所有的位都保持不变。

Finally, we might want to know how the student at position 27 fared. To do so, we could write

最后，可通过以下代码获知第 27 个学生是否及格：

```
bool status;
status = bitset_quiz1[27]; // how did student number 27 do?
status = int_quiz1 & (1UL<<27); // how did student number 27 do?
```

In the `bitset` case we can fetch the value directly to determine how that student did. In the `unsigned long` case, the first step is to set the 27th bit of an integer to 1. The bitwise AND of this value with `int_quiz1` evaluates to nonzero if bit 27 of `int_quiz1` is also on; otherwise, it evaluates to

Section 5.3. The Bitwise Operators

zero.

使用 `bitset` 的版本中，可直接读取其值判断他是否及格。使用 `unsigned long` 时，首先要把一个整数的第 27 位设置为 1，然后用该整数和 `int_quiz1` 做位与操作，如果 `int_quiz1` 的第 27 位为 1，则结果为非零值，否则，结果为零。



In general, the library `bitset` operations are more direct, easier to read, easier to write, and more likely to be used correctly. Moreover, the size of a `bitset` is not limited by the number of bits in an `unsigned`. Ordinarily `bitset` should be used in preference to lower-level direct bit manipulation of integral values.

一般而言，标准库提供的 `bitset` 操作更直接、更容易阅读和书写、正确使用的可能性更高。而且，`bitset` 对象的大小不受 `unsigned` 数的位数限制。通常来说，`bitset` 优于整型数据的低级直接位操作。

Exercises Section 5.3.1

Exercise 5.9: Assume the following two definitions:

假设有下面两个定义

```
unsigned long ull = 3, ul2 = 7;
```

What is the result of each of the following expressions?

下列表达式的结果是什么？

- (a) `ull & ul2` (c) `ull | ul2`
(b) `ull && ul2` (d) `ull || ul2`

Exercise 5.10: Rewrite the `bitset` expressions that set and reset the quiz results using a subscript operator.

重写 `bitset` 表达式：使用下标操作符对测验结果进行置位（置 1）和复位（置 0）。

5.3.2. Using the Shift Operators for IO

5.3.2. 将移位操作符用于IO

The IO library redefines the bitwise `>>` and `<<` operators to do input and output. Even though many programmers never need to use the bitwise operators directly, most programs do make extensive use of the overloaded versions of these operators for IO. When we use an overloaded operator, it has the same precedence and associativity as is defined for the built-in version of the operator. Therefore, programmers need to understand the precedence and associativity of these operators even if they never use them with their built-in meaning as the shift operators.

输入输出标准库（IO library）分别重载了位操作符 `>>` 和 `<<` 用于输入和输出。即使很多程序员从未直接使用过位操作符，但是相当多的程序都大量用到这些操作符在IO标准库中的重载版本。重载的操作符与该操作符的内置类型版本有相同的优先级和结合性。因此，即使程序员从不使用这些操作符的内置含义来实现移位操作，但是还是应该先了解这些操作符的优先级和结合性。

The IO Operators Are Left Associative

IO 操作符为左结合

Like the other [binary operators](#), the shift operators are left associative. These operators group from left to right, which accounts for the fact that we can concatenate input and output operations into a single statement:

像其他[二元操作符](#)一样，移位操作符也是左结合的。这类操作符从左向右地结合，正好说明了程序员为什么可以把多个输入或输出操作连接为单个语句：

```
cout << "hi" << " there" << endl;
```

executes as:

执行为：

Section 5.3. The Bitwise Operators

```
( (cout << "hi") << " there" ) << endl;
```

In this statement, the operand "hi" is grouped with the first `<<` symbol. Its result is grouped with the second, and then that result is grouped to the third.

在这个语句中，操作数"hi"与第一个 `<<` 符号结合，其计算结果与第二个 `<<` 符号结合，第二个 `<<` 符号操作后，其结果再与第三个 `<<` 符号结合。

The shift operators have midlevel precedence: lower precedence than the arithmetic operators but higher than the relational, assignment, or conditional operators. These relative precedence levels affect how we write IO expressions involving operands that use operators with lower precedence. We often need to use parentheses to force the right grouping:

移位操作符具有中等优先级：其优先级比算术操作符低，但比关系操作符、赋值操作符和条件操作符优先级高。若 IO 表达式的操作数包含了比IO操作符优先级低的操作符，相关的优先级别将影响书写该表达式的方式。通常需使用圆括号强制先实现右结合：

```
cout << 42 + 10; // ok, + has higher precedence, so the sum is printed  
cout << (10 < 42); // ok: parentheses force intended grouping; prints 1  
cout << 10 < 42; // error: attempt to compare cout to 42!
```

The second `cout` is interpreted as

第二个`cout`语句解释为：

```
(cout << 10) < 42;
```

this expression says to "write 10 onto `cout` and then compare the result of that operation (e.g., `cout`) to 42."

该表达式说"将 10 写到 `cout`，然后用此操作（也就是 `cout`）的结果与 42 做比较"。

Team LiB

◀ PREVIOUS NEXT ▶

5.4. Assignment Operators

5.4. 赋值操作符

The left-hand operand of an assignment operator must be a non`const` lvalue. Each of these assignments is illegal:

赋值操作符的左操作数必须是非 `const` 的左值。下面的赋值语句是不合法的：

```
int i, j, ival;
const int ci = i; // ok: initialization not assignment
1024 = ival;      // error: literals are rvalues
i + j = ival;    // error: arithmetic expressions are rvalues
ci = ival;       // error: can't write to ci
```

Array names are nonmodifiable lvalues: An array cannot be the target of an assignment. Both the subscript and dereference operators return lvalues. The result of dereference or subscript, when applied to a non`const` array, can be the left-hand operand of an assignment:

数组名是不可修改的左值：因此数组不可用作赋值操作的目标。而下标和解引用操作符都返回左值，因此当将这两种操作用于非 `const` 数组时，其结果可作为赋值操作的左操作数：

```
int ia[10];
ia[0] = 0;    // ok: subscript is an lvalue
*ia = 0;      // ok: dereference also is an lvalue
```

The result of an assignment is the left-hand operand; the type of the result is the type of the left-hand operand.

赋值表达式的值是其左操作数的值，其结果的类型为左操作数的类型。

The value assigned to the left-hand operand ordinarily is the value that is in the right-hand operand. However, assignments where the types of the left and right operands differ may require conversions that might change the value being assigned. In such cases, the value stored in the left-hand operand might differ from the value of the right-hand operand:

通常，赋值操作将其右操作数的值赋给左操作数。然而，当左、右操作数的类型不同时，该操作实现的类型转换可能会修改被赋的值。此时，存放在左、右操作数里的值并不相同：

```
ival = 0;        // result: type int value 0
ival = 3.14159; // result: type int value 3
```

Both these assignments yield values of type `int`. In the first case the value stored in `ival` is the same value as in its right-hand operand. In the second case the value stored in `ival` is different from the right-hand operand.

上述两个赋值语句都产生 `int` 类型的值，第一个语句中 `ival` 的值与右操作数的值相同；但是在第二个语句中，`ival` 的值则与右操作数的值不相同。

5.4.1. Assignment Is Right Associative

5.4.1. 赋值操作的右结合性

Like the subscript and dereference operators, assignment returns an lvalue. As such, we can perform multiple assignments in a single expression, provided that each of the operands being assigned is of the same general type:

与下标和解引用操作符一样，赋值操作也返回左值。同理，只要被赋值的每个操作数都具有相同的通用类型，C++ 语言允许将这多个赋值操作写在一个表达式中：

```
int ival, jval;
ival = jval = 0; // ok: each assigned 0
```

Unlike the other binary operators, the assignment operators are right associative. We group an expression with multiple assignment operators from right to left. In this expression, the result of the rightmost assignment (i.e., `jval`) is assigned to `ival`. The types of the objects in a multiple assignment either must be the same type or of types that can be converted ([Section 5.12](#), p. 178) to one another:

与其他二元操作符不同，赋值操作具有右结合特性。当表达式含有多个赋值操作符时，从右向左结合。上述表达式，将右边赋值操作的结果（也就是 `jval`）赋给 `ival`。多个赋值操作中，各对象必须具有相同的数据类型，或者具有可转换（[第 5.12 节](#)）为同一类型的数据类型：

```
int ival; int *pval;
ival = pval = 0; // error: cannot assign the value of a pointer to an int
string s1, s2;
s1 = s2 = "OK"; // ok: "OK" converted to string
```

Section 5.4. Assignment Operators

The first assignment is illegal because `ival` and `pval` are objects of different types. It is illegal even though zero happens to be a value that could be assigned to either object. The problem is that the result of the assignment to `pval` is a value of type `int*`, which cannot be assigned to an object of type `int`. On the other hand, the second assignment is fine. The string literal is converted to `string`, and that `string` is assigned to `s2`. The result of that assignment is `s2`, which is then assigned to `s1`.

第一个赋值语句是不合法的，因为 `ival` 和 `pval` 是不同类型的对象。虽然 0 值恰好都可以赋给这两个对象，但该语句仍然错误。因为问题在于给 `pval` 赋值的结果是一个 `int*` 类型的值，不能将此值赋给 `int` 类型的对象。另一方面，第二个赋值语句则是正确的。字符串字面值可以转换为 `string` 类型，`string` 类型的值可赋给 `s2` 变量。右边赋值操作的结果为 `s2`，再将此结果值赋给 `s1`。

5.4.2. Assignment Has Low Precedence

5.4.2. 赋值操作具有低优先级

Inside a condition is another common place where assignment is used as a part of a larger expression. Writing an assignment in a condition can shorten programs and clarify the programmer's intent. For example, the following loop uses a function named `get_value`, which we assume returns `int` values. We can test those values until we obtain some desired valuesay, 42:

另一种通常的用法，是将赋值操作写在条件表达式中，把赋值操作用作长表达式的一部分。这种做法可缩短程序代码并阐明程序员的意图。例如，下面的循环调用函数 `get_value`，假设该函数返回 `int` 数值，通过循环检查这些返回值，直到获得需要的值为止——这里是 42：

```
int i = get_value(); // get_value returns an int
while (i != 42) {
    // do something ...
    i = get_value();
}
```

The program begins by getting the first value and storing it in `i`. Then it establishes the loop, which tests whether `i` is 42, and if not, does some processing. The last statement in the loop gets a value from `get_value()`, and the loop repeats. We can write this loop more succinctly as

首先，程序将所获得的第一个值存储在 `i` 中，然后建立循环检查 `i` 的值是否为 42，如果不是，则做某些处理。循环中的最后一条语句调用 `get_value()` 返回一个值，然后继续循环。该循环可简洁地写为：

```
int i;
while ((i = get_value()) != 42) {
    // do something ...
}
```

The condition now more clearly expresses our intent: We want to continue until `get_value` returns 42. The condition executes by assigning the result returned by `get_value` to `i` and then comparing the result of that assignment with 42.

现在，循环条件更清晰地表达了程序员的意图：持续循环直到 `get_value` 返回 42 为止。在循环条件中，将 `get_value` 返回的值赋给 `i`，然后判断赋值的结果是否为 42。



The additional parentheses around the assignment are necessary because assignment has lower precedence than inequality.

在赋值操作上加圆括号是必需的，因为赋值操作符的优先级低于不等操作符。

Without the parentheses, the operands to `!=` would be the value returned from calling `get_value` and 42. The `true` or `false` result of that test would be assigned to `i` clearly not what we intended!

如果没有圆括号，操作符 `!=` 的操作数则是调用 `get_value` 返回的值和 42，然后将该操作的结果 `true` 或 `false` 赋给 `i`——显然这并不是我们想要的。

Beware of Confusing Equality and Assignment Operators

谨防混淆相等操作符和赋值操作符

The fact that we can use assignment in a condition can have surprising effects:

可在条件表达式中使用赋值操作，这个事实往往会带来意外的效果：

```
if (i = 42)
```

This code is legal: What happens is that 42 is assigned to `i` and then the result of the assignment is tested. In this case, 42 is nonzero, which is interpreted as a `true` value. The author of this code almost surely intended to test whether `i` was 42:

此代码是合法的：将 42 赋给 `i`，然后检验赋值的结果。此时，42 为非零值，因此解释为 `true`。其实，程序员的目的显然是想判断 `i` 的值是否为 42：

Section 5.4. Assignment Operators

```
if (i == 42)
```

Bugs of this sort are notoriously difficult to find. Some, but not all, compilers are kind enough to warn about code such as this example.

这种类型的程序错误很难发现。有些（并非全部）编译器会为类似于上述例子的代码提出警告。

Exercises Section 5.4.2

Exercise 5.11: What are the values of `i` and `d` after the each assignment?

请问每次赋值操作完成后，`i` 和 `d` 的值分别是多少？

```
int i; double d;
d = i = 3.5;
i = d = 3.5;
```

Exercise 5.12: Explain what happens in each of the `if` tests:

解释每个 `if` 条件判断产生什么结果？

```
if (42 = i) // . .
if (i = 42) // . . .
```

5.4.3. Compound Assignment Operators

5.4.3. 复合赋值操作符

We often apply an operator to an object and then reassign the result to that same object. As an example, consider the `sum` program from page 14:

我们常常在对某个对象做某种操作后，再将操作结果重新赋给该对象。例如，考虑[第 1.4.2 节](#)的求和程序：

```
int sum = 0;
// sum values from 1 up to 10 inclusive
for (int val = 1; val <= 10; ++val)
    sum += val; // equivalent to sum = sum + val
```

This kind of operation is common not just for addition but for the other arithmetic operators and the bitwise operators. There are compound assignments for each of these operators. The general syntactic form of a compound assignment operator is

C++ 语言不仅对加法，而且还对其他算术操作符和位操作符提供了这种用法，称为复合赋值操作。复合赋值操作符的一般语法格式为：

```
a op= b;
```

where `op=` may be one of the following ten operators:

其中，`op=` 可以是下列十个操作符之一：

```
+ = - = * = / = % = // arithmetic operators
<<= >>= & = ^ = |= // bitwise operators
```

Each compound operator is essentially equivalent to

这两种语法形式存在一个显著的差别：使用复合赋值操作时，左操作数只计算了一次；而使用相似的长表达式时，该操作数则计算了两次，第一次作为右操作数，而第二次则用做左操作数。除非考虑可能的性能价值，在很多（可能是大部分的）上下文环境里这个差别不是本质性的。

```
a = a op b;
```

There is one important difference: When we use the compound assignment, the left-hand operand is evaluated only once. If we write the similar longer version, that operand is evaluated twice: once as the right-hand operand and again as the left. In many, perhaps most, contexts this difference is immaterial aside from possible performance consequences.

这两种语法形式存在一个显著的差别：使用复合赋值操作时，左操作数只计算了一次；而使用相似的长表达式时，该操作数则计算了两次，第一次作为右操作数，而第二次则用做左操作数。除非考虑可能的性能价值，在很多（可能是大部分的）上下文环境里这个差别不是本质性的。

Exercises Section 5.4.3

Exercise The following assignment is illegal. Why? How would you correct it?

5.13:

下列赋值操作是不合法的，为什么？怎样改正？

```
double dval; int ival; int *pi;  
dval = ival = pi = 0;
```

Exercise Although the following are legal, they probably do not behave as the programmer expects.

5.14: Why? Rewrite the expressions as you think they should be.

虽然下列表达式都是合法的，但并不是程序员期望的操作，为什么？怎样修改这些表达式以使其能反映程序员的意图？

```
(a) if (ptr = retrieve_pointer() != 0)  
(b) if (ival = 1024)  
(c) ival += ival + 1;
```

5.5. Increment and Decrement Operators

5.5. 自增和自减操作符

The increment (`++`) and decrement (`--`) operators provide a convenient notational shorthand for adding or subtracting 1 from an object. There are two forms of these operators: prefix and postfix. So far, we have used only the prefix increment, which increments its operand and yields the *changed* value as its result. The prefix decrement operates similarly, except that it decrements its operand. The postfix versions of these operators increment (or decrement) the operand but yield a copy of the original, *unchanged* value as its result:

自增 (`++`) 和自减 (`--`) 操作符为对象加1或减1操作提供了方便简短的实现方式。它们有前置和后置两种使用形式。到目前为止，我们已经使用过前自增操作，该操作使其操作数加1，操作结果是修改后的值。同理，前自减操作使其操作数减1。这两种操作符的后置形式同样对其操作数加1（或减1），但操作后产生操作数原来的、未修改的值作为表达式的结果：

```
int i = 0, j;
j = ++i; // j = 1, i = 1: prefix yields incremented value
j = i++; // j = 1, i = 2: postfix yields unincremented value
```

Because the prefix version returns the incremented value, it returns the object itself as an lvalue. The postfix versions return an rvalue.

因为前置操作返回加1后的值，所以返回对象本身，这是左值。而后置操作返回的则是右值。

Advice: Use Postfix Operators Only When Necessary

建议：只有在必要时才使用后置操作符

Readers from a C background might be surprised that we use the prefix increment in the programs we've written. The reason is simple: The prefix version does less work. It increments the value and returns the incremented version. The postfix operator must store the original value so that it can return the unincremented value as its result. For `int`s and pointers, the compiler can optimize away this extra work. For more complex iterator types, this extra work potentially could be more costly. By habitually favoring the use of the prefix versions, we do not have to worry if the performance difference matters.

有使用 C 语言背景的读者可能会觉得奇怪，为什么要在程序中使用前自增操作。道理很简单：因为前置操作需要做的工作更少，只需加 1 后返回加 1 后的结果即可。而后置操作符则必须先保存操作数原来的值，以便返回未加 1 之前的值作为操作的结果。对于 `int` 型对象和指针，编译器可优化掉这项额外工作。但是对于更多的复杂迭代器类型，这种额外工作可能会花费更大的代价。因此，养成使用前置操作这个好习惯，就不必操心性能差异的问题。

Postfix Operators Return the Unincremented Value

后置操作符返回未加1的值

The postfix version of `++` and `--` is used most often when we want to use the current value of a variable and increment it in a single compound expression:

当我们希望在单个复合表达式中使用变量的当前值，然后再加1时，通常会使用后置的 `++` 和 `--` 操作：

```
vector<int> ivec;           // empty vector
int cnt = 10;
// add elements 10...1 to ivec
while (cnt > 0)
    ivec.push_back(cnt--); // int postfix decrement
```

This program uses the postfix version of `--` to decrement `cnt`. We want to assign the value of `cnt` to the next element in the `vector` and then decrement `cnt` before the next iteration. Had the loop used the prefix version, then the decremented value of `cnt` would be used when creating the elements in `ivec` and the effect would be to add elements from 9 down to 0.

这段程序使用了后置的 `--` 操作实现 `cnt` 减 1。我们希望把 `cnt` 的值赋给 `vector` 对象的下一个元素，然后在下次迭代前 `cnt` 的值减 1。如果在循环中使用前置操作，则是用 `cnt` 减 1 后的值创建 `ivec` 的新元素，结果是将 9 至 0 十个元素依次添加到 `ivec` 中。

Combining Dereference and Increment in a Single Expression

在单个表达式中组合使用解引用和自增操作

The following program, which prints the contents of `ivec`, represents a very common C++ programming pattern:

下面的程序使用了一种非常通用的 C++ 编程模式输出 `ivec` 的内容:

```
vector<int>::iterator iter = ivec.begin();
// prints 10 9 8 ... 1
while (iter != ivec.end())
    cout << *iter++ << endl; // iterator postfix increment
```



The expression `*iter++` is usually very confusing to programmers new to both C++ and C.

如果程序员对 C++ 和 C 语言都不太熟悉，则常常会弄不清楚表达式 `*iter++` 的含义。

The precedence of postfix increment is higher than that of the dereference operator, so `*iter++` is equivalent to `*(iter++)`. The subexpression `iter++` increments `iter` and yields a copy of the previous value of `iter` as its result. Accordingly, the operand of `*` is a copy of the unincremented value of `iter`.

由于后自增操作的优先级高于解引用操作，因此 `*iter++` 等效于 `*(iter++)`。子表达式 `iter++` 使 `iter` 加 1，然后返回 `iter` 原值的副本作为该表达式的结果。因此，解引用操作 `*` 的操作数是 `iter` 未加 1 前的副本。

This usage relies on the fact that postfix increment returns a copy of its original, unincremented operand. If it returned the incremented value, we'd dereference the incremented value, with disastrous results: The first element of `ivec` would not get written. Worse, we'd attempt to dereference one too many elements!

这种用法的根据在于后自增操作返回其操作数原值（没有加 1）的副本。如果返回的是加 1 后的值，则解引用该值将导致错误的结果：`ivec` 的第一个元素没有输出，并企图对一个多余的元素进行解引用。

Advice: Brevity Can Be a Virtue

建议：简洁即是美

Programmers new to C++ who have not previously programmed in a C-based language often have trouble with the terseness of some expressions. In particular, expressions such as `*iter++` can be bewildering at first. Experienced C++ programmers value being concise. They are much more likely to write

没有 C 语言基础的 C++ 新手，时常会因精简的表达式而苦恼，特别是像 `*iter++` 这类令人困惑的表达式。有经验的 C++ 程序员非常重视简练，他们更喜欢这么写：

```
cout << *iter++ << endl;
```

than the more verbose equivalent

而不采用下面这种冗长的等效代码：

```
cout << *iter << endl;
++iter;
```

For programmers new to C++, the second form is clearer because the action of incrementing the iterator and fetching the value to print are kept separate. However, the first version is much more natural to most C++ programmers.

对于初学 C++ 的程序员来说，第二种形式更清晰，因为给迭代器加 1 和获取输出值这两个操作是分开来实现的。但是更多的 C++ 程序员更习惯使用第一种形式。

It is worthwhile to study examples of such code until their meanings are immediately clear. Most C++ programs use succinct expressions rather than more verbose equivalents. Therefore, C++ programmers must be comfortable with such usages. Moreover, once these expressions are familiar, you will find them less error-prone.

要不断地研究类似的代码，最后达到一目了然的地步。大部分的 C++ 程序员更喜欢使用简洁的表达式而非冗长的等效表达式。因此，C++ 程序员必须熟悉这种用法。而且，一旦熟悉了这类表达式，我们会发现使用起来不容易出错。

Exercises Section 5.5

Exercise Explain the difference between prefix and postfix increment.

5.15: 解释前自增操作和后自增操作的差别。

Exercise Why do you think C++ wasn't named ++C?

5.16: 你认为为什么C++不叫做++C?

Exercise What would happen if the `while` loop that prints the contents of a `vector` used the prefix increment operator?

如果输出vector内容的while循环使用前自增操作符，那会怎么样？

5.6. The Arrow Operator

5.6. 箭头操作符

The arrow operator (`->`) provides a synonym for expressions involving the dot and dereference operators. The dot operator (Section 1.5.2, p. 25) fetches an element from an object of class type:

C++ 语言为包含点操作符和解引用操作符的表达式提供了一个同义词：箭头操作符 (`->`)。点操作符 (第 1.5.2 节) 用于获取类类型对象的成员：

```
item1.same_isbn(item2); // run the same_isbn member of item1
```

If we had a pointer (or iterator) to a `Sales_item`, we would have to dereference the pointer (or iterator) before applying the dot operator:

如果有一个指向 `Sales_item` 对象的指针 (或迭代器)，则在使用点操作符前，需对该指针 (或迭代器) 进行解引用：

```
Sales_item *sp = &item1;
(*sp).same_isbn(item2); // run same_isbn on object to which sp points
```

Here we dereference `sp` to get the underlying `Sales_item`. Then we use the dot operator to run `same_isbn` on that object. We must parenthesize the dereference because dereference has a lower precedence than dot. If we omit the parentheses, this code means something quite different:

这里，对 `sp` 进行解引用以获得指定的 `Sales_item` 对象。然后使用点操作符调用指定对象的 `same_isbn` 成员函数。在上述用法中，注意必须用圆括号把解引用括起来，因为解引用的优先级低于点操作符。如果漏掉圆括号，则这段代码的含义就完全不同了：

```
// run the same_isbn member of sp then dereference the result!
*sp.same_isbn(item2); // error: sp has no member named same_isbn
```

This expression attempts to fetch the `same_isbn` member of the object `sp`. It is equivalent to

这个表达式企图获得 `sp` 对象的 `same_isbn` 成员。等价于：

```
*(sp.same_isbn(item2)); // equivalent to *sp.same_isbn(item2);
```

However, `sp` is a pointer, which has no members; this code will not compile.

然而，`sp` 是一个没有成员的指针；这段代码无法通过编译。

Because it is easy to forget the parentheses and because this kind of code is a common usage, the language defines the arrow operator as a synonym for a dereference followed by the dot operator. Given a pointer (or iterator) to an object of class type, the following expressions are equivalent:

因为编程时很容易忘记圆括号，而且这类代码又经常使用，所以 C++ 为在点操作符后使用的解引用操作定义了一个同义词：箭头操作符 (`->`)。假设有一个指向类类型对象的指针 (或迭代器)，下面的表达式相互等价：

```
(*p).foo; // dereference p to get an object and fetch its member named foo
p->foo; // equivalent way to fetch the foo from the object to which p points
```

More concretely, we can rewrite the call to `same_isbn` as

具体地，可将 `same_isbn` 的调用重写为：

```
sp->same_isbn(item2); // equivalent to (*sp).same_isbn(item2)
```

Exercises Section 5.6

Exercise 5.18: Write a program that defines a `vector` of pointers to `strings`. Read the `vector`, printing each `string` and its corresponding size.

编写程序定义一个 `vector` 对象，其每个元素都是指向 `string` 类型的指针，读取该 `vector` 对象，输出每个 `string` 的内容及其相应的长度。

Exercise 5.19: Assuming that `iter` is a `vector<string>::iterator`, indicate which, if any, of the following expressions is legal. Explain the behavior of the legal expressions.

假设 `iter` 为 `vector<string>::iterator` 类型的变量，指出下面哪些表达式是合法的，并解释这些合法表达式的行为。

Section 5.6. The Arrow Operator

- (a) `*iter++;`
- (b) `(*iter)++;`
- (c) `*iter.empty()`
- (d) `iter->empty();`
- (e) `++*iter;`
- (f) `iter++->empty();`

Team LiB

◀ PREVIOUS NEXT ▶

5.7. The Conditional Operator

5.7. 条件操作符

The **conditional operator** is the only ternary operator in C++. It allows us to embed simple if-else tests inside an expression. The conditional operator has the following syntactic form

条件操作符是 C++ 中唯一的三元操作符，它允许将简单的 if-else 判断语句嵌入表达式中。条件操作符的语法格式为：

```
cond ? expr1 : expr2;
```

where *cond* is an expression that is used as a condition ([Section 1.4.1](#), p. 12). The operator executes by evaluating *cond*. If *cond* evaluates to 0, then the condition is `false`; any other value is `true`. *cond* is always evaluated. If it is `true`, then *expr1* is evaluated; otherwise, *expr2* is evaluated. Like the logical AND and OR (`&&` and `||`) operators, the conditional operator guarantees this order of evaluation for its operands. Only one of *expr1* or *expr2* is evaluated. The following program illustrates use of the conditional operator:

其中，*cond* 是一个条件判断表达式（[第 1.4.1 节](#)）。条件操作符首先计算 *cond* 的值，如果 *cond* 的值为 0，则条件为 `false`；如果 *cond* 非 0，则条件为 `true`。无论如何，*cond* 总是要被计算的。然后，条件为 `true` 时计算 *expr1*，否则计算 *expr2*。和逻辑与、逻辑或（`&&` 和 `||`）操作符一样，条件操作符保证了上述操作数的求解次序。*expr1* 和 *expr2* 中只有一个表达式被计算。下面的程序说明了条件操作符的用法：

```
int i = 10, j = 20, k = 30;
// if i > j then maxVal = i else maxVal = j
int maxVal = i > j ? i : j;
```

Avoid Deep Nesting of the Conditional Operator

避免条件操作符的深度嵌套

We could use a set of nested conditional expressions to set `max` to the largest of three variables:

可以使用一组嵌套的条件操作符求出三个变量的最大值，并将最大值赋给 `max`：

```
int max = i > j
    ? i > k ? i : k
    : j > k ? j : k;
```

We could do the equivalent comparison in the following longer but simpler way:

我们也可以用下面更长却更简单的比较语句实现相同的功能：

```
int max = i;
if (j > max)
    max = j;
if (k > max)
    max = k;
```

Using a Conditional Operator in an Output Expression

在输出表达式中使用条件操作符

The conditional operator has fairly low precedence. When we embed a conditional expression in a larger expression, we usually must parenthesize the conditional subexpression. For example, the conditional operator is often used to print one or another value, depending on the result of a condition. Incompletely parenthesized uses of the conditional operator in an output expression can have surprising results:

条件操作符的优先级相当低。当我们要在更大的表达式中嵌入条件表达式时，通常必须用圆括号把条件表达式括起来。例如，经常使用条件操作符根据一定的条件输出一个或另一个值，在输出表达式中，如果不严格使用圆括号将条件操作符括起来，将会得到意外的结果：

```
cout << (i < j ? i : j); // ok: prints larger of i and j
cout << (i < j) ? i : j; // prints 1 or 0!
cout << i < j ? i : j; // error: compares cout to int
```

The second expression is the most interesting: It treats the comparison between `i` and `j` as the operand to the `<<` operator. The value 1 or 0 is printed, depending on whether `i < j` is true or false. The `<<` operator returns `cout`, which is tested as the condition for the conditional operator. That is, the second expression is equivalent to

Section 5.7. The Conditional Operator

第二个表达式比较有趣：它将*i*和*j*的比较结果视为 `<<` 操作符的操作数，输出 1 或 0。 `<<` 操作符返回 `cout` 值，然后将返回结果作为条件操作符的判断条件。也就是，第二个表达式等效于：

```
cout << (i < j); // prints 1 or 0
cout ? i : j;   // test cout and then evaluate i or j
                // depending on whether cout evaluates to true or false
```

Exercises Section 5.7

Exercise Write a program to prompt the user for a pair of numbers and report which is smaller.

5.20:

编写程序提示用户输入两个数，然后报告哪个数比较小。

Exercise Write a program to process the elements of a `vector<int>`. Replace each element with an odd value by twice that value.

5.21:

编写程序处理 `vector<int>` 对象的元素：将每个奇数值元素用该值的两倍替换。

5.8. The `sizeof` Operator

5.8. `sizeof` 操作符

The `sizeof` operator returns a value of type `size_t` (Section 3.5.2, p. 104) that is the size, in bytes (Section 2.1, p. 35), of an object or type name. The result of `sizeof` expression is a compile-time constant. The `sizeof` operator takes one of the following forms:

`sizeof` 操作符的作用是返回一个对象或类型名的长度，返回值的类型为 `size_t` (第 3.5.2 节)，长度的单位是字节 (第 2.1 节)。`size_t` 表达式的结果是编译时常量，该操作符有以下三种语法形式：

```
sizeof (type name);
sizeof (expr);
sizeof expr;
```

Applying `sizeof` to an `expr` returns the size of the result type of that expression:

将 `sizeof` 应用在表达式 `expr` 上，将获得该表达式的结果的类型长度：

```
Sales_item item, *p;
// three ways to obtain size required to hold an object of type Sales_item
sizeof(Sales_item); // size required to hold an object of type Sales_item
sizeof item; // size of item's type, e.g., sizeof(Sales_item)
sizeof *p; // size of type to which p points, e.g., sizeof(Sales_item)
```

Evaluating `sizeof expr` does not evaluate the expression. In particular, in `sizeof *p`, the pointer `p` may hold an invalid address, because `p` is not dereferenced.

将 `sizeof` 用于 `expr` 时，并没有计算表达式 `expr` 的值。特别是在 `sizeof *p` 中，指针 `p` 可以持有一个无效地址，因为不需要对 `p` 做解引用操作。

The result of applying `sizeof` depends in part on the type involved:

使用 `sizeof` 的结果部分地依赖所涉及的类型：

- `sizeof char` or an expression of type `char` is guaranteed to be 1
对 `char` 类型或值为 `char` 类型的表达式做 `sizeof` 操作保证得 1。
- `sizeof` a reference type returns the size of the memory necessary to contain an object of the referenced type
对引用类型做 `sizeof` 操作将返回存放此引用类型对象所需的内存空间大小。
- `sizeof` a pointer returns the size needed hold a pointer; to obtain the size of the object to which the pointer points, the pointer must be dereferenced
对指针做 `sizeof` 操作将返回存放指针所需的内存大小；注意，如果要获取该指针所指向对象的大小，则必须对指针进行引用。
- `sizeof` an array is equivalent to taking the `sizeof` the element type times the number of elements in the array
对数组做 `sizeof` 操作等效于将对其元素类型做 `sizeof` 操作的结果乘上数组元素的个数。

Because `sizeof` returns the size of the entire array, we can determine the number of elements by dividing the `sizeof` the array by the `sizeof` an element:

因为 `sizeof` 返回整个数组在内存中的存储长度，所以用 `sizeof` 数组的结果除以 `sizeof` 其元素类型的结果，即可求出数组元素的个数：

```
// sizeof(ia)/sizeof(*ia) returns the number of elements in ia
int sz = sizeof(ia)/sizeof(*ia);
```

Exercises Section 5.8

Exercise Write a program to print the size of each of the built-in types.

5.22:

编写程序输出的每种内置类型的长度。

Exercise Predict the output of the following program and explain your reasoning. Now run the program.

5.23:

Is the output what you expected? If not, figure out why.

Section 5.8. The sizeof Operator

预测下列程序的输出是，并解释你的理由。然后运行该程序，输出的结果和你的预测的一样吗？如果不一样，为什么？

```
int x[10];    int *p = x;
cout << sizeof(x)/sizeof(*x) << endl;
cout << sizeof(p)/sizeof(*p) << endl;
```

Team LiB

◀ PREVIOUS NEXT ▶

5.9. Comma Operator

5.9. 逗号操作符

A **comma expression** is a series of expressions separated by commas. The expressions are evaluated from left to right. The result of a comma expression is the value of the rightmost expression. The result is an lvalue if the rightmost operand is an lvalue. One common use for the comma operator is in a `for` loop.

逗号表达式是一组由逗号分隔的表达式，这些表达式从左向右计算。逗号表达式的结果是其最右边表达式的值。如果最右边的操作数是左值，则逗号表达式的值也是左值。此类表达式通常用于`for`循环：

```
int cnt = ivect.size();
// add elements from size... 1 to ivect
for(vector<int>::size_type ix = 0;
    ix != ivect.size(); ++ix, --cnt)
    ivect[ix] = cnt;
```

This loop increments `ix` and decrements `cnt` in the expression in the `for` header. Both `ix` and `cnt` are changed on each trip through the loop. As long as the test of `ix` succeeds, we reset the next element to the current value of `cnt`.

上述的 `for` 语句在循环表达式中使 `ix` 自增 1 而 `cnt` 自减 1。每次循环均要修改 `ix` 和 `cnt` 的值。当检验 `ix` 的条件判断成立时，程序将下一个元素重新设置为 `cnt` 的当前值。

Exercises Section 5.9

Exercise 5.24: The program in this section is similar to the program on page [163](#) that added elements to a `vector`. Both programs decremented a counter to generate the element values. In this program we used the prefix decrement and the earlier one used postfix. Explain why we used prefix in one and postfix in the other.

本节的程序与[第 5.5 节](#)在 `vector` 对象中添加元素的程序类似。两段程序都使用递减的计数器生成元素的值。本程序中，我们使用了前自减操作，而[第 5.5 节](#)的程序则使用了后自减操作。解释为什么一段程序中使用前自减操作而在另一段程序中使用后自减操作。

5.10. Evaluating Compound Expressions

5.10. 复合表达式的求值

An expression with two or more operators is a **compound expression**. In a compound expression, the way in which the operands are grouped to the operators may determine the result of the overall expression. If the operands group in one way, the result differs from what it would be if they grouped another way.

含有两个或更多操作符的表达式称为**复合表达式**。在复合表达式中，操作数和操作符的结合方式决定了整个表达式的值。表达式的结果会因为操作符和操作数的分组结合方式的不同而不同。

Precedence and associativity determine how the operands are grouped. That is, precedence and associativity determine which part of the expression is the operand for each of the operators in the expression. Programmers can override these rules by parenthesizing compound expressions to force a particular grouping.

操作数的分组结合方式取决于操作符的优先级和结合性。也就是说，优先级和结合性决定了表达式的哪个部分用作哪个操作符的操作数。如果程序员不想考虑这些规则，可以在复合表达式中使用圆括号强制实现某个特殊的分组。



Precedence specifies how the operands are grouped. It says nothing about the order in which the operands are evaluated. In most cases, operands may be evaluated in whatever order is convenient.

优先级规定的是操作数的结合方式，但并没有说明操作数的计算顺序。在大多数情况下，操作数一般以最方便的次序求解。

5.10.1. Precedence

5.10.1. 优先级

The value of an expression depends on how the subexpressions are grouped. For example, in the following expression, a purely left-to-right evaluation yields 20:

表达式的值取决于其子表达式如何分组。例如，下面的表达式，如果纯粹从左向右计算，结果为 20：

```
6 + 3 * 4 / 2 + 2;
```

Other imaginable results include 9, 14, and 36. In C++, the result is 14.

想像中其他可能的结果包括 9、14 和 36。在 C++ 中，该表达式的值应为 14。

Multiplication and division have higher precedence than addition. Their operands are bound to the operator in preference to the operands to addition. Multiplication and division have the same precedence as each other. Operators also have associativity, which determines how operators at the same precedence level are grouped. The arithmetic operators are left associative, which means they group left to right. We now can see that our expression is equivalent to

乘法和除法的优先级高于加法操作，于是它们的操作数先于加法操作的操作数计算。但乘法和除法的优先级相同。当操作符的优先级相同时，由其结合性决定求解次序。算术操作具有左结合性，这意味着它们从左向右结合。因此上面表达式等效于：

```
int temp = 3 * 4;           // 12
int temp2 = temp / 2;        // 6
int temp3 = temp2 + 6;        // 12
int result = temp3 + 2;       // 14
```

Parentheses Override Precedence

圆括号凌驾于优先级之上

We can override precedence with parentheses. Parenthesized expressions are evaluated by treating each parenthesized subexpression as a unit and otherwise applying the normal precedence rules. For example, we can use parentheses on our initial expression to force the evaluation to result in any of the four possible values:

Section 5.10. Evaluating Compound Expressions

我们可使用圆括号推翻优先级的限制。使用圆括号的表达式将用圆括号括起来的子表达式视为独立单元先计算，其他部分则以普通的优先级规则处理。例如，下面的程序在前述表达式上添加圆括号，强行更改其操作次序，可能得到四种结果：

```
// parentheses on this expression match default precedence/associativity
cout << ((6 + ((3 * 4) / 2)) + 2) << endl; // prints 14
// parentheses result in alternative groupings
cout << (6 + 3) * (4 / 2 + 2) << endl;      // prints 36
cout << ((6 + 3) * 4) / 2 + 2 << endl;      // prints 20
cout << 6 + 3 * 4 / (2 + 2) << endl;        // prints 9
```

We have already seen examples where precedence rules affect the correctness of our programs. For example, consider the expression described in the "Advice" box on page [164](#):

我们已经通过前面的例子了解了优先级规则如何影响程序的正确性。例如，考虑[第 5.5 节](#)第二个建议框中描述的表达式：

```
*iter++;
```

Precedence says that `++` has higher precedence than `*`. That means that `iter++` is grouped first. The operand of `*`, therefore, is the result of applying the increment operator to `iter`. If we wanted to increment the value that `iter` denotes, we'd have to use parentheses to force our intention:

其中，`++` 的优先级高于`*`操作符，这就意味着 `iter++` 先结合。而操作符 `*` 的操作数是 `iter` 做了自增操作后的结果。如果我们希望对 `iter` 所指向的值做自增操作，则必须使用圆括号强制实现我们的目的：

```
(*iter)++; // increment value to which iter refers and yield unincremented value
```

The parentheses specify that the operand of `*` is `iter`. The expression now uses `*iter` as the operand to `++`.

圆括号指明操作符 `*` 的操作数是 `iter`，然后表达式以 `*iter` 作为 `++` 操作符的操作数。

As another example, recall the condition in the [while](#) on page [161](#):

另一个例子，回顾一下[第 5.4.2 节](#)中的 `while` 循环条件：

```
while ((i = get_value()) != 42) {
```

The parentheses around the assignment were necessary to implement the desired operation, which was to assign to `i` the value returned from `get_value` and then test that value to see whether it was 42. Had we failed to parenthesize the assignment, the effect would be to test the return value to see whether it was 42. The true or false value of that test would then be assigned to `i`, meaning that `i` would either be 1 or 0.

赋值操作上的圆括号是必需的，这样才能实现预期的操作：将 `get_value` 的返回值赋给 `i`，然后检查刚才赋值的结果是否为42。如果赋值操作上没有加圆括号，结果将是先判断 `get_value` 的返回值是否为 42，然后将判断结果 `true` 或 `false` 值赋给 `i`，这意味着 `i` 的值只能是 1 或 0。

5.10.2. Associativity

5.10.2. 结合性

Associativity specifies how to group operators at the same precedence level. We have also seen cases where associativity matters. As one example, the assignment operator is right associative. This fact allows concatenated assignments:

结合性规定了具有相同优先级的操作符如何分组。我们已经遇到过涉及结合性的例子。其中之一使用了赋值操作的右结合性，这个特性允许将多个赋值操作串接起来：

```
ival = jval = kval = lval      // right associative
(ival = (jval = (kval = lval))) // equivalent, parenthesized version
```

This expression first assigns `lval` to `kval`, then the result of that to `jval`, and finally the result of that to `ival`.

该表达式首先将 `lval` 赋给 `kval`，然后将 `kval` 的值赋给 `jval`，最后将 `jval` 的值再赋给 `ival`。

The arithmetic operators, on the other hand, are left associative. The expression

另一方面，算术操作符为左结合。表达式

```
ival * jval / kval * lval      // left associative
(((ival * jval) / kval) * lval) // equivalent, parenthesized version
```

multiples `ival` and `jval`, then divides that result by `kval`, and finally multiplies the result of the division by `lval`.

先对 `ival` 和 `jval` 做乘法操作，然后乘积除以 `kval`，最后再将其商与 `lval` 相乘。

[Table 5.4](#) presents the full set of operators ordered by precedence. The table is organized into segments separated by double lines. Operators in each segment have the same precedence, and have higher precedence than operators in sub-sequent segments. For example, the prefix increment and dereference operators share the same precedence and have higher precedence than the arithmetic or relational operators. We have seen most of these operators, although a few will not be defined until later chapters.

表 5.4 按照优先级顺序列出了 C++ 的全部操作符。该表以双横线分割成不同的段，每段内各个操作符的优先级相同，且都高于后面各段中的操作符。例如，前自增操作符和解引用操作符的优先级相同，它们的优先级都比算术操作符或关系操作符高。此表中大部分操作符已经介绍过，而少数未介绍的操作符将在后续章节中学习。

Table 5.4. Operator Precedence**表 5.4. 操作符的优先级**

Associativity and Operator	Function 功能	Use 用法	See Page 参见页码
操作符及其结合性	功能	用法	参见页码
L ::	global scope (全局作用域)	:: name	p. 450
L ::	class scope (类作用域)	class :: name	p. 85
L ::	namespace scope (名字空间作用域)	namespace :: name	p. 78
L .	member selectors (成员选择)	object . member	p. 25
L ->	member selectors (成员选择)	pointer -> member	p. 164
L []	subscript (下标)	variable [expr]	p. 113
L ()	function call (函数调用)	name (expr_list)	p. 25
L ()	type construction (类型构造)	type (expr_list)	p. 460
R ++	postfix increment (后自增操作)	lvalue++	p. 162
R --	postfix decrement (后自减操作)	lvalue--	p. 162
R typeid	type ID (类型 ID)	typeid (type)	p. 775
R typeid	run-time type ID (运行时类型 ID)	typeid (expr)	p. 775
R explicit cast (显式强制类型转换)	type conversion (类型转换)	cast_name <type>(expr)	p. 183
R sizeof	size of object (对象的大小)	sizeof expr	p. 167
R sizeof	size of type (类型的大小)	sizeof(type)	p. 167
R ++	prefix increment (前自增操作)	++ lvalue	p. 162
R --	prefix decrement (前自减操作)	-- lvalue	p. 162
R ~	bitwise NOT (位求反)	~expr	p. 154
R !	logical NOT (逻辑非)	!expr	p. 152
R -	unary minus (一元负号)	-expr	p. 150
R +	unary plus (一元正号)	+expr	p. 150
R *	dereference (解引用)	*expr	p. 119
R &	address-of (取地址)	&expr	p. 115
R ()	type conversion (类型转换)	(type) expr	p. 186
R new	allocate object (创建对象)	new type	p. 174
R delete	deallocate object (释放对象)	delete expr	p. 176
R delete[]	deallocate array (释放数组)	delete[] expr	p. 137
L ->*	ptr to member select (指向成员操作的指针)	ptr ->* ptr_to_member	p. 783
L .*	ptr to member select (指向成员操作的指针)	obj .*ptr_to_member	p. 783
L *	multiply (乘法)	expr * expr	p. 149
L /	divide (除法)	expr / expr	p. 149
L %	modulo (remainder) (求模 (求余))	expr % expr	p. 149

Section 5.10. Evaluating Compound Expressions

L	<code>+</code>	add (加法)	<code>expr + expr</code>	p. 149
L	<code>-</code>	subtract (减法)	<code>expr - expr</code>	p. 149
L	<code><<</code>	bitwise shift left (位左移)	<code>expr << expr</code>	p. 154
L	<code>>></code>	bitwise shift right (位右移)	<code>expr >> expr</code>	p. 154
L	<code><</code>	less than (小于)	<code>expr < expr</code>	p. 152
L	<code><=</code>	less than or equal (小于或等于)	<code>expr <= expr</code>	p. 152
L	<code>></code>	greater than (大于)	<code>expr > expr</code>	p. 152
L	<code>>=</code>	greater than or equal (大于或等于)	<code>expr >= expr</code>	p. 152
L	<code>==</code>	equality (相等)	<code>expr == expr</code>	p. 152
L	<code>!=</code>	inequality (不等)	<code>expr != expr</code>	p. 152
L	<code>&</code>	bitwise AND (位与)	<code>expr & expr</code>	p. 154
L	<code>^</code>	bitwise XOR ()	<code>expr ^ expr</code>	p. 154
L	<code> </code>	bitwise OR (位异或)	<code>expr expr</code>	p. 154
L	<code>&&</code>	logical AND (逻辑与)	<code>expr && expr</code>	p. 152
L	<code> </code>	logical OR (逻辑或)	<code>expr expr</code>	p. 152
R	<code>? :</code>	conditional (条件操作)	<code>expr ? expr : expr</code>	p. 165
R	<code>=</code>	assignment (赋值操作)	<code>lvalue = expr</code>	p. 159
R	<code>*=, /=, %=,</code>	compound assign (复合赋值操作)	<code>lvalue += expr, etc.</code>	p. 159
R	<code>+=, -=,</code>			p. 159
R	<code><<=, >>=,</code>			p. 159
R	<code>&=, =, ^=</code>			p. 159
R	<code>throw</code>	throw exception (抛出异常)	<code>throw expr</code>	p. 216
L	<code>,</code>	comma (逗号)	<code>expr , expr</code>	p. 168

Exercises Section 5.10.2

Exercise

5.25: Using [Table 5.4](#) (p. [170](#)), parenthesize the following expressions to indicate the order in which the operands are grouped:

根据表 5.4 的内容，在下列表达式中添加圆括号说明其操作数分组的顺序（即计算顺序）：

(a) `! ptr == ptr->next`
(b) `ch = buf[bp++] != '\n'`

Exercise

5.26: The expressions in the previous exercise evaluate in an order that is likely to be surprising. Parenthesize these expressions to evaluate in an order you imagine is intended.

习题 5.25 中的表达式的计算次序与你的意图不同，给它们加上圆括号使其以你所希望的操作次序求解。

Exercise

5.27: The following expression fails to compile due to operator precedence. Using [Table 5.4](#) (p. [170](#)), explain why it fails. How would you fix it?

由于操作符优先级的问题，下列表达式编译失败。请参照表 5.4 解释原因，应该如何改正？

```
string s = "word";
// add an 's' to the end, if the word doesn't already end in 's'
string pl = s + s[s.size() - 1] == 's' ? "" : "s" ;
```

5.10.3. Order of Evaluation

5.10.3. 求值顺序

In [Section 5.2](#) (p. 152) we saw that the `&&` and `||` operators specify the order in which their operands are evaluated: In both cases the right-hand operand is evaluated if and only if doing so might affect the truth value of the overall expression. Because we can rely on this property, we can write code such as

在第 5.2 节中，我们讨论了 `&&` 和 `||` 操作符计算其操作数的次序：当且仅当其右操作数确实影响了整个表达式的值时，才计算这两个操作符的右操作数。根据这个原则，可编写如下代码：

```
// iter only dereferenced if it isn't at end
while (iter != vec.end() && *iter != some_val)
```

The only other operators that guarantee the order in which operands are evaluated are the conditional (`?:`) and comma operators. In all other cases, the order is unspecified.

C++ 中，规定了操作数计算顺序的操作符还有条件 (`?:`) 和逗号操作符。除此之外，其他操作符并未指定其操作数的求值顺序。

For example, in the expression

例如，表达式

```
f1() * f2();
```

we know that both `f1` and `f2` must be called before the multiplication can be done. After all, their results are what is multiplied. However, we have no way to know whether `f1` will be called before `f2` or vice versa.

在做乘法操作之前，必须调用 `f1` 函数和 `f2` 函数，毕竟其调用结果要相乘。然而，我们却无法知道到底是先调用 `f1` 还是先调用 `f2`。



The order of operand evaluation often, perhaps even usually, doesn't matter. It can matter greatly, though, if the operands refer to and change the same objects.

其实，以什么次序求解操作数通常没有多大关系。只有当操作符的两个操作数涉及到同一个对象，并改变其值时，操作数的计算次序才会影响结果。

The order of operand evaluation matters if one subexpression changes the value of an operand used in another subexpression:

如果一个子表达式修改了另一个子表达式的操作数，则操作数的求解次序就变得相当重要：

```
// oops! language does not define order of evaluation
if (ia[index++] < ia[index])
```

The behavior of this expression is undefined. The problem is that the left- and right-hand operands to the `<` both use the variable `index`. However, the left-hand operand involves changing the value of that variable. Assuming `index` is zero, the compiler might evaluate this expression in one of the following two ways:

此表达式的行为没有明确定义。问题在于：`<` 操作符的左右操作数都使用了 `index` 变量，但是，左操作数更改了该变量的值。假设 `index` 初值为 0，编译器可以用下面两种方式之一求该表达式的值：

```
if (ia[0] < ia[0]) // execution if rhs is evaluated first
if (ia[0] < ia[1]) // execution if lhs is evaluated first
```

We can guess that the programmer intended that the left operand be evaluated, thereby incrementing `index`. If so, the comparison would be between `ia[0]` and `ia[1]`. The language, however, does not guarantee a left-to-right evaluation order. In fact, an expression like this is undefined. An implementation might evaluate the right-hand operand first, in which case `ia[0]` is compared to itself. Or the implementation might do something else entirely.

可以假设程序员希望先求左操作数的值，因此 `index` 的值加 1。如果是这样的话，比较 `ia[0]` 和 `ia[1]` 的值。然而，C++ 语言不能确保从左到右的计算次序。事实上，这类表达式的行为没有明确定义。一种实现可能是先计算右操作数，于是 `ia[0]` 与自己做比较，要不然就是做完全不同的操作。

Advice: Managing Compound Expressions

建议：复合表达式的处理

Beginning C and C++ programmers often have difficulties understanding order of evaluation and the rules of precedence and associativity. Misunderstanding how expressions and operands are evaluated is a rich source of bugs. Moreover, the resulting bugs are difficult to find because reading the program does not reveal the error unless the

programmer already understands the rules.

初学 C 和 C++ 的程序员一般很难理解求值顺序、优先级和结合性规则。误解表达式和操作数如何求解将导致大量的程序错误。此外，除非程序员已经完全理解了相关规则，否则这类错误很难发现，因为仅靠阅读程序是无法排除这些错误的。

Two rules of thumb can be helpful:

下面两个指导原则有助于处理复合表达式：

1. When in doubt, parenthesize expressions to force the grouping that the logic of your program requires.

如果有怀疑，则在表达式上按程序逻辑要求使用圆括号强制操作数的组合。

2. If you change the value of an operand, don't use that operand elsewhere in the same statement. If you need to use the changed value, then break the expression up into separate statements in which the operand is changed in one statement and then used in a subsequent statement.

如果要修改操作数的值，则不要在同一个语句的其他地方使用该操作数。如果必须使用改变的值，则把该表达式分割成两个独立语句：在一个语句中改变该操作数的值，再在下一个语句使用它。

An important exception to the second rule is that subexpressions that use the result of the subexpression that changes the operand are safe. For example, in `*++iter` the increment changes the value of `iter`, and the (changed) value of `iter` is then used as the operand to `*`. In this, and similar, expressions, order of evaluation of the operand isn't an issue. To evaluate the larger expression, the subexpression that changes the operand must first be evaluated. Such usage poses no problems and is quite common.

第二个规则有一个重要的例外：如果一个子表达式修改操作数的值，然后将该子表达式的结果用于另一个子表达式，这样则是安全的。例如，`*++iter` 表达式的自增操作修改了 `iter` 的值，然后将 `iter`（修改后）的值用作 `*` 操作符的操作数。对于这个表达式或其他类似的表达式，其操作数的计算次序无关紧要。而为了计算更复杂的表达式，改变操作数值的子表达式必须首先计算。这种方法很常用，不会产生什么问题。



Do not use an increment or decrement operator on the same object in more than two subexpressions of the same expression.

一个表达式里，不要在两个或更多的子表达式中对同一对象做自增或自减操作。

One safe and machine-independent way to rewrite the previous comparison of two array elements is

以一种安全而且独立于机器的方式重写上述比较两个数组元素的程序：

```
if (ia[index] < ia[index + 1]) {
    // do whatever
}
++index;
```

Now neither operand can affect the value of the other.

现在，两个操作数的值不会相互影响。

Exercises Section 5.10.3

Exercise

5.28:

With the exception of the logical AND and OR, the order of evaluation of the binary operators is left undefined to permit the compiler freedom to provide an optimal implementation. The trade-off is between an efficient implementation and a potential pitfall in the use of the language by the programmer. Do you consider that an acceptable trade-off? Why or why not?

除了逻辑与和逻辑或外，C++ 没有明确定义二元操作符的求解次序，编译器可自由地提供最佳的实现方式。只能在“实现效率”和程序语言使用中“潜在的缺陷”之间寻求平衡。你认为这可以接受吗？说出你的理由。

Exercise

5.29:

Given that `ptr` points to a class with an `int` member named `ival`, `vec` is a `vector` holding `int`s, and that `ival`, `jval`, and `kval` are also `int`s, explain the behavior of each of these expressions. Which, if any, are likely to be incorrect? Why? How might each be corrected?

假设 `ptr` 指向类类型对象，该类拥有一个名为 `ival` 的 `int` 型数据成员，`vec` 是保存 `int` 型元素的 `vector` 对象，而 `ival`、`jval` 和 `kval` 都是 `int` 型变量。请解释下列表达式的行为，并指出哪些（如果有的话）可能是不正确的，为什么？如何改正？

- | | |
|--|---|
| (a) <code>ptr->ival != 0</code> | (b) <code>ival != jval < kval</code> |
| (c) <code>ptr != 0 && *ptr++</code> | (d) <code>ival++ && ival</code> |
| (e) <code>vec[ival++] <= vec[ival]</code> | |

Team LiB

◀ PREVIOUS NEXT ▶

5.11. The `new` and `delete` Expressions

5.11. `new` 和 `delete` 表达式

In [Section 4.3.1](#) (p. 134) we saw how to use `new` and `delete` expressions to dynamically allocate and free arrays. We can also use `new` and `delete` to dynamically allocate and free single objects.

[第 4.3.1 节](#)介绍了如何使用 `new` 和 `delete` 表达式动态创建和释放数组，这两种表达式也可用于动态创建和释放单个对象。

When we define a variable, we specify a type and a name. When we dynamically allocate an object, we specify a type but do not name the object. Instead, the `new` expression returns a pointer to the newly allocated object; we use that pointer to access the object:

```
int i;           // named, uninitialized int variable
int *pi = new int; // pi points to dynamically allocated,
                  // unnamed, uninitialized int
```

This `new` expression allocates one object of type `int` from the free store and returns the address of that object. We use that address to initialize the pointer `pi`.

这个 `new` 表达式在自由存储区中分配创建了一个整型对象，并返回此对象的地址，并用该地址初始化指针 `pi`。

Initializing Dynamically Allocated Objects

动态创建对象的初始化

Dynamically allocated objects may be initialized, in much the same way as we initialize variables:

动态创建的对象可用初始化变量的方式实现初始化：

```
int i(1024);           // value of i is 1024
int *pi = new int(1024); // object to which pi points is 1024
string s(10, '9');      // value of s is "9999999999"
string *ps = new string(10, '9'); // *ps is "9999999999"
```

We must use the direct-initialization syntax ([Section 2.3.3](#), p. 48) to initialize dynamically allocated objects. When an initializer is present, the `new` expression allocates the required memory and initializes that memory using the given initializer(s). In the case of `pi`, the newly allocated object is initialized to 1024. The object pointed to by `ps` is initialized to a string of 10 nines.

C++ 使用直接初始化（direct-initialization）语法规则（[第 2.3.3 节](#)）初始化动态创建的对象。如果提供了初值，`new` 表达式分配到所需要的内存后，用给定的初值初始化该内存空间。在本例中，`pi` 所指向的新创建对象将被初始化为 1024，而 `ps` 所指向的对象则初始化为十个9的字符串。

Default Initialization of Dynamically Allocated Objects

动态创建对象的默认初始化

If we do not explicitly state an initializer, then a dynamically allocated object is initialized in the same way as is a variable that is defined inside a function. ([Section 2.3.4](#), p. 50) If the object is of class type, it is initialized using the default constructor for the type; if it is of built-in type, it is uninitialized.

如果不提供显式初始化，动态创建的对象与在函数内定义的变量初始化方式相同（[第 2.3.4 节](#)）。对于类类型的对象，用该类的默认构造函数初始化；而内置类型的对象则无初始化。

```
string *ps = new string; // initialized to empty string
int *pi = new int;       // pi points to an uninitialized int
```

As usual, it is undefined to use the value associated with an uninitialized object in any way other than to assign a good value to it.

通常，除了对其赋值之外，对未初始化的对象所关联的值的任何使用都是没有定义的。



Just as we (almost) always initialize the objects we define as variables, it is (almost) always a good idea to initialize dynamically allocated objects.

正如我们（几乎）总是要初始化定义为变量的对象一样，在动态创建对象时，（几乎）总是对它做初始化也是一个好办法。

We can also value-initialize ([Section 3.3.1](#), p. 92) a dynamically allocated object:

同样也可对动态创建的对象做值初始化 (value-initialize) ([第 3.3.1 节](#)) :

```
string *ps = new string(); // initialized to empty string
int *pi = new int(); // pi points to an int value-initialized to 0
cls *pc = new cls(); // pc points to a value-initialized object of type cls
```

We indicate that we want to value-initialize the newly allocated object by following the type name by a pair of empty parentheses. The empty parentheses signal that we want initialization but are not supplying a specific initial value. In the case of class types (such as `string`) that define their own constructors, requesting value-initialization is of no consequence: The object is initialized by running the default constructor whether we leave it apparently uninitialized or ask for value-initialization. In the case of built-in types or types that do not define any constructors, the difference is significant:

以上表明程序员想通过在类型名后面使用一对内容为空的圆括号对动态创建的对象做值初始化。内容为空的圆括号表示虽然要做初始化，但实际上并未提供特定的初值。对于提供了默认构造函数的类类型（例如 `string`），没有必要对其对象进行值初始化：无论程序是明确地不初始化还是要求进行值初始化，都会自动调用其默认构造函数初始化该对象。而对于内置类型或没有定义默认构造函数的类型，采用不同初始化方式则有显著的差别：

```
int *pi = new int; // pi points to an uninitialized int
int *pi = new int(); // pi points to an int value-initialized to 0
```

In the first case, the `int` is uninitialized; in the second case, the `int` is initialized to zero.

第一个语句的 `int` 型变量没有初始化，而第二个语句的 `int` 型变量则被初始化为0。



The `()` syntax for value initialization must follow a type name, not a variable. As we'll see in [Section 7.4](#) (p. 251)

值初始化的 `()` 语法必须置于类型名后面，而不是变量后。正如我们将要学习的[第 7.4 节](#)的例子：

```
int x(); // does not value initialize x
```

declares a function named `x` with no arguments that returns an `int`.

这个语句声明了一个名为 `x`、没有参数而且返回 `int` 值的函数。

Memory Exhaustion

耗尽内存

Although modern machines tend to have huge memory capacity, it is always possible that the free store will be exhausted. If the program uses all of available memory, then it is possible for a `new` expression to fail. If the `new` expression cannot acquire the requested memory, it throws an exception named `bad_alloc`. We'll look at how exceptions are thrown in [Section 6.13](#) (p. 215).

尽管现代机器的内存容量越来越大，但是自由存储区总有可能被耗尽。如果程序用完了所有可用的内存，`new` 表达式就有可能失败。如果 `new` 表达式无法获取需要的内存空间，系统将抛出名为 `bad_alloc` 的异常。我们将在[第 6.13 节](#)介绍如何抛出异常。

Destroying Dynamically Allocated Objects

撤销动态创建的对象

Section 5.11. The new and delete Expressions

When our use of the object is complete, we must *explicitly* return the object's memory to the free store. We do so by applying the `delete` expression to a pointer that addresses the object we want to release.

动态创建的对象用完后，程序员必须显式地将该对象占用的内存返回给自由存储区。C++ 提供了 `delete` 表达式释放指针所指向的地址空间。

```
delete pi;
```

frees the memory associated with the `int` object addressed by `pi`.

该命令释放 `pi` 指向的 `int` 型对象所占用的内存空间。



It is illegal to apply `delete` to a pointer that addresses memory that was not allocated by `new`.

如果指针指向不是用 `new` 分配的内存地址，则在该指针上使用 `delete` 是不合法的。

The effect of deleting a pointer that addresses memory that was not allocated by `new` is undefined. The following are examples of safe and unsafe [delete expressions](#):

C++ 没有明确定义如何释放指向不是用 `new` 分配的内存地址的指针。下面提供了一些安全的和不安全的 [delete expressions](#) 表达式。

```
int i;
int *pi = &i;
string str = "dwarves";
double *pd = new double(33);
delete str; // error: str is not a dynamic object
delete pi; // error: pi refers to a local
delete pd; // ok
```

It is worth noting that the compiler might refuse to compile the `delete` of `str`. The compiler knows that `str` is not a pointer and so can detect this error at compile-time. The second error is more insidious: In general, compilers cannot tell what kind of object a pointer addresses. Most compilers will accept this code, even though it is in error.

值得注意的是：编译器可能会拒绝编译 `str` 的 `delete` 语句。编译器知道 `str` 并不是一个指针，因此会在编译时就能检查出这个错误。第二个错误则比较隐蔽：通常来说，编译器不能断定一个指针指向什么类型的对象，因此尽管这个语句是错误的，但在大部分编译器上仍能通过。

`delete` of a Zero-Valued Pointer

零值指针的删除

It is legal to `delete` a pointer whose value is zero; doing so has no effect:

如果指针的值为 0，则在其上做 `delete` 操作是合法的，但这样做没有任何意义：

```
int *ip = 0;
delete ip; // ok: always ok to delete a pointer that is equal to 0
```

The language guarantees that deleting a pointer that is equal to zero is safe.

C++ 保证：删除 0 值的指针是安全的。

Resetting the Value of a Pointer after a `delete`

在 `delete` 之后，重设指针的值

When we write

执行语句

```
delete p;
```

`p` becomes undefined. Although `p` is undefined, on many machines, `p` still contains the address of the object to which it pointed. However, the memory to which `p` points was freed, so `p` is no longer valid.

后，`p` 变成没有定义。在很多机器上，尽管 `p` 没有定义，但仍然存放了它之前所指向对象的地址，然而 `p` 所指向的内存已经被释放，因此 `p` 不再有效。

After deleting a pointer, the pointer becomes what is referred to as a [dangling pointer](#). A dangling pointer is one that refers to memory that once held an object but does so no longer. A dangling pointer can be the source of program errors that are difficult to detect.

删除指针后，该指针变成[悬垂指针](#)。悬垂指针指向曾经存放对象的内存，但该对象已经不再存在了。悬垂指针往往导致程序错误，而且很难检测出来。



Setting the pointer to 0 after the object it refers to has been deleted makes it clear that the pointer points to no object.

一旦删除了指针所指向的对象，立即将指针置为 0，这样就非常清楚地表明指针不再指向任何对象。

Dynamic Allocation and Deallocation of `const` Objects

`const` 对象的动态分配和回收

It is legal to dynamically create `const` objects:

C++ 允许动态创建 `const` 对象:

```
// allocate and initialize a const object
const int *pci = new const int(1024);
```

Like any `const`, a dynamically created `const` must be initialized when it is created and once initialized cannot be changed. The value returned from this `new expression` is a pointer to `const int`. Like the address of any other `const` object, the return from a `new` that allocates a `const` object may only be assigned to a pointer to `const`.

与其他常量一样，动态创建的 `const` 对象必须在创建时初始化，并且一经初始化，其值就不能再修改。上述 `new 表达式` 返回指向 `int` 型 `const` 对象的指针。与其他 `const` 对象的地址一样，由于 `new` 返回的地址上存放的是 `const` 对象，因此该地址只能赋给指向 `const` 的指针。

A `const` dynamic object of a class type that defines a default constructor may be initialized implicitly:

对于类类型的 `const` 动态对象，如果该类提供了默认的构造函数，则此对象可隐式初始化:

```
// allocate default initialized const empty string
const string *pcs = new const string;
```

This `new` expression does not explicitly initialize the object pointed to by `pcs`. Instead, the object to which `pcs` points is implicitly initialized to the empty `string`. Objects of built-in type or of a class type that does not provide a default constructor must be explicitly initialized.

`new` 表达式没有显式初始化 `pcs` 所指向的对象，而是隐式地将 `pcs` 所指向的对象初始化为空的 `string` 对象。内置类型对象或未提供默认构造函数的类类型对象必须显式初始化。

Caution: Managing Dynamic Memory Is Error-Prone

警告：动态内存的管理容易出错

The following three common program errors are associated with dynamic memory allocation:

下面三种常见的程序错误都与动态内存分配相关:

1. Failing to `delete` a pointer to dynamically allocated memory, thus preventing the memory from being returned to the free store. Failure to delete dynamically allocated memory is spoken of as a "memory leak." Testing for memory leaks is difficult because they often do not appear until the application is run for a test period long enough to actually exhaust memory.

删除 (`delete`) 指向动态分配内存的指针失败，因而无法将该块内存返还给自由存储区。删除动态分配内存失败称为“内存泄漏 (memory leak)”。内存泄漏很难发现，一般需等应用程序运行了一段时间后，耗尽了所有内存空间时，内存泄漏才会显露出来。

2. Reading or writing to the object after it has been deleted. This error can sometimes be detected by setting the pointer to 0 after deleting the object to which the pointer had pointed.

读写已删除的对象。如果删除指针所指向的对象之后，将指针置为 0 值，则比较容易检测出这类错误。

3. Applying a `delete` expression to the same memory location twice. This error can happen when two pointers address the same dynamically allocated object. If `delete` is applied to one of the pointers, then the object's memory is returned to the free store. If we subsequently `delete` the second pointer, then the free store may be corrupted.

对同一个内存空间使用两次 `delete` 表达式。当两个指针指向同一个动态创建的对象，删除时就会发生错误。如果在其中一个指针上做 `delete` 运

算，将该对象的内存空间返还给自由存储区，然后接着 `delete` 第二个指针，此时则自由存储区可能会被破坏。

These kinds of errors in manipulating dynamically allocated memory are considerably easier to make than they are to track down and fix.

操纵动态分配的内存时，很容易发生上述错误，但这些错误却难以跟踪和修正。

Deleting a `const` Object

删除 `const` 对象

Although the value of a `const` object cannot be modified, the object itself can be destroyed. As with any other dynamic object, a `const` dynamic object is freed by deleting a pointer that points to it:

尽管程序员不能改变 `const` 对象的值，但可撤销对象本身。如同其他动态对象一样，`const` 动态对象也是使用删除指针来释放的：

```
delete pci; // ok: deletes a const object
```

Even though the operand of the delete expression is a pointer to `const int`, the delete expression is valid and causes the memory to which `pci` refers to be deallocated.

即使 `delete` 表达式的操作数是指向 `int` 型 `const` 对象的指针，该语句同样有效地回收 `pci` 所指向的内容。

Exercises Section 5.11

Exercise 5.30: Which of the following, if any, are illegal or in error?

下列语句哪些（如果有的话）是非法的或错误的？

- (a) `vector<string> svec(10);`
- (b) `vector<string> *pvec1 = new vector<string>(10);`
- (c) `vector<string> **pvec2 = new vector<string>[10];`
- (d) `vector<string> *pv1 = &svec;`
- (e) `vector<string> *pv2 = pvec1;`
- (f) `delete svec;`
- (g) `delete pvec1;`
- (h) `delete [] pvec2;`
- (i) `delete pv1;`
- (j) `delete pv2;`

5.12. Type Conversions

5.12. 类型转换

The type of the operand(s) determine whether an expression is legal and, if the expression is legal, determines the meaning of the expression. However, in C++ some types are related to one another. When two types are related, we can use an object or value of one type where an operand of the related type is expected. Two types are related if there is a [conversion](#) between them.

表达式是否合法取决于操作数的类型，而且合法的表达式其含义也由其操作数类型决定。但是，在 C++ 中，某些类型之间存在相关的依赖关系。若两种类型相关，则可在需要某种类型的操作数位置上，使用该类型的相关类型对象或值。如果两个类型之间可以相互[转换](#)，则称这两个类型相关。

As an example, consider

考虑下列例子：

```
int ival = 0;
ival = 3.541 + 3; // typically compiles with a warning
```

which assigns 6 to `ival`.

`ival` 的值为 6。

The operands to the addition operator are values of two different types: `3.541` is a literal of type `double`, and `3` is a literal of type `int`. Rather than attempt to add values of the two different types, C++ defines a set of conversions to transform the operands to a common type before performing the arithmetic. These conversions are carried out automatically by the compiler without programmer intervention and sometimes without programmer knowledge. For that reason, they are referred to as [implicit type conversions](#).

首先做加法操作，其操作数是两个不同类型的值：`3.541` 是 `double` 型的字面值常量，而 `3` 则是 `int` 型的字面值常量。C++ 并不是把两个不同类型的值直接加在一起，而是提供了一组转换规则，以便在执行算术操作之前，将两个操作数转换为同一种数据类型。这些转换规则由编译器自动执行，无需程序员介入——有时甚至不需要程序员了解。因此，它们也被称为隐式类型转换。

The built-in conversions among the arithmetic types are defined to preserve precision, if possible. Most often, if an expression has both integral and floating-point values, the integer is converted to floating-point. In this addition, the integer value `3` is converted to `double`. Floating-point addition is performed and the result, `6.541`, is of type `double`.

C++ 定义了算术类型之间的内置转换以尽可能防止精度损失。通常，如果表达式的操作数分别为整型和浮点型，则整型的操作数被转换为浮点型。本例中，整数`3`被转换为 `double` 类型，然后执行浮点类型的加法操作，得 `double` 类型的结果 `6.541`。

The next step is to assign that `double` value to `ival`, which is an `int`. In the case of assignment, the type of the left-hand operand dominates, because it is not possible to change the type of the object on the left-hand side. When the left- and right-hand types of an assignment differ, the right-hand side is converted to the type of the left-hand side. Here the `double` is converted to `int`. Converting a `double` to an `int` truncates the value; the decimal portion is discarded. `6.541` becomes `6`, which is the value assigned to `ival`. Because the conversion of a `double` to `int` may result in a loss of precision, most compilers issue a warning. For example, the compiler we used to check the examples in this book warns us:

下一步是将 `double` 型的值赋给 `int` 型变量 `ival`。在赋值操作中，因为不可能更改左操作数对象的类型，因此左操作数的类型占主导地位。如果赋值操作的左右操作数类型不相同，则右操作数会被转换为左边的类型。本例中，`double` 型的加法结果转换为 `int` 型。`double` 向 `int` 的转换自动按截尾形式进行，小数部分被舍弃。于是 `6.541` 变成 `6`，然后赋给 `ival`。因为从 `double` 到 `int` 的转换会导致精度损失，因此大多数编译器会给出警告。例如，本书所用的测试例程的编译器给出如下警告：

```
warning: assignment to 'int' from 'double'
```

To understand [implicit conversions](#), we need to know when they occur and what conversions are possible.

为了理解[隐式类型转换](#)，我们需要知道它们在什么时候发生，以及可能出现什么类型的转换。

5.12.1. When Implicit Type Conversions Occur

5.12.1. 何时发生隐式类型转换

The compiler applies conversions for both built-in and class type objects as necessary. Implicit type conversions take place in the following situations:

编译器在必要时将类型转换规则应用到内置类型和类类型的对象上。在下列情况下，将发生隐式类型转换：

- In expressions with operands of mixed types, the types are converted to a common type:

Section 5.12. Type Conversions

在混合类型的表达式中，其操作数被转换为相同的类型：

```
int ival;
double dval;
ival >= dval // ival converted to double
```

- An expression used as a condition is converted to `bool`:

用作条件的表达式被转换为 `bool` 类型：

```
int ival;
if (ival) // ival converted to bool
while (cin) // cin converted to bool
```

Conditions occur as the first operand of the conditional (`?:`) operator and as the operand(s) to the logical NOT (`!`) , logical AND (`&&`), and logical OR (`||`) operators. Conditions also appear in the `if`, `while`, `for`, and `do while` statements. (We cover the `do while` in [Chapter 6](#))

条件操作符 (`?:`) 中的第一个操作数以及逻辑非 (`!`)、逻辑与 (`&&`) 和逻辑或 (`||`) 的操作数都是条件表达式。出现在 `if`、`while`、`for` 和 `do while` 语句中的同样也是条件表达式（其中 `do while` 将在[第六章](#)中学习）。

- An expression used to initialize or assign to a variable is converted to the type of the variable:

用一表达式初始化某个变量，或将一表达式赋值给某个变量，则该表达式被转换为该变量的类型：

```
int ival = 3.14; // 3.14 converted to int
int *ip;
ip = 0; // the int 0 converted to a null pointer of type int *
```

In addition, as we'll see in [Chapter 7](#), implicit conversions also occur during function calls.

另外，在函数调用中也可能发生隐式类型转换，我们将在[第七章](#)学习这方面的内容。

5.12.2. The Arithmetic Conversions

5.12.2. 算术转换

The language defines a set of conversions among the built-in types. Among these, the most common are the [arithmetic conversions](#), which ensure that the two operands of a binary operator, such as an arithmetic or logical operator, are converted to a common type before the operator is evaluated. That common type is also the result type of the expression.

C++ 语言为内置类型提供了一组转换规则，其中最常用的是[算术转换](#)。算术转换保证在执行操作之前，将二元操作符（如算术或逻辑操作符）的两个操作数转换为同一类型，并使表达式的值也具有相同的类型。

The rules define a hierarchy of type conversions in which operands are converted to the widest type in the expression. The conversion rules are defined so as to preserve the precision of the values involved in a multi-type expression. For example, if one operand is of type `long double`, then the other is converted to type `long double` regardless of what the second type is.

算术转换规则定义了一个类型转换层次，该层次规定了操作数应按什么次序转换为表达式中最宽的类型。在包含多种类型的表达式中，转换规则要确保计算值的精度。例如，如果一个操作数的类型是 `long double`，则无论另一个操作数是什么类型，都将被转换为 `long double`。

The simplest kinds of conversion are [integral promotions](#). Each of the integral types that are smaller than `int` `char`, `signed char`, `unsigned char`, `short`, and `unsigned short` is promoted to `int` if all possible values of that type fit in an `int`. Otherwise, the value is promoted to `unsigned int`. When `bool` values are promoted to `int`, a `false` value promotes to zero and `true` to one.

最简单的转换为[整型提升](#)：对于所有比 `int` 小的整型，包括 `char`、`signed char`、`unsigned char`、`short` 和 `unsigned short`，如果该类型的所有的值都能包容在 `int` 内，它们就会被提升为 `int` 型，否则，它们将被提升为 `unsigned int`。如果将 `bool` 值提升为 `int`，则 `false` 转换为 0，而 `true` 则转换为 1。

Conversions between Signed and Unsigned Types

有符号与无符号类型之间的转换

When an `unsigned` value is involved in an expression, the conversion rules are defined to preserve the value of the operands. Conversions involving `unsigned` operands depend on the relative sizes of the integral types on the machine. Hence, such conversions are inherently machine dependent.

若表达式中使用了无符号（`unsigned`）数值，所定义的转换规则需保护操作数的精度。`unsigned` 操作数的转换依赖于机器中整型的相对大小，因此，这类转换本质上依赖于机器。

In expressions involving `shorts` and `ints`, values of type `short` are converted to `int`. Expressions involving `unsigned short` are converted to `int` if

Section 5.12. Type Conversions

the `int` type is large enough to represent all the values of an `unsigned short`. Otherwise, both operands are converted to `unsigned int`. For example, if `shorts` are a half word and `ints` a word, then any `unsigned` value will fit inside an `int`. On such a machine, `unsigned shorts` are converted to `int`.

包含 `short` 和 `int` 类型的表达式, `short` 类型的值转换为 `int`。如果 `int` 型足够表示所有 `unsigned short` 型的值, 则将 `unsigned short` 转换为 `int`, 否则, 将两个操作数均转换为 `unsigned int`。例如, 如果 `short` 用半字表示而 `int` 用一个字表示, 则所有 `unsigned` 值都能包容在 `int` 内, 在这种机器上, `unsigned short` 转换为 `int`。

The same conversion happens among operands of type `long` and `unsigned int`. The `unsigned int` operand is converted to `long` if type `long` on the machine is large enough to represent all the values of the `unsigned int`. Otherwise, both operands are converted to `unsigned long`.

`long` 和 `unsigned int` 的转换也是一样的。只要机器上的 `long` 型足够表示 `unsigned int` 型的所有值, 就将 `unsigned int` 转换为 `long` 型, 否则, 将两个操作数均转换为 `unsigned long`。

On a 32-bit machine, `long` and `int` are typically represented in a word. On such machines, expressions involving `unsigned ints` and `longs` are converted to `unsigned long`.

在 32 位的机器上, `long` 和 `int` 型通常用一个字长表示, 因此当表达式包含 `unsigned int` 和 `long` 两种类型, 其操作数都应转换为 `unsigned long` 型。

Conversions for expressions involving `signed` and `unsigned int` can be surprising. In these expressions the `signed` value is converted to `unsigned`. For example, if we compare a plain `int` and an `unsigned int`, the `int` is first converted to `unsigned`. If the `int` happens to hold a negative value, the result will be converted as described in [Section 2.1.1](#) (p. 36), with all the attendant problems discussed there.

对于包含 `signed` 和 `unsigned int` 型的表达式, 其转换可能出乎我们的意料。表达式中的 `signed` 型数值会被转换为 `unsigned` 型。例如, 比较 `int` 型和 `unsigned int` 型的简单变量, 系统首先将 `int` 型数值转换为 `unsigned int` 型, 如果 `int` 型的值恰好为负数, 其结果将以[第 2.1.1 节](#)介绍的方法转换, 并带来该节描述的所有副作用。

Understanding the Arithmetic Conversions

理解算术转换

The best way to understand the arithmetic conversions is to study lots of examples. In most of the following examples, either the operands are converted to the largest type involved in the expression or, in the case of assignment expressions, the right-hand operand is converted to the type of the left-hand operand:

研究大量例题是帮助理解算术转换的最好方法。下面大部分例题中, 要么是将操作数转换为表达式中的最大类型, 要么是在赋值表达式中将右操作数转换为左操作数的类型。

```
bool      flag;          char           cval;
short     sval;          unsigned short usval;
int       ival;          unsigned int   uival;
long      lval;          unsigned long  ulval;
float     fval;          double          dval;
3.14159L + 'a'; // promote 'a' to int, then convert to long double
dval + ival; // ival converted to double
dval + fval; // fval converted to double
ival = dval; // dval converted (by truncation) to int
flag = dval; // if dval is 0, then flag is false, otherwise true
cval + fval; // cval promoted to int, that int converted to float
sval + cval; // sval and cval promoted to int
cval + lval; // cval converted to long
ival + ulval; // ival converted to unsigned long
usval + ival; // promotion depends on size of unsigned short and int
uival + lval; // conversion depends on size of unsigned int and long
```

In the first addition, the character constant lowercase '`a`' has type `char`, which as we know from [Section 2.1.1](#) (p. 34) is a numeric value. The numeric value that '`a`' represents depends on the machine's character set. On our ASCII machine, '`a`' represents the number 97. When we add '`a`' to a `long double`, the `char` value is promoted to `int` and then that `int` value is converted to a `long double`. That converted value is added to the `long double` literal. The other interesting cases are the last two expressions involving `unsigned` values.

第一个加法操作的小写字母 '`a`' 是一个 `char` 类型的字符常量, 正如我们在[第 2.1.1 节](#)介绍的, 它是一个数值。字母 '`a`' 表示的数值取决于机器字符集。在 ASCII 机器中, 字母 '`a`' 的值为 97。将 '`a`' 与 `long double` 型数据相加时, `char` 型的值被提升为 `int` 型, 然后将 `int` 型转换为 `long double` 型, 转换后的值再与 `long double` 型字面值相加。另一个有趣的现象是最后两个表达式都包含 `unsigned` 数值。

5.12.3. Other Implicit Conversions

5.12.3. 其他隐式转换

Pointer Conversions

指针转换

In most cases when we use an array, the array is automatically converted to a pointer to the first element:

在使用数组时，大多数情况下数组都会自动转换为指向第一个元素的指针：

```
int ia[10]; // array of 10 ints
int* ip = ia; // convert ia to pointer to first element
```

The exceptions when an array is not converted to a pointer are: as the operand of the address-of (`&`) operator or of `sizeof`, or when using the array to initialize a reference to the array. We'll see how to define a reference (or pointer) to an array in [Section 7.2.4](#) (p. 240).

不将数组转换为指针的例外情况有：数组用作取地址 (`&`) 操作符的操作数或 `sizeof` 操作符的操作数时，或用数组对数组的引用进行初始化时，不会将数组转换为指针。我们将在[第 7.2.4 节](#)学习如何定义指向数组的引用（或指针）。

There are two other pointer conversions: A pointer to any data type can be converted to a `void*`, and a constant integral value of 0 can be converted to any pointer type.

C++ 还提供了另外两种指针转换：指向任意数据类型的指针都可转换为 `void*` 类型；整型数值常量 0 可转换为任意指针类型。

Conversions to `bool`

转换为 `bool` 类型

Arithmetic and pointer values can be converted to `bool`. If the pointer or arithmetic value is zero, then the `bool` is `false`; any other value converts to `true`:

算术值和指针值都可以转换为 `bool` 类型。如果指针或算术值为 0，则其 `bool` 值为 `false`，而其他值则为 `true`:

```
if (cp) /* ... */ // true if cp is not zero
while (*cp) /* ... */ // dereference cp and convert resulting char to bool
```

Here, the `if` converts any nonzero value of `cp` to `true`. The `while` dereferences `cp`, which yields a `char`. The null character has value zero and converts to `false`. All other `char` values convert to `true`.

这里，`if` 语句将 `cp` 的非零值转换为 `true`。`while` 语句则对 `cp` 进行解引用，操作结果产生一个 `char` 型的值。空字符（`null`）具有 0 值，被转换为 `false`，而其他字符值则转换为 `true`。

Arithmetic Type and `bool` Conversions

算术类型与 `bool` 类型的转换

Arithmetic objects can be converted to `bool` and `bool` objects can be converted to `int`. When an arithmetic type is converted to `bool`, zero converts as `false` and any other value converts as `true`. When a `bool` is converted to an arithmetic type, `true` becomes one and `false` becomes zero:

可将算术对象转换为 `bool` 类型，`bool` 对象也可转换为 `int` 型。将算术类型转换为 `bool` 型时，零转换为 `false`，而其他值则转换为 `true`。将 `bool` 对象转换为算术类型时，`true` 变成 1，而 `false` 则为 0:

```
bool b = true;
int ival = b; // ival == 1
double pi = 3.14;
bool b2 = pi; // b2 is true
pi = false; // pi == 0
```

Conversions and Enumeration Types

转换与枚举类型

Objects of an enumeration type ([Section 2.7](#), p. 62) or an enumerator can be automatically converted to an integral type. As a result, they can be used where an integral value is required for example, in an arithmetic expression:

C++ 自动将枚举类型（[第 2.7 节](#)）的对象或枚举成员（`enumerator`）转换为整型，其转换结果可用于任何要求使用整数值的地方。例如，用于算术表达式：

```
// point2d is 2, point2w is 3, point3d is 3, point3w is 4
enum Points { point2d = 2, point2w,
              point3d = 3, point3w };
```

Section 5.12. Type Conversions

```
const size_t array_size = 1024;  
// ok: pt2w promoted to int  
int chunk_size = array_size * pt2w;  
int array_3d = array_size * point3d;
```

The type to which an `enum` object or enumerator is promoted is machine-defined and depends on the value of the largest enumerator. Regardless of that value, an `enum` or enumerator is always promoted at least to `int`. If the largest enumerator does not fit in an `int`, then the promotion is to the smallest type larger than `int` (`unsigned int`, `long` or `unsigned long`) that can hold the enumerator value.

将 `enum` 对象或枚举成员提升为什么类型由机器定义，并且依赖于枚举成员的最大值。无论其最大值是什么，`enum` 对象或枚举成员至少提升为 `int` 型。如果 `int` 型无法表示枚举成员的最大值，则提升到能表示所有枚举成员值的、大于 `int` 型的最小类型（`unsigned int`、`long` 或 `unsigned long`）。

Conversion to `const`

转换为 `const` 对象

A `nonconst` object can be converted to a `const` object, which happens when we use a `nonconst` object to initialize a reference to `const` object. We can also convert the address of a `nonconst` object (or convert a `nonconst` pointer) to a pointer to the related `const` type:

当使用非 `const` 对象初始化 `const` 对象的引用时，系统将非 `const` 对象转换为 `const` 对象。此外，还可以将非 `const` 对象的地址（或非 `const` 指针）转换为指向相关 `const` 类型的指针：

```
int i;  
const int ci = 0;  
const int &j = i; // ok: convert non-const to reference to const int  
const int *p = &ci; // ok: convert address of non-const to address of a const
```

Conversions Defined by the Library Types

由标准库类型定义的转换

Class types can define conversions that the compiler will apply automatically. Of the library types we've used so far, there is one important conversion that we have used. When we read from an `istream` as a condition

类类型可以定义由编译器自动执行的类型转换。迄今为止，我们使用过的标准库类型中，有一个重要的类型转换。从 `istream` 中读取数据，并将此表达式作为 `while` 循环条件：

```
string s;  
while (cin >> s)
```

we are implicitly using a conversion defined by the IO library. In a condition such as this one, the expression `cin >> s` is evaluated, meaning `cin` is read. Whether the read succeeds or fails, the result of the expression is `cin`.

这里隐式使用了 IO 标准库定义的类型转换。在与此类似的条件下，求解表达式 `cin >> s`，即读 `cin`。无论读入是否成功，该表达式的结果都是 `cin`。

The condition in the `while` expects a value of type `bool`, but it is given a value of type `istream`. That `istream` value is converted to `bool`. The effect of converting an `istream` to `bool` is to test the state of the stream. If the last attempt to read from `cin` succeeded, then the state of the stream will cause the conversion to `bool` to be `true` the `while` test will succeed. If the last attempt failed say because we hit end-of-file then the conversion to `bool` will yield `false` and the `while` condition will fail.

`while` 循环条件应为 `bool` 类型的值，但此时给出的却是 `istream` 类型的值，于是 `istream` 类型的值应转换为 `bool` 类型。将 `istream` 类型转换为 `bool` 类型意味着要检验流的状态。如果最后一次读 `cin` 的尝试是成功的，则流的状态将导致上述类型转换为 `bool` 类型后获得 `true` 值——`while` 循环条件成立。如果最后一次尝试失败，比如说已经读到文件尾了，此时将 `istream` 类型转换为 `bool` 类型后得 `false`，`while` 循环条件不成立。

Exercises Section 5.12.3

Exercise 5.31: Given the variable definitions on page 180, explain what conversions take place when evaluating the following expressions:

记住，你可能需要考虑操作符的结合性，以便在表达式含有多个操作符的情况下确定答案。

- (a) `if (fval)`
- (b) `dval = fval + ival;`
- (c) `dval + ival + cval;`

Remember that you may need to consider associativity of the operators in order to determine the answer in the case of expressions involving more than one operator.

记住，你可能需要考虑操作符的结合性，以便在表达式含有多个操作符的情况下确定答案。

5.12.4. Explicit Conversions

5.12.4. 显式转换

An explicit conversion is spoken of as a [cast](#) and is supported by the following set of named cast operators: [`static cast`](#), [`dynamic cast`](#), [`const cast`](#), and [`reinterpret cast`](#).

显式转换也称为强制类型转换 ([cast](#))，包括以下列名字命名的强制类型转换操作符：[`static cast`](#)、[`dynamic cast`](#)、[`const cast`](#) 和 [`reinterpret cast`](#)。



Although necessary at times, casts are inherently dangerous constructs.

虽然有时候确实需要强制类型转换，但是它们本质上是非常危险的。

5.12.5. When Casts Might Be Useful

5.12.5. 何时需要强制类型转换

One reason to perform an explicit cast is to override the usual standard conversions. The following compound assignment

因为要覆盖通常的标准转换，所以需显式使用强制类型转换。下面的复合赋值：

```
double dval;
int ival;
ival *= dval; // ival = ival * dval
```

converts `ival` to `double` in order to multiply it by `dval`. That `double` result is then truncated to `int` in order to assign it to `ival`. We can eliminate the unnecessary conversion of `ival` to `double` by explicitly casting `dval` to `int`:

为了与 `dval` 做乘法操作，需将 `ival` 转换为 `double` 型，然后将乘法操作的 `double` 型结果截尾为 `int` 型，再赋值给 `ival`。为了去掉将 `ival` 转换为 `double` 型这个不必要的转换，可通过如下强制将 `dval` 转换为 `int` 型：

```
ival *= static_cast<int>(dval); // converts dval to int
```

Another reason for an explicit cast is to select a specific conversion when more than one conversion is possible. We will look at this case more closely in [Chapter 14](#).

显式使用强制类型转换的另一个原因是：可能存在多种转换时，需要选择一种特定的类型转换。我们将在[第 14 章](#)中详细讨论这种情况。

5.12.6. Named Casts

5.12.6. 命名的强制类型转换

The general form for the named cast notation is the following:

命名的强制类型转换符号的一般形式如下：

```
cast-name<type>(expression);
```

`cast-name` may be one of `static cast`, `const cast`, `dynamic cast`, or `reinterpret cast`. `type` is the target type of the conversion, and `expression` is the value to be cast. The type of cast determines the specific kind of conversion that is performed on the `expression`.

其中 `cast-name` 为 `static cast`、`dynamic cast`、`const cast` 和 `reinterpret cast` 之一，`type` 为转换的目标类型，而 `expression` 则是被强制转换的值。强制转换的类型指定了在 `expression` 上执行某种特定类型的转换。

Section 5.12. Type Conversions

dynamic_cast

A `dynamic_cast` supports the run-time identification of objects addressed either by a pointer or reference. We cover `dynamic_cast` in [Section 18.2](#) (p. 772).

`dynamic_cast` 支持运行时识别指针或引用所指向的对象。对 `dynamic_cast` 的讨论将在[第 18.2 节](#)中进行。

const_cast

A `const_cast`, as its name implies, casts away the `const`ness of its expression. For example, we might have a function named `string_copy` that we are certain reads, but does not write, its single parameter of type `char*`. If we have access to the code, the best alternative would be to correct it to take a `const char*`. If that is not possible, we could call `string_copy` on a `const` value using a `const_cast`:

`const char *pc_str;`
`char *pc = string_copy(const_cast<char*>(pc_str));`

Only a `const_cast` can be used to cast away `const`ness. Using any of the other three forms of cast in this case would result in a compile-time error. Similarly, it is a compile-time error to use the `const_cast` notation to perform any type conversion other than adding or removing `const`.

只有使用 `const_cast` 才能将 `const` 性质转换掉。在这种情况下，试图使用其他三种形式的强制转换都会导致编译时的错误。类似地，除了添加或删除 `const` 特性，用 `const_cast` 符来执行其他任何类型转换，都会引起编译错误。

static_cast

Any type conversion that the compiler performs implicitly can be explicitly requested by using a `static_cast`:

`double d = 97.0;`
`// cast specified to indicate that the conversion is intentional`
`char ch = static_cast<char>(d);`

Such casts are useful when assigning a larger arithmetic type to a smaller type. The cast informs both the reader of the program and the compiler that we are aware of and are not concerned about the potential loss of precision. Compilers often generate a warning for assignments of a larger arithmetic type to a smaller type. When we provide the explicit cast, the warning message is turned off.

当需要将一个较大的算术类型赋值给较小的类型时，使用强制转换非常有用。此时，强制类型转换告诉程序的读者和编译器：我们知道并且不关心潜在的精度损失。对于从一个较大的算术类型到一个较小类型的赋值，编译器通常会产生警告。当我们显式地提供强制类型转换时，警告信息就会被关闭。

A `static_cast` is also useful to perform a conversion that the compiler will not generate automatically. For example, we can use a `static_cast` to retrieve a pointer value that was stored in a `void*` pointer ([Section 4.2.2](#), p. 119):

如果编译器不提供自动转换，使用 `static_cast` 来执行类型转换也是很有用的。例如，下面的程序使用 `static_cast` 找回存放在 `void*` 指针中的值（[第 4.2.2 节](#)）：

```
void* p = &d; // ok: address of any data object can be stored in a void*
// ok: converts void* back to the original pointer type
double *dp = static_cast<double*>(p);
```

When we store a pointer in a `void*` and then use a `static_cast` to cast the pointer back to its original type, we are guaranteed that the pointer value is preserved. That is, the result of the cast will be equal to the original address value.

可通过 `static_cast` 将存放在 `void*` 中的指针值强制转换为原来的指针类型，此时我们应确保保持指针值。也就是说，强制转换的结果应与原来的地址值相等。

reinterpret_cast

A `reinterpret_cast` generally performs a low-level reinterpretation of the bit pattern of its operands.

`reinterpret_cast` 通常为操作数的位模式提供较低层次的重新解释。

A `reinterpret_cast` is inherently machine-dependent. Safely using `reinterpret_cast` requires completely understanding the types involved as well as the details of how the compiler implements the cast.

`reinterpret_cast` 本质上依赖于机器。为了安全地使用 `reinterpret_cast`，要求程序员完全理解所涉及的数据类型，以及编译器实现强制类型转换的细节。



Section 5.12. Type Conversions

As an example, in the following cast

例如, 对于下面的强制转换:

```
int *ip;
char *pc = reinterpret_cast<char*>(ip);
```

the programmer must never forget that the actual object addressed by `pc` is an `int`, not a character array. Any use of `pc` that assumes it's an ordinary character pointer is likely to fail at run time in interesting ways. For example, using it to initialize a `string` object such as

程序员必须永远记得 `pc` 所指向的真实对象其实是 `int` 型, 而并非字符数组。任何假设 `pc` 是普通字符指针的应用, 都有可能带来有趣的运行时错误。例如, 下面语句用 `pc` 来初始化一个 `string` 对象:

```
string str(pc);
```

is likely to result in bizarre run-time behavior.

它可能会引起运行时的怪异行为。

The use of `pc` to initialize `str` is a good example of why explicit casts are dangerous. The problem is that types are changed, yet there are no warnings or errors from the compiler. When we initialized `pc` with the address of an `int`, there is no error or warning from the compiler because we explicitly said the conversion was okay. Any subsequent use of `pc` will assume that the value it holds is a `char*`. The compiler has no way of knowing that it actually holds a pointer to an `int`. Thus, the initialization of `str` with `pc` is absolutely correct albeit in this case meaningless or worse! Tracking down the cause of this sort of problem can prove extremely difficult, especially if the cast of `ip` to `pc` occurs in a file separate from the one in which `pc` is used to initialize a `string`.

用 `pc` 初始化 `str` 这个例子很好地说明了显式强制转换是多么的危险。问题源于类型已经改变时编译器没有提供任何警告或错误提示。当我们用 `int` 型地址初始化 `pc` 时, 由于显式地声明了这样的转换是正确的, 因此编译器不提供任何错误或警告信息。后面对 `pc` 的使用都假设它存放的是 `char*` 型对象的地址, 编译器确实无法知道 `pc` 实际上是指向 `int` 型对象的指针。因此用 `pc` 初始化 `str` 是完全正确的——虽然实际上是无意义的或是错误的。查找这类问题的原因相当困难, 特别是如果 `ip` 到 `pc` 的强制转换和使用 `pc` 初始化 `string` 对象这两个应用发生在不同文件中的时候。

Advice: Avoid Casts

建议: 避免使用强制类型转换

By using a cast, the programmer turns off or dampens normal type-checking ([Section 2.3, p. 44](#)). We strongly recommend that programmers avoid casts and believe that most well-formed C++ programs can be written without relying on casts.

强制类型转换关闭或挂起了正常的类型检查 ([第 2.3 节](#))。强烈建议程序员避免使用强制类型转换, 不依赖强制类型转换也能写出很好的 C++ 程序。

This advice is particularly important regarding use of `reinterpret_casts`. Such casts are always hazardous. Similarly, use of `const_cast` almost always indicates a design flaw. Properly designed systems should not need to cast away `const`. The other casts, `static_cast` and `dynamic_cast`, have their uses but should be needed infrequently. Every time you write a cast, you should think hard about whether you can achieve the same result in a different way. If the cast is unavoidable, errors can be mitigated by limiting the scope in which the cast value is used and by documenting all assumptions about the types involved.

这个建议在如何看待 `reinterpret_cast` 的使用时非常重要。此类强制转换总是非常危险的。相似地, 使用 `const_cast` 也总是预示着设计缺陷。设计合理的系统应不需要使用强制转换抛弃 `const` 特性。其他的强制转换, 如 `static_cast` 和 `dynamic_cast`, 各有各的用途, 但都不应频繁使用。每次使用强制转换前, 程序员应该仔细考虑是否还有其他不同的方法可以达到同一目的。如果非强制转换不可, 则应限制强制转换值的作用域, 并且记录所有假定涉及的类型, 这样能减少错误发生的机会。

5.12.7. Old-Style Casts

5.12.7. 旧式强制类型转换

Prior to the introduction of named cast operators, an explicit cast was performed by enclosing a type in parentheses:

```
char *pc = (char*) ip;
```

The effect of this cast is the same as using the `reinterpret_cast` notation. However, the visibility of this cast is considerably less, making it even more difficult to track down the rogue cast.

效果与使用 `reinterpret_cast` 符号相同, 但这种强制转换的可视性比较差, 难以跟踪错误的转换。

Standard C++ introduced the named cast operators to make casts more visible and to give the programmer a more finely tuned tool to use when casts are necessary. For example, nonpointer `static_casts` and `const_casts` tend to be safer than `reinterpret_casts`. As a result, the programmer (as well as readers and tools operating on the program) can clearly identify the potential risk level of each explicit cast in code.

Section 5.12. Type Conversions

标准 C++ 为了加强类型转换的可视性，引入命名的强制转换操作符，为程序员在必须使用强制转换时提供了更好的工具。例如，非指针的 `static_cast` 和 `const_cast` 要比 `reinterpret_cast` 更安全。结果使程序员（以及读者和操纵程序的工具）可清楚地辨别代码中每个显式的强制转换潜在的风险级别。



Although the old-style cast notation is supported by Standard C++, we recommend it be used only when writing code to be compiled either under the C language or pre-Standard C++.

虽然标准 C++ 仍然支持旧式强制转换符号，但是我们建议，只有在 C 语言或标准 C++ 之前的编译器上编写代码时，才使用这种语法。

The old-style cast notation takes one of the following two forms:

旧式强制转换符号有下列两种形式：

```
type (expr); // Function-style cast notation  
(type) expr; // C-language-style cast notation
```

Depending on the types involved, an old-style cast has the same behavior as a `const_cast`, a `static_cast`, or a `reinterpret_cast`. When used where a `static_cast` or a `const_cast` would be legal, an old-style cast does the same conversion as the respective named cast. If neither is legal, then an old-style cast performs a `reinterpret_cast`. For example, we might rewrite the casts from the previous section less clearly using old-style notation:

旧式强制转换依赖于所涉及的数据类型，具有与 `const_cast`、`static_cast` 和 `reinterpret_cast` 一样的行为。在合法使用 `static_cast` 或 `const_cast` 的地方，旧式强制转换提供了与各自对应的命名强制转换一样的功能。如果这两种强制转换均不合法，则旧式强制转换执行 `reinterpret_cast` 功能。例如，我们可用旧式符号重写上一节的强制转换：

```
int ival; double dval;  
ival += int (dval); // static_cast: converts double to int  
const char* pc_str;  
string_copy((char*)pc_str); // const_cast: casts away const  
int *ip;  
char *pc = (char*)ip; // reinterpret_cast: treats int* as char*
```

The old-style cast notation remains supported for backward compatibility with programs written under pre-Standard C++ and to maintain compatibility with the C language.

支持旧式强制转换符号是为了对“在标准 C++ 之前编写的程序”保持向后兼容性，并保持与 C 语言的兼容性。

Exercises Section 5.12.7

Exercise Given the following set of definitions,
5.32: 给定下列定义：

```
char cval; int ival; unsigned int ui;  
float fval; double dval;
```

identify the implicit type conversions, if any, taking place:

指出可能发生的（如果有的话）隐式类型转换：

- (a) `cval = 'a' + 3;` (b) `fval = ui - ival * 1.0;`
(c) `dval = ui * fval;` (d) `cval = ival + fval + dval;`

Exercise Given the following set of definitions,
5.33: 给定下列定义：

```
int ival; double dval;  
const string *ps; char *pc; void *pv;
```

rewrite each of the following using a named cast notation:

用命名的强制类型转换符号重写下列语句：

- (a) `pv = (void*)ps;` (b) `ival = int(*pc);`
(c) `pv = &dval;` (d) `pc = (char*) pv;`

Team LiB

◀ PREVIOUS NEXT ▶

Chapter Summary

小结

C++ provides a rich set of operators and defines their meaning when applied to values of the built-in types. Additionally, the language supports operator overloading, which allows us to define the meaning of the operators for class types. We'll see in [Chapter 14](#) how to define operators for our own types.

C++ 提供了丰富的操作符，并定义了用于内置类型值时操作符的含义。除此之外，C++ 还支持操作符重载，允许由程序员自己来定义用于类类型时操作符的含义。我们将在[第十四章](#)中学习如何重载自定义的操作符。

To understand compound expressionsexpressions involving more than one operatorit is necessary to understand precedence, associativity, and order of operand evaluation. Each operator has a precedence level and associativity. Precedence determines how operators are grouped in a compound expression. Associativity determines how operators at the same precedence level are grouped.

要理解复合表达式（即含有多个操作符的表达式）就必须先了解优先级、结合性以及操作数的求值次序。每一个操作符都有自己的优先级别和结合性。优先级规定复合表达式中操作符结合的方式，而结合性则决定同一个优先级的操作符如何结合。

Most operators do not specify the order in which operands are evaluated: The compiler is free to evaluate either the left- or right-hand operand first. Often, the order of operand evaluation has no impact on the result of the expression. However, if both operands refer to the same object and one of the operands changes that object, then the program has a serious bugand a bug that may be hard to find.

大多数操作符没有规定其操作数的求值顺序：由编译器自由选择先计算左操作数还是右操作数。通常，操作数的求值顺序不会影响表达式的结果。但是，如果操作符的两个操作数都与同一个对象相关，而且其中一个操作数改变了该对象的值，则程序将会因此而产生严重的错误——而且这类错误很难发现。

Finally, it is possible to write an expression that is given one type but where a value of another type is required. In such cases, the compiler will automatically apply a conversion (either built-in or defined for a class type) to transform the given type into the type that is required. Conversions can also be requested explicitly by using a cast.

最后，可以使用某种类型编写表达式，而实际需要的是另一类型的值。此时，编译器自动实现类型转换（既可以是内置类型也可以是为类类型而定义的），将特定类型转换为所需的类型。C++ 还提供了强制类型转换显式地将数值转换为所需的数据类型。

Defined Terms

术语

arithmetic conversion (算术转换)

A conversion from one arithmetic type to another. In the context of the binary arithmetic operators, arithmetic conversions usually attempt to preserve precision by converting a smaller type to a larger type (e.g., small integral types, such as `char` and `short`, are converted to `int`).

算术类型之间的转换。在使用二元算术操作符的地方，算术转换通常将较小的类型转换为较大的类型，以确保精度（例如，将小的整型 `char` 型和 `short` 型转换为 `int` 型）。

associativity (结合性)

Determines how operators of the same precedence are grouped. Operators can be either right associative (operators are grouped from right to left) or left associative (operators are grouped from left to right).

决定同一优先级的操作符如何结合。`C++` 的操作符要么是左结合（操作符从左向右结合）要么是右结合（操作符从右向左结合）。

binary operators (二元操作符)

Operators that take two operands.

有两个操作数的操作符。

cast (强制类型转换)

An explicit conversion.

显式的类型转换。

compound expression (复合表达式)

An expression involving more than one operator.

含有多个操作符的表达式。

const cast

A cast that converts a `const` object to the corresponding non`const` type.

将 `const` 对象转换为相应的非 `const` 类型的强制转换。

conversion (类型转换)

Process whereby a value of one type is transformed into a value of another type. The language defines conversions among the built-in types. Conversions to and from class types are also possible.

将某种类型的值转换为另一种类型值的处理方式。`C++` 语言定义了内置类型之间的类型转换，也允许将某种类型转换为类类型或将类类型转换为某种类型。

dangling pointer (悬垂指针)

A pointer that refers to memory that once had an object but no longer does. Dangling pointers are the source of program errors that are quite difficult to detect.

指向曾经存在的对象的指针，但该对象已经不再存在了。悬垂指针容易导致程序错误，而且这种错误很难检测出来。

delete expression (`delete` 表达式)

A `delete` expression frees memory that was allocated by `new`. There are two forms of `delete`:

`delete` 表达式用于释放由 `new` 动态分配的内存。`delete` 有两种语法形式：

```
delete p;      // delete object
```

```
delete [] p; // delete array
```

In the first case, `p` must be a pointer to a dynamically allocated object; in the second, `p` must point to the first element in a dynamically allocated array. In C++ programs, `delete` replaces the use of the C library `free` function.

第一种形式的 `p` 必须是指向动态创建对象的指针；第二种形式的 `p` 则应指向动态创建数组的第一个元素。C++ 程序使用 `delete` 取代 C 语言的标准库函数 `free`。

dynamic cast

Used in combination with inheritance and run-time type identification. See [Section 18.2](#) (p. 772).

用于结合继承和运行时类型识别。参见[第 18.2 节](#)。

expression (表达式)

The lowest level of computation in a C++ program. Expressions generally apply an operator to one or more operands. Each expression yields a result. Expressions can be used as operands, so we can write compound expressions requiring the evaluation of multiple operators.

C++ 程序中的最低级的计算。表达式通常将一个操作符用于一个或多个操作数。每个表达式产生一个结果。表达式也可用作操作数，因此可用多个操作符编写复合表达式。

implicit conversion (隐式类型转换)

A conversion that is automatically generated by the compiler. Given an expression that needs a particular type but has an operand of a differing type, the compiler will automatically convert the operand to the desired type if an appropriate conversion exists.

编译器自动实现的类型转换。假设表达式需要某种特定类型的数值，但其操作数却是其他不同的类型，此时如果系统定义了适当的类型转换，编译器会自动根据转换规则将该操作数转换为需要的类型。

integral promotions (整型提升)

Subset of the standard conversions that take a smaller integral type to its most closely related larger type. Integral types (e.g. `short`, `char`, etc.) are promoted to `int` or `unsigned int`.

整型提升是标准类型转换规则的子集，它将较小的整型转换为最接近的较大数据类型。整型（如 `short`、`char` 等）被提升为 `int` 型或 `unsigned int` 型。

new expression (new 表达式)

A `new` expression allocates memory at run time from the free store. This chapter looked at the form that allocates a single object:

`new` 表达式用于运行时从自由存储区中分配内存空间。本章使用 `new` 创建单个对象，其语法形式为：

```
new type;
new type(inits);
```

allocates an object of the indicated `type` and optionally initializes that object using the initializers in `inits`. Returns a pointer to the object. In C++ programs, `new` replaces use of the C library `malloc` function.

`new` 表达式创建指定 `type` 类型的对象，并且可选择在创建时使用 `inits` 初值初始化该对象，然后返回指向该对象的指针。C++ 程序使用 `new` 取代 C 语言的标准库函数 `malloc`。

operands (操作数)

Values on which an expression

表达式操纵的值。

operator (操作符)

Symbol that determines what action an expression performs. The language defines a set of operators and what those operators mean when applied to values of built-in type. The language also defines the precedence and associativity of each operator and specifies how many operands each operator takes. Operators may be overloaded and applied to values of class type.

决定表达式执行什么功能的符号。C++ 语言定义了一组操作符以及将它们用于内置类型时的含义，还定义了每个操作符的优先级和结合性以及它们所需要的操作数个数。C++ 语言允许重载操作符，以使它们能用于类类型的对象。

operator overloading (操作符重载)

The ability to redefine an operator to apply to class types. We'll see in [Chapter 14](#) how to define overloaded versions of operators.

对操作符的功能重定义以用于类类型。我们将在[第十四章](#)中学习如何重载不同的操作符版本。

order of evaluation (求值顺序)

Order, if any, in which the operands to an operator are evaluated. In most cases in C++ the compiler is free to evaluate operands in any order.

操作符的操作数计算顺序（如果有的话）。大多数情况下，C++ 编译器可自由选择操作数求解的次序。

precedence (优先级)

Defines the order in which different operators in a compound expression are grouped. Operators with higher precedence are grouped more tightly than operators with lower precedence.

定义了复合表达式中不同操作符的结合方式。高优先级的操作符要比低优先级操作符结合得更紧密。

reinterpret_cast

Interprets the contents of the operand as a different type. Inherently machine-dependent and dangerous.

将操作数内容解释为另一种不同的类型。这类强制转换本质上依赖于机器，而且是非常危险的。

result (结果)

The value or object obtained by evaluating an expression.

计算表达式所获得的值或对象。

static_cast

An explicit request for a type conversion that the compiler would do implicitly. Often used to override an implicit conversion that the compiler would otherwise perform.

编译器隐式执行的任何类型转换都可以由 `static_cast` 显式完成。我们常常使用 `static_cast` 取代由编译器实现的隐式转换。

unary operators (一元操作符)

Operators that take a single operand.

只有一个操作数的操作符。

~ operator

The bitwise NOT operator. Inverts the bits of its operand.

逐位求反操作符，将其操作数的每个位都取反。

, operator

The comma operator. Expressions separated by a comma are evaluated left to right. Result of a comma expression is the value of the right-most expression.

逗号操作符。用逗号隔开的表达式从左向右计算，整个逗号表达式的结果为其最右边的表达式的值。

^ operator

The bitwise exclusive OR operator. Generates a new integral value in which each bit position is 1 if either but not both operands contain a 1 in that bit position; otherwise, the bit is 0.

位异或操作符。在做位异或操作时，如果两个操作数对应位置上的位只有一个（注意不是两个）为 1，则操作结果中该位为 1，否则为 0，位异或操作产生一个新的整数值。

| operator

The bitwise OR operator. Generates a new integral value in which each bit position is 1 if either operand has a 1 in that position; otherwise the bit is 0.

位或操作符。在做位或操作时，如果两个操作数对应位置上的位至少有一个为 1，则操作结果中该位为 1，否则为 0，位或操作产生一个新的整数值。

++ operator

The increment operator. The increment operator has two forms, prefix and postfix. Prefix increment yields an lvalue. It adds one to the operand and returns the changed value of the operand. Postfix increment yields an rvalue. It adds one to the operand and returns the original, unchanged value of the operand.

自增操作符。自增操作符有两种形式：前置操作和后置操作。前自增操作生成左值，在给操作数加1后返回改变后的操作数值。后自增操作生成右值，给操作数加1但返回未改变的操作数原值。

-- operator

The decrement operator. has two forms, prefix and postfix. Prefix decrement yields an lvalue. It subtracts one from the operand and returns the changed value of the operand. Postfix decrement yields an rvalue. It subtracts one from the operand and returns the original, unchanged value of the operand.

自减操作符。自减操作符也有两种形式：前置操作和后置操作。前自减操作生成左值，在给操作数减1后返回改变后的操作数值。后自减操作生成右值，给操作数减1但返回未改变的操作数原值。

<< operator

The left-shift operator. Shifts bits in the left-hand operand to the left. Shifts as many bits as indicated by the right-hand operand. The right-hand operand must be zero or positive and strictly less than the number of bits in the left-hand operand.

左移操作符。左移操作符将其左操作数的各个位向左移动若干个位，移动的位数由其右操作数指定。左移操作符的右操作数必须是0值或正整数，而且它的值必须严格小于左操作数的位数。

>> operator

The right-shift operator. Like the left-shift operator except that bits are shifted to the right. The right-hand operand must be zero or positive and strictly less than the number of bits in the left-hand operand.

右移操作符。与左移操作符类似，右移操作符将其左操作数的各个位向右移动，其右操作数必须是0值或正整数，而且它的值必须严格小于左操作数的位数。

Chapter 6. Statements

第六章 语句

CONTENTS

<u>Section 6.1</u> Simple Statements	192
<u>Section 6.2</u> Declaration Statements	193
<u>Section 6.3</u> Compound Statements (Blocks)	193
<u>Section 6.4</u> Statement Scope	194
<u>Section 6.5</u> The <code>if</code> Statement	195
<u>Section 6.6</u> The <code>switch</code> Statement	199
<u>Section 6.7</u> The <code>while</code> Statement	204
<u>Section 6.8</u> The <code>for</code> Loop Statement	207
<u>Section 6.9</u> The <code>do while</code> Statement	210
<u>Section 6.10</u> The <code>break</code> Statement	212
<u>Section 6.11</u> The <code>continue</code> Statement	214
<u>Section 6.12</u> The <code>goto</code> Statement	214
<u>Section 6.13</u> <code>try</code> Blocks and Exception Handling	215
<u>Section 6.14</u> Using the Preprocessor for Debugging	220
<u>Chapter Summary</u>	223
<u>Defined Terms</u>	223

Statements are analogous to sentences in a natural language. In C++ there are simple statements that execute a single task and compound statements that consist of a block of statements that execute as a unit. Like most languages, C++ provides statements for conditional execution and loops that repeatedly execute the same body of code. This chapter looks in detail at the statements supported by C++.

语句类似于自然语言中的句子。C++ 语言既有六项单一句分的简单语句，也有作为十单元执行的由一组语句组成的复合语句。和其他多数语言一样，C++ 也提供了实现条件分支结构的语句以及重复地执行同一段代码的循环结构。本章将详细讨论 C++ 所支持的语句。

By default, statements are executed sequentially. Except for the simplest programs, sequential execution is inadequate. Therefore, C++ also defines a set of **flow-of-control** statements that allow statements to be executed conditionally or repeatedly. The **`if`** and **`switch`** statements support conditional execution. The **`for`**, **`while`**, and **`do while`** statements support repetitive execution. These latter statements are often referred to as ***loops*** or ***iteration*** statements.

通常情况下，语句是顺序执行的。但是，除了最简单的程序外，只有顺序执行往往并不足够。为此，C++ 定义了 [结构的语句](#)，允许有条件地执行或者重复地执行某些部分功能。**if** 和 **switch** 语句提供了条件分支结构，而 **for**、**while** 和 **do while** 语句则支持重复执行的功能。后几种语句常称为循环或者迭代语句。

6.1. Simple Statements

6.1. 简单语句

Most statements in C++ end with a semicolon. An expression, such as `ival + 5`, becomes an [expression statement](#) by following it with a semicolon. Expression statements cause the expression to be evaluated. In the case of

C++ 中，大多数语句以分号结束。例如，像 `ival + 5` 这样的表达式，在后面加上分号，就是一条[表达式语句](#)。表达式语句用于计算表达式。但执行下面的语句

```
ival + 5; // expression statement
```

evaluating this expression is useless: The result is calculated but not assigned or otherwise used. More commonly, expression statements contain expressions that when evaluated affect the program's state. Examples of such expressions are those that use assignment, increment, input, or output operators.

却没有任何意义：因为计算出来的结果没有用于赋值或其他用途。通常，表达式语句所包含的表达式在计算时会影响程序的状态，使用赋值、自增、输入或输出操作符的表达式就是很好的例子。

Null Statements

空语句

The simplest form of statement is the empty, or [null statement](#). It takes the following form (a single semicolon):

程序语句最简单的形式是[空语句](#)，它使用以下的形式（只有一个单独的分号）：

```
; // null statement
```

A null statement is useful where the language requires a statement but the program's logic does not. Such usage is most common when a loop's work can be done within the condition. For example, we might want to read an input stream, ignoring everything we read until we encounter a particular value:

如果在程序的某个地方，语法上需要一个语句，但逻辑上并不需要，此时应该使用空语句。这种用法常见于在循环条件判断部分就能完成全部循环工作的情况。例如，下面程序从输入流中读取数据，在获得某个特殊值前无需作任何操作：

```
// read until we hit end-of-file or find an input equal to sought
while (cin >> s && s != sought)
    ; // null statement
```

The condition reads a value from the standard input and implicitly tests `cin` to see whether the read was successful. Assuming the read succeeded, the second part of the condition tests whether the value we read is equal to the value in `sought`. If we found the value we want, then the `while` loop is exited; otherwise, the condition is tested again, which involves reading another value from `cin`.

循环条件从标准输入中读入一个值并检验 `cin` 的读入是否成功。如果成功读取数据，循环条件紧接着检查该值是否等于 `sought`。如果找到了需要的值，则退出 `while` 循环；否则，循环条件再次从 `cin` 里读入另一个值继续检验。



A null statement should be commented, so that anyone reading the code can see that the statement was omitted intentionally.

使用空语句时应该加上注释，以便任何读这段代码的人都知道该语句是有意省略的。

Section 6.1. Simple Statements

Because a null statement is a statement, it is legal anywhere a statement is expected. For this reason, semicolons that might appear illegal are often nothing more than null statements:

由于空语句也是一个语句，因此可用在任何允许使用语句的地方。由于这个原因，那些看似非法的分号往往只不过是一个空语句而已：

```
// ok: second semicolon is superfluous null statement  
ival = v1 + v2;;
```

This fragment is composed of two statements: the expression statement and the null statement.

这个程序段由两条语句组成：一条表达式语句和一条空语句。



Extraneous null statements are not always harmless.

无关的空语句并非总是无害的。

An extra semicolon following the condition in a `while` or `if` can drastically alter the programmer's intent:

在 `while` 或 `if` 条件后面额外添加分号，往往会彻底改变程序员的意图：

```
// disaster: extra semicolon: loop body is this null statement  
while (iter != svec.end()) ; // null statement--while body is empty!  
    ++iter; // increment is not part of the loop
```

This program will loop indefinitely. Contrary to the indentation, the increment is not part of the loop. The loop body is a null statement caused by the extra semicolon following the condition.

这个程序将会无限次循环。与缩进的意义相反，此自增语句并不是循环的一部分。由于循环条件后面多了一个分号，因此循环体为空语句。

6.2. Declaration Statements

6.2. 声明语句

Defining or declaring an object or a class is a statement. Definition statements are usually referred to as [declaration statements](#) although definition statement might be more accurate. We covered definitions and declarations of variables in [Section 2.3](#) (p. 43). Class definitions were introduced in [Section 2.8](#) (p. 63) and will be covered in more detail in [Chapter 12](#).

在 C++ 中，对象或类的定义或声明也是语句。尽管定义语句这种说法也许更准确些，但定义语句经常被称为[声明语句](#)。[第 2.3 节](#)介绍了变量的定义和声明。[第 2.8 节](#)介绍了类的定义，相关内容将在[第十二章](#)进一步探讨。

6.3. Compound Statements (Blocks)

6.3. 复合语句 (块)

A **compound statement**, usually referred to as a **block**, is a (possibly empty) sequence of statements surrounded by a pair of curly braces. A block is a scope. Names introduced within a block are accessible only from within that block or from blocks nested inside the block. As usual, a name is visible only from its point of definition until the end of the enclosing block.

复合语句, 通常被称为**块**, 是用一对花括号括起来的语句序列 (也可能是空的)。块标识了一个作用域, 在块中引入的名字只能在该块内部或嵌套在块中的子块里访问。通常, 一个名字只从其定义处到该块的结尾这段范围内可见。

Compound statements can be used where the rules of the language require a single statement, but the logic of our program needs to execute more than one. For example, the body of a `while` or `for` loop must be a single statement. Yet, we often need to execute more than one statement in the body of a loop. We can do so by enclosing the statements in a pair of braces, thus turning the sequence of statements into a block.

复合语句用在语法规则要求使用单个语句但程序逻辑却需要不止一个语句的地方。例如, `while` 或 `for` 语句的循环体必须是单个语句。然而, 大多数情况都需要在循环体里执行多个语句。因而可使用一对花括号将语句序列括起来, 使其成为块语句。

As an example, recall the `while` loop from our solution to the bookstore problem on page 26:

回顾一下第 1.6 节处理书店问题的程序, 以其中用到的 `while` 循环为例:

```
// if so, read the transaction records
while (std::cin >> trans)
    if (total.same_isbn(trans))
        // match: update the running total
        total = total + trans;
    else {
        // no match: print & assign to total
        std::cout << total << std::endl;
        total = trans;
    }
```

In the `else` branch, the logic of our program requires that we print `total` and then reset it from `trans`. An `else` may be followed by only a single statement. By enclosing both statements in curly braces, we transform them into a single (com-pound) statement. This statement satisfies the rules of the language and the needs of our program.

在 `else` 分支中, 程序逻辑需要输出 `total` 的值, 然后用 `trans` 重置 `total`。但是, `else` 分支只能后接单个语句。于是, 用一对花括号将上述两条语句括起来, 使其在语法上成为单个语句 (复合语句)。这个语句既符合语法规则又满足程序的需要。



Unlike most other statements, a block is *not* terminated by a semicolon.

与其他大多数语句不同, 块并不是以分号结束的。

Just as there is a null statement, we also can define an empty block. We do so by using a pair of curlies with no statements:

像空语句一样, 程序员也可以定义空块, 用一对内部没有语句的花括号实现:

```
while (cin >> s && s != sought)
{ } // empty block
```

Exercises Section 6.3

Exercise What is a null statement? Give an example of when you might use a null statement.

6.1: 什么是空语句? 请给出一个使用空语句的例子。

Section 6.3. Compound Statements (Blocks)

Exercise What is a block? Give an example of when you might use a block.

6.2: 什么是块语句？请给出一个使用块的例子。

Exercise Use the comma operator ([Section 5.9](#), p. 168) to rewrite the `else` branch in the `while` loop from the bookstore problem so that it no longer requires a block. Explain whether this rewrite improves or diminishes the readability of this code.

使用逗号操作符（[第 5.9 节](#)）重写书店问题中 `while` 循环里的 `else` 分支，使它不再需要用块实现。解释一下重写后是提高还是降低了该段代码的可读性。

Exercise In the `while` loop that solved the bookstore problem, what effect, if any, would removing the curly brace following the `while` and its corresponding close curly have on the program?

在解决书店问题的 `while` 循环中，如果删去 `while` 后面的左花括号及相应的右花括号，将会给程序带来什么影响？

Team LiB

◀ PREVIOUS NEXT ▶

6.4. Statement Scope

6.4. 语句作用域

Some statements permit variable definitions within their control structure:

有些语句允许在它们的控制结构中定义变量：

```
while (int i = get_num())
    cout << i << endl;
i = 0; // error: i is not accessible outside the loop
```



Variables defined in a condition must be initialized. The value tested by the condition is the value of the initialized object.

在条件表达式中定义的变量必须初始化，该条件检验的就是初始化对象的值。

Variables defined as part of the control structure of a statement are visible only until the end of the statement in which they are defined. The scope of such variables is limited to the statement body. Often the statement body itself is a block, which in turn may contain other blocks. A name introduced in a control structure is local to the statement and the scopes nested inside the statement:

在语句的控制结构中定义的变量，仅在定义它们的块语句结束前有效。这种变量的作用域限制在语句体内。通常，语句体本身就是一个块语句，其中也可能包含了其他的块。一个在控制结构里引入的名字是该语句的局部变量，其作用域局限在语句内部。

```
// index is visible only within the for statement
for (vector<int>::size_type index = 0;
     index != vec.size(); ++index)
{ // new scope, nested within the scope of this for statement
    int square = 0;
    if (index % 2)                      // ok: index is in scope
        square = index * index;
    vec[index] = square;
}
if (index != vec.size()) // error: index is not visible here
```

If the program needs to access the value of a variable used in the control statement, then that variable must be defined outside the control structure:

如果程序需要访问某个控制结构中的变量，那么这个变量必须在控制语句外部定义。

```
vector<int>::size_type index = 0;
for (/* empty */; index != vec.size(); ++index)
    // as before
if (index != vec.size()) // ok: now index is in scope
    // as before
```



Earlier versions of C++ treated the scope of variables defined inside a `for` differently: Variables defined in the `for` header were treated as if they were defined just before the `for`. Older C++ programs may have code that relies on being able to access these control variables outside the scope of the `for`.

早期的 C++ 版本以不同的方式处理 `for` 语句中定义的变量的作用域：将 `for` 语句头定义的变量视为在 `for` 语句之前定义。有些更旧式的 C++ 程序代码允许在 `for` 语句作用域外访问控制变量。

One advantage of limiting the scope of variables defined within a control statement to that statement is that the names of such variables can be reused without worrying about whether their current value is correct at each use. If the name is not in scope, then it is impossible to use that

Section 6.4. Statement Scope

name with an incorrect, leftover value.

对于在控制语句中定义的变量，限制其作用域的一个好处是，这些变量名可以重复使用而不必担心它们的当前值在每一次使用时是否正确。对于作用域外的变量，是不可能用到其在作用域内的残留值的。

Team LiB

◀ PREVIOUS NEXT ▶

6.5. The `if` Statement

6.5. `if` 语句

An **`if statement`** conditionally executes another statement based on whether a specified expression is true. There are two forms of the `if`: one with an `else` branch and one without. The syntactic form of the plain `if` is the following:

`if` 语句根据特定表达式是否为真来有条件地执行另一个语句。`if` 语句有两种形式：其中一种带 `else` 分支而另一种则没有。根据语法结构，最简单的 `if` 语句是这样的：

```
if (condition)
    statement
```

The `condition` must be enclosed in parentheses. It can be an expression, such as

其中的 `condition` 部分必须用圆括号括起来。它可以是一个表达式，例如：

```
if (a + b > c) /* ... */
```

or an initialized declaration, such as

或者一个初始化声明，例如：

```
// ival only accessible within the if statement
if (int ival = compute_value()) /* ... */
```

As usual, `statement` could be a compound statement that is, a block of statements enclosed in curly braces.

通常，`statement` 部分可以是复合语句，即用花括号括起来的块语句。

When a condition defines a variable, the variable must be initialized. The value of the initialized variable is converted to `bool` (Section 5.12.3, p. 181) and the resulting `bool` determines the value of the condition. The variable can be of any type that can be converted to `bool`, which means it can be an arithmetic or pointer type. As we'll see in Chapter 14, whether a class type can be used in a condition depends on the class. Of the types we've used so far, the IO types can be used in a condition, but the `vector` and `string` types may not be used as a condition.

如果在条件表达式中定义了变量，那么变量必须初始化。将已初始化的变量值转换为 `bool` 值（第 5.12.3 节）后，该 `bool` 值决定条件是否成立。变量类型可以是任何可转换为 `bool` 型的类型，这意味着它可以是算术类型或指针类型。正如第十四章要提及的，一个类类型能否用在条件表达式中取决于类本身。迄今为止，在所有用过的类类型中，IO 类型可以用作条件，但 `vector` 类型和 `string` 类型一般不可用作条件。

To illustrate the use of the `if` statement, we'll find the smallest value in a `vector<int>`, keeping a count of how many times that minimum value occurs. To solve this problem, we'll need two `if` statements: one to determine whether we have a new minimum and the other to increment a count of the number of occurrences of the current minimum value:

为了说明 `if` 语句的用法，下面程序用于寻找 `vector<int>` 对象中的最小值，并且记录这个最小值出现的次数。为了解决这个问题，需要两个 `if` 语句：一个判断是否得到一个新的最小值，而另一个则用来增加当前最小值的数目。

```
if (minVal > ivec[i]) { /* process new minVal */ }
if (minVal == ivec[i]) { /* increment occurrence count */ }
```

Statement Block as Target of an `if`

语句块用作 `if` 语句的对象

We'll start by considering each `if` in isolation. One of these `if` statements will determine whether there is a new minimum and, if so, reset the counter and update `minVal`:

现在单独考虑上述例子中的每个 `if` 语句。其中一个 `if` 语句将要决定是否出现了一个新的最小值，如果是的话，则要重置计数器并更新最小值：

```
if (minVal > ivec[i]) { // execute both statements if condition is true
    minVal = ivec[i];
    occurs = 1;
```

Section 6.5. The if Statement

}

The other conditionally updates the counter. This `if` needs only one statement, so it need not be enclosed in curly braces:

另一个 `if` 语句则有条件地更新计数器，它只需要一个语句，因此不必用花括号起来：

```
if (minVal == ivec[i])
    ++occurs;
```



It is a somewhat common error to forget the curly braces when multiple statements must be executed as a single statement.

当多个语句必须作为单个语句执行时，比较常见的错误是漏掉了花括号。

In the following program, contrary to the indentation and intention of the programmer, the assignment to `occurs` is not part of the `if` statement:

在下面的程序中，与程序员缩进目的相反，对 `occurs` 的赋值并不是 `if` 语句的一部分：

```
// error: missing curly brackets to make a block!
if (minVal > ivec[i])
    minVal = ivec[i];
    occurs = 1; // executed unconditionally: not part of the if
```

Written this way, the assignment to `occurs` will be executed unconditionally. Uncovering this kind of error can be very difficult because the text of the program looks correct.

这样写的话，对 `occurs` 的赋值将会无条件地执行。这种错误很难发现，因为程序代码看起来是正确的。



Many editors and development environments have tools to automatically indent source code to match its structure. It is a good idea to use such tools if they are available.

很多编辑器和开发环境都是提供工具自动根据语句结构缩排源代码。有效地利用这些工具将是一种很好的编程方法。

6.5.1. The `if` Statement `else` Branch

6.5.1.1. `if` 语句的 `else` 分支

Our next task is to put these `if` statements together into an execution sequence. The order of the `if` statements is significant. If we use the following order

紧接着，我们要考虑如何将那些 `if` 语句放在一起形成一个执行语句序列。这些 `if` 语句的排列顺序非常重要。如果采用下面的顺序：

```
if (minVal > ivec[i]) {
    minVal = ivec[i];
    occurs = 1;
}
// potential error if minVal has just been set to ivec[i]
if (minVal == ivec[i])
    ++occurs;
```

our count will always be off by 1. This code double-counts the first occurrence of the minimum.

那么计数器将永远得不到 1。这段代码只是对第一次出现的最小值重复计数。

Section 6.5. The if Statement

Not only is the execution of both `if` statements on the same value potentially dangerous, it is also unnecessary. The same element cannot be both less than `minVal` and equal to it. If one condition is true, the other condition can be safely ignored. The `if` statement allows for this kind of *either-or* condition by providing an `else` clause.

这样两个 `if` 语句不但在值相同时执行起来有潜在的危险，而且还是没必要的。同一个元素不可能既小于 `minVal` 又等于它。如果其中一个条件是真的，那么另一个条件就可以安全地忽略掉。`if` 语句为这种只能二选一的条件提供了 `else` 子句。

The syntactic form of the [if else statement](#) is

[if else 语句](#)的语法形式为：

```
if (condition)
    statement1
else
    statement2
```

If `condition` is true, then `statement1` is executed; otherwise, `statement2` is executed:

如果 `condition` 为真，则执行 `statement1`; 否则，执行 `statement2`:

```
if (minVal == ivec[i])
    ++occurs;
else if (minVal > ivec[i]) {
    minVal = ivec[i];
    occurs = 1;
}
```

It is worth noting that `statement2` can be any statement or a block of statements enclosed in curly braces. In this example, `statement2` is itself an `if` statement.

值得注意的是，`statement2` 既可以是任意语句，也可以是用花括号起来的块语句。在这个例子里，`statement2` 本身是一个 `if` 语句。

Dangling `else`

悬垂 `else`

There is one important complexity in using `if` statements that we have not yet covered. Notice that neither `if` directly handles the case where the current element is greater than `minVal`. Logically, ignoring these elements is finethere is nothing to do if the element is greater than the minimum we've found so far. However, it is often the case that an `if` needs to do something on all three cases: Unique steps may be required if one value is greater than, less than, or equal to some other value. We've rewritten our loop to explicitly handle all three cases:

对于 `if` 语句的使用，还有一个重要的复杂问题没有考虑。上述例子中，注意到没有一个 `if` 分支能直接处理元素值大于 `minVal` 的情况。从逻辑上来说，可以忽略这些元素——如果该元素比当前已找到的最小值大，那就应该没什么要做的。然而，通常需要使用 `if` 语句为三种不同情况提供执行的内容，即如果一个值大于、小于或等于其他值时，可能都需要执行特定的步骤。为此重写循环部分，显式地处理这三种情况：

```
// note: indented to make clear how the else branches align with the corresponding if
if (minVal < ivec[i])
    {}
else if (minVal == ivec[i]) // empty block
else { // minVal > ivec[i]
    minVal = ivec[i];
    occurs = 1;
}
```

This three-way test handles each case correctly. However, a simple rewrite that collapses the first two tests into a single, nested `if` runs into problems:

上述的三路测试精确地控制了所有的情况。然而，简单地把前两个情况用一个嵌套 `if` 语句实现将会产生问题：

```
// oops: incorrect rewrite: This code won't work!
if (minVal <= ivec[i])
    if (minVal == ivec[i])
        ++occurs;
else { // this else goes with the inner if, not the outer one!
    minVal = ivec[i];
    occurs = 1;
}
```

This version illustrates a source of potential ambiguity common to `if` statements in all languages. The



problem, usually referred to as the **dangling-else** problem, occurs when a statement contains more `if` clauses than `else` clauses. The question then arises: To which `if` does each `else` clause belong?

这个版本表明了所有语言的 `if` 语句都普通存在着潜在的二义性。这种情况往往称为**悬垂 else** 问题，产生于一个语句包含的 `if` 子句多于 `else` 子句时：对于每一个 `else`，究竟它们归属哪个 `if` 语句？

The indentation in our code indicates the expectation that the `else` should match up with the outer `if` clause. In C++, however, the dangling-else ambiguity is resolved by matching the `else` with the last occurring unmatched `if`. In this case, the actual evaluation of the `if else` statement is as follows:

在上述的代码中，缩进的用法表明 `else` 应该与外层的 `if` 子句匹配。然而，C++ 中悬垂 `else` 问题带来的二义性，通过将 `else` 匹配给最后出现的尚未匹配的 `if` 子句来解决。在这个情况下，这个 `if else` 语句实际上等价于下面的程序：

```
// oops: still wrong, but now the indentation matches execution path
if (minVal <= ivect[i])
    // indented to match handling of dangling-else
    if (minVal == ivect[i])
        ++occurs;
    else {
        minVal = ivect[i];
        occurs = 1;
    }
```

We can force an `else` to match an outer `if` by enclosing the inner `if` in a compound statement:

可以通过用花括号将内层的 `if` 语句括起来成为复合语句，从而迫使这个 `else` 子句与外层的 `if` 匹配。

```
if (minVal <= ivect[i]) {
    if (minVal == ivect[i])
        ++occurs;
} else {
    minVal = ivect[i];
    occurs = 1;
}
```



Some coding styles recommend *always* using braces after any `if`. Doing so avoids any possible confusion and error in later modifications of the code. At a minimum, it is nearly always a good idea to use braces after an `if` (or `while`) when the statement in the body is anything other than a simple expression statement, such as an assignment or output expression.

有些编程风格建议总是在 `if` 后面使用花括号。这样做可以避免日后修改代码时产生混乱和错误。至少，无论 `if` (或者 `while`) 后面是简单语句，例如赋值和输出语句，还是其他任意语句，使用花括号都是一个比较好的做法。

Exercises Section 6.5.1

Exercise Correct each of the following:

6.5:

改正下列代码：

- (a) `if (ival1 != ival2)
 ival1 = ival2
else ival1 = ival2 = 0;`
- (b) `if (ival < minval)
 minval = ival; // remember new minimum
 occurs = 1; // reset occurrence counter`
- (c) `if (int ival = get_value())
 cout << "ival = " << ival << endl;
if (!ival)
 cout << "ival = 0\n";`
- (d) `if (ival = 0)
 ival = get_value();`

Exercise What is a "dangling else"? How are `else` clauses resolved in C++?

6.6:

什么是“悬垂 `else`”？C++ 是如何匹配 `else` 子句的？

Team LiB

◀ PREVIOUS NEXT ▶

6.6. The `switch` Statement

6.6. `switch` 语句

Deeply nested `if else` statements can often be correct syntactically and yet not correctly reflect the programmer's logic. For example, mistaken `else if` matchings are more likely to pass unnoticed. Adding a new condition and associated logic or making other changes to the statements is also hard to get right. A [switch statement](#) provides a more convenient way to write deeply nested `if/else` logic.

深层嵌套的 `if else` 语句往往在语法上是正确的，但逻辑上去却没有正确地反映程序员的意图。例如，错误的 `else if` 匹配很容易被忽略。添加新的条件和逻辑关系，或者对语句做其他修改，都很难保证正确。[switch 语句](#)提供了一种更方便的方法来实现深层嵌套的 `if/else` 逻辑。

Suppose that we have been asked to count how often each of the five vowels appears in some segment of text. Our program logic is as follows:

假设要统计五个元音在文本里分别出现的次数，程序逻辑结构如下：

- Read each character until there are no more characters to read
按顺序读入每个字符直到读入完成为止。
- Compare each character to the set of vowels
把每个字符与元音字符集做比较。
- If the character matches one of the vowels, add 1 to that vowel's count
如果该字符与某个元音匹配，则该元音的计数器加 1。
- Display the results
显示结果。

The program was used to analyze this chapter. Here is the output:

使用此程序分析本章的英文版，得到以下输出结果：

```
Number of vowel a: 3499
Number of vowel e: 7132
Number of vowel i: 3577
Number of vowel o: 3530
Number of vowel u: 1185
```

6.6.1. Using a `switch`

6.6.1. 使用 `switch`

We can solve our problem most directly using a `switch` statement:

直接使用 `switch` 语句解决上述问题：

```
char ch;
// initialize counters for each vowel
int aCnt = 0, eCnt = 0, iCnt = 0,
    oCnt = 0, uCnt = 0;
while (cin >> ch) {
    // if ch is a vowel, increment the appropriate counter
    switch (ch) {
        case 'a':
            ++aCnt;
            break;
        case 'e':
            ++eCnt;
            break;
        case 'i':
            ++iCnt;
            break;
        case 'o':
            ++oCnt;
            break;
        case 'u':
            ++uCnt;
            break;
    }
}
```

Section 6.6. The switch Statement

```
        break;
    case 'u':
        ++uCnt;
        break;
}
// print results
cout << "Number of vowel a: \t" << aCnt << '\n'
<< "Number of vowel e: \t" << eCnt << '\n'
<< "Number of vowel i: \t" << iCnt << '\n'
<< "Number of vowel o: \t" << oCnt << '\n'
<< "Number of vowel u: \t" << uCnt << endl;
```

A `switch` statement executes by evaluating the parenthesized expression that follows the keyword `switch`. That expression must yield an integral result. The result of the expression is compared with the value associated with each `case`. The `case` keyword and its associated value together are known as the **case label**. Each case label's value must be a constant expression (Section 2.7, p. 62). There is also a special case label, the `default` label, which we cover on page 203.

通过对圆括号内表达式的值与其后列出的关键字做比较，实现 `switch` 语句的功能。表达式必须产生一个整数结果，其值与每个 `case` 的值比较。关键字 `case` 和它所关联的值称为 **case 标号**。每个 `case` 标号的值都必须是一个常量表达式（第 2.7 节）。除此之外，还有一个特殊的 `case` 标号——`default` 标号，我们将在第 6.6.3 节介绍它。

If the expression matches the value of a `case` label, execution begins with the first statement following that label. Execution continues normally from that statement through the end of the `switch` or until a `break` statement. If no match is found, (and if there is no `default` label), execution falls through to the first statement following the `switch`. In this program, the `switch` is the only statement in the body of a `while`. Here, falling through the `switch` returns control to the `while` condition.

如果表达式与其中一个 `case` 标号的值匹配，则程序将从该标号后面的第一个语句开始依次执行各个语句，直到 `switch` 结束或遇到 `break` 语句为止。如果没有发现匹配的 `case` 标号（并且也没有 `default` 标号），则程序从 `switch` 语句后面的第一条继续执行。在这个程序中，`switch` 语句是 `while` 循环体中唯一的语句，于是，`switch` 语句匹配失败后，将控制流返回给 `while` 循环条件。

We'll look at `break` statements in Section 6.10 (p. 212). Briefly, a `break` statement interrupts the current control flow. In this case, the `break` transfers control out of the `switch`. Execution continues at the first statement following the `switch`. In this example, as we already know, transferring control to the statement following the `switch` returns control to the `while`.

第 6.10 节将讨论 `break` 语句。简单地说，`break` 语句中断当前的控制流。对于 `switch` 的应用，`break` 语句将控制跳出 `switch`，继续执行 `switch` 语句后面的第一个语句。在这个例子中，正如大家所知道的，将会把控制转移到 `switch` 后面的下一语句，即交回给 `while`。

6.6.2. Control Flow within a `switch`

6.6.2. `switch` 中的控制流

It is essential to understand that execution flows across `case` labels.

了解 `case` 标号的执行流是必要的。



It is a common misunderstanding to expect that only the statements associated with the matched `case` label are executed. However, execution continues across `case` boundaries until the end of the `switch` statement or a `break` is encountered.

存在一个普遍的误解：以为程序只会执行匹配的 `case` 标号相关联的语句。实际上，程序从该点开始执行，并跨越 `case` 边界继续执行其他语句，直到 `switch` 结束或遇到 `break` 语句为止。

Sometimes this behavior is indeed correct. We want to execute the code for a particular label as well as the code for following labels. More often, we want to execute only the code particular to a given label. To avoid executing code for subsequent cases, the programmer must explicitly tell the compiler to stop execution by specifying a `break` statement. Under most conditions, the last statement before the next `case` label is `break`. For example, here is an incorrect implementation of our vowel-counting `switch` statement:

有时候，这种行为的确是正确的。程序员也许希望执行完某个特定标号的代码后，接着执行后续标号关联的语句。但更常见的是，我们只需要执行某个特定标号对应的代码。为了避免继续执行其后续 `case` 标号的内容，程序员必须利用 `break` 语句清楚地告诉编译器停止执行 `switch` 中的语句。大多数情况下，在下一个 `case` 标号之前的最后一条语句是 `break`。例如，下面统计元音出现次数的 `switch` 语句是不正确的：

```
// warning: deliberately incorrect!
switch (ch) {
    case 'a':
        ++aCnt; // oops: should have a break statement
    case 'e':
        ++eCnt; // oops: should have a break statement
    case 'i':
        ++iCnt; // oops: should have a break statement
```

Section 6.6. The switch Statement

```
case 'o':  
    ++oCnt; // oops: should have a break statement  
case 'u':  
    ++uCnt; // oops: should have a break statement  
}
```

To understand what happens, we'll trace through this version assuming that value of `ch` is '`i`'. Execution begins following `case 'i'` thus incrementing `iCnt`. Execution does not stop there but continues across the `case` labels incrementing `oCnt` and `uCnt` as well. If `ch` had been '`e`', then `eCnt`, `iCnt`, `oCnt`, and `uCnt` would all be incremented.

为了搞清楚该程序导致了什么结果，假设 `ch` 的值是 '`i`' 来跟踪这个版本的代码。程序从 `case 'i'` 后面的语句开始执行，`iCnt` 的值加 1。但是，程序的执行并没有在这里停止，而是越过 `case` 标号继续执行，同时将 `oCnt` 和 `uCnt` 的值都加了 1。如果 `ch` 是 '`e`' 的话，那么 `eCnt`、`iCnt`、`oCnt` 以及 `uCnt` 的值都会加 1。



Forgetting to provide a `break` is a common source of bugs in `switch` statements.

对于 `switch` 结构，漏写 `break` 语句是常见的程序错误。



Although it is not strictly necessary to specify a `break` statement after the last label of a `switch`, the safest course is to provide a `break` after every label, even the last. If an additional `case` label is added later, then the `break` is already in place.

尽管没有严格要求在 `switch` 结构的最后一个标号之后指定 `break` 语句，但是，为了安全起见，最好在每个标号后面提供一个 `break` 语句，即使是最后一个标号也一样。如果以后在 `switch` 结构的末尾又需要添加一个新的 `case` 标号，则不用再在前面加 `break` 语句了。

`break` Statements Aren't Always Appropriate

慎用 `break` 语句，它并不总是恰当的

There is one common situation where the programmer might wish to omit a `break` statement from a `case` label, allowing the program to *fall through* multiple case labels. That happens when two or more values are to be handled by the same sequence of actions. Only a single value can be associated with a case label. To indicate a range, therefore, we typically stack case labels following one another. For example, if we wished only to count vowels seen rather than count the individual vowels, we might write the following:

有这么一种常见的情况，程序员希望在 `case` 标号后省略 `break` 语句，允许程序向下执行多个 `case` 标号。这时，两个或多个 `case` 值由相同的作用序列来处理。由于系统限制一个 `case` 标号只能与一个值相关联，于是为了指定一个范围，典型的做法是，把 `case` 标号依次排列。例如，如果只希望计算文本中元音的总数，而不是每一个元音的个数，则可以这样写：

```
int vowelCnt = 0;  
// ...  
switch (ch)  
{  
    // any occurrence of a,e,i,o,u increments vowelCnt  
    case 'a':  
    case 'e':  
    case 'i':  
    case 'o':  
    case 'u':  
        ++vowelCnt;  
        break;  
}
```

Case labels need not appear on a new line. We could emphasize that the cases represent a range of values by listing them all on a single line:

每个 `case` 标号不一定要另起一行。为了强调这些 `case` 标号表示的是一个要匹配的范围，可以将它们全部在一行中列出：

```
switch (ch)  
{  
    // alternative legal syntax  
    case 'a': case 'e': case 'i': case 'o': case 'u':
```

Section 6.6. The switch Statement

```
    ++vowelCnt;  
    break;  
}
```

Less frequently, we deliberately omit a `break` because we want to execute code for one case and then continue into the next case, executing that code as well.

比较少见的用法是，为了执行某个 `case` 的代码后继续执行下一个 `case` 的代码，故意省略 `break` 语句。



Deliberately omitting a `break` at the end of a case happens rarely enough that a comment explaining the logic should be provided.

故意省略 `case` 后面的 `break` 语句是很罕见的，因此应该提供一些注释说明其逻辑。

6.6.3. The `default` Label

6.6.3. `default` 标号

The `default label` provides the equivalent of an `else` clause. If no `case` label matches the value of the switch expression and there is a `default` label, then the statements following the `default` are executed. For example, we might add a counter to track how many nonvowels we read. We'll increment this counter, which we'll name `otherCnt`, in the `default` case:

`default` 标号提供了相当于 `else` 子句的功能。如果所有的 `case` 标号与 `switch` 表达式的值都不匹配，并且 `default` 标号存在，则执行 `default` 标号后面的语句。例如，在上述例子中添加一个计数器 `otherCnt` 统计读入多少个辅音字母，为 `switch` 结构增加 `default` 标号，其标志的分支实现 `otherCnt` 的自增：

```
// if ch is a vowel, increment the appropriate counter  
switch (ch) {  
    case 'a':  
        ++aCnt;  
        break;  
    // remaining vowel cases as before  
    default:  
        ++otherCnt;  
        break;  
}
```

In this version, if `ch` is not a vowel, execution will fall through to the `default` label, and we'll increment `otherCnt`.

在这个版本的代码中，如果 `ch` 不是元音，程序流程将执行 `default` 标号的相关语句，使 `otherCnt` 的值加 1。



It can be useful always to define a `default` label even if there is no processing to be done in the `default` case. Defining the label indicates to subsequent readers that the case was considered but that there is no work to be done.

哪怕没有语句要在 `default` 标号下执行，定义 `default` 标号仍然是有用的。定义 `default` 标号是为了告诉它的读者，表明这种情况已经考虑到了，只是没什么要执行的。

A label may not stand alone; it must precede a statement. If a `switch` ends with the `default` case in which there is no work to be done, then the `default` label must be followed by a null statement.

一个标号不能独立存在，它必须位于语句之前。如果 `switch` 结构以 `default` 标号结束，而且 `default` 分支不需要完成任何任务，那么该标号后面必须有一个空语句。

6.6.4. `switch` Expression and Case Labels

6.6.4. `switch` 表达式与 `case` 标号

Section 6.6. The switch Statement

The expression evaluated by a `switch` can be arbitrarily complex. In particular, the expression can define and initialize a variable:

`switch` 求解的表达式可以非常复杂。特别是，该表达式也可以定义和初始化一个变量：

```
switch(int ival = get_response())
```

In this case, `ival` is initialized, and the value of `ival` is compared with each `case` label. The variable `ival` exists throughout the entire `switch` statement but not outside it.

在这个例子中，`ival` 被初始化为 `get_response` 函数的调用结果，其值将要与每个 `case` 标号作比较。变量 `ival` 始终存在于整个 `switch` 语句中，在 `switch` 结构外面该变量就不再有效了。

Case labels must be constant integral expressions ([Section 2.7](#), p. 62). For example, the following labels result in compile-time errors:

`case` 标号必须是整型常量表达式 ([第 2.7 节](#))。例如，下面的标号将导致编译时的错误：

```
// illegal case label values
case 3.14: // noninteger
case ival: // nonconstant
```

It is also an error for any two case labels to have the same value.

如果两个 `case` 标号具有相同的值，同样也会导致编译时的错误。

6.6.5. Variable Definitions inside a `switch`

6.6.5. `switch` 内部的变量定义

Variables can be defined following only the last `case` or `default` label:

对于 `switch` 结构，只能在它的最后一个 `case` 标号或 `default` 标号后面定义变量：

```
case true:
    // error: declaration precedes a case label
    string file_name = get_file_name();
    break;
case false:
    // ...
```

The reason for this rule is to prevent code that might jump over the definition and initialization of a variable.

制定这个规则是为避免出现代码跳过变量的定义和初始化的情况。

Recall that a variable can be used from its point of definition until the end of the block in which it is defined. Now, consider what would happen if we could define a variable between two case labels. That variable would continue to exist until the end of the enclosing block. It could be used by code in case labels following the one in which it was defined. If the `switch` begins executing in one of these subsequent case labels, then the variable might be used even though it had not been defined.

回顾变量的作用域，变量从它的定义点开始有效，直到它所在块结束为止。现在考虑如果在两个 `case` 标号之间定义变量会出现什么情况。该变量会在块结束之前一直存在。对于定义该变量的标号后面的其他 `case` 标号，它们所关联的代码都可以使用这个变量。如果 `switch` 从那些后续 `case` 标号开始执行，那么这个变量可能还未定义就要使用了。

If we need to define a variable for a particular case, we can do so by defining the variable inside a block, thereby ensuring that the variable can be used only where it is guaranteed to have been defined and initialized:

在这种情况下，如果需要为某个特殊的 `case` 定义变量，则可以引入块语句，在该块语句中定义变量，从而保证这个变量在使用前被定义和初始化。

```
case true:
{
    // ok: declaration statement within a statement block
    string file_name = get_file_name();
    // ...
}
break;
case false:
// ...
```

Exercises Section 6.6.5

Exercise

- 6.7:** There is one problem with our vowel-counting program as we've implemented it: It doesn't count capital letters as vowels. Write a program that counts both lower- and uppercase letters as the appropriate vowel—that is, your program should count both '`a`' and '`A`' as part of `aCnt`, and so forth.

前面已实现的统计元音的程序存在一个问题：不能统计大写的元音字母。编写程序统计大小写的元音，也就是说，你的程序计算出来的 `aCnt`，既包括 '`a`' 也包括 '`A`' 出现的次数，其他四个元音也一样。

Exercise

- 6.8:** Modify our vowel-count program so that it also counts the number of blank spaces, tabs, and newlines read.

修改元音统计程序使其可统计出读入的空格、制表符和换行符的个数。

Exercise

- 6.9:** Modify our vowel-count program so that it counts the number of occurrences of the following two-character sequences: `ff`, `fl`, and `fi`.

修改元音统计程序使其可统计以下双字符序列出现的次数：`ff`、`fl` 以及 `fi`。

Exercise

- 6.10:** Each of the programs in the highlighted text on page [206](#) contains a common programming error. Identify and correct each error.

下面每段代码都暴露了一个常见编程错误。请指出并修改之。

Code for Exercises in Section 6.6.5

```
(a) switch (ival) {
    case 'a': aCnt++;
    case 'e': eCnt++;
    default: iouCnt++;
}

(b) switch (ival) {
    case 1:
        int ix = get_value();
        ivec[ ix ] = ival;
        break;
    default:
        ix = ivec.size()-1;
        ivec[ ix ] = ival;
}

(c) switch (ival) {
    case 1, 3, 5, 7, 9:
        oddcnt++;
        break;
    case 2, 4, 6, 8, 10:
        evencnt++;
        break;
}

(d) int ival=512 jval=1024, kval=4096;
int bufsize;
// ...
switch(swt) {
    case ival:
        bufsize = ival * sizeof(int);
        break;
    case jval:
        bufsize = jval * sizeof(int);
        break;
    case kval:
        bufsize = kval * sizeof(int);
        break;
}
```

6.7. The `while` Statement

6.7. `while` 语句

A **`while` statement** repeatedly executes a target statement as long as a condition is true. Its syntactic form is

当条件为真时，`while` 语句反复执行目标语句。它的语法形式如下：

```
while (condition)
    statement
```

The *statement* (which is often a block) is executed as long as the *condition* evaluates as `true`. The *condition* may not be empty. If the first evaluation of *condition* yields `false`, *statement* is not executed.

只要条件 *condition* 的值为 `true`，执行语句 *statement* (通常是一个块语句)。*condition* 不能为空。如果第一次求解 *condition* 就产生 `false` 值，则不执行 *statement*。

The condition can be an expression or an initialized variable definition:

循环条件 *condition* 可以是一个表达式，或者是提供初始化的变量定义。

```
bool quit = false;
while (!quit) { // expression as condition
    quit = do_something();
}
while (int loc = search(name)) { // initialized variable as condition
    // do something
}
```

Any variable defined in the condition is visible only within the block associated with the `while`. On each trip through the loop, the initialized value is converted to `bool` ([Section 5.12.3](#), p. 182). If the value evaluates as `true`, the `while` body is executed. Ordinarily, the condition itself or the loop body must do something to change the value of the expression. Otherwise, the loop might never terminate.

在循环条件中定义的任意变量都只在与 `while` 关联的块语句中可见。每一次循环都将该变量的初值转换为 `bool` ([第 5.12.3 节](#))。如果求得的值为 `true`，则执行 `while` 的循环体。通常，循环条件自身或者在循环体内必须做一些相关操作来改变循环条件表达式的值。否则，循环可能永远不会结束。



Variables defined in the condition are created and destroyed on each trip through the loop.

在循环条件中定义的变量在每次循环里都要经历创建和撤销的过程。

Using a `while` Loop

`while` 循环的使用

We have already seen a number of `while` loops, but for completeness, here is an example that copies the contents of one array into another:

前面的章节已经用过很多 `while` 循环，但为更完整地了解该结构，考虑下面将一个数组的内容复制到另一个数组的例子：

```
// arr1 is an array of ints
int *source = arr1;
size_t sz = sizeof(arr1)/sizeof(*arr1); // number of elements
int *dest = new int[sz]; // uninitialized elements
while (source != arr1 + sz)
    *dest++ = *source++; // copy element and increment pointers
```

We start by initializing `source` and `dest` to point to the first element of their respective arrays. The condition in the `while` tests whether we've

Section 6.7. The while Statement

reached the end of the array from which we are copying. If not, we execute the body of the loop. The body contains only a single statement, which copies the element and increments both pointers so that they point to the next element in their corresponding arrays.

首先初始化 `source` 和 `dest`, 并使它们各自指向所关联的数组的第一个元素。`while` 循环条件判断是否已经到达要复制的数组的末尾。如果没有, 继续执行循环。循环体只有单个语句, 实现元素的复制, 并对两个指针做自增操作, 使它们指向对应数组的下一个元素。

As we saw in the "Advice" box on page 164, C++ programmers tend to write terse expressions. The statement in the body of the `while`

正如 第 5.5. 节提出的关于“简洁即是美”的建议, C++ 程序员应尝试编写简洁的表达式。`while` 循环体中的语句:

```
*dest++ = *source++;
```

is a classic example. This expression is equivalent to

是一个经典的例子。这个表达式等价于:

```
{  
    *dest = *source; // copy element  
    ++dest; // increment the pointers  
    ++source;  
}
```



The assignment in the `while` loop represents a very common usage. Because such code is widespread, it is important to study this expression until its meaning is immediately clear.

`while` 循环内的赋值操作是一种常见的用法。因为这类代码广为流传, 所以学习这种表达式非常重要, 要一眼就能看出其含义来。

Exercises Section 6.7

Exercise

6.11: Explain each of the following loops. Correct any problems you detect.

解释下面的循环, 更正你发现的问题。

```
(a) string bufString, word;  
    while (cin >> bufString >> word) { /* ... */ }  
  
(b) while (vector<int>::iterator iter != ivec.end())  
    { /*...*/ }  
  
(c) while (ptr == 0)  
    ptr = find_a_value();  
  
(d) while (bool status = find(word))  
    { word = get_next_word(); }  
    if (!status)  
        cout << "Did not find any words\n";
```

Exercise

6.12: Write a small program to read a sequence of `strings` from standard input looking for duplicated words. The program should find places in the input where one word is followed immediately by itself. Keep track of the largest number of times a single repetition occurs and which word is repeated. Print the maximum number of duplicates, or else print a message saying that no word was repeated. For example, if the input is

编写一个小程序, 从标准输入读入一系列 `string` 对象, 寻找连续重复出现的单词。程序应该找出满足以下条件的单词的输入位置: 该单词的后面紧跟着再次出现自己本身。跟踪重复次数最多的单词及其重复次数。输出重复次数的最大值, 若没有单词重复则输出说明信息。例如, 如果输入是:

```
how, now now now brown cow cow
```

the output should indicate that the word "now" occurred three times.

则输出应表明“now”这个单词出现了三次。

Exercise

Explain in detail how the statement in the `while` loop is executed:

6.13:

详细解释下面 `while` 循环中的语句是如何执行的：

```
*dest++ = *source++;
```

Team LiB

◀ PREVIOUS NEXT ▶

6.8. The `for` Loop Statement

6.8. `for` 循环语句

The syntactic form of a `for` statement is

`for` 语句的语法形式是：

```
for (init-statement condition; expression)
      statement
```

The `init-statement` must be a declaration statement, an expression statement, or a null statement. Each of these statements is terminated by a semicolon, so the syntactic form can also be thought of as

`init-statement` 必须是声明语句、表达式语句或空语句。这些语句都以分号结束，因此其语法形式也可以看成：

```
for (initializer; condition; expression)
      statement
```

although technically speaking, the semicolon after the `initializer` is part of the statement that begins the `for` header.

当然，从技术上说，在 `initializer` 后面的分号是 `for` 语句头的一部分。

In general, the `init-statement` is used to initialize or assign a starting value that is modified over the course of the loop. The `condition` serves as the loop control. As long as `condition` evaluates as true, `statement` is executed. If the first evaluation of `condition` evaluates to `false`, `statement` is not executed. The `expression` usually is used to modify the variable(s) initialized in `init-statement` and tested in `condition`. It is evaluated after each iteration of the loop. If `condition` evaluates to false on the first iteration, `expression` is never executed. As usual, `statement` can be either a single or a compound statement.

一般来说，`init-statement` 用于对每次循环过程中都要修改的变量进行初始化，或者赋给一个起始值。而 `condition` 则是用来控制循环的。当 `condition` 为 `true` 时，循环执行 `statement`。如果第一次求解 `condition` 就得 `false` 值，则不执行 `statement`。`expression` 通常用于修改在 `init-statement` 中初始化并在 `condition` 中检查的变量。它在每次循环迭代后都要求解。如果第一次求解 `condition` 就得 `false` 值，则始终不执行 `expression`。通常，`statement` 既可以是单个语句也可以是复合语句。

Using a `for` Loop

`for` 循环的使用

Given the following `for` loop, which prints the contents of a `vector`,

假设有下面的 `for` 循环，用于输出一个 `vector` 对象的内容：

```
for (vector<string>::size_type ind = 0;
     ind != svec.size(); ++ind) {
    cout << svec[ind]; // print current element
    // if not the last element, print a space to separate from the next one
    if (ind + 1 != svec.size())
        cout << " ";
}
```

the order of evaluation is as follows:

它的计算顺序如下：

1. The `init-statement` is executed once at the start of the loop. In this example, `ind` is defined and initialized to zero.

循环开始时，执行一次 `init-statement`。在这个例子中，定义了 `ind`，并将它初始化为 0。

2. Next, `condition` is evaluated. If `ind` is not equal to `svec.size()`, then the `for` body is executed. Otherwise, the loop terminates. If the condition is false on the first trip, then the `for` body is not executed.

接着，求解 `condition`。如果 `ind` 不等于 `svec.size()`，则执行 `for` 循环体。否则，循环结束。如果在第一次循环时，条件就为 `false`，则不执行 `for` 循环体。

3. If the condition is true, the `for` body executes. In this case, the `for` body prints the current element and then tests whether this element is

Section 6.8. The for Loop Statement

the last one. If not, it prints a space to separate it from the next element.

如果条件为 `true`, 则执行 `for` 循环体。本例中, `for` 循环体输出当前元素值, 并检验这个元素是否是最后一个。如果不是, 则输出一个空格, 用于分隔当前元素和下一个元素。

4. Finally, `expression` is evaluated. In this example, `ind` is incremented by 1.

最后, 求解 `expression`。本例中, `ind` 自增 1。

These four steps represent the first iteration of the `for` loop. Step 2 is now repeated, followed by steps 3 and 4, until the condition evaluates to `false` that is, when `ind` is equal to `svec.size()`.

这四步描述了 `for` 循环的第一次完整迭代。接着重复第 2 步, 然后是第 3、4 步, 直到 `condition` 的值为 `false`, 即 `ind` 等于 `svec.size()` 为止。



It is worth remembering that the visibility of any object defined within the `for` header is limited to the body of the `for` loop. Thus, in this example, `ind` is inaccessible after the `for` completes.

应该谨记: 在 `for` 语句头定义的任何对象只限制在 `for` 循环体里可见。因此, 对本例而言, 在执行完 `for` 语句后, `ind` 不再有效 (即不可访问)。

6.8.1. Omitting Parts of the `for` Header

6.8.1. 省略 `for` 语句头的某些部分

A `for` header can omit any (or all) of `init-statement`, `condition`, or `expression`.

`for` 语句头中, 可以省略 `init-statement`、`condition` 或者 `expression` (表达式) 中的任何一个 (或全部)。

The `init-statement` is omitted if an initialization is unnecessary or occurs elsewhere. For example, if we rewrote the program to print the contents of a `vector` using iterators instead of subscripts, we might, for readability reasons, move the initialization outside the loop:

如果不需初始化或者初始化已经在别处实现了, 则可以省略 `init-statement`。例如, 使用迭代器代替下标重写输出 `vector` 对象内容的程序, 为了提高易读性, 可将初始化移到循环外面:

```
vector<string>::iterator iter = svec.begin();
for( /* null */ ; iter != svec.end(); ++iter) {
    cout << *iter; // print current element
    // if not the last element, print a space to separate from the next one
    if (iter+1 != svec.end())
        cout << " ";
}
```

Note that the semicolon is necessary to indicate the absence of the `init-statement` more precisely, the semicolon represents a null `init-statement`.

注意此时必须要有一个分号表明活力了 `init-statement`——更准确地说, 分号代表一个空的 `init-statement`。

If the `condition` is omitted, then it is equivalent to having written `true` as the condition:

省略 `condition`, 则等效于循环条件永远为 `true`:

```
for (int i = 0; /* no condition */ ; ++i)
```

It is as if the program were written as

相当于程序写为:

```
for (int i = 0; true ; ++i)
```

It is essential that the body of the loop contain a `break` or `return` statement. Otherwise the loop will execute until it exhausts the system resources. Similarly, if the `expression` is omitted, then the loop must exit through a `break` or `return` or the loop body must arrange to change the value tested in the condition:

这么一来, 循环体内就必须包含一个 `break` 或者 `return` 语句。否则, 循环会一直执行直到耗尽系统的资源为止。同样地, 如果省略 `expression`, 则必须利用 `break` 或 `return` 语句跳出循环, 或者在循环体内安排语句修改 `condition` 所检查的变量值。

Section 6.8. The for Loop Statement

```
for (int i = 0; i != 10; /* no expression */ ) {  
    // body must change i or the loop won't terminate  
}
```

If the body doesn't change the value of `i`, then `i` remains 0 and the test will always succeed.

如果循环体不修改 `i` 的值，则 `i` 始终为 0，循环条件永远成立。

6.8.2. Multiple Definitions in the `for` Header

6.8.2. `for` 语句头中的多个定义

Multiple objects may be defined in the `init-statement`; however, only one statement may appear, so all the objects must be of the same general type:

可以在 `for` 语句的 `init-statement` 中定义多个对象；但是不管怎么样，该处只能出现一个语句，因此所有的对象必须具有相同的一般类型：

```
const int size = 42;  
int val = 0, ia[size];  
// declare 3 variables local to the for loop:  
// ival is an int, pi a pointer to int, and ri a reference to int  
for (int ival = 0, *pi = ia, &ri = val;  
     ival != size;  
     ++ival, ++pi, ++ri)  
    // ...
```

Exercises Section 6.8.2

Exercise Explain each of the following loops. Correct any problems you detect.

6.14:

解释下面每个循环，更正你发现的任何问题。

- (a) `for (int *ptr = &ia, ix = 0;
 ix != size && ptr != ia+size;
 ++ix, ++ptr) { /* ... */ }`
- (b) `for (; ;) {
 if (some_condition) return;
 // ...
}`
- (c) `for (int ix = 0; ix != sz; ++ix) { /* ... */ }
if (ix != sz)
 // ...`
- (d) `int ix;
for (ix != sz; ++ix) { /* ... */ }`
- (e) `for (int ix = 0; ix != sz; ++ix, ++ sz) { /* ... */ }`

Exercise The `while` loop is particularly good at executing while some condition holds; for example, while the end-of-file is not reached, read a next value. The `for` loop is generally thought of as a step loop: An index steps through a range of values in a collection. Write an idiomatic use of each loop and then rewrite each using the other loop construct. If you were able to program with only one loop, which construct would you choose? Why?

`while` 循环特别擅长在某个条件保持为真时反复地执行；例如，当未到达文件尾时，一直读取下一个值。一般认为 `for` 循环是一种按步骤执行的循环；使用下标依次遍历集合中一定范围内的元素。按每种循环的习惯用法编写程序，然后再用另外一种结构重写。如果只能用一种循环来编写程序，你会选择哪种结构？为什么？

Exercise Given two `vectors` of `ints`, write a program to determine whether one `vectors` is a prefix of the other. For `vectors` of unequal length, compare the number of elements of the smaller `vector`. For example, given the `vectors` (0,1,1,2) and (0,1,1,2,3,5,8), your program should return `true`.

给出两个 `int` 型的 `vector` 对象，编写程序判断一个对象是否是另一个对象的前缀。如果两个 `vector` 对象的长度不相同，假设较短的 `vector` 对象长度为 `n`，则只对这两个对象的前面 `n` 个元素做比较。例如，对于 (0, 1, 1, 2) 和 (0, 1, 1, 2, 3, 5, 8) 这两个 `vector`，你的程序应该返回 `true`。

6.9. The `do while` Statement

6.9. `do while` 语句

We might want to write a program that interactively performs some calculation for its user. As a simple example, we might want to do sums for the user: Our program prompts the user for a pair of numbers and produces their sum. Having generated one sum, we'd like the program to give the user the option to repeat the process and generate another.

在实际应用中，可能会要求程序员编写一个交互程序，为用户实现某种计算。一个简单的例子是：程序提示用户输入两个数，然后输出读入数之和。在输出和值后，程序可以让用户选择是否重复这个过程计算下一个和。

The body of this program is pretty easy. We'll need to write a prompt, then read a pair of values and print the sum of the values we read. After we print the sum, we'll ask the user whether to continue.

程序的实现相当简单。只需输出一个提示，接着读入两个数，然后输出读入数之和。输出结果后，询问用户是否继续。

The hard part is deciding on a control structure. The problem is that we want to execute the loop until the user asks to exit. In particular, we want to do a sum even on the first iteration. The `do while` loop does exactly what we need. It guarantees that its body is always executed at least once. The syntactic form is as follows:

关键在于控制结构的选择。问题是要求到用户要求退出时，才中止循环的执行。尤其是，在第一次循环时就要求一次和。`do while` 循环正好满足这样的需要。它保证循环体至少执行一次。

```
do
    statement
while (condition);
```



Unlike a `while` statement, a `do-while` statement always ends with a semicolon.

与 `while` 语句不同。`do-while` 语句总是以分号结束。

The `statement` in a `do` is executed before `condition` is evaluated. The `condition` cannot be empty. If `condition` evaluates as `false`, then the loop terminates; otherwise, the loop is repeated. Using a `do while`, we can write our program:

在求解 `condition` 之前，先执行了 `do` 里面的 `statement`。`condition` 不能为空。如果 `condition` 的值为假，则循环结束，否则循环重复执行。使用 `do while` 循环，可以编写程序如下：

```
// repeatedly ask user for pair of numbers to sum
string rsp; // used in the condition; can't be defined inside the do
do {
    cout << "please enter two values: ";
    int val1, val2;
    cin >> val1 >> val2;
    cout << "The sum of " << val1 << " and " << val2
        << " = " << val1 + val2 << "\n\n"
        << "More? [yes][no] ";
    cin >> rsp;
} while (!rsp.empty() && rsp[0] != 'n');
```

The body of this loop is similar to others we've written and so should be easy to follow. What might be a bit surprising is that we defined `rsp` before the `do` rather than defining it inside the loop. Had we defined `rsp` inside the `do`, then `rsp` would go out of scope at the close curly brace before the `while`. Any variable referenced inside the condition must exist before the `do` statement itself.

循环体与之前编写的其他循环语句相似，因此很容易理解。奇怪的是此代码把 `rsp` 定义在 `do` 之前而不是在循环体内部。如果把 `rsp` 定义在 `do` 内部，那么 `rsp` 的作用域就被限制在 `while` 前的右花括号之前了。任何在循环条件中引用变量都必须在 `do` 语句之前就已经存在。

Because the condition is not evaluated until after the statement or block is executed, the `do while` loop does not allow variable definitions:

因为要到循环语句或者语句块执行之后，才求解循环条件，因此 `do while` 循环不可以采用如下方式定义变量：

```
// error: declaration statement within do condition is not supported
do {
    // ...
```

Section 6.9. The do while Statement

```
mumble(foo);
} while (int foo = get_foo()); // error: declaration in do condition
```

If we could define variables in the condition, then any use of the variable would happen *before* the variable was defined!

如果可以在循环条件中定义变量的话，则对变量的任何使用都将发生在变量定义之前！

Exercises Section 6.9

Exercise

- 6.17: Explain each of the following loops. Correct any problems you detect.

解释下列的循环。更正你发现的问题。

```
(a) do
    int v1, v2;
    cout << "Please enter two numbers to sum: ";
    cin >> v1 >> v2;
    if (cin)
        cout << "Sum is: "
            << v1 + v2 << endl;
    while (cin);

(b) do {
    // ...
} while (int ival = get_response());

(c) do {
    int ival = get_response();
    if (ival == some_value())
        break;
} while (ival);
if (!ival)
// ...
```

Exercise

- 6.18: Write a small program that requests two `string`s from the user and reports which `string` is lexicographically less than the other (that is, comes before the other alphabetically). Continue to solicit the user until the user requests to quit. Use the `string` type, the `string` less-than operator, and a `do while` loop.

编写一个小程序，由用户输入两个 `string` 对象，然后报告哪个 `string` 对象按字母排列次序而言比较小（也就是说，哪个的字典序靠前）。继续要求用户输入，直到用户请求退出为止。请使用 `string` 类型、`string` 类型的小于操作符以及 `do while` 循环实现。

6.10. The `break` Statement

6.10. `break` 语句

A **`break` statement** terminates the nearest enclosing `while`, `do while`, `for`, or `switch` statement. Execution resumes at the statement immediately following the terminated statement. For example, the following loop searches a `vector` for the first occurrence of a particular value. Once it's found, the loop is exited:

`break` 语句用于结束最近的 `while`、`do while`、`for` 或 `switch` 语句，并将程序的执行权传递给紧接在被终止语句之后的语句。例如，下面的循环在 `vector` 中搜索某个特殊值的第一次出现。一旦找到，则退出循环：

```
vector<int>::iterator iter = vec.begin();
while (iter != vec.end()) {
    if (*value == *iter)
        break; // ok: found it!
    else
        ++iter; // not found: keep looking
} // end of while
if (iter != vec.end()) // break to here ...
// continue processing
```

In this example, the `break` terminates the `while` loop. Execution resumes at the `if` statement immediately following the `while`.

本例中，`break` 终止了 `while` 循环。执行权交给紧跟在 `while` 语句后面的 `if` 语句，程序继续执行。

A `break` can appear only within a loop or `switch` statement or in a statement nested inside a loop or `switch`. A `break` may appear within an `if` only when the `if` is inside a `switch` or a loop. A `break` occurring outside a loop or `switch` is a compile-time error. When a `break` occurs inside a nested `switch` or loop statement, the enclosing loop or `switch` statement is unaffected by the termination of the inner `switch` or loop:

`break` 只能出现在循环或 `switch` 结构中，或者出现在嵌套于循环或 `switch` 结构中的语句里。对于 `if` 语句，只有当它嵌套在 `switch` 或循环里面时，才能使用 `break`。`break` 出现在循环外或者 `switch` 外将会导致编译时错误。当 `break` 出现在嵌套的 `switch` 或者循环语句中时，将会终止里层的 `switch` 或循环语句，而外层的 `switch` 或者循环不受影响：

```
string inBuf;
while (cin >> inBuf && !inBuf.empty()) {
    switch(inBuf[0]) {
        case '-':
            // process up to the first blank
            for (string::size_type ix = 1;
                 ix != inBuf.size(); ++ix) {
                if (inBuf[ix] == ' ')
                    break; // #1, leaves the for loop
                // ...
            }
            // remaining '-' processing: break #1 transfers control here
            break; // #2, leaves the switch statement
        case '+':
            // ...
    } // end switch
    // end of switch: break #2 transfers control here
} // end while
```

The `break` labeled `#1` terminates the `for` loop within the hyphen case label. It does not terminate the enclosing `switch` statement and in fact does not even terminate the processing for the current case. Processing continues with the first statement following the `for`, which might be additional code to handle the hyphen case or the `break` that completes that section.

`#1` 标记的 `break` 终止了连字符 ('-') `case` 标号内的 `for` 循环，但并没有终止外层的 `switch` 语句，而且事实上也并没有结束当前 `case` 语句的执行。接着程序继续执行 `for` 语句后面的第一个语句，即处理连字符 `case` 标号下的其他代码，或者执行结束这个 `case` 的 `break` 语句。

The `break` labeled `#2` terminates the `switch` statement after handling the hyphen case but does not terminate the enclosing `while` loop. Processing continues after that `break` by executing the condition in the `while`, which reads the next `string` from the standard input.

`#2` 标记的 `break` 终止了处理连字符情况的 `switch` 语句，但没有终止 `while` 循环。程序接着执行 `break` 后面的语句，即求解 `while` 的循环条件，从标准输入读入下一个 `string` 对象。

Exercises Section 6.10

Exercise 6.19: The first program in this section could be written more succinctly. In fact, its action could be contained entirely in the condition in the `while`. Rewrite the loop so that it has an empty body and does the work of finding the element in the condition.

本节的第一个程序可以写得更简洁。事实上，该程序的所有工作可以全部包含在 `while` 的循环条件中。重写这个循环，使得它的循环体为空，并找出满足条件的元素。

Exercise 6.20: Write a program to read a sequence of `strings` from standard input until either the same word occurs twice in succession or all the words have been read. Use a `while` loop to read the text one word at a time. Use the `break` statement to terminate the loop if a word occurs twice in succession. Print the word if it occurs twice in succession, or else print a message saying that no word was repeated.

编写程序从标准输入读入一系列 `string` 对象，直到同一个单词连续出现两次，或者所有的单词都已读完，才结束读取。请使用 `while` 循环，每次循环读入一个单词。如果连续出现相同的单词，便以 `break` 语句结束循环，此时，请输出这个重复出现的单词；否则输出没有任何单词连续重复出现的信息。

6.11. The `continue` Statement

6.11. `continue` 语句

A [continue statement](#) causes the current iteration of the nearest enclosing loop to terminate. Execution resumes with the evaluation of the condition in the case of a `while` or `do while` statement. In a `for` loop, execution continues by evaluating the *expression* inside the `for` header.

[continue](#) 语句导致最近的循环语句的当次迭代提前结束。对于 `while` 和 `do while` 语句，继续求解循环条件。而对于 `for` 循环，程序流程接着求解 `for` 语句头中的 *expression* 表达式。

For example, the following loop reads the standard input one word at a time. Only words that begin with an underscore will be processed. For any other value, we terminate the current iteration and get the next input:

例如，下面的循环每次从标准输入中读入一个单词，只有以下划线开头的单词才做处理。如果是其他的值，终止当前循环，接着读取下一个单词：

```
string inBuf;
while (cin >> inBuf && !inBuf.empty()) {
    if (inBuf[0] != '_')
        continue; // get another input
    // still here? process string ...
}
```

A `continue` can appear only inside a `for`, `while`, or `do while` loop, including inside blocks nested inside such loops.

`continue` 语句只能出现在 `for`、`while` 或者 `do while` 循环中，包括嵌套在这些循环内部的块语句中。

Exercises Section 6.11

Exercise 6.21: Revise the program from the last exercise in [Section 6.10](#) (p. 213) so that it looks only for duplicated words that start with an uppercase letter.

修改第 [6.10](#) 节最后一个习题的程序，使得它只寻找以大写字母开头的连续出现的单词。

6.12. The `goto` Statement

6.12. `goto` 语句

A `goto statement` provides an unconditional jump from the `goto` to a labeled statement in the same function.

`goto` 语句提供了函数内部的无条件跳转，实现从 `goto` 语句跳转到同一函数内某个带标号的语句。



Use of `gos` has been discouraged since the late 1960s. `gos` make it difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a `goto` can be rewritten so that it doesn't need the `goto`.

从上世纪 60 年代后期开始，不主张使用 `goto` 语句。`goto` 语句使跟踪程序控制流程变得很困难，并且使程序难以理解，也难以修改。所有使用 `goto` 的程序都可以改写为不用 `goto` 语句，因此也就没有必要使用 `goto` 语句了。

The syntactic form of a `goto` statement is

`goto` 语句的语法规则如下：

```
goto label;
```

where `label` is an identifier that identifies a labeled statement. A `labeled statement` is any statement that is preceded by an identifier followed by a colon:

其中 `label` 是用于标识带标号的语句的标识符。在任何语句前提供一个标识符和冒号，即得带标号的语句：

```
end: return; // labeled statement, may be target of a goto
```

The identifier that forms the label may be used only as the target of a `goto`. For this reason, label identifiers may be the same as variable names or other identifiers in the program without interfering with other uses of those identifiers. The `goto` and the labeled statement to which it transfers control must be in the same function.

形成标号的标识符只能用作 `goto` 的目标。因为这个原因，标号标识符可以与变量名以及程序里的其他标识符一样，不与别的标识符重名。`goto` 语句和获得所转移的控制权的带标号的语句必须位于于同一个函数内。

A `goto` may not jump forward over a variable definition:

`goto` 语句不能跨越变量的定义语句向前跳转：

```
// ...
goto end;
int ix = 10; // error: goto bypasses declaration statement
end:
// error: code here could use ix but the goto bypassed its declaration
ix = 42;
```

If definitions are needed between a `goto` and its corresponding label, the definitions must be enclosed in a block:

如果确实需要在 `goto` 和其跳转对应的标号之间定义变量，则定义必须放在一个块语句中：

```
// ...
goto end; // ok: jumps to a point where ix is not defined
{
    int ix = 10;
    // ... code using ix
}
end: // ix no longer visible here
```

A jump backward over an already executed definition is okay. Why? Jumping over an unexecuted definition would mean that a variable could be used even though it was never defined. Jumping back to a point before a variable is defined destroys the variable and constructs it again:

```
// backward jump over declaration statement ok
begin:
    int sz = get_size();
    if (sz <= 0) {
        goto begin;
    }
```

Note that `sz` is destroyed when the `goto` executes and is defined anew when control jumps back to `begin`.

注意：执行 `goto` 语句时，首先撤销变量 `sz`，然后程序的控制流程跳转到带 `begin` 标号的语句继续执行，再次重新创建和初始化 `sz` 变量。

Exercises Section 6.12

Exercise 6.22: The last example in this section that jumped back to `begin` could be better written using a loop. Rewrite the code to eliminate the `goto`.

对于本节的最后一个例子，跳回到 `begin` 标号的功能可以用循环更好地实现。请不使用 `goto` 语句重写这段代码。

6.13. `try` Blocks and Exception Handling

6.13. `try` 块和异常处理

Handling errors and other anomalous behavior in programs can be one of the most difficult parts of designing any system. Long-lived, interactive systems such as communication switches and routers can devote as much as 90 percent of their code to error detection and error handling. With the proliferation of Web-based applications that run indefinitely, attention to error handling is becoming more important to more and more programmers.

在设计各种软件系统的过程中，处理程序中的错误和其他反常行为是困难的部分之一。像通信交换机和路由器这类长期运行的交互式系统必须将 90% 的程序代码用于实现错误检测和错误处理。随着基于 Web 的应用程序在运行时不确定性的增多，越来越多的程序员更加注重错误的处理。

Exceptions are run-time anomalies, such as running out of memory or encountering unexpected input. Exceptions exist outside the normal functioning of the program and require immediate handling by the program.

异常就是运行时出现的不正常，例如运行时耗尽了内存或遇到意外的非法输入。异常存在于程序的正常功能之外，并要求程序立即处理。

In well-designed systems, exceptions represent a subset of the program's error handling. Exceptions are most useful when the code that detects a problem cannot handle it. In such cases, the part of the program that detects the problem needs a way to transfer control to the part of the program that can handle the problem. The error-detecting part also needs to be able to indicate what kind of problem occurred and may want to provide additional information.

在设计良好的系统中，异常是程序错误处理的一部分。当程序代码检查到无法处理的问题时，异常处理就特别有用。在这些情况下，检测出问题的那部分程序需要一种方法把控制权转到可以处理这个问题的那部分程序。错误检测程序还必须指出具体出现了什么问题，并且可能需要提供一些附加信息。

Exceptions support this kind of communication between the error-detecting and error-handling parts of a program. In C++ exception handling involves:

异常机制提供程序中错误检测与错误处理部分之间的通信。C++ 的异常处理中包括：

1. `throw` expressions, which the error-detecting part uses to indicate that it encountered an error that it cannot handle. We say that a `throw` raises an exceptional condition.
throw 表达式，错误检测部分使用这种表达式来说明遇到了不可处理的错误。可以说，`throw` 引发了异常条件。
2. `try` blocks, which the error-handling part uses to deal with an exception. A `try` block starts with keyword `try` and ends with one or more `catch` clauses. Exceptions thrown from code executed inside a `try` block are usually handled by one of the `catch` clauses. Because they "handle" the exception, catch clauses are known as handlers.
try 块，错误处理部分使用它来处理异常。`try` 语句块以 `try` 关键字开始，并以一个或多个 catch 子句结束。在 `try` 块中执行的代码所抛出（`throw`）的异常，通常会被其中一个 `catch` 子句处理。由于它们“处理”异常，`catch` 子句也称为处理代码。
3. A set of `exception` classes defined by the library, which are used to pass the information about an error between a `throw` and an associated `catch`.
由标准库定义的一组异常类，用来在 `throw` 和相应的 `catch` 之间传递有关的错误信息。

In the remainder of this section we'll introduce these three components of exception handling. We'll have more to say about exceptions in [Section 17.1](#) (p. 688).

在本节接下来的部分将要介绍这三种异常处理的构成。而[第 17.1 节](#)将会进一步了解异常的相关内容。

6.13.1. A `throw` Expression

6.13.1 `throw` 表达式

An exception is thrown using a `throw` expression, which consists of the keyword `throw` followed by an expression. A `throw` expression is usually followed by a semicolon, making it into an expression statement. The type of the expression determines what kind of exception is thrown.

系统通过 `throw` 表达式抛出异常。`throw` 表达式由关键字 `throw` 以及尾随的表达式组成，通常以分号结束，这样它就成为了表达式语句。`throw` 表达式的类型决定了所抛出异常的类型。

As a simple example, recall the program on page 24 that added two objects of type `Sales_item`. That program checked whether the records it read referred to the same book. If not, it printed a message and exited.

回顾[第 1.5.2 节](#)将两个 `Sales_item` 类型对象相加的程序，就是一个简单的例子。该程序检查读入的记录是否来自同一本书。如果不是，就输出一条信息然后退出程序。

Section 6.13. try Blocks and Exception Handling

```
Sales_item item1, item2;
std::cin >> item1 >> item2;
// first check that item1 and item2 represent the same book
if (item1.same_isbn(item2)) {
    std::cout << item1 + item2 << std::endl;
    return 0; // indicate success
} else {
    std::cerr << "Data must refer to same ISBN"
        << std::endl;
    return -1; // indicate failure
}
```

In a less simple program that used `Sales_items`, the part that adds the objects might be separated from the part that manages the interaction with a user. In this case, we might rewrite the test to `throw` an exception instead:

在使用 `Sales_items` 的更简单的程序中，把将对象相加的部分和负责跟用户交互的部分分开。在这个例子中，用 `throw` 抛出异常来改写检测代码：

```
// first check that data is for the same item
if (!item1.same_isbn(item2))
    throw runtime_error("Data must refer to same ISBN");
// ok, if we're still here the ISBNs are the same
std::cout << item1 + item2 << std::endl;
```

In this code we check whether the ISBNs differ. If so, we discontinue execution and transfer control to a handler that will know how to handle this error.

这段代码检查 ISBN 对象是否不相同。如果不同的话，停止程序的执行，并将控制转移给处理这种错误的处理代码。

A `throw` takes an expression. In this case, that expression is an object of type `runtime_error`. The `runtime_error` type is one of the standard library exception types and is defined in the `stdexcept` header. We'll have more to say about these types shortly. We create a `runtime_error` by giving it a `string`, which provides additional information about the kind of problem that occurred.

`throw` 语句使用了一个表达式。在本例中，该表达式是 `runtime_error` 类型的对象。`runtime_error` 类型是标准库异常类中的一种，在 `stdexcept` 头文件中定义。在后续章节中很快就会更详细地介绍这些类型。我们通过传递 `string` 对象来创建 `runtime_error` 对象，这样就可以提供更多关于所出现问题的相关信息。

6.13.2. The `try` Block

6.13.2. `try` 块

The general form of a `try` block is

`try` 块的通用语法形式是：

```
try {
    program-statements
} catch (exception-specifier) {
    handler-statements
} catch (exception-specifier) {
    handler-statements
} //...
```

A `try` block begins with the keyword `try` followed by a block enclosed in braces. Following the `try` block is a list of one or more catch clauses. A catch clause consists of three parts: the keyword `catch`, the declaration of a single type or single object within parentheses (referred to as an **exception specifier**), and a block, which as usual must be enclosed in curly braces. If the catch clause is selected to handle an exception, the associated block is executed. Once the catch clause finishes, execution continues with the statement immediately following the last catch clause.

`try` 块以关键字 `try` 开始，后面是用花括号起来的语句序列块。`try` 块后面是一个或多个 `catch` 子句。每个 `catch` 子句包括三部分：关键字 `catch`，圆括号内单个类型或者单个对象的声明——称为**异常说明符**，以及通常用花括号括起来的语句块。如果选择了一个 `catch` 子句来处理异常，则执行相关的块语句。一旦 `catch` 子句执行结束，程序流程立即继续执行紧接着最后一个 `catch` 子句的语句。

The `program-statements` inside the `try` constitute the normal logic of the program. They can contain any C++ statement, including declarations. Like any block, a `try` block introduces a local scope, and variables declared within a `try` block cannot be referred to outside the `try`, including within the catch clauses.

`try` 语句内的 `program-statements` 形成程序的正常逻辑。这里面可以包含任意 C++ 语句，包括变量声明。与其他块语句一样，`try` 块引入局部作用域，在 `try` 块中声明的变量，包括 `catch` 子句声明的变量，不能在 `try` 外面引用。

Writing a Handler

编写处理代码

In the preceding example we used a `throw` to avoid adding two `Sales_items` that represented different books. We imagined that the part of the program that added to `Sales_items` was separate from the part that communicated with the user. The part that interacts with the user might contain code something like the following to handle the exception that was thrown:

在前面的例子中，使用了 `throw` 来避免将两个表示不同书的 `Sales_items` 对象相加。想象一下将 `Sales_items` 对象相加的那部分程序与负责与用户交流的那部分是分开的，则与用户交互的部分也许会包含下面的用于处理所捕获异常的代码：

```
while (cin >> item1 >> item2) {
    try {
        // execute code that will add the two Sales_items
        // if the addition fails, the code throws a runtime_error exception
    } catch (runtime_error err) {
        // remind the user that ISBN must match and prompt for another pair
        cout << err.what()
           << "\nTry Again? Enter y or n" << endl;
        char c;
        cin >> c;
        if (cin && c == 'n')
            break;      // break out of the while loop
    }
}
```

Following the `try` keyword is a block. That block would invoke the part of the program that processes `Sales_item` objects. That part might `throw` an exception of type `runtime_error`.

关键字 `try` 后面是一个块语句。这个块语句调用处理 `Sales_item` 对象的程序部分。这部分也可能会抛出 `runtime_error` 类型的异常。

This `try` block has a single `catch` clause, which handles exceptions of type `runtime_error`. The statements in the block following the `catch` define the actions that will be executed if code inside the `try` block throws a `runtime_error`. Our `catch` handles the error by printing a message and asking the user to indicate whether to continue. If the user enters an '`n`', then we break out of the `while`. Otherwise the loop continues by reading two new `Sales_items`.

上述 `try` 块提供单个 `catch` 子句，用来处理 `runtime_error` 类型的异常。在执行 `try` 块代码的过程中，如果在 `try` 块中的代码抛出 `runtime_error` 类型的异常，则处理这类异常的动作在 `catch` 后面的块语句中定义。本例中，`catch` 输出信息并且询问用户是否继续进行异常处理。如果用户输入 '`n`'，则结束 `while`；否则继续循环，读入两个新的 `Sales_items` 对象。

The prompt to the user prints the return from `err.what()`. We know that `err` has type `runtime_error`, so we can infer that `what` is a member function ([Section 1.5.2](#), p. 24) of the `runtime_error` class. Each of the library exception classes defines a member function named `what`. This function takes no arguments and returns a C-style character string. In the case of `runtime_error`, the C-style string that `what` returns is a copy of the `string` that was used to initialize the `runtime_error`. If the code described in the previous section threw an exception, then the output printed by this `catch` would be

通过输出 `err.what()` 的返回值提示用户。大家都知道 `err` 返回 `runtime_error` 类型的值，因此可以推断出 `what` 是 `runtime_error` 类的一个成员函数 ([1.5.2 节](#))。每一个标准库异常类都定义了名为 `what` 的成员函数。这个函数不需要参数，返回 C 风格字符串。在出现 `runtime_error` 的情况下，`what` 返回的 C 风格字符串，是用于初始化 `runtime_error` 的 `string` 对象的副本。如果在前面章节描述的代码抛出异常，那么执行这个 `catch` 将输出。

```
Data must refer to same ISBN
Try Again? Enter y or n
```

functions are exited during the search for a handler

函数在寻找处理代码的过程中退出

in complex systems the execution path of a program may pass through multiple `try` blocks before encountering code that actually throws an exception. for example, a `try` block might call a function that contains a `try`, that calls another function with its own `try`, and so on.

在复杂的系统中，程序的执行路径也许在遇到抛出异常的代码之前，就已经经过了多个 `try` 块。例如，一个 `try` 块可能调用了包含另一 `try` 块的函数，它的 `try` 块又调用了含有 `try` 块的另一个函数，如此类推。

the search for a handler reverses the call chain. when an exception is thrown, the function that threw the exception is searched first. if no matching `catch` is found, the function terminates, and the function that called the one that threw is searched for a matching `catch`. if no handler is found, then that function also exits and the function that called it is searched; and so on back up the execution path until a catch of an appropriate type is found.

寻找处理代码的过程与函数调用链刚好相反。抛出一个异常时，首先要搜索的是抛出异常的函数。如果没有找到匹配的 `catch`，则终止这个函数的执行，并在调用这个函数的函数中寻找相配的 `catch`。如果仍然找到相应的处理代码，该函数同样要终止，搜索调用它的函数。如此类推，继续按执行路径回退，直到找到适当类型的 `catch` 为止。

If no catch clause capable of handling the exception exists, program execution is transferred to a library function named `terminate`, which is defined in the `exception` header. The behavior of that function is system dependent, but it usually aborts the program.

Section 6.13. try Blocks and Exception Handling

如果不存在处理该异常的 `catch` 子句，程序的运行就要跳转到名为 `terminate` 的标准库函数，该函数在 `exception` 头文件中定义。这个标准库函数的行为依赖于系统，通常情况下，它的执行将导致程序非正常退出。

Exceptions that occur in programs that define no `try` blocks are handled in the same manner: After all, if there are no `try` blocks, there can be no handlers for any exception that might be thrown. If an exception occurs, then `terminate` is called and the program (ordinarily) is aborted.

在程序中出现的异常，如果没有经 `try` 块定义，则都以相同的方式来处理：毕竟，如果没有任何 `try` 块，也就没有捕获异常的处理代码（`catch` 子句）。此时，如果发生了异常，系统将自动调用 `terminate` 终止程序的执行。

Exercises Section 6.13.2

Exercise 6.23: The `bitset` operation `to_ulong` throws an `overflow_error` exception if the `bitset` is larger than the size of an `unsigned long`. Write a program that generates this exception.

`bitset` 类提供 `to_ulong` 操作，如果 `bitset` 提供的位数大于 `unsigned long` 的长度时，抛出一个 `overflow_error` 异常。编写产生这种异常的程序。

Exercise 6.24: Revise your program to catch this exception and print a message.

修改上述的程序，使它能捕获这种异常并输出提示信息。

6.13.3. Standard Exceptions

6.13.3. 标准异常

The C++ library defines a set of classes that it uses to report problems encountered in the functions in the standard library. These standard exception classes are also intended to be used in the programs we write. Library exception classes are defined in four headers:

C++ 标准库定义了一组类，用于报告在标准库中的函数遇到的问题。程序员可在自己编写的程序中使用这些标准异常类。标准库异常类定义在四个头文件中：

1. The `exception` header defines the most general kind of exception class named `exception`. It communicates only that an exception occurs but provides no additional information.

`exception` 头文件定义了最常见的异常类，它的类名是 `exception`。这个类只通知异常的产生，但不会提供更多的信息。

2. The `stdexcept` header defines several general purpose exception classes. These types are listed in [Table 6.1](#) on the following page.

`stdexcept` 头文件定义了几种常见的异常类，这些类型在 [表 6.1](#) 中列出。

Table 6.1. Standard Exception Classes Defined in `<stdexcept>`

表 6.1 在 `<stdexcept>` 头文件中定义的标准异常类

<code>exception</code>	The most general kind of problem. 最常见的问题。
<code>runtime_error</code>	Problem that can be detected only at run time. 运行时错误：仅在运行时才能检测到问题
<code>range_error</code>	Run-time error: result generated outside the range of values that are meaningful. 运行时错误：生成的结果超出了有意义的值域范围
<code>overflow_error</code>	Run-time error: computation that overflowed. 运行时错误：计算上溢
<code>underflow_error</code>	Run-time error: computation that underflowed. 运行时错误：计算下溢

Section 6.13. try Blocks and Exception Handling

<code>logic_error</code>	Problem that could be detected before run time. 逻辑错误：可在运行前检测到问题
<code>domain_error</code>	Logic error: argument for which no result exists. 逻辑错误：参数的结果值不存在
<code>invalid_argument</code>	Logic error: inappropriate argument. 逻辑错误：不合适的参数
<code>length_error</code>	Logic error: attempt to create an object larger than the maximum size for that type. 逻辑错误：试图生成一个超出该类型最大长度的对象
<code>out_of_range</code>	Logic error: used a value outside the valid range. 逻辑错误：使用一个超出有效范围的值

3. The `new` header defines the `bad_alloc` exception type, which is the exception thrown by `new` ([Section 5.11](#), p. 174) if it cannot allocate memory.

`new` 头文件定义了 `bad_alloc` 异常类型，提供因无法分配内存而由 `new` ([第 5.11 节](#)) 抛出的异常。

4. The `type_info` header defines the `bad_cast` exception type, which we will discuss in [Section 18.2](#) (p. 772).

`type_info` 头文件定义了 `bad_cast` 异常类型，这种类型将[第 18.2 节](#)讨论。

Standard Library Exception Classes

标准库异常类

The library exception classes have only a few operations. We can create, copy, and assign objects of any of the exception types. The `exception`, `bad_alloc`, and `bad_cast` types define only a default constructor ([Section 2.3.4](#), p. 50); it is not possible to provide an initializer for objects of these types. The other exception types define only a single constructor that takes a `string` initializer. When we define any of these other exception types, we must supply a `string` argument. That `string` initializer is used to provide additional information about the error that occurred.

标准库异常类只提供很少的操作，包括创建、复制异常类型对象以及异常类型对象的赋值。`exception`、`bad_alloc` 以及 `bad_cast` 类型只定义了默认构造函数 ([第 2.3.4 节](#))，无法在创建这些类型的对象时为它们提供初值。其他的异常类型则只定义了一个使用 `string` 初始化式的构造函数。当需要定义这些异常类型的对象时，必须提供一想 `string` 参数。`string` 初始化式用于为所发生的错误提供更多的信息。

The exception types define only a single operation named `what`. That function takes no arguments and returns a `const char*`. The pointer it returns points to a C-style character string ([Section 4.3](#), p. 130). The purpose of this C-style character string is to provide some sort of textual description of the exception thrown.

异常类型只定义了一个名为 `what` 的操作。这个函数不需要任何参数，并且返回 `const char*` 类型值。它返回的指针指向一个 C 风格字符串 ([第 4.3 节](#))。使用 C 风格字符串的目的是为所抛出的异常提出更详细的文字描述。

The contents of the C-style character array to which `what` returns a pointer depends on the type of the exception object. For the types that take a `string` initializer, the `what` function returns that `string` as a C-style character array. For the other types, the value returned varies by compiler.

`what` 函数所返回的指针指向 C 风格字符数组的内容，这个数组的内容依赖于异常对象的类型。对于接受 `string` 初始化式的异常类型，`what` 函数将返回该 `string` 作为 C 风格字符数组。对于其他异常类型，返回的值则根据编译器的变化而不同。

6.14. Using the Preprocessor for Debugging

6.14. 使用预处理器进行调试

In [Section 2.9.2](#) (p. 71) we learned how to use preprocessor variables to prevent header files being included more than once. C++ programmers sometimes use a technique similar to header guards to conditionally execute debugging code. The idea is that the program will contain debugging code that is executed only while the program is being developed. When the application is completed and ready to ship, the debugging code is turned off. We can write conditional debugging code using the `NDEBUG` preprocessor variable:

[第 2.9.2 节](#) 介绍了如何使用预处理变量来避免重复包含头文件。C++ 程序员有时也会使用类似的技术有条件地执行用于调试的代码。这种想法是：程序所包含的调试代码仅在开发过程中执行。当应用程序已经完成，并且准备提交时，就会将调试代码关闭。可使用 `NDEBUG` 预处理变量实现有条件的调试代码：

```
int main()
{
#ifndef NDEBUG
cerr << "starting main" << endl;
#endif
// ...
```

If `NDEBUG` is not defined, then the program writes the message to `cerr`. If `NDEBUG` is defined, then the program executes without ever passing through the code between the `#ifndef` and the `#endif`.

如果 `NDEBUG` 未定义，那么程序就会将信息写到 `cerr` 中。如果 `NDEBUG` 已经定义了，那么程序执行时将会跳过 `#ifndef` 和 `#endif` 之间的代码。

By default, `NDEBUG` is not defined, meaning that by default, the code inside the `#ifndef` and `#endif` is processed. When the program is being developed, we leave `NDEBUG` undefined so that the debugging statements are executed. When the program is built for delivery to customers, these debugging statements can be (effectively) removed by defining the `NDEBUG` preprocessor variable. Most compilers provide a command line option that defines `NDEBUG`:

默认情况下，`NDEBUG` 未定义，这也就意味着必须执行 `#ifndef` 和 `#endif` 之间的代码。在开发程序的过程中，只要保持 `NDEBUG` 未定义就会执行其中的调试语句。开发完成后，要将程序交付给客户时，可通过定义 `NDEBUG` 预处理变量，（有效地）删除这些调试语句。大多数的编译器都提供定义 `NDEBUG` 命令行选项：

```
$ CC -DNDEBUG main.c
```

has the same effect as writing `#define NDEBUG` at the beginning of `main.c`.

这样的命令行行将于在 `main.c` 的开头提供 `#define NDEBUG` 预处理命令。

The preprocessor defines four other constants that can be useful in debugging:

预处理器还定义了其余四种在调试时非常有用的常量：

`_FILE_` name of the file.

`_FILE_` 文件名

`_LINE_` current line number.

`_LINE_` 当前行号

`_TIME_` time the file was compiled.

`_TIME_` 文件被编译的时间

`_DATE_` date the file was compiled.

`_DATE_` 文件被编译的日期

We might use these constants to report additional information in error messages:

可使用这些常量在错误消息中提供更多的信息：

```
if (word.size() < threshold)
cerr << "Error: " << _FILE_
<< " : line " << _LINE_
<< " Compiled on " << _DATE_
<< " at " << _TIME_
<< " Word read was " << word
<< " : Length too short" << endl;
```

If we give this program a `string` that is shorter than the `threshold`, then the following error message will be generated:

如果给这个程序提供一个比 `threshold` 短的 `string` 对象，则会产生下面的错误信息：

```
Error: wdebug.cc : line 21
        Compiled on Jan 12 2005 at 19:44:40
        Word read was "foo": Length too short
```

Another common debugging technique uses the `NDEBUG` preprocessor variable and the [assert preprocessor macro](#). The `assert` macro is defined in the `cassert` header, which we must include in any file that uses `assert`.

另一个常见的调试技术是使用 `NDEBUG` 预处理变量以及 [assert 预处理器宏](#)。`assert` 宏是在 `cassert` 头文件中定义的，所有使用 `assert` 的文件都必须包含这个头文件。

A preprocessor macro acts something like a function call. The `assert` macro takes a single expression, which it uses as a condition:

预处理宏有点像函数调用。`assert` 宏需要一个表达式作为它的条件：

```
assert(expr)
```

As long as `NDEBUG` is not defined, the `assert` macro evaluates the condition and if the result is false, then `assert` writes a message and terminates the program. If the expression has a nonzero (e.g., true) value, then `assert` does nothing.

只要 `NDEBUG` 未定义，`assert` 宏就求解条件表达式 `expr`，如果结果为 `false`，`assert` 输出信息并且终止程序的执行。如果该表达式有一个非零（例如，`true`）值，则 `assert` 不做任何操作。

Unlike exceptions, which deal with errors that a program expects might happen in production, programmers use `assert` to test conditions that "cannot happen." For example, a program that does some manipulation of input text might know that all words it is given are always longer than a threshold. That program might contain a statement such as:

与异常不同（异常用于处理程序执行时预期要发生的错误），程序员使用 `assert` 来测试“不可能发生”的条件。例如，对于处理输入文本的程序，可以预测全部给出的单词都比指定的阈值长。那么程序可以包含这样一个语句：

```
assert(word.size() > threshold);
```

During testing the `assert` has the effect of verifying that the data are always of the expected size. Once development and test are complete, the program is built and `NDEBUG` is defined. In production code, `assert` does nothing, so there is no run-time cost. Of course, there is also no run-time check. `assert` should be used only to verify things that truly should not be possible. It can be useful as an aid in getting a program debugged but should not be used to substitute for run-time logic checks or error checking that the program should be doing.

在测试过程中，`assert` 等效于检验数据是否总是具有预期的大小。一旦开发和测试工作完成，程序就已经建立好，并且定义了 `NDEBUG`。在成品代码中，`assert` 语句不做任何工作，因此也没有任何运行时代价。当然，也不会引起任何运行时检查。`assert` 仅用于检查确实不可能的条件，这仅对程序的调试有帮助，但不能用来代替运行时的逻辑检查，也不能代替对程序可能产生的错误的检测。

Exercises Section 6.14

Exercise 6.25: Revise the program you wrote for the exercise in [Section 6.11](#) (p. 214) to conditionally print information about its execution. For example, you might print each word as it is read to let you determine whether the loop correctly finds the first duplicated word that begins with an uppercase letter. Compile and run the program with debugging turned on and again with it turned off.

修改第 6.11 节习题所编写的程序，使其可以有条件地输出运行时的信息。例如，可以输出每一个读入的单词，用来判断循环是否正确地找到第一个连续出现的以大写字母开头的单词。分别在打开和关闭调试器的情况下编译和运行这个程序。

Exercise 6.26: What happens in the following loop:

下面循环会导致什么现象的发生：

```
string s;
while (cin >> s) {
    assert(cin);
    // process s
}
```

Explain whether this usage seems like a good application of the `assert` macro.

解释这种用法是否是 `assert` 宏的一种恰当应用。

Exercise Explain this loop:

6.27:

解释下面的循环：

```
string s;
while (cin >> s && s != sought) { } // empty body
assert(cin);
// process s
```

Chapter Summary

小结

C++ provides a fairly limited number of statements. Most of these affect the flow of control within a program:

C++ 提供了种类相当有限的语句，其中大多数都会影响程序的控制流：

`while`, `for`, and `do while` statements, which implement iterative loops

`while`、`for` 以及 `do while` 语句，实现反复循环；

`if` and `switch`, which provide conditional execution

`if` 和 `switch`，提供条件分支结构；

`continue`, which stops the current iteration of a loop

`continue`, 终止当次循环；

`break`, which exits a loop or `switch` statement

`break`，退出一个循环或 `switch` 语句；

`goto`, which transfers control to a labeled statement

`goto`，将控制跳转到某个标号语句；

`try`, `catch`, which define a `try` block enclosing a sequence of statements that might throw an exception. The catch clause(s) are intended to handle the exception(s) that the enclosed code might throw.

`try`、`catch` 语句，实现 `try` 块的定义，该语句包含一个可能抛出异常的语句序列，`catch` 子句则用来处理在 `try` 块里抛出的异常；

`throw` expressions, which exit a block of code, transferring control to an associated catch clause

`throw` 表达式，用于退出代码块的执行，将控制转移给相关的 `catch` 子句。

There is also a `return` statement, which will be covered in [Chapter 7](#).

当然还有将在[第七章](#)介绍的 `return` 语句。

In addition, there are expression statements and declaration statements. An expression statement causes the subject expression to be evaluated. Declarations and definitions of variables were described in [Chapter 2](#).

此外，C++ 还提供表达式语句和声明语句。表达式语句用于求解表达式。变量的声明和定义则已在[第二章](#)讲述过了。

Defined Terms

术语

assert

Preprocessor macro that takes a single expression, which it uses as a condition. If the preprocessor variable `NDEBUG` is not defined, then `assert` evaluates the condition. If the condition is false, `assert` writes a message and terminates the program.

一种预处理宏，使用单个表达式作为断言条件。如果预处理变量 `NDEBUG` 没有定义，则 `assert` 将求解它的条件表达式。若条件为 `false`，`assert` 输出信息并终止程序的执行。

block (块)

A sequence of statements enclosed in curly braces. A block is a statement, so it can appear anywhere a statement is expected.

包含在一对花括号里的语句序列。在语法上，块就是单语句，可出现在任何单语句可以出现的地方。

break statement (break 语句)

Terminates the nearest enclosing loop or `switch` statement. Execution transfers to the first statement following the terminated loop or `switch`.

一种语句，能够终止最近的循环或者 `switch` 语句的执行，将控制权交给被终止的循环或者 `switch` 后的第一条语句。

case label (case 标号)

Integral constant value that follows the keyword `case` in a `switch` statement. No two case labels in the same `switch` statement may have the same value. If the value in the `switch` condition is equal to that in one of the case labels, control transfers to the first statement following the matched label. Execution continues from that point until a `break` is encountered or it flows off the end of the `switch` statement.

`switch` 语句中跟在关键字 `case` 后的整型常量值。在同一个 `switch` 结构中不能有任何两个标号拥有相同的常量值。如果 `switch` 条件表达式的值与其中某个标号的值相等，则控制权转移到匹配标号后面的第一条语句，从这种语句开始依次继续各个语句，直到遇到 `break` 或者到达 `switch` 结尾为止。

catch clause (catch 子句)

The `catch` keyword, an exception specifier in parentheses, and a block of statements. The code inside a catch clause does whatever is necessary to handle an exception of the type defined in its exception specifier.

一种语句，包括关键字 `catch`、圆括号内的异常说明符以及一个块语句。`catch` 子句中的代码实现某种异常的处理，该异常的处理，该异常由圆括号内的异常说明符定义。

compound statement (复合语句)

Synonym for block.

块的同义词。

continue statement (continue 语句)

Terminates the current iteration of the nearest enclosing loop. Execution transfers to the loop condition in a `while` or `do` or to the expression in the `for` header.

一种语句，能够结束最近的循环结构的当次循环迭代，将控制流转移到 `while` 或 `do` 的循环条件表达式，或者 `for` 语句头中第三个表达式。

dangling else (悬垂 else)

Colloquial term used to refer to the problem of how to process nested `if` statements in which there are more `ifs` than `elses`. In C++, an `else` is always paired with the closest preceding unmatched `if`. Note that curly braces can be used to effectively hide an inner `if` so that the programmer can control with which `if` a given `else` should be matched.

一个通俗术语，指出如何处理嵌套 `if` 语句中 `if` 多于 `else` 时发生的二义性问题。C++ 中，`else` 总是与最近的未匹配的 `if` 配对。注意使用花括号能有效地隐藏内层 `if`，使程序员可以控制给定的 `else` 与哪个 `if` 相匹配。

declaration statement (声明语句)

A statement that defines or declares a variable. Declarations were covered in [Chapter 2](#).

定义或者声明变量的语句。声明已在[第二章](#)中介绍。

default label (default 标号)

The `switch` case label that matches any otherwise unmatched value computed in the `switch` condition.

`switch` 语句中的一种标号，当计算 `switch` 条件所得的值与所有 `case` 标号的值都不匹配时，则执行 `default` 标号关联的语句。

exception classes (异常类)

Set of classes defined by the standard library to be used to represent errors. [Table 6.1](#) (p. 220) lists the general purpose exceptions.

标准库定义的一组描述程序错误的类。[表 6.1](#) 列出了常见的异常。

exception handler (异常处理代码)

Code that deals with an exception raised in another part of the program. Synonym for catch clause.

一段代码，用于处理程序某个部分引起的异常。是 `catch` 子句的同义词。

exception specifier (异常说明符)

The declaration of an object or a type that indicates the kind of exceptions a catch clause can handle.

对象或类型的声明，用于指出当前的 `catch` 能处理的异常类型。

expression statement (表达式语句)

An expression followed by a semicolon. An expression statement causes the expression to be evaluated.

一种语句，由后接分号的表达式构成。表达式语句用于表达式的求解。

flow of control (控制流)

Execution path through a program.

程序的执行路径。

goto statement (goto 语句)

Statement that causes an unconditional transfer of control to a specified labeled statement elsewhere in the program. `gotos` obfuscate the flow of control within a program and should be avoided.

一种语句，能够使程序控制流程无条件跳转到指定标号语句。`goto` 扰乱了程序内部的控制流，应尽可能避免使用。

if else statement (if else 语句)

Conditional execution of code following the `if` or the `else`, depending on the truth value of the condition.

一种语句，有条件地执行 `if` 或 `else` 后的代码，如何执行取决于条件表达式的真值。

if statement (if 语句)

Conditional execution based on the value of the specified condition. If the condition is true, then the `if` body is executed. If not, control flows to the statement following the `if`.

基于指定条件值的条件分支语句。如果条件为真，则执行 `if` 语句体；否则，控制流转到 `if` 后面的语句。

labeled statement (带标号的语句)

A statement preceded by a label. A label is an identifier followed by a colon.

以标号开头的语句。标号是后面带一个冒号的标识符。

null statement (空语句)

An empty statement. Indicated by a single semicolon.

空白的语句。其语法形式为单个分号。

preprocessor macro (预处理宏)

Function like facility defined by the preprocessor. `assert` is a macro. Modern C++ programs make very little use of the preprocessor macros.

与预处理器定义的设施相似的函数。`assert` 是一个宏。现代 C++ 程序很少使用预处理宏。

raise (引发)

Often used as a synonym for `throw`. C++ programmers speak of "throwing" or "raising" an exception interchangably.

常用作 `throw` 的同义词。C++ 程序员所说的“抛出 (throwing) ”或者“引发 (raising) ”异常表示一样的含义。

switch statement (switch 语句)

A conditional execution statement that starts by evaluating the expression that follows the `switch` keyword. Control passes to the labeled statement with a case label that matches the value of the expression. If there is no matching label, execution either branches to the `default` label, if there is one, or falls out of the `switch` if there is no `default` label.

从计算关键字 `switch` 后面的表达式开始执行的条件分支语句。程序的控制流转跳到与表达式值匹配的 `case` 标号所标记的标号语句。如果没有匹配的标号，则执行 `default` 标号标记的分支，如果没有提供 `default` 分支则结束 `switch` 语句的执行。

terminate

Library function that is called if an exception is not caught. Usually aborts the program.

异常未被捕获时调用的标准库函数。通常会终止程序的执行。

throw expression (throw 表达式)

Expression that interrupts the current execution path. Each `throw` throws an object and transfers control to the nearest enclosing catch clause that can handle the type of exception that is thrown.

中断当前执行路径的表达式。每个 `throw` 都会抛出一个对象，并将控制转换到最近的可处理该类型异常的 `catch` 子句。

try block (try 块)

A block enclosed by the keyword `try` and one or more catch clauses. If the code inside the `try` block raises an exception and one of the catch clauses matches the type of the exception, then the exception is handled by that catch. Otherwise, the exception is handled by an enclosing `try` block or the program terminates.

跟在关键字 `try` 后面的块，以及一个或多个 `catch` 子句。如果 `try` 块中的代码产生了异常，而且该异常类型与其中某个 `catch` 子句匹配，则执行这个 `catch` 子句的语句处理这个异常。否则，异常将由外围 `try` 块处理，或者终止程序。

while loop (while 循环)

Control statement that executes its target statement as long as a specified condition is true. The statement is executed zero or more times, depending on the truth value of the condition.

当指定条件为 `true` 时，执行目标代码的控制语句。根据条件的真值，目标代码可能执行零次或多次。

Chapter 7. Functions

第七章 函数

CONTENTS

Section 7.1 Defining a Function	226
Section 7.2 Argument Passing	229
Section 7.3 The <code>return</code> Statement	245
Section 7.4 Function Declarations	251
Section 7.5 Local Objects	254
Section 7.6 Inline Functions	256
Section 7.7 Class Member Functions	258
Section 7.8 Overloaded Functions	265
Section 7.9 Pointers to Functions	276
Chapter Summary	280
Defined Terms	280

This chapter describes how to define and declare ***functions***. We'll cover how arguments are passed to and values are returned from a function. We'll then look at three special kinds of functions: `inline` functions, class member functions, and overloaded functions. The chapter closes with a more advanced topic: function pointers.

本章将介绍**函数**的定义和声明。其中讨论了如何给函数传递参数以及如何从函数返回值。然后具体分析三类特殊的函数：内联（`inline`）函数、类成员函数和重载函数。最后以一个更高级的话题“函数指针”来结束全章。

A *function* can be thought of as a programmer-defined operation. Like the built-in operators, each function performs some computation and (usually) yields a result. Unlike the operators, functions have names and may take an unlimited number of operands. Like operators, functions can be overloaded, meaning that the same name may refer to multiple different functions.

函数可以看作程序员定义的操作。与内置操作符相同的是，每个函数都会实现一系列的计算，然后（大多数时候）生成一个计算结果。但与操作符不同的是，函数有自己的函数名，而且操作数没有数量限制。与操作符一样，函数可以重载，这意味着同样的函数名可以对应多个不同的函数。

7.1. Defining a Function

7.1. 函数的定义

A function is uniquely represented by a name and a set of operand types. Its operands, referred to as **parameters**, are specified in a comma-separated list enclosed in parentheses. The actions that the function performs are specified in a block, referred to as the **function body**. Every function has an associated **return type**.

函数由函数名以及一组操作数类型唯一地表示。函数的操作数，也即形参，在一对圆括号中声明，形参与形参之间以逗号分隔。函数执行的运算在一个称为函数体的块语句中定义。每一个函数都有一个相关联的返回类型。

As an example, we could write the following function to find the greatest common divisor of two `int`s:

考虑下面的例子，这个函数用来求出两个 `int` 型数的最大公约数：

```
// return the greatest common divisor
int gcd(int v1, int v2)
{
    while (v2) {
        int temp = v2;
        v2 = v1 % v2;
        v1 = temp;
    }
    return v1;
}
```

Here we define a function named `gcd` that returns an `int` and has two `int` parameters. To call `gcd`, we must supply two `int` values and we get an `int` in return.

这里，定义了一个名为 `gcd` 的函数，该函数返回一个 `int` 型值，并带有两个 `int` 型形参。调用 `gcd` 函数时，必须提供两个 `int` 型值传递给函数，然后将得到一个 `int` 型的返回值。

Calling a Function

函数的调用

To invoke a function we use the **call operator**, which is a pair of parentheses. As with any operator, the call operator takes operands and yields a result. The operands to the call operator are the name of the function and a (possibly empty) comma-separated list of **arguments**. The result type of a call is the return type of the called function, and the result itself is the value returned by the function:

C++ 语言使用调用操作符（即一对圆括号）实现函数的调用。正如其他操作符一样，调用操作符需要操作数并产生一个结果。调用操作符的操作数是函数名和一组（有可能是空的）由逗号分隔的实参。函数调用的结果类型就是函数返回值的类型，该运算的结果本身就是函数的返回值：

```
// get values from standard input
cout << "Enter two values: \n";
int i, j;
cin >> i >> j;
// call gcd on arguments i and j
// and print their greatest common divisor
cout << "gcd: " << gcd(i, j) << endl;
```

If we gave this program 15 and 123 as input, the output would be 3.

如果给定 15 和 123 作为程序的输入，程序将输出 3。

Calling a function does two things: It initializes the function parameters from the corresponding arguments and transfers control to the function being invoked. Execution of the *calling* function is suspended and execution of the *called* function begins. Execution of a function begins with the (implicit) definition and initialization of its parameters. That is, when we invoke `gcd`, the first thing that happens is that variables of type `int` named `v1` and `v2` are created. These variables are initialized with the values passed in the call to `gcd`. In this case, `v1` is initialized by the value of `i` and `v2` by the value of `j`.

函数调用做了两件事情：用对应的实参初始化函数的形参，并将控制权转移给被调用函数。主调函数的执行被挂起，被调函数开始执行。函数的运行以形参的（隐式）定义和初始化开始。也就是说，当我们调用 `gcd` 时，第一件事就是创建名为 `v1` 和 `v2` 的 `int` 型变量，并将这两个变量初始化为调用 `gcd` 时传递的实参值。在上例中，`v1` 的初值为

Section 7.1. Defining a Function

i, 而 v2 则初始化为 j 的值。

Function Body Is a Scope

函数体是一个作用域

The body of a function is a statement block, which defines the function's operation. As usual, the block is enclosed by a pair of curly braces and hence forms a new scope. As with any block, the body of a function can define variables. Names defined inside a function body are accessible only within the function itself. Such variables are referred to as [local variables](#). They are "local" to that function; their names are visible only in the scope of the function. They exist only while the function is executing. [Section 7.5](#) (p. 254) covers local variables in more detail.

函数体是一个语句块，定义了函数的具体操作。通常，这个块语句包含在一对花括号中，形成了一个新的作用域。和其他的块语句一样，在函数体中可以定义变量。在函数体内定义的变量只在该函数中才可以访问。这种变量称为[局部变量](#)，它们相对于定义它们的函数而言是“局部”的，其名字只能在该函数的作用域中可见。这种变量只在函数运行时存在。[第 7.5 节](#)将详细讨论局部变量。

Execution completes when a `return` statement is encountered. When the called function finishes, it yields as its result the value specified in the `return` statement. After the return is executed, the suspended, calling function resumes execution at the point of the call. It uses the return value as the result of evaluating the call operator and continues processing whatever remains of the statement in which the call was performed.

当执行到 `return` 语句时，函数调用结束。被调用的函数完成时，将产生一个在 `return` 语句中指定的结果值。执行 `return` 语句后，被挂起的主调函数在调用处恢复执行，并将函数的返回值用作求解调用操作符的结果，继续处理在执行调用的语句中所剩余的工作。

Parameters and Arguments

形参和实参

Like local variables, the parameters of a function provide named, local storage for use by the function. The difference is that parameters are defined inside the function's parameter list and are initialized by arguments passed to the function when the function is called.

类似于局部变量，函数的形参为函数提供了已命名的局部存储空间。它们之间的差别在于形参是在函数的形参表中定义的，并由调用函数时传递函数的实参初始化。

An argument is an expression. It might be a variable, a literal constant or an expression involving one or more operators. We must pass exactly the same number of arguments as the function has parameters. The type of each argument must match the corresponding parameter in the same way that the type of an initializer must match the type of the object it initializes: The argument must have the same type or have a type that can be implicitly converted ([Section 5.12](#), p. 178) to the parameter type. We'll cover how arguments match a parameter in detail in [Section 7.8.2](#) (p. 269).

An argument is an expression. It might be a variable, a literal constant or an expression involving one or more operators. We must pass exactly the same number of arguments as the function has parameters. The type of each argument must match the corresponding parameter in the same way that the type of an initializer must match the type of the object it initializes: The argument must have the same type or have a type that can be implicitly converted ([Section 5.12](#), p. 178) to the parameter type. We'll cover how arguments match a parameter in detail in [Section 7.8.2](#) (p. 269).

实参则是一个表达式。它可以是变量或字面值常量，甚至是包含一个或几个操作符的表达式。在调用函数时，所传递的实参数必须与函数的形参数完全相同。与初始化式的类型必须与初始化对象的类型匹配一样，实参的类型也必须与其对应形参的类型完全匹配：实参必须具有与形参类型相同、或者能隐式转换（[第 5.12 节](#)）为形参类型的数据类型。本章[第 7.8.2 节](#)将详细讨论实参与形参的匹配。

7.1.1. Function Return Type

7.1.1. 函数返回类型

The return type of a function can be a built-in type, such as `int` or `double`, a class type, or a compound type, such as `int&` or `string*`. A return type also can be `void`, which means that the function does not return a value. The following are example definitions of possible function return types:

函数的返回类型可以是内置类型（如 `int` 或者 `double`）、类类型或复合类型（如 `int&` 或 `string*`），还可以是 `void` 类型，表示该函数不返回任何值。下面的例子列出了些可能的函数返回类型：

```
bool is_present(int *, int);      // returns bool
int count(const string &, char);   // returns int
Date &calendar(const char*);     // returns reference to Date
void process();                  // process does not return a value
```

Section 7.1. Defining a Function

A function may not return another function or a built-in array type. Instead, the function may return a pointer to the function or to a pointer to an element in the array:

函数不能返回另一个函数或者内置数组类型，但可以返回指向函数的指针，或指向数组元素的指针的指针：

```
// ok: pointer to first element of the array
int *foo_bar() { /* ... */ }
```

This function returns a pointer to `int` and that pointer could point to an element in an array.

这个函数返回一个 `int` 型指针，该指针可以指向数组中的一个元素。

We'll learn about function pointers in [Section 7.9](#) (p. 276).

[第 7.9 节](#) 将介绍有关函数指针的内容。

Functions Must Specify a Return Type

函数必须指定返回类型

It is illegal to define or declare a function without an explicit return type:

在定义或声明函数时，没有显式指定返回类型是不合法的：

```
// error: missing return type
test(double v1, double v2) { /* ... */ }
```

Earlier versions of C++ would accept this program and implicitly define the return type of `test` as an `int`. Under Standard C++, this program is an error.

早期的 C++ 版本可以接受这样的程序，将 `test` 函数的返回类型隐式地定义为 `int` 型。但在标准 C++ 中，上述程序则是错误的。



In pre-Standard C++, a function without an explicit return type was assumed to return an `int`. C++ programs compiled under earlier, non-standard compilers may still contain functions that implicitly return `int`.

在 C++ 标准化之前，如果缺少显式返回类型，函数的返回值将被假定为 `int` 型。早期未标准化的 C++ 编译器所编译的程序可能依然含有隐式返回 `int` 型的函数。

7.1.2. Function Parameter List

7.1.2. 函数形参表

The parameter list of a function can be empty but cannot be omitted. A function with no parameters can be written either with an empty parameter list or a parameter list containing the single keyword `void`. For example, the following declarations of `process` are equivalent:

函数形参表可以为空，但不能省略。没有任何形参的函数可以用空形参表或含有单个关键字 `void` 的形参表来表示。例如，下面关于 `process` 的声明是等价的：

```
void process() { /* ... */ }      // implicit void parameter list
void process(void){ /* ... */ }   // equivalent declaration
```

A parameter list consists of a comma-separated list of parameter types and (optional) parameter names. Even when the types of two parameters are the same, the type must be repeated:

形参表由一系列用逗号分隔的参数类型和（可选的）参数名组成。如果两个参数具有相同的类型，则其类型必须重复声明：

```
int manip(int v1, v2) { /* ... */ }    // error
int manip(int v1, int v2) { /* ... */ }  // ok
```

Section 7.1. Defining a Function

No two parameters can have the same name. Similarly, a variable local to a function may not use the same name as the name of any of the function's parameters.

参数表中不能出现同名的参数。类似地，局部于函数的变量也不能使用与函数的任意参数相同的名字。

Names are optional, but in a function definition, normally all parameters are named. A parameter must be named to be used.

参数名是可选的，但在函数定义中，通常所有参数都要命名。参数必须在命名后才能使用。

Parameter Type-Checking

参数类型检查



C++ is a statically typed language ([Section 2.3](#), p. 44). The arguments of every call are checked during compilation.

C++ 是一种静态强类型语句 ([第 2.3 节](#))，对于每一次的函数调用，编译时都会检查其实参。

When we call a function, the type of each argument must be either the same type as the corresponding parameter or a type that can be converted ([Section 5.12](#), p. 178) to that type. The function's parameter list provides the compiler with the type information needed to check the arguments. For example, the function `gcd`, which we defined on page [226](#), takes two parameters of type `int`:

调用函数时，对于每一个实参，其类型都必须与对应的形参类型相同，或具有可被转换 ([第 5.12 节](#)) 为该形参类型的类型。函数的形参表为编译器提供了检查实参需要的类型信息。例如，[第 7.1 节](#) 定义的 `gcd` 函数有两个 `int` 型的形参：

```
gcd("hello", "world"); // error: wrong argument types
gcd(24312);           // error: too few arguments
gcd(42, 10, 0);       // error: too many arguments
```

Each of these calls is a compile-time error. In the first call, the arguments are of type `const char*`. There is no conversion from `const char*` to `int`, so the call is illegal. In the second and third calls, `gcd` is passed the wrong number of arguments. The function must be called with two arguments; it is an error to call it with any other number.

以上所有的调用都会导致编译时的错误。在第一个调用中，实参的类型都是 `const char*`，这种类型无法转换为 `int` 型，因此该调用不合法。而第二和第三个调用传递的实参数量有误。在调用该函数时必须提供两个实参，实参数太多或太少都是不合法的。

But what happens if the call supplies two arguments of type `double`? Is this call legal?

如果两个实参都是 `double` 类型，又会怎样呢？调用是否合法？

```
gcd(3.14, 6.29); // ok: arguments are converted to int
```

In C++, the answer is yes; the call is legal. In [Section 5.12.1](#) (p. 179) we saw that a value of type `double` can be converted to a value of type `int`. This call involves such a conversion we want to use `double` values to initialize `int` objects. Therefore, flagging the call as an error would be too severe. Rather, the arguments are implicitly converted to `int` (through truncation). Because this conversion might lose precision, most compilers will issue a warning. In this case, the call becomes

在 C++ 中，答案是肯定的：该调用合法！正如[第 5.12.1 节](#) 所示，`double` 型的值可以转换为 `int` 型的值。本例中的函数调用正涉及了这种转换——用 `double` 型的值来初始化 `int` 型对象。因此，把该调用标记为不合法未免过于严格。更确切地说，（通过截断）`double` 型实参被隐式地转换为 `int` 型。由于这样的转换可能会导致精度损失，大多数编译器都会给出警告。对于本例，该调用实际上变为：

```
gcd(3, 6);
```

and returns a value of 3.

返回值是 3。

A call that passes too many arguments, omits an argument, or passes an argument of the wrong type almost certainly would result in serious run-time errors. Catching these so-called interface errors at compile time greatly reduces the compile-debug-test cycle for large programs.

调用函数时传递过多的实参、忽略某个实参或者传递错误类型的实参，几乎肯定会导致严重的运行时错误！对于大程序，在编译时检查出这些所谓的接口错误 (interface

error) , 将会大大地缩短“编译—调试—测试”的周期。

Exercises Section 7.1.2

Exercise 7.1: What is the difference between a parameter and an argument?

形参和实参有什么区别?

Exercise 7.2: Indicate which of the following functions are in error and why. Suggest how you might correct the problems.

下列哪些函数是错误的? 为什么? 请给出修改意见。

```
(a) int f() {
    string s;
    //...
    return s;
}
(b) f2(int i) { /* ... */ }
(c) int calc(int v1, int v1) /* ... */
(d) double square(double x) return x * x;
```

Exercise 7.3: Write a program to take two `int` parameters and generate the result of raising the first parameter to the power of the second. Write a program to call your function passing it two `ints`. Verify the result.

编写一个带有两个 `int` 型形参的函数, 产生第一个参数的第二个参数次幂的值。编写程序传递两个 `int` 数值调用该函数, 请检验其结果。

Exercise 7.4: Write a program to return the absolute value of its parameter.

编写一个函数, 返回其形参的绝对值。

7.2. Argument Passing

7.2. 参数传递

Each parameter is created anew on each call to the function. The value used to initialize a parameter is the corresponding argument passed in the call.

每次调用函数时，都会重新创建该函数所有的形参，此时所传递的实参将会初始化对应的形参。



Parameters are initialized the same way that variables are. If the parameter has a nonreference type, then the argument is copied. If the parameter is a reference ([Section 2.5](#), p. 58), then the parameter is just another name for the argument.

形参的初始化与变量的初始化一样：如果形参具有非引用类型，则复制实参的值，如果形参为引用类型（[第 2.5 节](#)），则它只是实参的别名。

7.2.1. Nonreference Parameters

7.2.1. 非引用形参

Parameters that are plain, nonreference types are initialized by copying the corresponding argument. When a parameter is initialized with a copy, the function has no access to the actual arguments of the call. It cannot change the arguments. Let's look again at the definition of `gcd`:

普通的非引用类型的参数通过复制对应的实参实现初始化。当用实参副本初始化形参时，函数并没有访问调用所传递的实参本身，因此不会修改实参的值。下面再次观察 `gcd` 这个函数的定义：

```
// return the greatest common divisor
int gcd(int v1, int v2)
{
    while (v2) {
        int temp = v2;
        v2 = v1 % v2;
        v1 = temp;
    }
    return v1;
}
```

Inside the body of the `while`, we change the values of both `v1` and `v2`. However, these changes are made to the local parameters and are not reflected in the arguments used to call `gcd`. Thus, when we call

`while` 循环体虽然修改了 `v1` 与 `v2` 的值，但这些变化仅限于局部参数，而对调用 `gcd` 函数使用的实参没有任何影响。于是，如果有函数调用

```
gcd(i, j)
```

the values `i` and `j` are unaffected by the assignments performed inside `gcd`.

则 `i` 与 `j` 的值不受 `gcd` 内执行的赋值操作的影响。



Nonreference parameters represent local *copies* of the corresponding argument. Changes made to the parameter are made to the local copy. Once the function terminates, these local values are gone.

非引用形参表示对应实参的局部副本。对这类形参的修改仅仅改变了局部副本的值。一旦函数执行结束，这些局部变量的值也就没有了。

Section 7.2. Argument Passing

Pointer Parameters

指针形参

A parameter can be a pointer ([Section 4.2](#), p. 114), in which case the argument pointer is copied. As with any nonreference type parameter, changes made to the parameter are made to the local copy. If the function assigns a new pointer value to the parameter, the calling pointer value is unchanged.

函数的形参可以是指针（[第 4.2 节](#)），此时将复制实参指针。与其他非引用类型的形参一样，该类形参的任何改变也仅作用于局部副本。如果函数将新指针赋给形参，主调函数使用的实参指针的值没有改变。

Recalling the discussion in [Section 4.2.3](#) (p. 121), the fact that the pointer is copied affects only assignments to the pointer. If the function takes a pointer to a non`const` type, then the function can assign through the pointer and change the value of the object to which the pointer points:

回顾[第 4.2.3 节](#)的讨论，事实上被复制的指针只影响对指针的赋值。如果函数形参是非`const`类型的指针，则函数可通过指针实现赋值，修改指针所指向对象的值：

```
void reset(int *ip)
{
    *ip = 0; // changes the value of the object to which ip points
    ip = 0; // changes only the local value of ip; the argument is unchanged
}
```

After a call to `reset`, the argument is unchanged but the object to which the argument points will be 0:

调用`reset`后，实参依然保持原来的值，但它所指向的对象的值将变为0：

```
int i = 42;
int *p = &i;
cout << "i: " << *p << '\n'; // prints i: 42
reset(p); // changes *p but not p
cout << "i: " << *p << endl; // ok: prints i: 0
```

If we want to prevent changes to the value to which the pointer points, then the parameter should be defined as a pointer to `const`:

如果保护指针指向的值，则形参需定义为指向`const`对象的指针：

```
void use_ptr(const int *p)
{
    // use_ptr may read but not write to *p
}
```

Whether a pointer parameter points to a `const` or non`const` type affects the arguments that we can use to call the function. We can call `use_ptr` on either an `int*` or a `const int*`; we can pass only on an `int*` to `reset`. This distinction follows from the initialization rules for pointers ([Section 4.2.5](#), p. 126). We may initialize a pointer to `const` to point to a non`const` object but may not use a pointer to non`const` to point to a `const` object.

指针形参是指向`const`类型还是非`const`类型，将影响函数调用所使用的实参。我们既可以用`int*`也可以用`const int*`类型的实参调用`use_ptr`函数；但仅能将`int*`类型的实参传递给`reset`函数。这个差别来源于指针的初始化规则（[第 4.2.5 节](#)）。可以将指向`const`对象的指针初始化为指向非`const`对象，但不可以让指向非`const`对象的指针向`const`对象。

`const` Parameters

`const` 形参

We can call a function that takes a nonreference, non`const` parameter passing either a `const` or non`const` argument. For example, we could pass two `const` `ints` to our `gcd` function:

在调用函数时，如果该函数使用非引用的非`const`形参，则既可给该函数传递`const`实参也可传递非`const`的实参。例如，可以传递两个`int`型`const`对象调用`gcd`：

```
const int i = 3, j = 6;
int k = rgcd(3, 6); // ok: k initialized to 3
```

This behavior follows from the normal initialization rules for `const` objects ([Section 2.4](#), p. 56). Because the initialization copies the value of the initializer, we can initialize a non`const` object from a `const` object, or vice versa.

Section 7.2. Argument Passing

这种行为源于 `const` 对象的标准初始化规则（[第 2.4 节](#)）。因为初始化复制了初始化式的值，所以可用 `const` 对象初始化非 `const` 对象，反之亦然。

If we make the parameter a `const` nonreference type:

如果将形参定义为非引用的 `const` 类型：

```
void fcn(const int i) { /* fcn can read but not write to i */ }
```

then the function cannot change its local copy of the argument. The argument is still passed as a copy so we can pass `fcn` either a `const` or non`const` object.

则在函数中，不可以改变实参的局部副本。由于实参仍然是以副本的形式传递，因此传递给 `fcn` 的既可以是 `const` 对象也可以是非 `const` 对象。

What may be surprising, is that although the parameter is a `const` inside the function, the compiler otherwise treats the definition of `fcn` as if we had defined the parameter as a plain `int`:

令人吃惊的是，尽管函数的形参是 `const`，但是编译器却将 `fcn` 的定义视为其形码被声明为普通的 `int` 型：

```
void fcn(const int i) { /* fcn can read but not write to i */ }
void fcn(int i) { /* ... */ }           // error: redefines fcn(int)
```

This usage exists to support compatibility with the C language, which makes no distinction between functions taking `const` or non`const` parameters.

这种用法是为了支持对 C 语言的兼容，因为在 C 语言中，具有 `const` 形参或非 `const` 形参的函数并无区别。

Limitations of Copying Arguments

复制实参的局限性

Copying an argument is not suitable for every situation. Cases where copying doesn't work include:

复制实参并不是在所有的情况下都适合，不适宜复制实参的情况包括：

- When we want the function to be able to change the value of an argument.

当需要在函数中修改实参的值时。

- When we want to pass a large object as an argument. The time and space costs to copy the object are often too high for real-world applications.

当需要以大型对象作为实参传递时。对实际的应用而言，复制对象所付出的时间和存储空间代价往往过大。

- When there is no way to copy the object.

当没有办法实现对象的复制时。

In these cases we can instead define the parameters as references or pointers.

对于上述几种情况，有效的解决办法是将形参定义为引用或指针类型。

Exercises Section 7.2.1

Exercise 7.5: Write a function that takes an `int` and a pointer to an `int` and returns the larger of the `int` value of the value to which the pointer points. What type should you use for the pointer?

编写一个函数，该函数具有两个形参，分别为 `int` 型和指向 `int` 型的指针，并返回这两个 `int` 值之中较大的数值。考虑应将其指针形参定义为什么类型？

Exercise 7.6: Write a function to swap the values pointed to by two pointers to `int`. Test the function by calling it and printing the swapped values.

编写函数交换两个 `int` 型指针所指向的值，调用并检验该函数，输出交换后的值。

7.2.2. Reference Parameters

7.2.2. 引用形参

As an example of a situation where copying the argument doesn't work, consider a function to swap the values of its two arguments:

考虑下面不适宜复制实参的例子，该函数希望交换两个实参的值：

```
// incorrect version of swap: The arguments are not changed!
void swap(int v1, int v2)
{
    int tmp = v2;
    v2 = v1;    // assigns new value to local copy of the argument
    v1 = tmp;   // local objects v1 and v2 no longer exist
}
```

In this case, we want to change the arguments themselves. As defined, though, `swap` cannot affect those arguments. When it executes, `swap` exchanges the *local copies* of its arguments. The arguments passed to `swap` are unchanged:

这个例子期望改变实参本身的值。但对于上述的函数定义，`swap` 无法影响实参本身。执行 `swap` 时，只交换了其实参的局部副本，而传递 `swap` 的实参并没有修改：

```
int main()
{
    int i = 10;
    int j = 20;
    cout << "Before swap():\ti: "
        << i << "\tj: " << j << endl;
    swap(i, j);
    cout << "After swap():\ti: "
        << i << "\tj: " << j << endl;
    return 0;
}
```

Compiling and executing this program results in the following output:

编译并执行程序，产生如下输出结果：

```
Before swap(): i: 10 j: 20
After swap(): i: 10 j: 20
```

For `swap` to work as intended and swap the values of its arguments, we need to make the parameters references:

为了使 `swap` 函数以期望的方式工作，交换实参的值，需要将形参定义为引用类型：

```
// ok: swap acts on references to its arguments
void swap(int &v1, int &v2)
{
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
}
```

Like all references, reference parameters refer directly to the objects to which they are bound rather than to copies of those objects. When we define a reference, we must initialize it with the object to which the reference will be bound. Reference parameters work exactly the same way. Each time the function is called, the reference parameter is created and bound to its corresponding argument. Now, when we call `swap`

与所有引用一样，引用形参直接关联到其所绑定的变量，而非这些对象的副本。定义引用时，必须用与该引用绑定的对象初始化该引用。引用形参完全以相同的方式工作。每次调用函数，引用形参被创建并与相应实参关联。此时，当调用 `swap`

```
swap(i, j);
```

the parameter `v1` is just another name for the object `i` and `v2` is another name for `j`. Any change to `v1` is actually a change to the argument `i`. Similarly, changes to `v2` are actually made to `j`. If we recompile `main` using this revised version of `swap`, we can see that the output is now correct:

形参 `v1` 只是对象 `i` 的另一个名字，而 `v2` 则是对象 `j` 的另一个名字。对 `v1` 的任何修改实际上也是对 `i` 的修改。同样地，`v2` 上的任何修改实际上也是对 `j` 的修改。重新编译使用 `swap` 的这个修订版本的 `main` 函数后，可以看到输出结果是正确的：

```
Before swap(): i: 10 j: 20
After swap(): i: 20 j: 10
```



Programmers who come to C++ from a C background are used to passing pointers to obtain access to the argument. In C++ it is safer and more natural to use reference parameters.

从 C 语言背景转到 C++ 的程序员习惯通过传递指针来实现对实参的访问。在 C++ 中，使用引用形参则更安全和更自然。

Using Reference Parameters to Return Additional Information

使用引用形参返回额外的信息

We've seen one example, `swap`, in which reference parameters were used to allow the function to change the value of its arguments. Another use of reference parameters is to return an additional result to the calling function.

通过对 `swap` 这个例子的讨论，了解了如何利用引用形参让函数修改实参的值。引用形参的另一种用法是向主调函数返回额外的结果。

Functions can return only a single value, but sometimes a function has more than one thing to return. For example, let's define a function named `find_val` that searches for a particular value in the elements of a `vector` of integers. It returns an iterator that refers to the element, if the element was found, or to the `end` value if the element isn't found. We'd also like the function to return an occurrence count if the value occurs more than once. In this case the iterator returned should be to the first element that has the value for which we're looking.

函数只能返回单个值，但有些时候，函数有不止一个的内容需要返回。例如，定义一个 `find_val` 函数。在一个整型 `vector` 对象的元素中搜索某个特定值。如果找到满足要求的元素，则返回指向该元素的迭代器；否则返回一个迭代器，指向该 `vector` 对象的 `end` 操作返回的元素。此外，如果该值出现了不止一次，我们还希望函数可以返回其出现的次数。在这种情况下，返回的迭代器应该指向具有要寻找的值的第一个元素。

How can we define a function that returns both an iterator and an occurrence count? We could define a new type that contains an iterator and a count. An easier solution is to pass an additional reference argument that `find_val` can use to return a count of the number of occurrences:

如何定义既返回一个迭代器又返回出现次数的函数？我们可以定义一种包含一个迭代器和一个计数器的新类型。而更简便的解决方案是给 `find_val` 传递一个额外的引用实参，用于返回出现次数的统计结果：

```
// returns an iterator that refers to the first occurrence of value
// the reference parameter occurs contains a second return value
vector<int>::const_iterator find_val(
    vector<int>::const_iterator beg,           // first element
    vector<int>::const_iterator end,          // one past last element
    int value,                            // the value we want
    vector<int>::size_type &occurs)        // number of times it occurs
{
    // res_iter will hold first occurrence, if any
    vector<int>::const_iterator res_iter = end;
    occurs = 0; // set occurrence count parameter
    for ( ; beg != end; ++beg)
        if (*beg == value) {
            // remember first occurrence of value
            if (res_iter == end)
                res_iter = beg;
            ++occurs; // increment occurrence count
        }
    return res_iter; // count returned implicitly in occurs
}
```

When we call `find_val`, we have to pass four arguments: a pair of iterators that denote the range of elements (Section 9.2.1, p. 314) in the `vector` in which to look, the value to look for, and a `size_type` (Section 3.2.3, p. 84) object to hold the occurrence count. Assuming `ivec` is a `vector<int>`, `it` is an iterator of the right type, and `ctr` is a `size_type`, we could call `find_val` as follows:

调用 `find_val` 时，需传递四个实参：一对标志 `vector` 对象中要搜索的元素范围（第 9.2.1 节）的迭代器，所查找的值，以及用于存储出现次数的 `size_type` 类型（第 3.2.3 节）对象。假设 `ivec` 是 `vector<int>`，`it` 类型的对象，`it` 是一个适当类型的迭代器，而 `ctr` 则是 `size_type` 类型的变量，则可如此调用该函数：

```
it = find_val(ivec.begin(), ivec.end(), 42, ctr);
```

After the call, the value of `ctr` will be the number of times 42 occurs, and `it` will refer to the first occurrence if there is one. Otherwise, `it` will be equal to `ivec.end()` and `ctr` will be zero.

调用后，`ctr` 的值将是 42 出现的次数，如果 42 在 `ivec` 中出现了，则 `it` 将指向其第一次出现的位置；否则，`it` 的值为 `ivec.end()`，而 `ctr` 则为 0。

Using (`const`) References to Avoid Copies

利用 `const` 引用避免复制

The other circumstance in which reference parameters are useful is when passing a large object to a function. Although copying an argument is okay for objects of built-in data types and for objects of class types that are small in size, it is (often) too inefficient for objects of most class types or large arrays. Moreover, as we'll learn in [Chapter 13](#), some class types cannot be copied. By using a reference parameter, the function can access the object directly without copying it.

在向函数传递大型对象时，需要使用引用形参，这是引用形参适用的另一种情况。虽然复制实参对于内置数据类型的对象或者规模较小的类类型对象来说没有什么问题，但是对于大部分的类类型或者大型数组，它的效率（通常）太低了；此外，我们将在[第十三章](#)学习到，某些类类型是无法复制的。使用引用形参，函数可以直接访问实参对象，而无须复制它。

As an example, we'll write a function that compares the length of two `string`s. Such a function needs to access the `size` of each `string` but has no need to write to the `string`s. Because `string`s can be long, we'd like to avoid copying them. Using `const` references we can avoid the copy:

编写一个比较两个 `string` 对象长度的函数作为例子。这个函数需要访问每个 `string` 对象的 `size`，但不必修改这些对象。由于 `string` 对象可能相当长，所以我们希望避免复制操作。使用 `const` 引用就可避免复制：

```
// compare the length of two strings
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

Each parameter is a reference to `const string`. Because the parameters are references the arguments are not copied. Because the parameters are `const` references, `isShorter` may not use the references to change the arguments.

其每一个形参都是 `const string` 类型的引用。因为形参是引用，所以不复制实参。又因为形参是 `const` 引用，所以 `isShorter` 函数不能使用该引用来修改实参。



When the only reason to make a parameter a reference is to avoid copying the argument, the parameter should be `const` reference.

如果使用引用形参的唯一目的是避免复制实参，则应将形参定义为 `const` 引用。

References to `const` Are More Flexible

更灵活的指向 `const` 的引用

It should be obvious that a function that takes a plain, non`const` reference may not be called on behalf of a `const` object. After all, the function might change the object it is passed and thus violate the `constness` of the argument.

如果函数具有普通的非 `const` 引用形参，则显然不能通过 `const` 对象进行调用。毕竟，此时函数可以修改传递进来的对象，这样就违背了实参的 `const` 特性。

What may be less obvious is that we also cannot call such a function with an rvalue ([Section 2.3.1](#), p. 45) or with an object of a type that requires a conversion:

但比较容易忽略的是，调用这样的函数时，传递一个右值（[第 2.3.1 节](#)）或具有需要转换的类型的对象同样是不允许的：

```
// function takes a non-const reference parameter
int incr(int &val)
{
    return ++val;
}
int main()
{
    short v1 = 0;
    const int v2 = 42;
    int v3 = incr(v1);    // error: v1 is not an int
    v3 = incr(v2);        // error: v2 is const
    v3 = incr(0);          // error: literals are not lvalues
    v3 = incr(v1 + v2);   // error: addition doesn't yield an lvalue
    int v4 = incr(v3);    // ok: v3 is a non const object type int
}
```

Section 7.2. Argument Passing

The problem is that a `nonconst` reference ([Section 2.5, p. 59](#)) may be bound only to `nonconst` object of exactly the same type.

问题的关键是非 `const` 引用形参 ([第 2.5 节](#)) 只能与完全同类型的非 `const` 对象关联。

Parameters that do not change the value of the corresponding argument should be `const` references. Defining such parameters as `nonconst` references needlessly restricts the usefulness of a function. As an example, we might write a program to find a given character in a `string`:

```
// returns index of first occurrence of c in s or s.size() if c isn't in s
// Note: s doesn't change, so it should be a reference to const
string::size_type find_char(string &s, char c)
{
    string::size_type i = 0;
    while (i != s.size() && s[i] != c)
        ++i; // not found, look at next character
    return i;
}
```

This function takes its `string` argument as a plain (`nonconst`) reference, even though it doesn't modify that parameter. One problem with this definition is that we cannot call it on a character string literal:

这个函数将其 `string` 类型的实参当作普通 (非 `const`) 的引用，尽管函数并没有修改这个形参的值。这样的定义带来的问题是不能通过字符串字面值来调用这个函数：

```
if (find_char("Hello World", 'o')) // ...
```

This call fails at compile time, even though we can convert the literal to a `string`.

虽然字符串字面值可以转换为 `string` 对象，但上述调用仍然会导致编译失败。

Such problems can be surprisingly pervasive. Even if our program has no `const` objects and we only call `find_char` on behalf of `string` objects (as opposed to on a string literal or an expression that yields a `string`), we can encounter compile-time problems. For example, we might have another function `is_sentence` that wants to use `find_char` to determine whether a `string` represents a sentence:

继续将这个问题延伸下去会发现，即使程序本身没有 `const` 对象，而且只使用 `string` 对象 (而非字符串字面值或产生 `string` 对象的表达式) 调用 `find_char` 函数，编译阶段的问题依然会出现。例如，可能有另一个函数 `is_sentence` 调用 `find_char` 来判断一个 `string` 对象是否是句子：

```
bool is_sentence (const string &s)
{
    // if there's a period and it's the last character in s
    // then s is a sentence
    return (find_char(s, '.') == s.size() - 1);
}
```

As written, the call to `find_char` from inside `is_sentence` is a compile-time error. The parameter to `is_sentence` is a reference to `const string` and cannot be passed to `find_char`, which expects a reference to a `nonconst string`.

如上代码，函数 `is_sentence` 中 `find_char` 的调用是一个编译错误。传递进 `is_sentence` 的形参是指向 `const string` 对象的引用，不能将这种类型的参数传递给 `find_char`，因为后者期待得到一个指向非 `const string` 对象的引用。



Reference parameters that are not changed should be references to `const`. Plain, `nonconst` reference parameters are less flexible. Such parameters may not be initialized by `const` objects, or by arguments that are literals or expressions that yield rvalues.

应该将不需要修改的引用形参定义为 `const` 引用。普通的非 `const` 引用形参在使用时不太灵活。这样的形参既不能用 `const` 对象初始化，也不能用字面值或产生右值的表达式实参初始化。

Passing a Reference to a Pointer

传递指向指针的引用

Suppose we want to write a function that swaps two pointers, similar to the program we wrote earlier that swaps two integers. We know that we

Section 7.2. Argument Passing

use `*` to define a pointer and `&` to define a reference. The question here is how to combine these operators to obtain a reference to a pointer. Here is an example:

```
// swap values of two pointers to int
void ptrswap(int *&v1, int *&v2)
{
    int *tmp = v2;
    v2 = v1;
    v1 = tmp;
}
```

The parameter

形参

```
int *&v1
```

should be read from right to left: `v1` is a reference to a pointer to an object of type `int`. That is, `v1` is just another name for whatever pointer is passed to `ptrswap`.

的定义应从右至左理解: `v1` 是一个引用, 与指向 `int` 型对象的指针相关联。也就是说, `v1` 只是传递进 `ptrswap` 函数的任意指针的别名。

We could rewrite the `main` function from page 233 to use `ptrswap` and swap pointers to the values 10 and 20:

重写第 7.2.2 节的 `main` 函数, 调用 `ptrswap` 交换分别指向值 10 和 20 的指针:

```
int main()
{
    int i = 10;
    int j = 20;
    int *pi = &i; // pi points to i
    int *pj = &j; // pj points to j
    cout << "Before ptrswap():\t*pi: "
        << *pi << "\t*pj: " << *pj << endl;

    ptrswap(pi, pj); // now pi points to j; pj points to i
    cout << "After ptrswap():\t*pi: "
        << *pi << "\t*pj: " << *pj << endl;
    return 0;
}
```

When compiled and executed, the program generates the following output:

编译并执行后, 该程序产生如下结果:

```
Before ptrswap(): *pi: 10 *pj: 20
After ptrswap(): *pi: 20 *pj: 10
```

What happens is that the `pointer` values are swapped. When we call `ptrswap`, `pi` points to `i` and `pj` points to `j`. Inside `ptrswap` the pointers are swapped so that after `ptrswap`, `pi` points to the object `pj` had addressed. In other words, `pi` now points to `j`. Similarly, `pj` points to `i`.

即指针的值被交换了。在调用 `ptrswap` 时, `pi` 指向 `i`, 而 `pj` 则指向 `j`。在 `ptrswap` 函数中, 指针被交换, 使得调用 `ptrswap` 结束后, `pi` 指向了原来 `pj` 所指向的对象。换句话说, 现在 `pi` 指向 `j`, 而 `pj` 则指向了 `i`。

Exercises Section 7.2.2

Exercise 7.7: Explain the difference in the following two parameter declarations:

解释下面两个形参声明的不同之处:

```
void f(T);
void f(T&);
```

Exercise 7.8: Give an example of when a parameter should be a reference type. Give an example of when a parameter should not be a reference.

举一个例子说明什么时候应该将形参定义为引用类型。再举一个例子说明什么时候不应该将形参定义为引用。

Section 7.2. Argument Passing

Exercise 7.9: Change the declaration of `occurs` in the parameter list of `find_val` (defined on page 234) to be a nonreference argument type and rerun the program. How does the behavior of the program change?

将第 7.2.2 节定义的 `find_val` 函数的形参表中 `occurs` 的声明修改为非引用参数类型，并重新执行这个程序，该函数的行为发生了什么改变？

Exercise 7.10: The following program, although legal, is less useful than it might be. Identify and correct the limitation on this program:

下面的程序虽然是合法的，但可用性还不够好，指出并改正该程序的局限：

```
bool test(string& s) { return s.empty(); }
```

Exercise 7.11: When should reference parameters be `const`? What problems might arise if we make a parameter a plain reference when it could be a `const` reference?

何时应将引用形参定义为 `const` 对象？如果在需要 `const` 引用时，将形参定义为普通引用，则会出现什么问题？

7.2.3. `vector` and Other Container Parameters

7.2.3. `vector` 和其他容器类型的形参



Ordinarily, functions should not have `vector` or other library container parameters. Calling a function that has a plain, nonreference `vector` parameter will copy every element of the `vector`.

通常，函数不应该有 `vector` 或其他标准库容器类型的形参。调用含有普通的非引用 `vector` 形参的函数将会复制 `vector` 的每一个元素。

In order to avoid copying the `vector`, we might think that we'd make the parameter a reference. However, for reasons that will be clearer after reading [Chapter 11](#), in practice, C++ programmers tend to pass containers by passing iterators to the elements we want to process:

从避免复制 `vector` 的角度出发，应考虑将形参声明为引用类型。然而，看过[第十一章](#)后我们会知道，事实上，C++ 程序员倾向于通过传递指向容器中需要处理的元素的迭代器来传递容器：

```
// pass iterators to the first and one past the last element to print
void print(vector<int>::const_iterator beg,
           vector<int>::const_iterator end)
{
    while (beg != end) {
        cout << *beg++;
        if (beg != end) cout << " ";
    }
    cout << endl;
}
```

This function prints the elements starting with one referred to by `beg` up to but not including the one referred to by `end`. We print a space after each element but the last.

这个函数将输出从 `beg` 指向的元素开始到 `end` 指向的元素（不含）为止的范围内所有的元素。除了最后一个元素外，每个元素后面都输出一个空格。

7.2.4. Array Parameters

7.2.4. 数组形参

Arrays have two special properties that affect how we define and use functions that operate on arrays: We cannot copy an array ([Section 4.1.1](#), p. 112) and when we use the name of an array it is automatically converted to a pointer to the first element ([Section 4.2.4](#), p. 122). Because we

Section 7.2. Argument Passing

cannot copy an array, we cannot write a function that takes an array type parameter. Because arrays are automatically converted to pointers, functions that deal with arrays usually do so indirectly by manipulating pointers to elements in the array.

数组有两个特殊的性质，影响我们定义和使用作用在数组上的函数：一是不能复制数组（[第 4.1.1 节](#)）；二是使用数组名字时，数组名会自动转化为指向其第一个元素的指针（[第 4.2.4 节](#)）。因为数组不能复制，所以无法编写使用数组类型形参的函数。因为数组会被自动转化为指针，所以处理数组的函数通常通过操纵指向数组指向数组中的元素的指针来处理数组。

Defining an Array Parameter

数组形参的定义

Let's assume that we want to write a function that will print the contents of an array of `ints`. We could specify the array parameter in one of three ways:

如果要编写一个函数，输出 `int` 型数组的内容，可用下面三种方式指定数组形参：

```
// three equivalent definitions of printValues
void printValues(int*) { /* ... */ }
void printValues(int[]) { /* ... */ }
void printValues(int[10]) { /* ... */ }
```

Even though we cannot pass an array directly, we can write a function parameter that looks like an array. Despite appearances, a parameter that uses array syntax is treated as if we had written a pointer to the array element type. These three definitions are equivalent; each is interpreted as taking a parameter of type `int*`.

虽然不能直接传递数组，但是函数的形参可以写成数组的形式。虽然形参表示方式不同，但可将使用数组语法定义的形参看作指向数组元素类型的指针。上面的三种定义是等价的，形参类型都是 `int*`。



It is usually a good idea to define array parameters as pointers, rather than using the array syntax. Doing so makes it clear that what is being operated on is a pointer to an array element, not the array itself. Because an array dimension is ignored, including a dimension in a parameter definition is particularly misleading.

通常，将数组形参直接定义为指针要比使用数组语法定义更好。这样就明确地表示，函数操纵的是指向数组元素的指针，而不是数组本身。由于忽略了数组长度，形参定义中如果包含了数组长度则特别容易引起误解。

Parameter Dimensions Can Be Misleading

形参的长度会引起误解

The compiler ignores any dimension we might specify for an array parameter. Relying, incorrectly, on the dimension, we might write `printValues` as 编译器忽略为任何数组形参指定的长度。根据数组长度（权且这样说），可将函数 `printValues` 编写为：

```
// parameter treated as const int*, size of array is ignored
void printValues(const int ia[10])
{
    // this code assumes array has 10 elements;
    // disaster if argument has fewer than 10 elements!
    for (size_t i = 0; i != 10; ++i)
    {
        cout << ia[i] << endl;
    }
}
```

Although this code *assumes* that the array it is passed has at least 10 elements, nothing in the language enforces that assumption. The following calls are all legal:

尽管上述代码假定所传递的数组至少含有 10 个元素，但 C++ 语言没有任何机制强制实现这个假设。下面的调用都是合法的：

```
int main()
{
    int i = 0, j[2] = {0, 1};
    printValues(&i);      // ok: &i is int*; probable run-time error
```

Section 7.2. Argument Passing

```
printValues(j);      // ok: j is converted to pointer to 0th
                     // element; argument has type int*;
                     // probable run-time error
    return 0;
}
```

Even though the compiler issues no complaints, both calls are in error, and probably will fail at run time. In each case, memory beyond the array will be accessed because `printValues` assumes that the array it is passed has at least 10 elements. Depending on the values that happen to be in that memory, the program will either produce spurious output or crash.

虽然编译没有问题，但是这两个调用都是错误的，可能导致运行失败。在这两个调用中，由于函数 `printValues` 假设传递进来的数组至少含有 10 个元素，因此造成数组内的越界访问。程序的执行可能产生错误的输出，也可能崩溃，这取决于越界访问的内存中恰好存储的数值是什么。



When the compiler checks an argument to an array parameter, it checks only that the argument is a pointer and that the types of the pointer and the array elements match. The size of the array is not checked.

当编译器检查数组形参关联的实参时，它只会检查实参是不是指针、指针的类型和数组元素的类型时是否匹配，而不会检查数组的长度。

Array Arguments

数组实参

As with any other type, we can define an array parameter as a reference or nonreference type. Most commonly, arrays are passed as plain, nonreference types, which are quietly converted to pointers. As usual, a nonreference type parameter is initialized as a copy of its corresponding argument. When we pass an array, the argument is a pointer to the first element in the array. That pointer value is copied; the array elements themselves are not copied. The function operates on a copy of the pointer, so it cannot change the value of the argument pointer. The function can, however, use that pointer to change the element values to which the pointer points. Any changes through the pointer parameter are made to the array elements themselves.

和其他类型一样，数组形参可定义为引用或非引用类型。大部分情况下，数组以普通的非引用类型传递，此时数组会悄悄地转换为指针。一般来说，非引用类型的形参会初始化为其相应实参的副本。而在传递数组时，实参是指向数组第一个元素的指针，形参复制的是这个指针的值，而不是数组元素本身。函数操纵的是指针的副本，因此不会修改实参指针的值。然而，函数可通过该指针改变它所指向的数组元素的值。通过指针形参做的任何改变都在修改数组元素本身。



Functions that do not change the elements of their array parameter should make the parameter a pointer to `const`:

不需要修改数组形参的元素时，函数应该将形参定义为指向 `const` 对象的指针：

```
// f won't change the elements in the array
void f(const int*) { /* ... */ }
```

Passing an Array by Reference

通过引用传递数组

As with any type, we can define an array parameter as a reference to the array. If the parameter is a reference to the array, then the compiler does not convert an array argument into a pointer. Instead, a reference to the array itself is passed. In this case, the array size is part of the parameter and argument types. The compiler will check that the size of the array argument matches the size of the parameter:

和其他类型一样，数组形参可声明为数组的引用。如果形参是数组的引用，编译器不会将数组实参转化为指针，而是传递数组的引用本身。在这种情况下，数组大小成为形参和实参类型的一部分。编译器检查数组的实参的大小与形参的大小是否匹配：

```
// ok: parameter is a reference to an array; size of array is fixed
```

Section 7.2. Argument Passing

```
void printValues(int (&arr)[10]) { /* ... */ }
int main()
{
    int i = 0, j[2] = {0, 1};
    int k[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    printValues(&i); // error: argument is not an array of 10 ints
    printValues(j); // error: argument is not an array of 10 ints
    printValues(k); // ok: argument is an array of 10 ints
    return 0;
}
```

This version of `printValues` may be called only for arrays of exactly 10 `ints`, limiting which arrays can be passed. However, because the parameter is a reference, it is safe to rely on the size in the body of the function:

这个版本的 `printValues` 函数只严格地接受含有 10 个 `int` 型数值的数组，这限制了哪些数组可以传递。然而，由于形参是引用，在函数体中依赖数组的大小是安全的：

```
// ok: parameter is a reference to an array; size of array is fixed
void printValues(int (&arr)[10])
{
    for (size_t i = 0; i != 10; ++i) {
        cout << arr[i] << endl;
    }
}
```



The parentheses around `&arr` are necessary because of the higher precedence of the subscript operator:

`&arr` 两边的圆括号是必需的，因为下标操作符具有更高的优先级：

```
f(int &arr[10]) // error: arr is an array of references
f(int (&arr)[10]) // ok: arr is a reference to an array of 10 ints
```

We'll see in [Section 16.1.5](#) (p. 632) how we might write this function in a way that would allow us to pass a reference parameter to an array of any size.

在[第 16.1.5 节](#)将会介绍如何重新编写此函数，允许传递指向任意大小的数组的引用形参。

Passing a Multidimensioned Array

多维数组的传递

Recall that there are no multidimensioned arrays in C++ ([Section 4.4](#), p. 141). Instead, what appears to be a multidimensioned array is an array of arrays.

回顾前面我们说过在 C++ 中没有多维数组（[第 4.4 节](#)）。所谓多维数组实际是指数组的数组。

As with any array, a multidimensioned array is passed as a pointer to its zeroth element. An element in a multidimensioned array is an array. The size of the second (and any subsequent dimensions) is part of the element type and must be specified:

和其他数组一样，多维数组以指向 0 号元素的指针方式传递。多维数组的元素本身就是数组。除了第一维以外的所有维的长度都是元素类型的一部分，必须明确指定：

```
// first parameter is an array whose elements are arrays of 10 ints
void printValues(int (matrix*)[10], int rowSize);
```

declares `matrix` as a pointer to an array of ten `ints`.

上面的语句将 `matrix` 声明为指向含有 10 个 `int` 型元素的数组的指针。



Again, the parentheses around `*matrix` are necessary:

再次强调，`*matrix` 两边的圆括号是必需的：

Section 7.2. Argument Passing

```
int *matrix[10]; // array of 10 pointers
int (*matrix)[10]; // pointer to an array of 10 ints
```

We could also declare a multidimensioned array using array syntax. As with a single-dimensioned array, the compiler ignores the first dimension and so it is best not to include it:

我们也可以用数组语法定义多维数组。与一维数组一样，编译器忽略第一维的长度，所以最好不要把它包括在形参表内：

```
// first parameter is an array whose elements are arrays of 10 ints
void printValues(int matrix[][10], int rowSize);
```

declares `matrix` to be what looks like a two-dimensioned array. In fact, the parameter is a pointer to an element in an array of arrays. Each element in the array is itself an array of ten `ints`.

这条语句把 `matrix` 声明为二维数组的形式。实际上，形参是一个指针，指向数组的数组中的元素。数组中的每个元素本身就是含有 10 个 `int` 型对象的数组。

7.2.5. Managing Arrays Passed to Functions

7.2.5. 传递给函数的数组的处理

As we've just seen, type checking for a nonreference array parameter confirms only that the argument is a pointer of the same type as the elements in the array. Type checking does not verify that the argument actually points to an array of a specified size.

就如刚才所见的，非引用数组形参的类型检查只是确保实参是和数组元素具有同样类型的指针，而不会检查实参实际上是否指向指定大小的数组。



It is up to any program dealing with an array to ensure that the program stays within the bounds of the array.

任何处理数组的程序都要确保程序停留在数组的边界内。

There are three common programming techniques to ensure that a function stays within the bounds of its array argument(s). The first places a marker in the array itself that can be used to detect the end of the array. C-style character strings are an example of this approach. C-style strings are arrays of characters that encode their termination point with a null character. Programs that deal with C-style strings use this marker to stop processing elements in the array.

有三种常见的编程技巧确保函数的操作不超出数组实参的边界。第一种方法是在数组本身放置一个标记来检测数组的结束。C 风格字符串就是采用这种方法的一个例子，它是一种字符数组，并且以空字符 `null` 作为结束的标记。处理 C 风格字符串的程序就是使用这个标记停止数组元素的处理。

Using the Standard Library Conventions

使用标准库规范

A second approach is to pass pointers to the first and one past the last element in the array. This style of programming is inspired by techniques used in the standard library. We'll learn more about this style of programming in [Part II](#).

第二种方法是传递指向数组第一个和最后一个元素的下一个位置的指针。这种编程风格由标准库所使用的启发而得，在[第二部分](#)将会进一步介绍这种编程风格。

Using this approach, we could rewrite `printValues` and call the new version as follows:

使用这种方法重写函数 `printValues` 并调用该函数，如下所示：

```
void printValues(const int *beg, const int *end)
{
```

Section 7.2. Argument Passing

```
while (beg != end) {
    cout << *beg++ << endl;
}
int main()
{
    int j[2] = {0, 1};
    // ok: j is converted to pointer to 0th element in j
    //      j + 2 refers one past the end of j
    printValues(j, j + 2);
    return 0;
}
```

The loop inside `printValues` looks like other programs we've written that used `vector` iterators. We march the `beg` pointer one element at a time through the array. We stop the loop when `beg` is equal to the end marker, which was passed as the second parameter to the function.

`printValues` 中的循环很像用 `vector` 迭代器编写的程序。每次循环都使 `beg` 指针指向下一个元素，从而实现数组的遍历。当 `beg` 指针等于结束标记时，循环结束。结束标记就是传递给函数的第二个形参。

When we call this version, we pass two pointers: one to the first element we want to print and one just past the last element. The program is safe, as long as we correctly calculate the pointers so that they denote a range of elements.

调用这个版本的函数需要传递两个指针：一个指向要输出的第一个元素，另一个则指向最后一个元素的下一个位置。只要正确计算指针，使它们标记一段有效的元素范围，程序就会安全。

Explicitly Passing a Size Parameter

显式传递表示数组大小的形参

A third approach, which is common in C programs and pre-Standard C++ programs, is to define a second parameter that indicates the size of the array.

第三种方法是将第二个形参定义为表示数组的大小，这种用法在 C 程序和标准化之前的 C++ 程序中十分普遍。

Using this approach, we could rewrite `printValues` one more time. The new version and a call to it looks like:

用这种方法再次重写函数 `printValues`，新版本及其调用如下所示：

```
// const int ia[] is equivalent to const int* ia
// size is passed explicitly and used to control access to elements of ia
void printValues(const int ia[], size_t size)
{
    for (size_t i = 0; i != size; ++i) {
        cout << ia[i] << endl;
    }
}
int main()
{
    int j[] = { 0, 1 }; // int array of size 2
    printValues(j, sizeof(j)/sizeof(*j));
    return 0;
}
```

This version uses the `size` parameter to determine how many elements there are to print. When we call `printValues`, we must pass an additional parameter. The program executes safely as long as the size passed is no greater than the actual size of the array.

这个版本使用了形参 `size` 来确定要输出的元素的个数。调用 `printValues` 时，要额外传递一个形参。只要传递给函数的 `size` 值不超过数组的实际大小，程序就能安全运行。

Exercises Section 7.2.5

Exercise 7.12: When would you use a parameter that is a pointer? When would you use a parameter that is a reference? Explain the advantages and disadvantages of each.

什么时候应使用指针形参？什么时候就使用引用形参？解释两者的优点和缺点。

Exercise 7.13: Write a program to calculate the sum of the elements in an array. Write the function three times, each one using a different approach to managing the array bounds.

编写程序计算数组元素之和。要求编写函数三次，每次以不同的方法处理数组边界。

Section 7.2. Argument Passing

Exercise Write a program to sum the elements in a `vector<double>`.
7.14: 编写程序求 `vector<double>` 对象中所有元素之和。

7.2.6. `main`: Handling Command-Line Options

7.2.6. `main`: 处理命令行选项

It turns out that `main` is a good example of how C programs pass arrays to functions. Up to now, we have defined `main` with an empty parameter list:

主函数 `main` 是演示 C 程序如何将数组传递给函数的好例子。直到现在，我们所定义的主函数都只有空的形参表：

```
int main() { ... }
```

However, we often need to pass arguments to `main`. Traditionally, such arguments are options that determine the operation of the program. For example, assuming our `main` program was in an executable file named `prog`, we might pass options to the program as follows:

但是，我们通常需要给 `main` 传递实参。传统上，主函数的实参是可选的，用来确定程序要执行的操作。比如，假设我们的主函数 `main` 位于名为 `prog` 的可执行文件中，可如下将实参选项传递给程序：

```
prog -d -o ofile data0
```

The way this usage is handled is that `main` actually defines two parameters:

这种用法的处理方法实际上是在主函数 `main` 中定义了两个形参：

```
int main(int argc, char *argv[]) { ... }
```

The second parameter, `argv`, is an array of C-style character strings. The first parameter, `argc`, passes the number of strings in that array. Because the second parameter is an array, we might alternatively define `main` as

第二个形参 `argv` 是一个 C 风格字符串数组。第一个形参 `argc` 则用于传递该数组中字符串的个数。由于第二个参数是一个数组，主函数 `main` 也可以这样定义：

```
int main(int argc, char **argv) { ... }
```

indicating that `argv` points to a `char*`.

表示 `argv` 是指向 `char*` 的指针。

When arguments are passed to `main`, the first string in `argv`, if any, is always the name of the program. Subsequent elements pass additional optional strings to `main`. Given the previous command line, `argc` would be set to 5, and `argv` would hold the following C-style character strings:

当将实参传递给主函数 `main` 时，`argv` 中的第一个字符串（如果有的话）通常是程序的名字。接下来的元素将额外的可选字符串传递给主函数 `main`。以前面的命令行为例，`argc` 应设为 5，`argv` 会保存下面几个 C 风格字符串：

```
argv[0] = "prog";
argv[1] = "-d";
argv[2] = "-o";
argv[3] = "ofile";
argv[4] = "data0";
```

Exercises Section 7.2.6

Exercise Write a `main` function that takes two values as arguments and print their sum.
7.15: 编写一个主函数 `main`，使用两个值作为实参，并输出它们的和。

Exercise Write a program that could accept the options presented in this section. Print the values of the arguments passed to `main`.
7.16:

编写程序使之可以接受本节介绍的命令行选项，并输出传递给 `main` 的实参值。

7.2.7. Functions with Varying Parameters

7.2.7. 含有可变形参的函数



Ellipsis parameters are in C++ in order to compile C programs that use `varargs`. See your C compiler documentation for how to use `varargs`. Only simple data types from the C++ program should be passed to functions with ellipses parameters. In particular, objects of most class types are not copied properly when passed to ellipses parameters.

C++ 中的省略符形参是为了编译使用了 `varargs` 的 C 语言程序。关于如何使用 `varargs`, 请查阅所用 C 语言编译器的文档。对于 C++ 程序, 只能将简单数据类型传递给含有省略符形参的函数。实际上, 当需要传递给省略符形参时, 大多数类类型对象都不能正确地复制。

Ellipses parameters are used when it is impossible to list the type and number of all the arguments that might be passed to a function. Ellipses suspend type checking. Their presence tells the compiler that when the function is called, zero or more arguments may follow and that the types of the arguments are unknown. Ellipses may take either of two forms:

在无法列举出传递给函数的所有实参的类型和数目时, 可以使用省略符形参。省略符暂停了类型检查机制。它们的出现告知编译器, 当调用函数时, 可以有 0 或多个实参, 而实参的类型未知。省略符形参有下列两种形式:

```
void foo(parm_list, ...);
void foo(...);
```

The first form provides declarations for a certain number of parameters. In this case, type checking is performed when the function is called for the arguments that correspond to the parameters that are explicitly declared, whereas type checking is suspended for the arguments that correspond to the ellipsis. In this first form, the comma following the parameter declarations is optional.

第一种形式为特定数目的形参提供了声明。在这种情况下, 当函数被调用时, 对于与显示声明的形参相对应的实参进行类型检查, 而对于与省略符对应的实参则暂停类型检查。在第一种形式中, 形参声明后面的逗号是可选的。

Most functions with an ellipsis use some information from a parameter that is explicitly declared to obtain the type and number of optional arguments provided in a function call. The first form of function declaration with ellipsis is therefore most commonly used.

大部分带有省略符形参的函数都利用显式声明的参数中的一些信息, 来获取函数调用中提供的其他可选实参的类型和数目。因此带有省略符的第一种形式的函数声明是最常用的。

7.3. The `return` Statement

7.3.1. `return` 语句

A `return` statement terminates the function that is currently executing and returns control to the function that called the now-terminated function. There are two forms of `return` statements:

`return` 语句用于结束当前正在执行的函数，并将控制权返回给调用此函数的函数。`return` 语句有两种形式：

```
return;
return expression;
```

7.3.1.1. Functions with No Return Value

7.3.1.1. 没有返回值的函数

A `return` with no value may be used only in a function that has a return type of `void`. Functions that return `void` are not required to contain a `return` statement. In a `void` function, an implicit `return` takes place after the function's final statement.

不带返回值的 `return` 语句只能用于返回类型为 `void` 的函数。在返回类型为 `void` 的函数中，`return` 返回语句不是必需的，隐式的 `return` 发生在函数的最后一个语句完成时。

Typically, a `void` function uses a `return` to cause premature termination of the function. This use of `return` parallels the use of the `break` ([Section 6.10](#), p. 212) statement inside a loop. For example, we could rewrite our `swap` program to avoid doing any work if the values are identical:

一般情况下，返回类型是 `void` 的函数使用 `return` 语句是为了引起函数的强制结束，这种 `return` 的用法类似于循环结构中的 `break` 语句（[第 6.10 节](#)）的作用。例如，可如下重写 `swap` 程序，使之在输入的两个数值相同时不执行任何工作：

```
// ok: swap acts on references to its arguments
void swap(int &v1, int &v2)
{
    // if values already the same, no need to swap, just return
    if (v1 == v2)
        return;
    // ok, have work to do
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
    // no explicit return necessary
}
```

This function first checks if the values are equal and if so exits the function. If the values are unequal, the function swaps them. An implicit return occurs after the last assignment statement.

这个函数首先检查两个值是否相等，如果相等则退出函数；如果不相等，则交换这两个值，隐式的 `return` 发生在最后一个赋值语句后。

A function with a `void` return type ordinarily may not use the second form of the `return` statement. However, a `void` function may return the result of calling another function that returns `void`:

返回类型为 `void` 的函数通常不能使用第二种形式的 `return` 语句，但是，它可以返回另一个返回类型同样是 `void` 的函数的调用结果：

```
void do_swap(int &v1, int &v2)
{
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
    // ok: void function doesn't need an explicit return
}
void swap(int &v1, int &v2)
{
    if (v1 == v2)
        return false; // error: void function cannot return a value
    return do_swap(v1, v2); // ok: returns call to a void function
}
```

Section 7.3. The return Statement

Attempting to return any other expression is a compile-time error.

返回任何其他表达式的尝试都会导致编译时的错误。

7.3.2. Functions that Return a Value

7.3.2. 具有返回值的函数

The second form of the `return` statement provides the function's result. Every return in a function with a return type other than `void` must return a value. The value returned must have the same type as the function return type, or must have a type that can be implicitly converted to that type.

`return` 语句的第二种形式提供了函数的结果。任何返回类型不是 `void` 的函数必须返回一个值，而且这个返回值的类型必须和函数的返回类型相同，或者能隐式转化为函数的返回类型。

Although C++ cannot guarantee the correctness of a result, it can guarantee that every `return` from a function returns a result of the appropriate type. The following program, for example, won't compile:

尽管 C++ 不能确保结果的正确性，但能保证函数每一次 `return` 都返回适当类型的结果。例如，下面的程序就不能通过编译：

```
// Determine whether two strings are equal.  
// If they differ in size, determine whether the smaller  
// one holds the same characters as the larger one  
bool str_subrange(const string &str1, const string &str2)  
{  
    // same sizes: return normal equality test  
    if (str1.size() == str2.size())  
        return str1 == str2;    // ok, == returns bool  
    // find size of smaller string  
    string::size_type size = (str1.size() < str2.size())  
                           ? str1.size() : str2.size();  
    string::size_type i = 0;  
    // look at each element up to size of smaller string  
    while (i != size) {  
        if (str1[i] != str2[i])  
            return;    // error: no return value  
    }  
    // error: control might flow off the end of the function without a return  
    // the compiler is unlikely to detect this error  
}
```

The `return` from within the `while` loop is an error because it fails to return a value. The compiler should detect this error.

`while` 循环中的 `return` 语句是错误的，因为它没有返回任何值，编译器将检查出这个错误。

The second error occurs because the function fails to provide a `return` after the `while` loop. If we call this function with one `string` that is a subset of the other, execution would fall out of the `while`. There should be a turn to handle this case. The compiler may or may not detect this error. If a program is generated, what happens at run time is undefined.

第二个错误源于函数没有在 `while` 循环后提供 `return` 语句。调用这个函数时，如果一个 `string` 是另一个 `string` 的子集，执行会退出 `while` 循环。这里应该有一个 `return` 语句来处理这种情况。编译器有可能检查出也有可能检查不出这种错误。执行程序时，不确定在运行阶段会出现什么问题。



Failing to provide a `return` after a loop that does contain a `return` is particularly insidious because many compilers will not detect it. The behavior at run time is undefined.

在含有 `return` 语句的循环后没有提供 `return` 语句是很危险的，因为大部分的编译器不能检测出这个漏洞，运行时会出现什么问题是不确定的。

Return from `main`

主函数 `main` 的返回值

There is one exception to the rule that a function with a return type other than `void` must return a value: The `main` function is allowed to terminate

Section 7.3. The return Statement

without a return. If control reaches the end of `main` and there is no return, then the compiler implicitly inserts a return of 0.

返回类型不是 `void` 的函数必须返回一个值，但此规则有一个例外情况：允许主函数 `main` 没有返回值就可结束。如果程序控制执行到主函数 `main` 的最后一个语句都还没有返回，那么编译器会隐式地插入返回 0 的语句。

Another way in which the return from `main` is special is how its returned value is treated. As we saw in [Section 1.1 \(p. 2\)](#), the value returned from `main` is treated as a status indicator. A zero return indicates success; most other values indicate failure. A nonzero value has a machine-dependent meaning. To make return values machine-independent, the `cstdlib` header defines two preprocessor variables ([Section 2.9.2, p. 69](#)) that we can use to indicate success or failure:

```
#include <cstdlib>
int main()
{
    if (some_failure)
        return EXIT_FAILURE;
    else
        return EXIT_SUCCESS;
}
```

Our code no longer needs to use the precise machine-dependent values. Instead, those values are defined in `cstdlib`, and our code need not change.

我们的代码不再需要使用那些依赖于机器的精确返回值。相应地，这些值都在 `cstdlib` 库中定义，我们的代码不需要做任何修改。

Returning a Nonreference Type

返回非引用类型

The value returned by a function is used to initialize a [temporary object](#) created at the point at which the call was made. A temporary object is an unnamed object created by the compiler when it needs a place to store a result from evaluating an expression. C++ programmers usually use the term "temporary" as an abbreviation of "temporary object."

函数的返回值用于初始化在调用函数处创建的[临时对象](#)。在求解表达式时，如果需要一个地方储存其运算结果，编译器会创建一个没有命名的对象，这就是临时对象。在英语中，C++ 程序员通常用 `temporary` 这个术语来代替 `temporary object`。

The temporary is initialized by the value returned by a function in much the same way that parameters are initialized by their arguments. If the return type is not a reference, then the return value is copied into the temporary at the call site. The value returned when a function returns a nonreference type can be a local object or the result of evaluating an expression.

用函数返回值初始化临时对象与用实参初始化形参的方法是一样的。如果返回类型不是引用，在调用函数的地方会将函数返回值复制给临时对象。当函数返回非引用类型时，其返回值既可以是局部对象，也可以是求解表达式的结果。

As an example, we might want to write a function that, given a counter, a word, and an ending, gives us back the plural version of the word if the counter is greater than one:

例如，下面的程序提供了一个计数器、一个单词 `word` 和单词结束字符串 `ending`，当计数器的值大于 1 时，返回该单词的复数版本：

```
// return plural version of word if ctr isn't 1
string make_plural(size_t ctr, const string &word,
                    const string &ending)
{
    return (ctr == 1) ? word : word + ending;
}
```

We might use such a function to print a message with either a plural or singular ending.

我们可以使用这样的函数来输出单词的单数或复数形式。

This function either returns a copy of its parameter named `word` or it returns an unnamed temporary `string` that results from adding `word` and `ending`. In either case, the `return` copies that `string` to the call site.

这个函数要么返回其形参 `word` 的副本，要么返回一个未命名的临时 `string` 对象，这个临时对象是由字符串 `word` 和 `ending` 的相加而产生的。这两种情况下，`return` 都在调用该函数的地方复制了返回的 `string` 对象。

Returning a Reference

Section 7.3. The return Statement

返回引用

When a function returns a reference type, the return value is not copied. Instead, the object itself is returned. As an example, consider a function that returns a reference to the shorter of its two `string` parameters:

当函数返回引用类型时，没有复制返回值。相反，返回的是对象本身。例如，考虑下面的函数，此函数返回两个 `string` 类型参数中较短的那个字符串的引用：

```
// find longer of two strings
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() < s2.size() ? s1 : s2;
}
```

The parameters and return type are references to `const string`. The `strings` are not copied either when calling the function or when returning the result.

形参和返回类型都是指向 `const string` 对象的引用，调用函数和返回结果时，都没有复制这些 `string` 对象。

Never Return a Reference to a Local Object

千万不要返回局部对象的引用



There's one crucially important thing to understand about returning a reference: Never return a reference to a local variable.

理解返回引用至关重要的是：千万不能返回局部变量的引用。

When a function completes, the storage in which the local objects were allocated is freed. A reference to a local object refers to undefined memory after the function terminates. Consider the following function:

当函数执行完毕时，将释放分配给局部对象的存储空间。此时，对局部对象的引用就会指向不确定的内存。考虑下面的程序：

```
// Disaster: Function returns a reference to a local object
const string &manip(const string& s)
{
    string ret = s;
    // transform ret in some way
    return ret; // Wrong: Returning reference to a local object!
}
```

This function will fail at run time because it returns a reference to a local object. When the function ends, the storage in which `ret` resides is freed. The return value refers to memory that is no longer available to the program.

这个函数会在运行时出错，因为它返回了局部对象的引用。当函数执行完毕，字符串 `ret` 占用的储存空间被释放，函数返回值指向了对于这个程序来说不再有效的内存空间。



One good way to ensure that the return is safe is to ask: To what *pre-existing* object is the reference referring?

确保返回引用安全的一个好方法是：请自问，这个引用指向哪个在此之前存在的对象？

Reference Returns Are Lvalues

引用返回左值

Section 7.3. The return Statement

A function that returns a reference returns an lvalue. That function, therefore, can be used wherever an lvalue is required:

返回引用的函数返回一个左值。因此，这样的函数可用于任何要求使用左值的地方：

```
char &get_val(string &str, string::size_type ix)
{
    return str[ix];
}
int main()
{
    string s("a value");
    cout << s << endl; // prints a value
    get_val(s, 0) = 'A'; // changes s[0] to A

    cout << s << endl; // prints A value
    return 0;
}
```

It may be surprising to assign to the return of a function, but the return is a reference. As such, it is just a synonym for the element returned.

给函数返回值赋值可能让人惊讶，由于函数返回的是一个引用，因此这是正确的，该引用是被返回元素的同义词。

If we do not want the reference return to be modifiable, the return value should be declared as `const`:

如果不希望引用返回值被修改，返回值应该声明为 `const`：

```
const char &get_val(...
```

Never Return a Pointer to a Local Object

千万不要返回指向局部对象的指针

The return type for a function can be most any type. In particular, it is possible for a function to return a pointer. For the same reasons that it is an error to return a reference to a local object, it is also an error to return a pointer to a local object. Once the function completes, the local objects are freed. The pointer would be a dangling pointer ([Section 5.11, p. 176](#)) that refers to a nonexistent object.

函数的返回类型可以是大多数类型。特别地，函数也可以返回指针类型。和返回局部对象的引用一样，返回指向局部对象的指针也是错误的。一旦函数结束，局部对象被释放，返回的指针就变成了指向不再存在的对象的悬垂指针 ([第 5.11 节](#))。

Exercises Section 7.3.2

Exercise

7.17: When is it valid to return a reference? A `const` reference?

什么时候返回引用是正确的？而什么时候返回 `const` 引用是正确的？

Exercise

7.18: What potential run-time problem does the following function have?

下面函数存在什么潜在的运行时问题？

```
string &processText() {
    string text;
    while (cin >> text) { /* ... */ }
    // ...
    return text;
}
```

Exercise

7.19: Indicate whether the following program is legal. If so, explain what it does; if not, make it legal and then explain it:

判断下面程序是否合法；如果合法，解释其功能；如果不合法，更正它并解释原因。

```
int &get(int *arry, int index) { return arry[index]; }
int main() {
    int ia[10];
    for (int i = 0; i != 10; ++i)
        get(ia, i) = 0;
}
```

7.3.3. Recursion

7.3.3. 递归

A function that calls itself, either directly or indirectly, is a ***recursive function***. An example of a simple recursive function is one that computes the factorial of a number `n` is the product of the numbers from 1 to `n`. The factorial of 5, for example, is 120.

直接或间接调用自己的函数称为**递归函数**。一个简单的递归函数例子是阶乘的计算。数 `n` 阶乘是从 1 到 `n` 的乘积。例如，5 的阶乘就是 120。

```
1 * 2 * 3 * 4 * 5 = 120
```

A natural way to solve this problem is recursively:

解决这个问题的自然方法就是递归：

```
// calculate val!, which is 1*2 *3 ... * val
int factorial(int val)
{
    if (val > 1)
        return factorial(val-1) * val;
    return 1;
}
```

A recursive function must always define a stopping condition; otherwise, the function will recurse "forever," meaning that the function will continue to call itself until the program stack is exhausted. This is sometimes called an "infinite recursion error." In the case of `factorial`, the stopping condition occurs when `val` is 1.

递归函数必须定义一个终止条件；否则，函数就会“永远”递归下去，这意味着函数会一直调用自身直到程序栈耗尽。有时候，这种现象称为“无限递归错误”。对于函数 `factorial`, `val` 为 1 是终止条件。

As another example, we can define a recursive function to find the greatest common divisor:

另一个例子是求最大公约数的递归函数：

```
// recursive version greatest common divisor program
int rgcd(int v1, int v2)
{
    if (v2 != 0)           // we're done once v2 gets to zero
        return rgcd(v2, v1%v2); // recurse, reducing v2 on each call
    return v1;
}
```

In this case the stopping condition is a remainder of 0. If we call `rgcd` with the arguments `(15, 123)`, then the result is three. [Table 7.1](#) on the next page traces the execution.

这个例子中，终止条件是余数为 0。如果用实参 `(15, 123)` 来调用 `rgcd` 函数，结果为 3。[表 7.1](#) 跟踪了它的执行过程。

Table 7.1. Trace of rgcd(15,123)

表 7.1. `rgcd(15, 123)` 的跟踪过程

v1	v2	Return
15	123	<code>rgcd(123, 15)</code>
123	15	<code>rgcd(15, 3)</code>
15	3	<code>rgcd(3, 0)</code>
3	0	3

The last call,

最后一次调用:

```
rgcd(3,0)
```

satisfies the stopping condition. It returns the greatest common denominator, 3. This value successively becomes the return value of each prior call. The value is said to percolate upward until the execution returns to the function that called `rgcd` in the first place.

满足了终止条件，它返回最大公约数 3。该值依次成为前面每个调用的返回值。这个过程称为此值向上回渗（percolate），直到执行返回到第一次调用 `rgcd` 的函数。



The `main` function may *not* call itself.

主函数 `main` 不能调用自身。

Exercises Section 7.3.3

Exercise Rewrite `factorial` as an iterative function.

7.20: 将函数 `factorial` 重写为迭代函数（即非递归函数）。

Exercise What would happen if the stopping condition in `factorial` were:

7.21: 如是函数 `factorial` 的终止条件为:

```
if (val != 0)
```

会出现什么问题?

7.4. Function Declarations

7.4. 函数声明

Just as variables must be declared before they are used, a function must be declared before it is called. As with a variable definition ([Section 2.3.5](#), p. 52), we can declare a function separately from its definition; a function may be defined only once but may be declared multiple times.

正如变量必须先声明后使用一样，函数也必须在被调用之前先声明。与变量的定义（[第 2.3.5 节](#)）类似，函数的声明也可以和函数的定义分离；一个函数只能定义一次，但是可声明多次。

A function declaration consists of a return type, the function name, and parameter list. The parameter list must contain the types of the parameters but need not name them. These three elements are referred to as the **function prototype**. A function prototype describes the interface of the function.

函数声明由函数返回类型、函数名和形参列表组成。形参列表必须包括形参类型，但是不必对形参命名。这三个元素被称为**函数原型**，函数原型描述了函数的接口。



Function prototypes provide the interface between the programmer who defines the function and programmers who use it. When we use a function, we program to the function's prototype.

函数原型为定义函数的程序员和使用函数的程序员之间提供了接口。在使用函数时，程序员只对函数原型编程即可。

Parameter names in a function declaration are ignored. If a name is given in a declaration, it should serve as a documentation aid:

函数声明中的形参名会被忽略，如果在声明中给出了形参的名字，它应该用作辅助文档：

```
void print(int *array, int size);
```

Function Declarations Go in Header Files

在头文件中提供函数声明

Recall that variables are declared in header files ([Section 2.9](#), p. 67) and defined in source files. For the same reasons, functions should be declared in header files and defined in source files.

回顾前面章节，变量可在头文件中声明（[第 2.9 节](#)），而在源文件中定义。同理，函数也应当在头文件中声明，并在源文件中定义。

It may be tempting and would be legal to put a function declaration directly in each source file that uses the function. The problem with this approach is that it is tedious and error-prone. By putting function declarations into header files, we can ensure that all the declarations for a given function agree. If the interface to the function changes, only one declaration must be changed.

把函数声明直接放到每个使用该函数的源文件中，这可能是大家希望的方式，而且也是合法的。但问题在于这种用法比较呆板而且容易出错。解决的方法是把函数声明放在头文件中，这样可以确保对于指定函数其所有声明保持一致。如果函数接口发生变化，则只要修改其唯一的声明即可。



The source file that *defines* the function should include the header that *declares* the function.

定义函数的源文件应包含声明该函数的头文件。

Including the header that contains a function's declaration in the same file that defines the function lets the compiler check that the definition and declaration are the same. In particular, if the definition and declaration agree as to parameter list but differ as to return type, the compiler will issue a warning or error message indicating the discrepancy.

Section 7.4. Function Declarations

将提供函数声明头文件包含在定义该函数的源文件中，可使编译器能检查该函数的定义和声明时是否一致。特别地，如果函数定义和函数声明的形参列表一致，但返回类型不一致，编译器会发出警告或出错信息来指出这种差异。

Exercises Section 7.4

Exercise Write the prototypes for each of the following functions:

7.22:

编写下面函数的原型：

- a. A function named `compare` with two parameters that are references to a class named `matrix` and with a return value of type `bool`.

函数名为 `compare`，有两个形参，都是名为 `matrix` 的类的引用，返回 `bool` 类型的值。

- b. A function named `change_val` that returns a `vector<int>` iterator and takes two parameters: one is an `int` and the other is an iterator for a `vector<int>`.

函数名为 `change_val`，返回 `vector<int>` 类型的迭代器，有两个形参：一个是 `int` 型形参，另一个是 `vector<int>` 类型的迭代器。

Hint: When you write these prototypes, use the name of the function as an indicator as to what the function does. How does this hint affect the types you use?

提示：写函数原型时，函数名应当暗示函数的功能。考虑这个提示会如何影响你用的类型？

Exercise Given the following declarations, determine which calls are legal and which are illegal. For those that are illegal, explain why.

7.23: 给出下面函数，判断哪些调用是合法的，哪些是不合法的。对于那些不合法的调用，解释原因。

```
double calc(double);
int count(const string &, char);
int sum(vector<int>::iterator, vector<int>::iterator, int);
vector<int> vec(10);

(a) calc(23.4, 55.1);
(b) count("abcd", 'a');
(c) calc(66);
(d) sum(vec.begin(), vec.end(), 3.8);
```

7.4.1. Default Arguments

7.4.1. 默认实参

A default argument is a value that, although not universally applicable, is the argument value that is expected to be used most of the time. When we call the function, we may omit any argument that has a default. The compiler will supply the default value for any argument we omit.

默认实参是一种虽然并不普遍、但在多数情况下仍然适用的实参值。调用函数时，可以省略有默认值的实参。编译器会为我们省略的实参提供默认值。

A default argument is specified by providing an explicit initializer for the parameter in the parameter list. We may define defaults for one or more parameters. However, if a parameter has a default argument, all the parameters that follow it must also have default arguments.

默认实参是通过给形参表中的形参提供明确的初始值来指定的。程序员可为一个或多个形参定义默认值。但是，如果有一个形参具有默认实参，那么，它后面所有的形参都必须有默认实参。

For example, a function to create and initialize a `string` intended to simulate a window can provide default arguments for the height, width, and background character of the screen:

例如，下面的函数创建并初始化了一个 `string` 对象，用于模拟窗口屏幕。此函数为窗口屏幕的高、宽和背景字符提供了默认实参：

```
string screenInit(string::size_type height = 24,
                  string::size_type width = 80,
                  char background = ' ' );
```

A function that provides a default argument for a parameter can be invoked with or without an argument for this parameter. If an argument is

Section 7.4. Function Declarations

provided, it overrides the default argument value; otherwise, the default argument is used. Each of the following calls of `screenInit` is correct:

调用包含默认实参的函数时, 可以为该形参提供实参, 也可以不提供。如果提供了实参, 则它将覆盖默认的实参值; 否则, 函数将使用默认实参值。下面的函数 `screenInit` 的调用都是正确的:

```
string screen;
screen = screenInit();           // equivalent to screenInit (24,80,' ')
screen = screenInit(66);         // equivalent to screenInit (66,80,' ')
screen = screenInit(66, 256);    // screenInit(66,256,' ')
screen = screenInit(66, 256, '#');
```

Arguments to the call are resolved by position, and default arguments are used to substitute for the *trailing* arguments of a call. If we want to specify an argument for `background`, we must also supply arguments for `height` and `width`:

```
screen = screenInit(, , '?'); // error, can omit only trailing arguments
screen = screenInit( '?');   // calls screenInit('?',80,'')
```

Note that the second call, which passes a single character value, is legal. Although legal, it is unlikely to be what the programmer intended. The call is legal because `'?'` is a `char`, and a `char` can be promoted to the type of the leftmost parameter. That parameter is `string::size_type`, which is an `unsigned` integral type. In this call, the `char` argument is implicitly promoted to `string::size_type`, and passed as the argument to `height`.

注意第二个调用, 只传递了一个字符值, 虽然这是合法的, 但是却并不是程序员的原意。因为 `'?'` 是一个 `char`, `char` 可提升为最左边形参的类型, 所以这个调用是合法的。最左边的形参具有 `string::size_type` 类型, 这是 `unsigned` 整型。在这个调用中, `char` 实参隐式地提升为 `string::size_type` 类型, 并作为实参传递给形参 `height`。



Because `char` is an integral type ([Section 2.1.1](#), p. 34), it is legal to pass a `char` to an `int` parameter and vice versa. This fact can lead to various kinds of confusion, one of which arises in functions that take both `char` and `int` parameters. It can be easy for callers to pass the arguments in the wrong order. Using default arguments can compound this problem.

因为 `char` 是整型 ([第 2.1.1 节](#)), 因此把一个 `char` 值传递给 `int` 型形参是合法的, 反之亦然。这个事实会导致很多误解。例如, 如果函数同时含有 `char` 型和 `int` 型形参, 则调用者很容易以错误的顺序传递实参。如果使用默认实参, 则这个问题会变得更加复杂。

Part of the work of designing a function with default arguments is ordering the parameters so that those least likely to use a default value appear first and those most likely to use a default appear last.

设计带有默认实参的函数, 其中部分工作就是排列形参, 使最少使用默认实参的形参排在最前, 最可能使用默认实参的形参排在最后。

Default Argument Initializers

默认实参的初始化式

A default argument can be any expression of an appropriate type:

默认实参可以是任何适当类型的表达式:

```
string::size_type screenHeight();
string::size_type screenWidth(string::size_type);
char screenDefault(char = ' ');
string screenInit(
    string::size_type height = screenHeight(),
    string::size_type width = screenWidth(screenHeight()),
    char background = screenDefault());
```

When the default argument is an expression, and the default is used as the argument, then the expression is evaluated at the time the function is called. For example, `screenDefault` is called to obtain a value for `background` every time `screenInit` is called without a third argument.

如果默认实参是一个表达式, 而且默认值用作实参, 则在调用函数时求解该表达式。例如, 每次不带第三个实参调用函数 `screenInit` 时, 编译器都会调用函数 `screenDefault` 为 `background` 获得一个值。

Constraints on Specifying Default Arguments

指定默认实参的约束

We can specify default argument(s) in either the function definition or declaration. However, a parameter can have its default argument specified only once in a file. The following is an error:

既可以在函数声明也可以在函数定义中指定默认实参。但是，在一个文件中，只能为一个形参指定默认实参一次。下面的例子是错误的：

```
// ff.h
int ff(int = 0);

// ff.cc
#include "ff.h"
int ff(int i = 0) { /* ... */ } // error
```



Default arguments ordinarily should be specified with the declaration for the function and placed in an appropriate header.

通常，应在函数声明中指定默认实参，并将该声明放在合适的头文件中。

If a default argument is provided in the parameter list of a function definition, the default argument is available only for function calls in the source file that contains the function definition.

如果在函数定义的形参表中提供默认实参，那么只有在包含该函数定义的源文件中调用该函数时，默认实参才是有效的。

Exercises Section 7.4.1

Exercise

7.24: Which, if any, of the following declarations are errors? Why?

如果有的话，指出下面哪些函数声明是错误的？为什么？

- (a) int ff(int a, int b = 0, int c = 0);
- (b) char *init(int ht = 24, int wd, char bckgrnd);

Exercise

7.25: Given the following function declarations and calls, which, if any, of the calls are illegal? Why?
Which, if any, are legal but unlikely to match the programmer's intent? Why?

假设有如下函数声明和调用，指出哪些调用是不合法的？为什么？哪些是合法的但可能不符合程序员的原意？为什么？

```
// declarations
char *init(int ht, int wd = 80, char bckgrnd = ' ');

(a) init();
(b) init(24,10);
(c) init(14, '*');
```

Exercise

7.26: Write a version of `make_plural` with a default argument of '`s`'. Use that version to print singular and plural versions of the words "success" and "failure".

用字符 '`s`' 作为默认实参重写函数 `make_plural`。利用这个版本的函数输出单词“success”和“failure”的单数和复数形式。

7.5. Local Objects

7.5. 局部对象

In C++, names have scope, and objects have **lifetimes**. To understand how functions operate, it is important to understand both of these concepts. The scope of a name is the part of the program's text in which that name is known. The lifetime of an object is the time during the program's execution that the object exists.

在 C++ 语言中，每个名字都有作用域，而每个对象都有生命期。要弄清楚函数是怎么运行的，理解这两个概念十分重要。名字的作用域指的是知道该名字的程序文本区。对象的生命期则是在程序执行过程中对象存在的时间。

The names of parameters and variables defined within a function are in the scope of the function: The names are visible only within the function body. As usual, a variable's name can be used from the point at which it is declared or defined until the end of the enclosing scope.

在函数中定义的形参和变量的名字只位于函数的作用域中：这些名字只在函数体中可见。通常，变量名从声明或定义的地方开始到包围它的作用域结束处都是可用的。

7.5.1. Automatic Objects

7.5.1. 自动对象

By default, the lifetime of a local variable is limited to the duration of a single execution of the function. Objects that exist only while a function is executing are known as **automatic objects**. Automatic objects are created and destroyed on each call to a function.

默认情况下，局部变量的生命期局限于所在函数的每次执行期间。只有当定义它的函数被调用时才存在的对象称为**自动对象**。自动对象在每次调用函数时创建和撤销。

The automatic object corresponding to a local variable is created when the function control path passes through the variable's definition. If the definition contains an initializer, then the object is given an initial value each time the object is created. Uninitialized local variables of built-in type have undefined values. When the function terminates, the automatic objects are destroyed.

局部变量所对应的自动对象在函数控制经过变量定义语句时创建。如果在定义时提供了初始化式，那么每次创建对象时，对象都会被赋予指定的初值。对于未初始化的内置类型局部变量，其初值不确定。当函数调用结束时，自动对象就会撤销。

Parameters are automatic objects. The storage in which the parameters reside is created when the function is called and is freed when the function terminates.

形参也是自动对象。形参所占用的存储空间在调用函数时创建，而在函数结束时撤销。

Automatic objects, including parameters, are destroyed at the end of the block in which they were defined. Parameters are defined in the function's block and so are destroyed when the function terminates. When a function exits, its local storage is deallocated. After the function exits, the values of its automatic objects and parameters are no longer accessible.

自动对象，包括形参，都在定义它们的块语句结束时撤销。形参在函数块中定义，因此当函数的执行结束时撤销。当函数结束时，会释放它的局部存储空间。在函数结束后，自动对象和形参的值都不能再访问了。

7.5.2. Static Local Objects

7.5.2. 静态局部对象

It is can be useful to have a variable that is in the scope of a function but whose lifetime persists across calls to the function. Such objects are defined as **static**.

一个变量如果位于函数的作用域内，但生命期跨越了这个函数的多次调用，这种变量往往很有用。则应该将这样的对象定义为 **static**（静态的）。

A **static local object** is guaranteed to be initialized no later than the first time that program execution passes through the object's definition. Once it is created, it is not destroyed until the program terminates; local **statics** are not destroyed when the function ends. Local **statics** continue to exist and hold their value across calls to the function. As a trivial example, consider a function that counts how often it is called:

static 局部对象确保不迟于在程序执行流程第一次经过该对象的定义语句时进行初始化。这种对象一旦被创建，在程序结束前都不会撤销。当定义静态局部对象的函数结束时，静态局部对象不会撤销。在该函数被多次调用的过程中，静态局部对象会持续存在并保持它的值。考虑下面的小例子，这个函数计算了自己被调用的次数：

```
size_t count_calls()
{
```

Section 7.5.聽 Local Objects

```
static size_t ctr = 0; // value will persist across calls
return ++ctr;
}
int main()
{
    for (size_t i = 0; i != 10; ++i)
        cout << count_calls() << endl;
    return 0;
}
```

This program will print the numbers from 1 through 10 inclusive.

这个程序会依次输出 1 到 10 (包含 10) 的整数。

Before `count_calls` is called for the first time, `ctr` is created and given an initial value of `0`. Each call increments `ctr` and returns its current value. Whenever `count_calls` is executed, the variable `ctr` already exists and has whatever value was in the variable the last time the function exited. Thus, on the second invocation, the value is `1`, on the third it is `2`, and soon.

在第一次调用函数 `count_calls` 之前，`ctr` 就已创建并赋予初值 `0`。每次函数调用都使加 `1`，并且返回其当前值。在执行函数 `count_calls` 时，变量 `ctr` 就已经存在并且保留上次调用该函数时的值。因此，第二次调用时，`ctr` 的值为 `1`，第三次为 `2`，依此类推。

Exercises Section 7.5.2

Exercise 7.27: Explain the differences between a parameter, a local variable and a static local variable. Give an example of a program in which each might be useful.

解释形参、局部变量和静态局部变量的差别。并给出一个有效使用了这三种变量的程序例子。

Exercise 7.28: Write a function that returns 0 when it is first called and then generates numbers in sequence each time it is called again.

编写函数，使其在第一次调用时返回 `0`，然后再次调用时按顺序产生正整数（即返回其当前的调用次数）。

7.6. Inline Functions

7.6. 内联函数

Recall the function we wrote on page [248](#) that returned a reference to the shorter of its two `string` parameters:

回顾在[第 7.3.2 节](#)编写的那个返回两个 `string` 形参中较短的字符串的函数：

```
// find longer of two strings
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() < s2.size() ? s1 : s2;
}
```

The benefits of defining a function for such a small operation include:

为这样的小操作定义一个函数的好处是：

- It is easier to read and understand a call to `shorterString` than it would be to read and interpret an expression that used the equivalent conditional expression in place of the function call.

阅读和理解函数 `shorterString` 的调用，要比读一条用等价的条件表达式取代函数调用表达式并解释它的含义要容易得多。

- If a change needs to be made, it is easier to change the function than to find and change every occurrence of the equivalent expression.

如果需要做任何修改，修改函数要比找出并修改每一处等价表达式容易得多。

- Using a function ensures uniform behavior. Each test is guaranteed to be implemented in the same manner.

使用函数可以确保统一的行为，每个测试都保证以相同的方式实现。

- The function can be reused rather than rewritten for other applications.

函数可以重用，不必为其他应用重写代码。

There is, however, one potential drawback to making `shorterString` a function: Calling a function is slower than evaluating the equivalent expression. On most machines, a function call does a lot of work: registers are saved before the call and restored after the return; the arguments are copied; and the program branches to a new location.

但是，将 `shorterString` 写成函数有一个潜在的缺点：调用函数比求解等价表达式要慢得多。在大多数的机器上，调用函数都要做很多工作；调用前要先保存寄存器，并在返回时恢复；复制实参；程序还必须转向一个新位置执行。

`inline` Functions Avoid Function Call Overhead

`inline` 函数避免函数调用的开销

A function specified as `inline` (usually) is expanded "in line" at each point in the program in which it is invoked. Assuming we made `shorterString` an `inline` function, then this call

将函数指定为 `inline` 函数，（通常）就是将它在程序中每个调用点上“内联地”展开。假设我们将 `shorterString` 定义为内联函数，则调用：

```
cout << shorterString(s1, s2) << endl;
```

would be expanded during compilation into something like

在编译时将展开为：

```
cout << (s1.size() < s2.size() ? s1 : s2)
      << endl;
```

Section 7.6. Inline Functions

The run-time overhead of making `shorterString` a function is thus removed.

从而消除了把 `shorterString` 写成函数的额外执行开销。

We can define `shorterString` as an inline function by specifying the keyword `inline` before the function's return type:

从而消除了把 `shorterString` 写成函数的额外执行开销。

```
// inline version: find longer of two strings
inline const string &
shorterString(const string &s1, const string &s2)
{
    return s1.size() < s2.size() ? s1 : s2;
}
```



The `inline` specification is only a *request* to the compiler. The compiler may choose to ignore this request.

`inline` 说明对于编译器来说只是一个建议，编译器可以选择忽略这个。

In general, the `inline` mechanism is meant to optimize small, straight-line functions that are called frequently. Many compilers will not inline a recursive function. A 1,200-line function is also not likely to be expanded inline.

一般来说，内联机制适用于优化小的、只有几行的而且经常被调用的函数。大多数的编译器都不支持递归函数的内联。一个 1200 行的函数也不太可能在调用点内联展开。

Put `inline` Functions in Header Files

把 `inline` 函数放入头文件



Unlike other function definitions, inlines should be defined in header files.

内联函数应该在头文件中定义，这一点不同于其他函数。

To expand the code of an `inline` function at the point of call, the compiler must have access to the function definition. The function prototype is insufficient.

`inline` 函数的定义对编译器而言必须是可见的，以便编译器能够在调用点内联展开该函数的代码。此时，仅有函数原型是不够的。

An `inline` function may be defined more than once in a program as long as the definition appears only once in a given source file and the definition is exactly the same in each source file. By putting `inline` functions in headers, we ensure that the same definition is used whenever the function is called and that the compiler has the function definition available at the point of call.

`inline` 函数可能要在程序中定义不止一次，只要 `inline` 函数的定义在某个源文件中只出现一次，而且在所有源文件中，其定义必须是完全相同的。把 `inline` 函数的定义放在头文件中，可以确保在调用函数时所使用的定义是相同的，并且保证在调用点该函数的定义对编译器可见。



Whenever an `inline` function is added to or changed in a header file, every source file that uses that header must be recompiled.

在头文件中加入或修改 `inline` 函数时，使用了该头文件的所有源文件都必须重新编译。

Exercises Section 7.6

Exercise 7.29: Which one of the following declarations and definitions would you put in a header? In a program text file? Explain why.

对于下面的声明和定义，你会将哪个放在头文件，哪个放在程序文本文件呢？为什么？

- (a) `inline bool eq(const BigInt&, const BigInt&) {...}`
- (b) `void putValues(int *arr, int size);`

Exercise 7.30: Rewrite the `is Shorter` function from page 235 as an `inline` function.

第 7.2.2 节 的函数 `is Shorter` 改写为 `inline` 函数。

7.7. Class Member Functions

7.7. 类的成员函数

In [Section 2.8](#) (p. 63) we began the definition of the `Sales_item` class used in solving the bookstore problem from [Chapter 1](#). Now that we know how to define ordinary functions, we can continue to fill in our class by defining the member functions of this class.

[第 2.8 节](#)开始定义类 `Sales_item`, 用于解决[第一章](#)的书店问题。至此, 我们已经了解了如何定义普通函数, 现在来定义类的成员函数, 以继续完善这个类。

We define member functions similarly to how we define ordinary functions. As with any function, a member function consists of four parts:

成员函数的定义与普通函数的定义类似。和任何函数一样, 成员函数也包含下面四个部分:

- A return type for the function
函数返回类型。
- The function name
函数名。
- A (possibly empty) comma-separated list of parameters
用逗号隔开的形参表 (也可能是空的)。
- The function body, which is contained between a pair of curly braces
包含在一对花括号里面的函数体。

As we know, the first three of these parts constitute the function prototype. The function prototype defines all the type information related to the function: what its return type is, the function name, and what types of arguments may be passed to it. The function prototype *must* be defined within the class body. The body of the function, however, *may* be defined within the class itself or outside the class body.

正如我们知道的, 前面三部分组成函数原型。函数原型定义了所有和函数相关的类型信息: 函数返回类型是什么、函数的名字、应该给这个函数传递什么类型的实参。函数原型必须在类中定义。但是, 函数体则既可以在类中也可以在类外定义。

With this knowledge, let's look at our expanded class definition, to which we've added two new members: the member functions `avg_price` and `same_isbn`. The `avg_price` function has an empty parameter list and returns a value of type `double`. The `same_isbn` function returns a `bool` and takes a single parameter of type reference to `const Sales_item`.

知道这些后, 观察下面扩展的类定义, 我们为这个类增加了两个新成员: 成员函数 `avg_price` 和 `same_isbn`。其中 `avg_price` 函数的形参表是空的, 返回 `double` 类型的值。而 `same_isbn` 函数则返回 `bool` 对象, 有一个 `const Sales_item` 类型的引用形参。

```
class Sales_item {
public:
    // operations on Sales_item objects
    double avg_price() const;
    bool same_isbn(const Sales_item &rhs) const
        { return isbn == rhs.isbn; }
    // private members as before
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};
```

We'll explain the meaning of the `const` that follows the parameter lists shortly, but first we need to explain how member functions are defined.

在解释跟在形参表后面的 `const` 之前, 必须先说明成员函数是如何定义的。

7.7.1. Defining the Body of a Member Function

7.7.1. 定义成员函数的函数体

We must declare all the members of a class within the curly braces that delimit the class definition. There is no way to subsequently add any

Section 7.7. Class Member Functions

members to the class. Members that are functions must be defined as well as declared. We can define a member function either inside or outside of the class definition. In `Sales_item`, we have one example of each: `same_isbn` is defined inside the `Sales_item` class, whereas `avg_price` is declared inside the class but defined elsewhere.

类的所有成员都必须在类定义的花括号里面声明，此后，就不能再为类增加任何成员。类的成员函数必须如声明的一般定义。类的成员函数既可以在类的定义内也可以在类的定义外定义。在类 `Sales_item` 中，这两种情况各有一例说明：函数 `same_isbn` 在类内定义，而函数 `avg_price` 则在类外声明，在类外定义。

A member function that is defined inside the class is implicitly treated as an inline function ([Section 7.6, p. 256](#)).

编译器隐式地将在类内定义的成员函数当作内联函数 ([第 7.6 节](#))。

Let's look in more detail at the definition of `same_isbn`:

再详细观察函数 `same_isbn` 的定义：

```
bool same_isbn(const Sales_item &rhs) const
{ return isbn == rhs.isbn; }
```

As with any function, the body of this function is a block. In this case, the block contains a single statement that returns the result of comparing the value of the `isbn` data members of two `Sales_item` objects.

与任何函数一样，该函数的函数体也是一个块。在这个函数中，块中只有一个语句，比较两个 `Sales_item` 对象的数据成员 `isbn` 的值，并返回比较结果。

The first thing to note is that the `isbn` member is `private`. Even though these members are `private`, there is no error.

首先要注意的是：成员 `isbn` 是 `private` 的。尽管如此，上述语句却没有任何错误。



A member function may access the `private` members of its class.

类的成员函数可以访问该类的 `private` 成员。

More interesting is understanding from which `Sales_item` objects does the function get the values that it compares. The function refers both to `isbn` and `rhs.isbn`. Fairly clearly, `rhs.isbn` uses the `isbn` member from the argument passed to the function. The unqualified use of `isbn` is more interesting. As we shall see, the unqualified `isbn` refers to the `isbn` member of the object on behalf of which the function is called.

更有意思的是，函数从哪个 `Sales_item` 类对象得到这个用于比较的值？函数涉及到 `isbn` 和 `rhs.isbn`。很明显，`rhs.isbn` 使用的是传递给此函数的实参的 `isbn` 成员。没有前缀的 `isbn` 的用法更加有意思。正如我们所看见的，这个没有前缀的 `isbn` 指的是用于调用函数的对象的 `isbn` 成员。

Member Functions Have an Extra, Implicit Parameter

成员函数含有额外的、隐含的形参

When we call a member function, we do so on behalf of an object. For example, when we called `same_isbn` in the bookstore program on page 26, we executed the `same_isbn` member on the object named `total`:

调用成员函数时，实际上是使用对象来调用的。例如，调用 [第 1.6 节](#) 书店程序中的函数 `same_isbn`，是通过名为 `total` 的对象来执行 `same_isbn` 函数的：

```
if (total.same_isbn(trans))
```

In this call, we pass the object `trans`. As part of executing the call, the object `trans` is used to initialize the parameter `rhs`. Thus, in this call, `rhs.isbn` is a reference to `trans.isbn`.

在这个调用中，传递了对象 `trans`。作为执行调用的一部分，使用对象 `trans` 初始化形参 `rhs`。于是，`rhs.isbn` 是 `trans.isbn` 的引用。

The same argument-binding process is used to bind the unqualified use of `isbn` to the object named `total`. Each member function has an extra, implicit parameter that binds the function to the object on which the function was called. When we call `same_isbn` on the object named `total`, that object is also passed to the function. When `same_isbn` refers to `isbn`, it is implicitly referring to the `isbn` member of the object on which the function was called. The effect of this call is to compare `total.isbn` with `Trans.isbn`.

而没有前缀的 `isbn` 使用了相同的实参绑定过程，使之与名为 `total` 的对象绑定起来。每个成员函数都有一个额外的、隐含的形参将该成员函数与调用该函数的类对象捆绑在一起。当调用名为 `total` 的对象的 `same_isbn` 时，这个对象也传递给了函数。而 `same_isbn` 函数使用 `isbn` 时，就隐式地使用了调用该函数的对象的 `isbn` 成员。这个函数调用的效果是比较 `total.isbn` 和 `trans.isbn` 两个值。

Introducing `this`

`this` 指针的引入

Each member function (except for `static` member functions, which we cover in [Section 12.6](#) (p. 467)) has an extra, implicit parameter named `this`. When a member function is called, the `this` parameter is initialized with the address of the object on which the function was invoked. To understand a member function call, we might think that when we write

每个成员函数（除了在[第 12.6 节](#)介绍的 `static` 成员函数外）都有一个额外的、隐含的形参 `this`。在调用成员函数时，形参 `this` 初始化为调用函数的对象的地址。为了理解成员函数的调用，可考虑下面的语句：

```
total.same_isbn(trans);
```

it is as if the compiler rewrites the call as

就如编译器这样重写这个函数调用：

```
// pseudo-code illustration of how a call to a member function is translated
Sales_item::same_isbn(&total, trans);
```

In this call, the data member `isbn` inside `same_isbn` is bound to the one belonging to `total`.

在这个调用中，函数 `same_isbn` 中的数据成员 `isbn` 属于对象 `total`。

Introducing `const` Member Functions

`const` 成员函数的引入

We now can understand the role of the `const` that follows the parameter lists in the declarations of the `Sales_item` member functions: That `const` modifies the type of the implicit `this` parameter. When we call `total.same_isbn(trans)`, the implicit `this` parameter will be a `const Sales_Item*` that points to `total`. It is as if the body of `same_isbn` were written as

现在，可以理解跟在 `Sales_item` 成员函数声明的形参表后面的 `const` 所起的作用了：`const` 改变了隐含的 `this` 形参的类型。在调用 `total.same_isbn(trans)` 时，隐含的 `this` 形参将是一个指向 `total` 对象的 `const Sales_Item*` 类型的指针。就像如下编写 `same_isbn` 的函数体一样：

```
// pseudo-code illustration of how the implicit this pointer is used
// This code is illegal: We may not explicitly define the this pointer ourselves
// Note that this is a pointer to const because same_isbn is a const member
bool Sales_item::same_isbn(const Sales_item *const this,
                           const Sales_item &rhs) const
{ return (this->isbn == rhs.isbn); }
```

A function that uses `const` in this way is called a **const member function**. Because `this` is a pointer to `const`, a `const` member function cannot change the object on whose behalf the function is called. Thus, `avg_price` and `same_isbn` may read but not write to the data members of the objects on which they are called.

用这种方式使用 `const` 的函数称为**常量成员函数**。由于 `this` 是指向 `const` 对象的指针，`const` 成员函数不能修改调用该函数的对象。因此，函数 `avg_price` 和函数 `same_isbn` 只能读取而不能修改调用它们的对象的数据成员。



A `const` object or a pointer or reference to a `const` object may be used to call only `const` member functions. It is an error to try to call a non`const` member function on a `const` object or through a pointer or reference to a `const` object.

`const` 对象、指向 `const` 对象的指针或引用只能用于调用其 `const` 成员函数，如果尝试用它们来调用非 `const` 成员函数，则是错误的。

Using the `this` Pointer

`this`

Section 7.7. Class Member Functions

指针的使用

Inside a member function, we need not explicitly use the `this` pointer to access the members of the object on which the function was called. Any unqualified reference to a member of our class is assumed to be a reference through `this`:

在成员函数中，不必显式地使用 `this` 指针来访问被调用函数所属对象的成员。对这个类的成员的任何没有前缀的引用，都被假定为通过指针 `this` 实现的引用：

```
bool same_isbn(const Sales_item &rhs) const
{ return isbn == rhs.isbn; }
```

The uses of `isbn` in this function are as if we had written `this->units_sold` or `this->revenue`.

在这个函数中 `isbn` 的用法与 `this->units_sold` 或 `this->revenue` 的用法一样。

The `this` parameter is defined implicitly, so it is unnecessary and in fact illegal to include the `this` pointer in the function's parameter list. However, in the body of the function we can refer to the `this` pointer explicitly. It is legal, although unnecessary, to define `same_isbn` as follows:

由于 `this` 指针是隐式定义的，因此不需要在函数的形参表中包含 `this` 指针，实际上，这样做也是非法的。但是，在函数体中可以显式地使用 `this` 指针。如下定义函数 `same_isbn` 尽管没有必要，但是却是合法的：

```
bool same_isbn(const Sales_item &rhs) const
{ return this->isbn == rhs.isbn; }
```

7.7.2. Defining a Member Function Outside the Class

7.7.2. 在类外定义成员函数

Member functions defined outside the class definition must indicate that they are members of the class:

在类的定义外面定义成员函数必须指明它们是类的成员：

```
double Sales_item::avg_price() const
{
    if (units_sold)
        return revenue/units_sold;
    else
        return 0;
}
```

This definition is like the other functions we've seen: It has a return type of `double` and an empty parameter list enclosed in parentheses after the function name. What is new is the `const` following the parameter list and the form of the function name. The function name

`Sales_item::avg_price`

uses the scope operator ([Section 1.2.2, p. 8](#)) to say that we are defining the function named `avg_price` that is defined in the scope of the `Sales_item` class.

使用作用域操作符 ([第 1.2.2 节](#)) 指明函数 `avg_price` 是在类 `Sales_item` 的作用域范围内定义的。

The `const` that follows the parameter list reflects the way we declared the member function inside the `Sales_item` header. In any definition, the return type and parameter list must match the declaration, if any, of the function. In the case of a member function, the declaration is as it appears in the class definition. If the function is declared to be a `const` member function, then the `const` after the parameter list must be included in the definition as well.

形参表后面的 `const` 则反映了在类 `Sales_item` 中声明成员函数的形式。在任何函数定义中，返回类型和形参表必须和函数声明（如果有的话）一致。对于成员函数，函数声明必须与其定义一致。如果函数被声明为 `const` 成员函数，那么函数定义时形参表后面也必须有 `const`。

We can now fully understand the first line of this code: It says we are defining the `avg_price` function from the `Sales_item` class and that the function is a `const` member. The function takes no (explicit) parameters and returns a `double`.

现在可以完全理解第一行代码了：这行代码说明现在正在定义类 `Sales_item` 的函数 `avg_price`，而且这是一个 `const` 成员函数，这个函数没有（显式的）形参，返回 `double` 类型的值。

The body of the function is easier to understand: It tests whether `units_sold` is nonzero and, if so, `returns` the result of dividing `revenue` by `units_sold`. If `units_sold` is zero, we can't safely do the divisiondividing by zero has undefined behavior. In this program, we return 0, indicating that if there were no sales the average price would be zero. Depending on the sophistication of our error-handling strategy, we might instead throw an exception ([Section 6.13, p. 215](#)).

Section 7.7. Class Member Functions

函数体更加容易理解：检查 `units_sold` 是否为 0，如果不为 0，返回 `revenue` 除以 `units_sold` 的结果；如果 `units_sold` 是 0，不能安全地进行除法运算——除以 0 是未定义的行为。此时程序返回 0，表示没有任何销售时平均售价为 0。根据异常错误处理策略，也可以抛出异常来代替刚才的处理（[第 6.13 节](#)）。

7.7.3. Writing the `Sales_item` Constructor

7.7.3. 编写 `Sales_item` 类的构造函数

There's one more member that we need to write: a constructor. As we learned in [Section 2.8](#) (p. 65), class data members are not initialized when the class is defined. Instead, data members are initialized through a constructor.

还必须编写一个成员，那就是构造函数。正如在[第 2.8 节](#)所学习的，在定义类时没有初始化它的数据成员，而是通过构造函数来初始化其数据成员。

Constructors Are Special Member Functions

构造函数是特殊的成员函数

A **constructor** is a special member function that is distinguished from other member functions by having the same name as its class. Unlike other member functions, constructors have no return type. Like other member functions they take a (possibly empty) parameter list and have a function body. A class can have multiple constructors. Each constructor must differ from the others in the number or types of its parameters.

构造函数是特殊的成员函数，与其他成员函数不同，构造函数和类同名，而且没有返回类型。而与其他成员函数相同的是，构造函数也有形参表（可能为空）和函数体。一个类可以有多个构造函数，每个构造函数必须有与其他构造函数不同数目或类型的形参。

The constructor's parameters specify the initializers that may be used when creating objects of the class type. Ordinarily these initializers are used to initialize the data members of the newly created object. Constructors usually should ensure that every data member is initialized.

构造函数的形参指定了创建类类型对象时使用的初始化式。通常，这些初始化式会用于初始化新创建对象的数据成员。构造函数通常应确保其每个数据成员都完成了初始化。

The `Sales_item` class needs to explicitly define only one constructor, the **default constructor**, which is the one that takes no arguments. The default constructor says what happens when we define an object but do not supply an (explicit) initializer:

`Sales_item` 类只需要显式定义一个构造函数：没有形参的**默认构造函数**。默认构造函数说明当定义对象却没有为它提供（显式的）初始化式时应该怎么办：

```
vector<int> vi;           // default constructor: empty vector
string s;                 // default constructor: empty string
Sales_item item;          // default constructor: ???
```

We know the behavior of the `string` and `vector` default constructors: Each of these constructors initializes the object to a sensible default state. The default `string` constructor generates an empty string, the one that is equal to `""`. The default `vector` constructor generates a `vector` with no elements.

我们知道 `string` 和 `vector` 类默认构造函数的行为：这些构造函数会将对象初始化为合理的默认状态。`string` 的默认构造函数会产生空字符串上，相当于 `""`。`vector` 的默认构造函数则生成一个没有元素的 `vector` 向量对象。

Similarly, we'd like the default constructor for `Sales_items` to generate an empty `Sales_item`. Here "empty" means an object in which the `isbn` is the empty string and the `units_sold` and `revenue` members are initialized to zero.

同样地，我们希望类 `Sales_items` 的默认构造函数为它生成一个空的 `Sales_item` 对象。这里的“空”意味着对象中的 `isbn` 是空字符串，`units_sold` 和 `revenue` 则初始化为 0。

Defining a Constructor

构造函数的定义

Like any other member function, a constructor is declared inside the class and may be defined there or outside the class. Our constructor is simple, so we will define it inside the class body:

和其他成员函数一样，构造函数也必须在类中声明，但是可以在类中或类外定义。由于我们的构造函数很简单，因此在类中定义它：

```
class Sales_item {
public:
    // operations on Sales_item objects
    double avg_price() const;
    bool same_isbn(const Sales_item &rhs) const
```

Section 7.7. Class Member Functions

```
{ return isbn == rhs.isbn; }
// default constructor needed to initialize members of built-in type
Sales_item(): units_sold(0), revenue(0.0) { }
// private members as before
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};
```

Before we explain the constructor definition, note that we put the constructor in the `public` section of the class. Ordinarily, and certainly in this case, we want the constructor(s) to be part of the interface to the class. After all, we want code that uses the `Sales_item` type to be able to define and initialize `Sales_item` objects. Had we made the constructor `private`, it would not be possible to define `Sales_item` objects, which would make the class pretty useless.

在解释任何构造函数的定义之前，注意到构造函数是放在类的 `public` 部分的。通常构造函数会作为类的接口的一部分，这个例子也是这样。毕竟，我们希望使用类 `Sales_item` 的代码可以定义和初始化类 `Sales_item` 的对象。如果将构造函数定义为 `private` 的，则不能定义类 `Sales_item` 的对象，这样的话，这个类就没有什么用了。

As to the definition itself

对于定义本身：

```
// default constructor needed to initialize members of built-in type
Sales_item(): units_sold(0), revenue(0.0) { }
```

it says that we are defining a constructor for the `Sales_item` class that has an empty parameter list and an empty function body. The interesting part is the colon and the code between it and the curly braces that define the (empty) function body.

上述语句说明现在正在定义类 `Sales_item` 的构造函数，这个构造函数的形参表和函数体都为空。令人感兴趣的是冒号和冒号与定义（空）函数体的花括号之间的代码。

Constructor Initialization List

构造函数和初始化列表

The colon and the following text up to the open curly is the **constructor initializer list**. A constructor initializer list specifies initial values for one or more data members of the class. It follows the constructor parameter list and begins with a colon. The constructor initializer is a list of member names, each of which is followed by that member's initial value in parentheses. Multiple member initializations are separated by commas.

在冒号和花括号之间的代码称为**构造函数的初始化列表**。构造函数的初始化列表为类的一个或多个数据成员指定初值。它跟在构造函数的形参表之后，以冒号开关。构造函数的初始化式是一系列成员名，每个成员后面是括在圆括号中的初始值。多个成员的初始化用逗号分隔。

This initializer list says that both the `units_sold` and `revenue` members should be initialized to 0. Whenever a `Sales_item` object is created, these members will start out as 0. We need not specify an initial value for the `isbn` member. Unless we say otherwise in the constructor initializer list, members that are of class type are automatically initialized by that class' default constructor. Hence, `isbn` is initialized by the `string` default constructor, meaning that `isbn` initially is the empty string. Had we needed to, we could have specified a default value for `isbn` in the initializer list as well.

上述例题的初始化列表表明 `units_sold` 和 `revenue` 成员都应初始化为 0。每当创建 `Sales_item` 对象时，它的这两个成员都以初值 0 出现。而 `isbn` 成员可以不必准确指明其初值。除非在初始化列表中有其他表述，否则具有类类型的成员皆被其默认构造函数自动初始化。于是，`isbn` 由 `string` 类的默认构造函数初始化为空串。当然，如果有必要的话，也可以在初始化列表中指明 `isbn` 的默认初值。

Having explained the initializer list, we can now understand the constructor: Its parameter list and the function body are both empty. The parameter list is empty because we are defining the constructor that is run by default, when no initializer is present. The body is empty because there is no work to do other than initializing `units_sold` and `revenue`. The initializer list explicitly initializes `units_sold` and `revenue` to zero and implicitly initializes `isbn` to the empty `string`. Whenever we create a `Sales_item` object, the data members will start out with these values.

解释了初始化列表后，就可以深入地了解这个构造函数了：它的形参表和函数体都为空。形参表为空是因为正在定义的构造函数是默认调用的，无需提供任何初值。函数体为空是因为除了初始化 `units_sold` 和 `revenue` 成员外没有其他工作可做了。初始化列表显式地将 `units_sold` 和 `revenue` 初始化为 0，并隐式地将 `isbn` 初始化为空串。当创建新 `Sales_item` 对象时，数据成员将以这些值出现。

Synthesized Default Constructor

合成的默认构造函数

If we do not explicitly define any constructors, then the compiler will generate the default constructor for us.



如果没有为一个类显式定义任何构造函数，编译器将自动为这个类生成默认构造函数。

The compiler-created default constructor is known as a [synthesized default constructor](#). It initializes each member using the same rules as are applied for variable initializations ([Section 2.3.4](#), p. 50). Members that are of class type, such as `isbn`, are initialized by using the default constructor of the member's own class. The initial value of members of built-in type depend on how the object is defined. If the object is defined at global scope (outside any function) or is a local static object, then these members will be initialized to 0. If the object is defined at local scope, these members are uninitialized. As usual, using an uninitialized member for any purpose other than giving it a value is undefined.

由编译器创建的默认构造函数通常称为[默认构造函数](#)，它将依据如同变量初始化（[第 2.3.4 节](#)）的规则初始化类中所有成员。对于具有类类型的成员，如 `isbn`，则会调用该成员所属类自身的默认构造函数实现初始化。内置类型成员的初值依赖于对象如何定义。如果对象在全局作用域中定义（即不在任何函数中）或定义为静态局部对象，则这些成员将被初始化为 0。如果对象在局部作用域中定义，则这些成员没有初始化。除了给它们赋值之外，出于其他任何目的对未初始化成员的使用都没有定义。



The synthesized default constructor often suffices for classes that contain only members of class type. Classes with members of built-in or compound type should usually define their own default constructors to initialize those members.

合成的默认构造函数一般适用于仅包含类类型成员的类。而对于含有内置类型或复合类型成员的类，则通常应该定义他们自己的默认构造函数初始化这些成员。

Because the synthesized constructor does not automatically initialize members of built-in type, we had to define the `Sales_item` default constructor explicitly.

由于合成的默认构造函数不会自动初始化内置类型的成员，所以必须明确定义 `Sales_item` 类的默认构造函数。

7.7.4. Organizing Class Code Files

7.7.4. 类代码文件的组织

As we saw in [Section 2.9](#) (p. 67), class declarations ordinarily are placed in headers. Usually, member functions defined outside the class are put in ordinary source files. C++ programmers tend to use a simple naming convention for headers and the associated class definition code. The class definition is put in a file named `type.h` or `type.H`, where `type` is the name of the class defined in the file. Member function definitions usually are stored in a source file whose name is the name of the class. Following this convention we put the `Sales_item` class definition in a file named `Sales_item.h`. Any program that wants to use the class must include that header. We should put the definition of our `Sales_item` functions in a file named `Sales_item.cc`. That file, like any other file that uses the `Sales_item` type, would include the `Sales_item.h` header.

正如在[第 2.9 节](#)提及的，通常将类的声明放置在头文件中。大多数情况下，在类外定义的成员函数则置于源文件中。C++ 程序员习惯使用一些简单的规则给头文件及其关联的类定义代码命名。类定义应置于名为 `type.h` 或 `type.H` 的文件中，`type` 指在该文件中定义的类的名字。成员函数的定义则一般存储在与类同名的源文件中。依照这些规则，我们将类 `Sales_item` 放在名为 `Sales_item.h` 的文件中定义。任何需使用这个类的程序，都必须包含这个头文件。而 `Sales_item` 的成员函数的定义则应该放在名为 `Sales_item.cc` 的文件中。这个文件同样也必须包含 `Sales_item.h` 头文件。

Exercises Section 7.7.4

- Exercise 7.31:** Write your own version of the `Sales_item` class, adding two new `public` members to read and write `Sales_item` objects. These functions should operate similarly to the input and output operators used in [Chapter 1](#). Transactions should look like the ones defined in that chapter as well. Use this class to read and write a set of transactions.

编写你自己的 `Sales_item` 类，添加两个公用（`public`）成员用于读和写 `Sales_item` 对象。这两个成员函数的功能应类似于[第一章](#)介绍的输入输出操作符。交易也应类似于那一章所定义的。利用这个类读入并输出一组交易。

- Exercise 7.32:** Write a header file to contain your version of the `Sales_item` class. Use ordinary C++ conventions to name the header and any associated file needed to hold non-`inline` functions

Section 7.7. Class Member Functions

defined outside the class.

编写一个头文件，包含你自己的 `Sales_item` 类。使用通用的 C++ 规则给这个头文件以及任何相关的文件命名，这些文件用于存储在类外定义的非内联函数。

Exercise 7.33: Add a member that adds two `Sales_items`. Use the revised class to reimplement your solution to the average price problem presented in [Chapter 1](#).

在 `Sales_item` 类中加入一个成员，用于添加两个 `Sales_item` 对象。使用修改后的类重新解决[第一章](#)给出的平均价格问题。

7.8. Overloaded Functions

7.8. 重载函数

Two functions that appear in the same scope are **overloaded** if they have the same name but have different parameter lists.

出现在相同作用域中的两个函数，如果具有相同的名字而形参表不同，则称为**重载函数**。

If you have written an arithmetic expression in a programming language, you have used an overloaded function. The expression

使用某种程序设计语言编写过算术表达式的程序员都肯定使用过重载函数。表达式

`1 + 3`

invokes the addition operation for integer operands, whereas the expression

调用了针对整型操作数加法操作符，而表达式

`1.0 + 3.0`

invokes a *different* operation that adds floating-point operands. It is the compiler's responsibility, not the programmer's, to distinguish between the different operations and to apply the appropriate operation depending on the operands' types.

调用了另外一个专门处理浮点操作数的不同的加法操作。根据操作数的类型来区分不同的操作，并应用适当的操作，是编译器的责任，而不是程序员的事情。

Similarly, we may define a set of functions that perform the same general action but that apply to different parameter types. These functions may be called without worrying about which function is invoked, much as we can add `ints` or `doubles` without worrying whether integer arithmetic or floating-point arithmetic is performed.

类似地，程序员可以定义一组函数，它们执行同样的一般性动作，但是应用在不同的形参类型上，调用这些函数时，无需担心调用的是哪个函数，就像我们不必操心执行的是整数算术操作还是浮点数自述操作就可以实现 `int` 型加法或 `double` 型加法一样。

Function overloading can make programs easier to write and to understand by eliminating the need to invent and remember names that exist only to help the compiler figure out which function to call. For example, a database application might well have several `lookup` functions that could do the lookup based on name, phone number, account number, and so on. Function overloading allows us to define a collection of functions, each named `lookup`, that differ in terms of what values they use to do the search. We can call `lookup` passing a value of any of several types:

通过省去为函数起名并记住函数名字的麻烦，函数重载简化了程序的实现，使程序更容易理解。函数名只是为了帮助编译器判断调用的是哪个函数而已。例如，一个数据库应用可能需要提供多个 `lookup` 函数，分别实现基于姓名、电话号码或账号之类的查询功能。函数重载使我们可以定义一系列的函数，它们的名字都是 `lookup`，不同之处在于用于查询的值不相同。如此可传递几种类型中的任一种值调用 `lookup` 函数：

```
Record lookup(const Account&); // find by Account
Record lookup(const Phone&); // find by Phone
Record lookup(const Name&); // find by Name
Record r1, r2;
r1 = lookup(acct); // call version that takes an Account
r2 = lookup(phone); // call version that takes a Phone
```

Here, all three functions share the same name, yet they are three distinct functions. The compiler uses the argument type(s) passed in the call to figure out which function to call.

这里的三个函数共享同一个函数名，但却是三个不同的函数。编译器将根据所传递的实参类型来判断调用的是哪个函数。

To understand function overloading, we must understand how to define a set of overloaded functions and how the compiler decides which function to use for a given call. We'll review these topics in the remainder of this section.

要理解函数重载，必须理解如何定义一组重载函数和编译器如何决定对某一调用使用哪个函数。本节的其余部分将会回顾这些主题。

 There may be only one instance of `main` in any program. The `main` function may *not* be overloaded.

任何程序都仅有一个 `main` 函数的实例。`main` 函数不能重载。

Distinguishing Overloading from Redeclaring a Function

函数重载和重复声明的区别

If the return type and parameter list of two functions declarations match exactly, then the second declaration is treated as a redeclaration of the first. If the parameter lists of two functions match exactly but the return types differ, then the second declaration is an error:

如果两个函数声明的返回类型和形参表完全匹配，则将第二个函数声明视为第一个的重复声明。如果两个函数的形参表完全相同，但返回类型不同，则第二个声明是错误的：

```
Record lookup(const Account&);
bool lookup(const Account&); // error: only return type is different
```

Functions cannot be overloaded based only on differences in the return type.

函数不能仅仅基于不同的返回类型而实现重载。

Two parameter lists can be identical, even if they don't look the same:

有些看起来不相同的形参表本质上是相同的：

```
// each pair declares the same function
Record lookup(const Account &acct);
Record lookup(const Account&); // parameter names are ignored
typedef Phone Telno;
Record lookup(const Phone&);

Record lookup(const Telno&); // Telno and Phone are the same type
Record lookup(const Phone&, const Name&);

// default argument doesn't change the number of parameters
Record lookup(const Phone&, const Name& = "");

// const is irrelevant for nonreference parameters
Record lookup(Phone);

Record lookup(const Phone); // redeclaration
```

In the first pair, the first declaration names its parameter. Parameter names are only a documentation aid. They do not change the parameter list.

在第一对函数声明中，第一个声明给它的形参命了名。形参名只是帮助文档，并没有修改形参表。

In the second pair, it looks like the types are different, but `Telno` is not a new type; it is a synonym for `Phone`. A `typedef` name provides an alternative name for an existing data type; it does not create a new data type. Therefore, two parameters that differ only in that one uses a `typedef` and the other uses the type to which the `typedef` corresponds are not different.

在第二对函数声明中，看似形参类型不同，但注意到 `Telno` 其实并不是新类型，只是 `Phone` 类型的同义词。`typedef` 给已存在的数据类型提供别名，但并没有创建新的数据类型。所以，如果两个形参的差别只是一个使用 `typedef` 定义的类型名，而另一个使用 `typedef` 对应的原类型名，则这两个形参并无不同。

In the third pair, the parameter lists differ only in their default arguments. A default argument doesn't change the number of parameters. The function takes two arguments, whether they are supplied by the user or by the compiler.

在第三对中，形参列表只有默认实参不同。默认实参并没有改变形参的个数。无论实参是由用户还是由编译器提供的，这个函数都带有两个实参。

The last pair differs only as to whether the parameter is `const`. This difference has no effect on the objects that can be passed to the function; the second declaration is treated as a redeclaration of the first. The reason follows from how arguments are passed. When the parameter is copied, whether the parameter is `const` is irrelevant—the function executes on a copy. Nothing the function does can change the argument. As a result, we can pass a `const` object to either a `const` or `nonconst` parameter. The two parameters are indistinguishable.

最后一对的区别仅在于是否将形参定义为 `const`。这种差异并不影响传递至函数的对象；第二个函数声明被视为第一个的重复声明。其原因在于实参传递的方式。复制形参时并不考虑形参是否为 `const`——函数操纵的只是副本。函数的无法修改实参。结果，既可将 `const` 对象传递给 `const` 形参，也可传递给非 `const` 形参，这两种形参并无本质区别。

It is worth noting that the equivalence between a parameter and a `const` parameter applies only to nonreference parameters. A function that takes a `const` reference is different from one that takes a `nonconst` reference. Similarly, a function that takes a pointer to a `const` type differs from a function that takes a pointer to the `nonconst` object of the same type.

值得注意的是，形参与 `const` 形参的等价性仅适用于非引用形参。有 `const` 引用形参的函数与有非 `const` 引用形参的函数是不同的。类似地，如果函数带有指向 `const` 类型的指针形参，则与带有指向相同类型的非 `const` 对象的指针形参的函数不相同。

Advice: When Not to Overload a Function Name

建议：何时不重载函数名

Although overloading can be useful in avoiding the necessity to invent (and remember) names for common operations, it is easy to take this advantage too far. There are some cases where providing different function names adds information that makes the program easier to understand. Consider a set of member functions for a `Screen` class that move `Screen`'s cursor.

虽然，对于通常的操作，重载函数能避免不必要的函数命名（和名字记忆），但很容易就会过分使用重载。在一些情况下，使用不同的函数名能提供较多的信息，使程序易于理解。考虑下面 `Screen` 类的一组用于移动屏幕光标的成员函数：

```
Screen& moveHome();
Screen& moveAbs(int, int);
Screen& moveRel(int, int, char *direction);
```

It might at first seem better to overload this set of functions under the name `move`:

乍看上去，似乎把这组函数重载为名为 `move` 的函数更好一些：

```
Screen& move();
Screen& move(int, int);
Screen& move(int, int, *direction);
```

However, by overloading these functions we've lost information that was inherent in the function names and by doing so may have rendered the program more obscure.

其实不然，重载过后的函数失去了原来函数名所包含的信息，如此一来，程序变得晦涩难懂了。

Although cursor movement is a general operation shared by all these functions, the specific nature of that movement is unique to each of these functions. `moveHome`, for example, represents a special instance of cursor movement. Which of the two calls is easier to understand for a reader of the program? Which of the two calls is easier to remember for a programmer using the `Screen` class?

虽则这几个函数共享的一般性动作都是光标移动，但特殊的移动性质却互不相同。例如，`moveHome` 表示的是光标移动的一个特殊实例。对于程序的读者，下面两种调用中，哪种更易于理解？而对于使用 `Screen` 类的程序员，哪一个调用又更容易记忆呢？

```
// which is easier to understand?
myScreen.home(); // we think this one!
myScreen.move();
```

7.8.1. Overloading and Scope

7.8.1. 重载与作用域

We saw in the program on page 54 that scopes in C++ nest. A name declared local to a function hides the same name declared in the global scope (Section 2.3.6, p. 54). The same is true for function names as for variable names:

第 2.3.6 节的程序演示了 C++ 作用域的嵌套。在函数中局部声明的名字将屏蔽在全局作用域（第 2.3.6 节）内声明的同名名字。这个关于变量名字的性质对于函数名同样成立：

```
/* Program for illustration purposes only:
 * It is bad style for a function to define a local variable
 * with the same name as a global name it wants to use
 */
string init(); // the name init has global scope
void fcn()
{
    int init = 0;      // init is local and hides global init
    string s = init(); // error: global init is hidden
}
```

Normal scoping rules apply to names of overloaded functions. If we declare a function locally, that function hides rather than overloads the same function declared in an outer scope. As a consequence, declarations for every version of an overloaded function must appear in the same scope.

Section 7.8. Overloaded Functions

一般的作用域规则同样适用于重载函数名。如果局部地声明一个函数，则该函数将屏蔽而不是重载在外层作用域中声明的同名函数。由此推论，每一个版本的重载函数都应在同一个作用域中声明。



In general, it is a bad idea to declare a function locally. Function declarations should go in header files.

一般来说，局部地声明函数是一种不明智的选择。函数的声明应放在头文件中。

To explain how scope interacts with overloading we will violate this practice and use a local function declaration.

但为了解释作用域与重载的相互作用，我们将违反上述规则而使用局部函数声明。

As an example, consider the following program:

作为例子，考虑下面的程序：

```
void print(const string &);  
void print(double); // overloads the print function  
void fooBar(int ival)  
{  
    void print(int); // new scope: hides previous instances of print  
    print("Value: "); // error: print(const string &) is hidden  
    print(ival); // ok: print(int) is visible  
    print(3.14); // ok: calls print(int); print(double) is hidden  
}
```

The declaration of `print(int)` in the function `fooBar` hides the other declarations of `print`. It is as if there is only one `print` function available: the one that takes a single `int` parameter. Any use of the name `print` at this scope or a scope nested in this scope will resolve to this instance.

函数 `fooBar` 中的 `print(int)` 声明将屏蔽 `print` 的其他声明，就像只有一个有效的 `print` 函数一样：该函数仅带有一个 `int` 型形参。在这个作用域或嵌套在这个作用域里的其他作用域中，名字 `print` 的任何使用都将解释为这个 `print` 函数实例。

When we call `print`, the compiler first looks for a declaration of that name. It finds the local declaration for `print` that takes an `int`. Once the name is found, the compiler does no further checks to see if the name exists in an outer scope. Instead, the compiler assumes that this declaration is the one for the name we are using. What remains is to see if the use of the name is valid.

调用 `print` 时，编译器首先检索这个名字的声明，找到只有一个 `int` 型形参的 `print` 函数的局部声明。一旦找到这个名字，编译器将不再继续检查这个名字是否在外层作用域中存在，即编译器将认同找到的这个声明即是程序需要调用的函数，余下的工作只是检查该名字的使用是否有效。

The first call passes a string literal but the function parameter is an `int`. A string literal cannot be implicitly converted to an `int`, so the call is an error. The `print(const string&)` function, which would have matched this call, is hidden and is not considered when resolving this call.

第一个函数调用传递了一个字符串字面值，但是函数的形参却是 `int` 型的。字符串字面值无法隐式地转换为 `int` 型，因而该调用是错误的。`print(const string&)` 函数与这个函数调用匹配，但已被屏蔽，因此不在解释该调用时考虑。

When we call `print` passing a `double`, the process is repeated. The compiler finds the local definition of `print(int)`. The `double` argument can be converted to an `int`, so the call is legal.

当传递一个 `double` 数据调用 `print` 函数时，编译器重复了同样的匹配过程：首先检索到 `print(int)` 局部声明，然后将 `double` 型的实参隐式转换为 `int` 型。因此，该调用合法。



In C++ name lookup happens before type checking.

在 C++ 中，名字查找发生在类型检查之前。

Had we declared `print(int)` in the same scope as the other `print` functions, then it would be another overloaded version of `print`. In that case, these calls would be resolved differently:

另一种情况是，在与其他 `print` 函数相同的作用域中声明 `print(int)`，这样，它就成为 `print` 函数的另一个重载版本。此时，所有的调用将以不同的方式解释：

```
void print(const string &);  
void print(double); // overloads print function  
void print(int); // another overloaded instance  
void fooBar2(int ival)  
{  
    print("Value: "); // ok: calls print(const string &)  
    print(ival); // ok: print(int)  
    print(3.14); // ok: calls print (double)  
}
```

Now when the compiler looks for the name `print` it finds three functions with that name. On each call it selects the version of `print` that matches

Section 7.8. Overloaded Functions

the argument that is passed.

现在, 编译器在检索名字 `print` 时, 将找到这个名字的三个函数。每一个调用都将选择与其传递的实参相匹配的 `print` 版本。

Exercises Section 7.8.1

Exercise

- 7.34: Define a set of overloaded functions named `error` that would match the following calls:

定义一组名为 `error` 的重载函数, 使之与下面的调用匹配:

```
int index, upperBound;
char selectVal;
// ...
error("Subscript out of bounds: ", index, upperBound);
error("Division by zero");
error("Invalid selection", selectVal);
```

Exercise

- 7.35: Explain the effect of the second declaration in each one of the following sets of declarations.

Indicate which, if any, are illegal.

下面提供了三组函数声明, 解释每组中第二个声明的效果, 并指出哪些 (如果有) 是不合法的。

```
(a) int calc(int, int);
     int calc(const int, const int);

(b) int get();
     double get();

(c) int *reset(int *);
     double *reset(double *);
```

7.8.2. Function Matching and Argument Conversions

7.8.2. 函数匹配与实参转换

Function **overload resolution** (also known as **function matching**) is the process by which a function call is associated with a specific function from a set of overloaded functions. The compiler matches a call to a function automatically by comparing the actual arguments used in the call with the parameters offered by each function in the overload set. There are three possible outcomes:

函数**重载确定**, 即**函数匹配**是将函数调用与重载函数集合中的一个函数相关联的过程。通过自动提取函数调用中实际使用的实参与重载集合中各个函数提供的形参做比较, 编译器实现该调用与函数的匹配。匹配结果有三种可能:

1. The compiler finds one function that is a **best match** for the actual arguments and generates code to call that function.
编译器找到与实参**最佳匹配**的函数, 并生成调用该函数的代码。
2. There is no function with parameters that match the arguments in the call, in which case the compiler indicates a compile-time error.
找不到形参与函数调用的实参匹配的函数, 在这种情况下, 编译器将给出编译错误信息。
3. There is more than one function that matches and none of the matches is clearly best. This case is also an error; the call is **ambiguous**.
存在多个与实参匹配的函数, 但没有一个是明显最佳选择。这种情况也是, 该调用具有**二义性**。

Most of the time it is straightforward to determine whether a particular call is legal and if so, which function will be invoked by the compiler. Often the functions in the overload set differ in terms of the number of arguments, or the types of the arguments are unrelated. Function matching gets tricky when multiple functions have parameters that are related by conversions (Section 5.12, p. 178). In these cases, programmers need to have a good grasp of the process of function matching.

大多数情况下, 编译器都可以直接明确地判断一个实际的调用是否合法, 如果合法, 则应该调用哪一个函数。重载集合中的函数通常有不同个数的参数或无关联的参数类型。当多个函数的形参具有可通过隐式转换 (第 5.12 节) 关联起来的类型, 则函数匹配将相当灵活。在这种情况下, 需要程序员充分地掌握函数匹配的过程。

7.8.3. The Three Steps in Overload Resolution

7.8.3. 重载确定的三个步骤

Section 7.8. Overloaded Functions

Consider the following set of functions and function call:

考虑下面的这组函数和函数调用：

```
void f();  
void f(int);  
void f(int, int);  
void f(double, double = 3.14);  
f(5.6); // calls void f(double, double)
```

Candidate Functions

候选函数

The first step of function overload resolution identifies the set of overloaded functions considered for the call. The functions in this set are the **candidate functions**. A candidate function is a function with the same name as the function that is called and for which a declaration is visible at the point of the call. In this example, there are four candidate functions named `f`.

函数重载确定的第一步是确定该调用所考虑的重载函数集合，该集合中的函数称为**候选函数**。候选函数是与被调函数同名的函数，并且在调用点上，它的声明可见。在这个例子中，有四个名为 `f` 的候选函数。

Determining the Viable Functions

选择可行函数

The second step selects the functions from the set of candidate functions that can be called with the arguments specified in the call. The selected functions are the **viable functions**. To be viable, a function must meet two tests. First, the function must have the same number of parameters as there are arguments in the call. Second, the type of each argument must match or be convertible to the type of its corresponding parameter.

第二步是从候选函数中选择一个或多个函数，它们能够用该调用中指定的实参来调用。因此，选出来的函数称为**可行函数**。可行函数必须满足两个条件：第一，函数的形参数个数与该调用的实参数数相同；第二，每一个实参的类型必须与对应形参的类型匹配，或者可被隐式转换为对应的形参类型。



When a function has default arguments ([Section 7.4.1, p. 253](#)), a call may appear to have fewer arguments than it actually does. Default arguments are arguments and are treated the same way as any other argument during function matching.

如果函数具有默认实参（[第 7.4.1 节](#)），则调用该函数时，所用的实参可能比实际需要的少。默认实参也是实参，在函数匹配过程中，它的处理方式与其他实参一样。

For the call `f(5.6)`, we can eliminate two of our candidate functions because of a mismatch on number of arguments. The function that has no parameters and the one that has two `int` parameters are not viable for this call. Our call has only one argument, and these functions have zero and two parameters, respectively.

对于函数调用 `f(5.6)`，可首先排除两个实参数不匹配的候选函数。没有形参的 `f` 函数和有两个 `int` 型形参的 `f` 函数对于这个函数调用来说都不可行。例中的调用只有一个实参，而这些函数分别带有零个和两个形参。

On the other hand, the function that takes two `doubles` might be viable. A call to a function declaration that has a default argument ([Section 7.4.1, p. 253](#)) may omit that argument. The compiler will automatically supply the default argument value for the omitted argument. Hence, a given call might have more arguments than appear explicitly.

另一方面，有两个 `double` 型参数的 `f` 函数可能是可行的。调用带有默认实参（[第 7.4.1 节](#)）的函数时可忽略这个实参。编译器自动将默认实参的值提供给被忽略的实参。因此，某个调用拥有的实参可能比显式给出的多。

Having used the number of arguments to winnow the potentially viable functions, we must now look at whether the argument types match those of the parameters. As with any call, an argument might match its parameter either because the types match exactly or because there is a conversion from the argument type to the type of the parameter. In the example, both of our remaining functions are viable.

根据实参数选出潜在的可行函数后，必须检查实参的类型是否与对应的形参类型匹配。与任意函数调用一样，实参必须与它的形参匹配，它们的类型要么精确匹配，要么实参类型能够转换为形参类型。在这个例子中，余下的两个函数都是可行的。

- `f(int)` is a viable function because a conversion exists that can convert the argument of type `double` to the parameter of type `int`.

`f(int)` 是一个可行函数，因为通过隐式转换可将函数调用中的 `double` 型实参转换为该函数唯一的 `int` 型形参。

Section 7.8. Overloaded Functions

- `f(double, double)` is a viable function because a default argument is provided for the function's second parameter and its first parameter is of type `double`, which exactly matches the type of the parameter.

`f(double, double)` 也是一个可行函数，因为该函数为其第二个形参提供了默认实参，而且第一个形参是 `double` 类型，与实参类型精确匹配。



If there are no viable functions, then the call is in error.

如果没有找到可行函数，则该调用错误。

Finding the Best Match, If Any

寻找最佳匹配 (如果有的话)

The third step of function overload resolution determines which viable function has the best match for the actual arguments in the call. This process looks at each argument in the call and selects the viable function (or functions) for which the corresponding parameter best matches the argument. The details of "best" here will be explained in the next section, but the idea is that the closer the types of the argument and parameter are to each other, the better the match. So, for example, an exact type match is better than a match that requires a conversion from the argument type to the parameter type.

函数重载确定的第三步是确定与函数调用中使用的实际参数匹配最佳的可行函数。这个过程考虑函数调用中的每一个实参，选择对应形参与之最匹配的一个或多个可行函数。这里所谓“最佳”的细节将在下一节中解释，其原则是实参类型与形参类型越接近则匹配越佳。因此，实参类型与形参类型之间的精确类型匹配比需要转换的匹配好。

In our case, we have only one explicit argument to consider. That argument has type `double`. To call `f(int)`, the argument would have to be converted from `double` to `int`. The other viable function, `f(double, double)`, is an exact match for this argument. Because an exact match is better than a match that requires a conversion, the compiler will resolve the call `f(5.6)` as a call to the function that has two `double` parameters.

在上述例子中，只需考虑一个 `double` 类型的显式实参。如果调用 `f(int)`，实参需从 `double` 型转换为 `int` 型。而另一个可行函数 `f(double, double)` 则与该实参精确匹配。由于精确匹配优于需要类型转换的匹配，因此编译器将会把函数调用 `f(5.6)` 解释为对带有两个 `double` 形参的 `f` 函数的调用。

Overload Resolution with Multiple Parameters

含有多个形参的重载确定

Function matching is more complicated if there are two or more explicit arguments. Given the same functions named `f`, let's analyze the following call:

如果函数调用使用了两个或两个以上的显式实参，则函数匹配会更加复杂。假设有两样的名为 `f` 的函数，分析下面的函数调用：

```
f(42, 2.56);
```

The set of viable functions is selected in the same way. The compiler selects those functions that have the required number of parameters and for which the argument types match the parameter types. In this case, the set of viable functions are `f(int, int)` and `f(double, double)`. The compiler then determines argument by argument which function is (or functions are) the best match. There is a match if there is one and only one function for which

可行函数将以同样的方式选出。编译器将选出形参个数和类型都与实参匹配的函数。在本例中，可行函数是 `f(int, int)` 和 `f(double, double)`。接下来，编译器通过依次检查每一个实参来决定哪个或哪些函数匹配最佳。如果有且仅有一个函数满足下列条件，则匹配成功：

1. The match for each argument is no worse than the match required by any other viable function.

其每个实参的匹配都不劣于其他可行函数需要的匹配。

2. There is at least one argument for which the match is better than the match provided by any other viable function.

至少有一个实参的匹配优于其他可行函数提供的匹配。

If after looking at each argument there is no single function that is preferable, then the call is in error. The compiler will complain that the call is ambiguous.

如果在检查了所有实参后，仍找不到唯一最佳匹配函数，则该调用错误。编译器将提示该调用具有二义性。

In this call, when we look only at the first argument, we find that the function `f(int, int)` is an exact match. To match the second function, the `int` argument `42` must be converted to a `double`. A match through a built-in conversion is "less good" than one that is exact. So, considering only this parameter, the function that takes two `ints` is a better match than the function that takes two `doubles`.

Section 7.8. Overloaded Functions

在本例子的调用中，首先分析第一个实参，发现函数 `f(int, int)` 匹配精确。如果使之与第二个函数匹配，就必须将 `int` 型实参 `42` 转换为 `double` 型的值。通过内置转换的匹配“劣于”精确匹配。所以，如果只考虑这个形参，带有两个 `int` 型形参的函数比带有两个 `double` 型形参的函数匹配更佳。

However, when we look at the second argument, then the function that takes two `doubles` is an exact match to the argument `2.56`. Calling the version of `f` that takes two `ints` would require that `2.56` be converted from `double` to `int`. When we consider only the second parameter, then the function `f(double, double)` is the better match.

但是，当分析第二个实参时，有两个 `double` 型形参的函数为实参 `2.56` 提供了精确匹配。而调用两个 `int` 型形参的 `f` 函数版本则需要把 `2.56` 从 `double` 型转换为 `int` 型。所以只考虑第二个形参的话，函数 `f(double, double)` 匹配更佳。

This call is therefore ambiguous: Each viable function is a better match on one of the arguments to the call. The compiler will generate an error. We could force a match by explicitly casting one of our arguments:

因此，这个调用有二义性：每个可行函数都对函数调用的一个实参实现更好的匹配。编译器将产生错误。解决这样的二义性，可通过显式的强制类型转换强制函数匹配：

```
f(static_cast<double>(42), 2.56); // calls f(double, double)
f(42, static_cast<int>(2.56));    // calls f(int, int)
```



In practice, arguments should not need casts when calling over-loaded functions: The need for a cast means that the parameter sets are designed poorly.

在实际应用中，调用重载函数时应尽量避免对实参做强制类型转换：需要使用强制类型转换意味着所设计的形参数合不合理。

Exercises Section 7.8.3

Exercise

7.36: What is a candidate function? What is a viable function?

什么是候选函数？什么是可行函数？

Exercise

7.37: Given the declarations for `f`, determine whether the following calls are legal. For each call list the viable functions, if any. If the call is illegal, indicate whether there is no match or why the call is ambiguous. If the call is legal, indicate which function is the best match.

已知本节所列出的 `f` 函数的声明，判断下面哪些函数调用是合法的。如果有的话，列出每个函数调用的可行函数。如果调用非法，指出是没有函数匹配还是该调用存在二义性。如果调用合法，指出哪个函数是最佳匹配。

- (a) `f(2.56, 42);`
- (b) `f(42);`
- (c) `f(42, 0);`
- (d) `f(2.56, 3.14);`

7.8.4. Argument-Type Conversions

7.8.4. 实参类型转换

In order to determine the best match, the compiler ranks the conversions that could be used to convert each argument to the type of its corresponding parameter. Conversions are ranked in descending order as follows:

为了确定最佳匹配，编译器将实参类型到相应形参类型转换划分等级。转换等级以降序排列如下：

1. An exact match. The argument and parameter types are the same.

精确匹配。实参与形参类型相同。

2. Match through a promotion ([Section 5.12.2](#), p. [180](#)).

通过类型提升实现的匹配（[第 5.12.2 节](#)）。

3. Match through a standard conversion ([Section 5.12.3](#), p. [181](#)).

通过标准转换实现的匹配（[第 5.12.3 节](#)）。

4. Match through a class-type conversion. ([Section 14.9](#) (p. [535](#)) covers these conversions.)

Section 7.8. Overloaded Functions

通过类型转换实现的匹配（[第 14.9 节](#)将介绍这类转换）。



Promotions and conversions among the built-in types can yield surprising results in the context of function matching. Fortunately, well-designed systems rarely include functions with parameters as closely related as those in the following examples.

内置类型的提升和转换可能会使函数匹配产生意想不到的结果。但幸运的是，设计良好的系统很少会包含与下面例子类似的形参类型如此接近的函数。

These examples bear study to cement understanding both of function matching in particular and of the relationships among the built-in types in general.

通过这些例子，学习并加深了解特殊的函数匹配和内置类型之间的一般关系。

Matches Requiring Promotion or Conversion

需要类型提升或转换的匹配

Promotions or conversions are applied when the type of the argument can be promoted or converted to the appropriate parameter type using one of the standard conversions.

类型提升或转换适用于实参类型可通过某种标准转换提升或转换为适当的形参类型情况。

One important point to realize is that the small integral types promote to `int`. Given two functions, one of which takes an `int` and the other a `short`, the `int` version will be a better match for a value of any integral type other than `short`, even though `short` might appear on the surface to be a better match:

必须注意的一个重点是较小的整型提升为 `int` 型。假设有两个函数，一个的形参为 `int` 型，另一个的形参则是 `short` 型。对于任意整型的实参值，`int` 型版本都是优于 `short` 型版本的较佳匹配，即使从形式上看 `short` 型版本的匹配较佳：

```
void ff(int);
void ff(short);
ff('a'); // char promotes to int, so matches ff(int)
```

A character literal is type `char`, and `chars` are promoted to `int`. That promoted type matches the type of the parameter of function `ff(int)`. A `char` could also be converted to `short`, but a conversion is a "less good" match than a promotion. And so this call will be resolved as a call to `ff (int)`.

字符字面值是 `char` 类型，`char` 类型可提升为 `int` 型。提升后的类型与函数 `ff(int)` 的形参类型匹配。`char` 类型同样也可转换为 `short` 型，但需要类型转换的匹配“劣于”需要类型提升的匹配。结果应将该调用解释为对 `ff (int)` 的调用。

A conversion that is done through a promotion is preferred to another standard conversion. So, for example, a `char` is a better match for a function taking an `int` than it is for a function taking a `double`. All other standard conversions are treated as equivalent. The conversion from `char` to `unsigned char`, for example, does not take precedence over the conversion from `char` to `double`. As a concrete example, consider:

通过类型提升实现的转换优于其他标准转换。例如，对于 `char` 型实参来说，有 `int` 型形参的函数是优于有 `double` 型形参的函数的较佳匹配。其他的标准转换也以相同的规则处理。例如，从 `char` 型到 `unsigned char` 型的转换的优先级不比从 `char` 型到 `double` 型的转换高。再举一个具体的例子，考虑：

```
extern void manip(long);
extern void manip(float);
manip(3.14); // error: ambiguous call
```

The literal constant 3.14 is a `double`. That type could be converted to either `long` or `float`. Because there are two possible standard conversions, the call is ambiguous. No one standard conversion is given precedence over another.

字面值常量 3.14 的类型为 `double`。这种类型既可转为 `long` 型也可转为 `float` 型。由于两者都是可行的标准转换，因此该调用具有二义性。没有哪个标准转换比其他标准转换具有更高的优先级。

Parameter Matching and Enumerations

参数匹配和枚举类型

Recall that an object of `enum` type may be initialized only by another object of that `enum` type or one of its enumerators ([Section 2.7, p. 63](#)). An integral object that happens to have the same value as an enumerator cannot be used to call a function expecting an `enum` argument:

回顾枚举类型 `enum`，我们知道这种类型的对象只能用同一枚举类型的另一个对象或一个枚举成员进行初始化（[第 2.7 节](#)）。整数对象即使具有与枚举元素相同的值也不能用于调用期望获得枚举类型实参的函数。

Section 7.8. Overloaded Functions

```
enum Tokens {INLINE = 128, VIRTUAL = 129};  
void ff(Tokens);  
void ff(int);  
int main() {  
    Tokens curTok = INLINE;  
    ff(128); // exactly matches ff(int)  
    ff(INLINE); // exactly matches ff(Tokens)  
    ff(curTok); // exactly matches ff(Tokens)  
    return 0;  
}
```

The call that passes the literal 128 matches the version of `ff` that takes an `int`.

传递字面值常量 128 的函数调用与有一个 `int` 型参数的 `ff` 版本匹配。

Although we cannot pass an integral value to a `enum` parameter, we can pass an `enum` to a parameter of integral type. When we do so, the `enum` value promotes to `int` or to a larger integral type. The actual promotion type depends on the values of the enumerators. If the function is overloaded, the type to which the `enum` promotes determines which function is called:

虽然无法将整型值传递给枚举类型的形参，但可以将枚举值传递给整型形参。此时，枚举值被提升为 `int` 型或更大的整型。具体的提升类型取决于枚举成员的值。如果是重载函数，枚举值提升后的类型将决定调用哪个函数：

```
void newf(unsigned char);  
void newf(int);  
unsigned char uc = 129;  
newf(VIRTUAL); // calls newf(int)  
newf(uc); // calls newf(unsigned char)
```

The `enum Tokens` has only two enumerators, the largest of which has a value of 129. That value can be represented by the type `unsigned char`, and many compilers would store the `enum` as an `unsigned char`. However, the type of `VIRTUAL` is not `unsigned char`. Enumerators and values of an `enum` type, are not promoted to `unsigned char`, even if the values of the enumerators would fit.

枚举类型 `Tokens` 只有两个枚举成员，最大的值为 129。这个值可以用 `unsigned char` 类型表示，很多编译器会将这个枚举类型存储为 `unsigned char` 类型。然而，枚举成员 `VIRTUAL` 却并不是 `unsigned char` 类型。就算枚举成员的值能存储在 `unsigned char` 类型中，枚举成员和枚举类型的值也不会提升为 `unsigned char` 类型。



When using overloaded functions with `enum` parameters, remember: Two enumeration types may behave quite differently during function overload resolution, depending on the value of their enumeration constants. The enumerators determine the type to which they promote. And that type is machine-dependent.

在使用有枚举类型形参的重载函数时，请记住：由于不同枚举类型的枚举常量值不相同，在函数重载确定过程中，不同的枚举类型会具有完全不同的行为。其枚举成员决定了它们提升的类型，而所提升的类型依赖于机器。

Overloading and `const` Parameters

重载和 `const` 形参



Whether a parameter is `const` only matters when the parameter is a reference or pointer.

仅当形参是引用或指针时，形参是否为 `const` 才有影响。

We can overload a function based on whether a reference parameter refers to a `const` or non`const` type. Overloading on `const` for a reference parameter is valid because the compiler can use whether the argument is `const` to determine which function to call:

可基于函数的引用形参是指向 `const` 对象还是指向非 `const` 对象，实现函数重载。将引用形参定义为 `const` 来重载函数是合法的，因为编译器可以根据实参是否为 `const` 确定调用哪一个函数：

```
Record lookup(Account&);  
Record lookup(const Account&); // new function  
const Account a(0);  
Account b;  
lookup(a); // calls lookup(const Account&)  
lookup(b); // calls lookup(Account&)
```

If the parameter is a plain reference, then we may not pass a `const` object for that parameter. If we pass a `const` object, then the only function that is viable is the version that takes a `const` reference.

如果形参是普通的引用，则不能将 `const` 对象传递给这个形参。如果传递了 `const` 对象，则只有带 `const` 引用形参的版本才是该调用的可行函数。

When we pass a non`const` object, either function is viable. We can use a non`const` object to initializer either a `const` or non`const` reference. However, initializing a `const` reference to a non`const` object requires a conversion, whereas initializing a non`const` parameter is an exact match.

Section 7.8. Overloaded Functions

如果传递的是非 `const` 对象，则上述任意一种函数皆可行。非 `const` 对象既可用于初始化 `const` 引用，也可用于初始化非 `const` 引用。但是，将 `const` 引用初始化为非 `const` 对象，需通过转换来实现，而非 `const` 形参的初始化则是精确匹配。

Pointer parameters work in a similar way. We may pass the address of a `const` object only to a function that takes a pointer to `const`. We may pass a pointer to a non`const` object to a function taking a pointer to a `const` or non`const` type. If two functions differ only as to whether a pointer parameter points to `const` or non`const`, then the parameter that points to the non`const` type is a better match for a pointer to a non`const` object. Again, the compiler can distinguish: If the argument is `const`, it calls the function that takes a `const*`; otherwise, if the argument is a non`const`, the function taking a plain pointer is called.

对指针形参的相关处理如出一辙。可将 `const` 对象的地址值只传递给带有指向 `const` 对象的指针形参的函数。也可将指向非 `const` 对象的指针传递给函数的 `const` 或非 `const` 类型的指针形参。如果两个函数仅在指针形参时是否指向 `const` 对象上不同，则指向非 `const` 对象的指针形参对于指向非 `const` 对象的指针（实参）来说是更佳的匹配。重复强调，编译器可以判断：如果实参是 `const` 对象，则调用带有 `const*` 类型形参的函数；否则，如果实参不是 `const` 对象，将调用带有普通指针形参的函数。

It is worth noting that we cannot overload based on whether the pointer itself is `const`:

注意不能基于指针本身是否为 `const` 来实现函数的重载：

```
f(int *);  
f(int *const); // redeclaration
```

Here the `const` applies to the pointer, not the type to which the pointer points. In both cases the pointer is copied; it makes no difference whether the pointer itself is `const`. As we noted on page 267, when a parameter is passed as a copy, we cannot overload based on whether that parameter is `const`.

此时，`const` 用于修改指针本身，而不是修饰指针所指向的类型。在上述两种情况中，都复制了指针，指针本身是否为 `const` 并没有带来区别。正如前面第 7.8 节所提到的，当形参以副本传递时，不能基于形参是否为 `const` 来实现重载。

Exercises Section 7.8.4

Exercise

7.38: Given the following declarations,

给出如下声明：

```
void manip(int, int);  
double dobj;
```

what is the rank (Section 7.8.4, p. 272) of each conversion in the following calls?

对于下面两组函数调用，请指出实参上每个转换的优先级等级（第 7.8.4 节）？

(a) `manip('a', 'z');` (b) `manip(55.4, dobj);`

Exercise

7.39: Explain the effect of the second declaration in each one of the following sets of declarations.

Indicate which, if any, are illegal.

解释以下每组声明中的第二个函数声明所造成的影响，并指出哪些不合法（如果有的话）。

(a) `int calc(int, int);`
`int calc(const int&, const int&);`

(b) `int calc(char*, char*);`
`int calc(const char*, const char*);`

(c) `int calc(char*, char*);`
`int calc(char* const, char* const);`

Exercise

7.40: Is the following function call legal? If not, why is the call in error?

下面的函数调用是否合法？如果不合法，请解释原因。

```
enum Stat { Fail, Pass };  
void test(Stat);  
test(0);
```

7.9. Pointers to Functions

7.9. 指向函数的指针

A function pointer is just thata pointer that denotes a function rather than an object. Like any other pointer, a function pointer points to a particular type. A function's type is determined by its return type and its parameter list. A function's name is not part of its type:

函数指针是指指向函数而非指向对象的指针。像其他指针一样，函数指针也指向某个特定的类型。函数类型由其返回类型以及形参表确定，而与函数名无关：

```
// pf points to function returning bool that takes two const string references
bool (*pf)(const string &, const string &);
```

This statement declares `pf` to be a pointer to a function that takes two `const string&` parameters and has a return type of `bool`.

这个语句将 `pf` 声明为指向函数的指针，它所指向的函数带有两个 `const string&` 类型的形参和 `bool` 类型的返回值。



The parentheses around `*pf` are necessary:

`*pf` 两侧的圆括号是必需的：

```
// declares a function named pf that returns a bool*
bool *pf(const string &, const string &);
```

Using Typedefs to Simplify Function Pointer Definitions

用 `typedef` 简化函数指针的定义

Function pointer types can quickly become unwieldy. We can make function pointers easier to use by defining a synonym for the pointer type using a `typedef` ([Section 2.6](#), p. 61):

函数指针类型相当地冗长。使用 `typedef` 为指针类型定义同义词，可将函数指针的使用大大简化：([第 2.6 节](#))：

```
typedef bool (*cmpFcn)(const string &, const string &);
```

This definition says that `cmpFcn` is the name of a type that is a pointer to function. That pointer has the type "pointer to a function that returns a `bool` and takes two references to `const string`." When we need to use this function pointer type, we can do so by using `cmpFcn`, rather than having to write the full type definition each time.

该定义表示 `cmpFcn` 是一种指向函数的指针类型的名字。该指针类型为“指向返回 `bool` 类型并带有两个 `const string` 引用形参的函数的指针”。在要使用这种函数指针类型时，只需直接使用 `cmpFcn` 即可，不必每次都把整个类型声明全部写出来。

Initializing and Assigning Pointers to Functions

指向函数的指针的初始化和赋值

When we use a function name without calling it, the name is automatically treated as a pointer to a function. Given

在引用函数名但又没有调用该函数时，函数名将被自动解释为指向函数的指针。假设有函数：

```
// compares lengths of two strings
bool lengthCompare(const string &, const string &);
```

any use of `lengthCompare`, except as the left-hand operand of a function call, is treated as a pointer whose type is

除了用作函数调用的左操作数以外，对 `lengthCompare` 的任何使用都被解释为如下类型的指针：

```
bool (*) (const string &, const string &);
```

Section 7.9. Pointers to Functions

We can use a function name to initialize or assign to a function pointer:

可使用函数名对函数指针做初始化或赋值：

```
cmpFcn pf1 = 0;           // ok: unbound pointer to function
cmpFcn pf2 = lengthCompare; // ok: pointer type matches function's type
pf1 = lengthCompare;       // ok: pointer type matches function's type
pf2 = pf1;                 // ok: pointer types match
```

Using the function name is equivalent to applying the address-of operator to the function name:

此时，直接引用函数名等效于在函数名上应用取地址操作符：

```
cmpFcn pf1 = lengthCompare;
cmpFcn pf2 = &lengthCompare;
```



A function pointer may be initialized or assigned only by a function or function pointer that has the same type or by a zero-valued constant expression.

函数指针只能通过同类型的函数或函数指针或 0 值常量表达式进行初始化或赋值。

Initializing a function pointer to zero indicates that the pointer does not point to any function.

将函数指针初始化为 0，表示该指针不指向任何函数。

There is no conversion between one pointer to function type and another:

指向不同函数类型的指针之间不存在转换：

```
string::size_type sumLength(const string&, const string&);
bool cstringCompare(char*, char*);  

// pointer to function returning bool taking two const string&
cmpFcn pf;
pf = sumLength;      // error: return type differs
pf = cstringCompare; // error: parameter types differ
pf = lengthCompare; // ok: function and pointer types match exactly
```

Calling a Function through a Pointer

通过指针调用函数

A pointer to a function can be used to call the function to which it refers. We can use the pointer directly there is no need to use the dereference operator to call the function

指向函数的指针可用于调用它所指向的函数。可以不需要使用解引用操作符，直接通过指针调用函数：

```
cmpFcn pf = lengthCompare;
lengthCompare("hi", "bye"); // direct call
pf("hi", "bye");          // equivalent call: pf1 implicitly dereferenced
(*pf)("hi", "bye");       // equivalent call: pf1 explicitly dereferenced
```



If the pointer to function is uninitialized or has a value of zero, it may not be used in a call. Only pointers that have been initialized or assigned to refer to a function can be safely used to call a function.

如果指向函数的指针没有初始化，或者具有 0 值，则该指针不能在函数调用中使用。只有当指针已经初始化，或被赋值为指向某个函数，方能安全地用来调用函数。

Function Pointer Parameters

函数指针形参

A function parameter can be a pointer to function. We can write such a parameter in one of two ways:

函数的形参可以是指向函数的指针。这种形参可以用以下两种形式编写：

[View full width]

```
/* useBigger function's third parameter is a pointer to function
 * that function returns a bool and takes two const string references
 * two ways to specify that parameter:
 */
// third parameter is a function type and is automatically treated as a pointer to
function
void useBigger(const string &, const string &,
               bool(const string &, const string &));
// equivalent declaration: explicitly define the parameter as a pointer to function
void useBigger(const string &, const string &,
               bool (*)(const string &, const string &));
```

Returning a Pointer to Function

返回指向函数的指针

A function can return a pointer to function, although correctly writing the return type can be a challenge:

函数可以返回指向函数的指针，但是，正确写出这种返回类型相当不容易：

```
// ff is a function taking an int and returning a function pointer
// the function pointed to returns an int and takes an int* and an int
int (*ff(int))(int*, int);
```



The best way to read function pointer declarations is from the inside out, starting with the name being declared.

阅读函数指针声明的最佳方法是从声明的名字开始由里而外理解。

We can figure out what this declaration means by observing that

要理解该声明的含义，首先观察：

```
ff(int)
```

says that `ff` is a function taking one parameter of type `int`. This function returns

将 `ff` 声明为一个函数，它带有一个 `int` 型的形参。该函数返回

```
int (*)(int*, int);
```

a pointer to a function that returns an `int` and takes two parameters of type `int*` and an `int`.

它是一个指向函数的指针，所指向的函数返回 `int` 型并带有两个分别是 `int*` 型和 `int` 型的形参。

Typedefs can make such declarations considerably easier to read:

使用 `typedef` 可使该定义更简明易懂：

```
// PF is a pointer to a function returning an int, taking an int* and an int
typedef int (*PF)(int*, int);
PF ff(int); // ff returns a pointer to function
```



We can define a parameter as a function type. A function return type must be a pointer to function; it cannot be a function.

允许将形参定义为函数类型，但函数的返回类型则必须是指向函数的指针，而不能是函数。

An argument to a parameter that has a function type is automatically converted to the corresponding pointer to function type. The same conversion does not happen when returning a function:

具有函数类型的形参所对应的实参将被自动转换为指向相应函数类型的指针。但是，当返回的是函数时，同样的转换操作则无法实现：

```
// func is a function type, not a pointer to function!
```

Section 7.9. Pointers to Functions

```
typedef int func(int*, int);
void f1(func); // ok: f1 has a parameter of function type
func f2(int); // error: f2 has a return type of function type
func *f3(int); // ok: f3 returns a pointer to function type
```

Pointers to Overloaded Functions

指向重载函数的指针

It is possible to use a function pointer to refer to an overloaded function:

C++ 语言允许使用函数指针指向重载的函数：

```
extern void ff(vector<double>);
extern void ff(unsigned int);

// which function does pf1 refer to?
void (*pf1)(unsigned int) = &ff; // ff(unsigned)
```

The type of the pointer and one of the overloaded functions must match exactly. If no function matches exactly, the initialization or assignment results in a compile-time error:

指针的类型必须与重载函数的一个版本精确匹配。如果没有精确匹配的函数，则对该指针的初始化或赋值都将导致编译错误：

```
// error: no match: invalid parameter list
void (*pf2)(int) = &ff;

// error: no match: invalid return type
double (*pf3)(vector<double>);
pf3 = &ff;
```

Chapter Summary

小结

Functions are named units of computation and are essential to structuring even modest programs. They are defined by specifying a return type, a name, a (possibly empty) list of parameters, and a function body. The function body is a block that is executed when the function is called. When a function is called, the arguments passed to the function must be compatible with the types of the corresponding parameters.

函数是有名字的计算单元，对程序（就算是小程序）的结构化至关重要。函数的定义由返回类型、函数名、形参表（可能为空）以及函数体组成。函数体是调用函数时执行的语句块。在调用函数时，传递给函数的实参必须与相应的形参类型兼容。

Passing an argument to a function follows the same rules as initializing a variable. Each parameter that has nonreference type is initialized as a copy of the corresponding argument. Any changes made to a (nonreference) parameter are made to the local copy, not to the argument itself.

给函数传递实参遵循变量初始化的规则。非引用类型的形参以相应实参的副本初始化。对（非引用）形参的任何修改仅作用于局部副本，并不影响实参本身。

Copying large, complex values can be expensive. To avoid the overhead of passing a copy, parameters can be specified as references. Changes made to reference parameters are reflected in the argument itself. A reference parameter that does not need to change its argument should be `const` reference.

复制庞大而复杂的值有昂贵的开销。为了避免传递副本的开销，可将形参指定为引用类型。对引用形参的任何修改会直接影响实参本身。应将不需要修改相应实参的引用形参定义为 `const` 引用。

In C++, functions may be overloaded. The same name may be used to define different functions as long as the number or types of the parameters in the functions differ. The compiler automatically figures out which function to call based the arguments in a call. The process of selecting the right function from a set of overloaded functions is referred to as function matching.

在 C++ 中，函数可以重载。只要函数中形参的个数或类型不同，则同一个函数名可用于定义不同的函数。编译器将根据函数调用时的实参确定调用哪一个函数。在重载函数集合中选择适合的函数的过程称为函数匹配。

C++ provides two special kinds of functions: `inline` and member functions. Specifying `inline` on a function is a hint to the compiler to expand the function into code directly at the call point. Inline functions avoid the overhead associated with calling a function. Member functions are just that: class members that are functions. This chapter introduced simple member functions. [Chapter 12](#) will cover member functions in more detail.

C++ 提供了两种特殊的函数：内联函数和成员函数。将函数指定为内联是建议编译器在调用点直接把函数代码展开。内联函数避免了调用函数的代价。成员函数则是身为类成员的函数。本章介绍了简单的成员函数，在[第十二章](#)将会更详细地介绍成员函数。

Defined Terms

术语

ambiguous call (有二义性的调用)

Compile-time error that results when there is not a single best match for a call to an overloaded function.

一种编译错误，当调用重载函数，找不到唯一最佳匹配时产生。

arguments (实参)

Values supplied when calling a function. These values are used to initialize the corresponding parameters in the same way that variables of the same type are initialized.

调用函数时提供的值。这些值用于初始化相应的形参，其方式类似于初始化同类型变量的方法。

automatic objects (自动对象)

Objects that are local to a function. Automatic objects are created and initialized anew on each call and are destroyed at the end of the block in which they are defined. They no longer exist once the function terminates.

局部于函数的对象。自动对象会在每一次函数调用时重新创建和初始化，并在定义它的函数块结束时撤销。一旦函数执行完毕，这些对象就不再存在了。

best match (最佳匹配)

The single function from a set of overloaded functions that has the best match for the arguments of a given call.

在重载函数集合里找到的与给定调用的实参数达到最佳匹配的唯一函数。

call operator (调用操作符)

The operator that causes a function to be executed. The operator is a pair of parentheses and takes two operands: The name of the function to call and a (possibly empty) comma-separated list of arguments to pass to the function.

使函数执行的操作符。该操作符是一对圆括号，并且有两个操作数：被调用函数的名字，以及由逗号分隔的（也可能为空）形参数。

candidate functions (候选函数)

The set of functions that are considered when resolving a function call. The candidate functions are all the functions with the name used in the call for which a declaration is in scope at the time of the call.

在解析函数调用时考虑的函数集合。候选函数包括了所有在该调用发生的作用域中声明的、具有该调用所使用的名字的函数。

const member function (常量成员函数)

Function that is member of a class and that may be called for `const` objects of that type. `const` member functions may not change the data members of the object on which they operate.

类的成员函数，并可以由该类类型的常量对象调用。常量成员函数不能修改所操纵的对象的数据成员。

constructor (构造函数)

Member function that has the same name as its class. A constructor says how to initialize objects of its class. Constructors have no return type. Constructors may be overloaded.

与所属类同名的类成员函数。构造函数说明如何初始化本类的对象。构造函数没有返回类型，而且可以重载。

constructor initializer list (构造函数初始化列表)

List used in a constructor to specify initial values for data members. The initializer list appears in the definition of a constructor between the parameter list and the constructor body. The list consists of a colon followed by a comma-separated list of member names, each of which is followed by that member's initial value in parentheses.

在构造函数中用于为数据成员指定初值的表。初始化列表出现在构造函数的定义中，位于构造函数体与形参数之间。该表由冒号和冒号后面的一组用逗号分隔的成员名

Keyterm Defined Terms

组成，每一个成员名后面跟着用圆括号括起来的该成员的初值。

default constructor (默认构造函数)

The constructor that is used when no explicit initializer is supplied. The compiler will synthesize a default constructor if the class defines no other constructors.

在没有显式提供初始化式时调用的构造函数。如果类中没有定义任何构造函数，编译器会自动为这个类合成默认构造函数。

function (函数)

A callable unit of computation.

可调用的计算单元。

function body (函数体)

Block that defines the actions of a function.

定义函数动作的语句块。

function matching (函数匹配)

Compiler process by which a call to an overloaded function is resolved. Arguments used in the call are compared to the parameter list of each overloaded function.

确定重载函数调用的编译器过程。调用时使用的实参将与每个重载函数的形参表作比较。

function prototype (函数原型)

Synonym for function declaration. The name, return type, and parameter types of a function. To call a function, its prototype must have been declared before the point of call.

函数声明的同义词。包括了函数的名字、返回类型和形参类型。调用函数时，必须在调用点之前声明函数原型。

inline function (内联函数)

Function that is expanded at the point of call, if possible. Inline functions avoid the normal function-calling overhead by replacing the call by the function's code.

如果可能的话，将在调用点展开的函数。内联函数直接以函数代码替代了函数调用点展开的函数。内联函数直接以函数代码替代了函数调用语句，从而避免了一般函数调用的开销。

local static objects (局部静态对象)

Local object that is created and initialized once before the function is first called and whose value persists across invocations of the function.

在函数第一次调用前就已经创建和初始化的局部对象，其值在函数的调用之间保持有效。

local variables (局部变量)

Variables defined inside a function. Local variables are accessible only within the function body.

在函数内定义的变量，仅能在函数体内访问。

object lifetime (对象生命期)

Every object has an associated lifetime. Objects that are defined inside a block exist from when their definition is encountered until the end of the block in which they are defined. Local static objects and global objects defined outside any function are created during program startup and are destroyed when the `main` function ends. Dynamically created objects that are created through a `new` expression exist until the memory in which they were created is freed through a corresponding `delete`.

每个对象皆有与之关联的生命期。在块中定义的对象从定义时开始存在，直到它的定义所在的语句块结束为止。静态局部对象和函数外定义的全局变量则在程序开始执行时创建，当 `main` 函数结束时撤销。动态创建的对象由 `new` 表达式创建，从此开始存在，直到由相应的 `delete` 表达式释放所占据的内存空间为止。

overload resolution (重载确定)

A synonym for function matching.

函数匹配的同义词。

[overloaded function](#) (重载函数)

A function that has the same name as at least one other function. Overloaded functions must differ in the number or type of their parameters.

和至少一个其他函数同名的函数。重载函数必须在形参的个数或类型上有所不同。

[parameters](#) (形参)

Variables local to a function whose initial values are supplied when the function is called.

函数的局部变量，其初值由函数调用提供。

[recursive function](#) (递归函数)

Function that calls itself directly or indirectly.

直接或间接调用自己的函数。

[return type](#) (返回类型)

The type of the value returned from a function.

函数返回值的类型。

[synthesized default constructor](#) (合成默认构造函数)

If there are no constructors defined by a class, then the compiler will create (synthesize) a default constructor. This constructor default initializes each data member of the class.

如果类没有定义任何构造函数，则编译器会为这个类创建（合成）一个默认构造函数。该函数以默认的方式初始化类中的所有数据成员。

[temporary object](#) (临时对象)

Unnamed object automatically created by the compiler in the course of evaluating an expression. The phrase *temporary object* is usually abbreviated as *temporary*. A temporary persists until the end of the largest expression that encloses the expression for which it was created.

在求解表达式的过程中由编译器自动创建的没有名字的对象。“临时对象”这个术语通常简称为“临时”。临时对象一直存在直到最大表达式结束为止，最大表达式指的是包含创建该临时对象的表达式的最大范围内的表达式。

[this pointer](#) (this 指针)

Implicit parameter of a member function. `this` points to the object on which the function is invoked. It is a pointer to the class type. In a `const` member function the pointer is a pointer to `const`.

成员函数的隐式形参。`this` 指针指向调用该函数的对象，是指向类类型的指针。在 `const` 成员函数中，该指针也指向 `const` 对象。

[viable functions](#) (可行函数)

The subset of overloaded functions that could match a given call. Viable functions have the same number of parameters as arguments to the call and each argument type can potentially be converted to the corresponding parameter type.

重载函数中可与指定的函数调用匹配的子集。可行函数的形参数个数必须与该函数调用的实参数个数相同，而且每个实参类型都可潜在地转换为相应形参的类型。

Chapter 8. The IO Library

第八章 标准 IO 库

CONTENTS

Section 8.1 An Object-Oriented Library	284
Section 8.2 Condition States	287
Section 8.3 Managing the Output Buffer	290
Section 8.4 File Input and Output	293
Section 8.5 String Streams	299
Chapter Summary	302
Defined Terms	302

In C++, input/output is provided through the library. The library defines a family of types that support IO to and from devices such as files and console windows. Additional types allow `strings` to act like files, which gives us a way to convert data to and from character forms without also doing IO. Each of these IO types defines how to read and write values of the built-in data types. In addition, class designers generally use the library IO facilities to read and write objects of the classes that they define. Class types are usually read and written using the same operators and conventions that the IO library defines for the built-in types.

C++ 的输入／输出 (input/output) 由标准库提供。标准库定义了一族类型，支持对文件和控制窗口等设备的读写 (IO)。还定义了其他一些类型，使 `string` 对象能够像文件一样操作，从而使我们无须 IO 就能实现数据与字符之间的转换。这些 IO 类型都定义了如何读写内置数据类型的值。此外，一般来说，类的设计者还可以很方便地使用 IO 标准库设施读写自定义类的对象。类类型通常使用 IO 标准库为内置类型定义的操作符和规则来进行读写。

This chapter introduces the fundamentals of the IO library. Later chapters will cover additional capabilities: [Chapter 14](#) will look at how we can write our own input and output operators; [Appendix A](#) will cover ways to control formatting and random access to files.

本章将介绍 IO 标准库的基础知识，而更多的内容会在后续章节中介绍：第十四章考虑如何编写自己的输入输出操作符：[附录 A](#) 则介绍格式控制以及文件的随机访问。

Our programs have already used many IO library facilities:

前面的程序已经使用了多种 IO 标准库提供的工具：

- `istream` (input stream) type, which supports input operations
`istream` (输入流) 类型，提供输入操作。
- `ostream` (output stream) type, which provides output operations
`ostream` (输出流) 类型，提供输出操作。
- `cin` (pronounced see-in) an `istream` object that reads the standard input.
`cin` (发音为 see-in) : 读入标准输入的 `istream` 对象。
- `cout` (pronounced see-out) an `ostream` object that writes to the standard output
`cout` (发音为 see-out) : 写到标准输出的 `ostream` 对象。
- `cerr` (pronounced see-err) an `ostream` object that writes to the standard error. `cerr` is usually used for program error messages.
`cerr` (发音为 see-err) : 输出标准错误的 `ostream` 对象。`cerr` 常用于程序错误信息。
- operator `>>`, which is used to read input from an `istream` object
`>>` 操作符，用于从 `istream` 对象中读入输入。
- operator `<<`, which is used to write output to an `ostream` object
`<<` 操作符，用于把输出写到 `ostream` 对象中。

- `getline` function, which takes a reference to an `istream` and a reference to a `string` and reads a word from the `istream` into the `string`
`getline` 函数，需要分别取 `istream` 类型和 `string` 类型的两个引用形参，其功能是从 `istream` 对象读取一个单词，然后写入 `string` 对象中。

This chapter looks briefly at some additional IO operations, and discusses support for reading and writing files and `strings`. [Appendix A](#) covers how to control formatting of IO operations, support for random access to files, and support for unformatted IO. This primer does not describe the entire `iostream` library in particular, we do not cover the system-specific implementation details, nor do we discuss the mechanisms by which the library manages input and output buffers or how we might write our own buffer classes. These topics are beyond the scope of this book. Instead, we'll focus on those portions of the IO library that are most useful in ordinary programs.

本章简要地介绍一些附加的 IO 操作，并讨论文件对象和 `string` 对象的读写。[附录 A](#) 会介绍如何控制 IO 操作的格式、文件的随机访问以及无格式的 IO。本书是初级读本，因此不会详细讨论完整的 `iostream` 标准库——特别是，我们不但没有涉及系统特定的实现细则，也不讨论标准库管理输入输出缓冲区的机制，以及如何编写自定义的缓冲区类。这些话题已超出了本书的范畴。相对而言，本书把重点放在 IO 标准库对普通程序最有用的部分。

8.1. An Object-Oriented Library

8.1. 面向对象的标准库

The IO types and objects we've used so far read and write streams of data and are used to interact with a user's console window. Of course, real programs cannot be limited to doing IO solely to or from a console window. Programs often need to read or write named files. Moreover, it can be quite convenient to use the IO operations to format data in memory, thereby avoiding the complexity and run-time expense of reading or writing to a disk or other device. Applications also may have to read and write languages that require wide-character support.

迄今为止，我们已经使用 IO 类型和对象读写数据流，它们常用于与用户控制窗口的交互。当然，实际的程序不能仅限于对控制窗口的 IO，通常还需要读或写已命名的文件。此外，程序还应该能方便地使用 IO 操作格式化内存中的数据，从而避免读写磁盘或其他设备的复杂性和运行代价。应用程序还需要支持宽字符（wide-character）语言的读写。

Conceptually, neither the kind of device nor the character size affect the IO operations we want to perform. For example, we'd like to use `>>` to read data regardless of whether we're reading a console window, a disk file, or an in-memory string. Similarly, we'd like to use that operator regardless of whether the characters we read fit in a `char` or require the `wchar_t` (Section 2.1.1, p. 34) type.

从概念上看，无论是设备的类型还是字符的大小，都不影响需要执行的 IO 操作。例如，不管我们是从控制窗口、磁盘文件或内存中的字符串读入数据，都可使用 `>>` 操作符。相似地，无论我们读的是 `char` 类型的字符还是 `wchar_t` (第 2.1.1 节) 的字符，也都可以使用该操作符。

At first glance, the complexities involved in supporting or using these different kinds of devices and different sized character streams might seem a daunting problem. To manage the complexity, the library uses **inheritance** to define a set of **object-oriented** classes. We'll have more to say about inheritance and object-oriented programming in Part IV, but generally speaking, types related by inheritance share a common interface. When one class inherits from another, we (usually) can use the same operations on both classes. More specifically, when two types are related by inheritance, we say that one class "inherits" the behavior of its parent. In C++ we speak of the parent as the **base class** and the inheriting class as a **derived class**.

乍看起来，要同时支持或使用不同类型设备以及不同大小的字符流，其复杂程度似乎相当可怕。为了管理这样的复杂性，标准库使用了**继承 (inheritance)** 来定义一组**面向对象 (object-oriented)** 类。在本书的**第四部分**将会更详细地讨论继承和面向对象程序设计，不过，一般而言，通过继承关联起来的类型都共享共同的接口。当一个类继承另一个类时，这两个类通常可以使用相同的操作。更确切地说，如果两种类型存在继承关系，则可以说一个类“继承”了其父类的行为——接口。C++ 中所提及的父类称为**基类 (base class)**，而继承而来的类则称为**派生类 (derived class)**。

The IO types are defined in three separate headers: `iostream` defines the types used to read and write to a console window, `fstream` defines the types used to read and write named files, and `sstream` defines the types used to read and write in-memory strings. Each of the types in `fstream` and `sstream` is derived from a corresponding type defined in the `iostream` header. Table 8.1 lists the IO classes and Figure 8.1 on the next page illustrates the inheritance relationships among these types. Inheritance is usually illustrated similarly to how a family tree is displayed. The topmost circle represents a base (or parent) class. Lines connect a base class to its derived (or children) class(es). So, for example, this figure indicates that `istream` is the base class of `ifstream` and `istringstream`. It is also the base class for `iostream`, which in turn is the base class for `sstream` and `fstream` classes.

IO 类型在三个独立的头文件中定义：`iostream` 定义读写控制窗口的类型，`fstream` 定义读写已命名文件的类型，而 `sstream` 所定义的类型则用于读写存储在内存中的 `string` 对象。在 `fstream` 和 `sstream` 里定义的每种类型都是从 `iostream` 头文件中定义的相关类型派生而来。表 8.1 列出了 C++ 的 IO 类，而图 8.1 则阐明这些类型之间的继承关系。继承关系通常可以用类似于家庭树的图解说明。最顶端的圆圈代表基类（或称“父类”），基类和派生类（或称“子类”）之间用线段连接。因此，图 8.1 所示，`istream` 是 `ifstream` 和 `istringstream` 的基类，同时也是 `iostream` 的基类，而 `iostream` 则是 `sstream` 和 `fstream` 的基类。

Table 8.1. IO Library Types and Headers

表 8.1. IO 标准库类型和头文件

Header	Type
<code>iostream</code>	<code>istream</code> reads from a stream <code>istream</code> 从流中读取
	<code>ostream</code> writes to a stream <code>ostream</code> 写到流中去
	<code>iostream</code> reads and writes a stream; derived from <code>istream</code> and <code>ostream</code> , <code>iostream</code> 对流进行读写；从 <code>istream</code> 和 <code>ostream</code> 派生而来
<code>fstream</code>	<code>ifstream</code> , reads from a file; derived from <code>iostream</code> <code>ifstream</code> 从文件中读取；由 <code>iostream</code> 派生而来

```

ofstream writes to a file; derived from ostream
ofstream 写到文件中去；由 ostream 派生而来

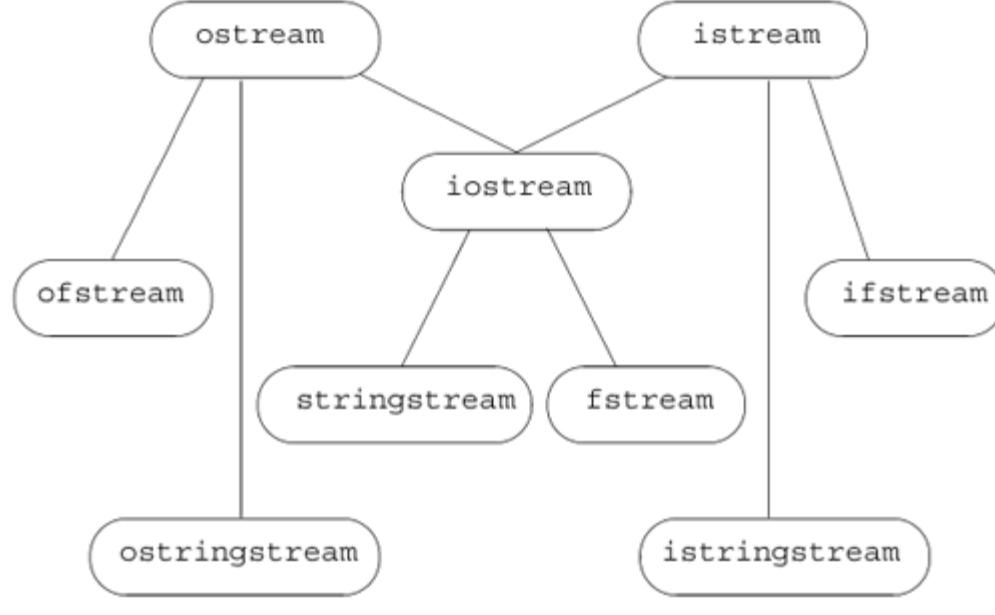
fstream, reads and writes a file; derived from iostream
fstream 读写文件；由 iostream 派生而来

sstream istringstream reads from a string; derived from istream
istringstream 从 string 对象中读取；由 istream 派生而来

ostringstream writes to a string; derived from ostream
ostringstream 写到 string 对象中去；由 ostream 派生而来

stringstream reads and writes a string; derived from iostream
stringstream 对 string 对象进行读写；由 iostream 派生而来

```

Figure 8.1. Simplified `iostream` Inheritance Hierarchy图 8.1. 简单的 `iostream` 继承层次

Because the types `ifstream` and `istringstream` inherit from `istream`, we already know a great deal about how to use these types. Each program we've written that read an `istream` could be used to read a file (using the `ifstream` type) or a `string` (using the `istringstream` type). Similarly, programs that did output could use an `ofstream` or `ostringstream` instead of `ostream`. In addition to the `istream` and `ostream` types, the `iostream` header also defines the `iostream` type. Although our programs have not used this type, we actually know a good bit about how to use an `iostream`. The `iostream` type is derived from both `istream` and `ostream`. Being derived from both types means that an `iostream` object shares the interface of both its parent types. That is, we can use an `iostream` type to do both input and output to the same stream. The library also defines two types that inherit from `iostream`. These types can be used to read or write to a file or a `string`.

由于 `ifstream` 和 `istringstream` 类型继承了 `istream` 类，因此已知这两种类型的大量用法。我们曾经编写过的读 `istream` 对象的程序也可用于读文件（使用 `ifstream` 类型）或者 `string` 对象（使用 `istringstream` 类型）。类似地，提供输出功能的程序同样可用 `ofstream` 或 `ostringstream` 取代 `ostream` 类型实现。除了 `istream` 和 `ostream` 类型之外，`iostream` 头文件还定义了 `iostream` 类型。尽管我们的程序还没用过这种类型，但事实上可以多了解一些关于 `iostream` 的用法。`iostream` 类型由 `istream` 和 `ostream` 两者派生而来。这意味着 `iostream` 对象共享了它的两个父类的接口。也就是说，可使用 `iostream` 类型在同一个流上实现输入和输出操作。标准库还定义了另外两个继承 `iostream` 的类型。这些类型可用于读写文件或 `string` 对象。

Using inheritance for the IO types has another important implication: As we'll see in [Chapter 15](#), when we have a function that takes a reference to a base-class type, we can pass an object of a derived type to that function. This fact means that a function written to operate on `istream&` can be called with an `ifstream` or `istringstream` object. Similarly, a function that takes an `ostream&` can be called with an `ofstream` or `ostringstream` object. Because the IO types are related by inheritance, we can write one function and apply it to all three kinds of streams: console, disk files, or [string streams](#).

对 IO 类型使用继承还有另外一个重要的含义：正如在[第十五章](#)可以看到的，如果函数有基类类型的引用形参时，可以给函数传递其派生类型的对象。这就意味着：对

Section 8.1. An Object-Oriented Library

`istream&` 进行操作的函数，也可使用 `ifstream` 或者 `istringstream` 对象来调用。类似地，形参为 `ostream&` 类型的函数也可用 `ofstream` 或者 `ostringstream` 对象调用。因为 IO 类型通过继承关联，所以可以只编写一个函数，而将它应用到三种类型的流上：控制台、磁盘文件或者字符串流 (string streams)。

International Character Support

国际字符的支持

The stream classes described thus far read and write streams composed of type `char`. The library defines a corresponding set of types supporting the `wchar_t` type. Each class is distinguished from its `char` counterpart by a "`w`" prefix. Thus, the types `wostream`, `wistream`, and `wiostream` read and write `wchar_t` data to or from a console window. The file input and output classes are `wifstream`, `wofstream`, and `wfstream`. The `wchar_t` versions of `string` stream input and output are `wistringstream`, `wostream`, and `wstringstream`. The library also defines objects to read and write wide characters from the standard input and standard output. These objects are distinguished from the `char` counterparts by a "`w`" prefix: The `wchar_t` standard input object is named `wcin`; standard output is `wcout`; and standard error is `wcerr`.

迄今为止，所描述的流类 (stream class) 读写的是由 `char` 类型组成的流。此外，标准库还定义了一组相关的类型，支持 `wchar_t` 类型。每个类都加上 "`w`" 前缀，以此与 `char` 类型的版本区分开来。于是，`wostream`、`wistream` 和 `wiostream` 类型从控制窗口读写 `wchar_t` 数据。相应的文件输入输出类是 `wifstream`、`wofstream` 和 `wfstream`。而 `wchar_t` 版本的 `string` 输入／输出流则是 `wistringstream`、`wostream` 和 `wstringstream`。标准库还定义了从标准输入输出读写宽字符的对象。这些对象加上 "`w`" 前缀，以此与 `char` 类型版本区分：`wchar_t` 类型的标准输入对象是 `wcin`；标准输出是 `wcout`；而标准错误则是 `wcerr`。

Each of the IO headers defines both the `char` and `wchar_t` classes and standard input/output objects. The stream-based `wchar_t` classes and objects are defined in `iostream`, the wide character file stream types in `fstream`, and the wide character `stringstreams` in `sstream`.

每一个 IO 头文件都定义了 `char` 和 `wchar_t` 类型的类和标准输入／输出对象。基于流的 `wchar_t` 类型的类和对象在 `iostream` 中定义，宽字符文件流类型在 `fstream` 中定义，而宽字符 `stringstream` 则在 `sstream` 头文件中定义。

No Copy or Assign for IO Objects

IO 对象不可复制或赋值

For reasons that will be more apparent when we study classes and inheritance in [Parts III](#) and [IV](#), the library types do not allow copy or assignment:

出于某些原因，标准库类型不允许做复制或赋值操作。其原因将在后面[第三部分](#)和[第四部分](#)学习类和继承时阐明。

```
ofstream out1, out2;
out1 = out2; // error: cannot assign stream objects
// print function: parameter is copied
ofstream print(ofstream);
out2 = print(out2); // error: cannot copy stream objects
```

This requirement has two particularly important implications. As we'll see in [Chapter 9](#), only element types that support copy can be stored in `vectors` or other container types. Because we cannot copy stream objects, we cannot have a `vector` (or other container) that holds stream objects.

这个要求有两层特别重要的含义。正如在[第九章](#)看到的，只有支持复制的元素类型可以存储在 `vector` 或其他容器类型里。由于流对象不能复制，因此不能存储在 `vector` (或其他) 容器中 (即不存在存储流对象的 `vector` 或其他容器)。

The second implication is that we cannot have a parameter or return type that is one of the stream types. If we need to pass or return an IO object, it must be passed or returned as a pointer or reference:

第二个含义是：形参或返回类型也不能为流类型。如果需要传递或返回 IO 对象，则必须传递或返回指向该对象的指针或引用：

```
ofstream &print(ofstream); // ok: takes a reference, no copy
while (print(out2)) { /* ... */ } // ok: pass reference to out2
```

Typically, we pass a stream as a non`const` reference because we pass an IO object intending to read from it or write to it. Reading or writing an IO object changes its state, so the reference must be non`const`.

一般情况下，如果要传递 IO 对象以便对它进行读写，可用非 `const` 引用的方式传递这个流对象。对 IO 对象的读写会改变它的状态，因此引用必须是非 `const` 的。

Exercises Section 8.1

Exercise Assuming `os` is an `ofstream`, what does the following program do?

8.1:

假设 `os` 是一个 `ofstream` 对象，下面程序做了什么？

```
os << "Goodbye!" << endl;
```

What if `os` is an `ostringstream`? What if `os` is an `ifstream`?

如果 `os` 是 `ostringstream` 对象呢？或者，`os` 是 `ifstream` 呢？

Exercise The following declaration is in error. Identify and correct the problem(s):

8.2:

下面的声明是错误的，指出其错误并改正之：

```
ostream print(ostream os);
```

8.2. Condition States

8.2. 条件状态

Before we explore the types defined in `fstream` and `sstream`, we need to understand a bit more about how the IO library manages its buffers and the state of a stream. Keep in mind that the material we cover in this section and the next applies equally to plain streams, file streams, or `string` streams.

在展开讨论 `fstream` 和 `sstream` 头文件中定义的类型之前，需要了解更多 IO 标准库如何管理其缓冲区及其流状态的相关内容。谨记本节和下一节所介绍的内容同样适用于普通流、文件流以及 `string` 流。

Inherent in doing IO is the fact that errors can occur. Some errors are recoverable; others occur deep within the system and are beyond the scope of a program to correct. The IO library manages a set of `condition state` members that indicate whether a given IO object is in a usable state or has encountered a particular kind of error. The library also defines a set of functions and flags, listed in [Table 8.2](#), that give us access to and let us manipulate the state of each stream.

实现 IO 的继承正是错误发生的根源。一些错误是可恢复的；一些错误则发生在系统底层，位于程序可修正的范围之外。IO 标准库管理一系列条件状态 (`condition state`) 成员，用来标记给定的 IO 对象是否处于可用状态，或者碰到了哪种特定的错误。[表 8.2](#) 列出了标准库定义的一组函数和标记，提供访问和操纵流状态的手段。

Table 8.2. IO Library Condition State

表 8.2. IO 标准库的条件状态

<code>strm::iostate</code>	Name of the machine-dependent integral type, defined by each <code>iostream</code> class that is used to define the condition states. 机器相关的整型名，由各个 <code>iostream</code> 类定义，用于定义条件状态
<code>strm::badbit</code>	<code>strm::iostate</code> value used to indicate that a stream is corrupted. <code>strm::iostate</code> 类型的值，用于指出被破坏的流
<code>strm::failbit</code>	<code>strm::iostate</code> value used to indicate that an IO operation failed. <code>strm::iostate</code> 类型的值，用于指出失败的 IO 操作
<code>strm::eofbit</code>	<code>strm::iostate</code> value used to indicate the a stream hit end-of-file. <code>strm::iostate</code> 类型的值，用于指出流已经到达文件结束符
<code>s.eof()</code>	<code>true</code> if <code>eofbit</code> in the stream <code>s</code> is set. 如果设置了流 <code>s</code> 的 <code>eofbit</code> 值，则该函数返回 <code>true</code>
<code>s.fail()</code>	<code>true</code> if <code>failbit</code> in the stream <code>s</code> is set. 如果设置了流 <code>s</code> 的 <code>failbit</code> 值，则该函数返回 <code>true</code>
<code>s.bad()</code>	<code>true</code> if <code>badbit</code> in the stream <code>s</code> is set. 如果设置了流 <code>s</code> 的 <code>badbit</code> 值，则该函数返回 <code>true</code>
<code>s.good()</code>	<code>true</code> if the stream <code>s</code> is in a valid state. 如果流 <code>s</code> 处于有效状态，则该函数返回 <code>true</code>
<code>s.clear()</code>	Reset all condition values in the stream <code>s</code> to valid state. 将流 <code>s</code> 中的所有状态值都重设为有效状态
<code>s.clear(flag)</code>	Set specified condition state(s) in <code>s</code> to valid. Type of <code>flag</code> is <code>strm::iostate</code> . 将流 <code>s</code> 中的某个指定条件状态设置为有效。 <code>flag</code> 的类型是 <code>strm::iostate</code>
<code>s.setstate(flag)</code>	Add specified condition to <code>s</code> . Type of <code>flag</code> is <code>strm::iostate</code> .

Section 8.2. Condition States

给流 <code>s</code> 添加指定条件。 <code>flag</code> 的类型是 <code>strm::iostate</code>
<code>s.rdstate()</code>
Returns current condition of <code>s</code> as an <code>strm::iostate</code> value.
返回流 <code>s</code> 的当前条件，返回值类型为 <code>strm::iostate</code>

As an example of an IO error, consider the following code:

考虑下面 IO 错误的例子：

```
int ival;  
cin >> ival;
```

If we enter `Borges` on the standard input, then `cin` will be put in an error state following the unsuccessful attempt to read a string of characters as an `int`. Similarly, `cin` will be in an error state if we enter an end-of-file. Had we entered 1024, then the read would be successful and `cin` would be in a good, non-error state.

如果在标准输入设备输入 `Borges`，则 `cin` 在尝试将输入的字符串读为 `int` 型数据失败后，会生成一个错误状态。类似地，如果输入文件结束符（end-of-file），`cin` 也会进入错误状态。而如果输入 1024，则成功读取，`cin` 将处于正确的无错误状态。

To be used for input or output, a stream must be in a non-error state. The easiest way to test whether a stream is okay is to test its truth value:

流必须处于无错误状态，才能用于输入或输出。检测流是否用的最简单的方法是检查其真值：

```
if (cin)  
    // ok to use cin, it is in a valid state  
  
while (cin >> word)  
    // ok: read operation successful ...
```

The `if` directly tests the state of the stream. The `while` does so indirectly by testing the stream returned from the expression in the condition. If that input operation succeeds, then the condition tests `true`.

`if` 语句直接检查流的状态，而 `while` 语句则检测条件表达式返回的流，从而间接地检查了流的状态。如果成功输入，则条件检测为 `true`。

Condition States

条件状态

Many programs need only know whether a stream is valid. Other programs need more fine-grained access to and control of the state of the stream. Rather than knowing that the stream is in an error state, we might want to know what kind of error was encountered. For example, we might want to distinguish between reaching end-of-file and encountering an error on the IO device.

许多程序只需知道是否有效。而某些程序则需要更详细地访问或控制流的状态，此时，除了知道流处于错误状态外，还必须了解它遇到了哪种类型的错误。例如，程序员也许希望弄清是到达了文件的结尾，还是遇到了 IO 设备上的错误。

Each stream object contains a condition state member that is managed through the `setstate` and `clear` operations. This state member has type `iostate`, which is a machine-dependent integral type defined by each `iostream` class. It is used as a collection of bits, much the way we used the `int_quiz1` variable to represent test scores in the example in [Section 5.3.1](#) (p. 156).

所有流对象都包含一个条件状态成员，该成员由 `setstate` 和 `clear` 操作管理。这个状态成员为 `iostate` 类型，这是由各个 `iostream` 类分别定义的机器相关的整型。该状态成员以二进制位（bit）的形式使用，类似于第 5.3.1 节的例子中用于记录测验成绩的 `int_quiz1` 变量。

Each IO class also defines three `const` values of type `iostate` that represent particular bit patterns. These `const` values are used to indicate particular kinds of IO conditions. They can be used with the bitwise operators ([Section 5.3](#), p. 154) to test or set multiple flags in one operation.

每个 IO 类还定义了三个 `iostate` 类型的常量值，分别表示特定的位模式。这些常量值用于指出特定类型的 IO 条件，可与位操作符（第 5.3 节）一起使用，以便在一次操作中检查或设置多个标志。

The `badbit` indicates a system level failure, such as an unrecoverable read or write error. It is usually not possible to continue using a stream after such an error. The `failbit` is set after a recoverable error, such as reading a character when numeric data was expected. It is often possible to correct the problem that caused the `failbit` to be set. The `eofbit` is set when an end-of-file is encountered. Hitting end-of-file also sets the `failbit`.

`badbit` 标志着系统级的故障，如无法恢复的读写错误。如果出现了这类错误，则该流通常就不能再继续使用了。如果出现的是可恢复的错误，如在希望获得数值型数据时输入了字符，此时则设置 `failbit` 标志，这种导致设置 `failbit` 的问题通常是可以修正的。`eofbit` 是在遇到文件结束符时设置的，此时同时还设置了 `failbit`。

The state of the stream is revealed by the `bad`, `fail`, `eof`, and `good` operations. If any of `bad`, `fail`, or `eof` are `true`, then testing the stream itself will indicate that the stream is in an error state. Similarly, the `good` operation returns `TRUE` if none of the other conditions is `true`.

Section 8.2. Condition States

流的状态由 `bad`、`fail`、`eof` 和 `good` 操作提示。如果 `bad`、`fail` 或者 `eof` 中的任意一个为 `true`，则检查流本身将显示该流处于错误状态。类似地，如果这三个条件没有一个为 `true`，则 `good` 操作将返回 `true`。

The `clear` and `setstate` operations change the state of the condition member. The `clear` operations put the condition back in its valid state. They are called after we have remedied whatever problem occurred and we want to reset the stream to its valid state. The `setstate` operation turns on the specified condition to indicate that a problem occurred. `setstate` leaves the existing state variables unchanged except that it adds the additional indicated state(s).

`clear` 和 `setstate` 操作用于改变条件成员的状态。`clear` 操作将条件重设为有效状态。在流的使用出现了问题并做出补救后，如果我们希望把流重设为有效状态，则可以调用 `clear` 操作。使用 `setstate` 操作可打开某个指定的条件，用于表示某个问题的发生。除了添加的标记状态，`setstate` 将保留其他已存在的状态变量不变。

Interrogating and Controlling the State of a Stream

流状态的查询和控制

We might manage an input operation as follows:

可以如下管理输入操作

```
int ival;
// read cin and test only for EOF; loop is executed even if there are other IO failures
while (cin >> ival, !cin.eof()) {
    if (cin.bad())           // input stream is corrupted; bail out
        throw runtime_error("IO stream corrupted");
    if (cin.fail()) {          // bad input
        cerr << "bad data, try again"; // warn the user
        cin.clear(iostream::failbit); // reset the stream
        continue;               // get next input
    }
    // ok to process ival
}
```

This loop reads `cin` until end-of-file or an unrecoverable read error occurs. The condition uses a comma operator ([Section 5.9](#), p. 168). Recall that the comma operator executes by evaluating each operand and returns its rightmost operand as its result. The condition, therefore, reads `cin` and ignores its result. The result of the condition is the result of `!cin.eof()`. If `cin` hit end-of-file, the condition is false and we fall out of the loop. If `cin` did not hit end-of-file, we enter the loop, regardless of any other error the read might have encountered.

这个循环不断读入 `cin`，直到到达文件结束符或者发生不可恢复的读取错误为止。循环条件使用了逗号操作符（[第 5.9 节](#)）。回顾逗号操作符的求解过程：首先计算它的每一个操作数，然后返回最右边操作数作为整个操作的结果。因此，循环条件只读入 `cin` 而忽略了其结果。该条件的结果是 `!cin.eof()` 的值。如果 `cin` 到达文件结束符，条件则为假，退出循环。如果 `cin` 没有到达文件结束符，则不管在读取时是否发生了其他可能遇到的错误，都进入循环。

Inside the loop, we first check whether the stream is corrupted. If so, we exit by throwing an exception ([Section 6.13](#), p. 215). If the input was invalid, we print a warning, and clear the `failbit` state. In this case, we execute a `continue` ([Section 6.11](#), p. 214) to return to the start of the `while` to read another value into `ival`. If there were no errors, the rest of the loop can safely use `ival`.

在循环中，首先检查流是否已破坏。如果是的放，抛出 `异常` 并退出循环。如果输入无效，则输出警告并清除 `failbit` 状态。在本例中，执行 `continue` 语句（[第 6.11 节](#)）回到 `while` 的开头，读入另一个值 `ival`。如果没有出现任何错误，那么循环体中余下的部分则可以很安全地使用 `ival`。

Accessing the Condition State

条件状态的访问

The `rdstate` member function returns an `iostate` value that corresponds to the entire current condition state of the stream:

`rdstate` 成员函数返回一个 `iostate` 类型值，该值对应于流当前的整个条件状态：

```
// remember current state of cin
istream::iostate old_state = cin.rdstate();
cin.clear();
process_input(); // use cin
cin.clear(old_state); // now reset cin to old state
```

Dealing with Multiple States

多种状态的处理

Often we need to set or clear multiple state bits. We could do so by making multiple calls to the `setstate` or `clear` functions. Alternatively, we could use the bitwise OR ([Section 5.3](#), p. 154) operator to generate a value to pass two or more state bits in a single call. The bitwise OR generates an integral value using the bit patterns of its operands. For each bit in the result, the bit is 1 if the corresponding bit is 1 in either of its operands. For example:

常常会出现需要设置或清除多个状态二进制位的情况。此时，可以通过多次调用 `setstate` 或者 `clear` 函数实现。另外一种方法则是使用按位或 (OR) 操作符 ([第 5.3 节](#)) 在一次调用中生成“传递两个或更多状态位”的值。按位或操作使用其操作数的二进制位模式产生一个整型数值。对于结果中的每一个二进制位，如果其值为 1，则该操作的两个操作数中至少有一个的对应二进制位是 1。例如：

```
// sets both the badbit and the failbit
is.setstate(ifstream::badbit | ifstream::failbit);
```

tells the object `is` to turn on both the `failbit` and the `badbit`. The argument

将对象 `is` 的 `failbit` 和 `badbit` 位同时打开。实参：

```
is.badbit | is.failbit
```

creates a value in which the bits corresponding to the `badbit` and to the `failbit` are both turned on that is they are both set to 1. All other bits in the value are zero. The call to `setstate` uses this value to turn on the bits corresponding to `badbit` and `failbit` in the stream's condition state member.

生成了一个值，其对应于 `badbit` 和 `failbit` 的位都打开了，也就是将这两个位都设置为 1，该值的其他位则都为 0。在调用 `setstate` 时，使用这个值来开启流条件状态成员中对应的 `badbit` 和 `failbit` 位。

Exercises Section 8.2

Exercise 8.3: Write a function that takes and returns an `istream&`. The function should read the stream until it hits end-of-file. The function should print what it reads to the standard output. Reset the stream so that it is valid and return the stream.

编写一个函数，其唯一的形参和返回值都是 `istream&` 类型。该函数应一直读取流直到到达文件结束符为止，还应将读到的内容输出到标准输出中。最后，重设流使其有效，并返回该流。

Exercise 8.4: Test your function by calling it passing `cin` as an argument.

通过以 `cin` 为实参实现调用来测试上题编写的函数。

Exercise 8.5: What causes the following `while` to terminate?

导致下面的 `while` 终止的原因是什么？

```
while (cin >> i) /* . . . */
```

8.3. Managing the Output Buffer

8.3. 输出缓冲区的管理

Each IO object manages a buffer, which is used to hold the data that the program reads and writes. When we write

每个 IO 对象管理一个缓冲区，用于存储程序读写的数据。如有下面语句：

```
os << "please enter a value: ";
```

the literal string is stored in the buffer associated with the stream `os`. There are several conditions that cause the buffer to be flushed that is, written to the actual output device or file:

系统将字符串字面值存储在与流 `os` 关联的缓冲区中。下面几种情况将导致缓冲区的内容被刷新，即写入到真实的输出设备或者文件：

1. The program completes normally. All output buffers are emptied as part of the `return` from `main`.

程序正常结束。作为 `main` 返回工作的一部分，将清空所有输出缓冲区。

2. At some indeterminate time, the buffer can become full, in which case it will be flushed before writing the next value.

在一些不确定的时候，缓冲区可能已经满了，在这种情况下，缓冲区将会在写下一个值之前刷新。

3. We can flush the buffer explicitly using a manipulator ([Section 1.2.2](#), p. 7) such as `endl`.

用操纵符（[第 1.2.2 节](#)）显式地刷新缓冲区，例如行结束符 `endl`。

4. We can use the `unitbuf` manipulator to set the stream's internal state to empty the buffer after each output operation.

在每次输出操作执行完后，用 `unitbuf` 操纵符设置流的内部状态，从而清空缓冲区。

5. We can `tie` the output stream to an input stream, in which case the output buffer is flushed whenever the associated input stream is read.

可将输出流与输入流关联 (`tie`) 起来。在这种情况下，在读输入流时将刷新其关联的输出缓冲区。

Flushing the Output Buffer

输出缓冲区的刷新

Our programs have already used the `endl` manipulator, which writes a newline and flushes the buffer. There are two other similar manipulators. The first, `flush`, is used quite frequently. It flushes the stream but adds no characters to the output. The second, `ends`, is used much less often. It inserts a null character into the buffer and then flushes it:

我们的程序已经使用过 `endl` 操纵符，用于输出一个换行符并刷新缓冲区。除此之外，C++ 语言还提供了另外两个类似的操纵符。第一个经常使用的 `flush`，用于刷新流，但不在输出中添加任何字符。第二个则是比较少用的 `ends`，这个操纵符在缓冲区中插入空字符 `null`，然后刷新它：

```
cout << "hi!" << flush;      // flushes the buffer; adds no data
cout << "hi!" << ends;       // inserts a null, then flushes the buffer
cout << "hi!" << endl;        // inserts a newline, then flushes the buffer
```

The `unitbuf` Manipulator

`unitbuf` 操纵符

If we want to flush every output, it is better to use the `unitbuf` manipulator. This manipulator flushes the stream after every write:

如果需要刷新所有输出，最好使用 `unitbuf` 操纵符。这个操纵符在每次执行完写操作后都刷新流：

```
cout << unitbuf << "first" << " second" << nounitbuf;
```

is equivalent to writing

等价于：

```
cout << "first" << flush << "second" << flush;
```

The `nounitbuf` manipulator restores the stream to use normal, system-managed buffer flushing.

`nounitbuf` 操纵符将流恢复为使用正常的、由系统管理的缓冲区刷新方式。

Caution: Buffers Are Not Flushed if the Program Crashes

警告：如果程序崩溃了，则不会刷新缓冲区

Output buffers are *not flushed* if the program terminates abnormally. When attempting to debug a program that has crashed, we often use the last output to help isolate the region of program in which the bug might occur. If the crash is after a particular print statement, then we know that the crash happened after that point in the program.

如果程序不正常结束，输出缓冲区将不会刷新。在尝试调试已崩溃的程序时，通常会根据最后的输出找出程序发生错误的区域。如果崩溃出现在某个特定的输出语句后面，则可知是在程序的这个位置之后出错。

When debugging a program, it is essential to make sure that any output you *think* should have been written was actually flushed. Because the system does not automatically flush the buffers when the program crashes, it is likely that there is output that the program wrote but that has not shown up on the standard output. It is still sitting in an output buffer waiting to be printed.

调试程序时，必须保证期待写入的每个输出都确实被刷新了。因为系统不会在程序崩溃时自动刷新缓冲区，这就可能出现这样的情况：程序做了写输出的工作，但写的内容并没有显示在标准输出上，仍然存储在输出缓冲区中等待输出。

If you use the last output to help locate the bug, you need to be certain that all the output really did get printed. Making sure that all output operations include an explicit `flush` or call to `endl` is the best way to ensure that you are seeing all the output that the program actually processed.

如果需要使用最后的输出给程序错误定位，则必须确定所有要输出的都已经输出。为了确保用户看到程序实际上处理的所有输出，最好的方法是保证所有的输出操作都显式地调用了 `flush` 或 `endl`。

Countless hours of programmer time have been wasted tracking through code that appeared not to have executed when in fact the buffer simply had not been flushed. For this reason, we tend to use `endl` rather than `\n` when writing output. Using `endl` means we do not have to wonder whether output is pending when a program crashes.

如果仅因为缓冲区没有刷新，程序员将浪费大量的时间跟踪调试并没有执行的代码。基于这个原因，输出时应多使用 `endl` 而非 '`\n`'。使用 `endl` 则不必担心程序崩溃时输出是否悬而未决（即还留在缓冲区，未输出到设备中）。

Tying Input and Output Streams Together

将输入和输出绑在一起

When an input stream is tied to an output stream, any attempt to read the input stream will first flush the buffer associated output stream. The library ties `cout` to `cin`, so the statement

当输入流与输出流绑在一起时，任何读输入流的尝试都将首先刷新其输出流关联的缓冲区。标准库将 `cout` 与 `cin` 绑在一起，因此语句：

```
cin >> ival;
```

causes the buffer associated with `cout` to be flushed.

导致 `cout` 关联的缓冲区被刷新。



Interactive systems usually should be sure that their input and output streams are tied. Doing so means that we are guaranteed that any output, which might include prompts to the user, has been written before attempting to read.

交互式系统通常应确保它们的输入和输出流是绑在一起的。这样做意味着可以保证任何输出，包括给用户的提示，都在试图读之前

Section 8.3. Managing the Output Buffer

输出。

The `tie` function can be called on either `istream` or an `ostream`. It takes a pointer to an `ostream` and ties the argument stream to the object on which `tie` was called. When a stream ties itself to an `ostream`, then any IO operation on the stream that called `tie` flushes the buffer associated with the argument it passed to `tie`.

`tie` 函数可用 `istream` 或 `ostream` 对象调用，使用一个指向 `ostream` 对象的指针形参。调用 `tie` 函数时，将实参流绑在调用该函数的对象上。如果一个流调用 `tie` 函数将其本身绑在传递给 `tie` 的 `ostream` 实参对象上，则该流上的任何 IO 操作都会刷新实参所关联的缓冲区。

```
cin.tie(&cout);    // illustration only: the library ties cin and cout for us
ostream *old_tie = cin.tie();
cin.tie(0); // break tie to cout, cout no longer flushed when cin is read
cin.tie(&cerr); // ties cin and cerr, not necessarily a good idea!
// ...
cin.tie(0);      // break tie between cin and cerr
cin.tie(old_tie); // reestablish normal tie between cin and cout
```

An `ostream` object can be tied to only one `istream` object at a time. To break an existing tie, we pass in an argument of 0.

一个 `ostream` 对象每次只能与一个 `istream` 对象绑在一起。如果在调用 `tie` 函数时传递实参 0，则打破该流上已存在的捆绑。

8.4. File Input and Output

8.4. 文件的输入和输出

The `fstream` header defines three types to support file IO:

`fstream` 头文件定义了三种支持文件 IO 的类型：

1. `ifstream`, derived from `istream`, reads from a file.
`ifstream`, 由 `istream` 派生而来, 提供读文件的功能。
2. `ofstream`, derived from `ostream`, writes to a file.
`ofstream`, 由 `ostream` 派生而来, 提供写文件的功能。
3. `fstream`, derived from `iostream`, reads and writes the same file.
`fstream`, 由 `iostream` 派生而来, 提供读写同一个文件的功能。

The fact that these types are derived from the corresponding `iostream` types means that we already know most of what we need to know about how to use the `fstream` types. In particular, we can use the IO operators (`<<` and `>>`) to do formatted IO on a file, and the material covered in the previous sections on condition states apply identically to `fstream` objects.

这些类型都由相应的 `iostream` 类型派生而来, 这个事实意味着我们已经知道使用 `fstream` 类型需要了解的大部分内容了。特别是, 可使用 IO 操作符 (`<<` 和 `>>`) 在文件上实现格式化的 IO, 而且在前面章节介绍的条件状态也同样适用于 `fstream` 对象。

In addition to the behavior that `fstream` types inherit, they also define two new operations of their own `open` and `close` along with a constructor that takes the name of a file to open. These operations can be called on objects of `fstream`, `ifstream`, or `ofstream` but not on the other IO types.

`fstream` 类型除了继承下来的行为外, 还定义了两个自己的新操作—— `open` 和 `close`, 以及形参为要打开的文件名的构造函数。`fstream`、`ifstream` 或 `ofstream` 对象可调用这些操作, 而其他的 IO 类型则不能调用。

8.4.1. Using File Stream Objects

8.4.1. 文件流对象的使用

So far our programs have used the library-defined objects, `cin`, `cout`, and `cerr`. When we want to read or write a file, we must define our own objects, and bind them to the desired files. Assuming that `infile` and `outfile` are `strings` with the names of the files we want to read and write, we might write code such as

迄今为止, 我们的程序已经使用过标准库定义的对象: `cin`、`cout` 和 `cerr`。需要读写文件时, 则必须定义自己的对象, 并将它们绑定在需要的文件上。假设 `infile` 和 `outfile` 是存储希望读写的文件名的 `strings` 对象, 可如下编写代码:

```
// construct an ifstream and bind it to the file named infile
ifstream infile(infile.c_str());
// ofstream output file object to write file named outfile
ofstream outfile(outfile.c_str());
```

to define and open a pair of `fstream` objects. `infile` is a stream that we can read and `outfile` is a stream that we can write. Supplying a file name as an initializer to an `ifstream` or `ofstream` object has the effect of opening the specified file.

上述代码定义并打开了一对 `fstream` 对象。`infile` 是读的流, 而 `outfile` 则是写的流。为 `ifstream` 或者 `ofstream` 对象提供文件名作为初始化式, 就相当于打开了特定的文件。

```
ifstream infile; // unbound input file stream
ofstream outfile; // unbound output file stream
```

These definitions define `infile` as a stream object that will read from a file and `outfile` as an object that we can use to write to a file. Neither object is as yet bound to a file. Before we use an `fstream` object, we must also bind it to a file to read or write:

上述语句将 `infile` 定义为读文件的流对象, 将 `outfile` 定义为写文件的对象。这两个对象都没有捆绑具体的文件。在使用 `fstream` 对象之前, 还必须使这些对象捆绑要读写

Section 8.4. File Input and Output

的文件:

```
infile.open("in"); // open file named "in" in the current directory  
outfile.open("out"); // open file named "out" in the current directory
```

We bind an existing `fstream` object to the specified file by calling the `open` member. The `open` function does whatever system-specific operations are required to locate the given file and open it for reading or writing as appropriate.

调用 `open` 成员函数将已存在的 `fstream` 对象与特定文件绑定。为了实现读写，需要将指定的文件打开并定位，`open` 函数完成系统指定的所有需要的操作。

Caution: File Names in C++

警告: C++ 中的文件名

For historical reasons, the IO library uses C-style character strings ([Section 4.3, p. 130](#)) rather than C++ `strings` to refer to file names. When we call `open` or use a file name as the initializer when creating an `fstream` object, the argument we pass is a C-style string, not a library `string`. Often our programs obtain file names by reading the standard input. As usual, it is a good idea to read into a `string`, not a C-style character array. Assuming that the name of the file we wish to use is in a `string`, we can use the `c_str` member ([Section 4.3.2, p. 139](#)) to obtain a C-style string.

由于历史原因，IO 标准库使用 C 风格字符串（[第 4.3 节](#)）而不是 C++ `strings` 类型的字符串作为文件名。在创建 `fstream` 对象时，如果调用 `open` 或使用文件名作初始化式，需要传递的实参应为 C 风格字符串，而不是标准库 `strings` 对象。程序常常从标准输入获得文件名。通常，比较好的方法是将文件名读入 `string` 对象，而不是 C 风格字符数组。假设要使用的文件名保存在 `string` 对象中，则可调用 `c_str` 成员（[第 4.3.2 节](#)）获取 C 风格字符串。

Checking Whether an Open Succeeded

检查文件打开是否成功

After opening a file, it is usually a good idea to verify that the open succeeded:

打开文件后，通常要检验打开是否成功，这是一个好习惯：

```
// check that the open succeeded  
if (!infile) {  
    cerr << "error: unable to open input file: "  
        << ifile << endl;  
    return -1;  
}
```

This condition is similar to those we've used to test whether `cin` had hit end-of-file or encountered some other error. When we test a stream, the effect is to test whether the object is "okay" for input or output. If the `open` fails, then the state of the `fstream` object is that it is not ready for doing IO. When we test the object

这个条件与之前测试 `cin` 是否到达文件尾或遇到某些其他错误的条件类似。检查流等效于检查对象是否“适合”输入或输出。如果打开（`open`）失败，则说明 `fstream` 对象还没有为 IO 做好准备。当测试对象

```
if (outfile) // ok to use outfile?
```

a `true` return means that it is okay to use the file. Because we want to know if the file is *not* okay, we invert the return from checking the stream:

返回 `true` 意味着文件已经可以使用。由于希望知道文件是否未准备好，则对返回值取反来检查流：

```
if (!outfile) // not ok to use outfile?
```

Rebinding a File Stream to a New File

将文件流与新文件重新捆绑

Section 8.4. File Input and Output

Once an `fstream` has been opened, it remains associated with the specified file. To associate the `fstream` with a different file, we must first `close` the existing file and then `open` a different file:

`fstream` 对象一旦打开，就保持与指定的文件相关联。如果要把 `fstream` 对象与另一个不同的文件关联，则必须先关闭 (`close`) 现在的文件，然后打开 (`open`) 另一个文件：要点是在尝试打开新文件之前，必须先关闭当前的文件流。`open` 函数会检查流是否已经打开。如果已经打开，则设置内部状态，以指出发生了错误。接下来使用文件流的任何尝试都会失败。

```
ifstream infile("in");      // opens file named "in" for reading
infile.close();              // closes "in"
infile.open("next");        // opens file named "next" for reading
```

Clearing the State of a File Stream

清除文件流的状态

Consider a program that has a `vector` containing names of files it should open and read, doing some processing on the words stored in each file. Assuming the `vector` is named `files`, such a program might have a loop like the following:

考虑这样的程序，它有一个 `vector` 对象，包含一些要打开并读取的文件名，程序要对每个文件中存储的单词做一些处理。假设该 `vector` 对象命名为 `files`，程序也许会有如下循环：

```
// for each file in the vector
while (it != files.end()) {
    ifstream input(it->c_str()); // open the file;
    // if the file is ok, read and "process" the input
    if (!input)
        break;                  // error: bail out!
    while(input >> s)          // do the work on this file
        process(s);
    ++it;                      // increment iterator to get next file
}
```

Each trip through the loop constructs the `ifstream` named `input` open to read the indicated file. The initializer in the constructor uses the arrow operator ([Section 5.6](#), p. 164) which dereferences `it` and fetches the `c_str` member from the underlying `string` that `it` currently denotes. The file is opened by the constructor, and assuming the `open` succeeded, we read that file until we hit end-of-file or some other error condition. At that point, `input` is in an error state. Any further attempt to read from `input` will fail. Because `input` is local to the `while` loop, it is created on each iteration. That means that it starts out each iteration in a clean state `input.good()` is `true`.

每一次循环都构造了名为 `input` 的 `ifstream` 对象，打开并读取指定的文件。构造函数的初始化式使用了箭头操作符（[第 5.6 节](#)）对 `it` 进行解引用，从而获取 `it` 当前表示的 `string` 对象的 `c_str` 成员。文件由构造函数打开，并假设打开成功，读取文件直到到达文件结束符或者出现其他的错误条件为止。在这个点上，`input` 处于错误状态。任何读 `input` 的尝试都会失败。因为 `input` 是 `while` 循环的局部变量，在每次迭代中创建。这就意味着它在每次循环中都以干净的状态即 `input.good()` 为 `true`，开始使用。

If we wanted to avoid creating a new stream object on each trip through the `while`, we might move the definition of `input` out of the `while`. This simple change means that we must manage the stream state more carefully. When we encounter end-of-file, or any other error, the internal state of the stream is set so that further reads or writes are not allowed. Closing a stream does not change the internal state of the stream object. If the last read or write operation failed, the state of the object remains in a failure mode until we execute `clear` to reset the condition of the stream. After the `clear`, it is as if we had created the object afresh.

如果希望避免在每次 `while` 循环过程中创建新流对象，可将 `input` 的定义移到 `while` 之前。这点小小的改动意味着必须更仔细地管理流的状态。如果遇到文件结束符或其他错误，将设置流的内部状态，以便之后不允许再对该流做读写操作。关闭流并不能改变流对象的内部状态。如果最后的读写操作失败了，对象的状态将保持为错误模式，直到执行 `clear` 操作重新恢复流的状态为止。调用 `clear` 后，就像重新创建了该对象一样。

If we wish to reuse an existing stream object, our `while` loop must remember to `close` and `clear` the stream on each trip through the loop:

如果打算重用已存在的流对象，那么 `while` 循环必须在每次循环进记得关闭 (`close`) 和清空 (`clear`) 文件流：

```
ifstream input;
vector<string>::const_iterator it = files.begin();
// for each file in the vector
while (it != files.end()) {
    input.open(it->c_str()); // open the file
    // if the file is ok, read and "process" the input
    if (!input)
        break;                  // error: bail out!
    while(input >> s)          // do the work on this file
        process(s);
    input.close();              // close file when we're done with it
    input.clear();              // reset state to ok
    ++it;                      // increment iterator to get next file
```

}

Had we neglected the call to `clear`, this loop would read only the first file. To see why, consider what happens in this loop: First we open the indicated file. Assuming `open` succeeded, we read the file until we hit end-of-file or some other error condition. At that point, `input` is in an error state. If we `close` but do not `clear` the stream, then any subsequent input operation on `input` will fail. Once we have `closed` the file, we can `open` the next one. However, the read of `input` in the inner `while` will fail after all, the last read from this stream hit end-of-file. The fact that the end-of-file was on a different file is irrelevant!

如果忽略 `clear` 的调用，则循环只能读入第一个文件。要了解其原因，就需要考虑在循环中发生了什么：首先打开指定的文件。假设打开成功，则读取文件直到文件结束或者出现其他错误条件为止。在这个点上，`input` 处于错误状态。如果在关闭（`close`）该流前没有调用 `clear` 清除流的状态，接着在 `input` 上做的任何输入运算都会失败。一旦关闭该文件，再打开下一个文件时，在内层 `while` 循环上读 `input` 仍然会失败——毕竟最后一次对流的读操作到达了文件结束符，事实上该文件结束符对应的是另一个与本文件无关的其他文件。



If we reuse a file stream to read or write more than one file, we must `clear` the stream before using it to read from another file.

如果程序员需要重用文件流读写多个文件，必须在读另一个文件之前调用 `clear` 清除该流的状态。

Exercises Section 8.4.1

Exercise 8.6: Because `ifstream` inherits from `istream`, we can pass an `ifstream` object to a function that takes a reference to an `istream`. Use the function you wrote for the first exercise in [Section 8.2](#) (p. 291) to read a named file.

由于 `ifstream` 继承了 `istream`，因此可将 `ifstream` 对象传递给形参为 `istream` 引用的函数。使用[第 8.2 节](#)第一个习题编写的函数读取已命名的文件。

Exercise 8.7: The two programs we wrote in this section used a `break` to exit the `while` loop if the open failed for any file in the `vector`. Rewrite these two loops to print a warning message if a file can't be opened and continue processing by getting the next file name from the `vector`.

本节编写的两个程序，在打开 `vector` 容器中存放的任何文件失败时，使用 `break` 跳出 `while` 循环。重写这两个循环，如果文件无法打开，则输出警告信息，然后从 `vector` 中获取下一个文件名继续处理。

Exercise 8.8: The programs in the previous exercise can be written without using a `continue` statement. Write the program with and without using a `continue`.

上一个习题的程序可以不用 `continue` 语句实现。分别使用或不使用 `continue` 语句编写该程序。

Exercise 8.9: Write a function to open a file for input and read its contents into a `vector` of `strings`, storing each line as a separate element in the `vector`.

编写函数打开文件用于输入，将文件内容读入 `string` 类型的 `vector` 容器，每一行存储为该容器对象的一个元素。

Exercise 8.10: Rewrite the previous program to store each word in a separate element.

重写上面的程序，把文件中的每个单词存储为容器的一个元素。

8.4.2. File Modes

8.4.2. 文件模式

Section 8.4. File Input and Output

Whenever we open a file either through a call to `open` or as part of initializing a stream from a file name a **file mode** is specified. Each `fstream` class defines a set of values that represent different modes in which the stream could be opened. Like the condition state flags, the file modes are integral constants that we use with the bitwise operators ([Section 5.3](#), p. 154) to set one or more modes when we open a given file. The file stream constructors and `open` have a default argument ([Section 7.4.1](#), p. 253) to set the file mode. The value of the default varies based on the type of the stream. Alternatively, we can supply the mode in which to open the file. [Table 8.3](#) on the next page lists the file modes and their meanings.

在打开文件时，无论是调用 `open` 还是以文件名作为流初始化的一部分，都需指定**文件模式 (file mode)**。每个 `fstream` 类都定义了一组表示不同模式的值，用于指定流打开的不同模式。与条件状态标志一样，文件模式也是整型常量，在打开指定文件时，可用位操作符（[第 5.3 节](#)）设置一个或多个模式。文件流构造函数和 `open` 函数都提供了默认实参（[第 7.4.1 节](#)）设置文件模式。默认值因流类型的不同而不同。此外，还可以显式地以模式打开文件。[表 8.3](#) 列出了文件模式及其含义。

Table 8.3. File Modes

表 8.3 文件模式

<code>in</code>	open for input 打开文件做读操作
<code>out</code>	open output 打开文件做写操作
<code>app</code>	seek to the end before every write 在每次写之前找到文件尾
<code>ate</code>	seek to the end immediately after the open 打开文件后立即将文件定位在文件尾
<code>trunc</code>	truncate an existing stream when opening it 打开文件时清空已存在的文件流
<code>binary</code>	do IO operations in binary mode 以二进制模式进行 IO 操作

The modes `out`, `trunc`, and `app` may be specified only for files associated with an `ofstream` or an `fstream`; `in` may be specified only for files associated with either `ifstream` or `fstream`. Any file may be opened in `ate` or `binary` mode. The `ate` mode has an effect only at the open: Opening a file in `ate` mode puts the file at the end-of-file immediately after the open. A stream opened in `binary` mode processes the file as a sequence of bytes; it does no interpretation of the characters in the stream.

`out`, `trunc` 和 `app` 模式只能用于指定与 `ofstream` 或 `fstream` 对象关联的文件；`in` 模式只能用于指定与 `ifstream` 或 `fstream` 对象关联的文件。所有的文件都可以用 `ate` 或 `binary` 模式打开。`ate` 模式只在打开时有效：文件打开后将定位在文件尾。以 `binary` 模式打开的流则将文件以字节序列的形式处理，而不解释流中的字符。

By default, files associated with an `ifstream` are opened in `in` mode, which is the mode that permits the file to be read. Files opened by an `ofstream` are opened in `out` mode, which permits the file to be written. A file opened in `out` mode is truncated: All data stored in the file is discarded.

默认时，与 `ifstream` 流对象关联的文件将以 `in` 模式打开，该模式允许文件做读的操作；与 `ofstream` 关联的文件则以 `out` 模式打开，使文件可写。以 `out` 模式打开的文件会被清空：丢弃该文件存储的所有数据。

In effect, specifying `out` mode for an `ofstream` is equivalent to specifying both `out` and `trunc`.

从效果来看，为 `ofstream` 对象指定 `out` 模式等效于同时指定了 `out` 和 `trunc` 模式。



The only way to preserve the existing data in a file opened by an `ofstream` is to specify `app` mode explicitly:

对于用 `ofstream` 打开的文件，要保存文件中存在的数据，唯一方法是显式地指定 `app` 模式打开：

Section 8.4. File Input and Output

```
// output mode by default; truncates file named "file1"
ofstream outfile("file1");
// equivalent effect: "file1" is explicitly truncated
ofstream outfile2("file1", ofstream::out | ofstream::trunc);
// append mode; adds new data at end of existing file named "file2"
ofstream appfile("file2", ofstream::app);
```

The definition of `outfile2` uses the bitwise OR operator ([Section 5.3](#), p. 154) to open `inOut` in both `out` and `trunc` mode.

`outfile2` 的定义使用了按位或操作符 ([第 5.3 节](#)) 将相应的文件同时以 `out` 和 `trunc` 模式打开。

Using the Same File for Input and Output

对同一个文件作输入和输出运算

An `fstream` object can both read and write its associated file. How an `fstream` uses its file depends on the mode specified when we open the file.

`fstream` 对象既可以读也可以写它所关联的文件。`fstream` 如何使用它的文件取决于打开文件时指定的模式。

By default, an `fstream` is opened with both `in` and `out` set. A file opened with both `in` and `out` mode set is not truncated. If we open the file associated with an `fstream` with `out` mode, but not `in` mode specified, then the file is truncated. The file is also truncated if `trunc` is specified, regardless of whether `in` is specified. The following definition opens the file `copyOut` in both input and output mode:

默认情况下, `fstream` 对象以 `in` 和 `out` 模式同时打开。当文件同时以 `in` 和 `out` 打开时不清空。如果打开 `fstream` 所关联的文件时, 只使用 `out` 模式, 而不指定 `in` 模式, 则文件会清空已存在的数据。如果打开文件时指定了 `trunc` 模式, 则无论是否同时指定了 `in` 模式, 文件同样会被清空。下面的定义将 `copyOut` 文件同时以输入和输出的模式打开:

```
// open for input and output
fstream inOut("copyOut", fstream::in | fstream::out);
```

[Appendix A.3.8](#) (p. 837) discusses how to use a file that is opened for both input and output.

对于同时以输入和输出的模式打开的文件, [附录 A.3.8](#) 将讨论其使用方法。

Mode Is an Attribute of a File, Not a Stream

模式是文件的属性而不是流的属性

The mode is set each time a file is opened:

每次打开文件时都会设置模式

```
ofstream outfile;
// output mode set to out, "scratchpad" truncated
outfile.open("scratchpad", ofstream::out);
outfile.close(); // close outfile so we can rebind it
// appends to file named "precious"
outfile.open("precious", ofstream::app);
outfile.close();
// output mode set by default, "out" truncated
outfile.open("out");
```

The first call to `open` specifies `ofstream::out`. The file named "scratchpad" in the current directory is opened in output mode; the file will be truncated. When we open the file named "precious," we ask for append mode. Any data in the file remains, and all writes are done at the end of the file. When we opened "out," we did not specify an output mode explicitly. It is opened in `out` mode, meaning that any data currently in "out" is discarded.

第一次调用 `open` 函数时, 指定的模式是 `ofstream::out`。当前目录中名为“scratchpad”的文件以输出模式打开并清空。而名为“precious”的文件, 则要求以添加模式打开: 保存文件里的原有数据, 所有的新内容在文件尾部写入。在打开“out”文件时, 没有明确指明输出模式, 该文件则以 `out` 模式打开, 这意味着当前存储在“out”文件中的任何数据都将被丢弃。

 Any time `open` is called, the file mode is set, either explicitly or implicitly. If a mode is not specified, the default value is used.

只要调用 `open` 函数, 就要设置文件模式, 其模式的设置可以是显式的也可以是隐式的。如果没有指定文件模式, 将使用默认值。

Valid Combinations for Open Mode

打开模式的有效组合

Not all open modes can be specified at once. Some are nonsensical, such as opening a file setting both `in` and `trunc`. That would yield a stream we intend to read but that we have truncated so that there is no data to read. [Table 8.4](#) lists the valid mode combinations and their meanings.

并不是所有的打开模式都可以同时指定。有些模式组合是没有意义的，例如同时以 `in` 和 `trunc` 模式打开文件，准备读取所生成的流，但却因为 `trunc` 操作而导致无数据可读。[表 8.4](#) 列出了有效的模式组合及其含义。

Table 8.4. File Mode Combinations

表 8.4 文件模式的组合

<code>out</code>	open for output; deletes existing data in the file 打开文件做写操作，删除文件中已有的数据
<code>out app</code>	open for output; all writes at end of file 打开文件做写操作，在文件尾写入
<code>out trunc</code>	same as <code>out</code> 与 <code>out</code> 模式相同
<code>in</code>	open for input 打开文件做读操作
<code>in out</code>	open for both input and output; positioned to read the beginning of the file 打开文件做读、写操作，并定位于文件开头处
<code>in out trunc</code>	open for both input and output, deletes existing data in the file 打开文件做读、写操作，删除文件中已有的数据

Any open mode combination may also include `ate`. The effect of adding `ate` to any of these modes changes only the initial position of the file. Adding `ate` to any of these mode combinations positions the file to the end before the first input or output operation is performed.

上述所有的打开模式组合还可以添加 `ate` 模式。对这些模式添加 `ate` 只会改变文件打开时的初始化定位，在第一次读或写之前，将文件定位于文件末尾处。

8.4.3. A Program to Open and Check Input Files

8.4.3. 一个打开并检查输入文件的程序

Several programs in this book open a given file for input. Because we need to do this work in several programs, we'll write a function, named `open_file`, to perform it. Our function takes references to an `ifstream` and a `string`. The `string` holds the name of a file to associate with the given `ifstream`:

本书好几个程序都要打开给定文件用输入。由于需要在多个程序里做这件工作，我们编写一个名为 `open_file` 的函数实现这个功能。这个函数有两个引用形参，分别是 `ifstream` 和 `string` 类型，其中 `string` 类型的引用形参存储与指定 `ifstream` 对象关联的文件名：

```
// opens in binding it to the given file
ifstream& open_file(ifstream &in, const string &file)
{
```

Section 8.4. File Input and Output

```
in.close();      // close in case it was already open  
in.clear();     // clear any existing errors  
// if the open fails, the stream will be in an invalid state  
in.open(file.c_str()); // open the file we were given  
return in; // condition state is good if open succeeded  
}
```

Because we do not know what state the stream is in, we start by calling `close` and `clear` to put the stream into a valid state. We next attempt to `open` the given file. If the `open` fails, the stream's condition state will indicate that the stream is unusable. We finish by returning the stream, which is either bound to the given file and ready to use or is in an error condition.

由于不清楚流 `in` 的当前状态，因此首先调用 `close` 和 `clear` 将这个流设置为有效状态。然后尝试打开给定的文件。如果打开失败，流的条件状态将标志这个流是不可用的。最后返回流对象 `in`，此时，`in` 要么已经与指定文件绑定起来了，要么处于错误条件状态。

Exercises Section 8.4.3

Exercise 8.11: In the `open_file` function, explain why we call `clear` before the call to `open`. What would happen if we neglected to make this call? What would happen if we called `clear` after the `open`?

对于 `open_file` 函数，请解释为什么在调用 `open` 前先调用 `clear` 函数。如果忽略这个函数调用，会出现什么问题？如果在 `open` 后面调用 `clear` 函数，又会怎样？

Exercise 8.12: In the `open_file` function, explain what the effect would be if the program failed to execute the `close`.

对于 `open_file` 函数，请解释如果程序执行 `close` 函数失败，会产生什么结果？

Exercise 8.13: Write a program similar to `open_file` that opens a file for output.

编写类似 `open_file` 的程序打开文件用于输出。

Exercise 8.14: Use `open_file` and the program you wrote for the first exercise in [Section 8.2](#) (p. 291) to open a given file and read its contents.

使用 `open_file` 函数以及[第 8.2 节](#)第一个习题编写的程序，打开给定的文件并读取其内容。

8.5. String Streams

8.5. 字符串流

The `iostream` library supports in-memory input/output, in which a stream is attached to a `string` within the program's memory. That `string` can be written to and read from using the `iostream` input and output operators. The library defines three kinds of string streams:

`iostream` 标准库支持内存中的输入／输出，只要将流与存储在程序内存中的 `string` 对象捆绑起来即可。此时，可使用 `iostream` 输入和输出操作符读写这个 `string` 对象。标准库定义了三种类型的字符串流：

- `istringstream`, derived from `istream`, reads from a `string`.
`istringstream`, 由 `istream` 派生而来，提供读 `string` 的功能。
- `ostringstream`, derived from `ostream`, writes to a `string`.
`ostringstream`, 由 `ostream` 派生而来，提供写 `string` 的功能。
- `stringstream`, derived from `iostream`, reads and writes a `string`.
`stringstream`, 由 `iostream` 派生而来，提供读写 `string` 的功能。

To use any of these classes, we must include the `sstream` header.

要使用上述类，必须包含 `sstream` 头文件。

Like the `fstream` types, these types are derived from the `iostream` types, meaning that all the operations on `iostreams` also apply to the types in `sstream`. In addition to the operations that the `sstream` types inherit, these types have a constructor that takes a `string`. The constructor copies the `string` argument into the `stringstream` object. The operations that read and write the `stringstream` read or write the `string` in the object. These classes also define a member named `str` to fetch or set the `string` value that the `stringstream` manipulates.

与 `fstream` 类型一样，上述类型由 `iostream` 类型派生而来，这意味着 `iostream` 上所有的操作适用于 `sstream` 中的类型。`sstream` 类型除了继承的操作外，还各自定义了一个有 `string` 形参的构造函数，这个构造函数将 `string` 类型的实参复制给 `stringstream` 对象。对 `stringstream` 的读写操作实际上读写的就是该对象中的 `string` 对象。这些类还定义了名为 `str` 的成员，用来读取或设置 `stringstream` 对象所操纵的 `string` 值。

Note that although `fstream` and `sstream` share a common base class, they have no other interrelationship. In particular, we cannot use `open` and `close` on a `stringstream`, nor can we use `str` on an `fstream`.

注意到尽管 `fstream` 和 `sstream` 共享相同的基类，但它们没有其他相互关系。特别是，`stringstream` 对象不使用 `open` 和 `close` 函数，而 `fstream` 对象则不允许使用 `str`。

Table 8.5. `stringstream`-Specific Operations

表 8.5. `stringstream` 特定的操作

<code>stringstream strm;</code>	Creates an unbound <code>stringstream</code> . 创建自由的 <code>stringstream</code> 对象
<code>stringstream strm(s);</code>	Creates a <code>stringstream</code> that holds a copy of the <code>string</code> s. 创建存储 s 的副本的 <code>stringstream</code> 对象，其中 s 是 <code>string</code> 类型的对象
<code>strm.str()</code>	Returns a copy of the <code>string</code> that <code>strm</code> holds. 返回 <code>strm</code> 中存储的 <code>string</code> 类型对象
<code>strm.str(s)</code>	Copies the <code>string</code> s into <code>strm</code> . Returns <code>void</code> . 将 <code>string</code> 类型的 s 复制给 <code>strm</code> ，返回 <code>void</code>

Using a `stringstream`

`stringstream` 对象的和使用

We've seen programs that need to deal with their input a word at a time or a line at a time. The first sort of programs use the `string` input operator and the second use the `getline` function. However, some programs need to do both: They have some processing to do on a per-line basis and other work that needs to be done on each word within each line. Using `stringstreams` lets us do so:

前面已经见过以每次一个单词或每次一行的方式处理输入的程序。第一种程序用 `string` 输入操作符，而第二种则使用 `getline` 函数。然而，有些程序需要同时使用这两种方式：有些处理基于每行实现，而其他处理则要操纵每行中每个单词。可用 `stringstreams` 对象实现：

```
string line, word;      // will hold a line and word from input, respectively
while (getline(cin, line)) {           // read a line from the input into line
    // do per-line processing
    istringstream stream(line);        // bind to stream to the line we read
    while (stream >> word){          // read a word from line
        // do per-word processing
    }
}
```

Here we use `getline` to get an entire line from the input. To get the words in each line, we bind an `istringstream` to the line that we read. We can then use the normal `string` input operator to read the words from each line.

这里，使用 `getline` 函数从输入读取整行内容。然后为了获得每行中的单词，将一个 `istringstream` 对象与所读取的行绑定起来，这样只需要使用普通的 `string` 输入操作符即可读出每行中的单词。

`stringstreams` Provide Conversions and/or Formatting

`stringstream` 提供的转换和／或格式化

One common use of `stringstreams` is when we want to obtain automatic formatting across multiple data types. For example, we might have a collection of numeric values but want their `string` representation or vice versa. The `sstream` input and output operations automatically convert an arithmetic type into its corresponding `string` representation or back again:

`stringstream` 对象的一个常见用法是，需要在多种数据类型之间实现自动格式化时使用该类类型。例如，有一个数值型数据集合，要获取它们的 `string` 表示形式，或反之。`sstream` 输入和输出操作可自动地把算术类型转化为相应的 `string` 表示形式，反过来也可以。

```
int val1 = 512, val2 = 1024;
ostringstream format_message;
// ok: converts values to a string representation
format_message << "val1: " << val1 << "\n"
<< "val2: " << val2 << "\n";
```

Here we create an empty `ostringstream` object named `format_message` and insert the indicated text into that object. What's important is that the `int` values are automatically converted to their printable `string` equivalents. The contents of `format_message` are the characters

这里创建了一个名为 `format_message` 的 `ostringstream` 类型空对象，并将指定的内容插入该对象。重点在于 `int` 型值自动转换为等价的可打印的字符串。`format_message` 的内容是以下字符：

```
val1: 512\nval2: 1024
```

We could retrieve the numeric value by using an `istringstream` to read from the `string`. Reading an `istringstream` automatically converts from the character representation of a numeric value to its corresponding arithmetic value:

相反，用 `istringstream` 读 `string` 对象，即可重新将数值型数据找回来。读取 `istringstream` 对象自动地将数值型数据的字符表示方式转换为相应的算术值。

```
// str member obtains the string associated with a stringstream
istringstream input_istring(format_message.str());
string dump; // place to dump the labels from the formatted message
// extracts the stored ascii values, converting back to arithmetic types
input_istring >> dump >> val1 >> dump >> val2;
cout << val1 << " " << val2 << endl; // prints 512 1024
```

Here we use the `str` member to obtain a copy of the `string` associated with the `ostringstream` we previously created. We bind `input_istring` to

Section 8.5. String Streams

that `string`. When we read `input_istring`, the values are converted back to their original numeric representations.

这里使用 `str` 成员获取与之前创建的 `ostringstream` 对象关联的 `string` 副本。再将 `input_istring` 与 `string` 绑定起来。在读 `input_istring` 时，相应的值恢复为它们原来的数值型表示形式



To read `input_string`, we must parse the `string` into its component parts. We want the numeric values; to get them we must read (and ignore) the labels that are interspersed with the data we want.

为了读取 `input_string`, 必须把该 `string` 对象分解为若干个部分。我们要的是数值型数据；为了得到它们，必须读取（和忽略）处于所需数据周围的标号。

Because the input operator reads typed values, it is essential that the types of the objects into which we read be compatible with the types of the values read from the `stringstream`. In this case, `input_istring` had four components: The `string` value `val1`: followed by `512` followed by the `string` `val2`: followed by `1024`. As usual, when we read `strings` using the input operator, whitespace is ignored. Thus, when we read the `string` associated with `format_message`, we can ignore the newlines that are part of that value.

因为输入操作符读取的是有类型的值，因此读入的对象类型必须和由 `stringstream` 读入的值的类型一致。在本例中，`input_istring` 分成四个部分：`string` 类型的值 `val1`，接着是 `512`，然后是 `string` 类型的值 `val2`，最后是 `1024`。一般情况下，使用输入操作符读 `string` 时，空白符将会忽略。于是，在读与 `format_message` 关联的 `string` 时，忽略其中的换行符。

Exercises Section 8.5

Exercise 8.15: Use the function you wrote for the first exercise in [Section 8.2](#) (p. 291) to print the contents of an `istringstream` object.

使用[第 8.2 节](#)第一个习题编写的函数输出 `istringstream` 对象的内容。

Exercise 8.16: Write a program to store each line from a file in a `vector<string>`. Now use an `istringstream` to read each line from the `vector` a word at a time.

编写程序将文件中的每一行存储在 `vector<string>` 容器对象中，然后使用 `istringstream` 从 `vector` 里以每次读一个单词的形式读取存储的行。

Chapter Summary

小结

C++ uses library classes to handle input and output:

C++ 使用标准库类处理输入和输出：

- The `iostream` classes handle stream-oriented input and output
`iostream` 类处理面向流的输入和输出。
- The `fstream` classes handle IO to named files
`fstream` 类处理已命名文件的 IO。
- The `stringstream` classes do IO to in-memory `strings`
`stringstream` 类处理内存中字符串的 IO。

All of these classes are related by inheritance. The input classes inherit from `istream` and the output classes from `ostream`. Thus, operations that can be performed on an `istream` object can also be performed on either an `ifstream` or an `istringstream`. Similarly for the output classes, which inherit from `ostream`.

所有的这些类都是通过继承相互关联的。输入类继承了 `istream`, 而输出类则继承了 `ostream`。因此, 可在 `istream` 对象上执行的操作同样适用于 `ifstream` 或 `istringstream` 对象。而继承 `ostream` 的输出类也是类似的。

Each IO object maintains a set of condition states that indicate whether IO can be done through this object. If an error is encountered such as hitting end-of-file on an input stream then the object's state will be such that no further input can be done until the error is rectified. The library provides a set of functions to set and test these states.

所有 IO 对象都有一组条件状态, 用来指示是否可以通过该对象进行 IO 操作。如果出现了错误 (例如遇到文件结束符) 对象的状态将标志无法再进行输入, 直到修正了错误为止。标准库提供了一组函数设置和检查这些状态。

Defined Terms

术语

base class (基类)

A class that is the parent of another class. The base class defines the interface that a derived class inherits.

是其他类的父类。基类定义了派生类所继承的接口。

condition state (条件状态)

Flags and associated functions usable by any of the stream classes that indicate whether a given stream is usable. States and functions to get and set these states are listed in [Table 8.2](#) (p. 288).

流类用于指示给定的流是否用的标志以及相关函数。[表 8.2](#) 列出了流的状态以及获取和设置这些状态的函数。

derived class (派生类)

A derived class is one that shares an interface with its parent class.

与父类共享接口的类。

file mode (文件模式)

Flags defined by the `fstream` classes that are specified when opening a file and control how a file can be used. Listed in [Table 8.3](#) (p. 297).

由 `fstream` 类定义的标志，在打开文件和控制文件如何使用时指定。[表 8.3](#) 列出了所有的文件模式。

fstream

Stream object that reads or writes a named file. In addition to the normal `iostream` operations, the `fstream` class also defines `open` and `close` members. The `open` member function takes a C-style character string that names the file to open and an optional open mode argument. By default `ifstream`s are opened with `in` mode, `ofstream`s with `out` mode, and `fstream`s with `in` and `out` mode set. The `close` member closes the file to which the stream is attached. It must be called before another file can be `opened`.

用来读或写已命名文件的流对象。除了普通的 `iostream` 操作外，`fstream` 类还定义了 `open` 和 `close` 成员。`open` 成员函数有一个表示打开文件名的 C 风格字符串参数和一个可选的打开模式参数。默认时，`ifstream` 对象以 `in` 模式打开，`ofstream` 对象以 `out` 模式打开，而 `fstream` 对象则同时以 `in` 和 `out` 模式打开。`close` 成员关闭流关联的文件，必须在打开另一个文件前调用。

inheritance (继承)

Types that are related by inheritance share a common interface. A derived class inherits properties from its base class. [Chapter 15](#) covers inheritance.

有继承关系的类型共享相同的接口。派生类继承其基类的属性。[第十五章](#)将继承。

object-oriented library (面向对象标准库)

A set of classes related by inheritance. Generally speaking, the base class of an object-oriented library defines an interface that is shared by the classes derived from that base class. In the IO library, the `istream` and `ostream` classes serve as base classes for the types defined in the `fstream` and `sstream` headers. We can use an object of a derived class as if it were an object of the base class. For example, we can use the operations defined for `istream` on an `ifstream` object.

有继承关系的类的集合。一般来说，面向对象标准库的基类定义了接口，由继承这个 基类的各个派生类共享。在 IO 标准库中，`istream` 和 `ostream` 类是 `fstream` 和 `sstream` 头文件中定义的类型的基类。派生类的对象可当做基类对象使用。例如，可在 `ifstream` 对象上使用 `istream` 定义的操作。

stringstream

Stream object that reads or writes a `string`. In addition to the normal `iostream` operations, it also defines an overloaded member named `str`. Calling `str` with no arguments returns the `string` to which the `stringstream` is attached. Calling it with a `string` attaches the `stringstream` to a copy of that `string`.

读写字符串的流对象。除了普通的 `iostream` 操作外，它还定义了名为 `str` 的重载成员。无实参地调用 `str` 将返回 `stringstream` 所关联的 `string` 值。用 `string` 对象做实参调用它则将该 `stringstream` 对象与实参副本相关联。

Team LiB

◀ PREVIOUS NEXT ▶

Chapter 9. Sequential Containers

CONTENTS

目录

Section 9.1 Defining a Sequential Container	307
Section 9.2 Iterators and Iterator Ranges	311
Section 9.3 Sequence Container Operations	316
Section 9.4 How a <code>vector</code> Grows	330
Section 9.5 Deciding Which Container to Use	333
Section 9.6 <code>strings</code> Revisited	335
Section 9.7 Container Adaptors	348
Chapter Summary	353
Defined Terms	353

This chapter completes our discussion of the standard-library sequential container types. It expands on the material from [Chapter 3](#), which introduced the most commonly used sequential container, the `vector` type. Elements in a sequential container are stored and accessed by position. The library also defines several associative containers, which hold elements whose order depends on a key. Associative containers are covered in the next chapter.

第三章介绍了最常用的顺序容器：`vector` 类型。本章将对第三章的内容进行扩充和完善，继续讨论标准库提供的顺序容器类型。顺序容器内的元素按其位置存储和访问。除顺序容器外，标准库还定义了几种关联容器，其元素按键（key）排序。我们将在下一章讨论它们。

The container classes share a common interface. This fact makes the library easier to learn; what we learn about one type applies to another. Each container type offers a different set of time and functionality tradeoffs. Often a program using one type can be fine-tuned by substituting another container without changing our code beyond the need to change type declarations.

容器类共享公共的接口，这使标准库更容易学习，只要学会其中一种类型就能运用另一种类型。每种容器类型提供一组不同的时间和功能折衷方案。通常不需要修改代码，只需改变类型声明，用一种容器类型替代另一种容器类型，就可以优化程序的性能。

A *container* holds a collection of objects of a specified type. We've used one kind of container already: the library `vector` type. It is a **sequential container**. It holds a collection of elements of a single type, making it a container. Those elements are stored and accessed by position, making it a sequential container. The order of elements in a sequential container is independent of the value of the elements. Instead, the order is determined by the order in which elements are added to the container.

容器容纳特定类型对象的集合。我们已经使用过一种容器类型：标准库 `vector` 类型，这是一种**顺序容器 (sequential container)**。它将单一类型元素聚集起来成为容器，然后根据位置来存储和访问这些元素，这就是顺序容器。顺序容器的元素排列次序与元素值无关，而是由元素添加到容器里的次序决定。

The library defines three kinds of sequential containers: `vector`, `list`, and `deque` (short for "double-ended queue" and pronounced "deck"). These types differ in how elements are accessed and the relative run-time cost of adding or removing elements. The library also provides three container **adaptors**. Effectively, an adaptor *adapts* an underlying container type by defining a new interface in terms of the operations provided by the original type. The sequential container adaptors are `stack`, `queue`, and `priority_queue`.

标准库定义了三种顺序容器类型：`vector`、`list` 和 `deque`（是双端队列“double-ended queue”的简写，发音为“deck”）。它们的差别在于访问元素的方式，以及添加或删除元素相关操作的运行代价。标准库还提供了三种容器适配器（**adaptors**）。实际上，适配器是根据原始的容器类型所提供的操作，通过定义新的操作接口，来适应基础的容器类型。顺序容器适配器包括 `stack`、`queue` 和 `priority_queue` 类型，见表 9-1。

Containers define only a small number of operations. Many additional operations are provided by the algorithms library, which we'll cover in [Chapter 11](#). For those operations that are defined by the containers, the library imposes a common interface. The containers vary as to which operations they provide, but if two containers provide the same operation, then the interface (name and number of arguments) will be the same for both container types. The set of operations on the container types form a kind of hierarchy:

容器只定义了少量操作。大多数额外操作则由算法库提供，我们将在[第十一章](#)学习算法库。标准库为容器类型定义的操作强加了公共的接口。这些容器类型的差别在于它们提供哪些操作，但是如果两个容器提供了相同的操作，则它们的接口（函数名字和参数个数）应该相同。容器类型的操作集合形成了以下层次结构：

- Some operations are supported by all container types.

一些操作适用于所有容器类型。

- Other operations are common to only the sequential or only the associative containers.

另外一些操作则只适用于顺序或关联容器类型。

- Still others are common to only a subset of either the sequential or associative containers.

还有一些操作只适用于顺序或关联容器类型的一个子集。

In the remainder of this chapter, we look at the sequential container types and their operations in detail.

在本章的后续部分，我们将详细描述顺序容器类型和它们所提供的操作。

Table 9.1. Sequential Container Types

<i>Sequential Containers</i>	
顺序容器	
<code>vector</code>	Supports fast random access 支持快速随机访问
<code>list</code>	Supports fast insertion/deletion 支持快速插入/删除
<code>deque</code>	Double-ended queue 双端队列
<i>Sequential Container Adaptors</i>	
顺序容器适配器	
<code>stack</code>	Last in/First out stack 后进先出 (LIFO) 堆栈
<code>queue</code>	First in/First out queue 先进先出 (FIFO) 队列
<code>priority_queue</code>	Priority-managed queue 有优先级管理的队列

9.1. Defining a Sequential Container

9.1. 顺序容器的定义

We already know a fair bit about how to use the sequential containers based on what we covered in [Section 3.3](#) (p. 90). To define a container object, we must include its associated header file, which is one of

在第 3.3 节中，我们已经了解了一些使用顺序容器类型的知识。为了定义一个容器类型的对象，必须先包含相关的头文件，即下列头文件之一：

```
#include <vector>
#include <list>
#include <deque>
```

Each of the containers is a class template ([Section 3.3](#), p. 90). To define a particular kind of container, we name the container followed by angle brackets that enclose the type of the elements the container will hold:

所有的容器都是类模板（[第 3.3 节](#)）。要定义某种特殊的容器，必须在容器名后加一对尖括号，尖括号里面提供容器中存放的元素的类型：

```
vector<string> svec;      // empty vector that can hold strings
list<int> ilist;         // empty list that can hold ints
deque<Sales_item> items; // empty deque that holds Sales_items
```

Each container defines a default constructor that creates an empty container of the specified type. Recall that a default constructor takes no arguments.

所有容器类型都定义了默认构造函数，用于创建指定类型的空容器对象。默认构造函数不带参数。



For reasons that shall become clear shortly, the most commonly used container constructor is the default constructor. In most programs, using the default constructor gives the best run-time performance and makes using the container easier.

为了使程序更清晰、简短，容器类型最常用的构造函数是默认构造函数。在大多数的程序中，使用默认构造函数能达到最佳运行时性能，并且使容器更容易使用。

9.1.1. Initializing Container Elements

9.1.1. 容器元素的初始化

In addition to defining a default constructor, each container type also supports constructors that allow us to specify initial element values.

除了默认构造函数，容器类型还提供其他的构造函数，使程序员可以指定元素初值，见[表 9.2](#)。

Table 9.2. Container Constructors

C<T> c;	Create an empty container named <code>c</code> . <code>c</code> is a container name, such as <code>vector</code> , and <code>T</code> is the element type, such as <code>int</code> or <code>string</code> . Valid for all containers.
	创建一个名为 <code>c</code> 的空容器。 <code>c</code> 是容器类型名，如 <code>vector</code> ， <code>T</code> 是元素类型，如 <code>int</code> 或 <code>string</code> 适用于所有容器。
C c(c2);	Create <code>c</code> as a copy of container <code>c2</code> ; <code>c</code> and <code>c2</code> must be the same container type and hold values of the same type. Valid for all containers.
	创建容器 <code>c2</code> 的副本 <code>c</code> ； <code>c</code> 和 <code>c2</code> 必须具有相同的容器类型，并存放相同类型的元素。适用于所有容器。
C c(b, e);	Create <code>c</code> with a copy of the elements from the range denoted by iterators <code>b</code> and <code>e</code> . Valid for all containers.

Section 9.1. Defining a Sequential Container

创建 `c`, 其元素是迭代器 `b` 和 `e` 标示的范围内元素的副本。适用于所有容器。

`C c(n,
t);`

Create `c` with `n` elements, each with value `t`, which must be a value of the element type of `c` or a type convertible to that type.

用 `n` 个值为 `t` 的元素创建容器 `c`, 其中值 `t` 必须是容器类型 `c` 的元素类型的值, 或者是可转换为该类型的值。

Sequential containers only.

只适用于顺序容器

`C c(n);` Create `c` with `n` value-initialized ([Section 3.3.1](#), p. 92) elements.

创建有 `n` 个值初始化 ([第 3.3.1 节](#)) (value-initialized) 元素的容器 `c`。

Sequential containers only.

只适用于顺序容器

Intializing a Container as a Copy of Another Container

将一个容器初始化为另一个容器的副本

When we initialize a sequential container using any constructor other than the default constructor, we must indicate how many elements the container will have. We must also supply initial values for those elements. One way to specify both the size and element values is to initialize a new container as a copy of an existing container of the same type:

当不使用默认构造函数, 而是用其他构造函数初始化顺序容器时, 必须指出该容器有多少个元素, 并提供这些元素的初值。同时指定元素个数和初值的一个方法是将新创建的容器初始化为一个同类型的已存在容器的副本:

```
vector<int> ivec;
vector<int> ivec2(ivec); // ok: ivec is vector<int>
list<int> ilist(ivec); // error: ivec is not list<int>
vector<double> dvec(ivec); // error: ivec holds int not double
```



When we copy one container into another, the types must match exactly: The container type and element type must be the same.

将一个容器复制给另一个容器时, 类型必须匹配: 容器类型和元素类型都必须相同。

Initialzing as a Copy of a Range of Elements

初始化为一段元素的副本

Although we cannot copy the elements from one kind of container to another directly, we can do so indirectly by passing a pair of iterators ([Section 3.4](#), p. 95). When we use iterators, there is no requirement that the container types be identical. The element types in the containers can differ as long as they are compatible. It must be possible to convert the element we copy into the type held by the container we are constructing.

尽管不能直接将一种容器内的元素复制给另一种容器, 但系统允许通过传递一对迭代器 ([第 3.4 节](#)) 间接实现该功能。使用迭代器时, 不要求容器类型相同。容器内的元素类型也可以不相同, 只要它们相互兼容, 能够将要复制的元素转换为所构建的新容器的元素类型, 即可实现复制。

The iterators denote a range of elements that we want to copy. These elements are used to initialize the elements of the new container. The iterators mark the first and one past the last element to be copied. We can use this form of initialization to copy a container that we could not copy directly. More importantly, we can use it to copy only a subsequence of the other container:

迭代器标记了要复制的元素范围, 这些元素用于初始化新容器的元素。迭代器标记出要复制的第一个元素和最后一个元素。采用这种初始化形式可复制不能直接复制的容器。更重要的是, 可以实现复制其他容器的一个子序列:

```
// initialize slist with copy of each element of svec
list<string> slist(svec.begin(), svec.end());
```

Section 9.1. Defining a Sequential Container

```
// find midpoint in the vector
vector<string>::iterator mid = svec.begin() + svec.size()/2;

// initialize front with first half of svec: The elements up to but not including *mid
deque<string> front(svec.begin(), mid);
// initialize back with second half of svec: The elements *mid through end of svec
deque<string> back(mid, svec.end());
```

Recall that pointers are iterators, so it should not be surprising that we can initialize a container from a pair of pointers into a built-in array:

回顾一下指针，我们知道指针就是迭代器，因此允许通过使用内置数组中的一对指针初始化容器也就不奇怪了：

```
char *words[] = {"stately", "plump", "buck", "mulligan"};

// calculate how many elements in words
size_t words_size = sizeof(words)/sizeof(char *);

// use entire array to initialize words2
list<string> words2(words, words + words_size);
```

Here we use `sizeof` ([Section 5.8](#), p. 167) to calculate the size of the array. We add that size to a pointer to the first element to get a pointer to a location one past the end of the array. The initializers for `words2` are a pointer to the first element in `words` and a second pointer one past the last element in that array. The second pointer serves as a stopping condition; the location it addresses is not included in the elements to be copied.

这里，使用 `sizeof` ([5.8 节](#)) 计算数组的长度。将数组长度加到指向第一个元素的指针上就可以得到指向超出数组末端的下一位置的指针。通过指向第一个元素的指针 `words` 和指向数组中最后一个元素的下一位置的指针，实现了 `words2` 的初始化。其中第二个指针提供停止复制的条件，其所指向的位置上存放的元素并没有复制。

Allocating and Initializing a Specified Number of Elements

分配和初始化指定数目的元素

When creating a sequential container, we may specify an explicit size and an (optional) initializer to use for the elements. The size can be either a constant or non-constant expression. The element initializer must be a valid value that can be used to initialize an object of the element type:

创建顺序容器时，可显式指定容器大小和一个（可选的）元素初始化式。容器大小可以是常量或非常量表达式，元素初始化则必须是可用于初始化其元素类型的对象的值：

```
const list<int>::size_type list_size = 64;
list<string> slist(list_size, "eh?"); // 64 strings, each is eh?
```

This code initializes `slist` to have 64 elements, each with the value `eh?`.

这段代码表示 `slist` 含有 64 个元素，每个元素都被初始化为“`eh?`”字符串。

As an alternative to specifying the number of elements and an element initializer, we can also specify only the size:

创建容器时，除了指定元素个数，还可选择是否提供元素初始化式。我们也可以只指定容器大小：

```
list<int> ilist(list_size); // 64 elements, each initialized to 0
// svec has as many elements as the return value from get_word_count
extern unsigned get_word_count(const string &file_name);
vector<string> svec(get_word_count("Chimera"));
```

When we do not supply an element initializer, the library generates a value-initialized ([Section 3.3.1](#), p. 92) one for us. To use this form of initialization, the element type must either be a built-in or compound type or be a class type that has a default constructor. If the element type does not have a default constructor, then an explicit element initializer must be specified.

不提供元素初始化式时，标准库将为该容器实现值初始化（[3.3.1 s;节](#)）。采用这种类型的初始化，元素类型必须是内置或复合类型，或者是提供了默认构造函数的类类型。如果元素类型没有默认构造函数，则必须显式指定其元素初始化式。



The constructors that take a size are valid *only* for sequential containers; they are not supported for the associative containers,

接受容器大小做形参的构造函数只适用于顺序容器，而关联容器不支持这种初始化。

Exercises Section 9.1.1

Exercise Explain the following initializations. Indicate if any are in error, and if so, why.

9.1:

解释下列初始化，指出哪些是错误的，为什么？

```
int ia[7] = { 0, 1, 1, 2, 3, 5, 8 };
string sa[6] = {
    "Fort Sumter", "Manassas", "Perryville",
    "Vicksburg", "Meridian", "Chancellorsville" };
(a) vector<string> svec(sa, sa+6);
(b) list<int> ilist( ia+4, ia+6);
(c) vector<int> ivec(ia, ia+8);
(d) list<string> slist(sa+6, sa);
```

Exercise Show an example of each of the four ways to create and initialize a `vector`. Explain what values each `vector` contains.

创建和初始化一个 `vector` 对象有 4 种方式，为每种方式提供一个例子，并解释每个例子生成的 `vector` 对象包含什么值。

Exercise Explain the differences between the constructor that takes a container to copy and the constructor that takes two iterators.

解释复制容器对象的构造函数和使用两个迭代器的构造函数之间的差别。

9.1.2. Constraints on Types that a Container Can Hold

9.1.2. 容器内元素的类型约束

While most types can be used as the element type of a container, there are two constraints that element types must meet:

C++ 语言中，大多数类型都可用作容器的元素类型。容器元素类型必须满足以下两个约束：

- The element type must support assignment.
元素类型必须支持赋值运算。
- We must be able to copy objects of the element type.
元素类型的对象必须可以复制。

There are additional constraints on the types used as the key in an associative container, which we'll cover in [Chapter 10](#).

此外，关联容器的键类型还需满足其他的约束，我们将在[第十一章](#)介绍相关内容。

Most types meet these minimal element type requirements. All of the built-in or compound types, with the exception of references, can be used as the element type. References do not support assignment in its ordinary meaning, so we cannot have containers of references.

大多数类型满足上述最低限度的元素类型要求。除了引用类型外，所有内置或复合类型都可用做元素类型。引用不支持一般意义的赋值运算，因此没有元素是引用类型的容器。

With the exception of the IO library types (and the `auto_ptr` type, which we cover in [Section 17.1.9](#) (p. 702)), all the library types are valid container element types. In particular, containers themselves satisfy these requirements. We can define containers with elements that are themselves containers. Our `Sales_item` type also satisfies these requirements.

除输入输出 (IO) 标准库类型（以及[第 17.1.9 节](#)介绍的 `auto_ptr` 类型）之外，所有其他标准库类型都是有效的容器元素类型。特别地，容器本身也满足上述要求，因此，可以定义元素本身就是容器类型的容器。`Sales_item` 类型也满足上述要求。

The IO library types do not support copy or assignment. Therefore, we cannot have a container that holds objects of the IO types.

IO 库类型不支持复制或赋值。因此，不能创建存放 IO 类型对象的容器。

Container Operations May Impose Additional Requirements

容器操作的特殊要求

The requirement to support copy and assignment is the minimal requirement on element types. In addition, some container operations impose additional requirements on the element type. If the element type doesn't support the additional requirement, then we cannot perform that operation: We can define a container of that type but may not use that particular operation.

支持复制和赋值功能是容器元素类型的最低要求。此外，一些容器操作对元素类型还有特殊要求。如果元素类型不支持这些特殊要求，则相关的容器操作就不能执行：我们可以定义该类型的容器，但不能使用某些特定的操作。

One example of an operation that imposes a type constraint is the constructors that take a single initializer that specifies the size of the container. If our container holds objects of a class type, then we can use this constructor only if the element type has a default constructor. Most types do have a default constructor, although there are some classes that do not. As an example, assume that `Foo` is a class that does *not* define a default constructor but that does have a constructor that takes an `int` argument. Now, consider the following declarations:

其中一种需外加类型要求的容器操作是指定容器大小并提供单个初始化式的构造函数。如果容器存储类类型的对象，那么只有当其元素类型提供默认构造函数时，容器才能使用这种构造函数。尽管有一些类没有提供默认构造函数，但大多数类类型都会有。例如，假设类 `Foo` 没有默认构造函数，但提供了需要一个 `int` 型形参的构造函数。现在，考虑下面的声明：

```
vector<Foo> empty;      // ok: no need for element default constructor
vector<Foo> bad(10);    // error: no default constructor for Foo
vector<Foo> ok(10, 1);  // ok: each element initialized to 1
```

We can define an empty container to hold `Foo` objects, but we can define one of a given size only if we also specify an initializer for each element.

我们定义一个存放 `Foo` 类型对象的空容器，但是，只有在同时指定每个元素的初始化式时，才能使用给定容器大小的构造函数来创建同类型的容器对象。

As we describe the container operations, we'll note the constraints, if any, that each container operation places on the element type.

在描述容器操作时，我们应该留意（如果有的话）每个操作对元素类型的约束。

Containers of Containers

容器的容器

Because the containers meet the constraints on element types, we can define a container whose element type is itself a container type. For example, we might define `lines` as a `vector` whose elements are a `vector` of `strings`:

因为容器受容器元素类型的约束，所以可定义元素是容器类型的容器。例如，可以定义 `vector` 类型的容器 `lines`，其元素为 `string` 类型的 `vector` 对象：

```
// note spacing: use ">>" not ">>>" when specifying a container element type
vector< vector<string> > lines; // vector of vectors
```

Note the spacing used when specifying a container element type as a container:

注意，在指定容器元素为容器类型时，必须如下使用空格：

```
vector< vector<string> > lines; // ok: space required between close >
vector< vector<string>> lines; // error: >> treated as shift operator
```

必须用空格隔开两个相邻的 `>` 符号，以示这是两个分开的符号，否则，系统会认为 `>>` 是单个符号，为右移操作符，并导致编译时错误。



We must separate the two closing `>` symbols with a space to indicate that these two characters represent two symbols. Without the space, `>>` is treated as a single symbol, the right shift operator, and results in a compile-time error.

Exercises Section 9.1.2

Exercise 9.4: Define a `list` that holds elements that are `deque`s that hold `ints`.

定义一个 `list` 对象来存储 `deque` 对象里的元素，该 `deque` 对象存放 `int` 型元素。

Exercise 9.5: Why can we not have containers that hold `iostream` objects?

为什么我们不可以使用容器来存储 `iostream` 对象？

Exercise 9.6: Given a class type named `Foo` that does not define a default constructor but does define a constructor that takes `int` values, define a `list` of `Foo` that holds 10 elements.

假设有一个名为 `Foo` 的类，这个类没有定义默认构造函数，但提供了需要一个 `int` 型参数的构造函数，定义一个存放 `Foo` 的 `list` 对象，该对象有 10 个元素。

9.2. Iterators and Iterator Ranges

9.2. 迭代器和迭代器范围

The constructors that take a pair of iterators are an example of a common form used extensively throughout the library. Before we look further at the container operations, we should understand a bit more about iterators and iterator ranges.

在整个标准库中，经常使用形参为一对迭代器的构造函数。在深入探讨容器操作之前，先来了解一下迭代器和迭代器范围。

In [Section 3.4](#) (p. 95), we first encountered `vector` iterators. Each of the container types has several companion iterator types. Like the containers, the iterators have a common interface: If an iterator provides an operation, then the operation is supported in the same way for each iterator that supplies that operation. For example, all the container iterators let us read an element from a container, and they all do so by providing the dereference operator. Similarly, they all provide increment and decrement operators to allow us to go from one element to the next. [Table 9.3](#) lists the iterator operations supported by the iterators for all of the library containers.

[第 3.4 节](#)首次介绍了 `vector` 类型的迭代器。每种容器类型都提供若干共同工作的迭代器类型。与容器类型一样，所有迭代器具有相同的接口：如果某种迭代器支持某种操作，那么支持这种操作的其他迭代器也会以相同的方式支持这种操作。例如，所有容器迭代器都支持以解引用运算从容器中读入一个元素。类似地，容器都提供自增和自减操作符来支持从一个元素到下一个元素的访问。[表 9.3](#)列出迭代器为所有标准库容器类型所提供的运算。

Table 9.3. Common Iterator Operations

<code>*iter</code>	Return a reference to the element referred to by the iterator <code>iter</code> .
	返回迭代器 <code>iter</code> 所指向的元素的引用
<code>iter->mem</code>	Dereference <code>iter</code> and fetch the member named <code>mem</code> from the underlying element. Equivalent to <code>(*iter).mem</code> .
	对 <code>iter</code> 进行解引用，获取指定元素中名为 <code>mem</code> 的成员。等效于 <code>(*iter).mem</code>
<code>++iter</code> <code>iter++</code>	Increment <code>iter</code> to refer to the next element in the container.
	给 <code>iter</code> 加 1，使其指向容器里的下一个元素
<code>--iter</code> <code>iter--</code>	Decrement <code>iter</code> to refer to the previous element in the container.
	给 <code>iter</code> 减 1，使其指向容器里的前一个元素
<code>iter1 == iter2</code> <code>iter1 != iter2</code>	Compare two iterators for equality (inequality). Two iterators are equal if they refer to the same element of the same container or if they are the off-the-end iterator (Section 3.4 , p. 97) for the same container.
	比较两个迭代器是否相等（或不等）。当两个迭代器指向同一个容器中的同一个元素，或者当它们都指向同一个容器的超出末端的下一位位置时，两个迭代器相等

Iterators on `vector` and `deque` Support Additional Operations

`vector` 和 `deque` 容器的迭代器提供额外的运算

There are two important sets of operations that only `vector` and `deque` support: iterator arithmetic ([Section 3.4.1](#), p. 100) and the use of the relational operators (in addition to `==` and `!=`) to compare two iterators. These operations are summarized in [Table 9.4](#) on the facing page.

C++ 定义的容器类型中，只有 `vector` 和 `deque` 容器提供下面两种重要的运算集合：迭代器算术运算 ([第 3.4.1 节](#))，以及使用除了 `==` 和 `!=` 之外的关系操作符来比较两个迭代器 (`==` 和 `!=` 这两种关系运算适用于所有容器)。[表 9.4](#)总结了这些相关的操作符。

Table 9.4. Operations Supported by `vector` and `deque` Iterators

<code>iter + n</code>	Adding (subtracting) an integral value <code>n</code> to (from) an iterator yields an iterator that many elements forward (backward) within the container. The resulting iterator must refer to an element in the container or one past the end of the container.
	在迭代器上加（减）整数值 <code>n</code> ，将产生指向容器中前面（后面）第 <code>n</code> 个元素的迭代器。新计算出来的迭代器必须指向容器中的元素或超出容器末端的下一位位置
<code>iter1 += iter2</code>	Compound-assignment versions of iterator addition and subtraction. Assigns the value of adding or subtracting <code>iter1</code> and <code>iter2</code> into <code>iter1</code> .
<code>-=</code>	这里迭代器加减法的复合赋值运算：将 <code>iter1</code> 加上或减去 <code>iter2</code> 的运算结果赋给 <code>iter1</code>

Section 9.2. Iterators and Iterator Ranges

iter2

iter1 - iter2 Subtracting two iterators yields the number that when added to the right-hand iterator yields the left-hand iterator. The iterators must refer to elements in the same container or one past the end of the container.

两个迭代器的减法，其运算结果加上右边的迭代器即得左边的迭代器。这两个迭代器必须指向同一个容器中的元素或超出容器末端的下一位位置

Supported only for vector and deque.

只适用于 `vector` 和 `deque` 容器

>, >=,
<, <=

Relational operators on iterators. One iterator is less than another if it refers to an element whose position in the container is ahead of the one referred to by the other iterator. The iterators must refer to elements in the same container or one past the end of the container.

迭代器的关系操作符。当一个迭代器指向的元素在容器中位于另一个迭代器指向的元素之前，则前一个迭代器小于后一个迭代器。关系操作符的两个迭代器必须指向同一个容器中的元素或超出容器末端的下一位位置

Supported only for vector and deque.

只适用于 `vector` 和 `deque` 容器

The reason that only `vector` and `deque` support the relational operators is that only `vector` and `deque` offer fast, random access to their elements. These containers are guaranteed to let us efficiently jump directly to an element given its position in the container. Because these containers support random access by position, it is possible for their iterators to efficiently implement the arithmetic and relational operations.

关系操作符只适用于 `vector` 和 `deque` 容器，这是因为只有这种两种容器为其元素提供快速、随机的访问。它们确保可根据元素位置直接有效地访问指定的容器元素。这两种容器都支持通过元素位置实现的随机访问，因此它们的迭代器可以有效地实现算术和关系运算。

For example, we could calculate the midpoint of a `vector` as follows:

例如，下面的语句用于计算 `vector` 对象的中点位置：

```
vector<int>::iterator iter = vec.begin() + vec.size()/2;
```

On the other hand, this code

另一方面，代码：

```
// copy elements from vec into ilist
list<int> ilist(vec.begin(), vec.end());
ilist.begin() + ilist.size()/2; // error: no addition on list iterators
```

is an error. The `list` iterator does not support the arithmetic operations addition or subtraction nor does it support the relational (`<=`, `<`, `>=`, `>`) operators. It does support pre- and postfix increment and decrement and the equality (inequality) operators.

是错误的。`list` 容器的迭代器既不支持算术运算（加法或减法），也不支持关系运算（`<=`, `<`, `>=`, `>`），它只提供前置和后置的自增、自减运算以及相等（不等）运算。

In [Chapter 11](#) we'll see that the operations an iterator supports are fundamental to using the library algorithms.

[第十一章](#) 中，我们将会了解到迭代器提供运算是使用标准库算法的基础。

Exercises Section 9.2

Exercise What is wrong with the following program? How might you correct it?

9.7:

下面的程序错在哪里？如何改正。

```
list<int> lst1;
list<int>::iterator iter1 = lst1.begin(),
                    iter2 = lst1.end();
while (iter1 < iter2) /* . . . */
```

Exercise Assuming `vec_iter` is bound to an element in a `vector` that holds `strings`, what does this statement do?

Section 9.2. Iterators and Iterator Ranges

假设 `vec_iter` 与 `vector` 对象的一个元素捆绑在一起，该 `vector` 对象存放 `string` 类型的元素，请问下面的语句实现什么功能？

```
if (vec_iter->empty()) /* . . . */
```

Exercise 9.9: Write a loop to write the elements of a `list` in reverse order.

编写一个循环将 `list` 容器的元素逆序输出。

Exercise 9.10: Which, if any, of the following iterator uses are in error?

下列迭代器的用法哪些（如果有的话）是错误的？

```
const vector< int > ivec(10);
vector< string >    svec(10);
list< int >          ilist(10);

(a) vector<int>::iterator    it = ivec.begin();
(b) list<int>::iterator     it = ilist.begin() + 2;
(c) vector<string>::iterator it = &svec[0];
(d) for (vector<string>::iterator
        it = svec.begin(); it != 0; ++it)
            // ...
```

9.2.1. Iterator Ranges

9.2.1. 迭代器范围



The concept of an iterator range is fundamental to the standard library.

迭代器范围这个概念是标准库的基础。

An **iterator range** is denoted by a pair of iterators that refer to two elements, or to *one past the last element*, in the same container. These two iterators, often referred to as `first` and `last`, or `beg` and `end`, mark a range of elements from the container.

C++ 语言使用一对迭代器标记**迭代器范围 (iterator range)**，这两个迭代器分别指向同一个容器中的两个元素或超出末端的下一位置，通常将它们命名为 `first` 和 `last`，或 `beg` 和 `end`，用于标记容器中的一段元素范围。

Although the names `last` and `end` are common, they are a bit misleading. The second iterator never refers to the last element of the range. Instead, it refers to a point one past the last element. The elements in the range include the element referred to by `first` and every element from `first` through the element just before `last`. If the iterators are equal, then the range is empty.

尽管 `last` 和 `end` 这两个名字很常见，但是它们却容易引起误解。其实第二个迭代器从来都不是指向元素范围的最后一个元素，而是指向最后一个元素的下一位置。该范围内的元素包括迭代器 `first` 指向的元素，以及从 `first` 开始一直到迭代器 `last` 指向的位置之前的所有元素。如果两个迭代器相等，则迭代器范围为空。

This element range is called a **left-inclusive interval**. The standard notation for such a range is

此类元素范围称为**左闭合区间 (left-inclusive interval)**，其标准表示方式为：

```
// to be read as: includes first and each element up to but not including last
[ first, last )
```

Section 9.2. Iterators and Iterator Ranges

indicating that the range begins with `first` and ends with, but does not include, `last`. The iterator in `last` may be equal to the `first` or may refer to an element that appears after the one referred to by `first`. The `last` iterator must not refer to an element ahead of the one referred to by `first`.

表示范围从 `first` 开始, 到 `last` 结束, 但不包括 `last`。迭代器 `last` 可以等于 `first`, 或者指向 `first` 标记的元素后面的某个元素, 但绝对不能指向 `first` 标记的元素前面的元素。

Requirements on Iterators Forming an Iterator Range

对形成迭代器范围的迭代器的要求

Two iterators, `first` and `last`, form an iterator range, if

迭代器 `first` 和 `last` 如果满足以下条件, 则可形成一个迭代器范围:

- **They refer to elements of, or one-past-the-end of, the same container.**
它们指向同一个容器中的元素或超出末端的下一位。
- **If the iterators are not equal, then it must be possible to reach `last` by repeatedly incrementing `first`. In other words, `last` must not precede `first` in the container.**
如果这两个迭代器不相等, 则对 `first` 反复做自增运算必须能够到达 `last`。换句话说, 在容器中, `last` 绝对不能位于 `first` 之前。



The compiler cannot itself enforce these requirements. It does not know to which container an iterator is bound, nor does it know how many elements are in a container. Failing to meet these requirements results in undefined run-time behavior.

编译器自己不能保证上述要求。编译器无法知道迭代器所关联的是哪个容器, 也不知道容器内有多少个元素。若不能满足上述要求, 将导致运行时未定义的行为。

Programming Implications of Using Left-Inclusive Ranges

使用左闭合区间的编程意义

The library uses left-inclusive ranges because such ranges have two convenient properties. Assuming `first` and `last` denote a valid iterator range, then

因为左闭合区间有两个方便使用的性质, 所以标准库使用此区间。假设 `first` 和 `last` 标记了一个有效的迭代器范围, 于是:

1. When `first` equals `last`, the range is empty.
当 `first` 与 `last` 相等时, 迭代器范围为空;
2. When `first` is not equal to `last`, there is at least one element in the range, and `first` refers to the first element in that range. Moreover, we can advance `first` by incrementing it some number of times until `first == last`.
当 `first` 与不相等时, 迭代器范围内至少有一个元素, 而且 `first` 指向该区间中的第一元素。此外, 通过若干次自增运算可以使 `first` 的值不断增大, 直到 `first == last` 为止。

These properties mean that we can safely write loops such as the following to process a range of elements by testing the iterators:

这两个性质意味着程序员可以安全地编写如下的循环, 通过测试迭代器处理一段元素:

```
while (first != last) {
    // safe to use *first because we know there is at least one element
    ++first;
}
```

Assuming `first` and `last` form a valid iterator range, then we know that either `first == last`, in which case the loop is exited, or the range is non-empty and `first` refers to an element. Because the condition in the `while` handles the case where the range is empty, there is no need for a special case to handle an empty range. When the range is non-empty, the loop will execute at least once. Because the loop body increments `first`, we know the loop will eventually terminate. Moreover, if we are in the loop, then we know that `*first` is safe: It must refer to an element in the non-empty range between `first` and `last`.

假设 `first` 和 `last` 标记了一段有效的迭代器范围，于是我们知道要么 `first == last`，这是退出循环的情况；要么该区间非空，`first` 指向其第一个元素。因为 `while` 循环条件处理了空区间情况，所以对此无须再特别处理。当迭代器范围非空时，循环至少执行一次。由于循环体每次循环就给 `first` 加 1，因此循环必定会终止。而且在循环内可确保 `*first` 是安全的：它必然指向 `first` 和 `last` 之间非空区间内的某个特定元素。

Exercises Section 9.2.1

Exercise 9.11: What are the constraints on the iterators that form iterator ranges?

要标记出有效的迭代器范围，迭代器需要满足什么约束？

Exercise 9.12: Write a function that takes a pair of iterators and an `int` value. Look for that value in the range and return a `bool` indicating whether it was found.

编写一个函数，其形参是一对迭代器和一个 `int` 型数值，实现在迭代器标记的范围内寻找该 `int` 型数值的功能，并返回一个 `bool` 结果，以指明是否找到指定数据。

Exercise 9.13: Rewrite the program that finds a value to return an iterator that refers to the element. Be sure your function works correctly if the element does not exist.

重写程序，查找元素的值，并返回指向找到的元素的迭代器。确保程序在要寻找的元素不存在时也能正确工作。

Exercise 9.14: Using iterators, write a program to read a sequence of `strings` from the standard input into a `vector`. Print the elements in the `vector`.

使用迭代器编写程序，从标准输入设备读入若干 `string` 对象，并将它们存储在一个 `vector` 对象中，然后输出该 `vector` 对象中的所有元素。

Exercise 9.15: Rewrite the program from the previous exercise to use a `list`. List the changes you needed to change the container type.

用 `list` 容器类型重写习题 9.14 得到的程序，列出改变了容器类型后要做的修改。

9.2.2. Some Container Operations Invalidate Iterators

9.2.2. 使迭代器失效的容器操作

In the sections that follow, we'll see that some container operations change the internal state of a container or cause the elements in the container to be moved. Such operations **invalidate** all iterators that refer to the elements that are moved and may invalidate other iterators as well. Using an invalidated iterator is undefined, and likely to lead to the same kinds of problems as using a dangling pointer.

在后面的几节里，我们将看到一些容器操作会修改容器的内在状态或移动容器内的元素。这样的操作使所有指向被移动的元素的迭代器失效，也可能同时使其他迭代器失效。使用无效迭代器是没有定义的，可能会导致与悬垂指针相同的问题。

For example, each container defines one or more `erase` functions. These functions remove elements from the container. Any iterator that refers to an element that is removed has an invalid value. After all, the iterator was positioned on an element that no longer exists within the container.

例如，每种容器都定义了一个或多个 `erase` 函数。这些函数提供了删除容器元素的功能。任何指向已删除元素的迭代器都具有无效值，毕竟，该迭代器指向了容器中不再存在的元素。

When writing programs that use iterators, we must be aware of which operations can invalidate the iterators. It is a serious run-time error to use an iterator that has been invalidated.



使用迭代器编写程序时，必须留意哪些操作会使迭代器失效。使用无效迭代器将会导致严重的运行时错误。

There is no way to examine an iterator to determine whether it has been invalidated. There is no test we can perform to detect that it has gone bad. Any use of an invalidated iterator is likely to yield a run-time error, but there is no guarantee that the program will crash or otherwise make it easy to detect the problem.

无法检查迭代器是否有效，也无法通过测试来发现迭代器是否已经失效。任何无效迭代器的使用都可能导致运行时错误，但程序不一定会崩溃，否则检查这种错误也许会容易些。



When using iterators, it is usually possible to write the program so that the range of code over which an iterator must stay valid is relatively short. It is important to examine each statement in this range to determine whether elements are added or removed and adjust iterator values accordingly.

使用迭代器时，通常可以编写程序使得要求迭代器有效的代码范围相对较短。然后，在该范围内，严格检查每一条语句，判断是否有元素添加或删除，从而相应地调整迭代器的值。

9.3. Sequence Container Operations

9.3. 每种顺序容器都提供了一组有用的类型定义以及以下操作：

Each sequential container defines a set of useful typedefs and supports operations that let us

每种顺序容器都提供了一组有用的类型定义以及以下操作：

- Add elements to the container
在容器中添加元素。
- Delete elements from the container
在容器中删除元素。
- Determine the size of the container
设置容器大小。
- Fetch the first and last elements from the container, if any
(如果有的话) 获取容器内的第一个和最后一个元素。

9.3.1. Container Typedefs

9.3.1. 容器定义的类型别名

We've used three of the container-defined types: `size_type`, `iterator`, and `const_iterator`. Each container defines these types, along with several others shown in [Table 9.5](#).

在前面的章节里，我们已经使用过三种由容器定义的类型：`size_type`、`iterator` 和 `const_iterator`。所有容器都提供这三种类型以及[表 9.5](#) 所列出的其他类型。

Table 9.5. Container-Defined Typedefs

表 9.5. 容器定义的类型别名

<code>size_type</code>	Unsigned integral type large enough to hold size of largest possible container of this container type 无符号整型，足以存储此容器类型的最大可能容器长度
<code>iterator</code>	Type of the iterator for this container type 此容器类型的迭代器类型
<code>const_iterator</code>	Type of the iterator that can read but not write the elements 元素的只读迭代器类型
<code>reverse_iterator</code>	Iterator that addresses elements in reverse order 按逆序寻址元素的迭代器
<code>const_reverse_iterator</code>	Reverse iterator that can read but not write the elements 元素的只读(不能写)逆序迭代器
<code>difference_type</code>	Signed integral type large enough to hold the difference, which might be negative, between two

Section 9.3. Sequence Container Operations

	iterators
	足够存储两个迭代器差值的有符号整型，可为负数
value_type	Element type
	元素类型
reference	Element's lvalue type; synonym for <code>value_type&</code>
	元素的左值类型，是 <code>value_type&</code> 的同义词
const_reference	Element's const lvalue type; same as <code>const value_type&</code>
	元素的常量左值类型，等效于 <code>const value_type&</code>

We'll have more to say about reverse iterators in [Section 11.3.3](#) (p. 412), but briefly, a reverse iterator is an iterator that goes backward through a container and inverts the iterator operations: For example, saying `++` on a reverse iterator yields the previous element in the container.

我们将在[第 11.3.3 节](#)中详细介绍逆序迭代器。简单地说，逆序迭代器从后向前遍历容器，并反转了某些相关的迭代器操作：例如，在逆序迭代器上做 `++` 运算将指向容器中的前一个元素。

The last three types in [Table 9.5](#) on the facing page let us use the type of the elements stored in a container without directly knowing what that type is. If we need the element type, we refer to the container's `value_type`. If we need a reference to that type, we use `reference` or `const_reference`. The utility of these element-related typedefs will be more apparent when we define our own generic programs in [Chapter 16](#).

[表 9.5](#) 的最后三种类型使程序员无须直接知道容器元素的真正类型，就能使用它。需要使用元素类型时，只要用 `value_type` 即可。如果要引用该类型，则通过 `reference` 和 `const_reference` 类型实现。在程序员编写自己的泛型程序（[第十六章](#)）时，这些元素相关类型的定义非常有用。

Expressions that use a container-defined type can look intimidating:

使用容器定义类型的表达式看上去非常复杂：

```
// iter is the iterator type defined by list<string>
list<string>::iterator iter;

// cnt is the difference_type type defined by vector<int>
vector<int>::difference_type cnt;
```

The declaration of `iter` uses the scope operator to say that we want the name on the right-hand side of the `::` from the scope of the left-hand side. The effect is to declare that `iter` has whatever type is defined for the `iterator` member from the `list` class that holds elements of type `string`.

`iter` 所声明使用了作用域操作符，以表明此时所使用的符号 `::` 右边的类型名字是在符号 `iter` 左边指定容器的作用域内定义的。其效果是将 `iter` 声明为 `iterator` 类型，而 `iterator` 是存放 `string` 类型元素的 `list` 类的成员。

Exercises Section 9.3.1

Exercise 9.16: What type should be used as the index into a `vector` of `ints`?

`int` 型的 `vector` 容器应该使用什么类型的索引？

Exercise 9.17: What type should be used to read the elements in a `list` of `strings`?

读取存放 `string` 对象的 `list` 容器时，应该使用什么类型？

9.3.2. `begin` and `end` Members

9.3.2. `begin` 和 `end` 成员

The `begin` and `end` operations yield iterators that refer to the first and one past the last element in the container. These iterators are most often used to form an iterator range that encompasses all the elements in the container.

`begin` 和 `end` 操作产生指向容器内第一个元素和最后一个元素的下一位置的迭代器，如表 9.6 所示。这两个迭代器通常用于标记包含容器中所有元素的迭代器范围。

Table 9.6. Container `begin` and `end` Operations

表 9.6. 容器的 `begin` 和 `end` 操作

<code>c.begin()</code>	Yields an iterator referring to the first element in <code>c</code>
	返回一个迭代器，它指向容器 <code>c</code> 的第一个元素
<code>c.end()</code>	Yields an iterator referring to the one past the last element in <code>c</code>
	返回一个迭代器，它指向容器 <code>c</code> 的最后一个元素的下一位置
<code>c.rbegin()</code>	Yields a reverse iterator referring to the last element in <code>c</code>
	返回一个逆序迭代器，它指向容器 <code>c</code> 的最后一个元素
<code>c.rend()</code>	Yields a reverse iterator referring one past (i.e., before) the first element in <code>c</code>
	返回一个逆序迭代器，它指向容器 <code>c</code> 的第一个元素前面的位置

There are two different versions of each of these operations: One is a `const` member (Section 7.7.1, p. 260) and the other is non`const`. The return type of these operations varies on whether the container is `const`. In each case, if the container is non`const`, then the result's type is `iterator` or `reverse_iterator`. If the object is `const`, then the type is prefixed by `const_`, that is, `const_iterator` or `const_reverse_iterator`. We cover reverse iterators in Section 11.3.3 (p. 412).

上述每个操作都有两个不同版本：一个是 `const` 成员（第 7.7.1 节），另一个是非 `const` 成员。这些操作返回什么类型取决于容器是否为 `const`。如果容器不是 `const`，则这些操作返回 `iterator` 或 `reverse_iterator` 类型。如果容器是 `const`，则其返回类型要加上 `const_` 前缀，也就是 `const_iterator` 和 `const_reverse_iterator` 类型。我们将在第 11.3.3 节中详细介绍逆序迭代器。

9.3.3. Adding Elements to a Sequential Container

9.3.3. 在顺序容器中添加元素

In Section 3.3.2 (p. 94) we saw one way to add elements: `push_back`. Every sequential container supports `push_back`, which appends an element to the back of the container. The following loop reads one `string` at a time into `text_word`:

第 3.3.2 节介绍了添加元素的一种方法：`push_back`。所有顺序容器都支持 `push_back` 操作（表 9.7），提供在容器尾部插入一个元素的功能。下面的循环每次读入一个 `string` 类型的值，并存放在 `text_word` 对象中：

```
// read from standard input putting each word onto the end of container
string text_word;
while (cin >> text_word)
    container.push_back(text_word);
```

The call to `push_back` creates a new element at the end of `container`, increasing the `size` of `container` by one. The value of that element is a copy of `text_word`. The type of `container` can be any of `list`, `vector`, or `deque`.

调用 `push_back` 函数会在容器 `container` 尾部创建一个新元素，并使容器的长度加 1。新元素的值为 `text_word` 对象的副本，而 `container` 的类型则可能是 `list`、`vector` 或 `deque`。

In addition to `push_back`, the `list` and `deque` containers support an analogous operation named `push_front`. This operation inserts a new element at the front of the container. For example,

除了 `push_back` 运算，`list` 和 `deque` 容器类型还提供了类似的操作：`push_front`。这个操作实现在容器首部插入新元素的功能。例如：

```
list<int> ilist;
// add elements at the end of ilist
for (size_t ix = 0; ix != 4; ++ix)
    ilist.push_back(ix);
```

uses `push_back` to add the sequence 0, 1, 2, 3 to the end of `ilist`.

Section 9.3. Sequence Container Operations

使用 `push_back` 操作在容器 `ilist` 尾部依次添加元素 0、1、2、3。

Alternatively, we could use `push_front`

然后，我们选择用 `push_front` 操作再次在 `ilist` 中添加元素：

```
// add elements to the start of ilist
for (size_t ix = 0; ix != 4; ++ix)
    ilist.push_front(ix);
```

to add the elements 0, 1, 2, 3 to the beginning of `ilist`. Because each element is inserted at the new beginning of the `list`, they wind up in reverse order. After executing both loops, `ilist` holds the sequence 3,2,1,0,0,1,2,3.

此时，元素 0、1、2、3 则被依次添加在 `ilist` 的开始位置。由于每个元素都在 `list` 的新起点插入，因此它们在容器中以逆序排列，循环结束后，`ilist` 内的元素序列为：3、2、1、0、0、1、2、3。

Key Concept: Container Elements Are Copies

关键概念：容器元素都是副本

When we add an element to a container, we do so by copying the element value into the container. Similarly, when we initialize a container by providing a range of elements, the new container contains copies of the original range of elements. There is no relationship between the element in the container and the value from which it was copied. Subsequent changes to the element in the container have no effect on the value that was copied, and vice versa.

在容器中添加元素时，系统是将元素值复制到容器里。类似地，使用一段元素初始化新容器时，新容器存放的是原始元素的副本。被复制的原始值与新容器中的元素各不相关，此后，容器内元素值发生变化时，被复制的原值不会受到影响，反之亦然。

Table 9.7. Operations that Add Elements to a Sequential Container

表 9.7 在顺序容器中添加元素的操作

<code>c.push_back(t)</code>	Adds element with value <code>t</code> to the end of <code>c</code> . Returns <code>void</code> . 在容器 <code>c</code> 的尾部添加值为 <code>t</code> 的元素。返回 <code>void</code> 类型
<code>c.push_front(t)</code>	Adds element with value <code>t</code> to front of <code>c</code> . Returns <code>void</code> . 在容器 <code>c</code> 的前端添加值为 <code>t</code> 的元素。返回 <code>void</code> 类型 Valid only for list or deque. 只适用于 <code>list</code> 和 <code>deque</code> 容器类型。
<code>c.insert(p,t)</code>	Inserts element with value <code>t</code> before the element referred to by iterator <code>p</code> . Returns an iterator referring to the element that was added. 在迭代器 <code>p</code> 所指向的元素前面插入值为 <code>t</code> 的新元素。返回指向新添加元素的迭代器
<code>c.insert(p,n,t)</code>	Inserts <code>n</code> elements with value <code>t</code> before the element referred to by iterator <code>p</code> . Returns <code>void</code> . 在迭代器 <code>p</code> 所指向的元素前面插入 <code>n</code> 个值为 <code>t</code> 的新元素。返回 <code>void</code> 类型
<code>c.insert(p,b,e)</code>	Inserts elements in the range denoted by iterators <code>b</code> and <code>e</code> before the element referred to by iterator <code>p</code> . Returns <code>void</code> . 在迭代器 <code>p</code> 所指向的元素前面插入由迭代器 <code>b</code> 和 <code>e</code> 标记的范围内的元素。 返回 <code>void</code> 类型

Adding Elements at a Specified Point in the Container

Section 9.3. Sequence Container Operations

在容器中指定位置添加元素

The `push_back` and `push_front` operations provide convenient ways to insert a single element at the end or beginning of a sequential container. More generally, `insert` allows us to insert elements at any particular point in the container. There are three versions of `insert`. The first takes an iterator and an element value. The iterator refers to the position at which to insert the value. We could use this version of `insert` to insert an element at the beginning of a container:

使用 `push_back` 和 `push_front` 操作可以非常方便地在顺序容器的尾部或首部添加单个元素。而 `insert` 操作则提供了一组更通用的插入方法，实现在容器的任意指定位置插入新元素。`insert` 操作有三个版本（表 9.7）。第一个版本需要一个迭代器和一个元素值参数，迭代器指向插入新元素的位置。下面的程序就是使用了这个版本的 `insert` 函数在容器首部插入新元素：

```
vector<string> svec;
list<string> slist;
string spouse("Beth");

// equivalent to calling slist.push_front (spouse);
slist.insert(slist.begin(), spouse);

// no push_front on vector but we can insert before begin()
// warning: inserting anywhere but at the end of a vector is an expensive operation
svec.insert(svec.begin(), spouse);
```

The value is inserted *before* the position referred to by the iterator. The iterator can refer to any position in the container, including one past the end of the container. Because the iterator might refer to a nonexistent element off the end of the container, `insert` inserts before the position rather than after it. This code

新元素是插入在迭代器指向的位置之前。迭代器可以指向容器的任意位置，包括超出末端的下一位置。由于迭代器可能指向超出容器末端的下一位置，这是一个不存在的元素，因此 `insert` 函数是在其指向位置之前而非其后插入元素。代码

```
slist.insert(iter, spouse); // insert spouse just before iter
```

inserts a copy of `spouse` just before the element referred to by `iter`.

就在 `iter` 指向的元素前面插入 `spouse` 的副本。

This version of `insert` returns an iterator referring to the newly inserted element. We could use the return value to repeatedly insert elements at a specified position in the container:

这个版本的 `insert` 函数返回指向新插入元素的迭代器。可使用该返回值在容器中的指定位置重复插入元素：

```
list<string> lst;
list<string>::iterator iter = lst.begin();
while (cin >> word)
    iter = lst.insert(iter, word); // same as calling push_front
```



要彻底地理解上述循环是如何执行的，这一点非常重要——特别是要明白我们为什么说上述循环等效于调用 `push_front` 函数。

Before the loop, we initialize `iter` to `lst.begin()`. Because the `list` is empty, `lst.begin()` and `lst.end()` are equal, so `iter` refers one past the end of the (empty) `list`. The first call to `insert` puts the element we just read in front of the element referred to by `iter`. The value returned by `insert` is an iterator referring to this new element, which is now the first, and only, element in `lst`. We assign that iterator to `iter` and repeat the `while`, reading another word. As long as there are words to insert, each trip through the `while` inserts a new element ahead of `iter` and reassigns to `iter` the value of the newly inserted element. That element is always the first element, so each iteration inserts an element ahead of the first element in the `list`.

循环前，将 `iter` 初始化为 `lst.begin()`。此时，由于该 `list` 对象是空的，因此 `lst.begin()` 与 `lst.end()` 相等，于是 `iter` 指向该（空）容器的超出末端的下一位置。第一次调用 `insert` 函数时，将刚读入的元素插入到 `iter` 所指向位置的前面，容器 `lst` 得到第一个也是唯一的元素。然后 `insert` 函数返回指向这个新元素的迭代器，并赋给 `iter`，接着重复 `while` 循环，读入下一个单词。只要有单词要插入，每次 `while` 循环都将新元素插入到 `iter` 前面，然后重置 `iter` 指向新插入元素。新插入的元素总是容器中的第一个元素，因此，每次迭代器都将元素插入在该 `list` 对象的第一元素前面。

Inserting a Range of Elements

插入一段元素

A second form of `insert` adds a specified number of identical elements at an indicated position:

`insert` 函数的第二个版本提供在指定位置插入指定数量的相同元素的功能:

```
svec.insert(svec.end(), 10, "Anna");
```

This code inserts ten elements at the end of `svec` and initializes each of those elements to the string "Anna".

上述代码在容器 `svec` 的尾部插入 10 个元素，每个新元素都初始化为 "Anna"。

The final form of `insert` adds a range of elements denoted by an iterator pair into the container. For example, given the following array of `strings`

`insert` 函数的最后一个版本实现在容器中插入由一对迭代器标记的一段范围内的元素。例如，给出以下 `string` 类型的数组:

```
string sarray[4] = {"quasi", "simba", "frollo", "scar"};
```

we can insert all or a subset of the array elements into our `list` of `strings`:

可将该数组中所有的或其中一部分元素插入到 `string` 类型的 `list` 容器中:

```
// insert all the elements in sarray at end of slist
slist.insert(slist.end(), sarray, sarray+4);
list<string>::iterator slist_iter = slist.begin();
// insert last two elements of sarray before slist_iter
slist.insert(slist_iter, sarray+2, sarray+4);
```

Inserting Elements Can Invalidate Iterators

添加元素可能会使迭代器失效

As we'll see in [Section 9.4](#) (p. 330), adding elements to a `vector` can cause the entire container to be relocated. If the container is relocated, then all iterators into the container are invalidated. Even if the `vector` does not have to be relocated, any iterator to an element after the one inserted is invalidated.

正如我们在[第 9.4 节](#)中了解的一样，在 `vector` 容器中添加元素可能会导致整个容器的重新加载，这样的话，该容器涉及的所有迭代器都会失效。即使需要重新加载整个容器，指向新插入元素后面的那个元素的迭代器也会失效。



Iterators may be invalidated after doing any `insert` or `push` operation on a `vector` or `deque`. When writing loops that `insert` elements into a `vector` or a `deque`, the program must ensure that the iterator is refreshed on each trip through the loop.

任何 `insert` 或 `push` 操作都可能导致迭代器失效。当编写循环将元素插入到 `vector` 或 `deque` 容器中时，程序必须确保迭代器在每次循环后都得到更新。

Avoid Storing the Iterator Returned from `end`

避免存储 `end` 操作返回的迭代器

When we add elements to a `vector` or `deque`, some or all of the iterators may be invalidated. It is safest to assume that all iterators are invalid. This advice is particularly true for the iterator returned by `end`. That iterator is *always* invalidated by any insertion anywhere in the container.

在 `vector` 或 `deque` 容器中添加元素时，可能会导致某些或全部迭代器失效。假设所有迭代器失效是最安全的做法。这个建议特别适用于由 `end` 操作返回的迭代器。在容器的任何位置插入任何元素都会使该迭代器失效。

As an example, consider a loop that reads each element, processes it and adds a new element following the original. We want the loop to process

Section 9.3. Sequence Container Operations

each original element. We'll use the form of `insert` that takes a single value and returns an iterator to the element that was just inserted. After each insertion, we'll increment the iterator that is returned so that the loop is positioned to operate on the next original element. If we attempt to "optimize" the loop, by storing an iterator to the `end()`, we'll have a disaster:

```
vector<int>::iterator first = v.begin(),
    last = v.end(); // cache end iterator
// disaster: behavior of this loop is undefined
while (first != last) {
    // do some processing
    // insert new value and reassign first, which otherwise would be invalid
    first = v.insert(first, 42);
    ++first; // advance first just past the element we added
}
```

The behavior of this code is undefined. On many implementations, we'll get an infinite loop. The problem is that we stored the value returned by the `end` operation in a local variable named `last`. In the body of the loop, we add an element. Adding an element invalidates the iterator stored in `last`. That iterator neither refers to an element in `v` nor any longer refers to one past the last element in `v`.

上述代码的行为未定义。在很多实现中，该段代码将导致死循环。问题在于这个程序将 `end` 操作返回的迭代器值存储在名为 `last` 的局部变量中。循环体中实现了元素的添加运算，添加元素会使得存储在 `last` 中的迭代器失效。该迭代器既没有指向容器 `v` 的元素，也不再指向 `v` 的超出末端的下一位置。



Don't cache the iterator returned from `end`. Inserting or deleting elements in a `deque` or `vector` will invalidate the cached iterator.

不要存储 `end` 操作返回的迭代器。添加或删除 `deque` 或 `vector` 容器内的元素都会导致存储的迭代器失效。

Rather than storing the `end` iterator, we must recompute it after each insertion:

为了避免存储 `end` 迭代器，可以在每次做完插入运算后重新计算 `end` 迭代器值：

```
// safer: recalculate end on each trip whenever the loop adds/erases elements
while (first != v.end()) {
    // do some processing
    first = v.insert(first, 42); // insert new value
    ++first; // advance first just past the element we added
}
```

Exercises Section 9.3.3

Exercise

9.18:

Write a program to copy elements from a `list` of `ints` into two `deques`. The `list` elements that are even should go into one `deque` and those that are odd into the second.

编写程序将 `int` 型的 `list` 容器的所有元素复制到两个 `deque` 容器中。`list` 容器的元素如果为偶数，则复制到一个 `deque` 容器中；如果为奇数，则复制到另一个 `deque` 容器里。

Exercise

9.19:

Assuming `iv` is a `vector` of `ints`, what is wrong with the following program? How might you correct the problem(s)?

假设 `iv` 是一个 `int` 型的 `vector` 容器，下列程序存在什么错误？如何改正之。

```
vector<int>::iterator mid = iv.begin() + iv.size()/2;
while (vector<int>::iterator iter != mid)
    if (iter == some_val)
        iv.insert(iter, 2 * some_val);
```

9.3.4. Relational Operators

9.3.4. 关系操作符

Each container supports the relational operators ([Section 5.2](#), p. 152) that can be used to compare two containers. The containers must be the same kind of container and must hold elements of the same type. We can compare a `vector<int>` only with another `vector<int>`. We cannot compare a `vector<int>` with a `list<int>` or a `vector<double>`.

所有的容器类型都支持用关系操作符（[第 5.2 节](#)）来实现两个容器的比较。显比较的容器必须具有相同的容器类型，而且其元素类型也必须相同。例如，`vector<int>` 容器只能与 `vector<int>` 容器比较，而不能与 `list<int>` 或 `vector<int>` 容器比较，而不能与 `list<int>` 或 `vector<double>` 类型的容器比较。

Comparing two containers is based on a pairwise comparison of the elements of the containers. The comparison uses the same relational operator as defined by the element type: Comparing two containers using `!=` uses the `!=` operator for the element type. If the element type doesn't support the operator, then the containers cannot be compared using that operator.

容器的比较是基于容器内元素的比较。容器的比较使用了元素类型定义的同一个关系操作符：两个容器做 `!=` 比较使用了其元素类型定义的 `!=` 操作符。如果容器的元素类型不支持某种操作符，则该容器就不能做这种比较运算。

These operators work similarly to the `string` relationals ([Section 3.2.3](#), p. 85):

下面的操作类似于 `string` 类型的关系运算（[第 3.2.3 节](#)）：

- If both containers are the same size and all the elements are equal, then the two containers are equal; otherwise, they are unequal.
如果两个容器具有相同的长度而且所有元素都相等，那么这两个容器就相等；否则，它们就不相等。
- If the containers have different sizes but every element of the shorter one is equal to the corresponding element of the longer one, then the shorter one is considered to be less than the other.
如果两个容器的长度不相同，但较短的容器中所有元素都等于较长容器中对应的元素，则称较短的容器小于另一个容器。
- If neither container is an initial subsequence of the other, then the comparison depends on comparing the first unequal elements.
如果两个容器都不是对文的初始子序列，则它们的比较结果取决于所比较的第一个不相等的元素。

The easiest way to understand these operators is by studying examples:

理解上述操作的最简单方法是研究例程：

```

/*
    ivec1: 1 3 5 7 9 12
    ivec2: 0 2 4 6 8 10 12
    ivec3: 1 3 9
    ivec4: 1 3 5 7
    ivec5: 1 3 5 7 9 12
*/
// ivec1 and ivec2 differ at element[0]: ivec1 greater than ivec2
ivec1 < ivec2 // false
ivec2 < ivec1 // true

// ivec1 and ivec3 differ at element[2]: ivec1 less than ivec3
ivec1 < ivec3 // true

// all elements equal, but ivec4 has fewer elements, so ivec1 is greater than ivec4
ivec1 < ivec4 // false

ivec1 == ivec5 // true; each element equal and same number of elements
ivec1 == ivec4 // false; ivec4 has fewer elements than ivec1
ivec1 != ivec4 // true; ivec4 has fewer elements than ivec1

```

Relational Operators Use Their Element's Relational Operator

使用元素提供的关系操作符实现容器的关系运算



We can compare two containers only if the same relational operator defined for the element types.

C++ 语言只允许两个容器做其元素类型定义的关系运算。

Each container relational operator executes by comparing pairs of elements from the two containers:

所有容器都通过比较其元素对来实现关系运算:

```
ivec1 < ivec2
```

Assuming `ivec1` and `ivec2` are `vector<int>`, then this comparison uses the built-in `int` less-than operator. If the `vectors` held `strings`, then the `string` less-than operator would be used.

假设 `ivec1` 和 `ivec2` 都是 `vector<int>` 类型的容器，则上述比较使用了内置 `int` 型定义的小于操作符。如果这两个 `vector` 容器存储的是 `strings` 对象，则使用 `string` 类型的小于操作符。

If the `vectors` held objects of the `Sales_item` type that we used in [Section 1.5](#) (p. 20), then the comparison would be illegal. We did not define the relational operators for `Sales_item`. If we have two containers of `Sales_items`, we could not compare them:

如果上述 `vector` 容器存储 [第 1.5 节](#) 定义的 `Sales_item` 类型的对象，则该比较运算不合法。因为 `Sales_item` 类型没有定义关系运算，所以不能比较存放 `Sales_items` 对象的容器：

```
vector<Sales_item> storeA;
vector<Sales_item> storeB;

if (storeA < storeB) // error: Sales_item has no less-than operator
```

Exercises Section 9.3.4

Exercise 9.20: Write a program to compare whether a `vector<int>` contains the same elements as a `list<int>`.

编写程序判断一个 `vector<int>` 容器所包含的元素是否与一个 `list<int>` 容器的完全相同。

Exercise 9.21: Assuming `c1` and `c2` are containers, what constraints does the following usage place on the element types in `c1` and `c2`?

假设 `c1` 和 `c2` 都是容器，下列用法给 `c1` 和 `c2` 的元素类型带来什么约束？

```
if (c1 < c2)
```

What, if any, constraints are there on `c1` and `c2`?

(如果有的话) 对 `c1` 和 `c2` 的约束又是什么？

9.3.5. Container Size Operations

9.3.5. 容器大小的操作

Each container type supports four size-related operations. We used `size` and `empty` in [Section 3.2.3](#) (p. 83): `size` returns the number of elements in the container; `empty` returns a `bool` that is `true` if `size` is zero and `false` otherwise.

所有容器类型都提供四种与容器大小相关的操作 ([表 9.8](#)) 第 3.2.3 节已经使用了 `size` 和 `empty` 函数：`size` 操作返回容器内元素的个数；`empty` 操作则返回一个布尔值，当容器的大小为 0 时，返回值为 `true`，否则为 `false`。

Table 9.8. Sequential Container Size Operations

表 9.8. 顺序容器的大小操作

<code>c.size()</code>	Returns the number of elements in <code>c</code> . Return type is <code>c::size_type</code> .
	返回容器 <code>c</code> 中的元素个数。返回类型为 <code>c::size_type</code>
<code>c.max_size()</code>	Returns maximum number of elements <code>c</code> can contain. Return type is <code>c::size_type</code> .
	返回容器 <code>c</code> 可容纳的最大元素个数，返回类型为 <code>c::size_type</code>
<code>c.empty()</code>	Returns a <code>bool</code> that indicates whether <code>size</code> is 0 or not.
	返回标记容器大小是否为 0 的布尔值
<code>c.resize(n)</code>	Resize <code>c</code> so that it has <code>n</code> elements. If <code>n < c.size()</code> , the excess elements are discarded. If new elements must be added, they are value initialized.
	调整容器 <code>c</code> 的长度大小，使其能容纳 <code>n</code> 个元素，如果 <code>n < c.size()</code> ，则删除多出来的元素；否则，添加采用值初始化的新元素
<code>c.resize(n, t)</code>	Resize <code>c</code> to have <code>n</code> elements. Any elements added have value <code>t</code> .
	调整容器 <code>c</code> 的长度大小，使其能容纳 <code>n</code> 个元素。所有新添加的元素值都为 <code>t</code>

The `resize` operation changes the number of elements in the container. If the current size is greater than the new size, then elements are deleted from the back of the container. If the current size is less than the new size, then elements are added to the back of the container:

容器类型提供 `resize` 操作来改变容器所包含的元素个数。如果当前的容器长度大于新的长度值，则该容器后部的元素会被删除；如果当前的容器长度小于新的长度值，则系统会在该容器后部添加新元素：

```
list<int> ilist(10, 42); // 10 ints: each has value 42
ilist.resize(15); // adds 5 elements of value 0 to back of ilist
ilist.resize(25, -1); // adds 10 elements of value -1 to back of ilist
ilist.resize(5); // erases 20 elements from the back of ilist
```

The `resize` operation takes an optional element-value argument. If this argument is present, then any newly added elements receive this value. If this argument is absent, then any new elements are value initialized ([Section 3.3.1](#), p. 92).

`resize` 操作可带有一个可选的元素值形参。如果在调用该函数时提供了这个参数，则所有新添加的元素都初始化为这个值。如果没有这个参数，则新添加的元素采用值初始化（[第 3.3.1 节](#)）。



`resize` can invalidate iterators. A `resize` operation on a `vector` or `deque` potentially invalidates all iterators.

`resize` 操作可能会使迭代器失效。在 `vector` 或 `deque` 容器上做 `resize` 操作有可能会使其所有的迭代器都失效。

For any container type, if `resize` shrinks the container, then any iterator to an element that is deleted is invalidated.

对于所有的容器类型，如果 `resize` 操作压缩了容器，则指向已删除的元素迭代器失效。

Exercises Section 9.3.5

Exercise 9.22: Given that `vec` holds 25 elements, what does `vec.resize(100)` do? What if we next wrote `vec.resize(10)`?

已知容器 `vec` 存放了 25 个元素，那么 `vec.resize(100)` 操作实现了什么功能？若再做操作 `vec.resize(10)`，实现的是什么功能？

Exercise 9.23: What, if any, restrictions does using `resize` with a single size argument place on the element types?

使用只带有一个长度参数的 `resize` 操作对元素类型有什么要求（如果有的话）？

9.3.6. Accessing Elements

9.3.6. 访问元素

If a container is not empty, then the `front` and `back` members return references bound to the first or last elements in the container:

如果容器非空，那么容器类型的 `front` 和 `back` 成员（表 9.9）将返回容器内第一个或最后一个元素的引用：

```
// check that there are elements before dereferencing an iterator
// or calling front or back
if (!ilist.empty()) {
    // val and val2 refer to the same element
    list<int>::reference val = *ilist.begin();
    list<int>::reference val2 = ilist.front();

    // last and last2 refer to the same element
    list<int>::reference last = *--ilist.end();
    list<int>::reference last2 = ilist.back(); }
```

Table 9.9. Operations to Access Elements in a Sequential Container

表 9.9. 访问顺序容器内元素的操作

<code>c.back()</code>	Returns a reference to the last element in <code>c</code> . Undefined if <code>c</code> is empty.
	返回容器 <code>c</code> 的最后一个元素的引用。如果 <code>c</code> 为空，则该操作未定义
<code>c.front()</code>	Returns a reference to the first element in <code>c</code> . Undefined if <code>c</code> is empty.
	返回容器 <code>c</code> 的第一个元素的引用。如果 <code>c</code> 为空，则该操作未定义
<code>c[n]</code>	Returns a reference to the element indexed by <code>n</code> .
	返回下标为 <code>n</code> 的元素的引用
	Undefined if <code>n < 0</code> or <code>n >= c.size()</code> .
	如果 <code>n < 0</code> 或 <code>n >= c.size()</code> ，则该操作未定义
	Valid only for vector and deque.
	只适用于 <code>vector</code> 和 <code>deque</code> 容器
<code>c.at(n)</code>	Returns a reference to the element indexed by <code>n</code> . If index is out of range, throws <code>out_of_range</code> exception.
	返回下标为 <code>n</code> 的元素的引用。如果下标越界，则该操作未定义
	Valid only for vector and deque.
	只适用于 <code>vector</code> 和 <code>deque</code> 容器

This program uses two different approaches to fetch a reference to the first and last elements in `ilist`. The direct approach is to call `front` or `back`. Indirectly, we can obtain a reference to the same element by dereferencing the iterator returned by `begin` or the element one before the iterator returned by `end`. Two things are noteworthy in this program: The `end` iterator refers "one past the end" of the container so to fetch the last element we must first decrement that iterator. The other important point is that before calling `front` or `back` or dereferencing the iterators from `begin` or `end` we check that `ilist` isn't empty. If the list were empty all of the operations in the `if` would be undefined.

Section 9.3. Sequence Container Operations

这段程序使用了两种不同的方法获取时 `ilist` 中的第一个和最后一个元素的引用。直接的方法是调用 `front` 或 `back` 函数。间接的方法是，通过对 `begin` 操作返回的迭代器进行解引用，或对 `end` 操作返回的迭代器的前一个元素位置进行解引用，来获取对同一元素的引用。在这段程序中，有两个地方值得注意：`end` 迭代器指向容器的超出末端的下一位置，因此必须先对其减 1 才能获取最后一个元素；另一点是，在调用 `front` 或 `back` 函数之前，或者在对 `begin` 或 `end` 返回的迭代器进行解引用运算之前，必须保证 `ilist` 容器非空。如果该 `list` 容器为空，则 `if` 语句内所有的操作都没有定义。

When we introduced subscripting in [Section 3.3.2](#) (p. 94), we noted that the programmer must ensure that an element exists at the indicated subscript location. The subscript operator itself does not check. The same caution applies to using the `front` or `back` operations. If the container is empty, these operations yield an undefined result. If the container has only one element, then `front` and `back` each return a reference to that element.

[第 3.3.2 节](#)介绍了下标运算，我们注意到程序员必须保证在指定下标位置上的元素确实存在。下标操作符本身不会做相关的检查。使用 `front` 或 `back` 运算时，必须注意同样的问题。如果容器为空，那么这些操作将产生未定义的结果。如果容器内只有一个元素，则 `front` 和 `back` 操作都返回对该元素的引用。



Using a subscript that is out-of-range or calling `front` or `back` on an empty container are serious programming errors.

使用越界的下标，或调用空容器的 `front` 或 `back` 函数，都会导致程序出现严重的错误。

An alternative to subscripting is to use the `at` member. This function acts like the subscript operation but if the index is invalid, `at` throws an `out_of_range` exception ([Section 6.13](#), p. 215):

```
vector<string> svec;           // empty vector
cout << svec[0];               // run-time error: There are no elements in svec!
cout << svec.at(0);            // throws out_of_range exception
```

Exercises Section 9.3.6

Exercise 9.24: Write a program that fetches the first element in a `vector`. Do so using `at`, the subscript operator, `front`, and `begin`. Test the program on an empty `vector`.

编写程序获取 `vector` 容器的第一个元素。分别使用下标操作符、`front` 函数以及 `begin` 函数实现该功能，并提供空的 `vector` 容器测试你的程序。

9.3.7. Erasing Elements

9.3.7. 删 除 元 素

Recall that there is both a general `insert` operation that inserts anywhere in the container and specific `push_front` and `push_back` operations to add elements only at the front or back. Similarly, there is a general `erase` and specific `pop_front` and `pop_back` operations to remove elements.

回顾前面的章节，我们知道容器类型提供了通用的 `insert` 操作在容器的任何位置插入元素，并支持特定的 `push_front` 和 `push_back` 操作在容器首部或尾部插入新元素。类似地，容器类型提供了通用的 `erase` 操作和特定的 `pop_front` 和 `pop_back` 操作来删除容器内的元素 ([表 9.10](#))。

Table 9.10. Operations to Remove Elements from a Sequential Container

表 9.10. 删 除 顺 序 容 器 内 元 素 的 操 作

Section 9.3. Sequence Container Operations

c.erase(p)	Removes element referred to by the iterator p. 删除迭代器 p 所指向的元素
	Returns an iterator referring to the element after the one deleted, or an off-the-end iterator if p referred to the last element. Undefined if p is an off-the-end iterator. 返回一个迭代器，它指向被删除元素后面的元素。如果 p 指向容器内的最后一个元素，则返回的迭代器指向容器的超出末端的下一位置。如果 p 本身就是指向超出末端的下一位置的迭代器，则该函数未定义
c.erase(b,e)	Removes the range of elements denoted by the iterators b and e. 删除迭代器 b 和 e 所标记的范围内所有的元素
	Returns an iterator referring after the last one in the range that was deleted, or an off-the-end iterator if e is itself an off-the-end iterator. 返回一个迭代器，它指向被删除元素段后面的元素。如果 e 本身就是指向超出末端的下一位置的迭代器，则返回的迭代器也指向容器的超出末端的下一位置
c.clear()	Removes all the elements in c. Returns void. 删除容器 c 内的所有元素。返回 void
c.pop_back()	Removes the last element in c. Returns void. Undefined if c is empty. 删除容器 c 的最后一个元素。返回 void。如果 c 为空容器，则该函数未定义
c.pop_front()	Removes the first element in c. Returns void. Undefined if c is empty. 删除容器 c 的第一个元素。返回 void。如果 c 为空容器，则该函数未定义 Valid only for list or deque. 只适用于 list 或 deque 容器

Removing the First or Last Element

删除第一个或最后一个元素

The `pop_front` and `pop_back` functions remove the first and last elements in the container. There is no `pop_front` operation for `vectors`. These operations remove the indicated element and return `void`.

`pop_front` 和 `pop_back` 函数用于删除容器内的第一个和最后一个元素。但 `vector` 容器类型不支持 `pop_front` 操作。这些操作删除指定的元素并返回 `void`。

One common use of `pop_front` is to use it together with `front` to process a container as a stack:

`pop_front` 操作通常与 `front` 操作配套使用，实现以栈的方式处理容器：

```
while (!ilist.empty()) {  
    process(ilist.front()); // do something with the current top of ilist  
    ilist.pop_front();      // done; remove first element  
}
```

This loop is pretty simple: We use `front` to get a value to operate on and then call `pop_front` to remove that element from the `list`.

这个循环非常简单：使用 `front` 操作获取要处理的元素，然后调用 `pop_front` 函数从容器 `list` 中删除该元素。



The `pop_front` and `pop_back` return `void`; they do not return the value of the element popped. To examine that value, it is necessary to call `front` or `back` prior to popping the element.

`pop_front` 和 `pop_back` 函数的返回值并不是删除的元素值，而是 `void`。要获取删除的元素值，则必须在删除元素之前调用 `notfront` 或 `back` 函数。

Removing an Element From within the Container

删除容器内的一个元素

The more general way to remove an element, or range of elements, is through `erase`. There are two versions of `erase`: We can delete a single element referred to by an iterator or a range of elements marked by a pair of iterators. Both forms of `erase` return an iterator referring to the location after the element or range that was removed. That is, if element `j` is the element immediately after `i` and we `erase` element `i` from the container, then the iterator returned will refer to `j`.

删除一个或一段元素更通用的方法是 `erase` 操作。该操作有两个版本：删除由一个迭代器指向的单个元素，或删除由一对迭代器标记的一段元素。`erase` 的这两种形式都返回一个迭代器，它指向被删除元素或元素段后面的元素。也就是说，如果元素 `j` 恰好紧跟在元素 `i` 后面，则将元素 `i` 从容器中删除后，删除操作返回指向 `j` 的迭代器。



As usual, the `erase` operations don't check their argument(s). It is up to the programmer to ensure that the iterator or iterator range is valid.

如同其他操作一样，`erase` 操作也不会检查它的参数。程序员必须确保用作参数的迭代器或迭代器范围是有效的。

The `erase` operation is often used after finding an element that should be removed from the container. The easiest way to find a given element is to use the library `find` algorithm. We'll see more about `find` in [Section 11.1](#) (p. 392). To use `find` or any other generic algorithm, we must include the `algorithm` header. The `find` function takes a pair of iterators that denote a range in which to look, and a value to look for within that range. `find` returns an iterator referring to the first element with that value or the off-the-end iterator:

通常，程序员必须在容器中找出要删除的元素后，才使用 `erase` 操作。寻找一个指定元素的最简单方法是使用标准库的 `find` 算法。我们将在[第 11.1 节](#)中进一步讨论 `find` 算法。为了使用 `find` 函数或其他泛型算法，在编程时，必须将 `algorithm` 头文件包含进来。`find` 函数需要一对标记查找范围的迭代器以及一个在该范围内查找的值作参数。查找完成后，该函数返回一个迭代器，它指向具有指定值的第一个元素，或超出末端的下一位置。

```
string searchValue("Quasimodo");
list<string>::iterator iter =
    find(slist.begin(), slist.end(), searchValue);

if (iter != slist.end())
    slist.erase(iter);
```

Note that we check that the iterator is not the `end` iterator before erasing the element. When we ask `erase` to erase a single element, the element must exist—the behavior of `erase` is undefined if we ask it to `erase` an off-the-end iterator.

注意，在删除元素之前，必须确保迭代器是不是 `end` 迭代器。使用 `erase` 操作删除单个必须确保元素确实存在——如果删除指向超出末端的下一位置的迭代器，那么 `erase` 操作的行为未定义。

Removing All the Elements in a Container

删除容器内所有元素

To delete all the elements in a container, we could either call `clear` or pass the iterators from `begin` and `end` to `erase`:

要删除容器内所有的元素，可以调用 `clear` 函数，或将 `begin` 和 `end` 迭代器传递给 `erase` 函数。

```
slist.clear(); // delete all the elements within the container
slist.erase(slist.begin(), slist.end()); // equivalent
```

The iterator-pair version of `erase` lets us delete a subrange of elements:

`erase` 函数的迭代器对版本提供了删除一部分元素的功能：

```
// delete range of elements between two values
```

Section 9.3. Sequence Container Operations

```
list<string>::iterator elem1, elem2;  
  
// elem1 refers to val1  
elem1 = find(slist.begin(), slist.end(), val1);  
  
// elem2 refers to the first occurrence of val2 after val1  
elem2 = find(elem1, slist.end(), val2);  
  
// erase range from val1 up to but not including val2  
slist.erase(elem1, elem2);
```

This code starts by calling `find` twice to obtain iterators to two elements. The iterator `elem1` refers to the first occurrence of `val1` or to the off-the-end iterator if `val1` is not present in the `list`. The iterator `elem2` refers to the first occurrence of `val2` that appears after `val1` if that element exists, otherwise, `elem2` is an off-the-end iterator. The call to `erase` removes the elements starting from the referred to by `elem1` up to but not including `elem2`.

这段代码首先调用了 `find` 函数两次，以获得指向特定元素的两个迭代器。迭代器 `elem1` 指向第一个具有 `val1` 值的元素，如果容器 `list` 中不存在值为 `val1` 的元素，则该迭代器指向超出末端的下一位置。如果在 `val1` 元素后面存在值为 `val2` 的元素，那么迭代器 `elem2` 就指向这段范围内第一个具有 `val2` 值的元素，否则，`elem2` 就是一个超出末端的迭代器。最后，调用 `erase` 函数删除从迭代器 `elem1` 开始一直到 `elem2` 之间的所有元素，但不包括 `elem2` 指向的元素。



The `erase`, `pop_front`, and `pop_back` functions invalidate any iterators that refer to the removed elements. For `vectors`, iterators to elements after the erasure point are also invalidated. For `deque`, if the `erase` does not include either the first or last element, all iterators into the `deque` are invalidated.

`erase`、`pop_front` 和 `pop_back` 函数使指向被删除元素的所有迭代器失效。对于 `vector` 容器，指向删除点后面的元素的迭代器通常也会失效。而对于 `deque` 容器，如果删除时不包含第一个元素或最后一个元素，那么该 `deque` 容器相关的所有迭代器都会失效。

Exercises Section 9.3.7

Exercise 9.25: What happens in the program that erased a range of elements if `val1` is equal to `val2`. What happens if either `val1` or `val2` or both are not present.

需要删除一段元素时，如果 `val1` 与 `val2` 相等，那么程序会发生什么事情？如果 `val1` 和 `val2` 中的一个不存在，或两个都不存在，程序又会怎么样？

Exercise 9.26: Using the following definition of `ia`, copy `ia` into a `vector` and into a `list`. Use the single iterator form of `erase` to remove the elements with odd values from your `list` and the even values from your `vector`.

假设有如下 `ia` 的定义，将 `ia` 复制到一个 `vector` 容器和一个 `list` 容器中。使用单个迭代器参数版本的 `erase` 函数将 `list` 容器中的奇数值元素删除掉，然后将 `vector` 容器中的偶数值元素删除掉。

```
int ia[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89 };
```

Exercise 9.27: Write a program to process a `list` of `strings`. Look for a particular value and, if found, remove it. Repeat the program using a `deque`.

编写程序处理一个 `string` 类型的 `list` 容器。在该容器中寻找一个特殊值，如果找到，则将它删除掉。用 `deque` 容器重写上述程序。

9.3.8. Assignment and `swap`

9.3.8. 赋值与 `swap`

The assignment-related operators act on the entire container. Except for `swap`, they can be expressed in terms of `erase` and `insert` operations. The

Section 9.3. Sequence Container Operations

assignment operator *erases* the entire range of elements in the left-hand container and then *inserts* the elements of the right-hand container object into the left-hand container:

```
c1 = c2; // replace contents of c1 with a copy of elements in c2
// equivalent operation using erase and insert
c1.erase(c1.begin(), c1.end()); // delete all elements in c1
c1.insert(c1.begin(), c2.begin(), c2.end()); // insert c2
```

After the assignment, the left- and right-hand containers are equal: Even if the containers had been of unequal size, after the assignment both containers have the size of the right-hand operand.

赋值后，左右两边的容器相等：尽管赋值前两个容器的长度可能不相等，但赋值后两个容器都具有右操作数的长度。



Assignment and the `assign` operations invalidate all iterators into the left-hand container. `swap` does not invalidate iterators. After `swap`, iterators continue to refer to the same elements, although those elements are now in a different container.

赋值和 `assign` 操作使左操作数容器的所有迭代器失效。`swap` 操作则不会使迭代器失效。完成 `swap` 操作后，尽管被交换的元素已经存放在另一容器中，但迭代器仍然指向相同的元素。

Table 9.11. Sequential Container Assignment Operations

表 9.11. 顺序容器的赋值操作

<code>c1 = c2</code>	Deletes elements in <code>c1</code> and copies elements from <code>c2</code> into <code>c1</code> . <code>c1</code> and <code>c2</code> must be the same type. 删除容器 <code>c1</code> 的所有元素，然后将 <code>c2</code> 的元素复制给 <code>c1</code> 。 <code>c1</code> 和 <code>c2</code> 的类型（包括容器类型和元素类型）必须相同
<code>c1.swap(c2)</code>	Swaps contents: After the call <code>c1</code> has elements that were in <code>c2</code> , and <code>c2</code> has elements that were in <code>c1</code> . <code>c1</code> and <code>c2</code> must be the same type. Execution time usually <i>much</i> faster than copying elements from <code>c2</code> to <code>c1</code> . 交换内容：调用完该函数后， <code>c1</code> 中存放的是 <code>c2</code> 原来的元素， <code>c2</code> 中存放的则是 <code>c1</code> 原来的元素。 <code>c1</code> 和 <code>c2</code> 的类型必须相同。该函数的执行速度通常要比将 <code>c2</code> 复制到 <code>c1</code> 的操作快
<code>c.assign(b,e)</code>	Replaces the elements in <code>c</code> by those in the range denoted by iterators <code>b</code> and <code>e</code> . The iterators <code>b</code> and <code>e</code> must not refer to elements in <code>c</code> . 重新设置 <code>c</code> 的元素：将迭代器 <code>b</code> 和 <code>e</code> 标记的范围内所有的元素复制到 <code>c</code> 中。 <code>b</code> 和 <code>e</code> 必须不是指向 <code>c</code> 中元素的迭代器
<code>c.assign(n,t)</code>	Replaces the elements in <code>c</code> by <code>n</code> elements with value <code>t</code> . 将容器 <code>c</code> 重新设置为存储 <code>n</code> 个值为 <code>t</code> 的元素

Using `assign`

使用 `assign`

The `assign` operation deletes all the elements in the container and then inserts new elements as specified by the arguments. Like the constructor that copies elements from a container, the assignment operator (`=`) can be used to assign one container to another only if the container and

Section 9.3. Sequence Container Operations

element type are the same. If we want to assign elements of a different but compatible element type and/or from a different container type, then we must use the `assign` operation. For example, we could use `assign` to assign a range of `char*` values from a `vector` into a `list` of `string`.

`assign` 操作首先删除容器中所有的元素，然后将其参数所指定的新元素插入到该容器中。与复制容器元素的构造函数一样，如果两个容器类型相同，其元素类型也相同，就可以使用赋值操作符 (`=`) 将一个容器赋值给另一个容器。如果在不同（或相同）类型的容器内，元素类型不相同但是相互兼容，则其赋值运算必须使用 `assign` 函数。例如，可通过 `assign` 操作实现将 `vector` 容器中一段 `char*` 类型的元素赋给 `string` 类型 `list` 容器。



Because the original elements are deleted, the iterators passed to `assign` must not refer to elements in the container on which `assign` is called.

由于 `assign` 操作首先删除容器中原来存储的所有元素，因此，传递给 `assign` 函数的迭代器不能指向调用该函数的容器内的元素。

The arguments to `assign` determine how many elements are inserted and what values the new elements will have. This statement:

`assign` 函数的参数决定了要插入多少个元素以及新元素的值是什么。语句

```
// equivalent to slist1 = slist2  
slist1.assign(slist2.begin(), slist2.end());
```

uses the version of `assign` that takes a pair of iterators. After deleting the elements in `slist1`, the function copies the elements in the range denoted by the iterators into `slist2`. Thus, this code is equivalent to assigning `slist2` to `slist1`.

使用了带一对迭代器参数的 `assign` 函数版本。在删除 `slist1` 的元素后，该函数将 `slist2` 容器内一段指定的元素复制到 `slist2` 中。于是，这段代码行等效于将 `slist2` 赋给 `slist1`。



The `assign` operator that takes an iterator pair lets us assign elements of one container type to another.

带有一对迭代器参数的 `assign` 操作允许我们将一个容器的元素赋给另一个不同类型的容器。

A second version of `assign` takes an integral value and an element value. It replaces the elements in the container by the specified number of elements, each of which has the specified element value

`assign` 运算的第二个版本需要一个整型数值和一个元素值做参数，它将容器重置为存储指定数量的元素，并且每个元素的值都为指定值：

```
// equivalent to: slist1.clear();  
// followed by slist1.insert(slist1.begin(), 10, "Hiya!");  
slist1.assign(10, "Hiya!"); // 10 elements; each one is Hiya!
```

After executing this statement, `slist1` has 10 elements, each of which has the value `Hiya!`.

执行了上述语句后，容器 `slist1` 有 10 个元素，每个元素的值都是 `Hiya!`。

Using `swap` to Avoid the Cost of Deleting Elements

使用 `swap` 操作以节省删除元素的成本

The `swap` operation swaps the values of its two operands. The types of the containers must match: The operands must be the same kind of container, and they must hold values of the same type. After the call to `swap`, the elements that had been in the right-hand operand are in the left, and vice versa:

`swap` 操作实现交换两个容器内所有元素的功能。要交换的容器的类型必须匹配：操作数必须是相同类型的容器，而且所存储的元素类型也必须相同。调用了 `swap` 函数后，右操作数原来存储的元素被存放在左操作数中，反之亦然。

Section 9.3. Sequence Container Operations

```
vector<string> svec1(10); // vector with 10 elements
vector<string> svec2(24); // vector with 24 elements
svec1.swap(svec2);
```

After the `swap`, `svec1` contains 24 `string` elements and `svec2` contains 10.

执行 `swap` 后，容器 `svec1` 中存储 24 个 `string` 类型的元素，而 `svec2` 则存储 10 个元素。



The important thing about `swap` is that it does not delete or insert any elements and is guaranteed to run in constant time. No elements are moved, and so iterators are not invalidated.

关于 `swap` 的一个重要问题在于：该操作不会删除或插入任何元素，而且保证在常量时间内实现交换。由于容器内没有移动任何元素，因此迭代器不会失效。

The fact that elements are not moved means that iterators are not invalidated. They refer to the same elements as they did before the swap. However, after the `swap`, those elements are in a different container. For example, had `iter` referred to the `string` at position `svec1[3]` before the `swap` it will refer to the element at position `svec2[3]` after the `swap`.

没有移动元素这个事实意味着迭代器不会失效。它们指向同一元素，就像没作 `swap` 运算之前一样。虽然，在 `swap` 运算后，这些元素已经被存储在不同的容器之中了。例如，在做 `swap` 运算之前，有一个迭代器 `iter` 指向 `svec1[3]` 字符串；实现 `swap` 运算后，该迭代器则指向 `svec2[3]` 字符串（这是同一个字符串，只是存储在不同的容器之中而已）。

Exercises Section 9.3.8

Exercise Write a program to assign the elements from a `list` of `char*` pointers to C-style character
9.28: strings to a `vector` of `string`s.

编写程序将一个 `list` 容器的所有元素赋值给一个 `vector` 容器，其中 `list` 容器中存储的是指向 C 风格字符串的 `char*` 指针，而 `vector` 容器的元素则是 `string` 类型。

9.4. How a `vector` Grows

9.4. `vector` 容器的自增长

When we `insert` or `push` an element onto a container object, the size of that object increases by one. Similarly, if we `resize` a container to be larger than its current `size`, then additional elements must be added to the container. The library takes care of allocating the memory to hold these new elements.

在容器对象中 `insert` 或压入一个元素时，该对象的大小增加 1。类似地，如果 `resize` 容器以扩充其容量，则必须在容器中添加额外的元素。标准库处理存储这些新元素的内存分配问题。

Ordinarily, we should not care about how a library type works: All we should care about is how to use it. However, in the case of `vectors`, a bit of the implementation leaks into its interface. To support fast random access, `vector` elements are stored contiguously each element is adjacent to the previous element.

一般来说，我们不应该关心标准库类型是如何实现的：我们只需要关心如何使用这些标准库类型就可以了。然而，对于 `vector` 容器，有一些实现也与其接口相关。为了支持快速的随机访问，`vector` 容器的元素以连续的方式存放——每一个元素都紧挨着前一个元素存储。

Given that elements are contiguous, let's think about what happens when we add an element to a `vector`: If there is no room in the `vector` for the new element, it cannot just add an element somewhere else in memory because the elements must be contiguous for indexing to work. Instead, the `vector` must allocate new memory to hold the existing elements plus the new one, copy the elements from the old location into the new space, add the new element, and deallocate the old memory. If `vector` did this memory allocation and deallocation each time we added an element, then performance would be unacceptably slow.

已知元素是连续存储的，当我们在容器内添加一个元素时，想想会发生什么事情：如果容器中已经没有空间容纳新的元素，此时，由于元素必须连续存储以便索引访问，所以不能在内存中随便找个地方存储这个新元素。于是，`vector` 必须重新分配存储空间，用来存放原来的元素以及新添加的元素：存放在旧存储空间中的元素被复制到新存储空间里，接着插入新元素，最后撤销旧的存储空间。如果 `vector` 容器在每次添加新元素时，都要这么分配和撤销内存空间，其性能将会非常慢，简直无法接受。

There is no comparable allocation issue for containers that do not hold their elements contiguously. For example, to add an element to a `list`, the library only needs to create the new element and chain it into the existing list. There is no need to reallocate or copy any of the existing elements.

对于不连续存储元素的容器，不存在这样的内存分配问题。例如，在 `list` 容器中添加一个元素，标准库只需创建一个新元素，然后将该新元素连接在已存在的链表中，不需要重新分配存储空间，也不必复制任何已存在的元素。

One might conclude, therefore, that in general it is a good idea to use a `list` rather than a `vector`. However, the contrary is usually the case: For most applications the best container to use is a `vector`. The reason is that library implementors use allocation strategies that minimize the costs of storing elements contiguously. That cost is usually offset by the advantages in accessing elements that contiguous storage allows.

由此可以推论：一般而言，使用 `list` 容器优于 `vector` 容器。但是，通常出现的反而是以下情况：对于大部分应用，使用 `vector` 容器是最好的。原因在于，标准库的实现者使用这样内存分配策略：以最小的代价连续存储元素。由此而带来的访问元素的便利弥补了其存储代价。

The way `vectors` achieve fast allocation is by allocating capacity beyond what is immediately needed. The `vector` holds this storage in reserve and uses it to allocate new elements as they are added. Thus, there is no need to reallocate the container for each new element. The exact amount of additional capacity allocated varies across different implementations of the library. This allocation strategy is dramatically more efficient than reallocating the container each time an element is added. In fact, its performance is good enough that in practice a `vector` usually grows more efficiently than a `list` or a `deque`.

为了使 `vector` 容器实现快速的内存分配，其实际分配的容量要比当前所需的空间多一些。`vector` 容器预留了这些额外的存储区，用于存放新添加的元素。于是，不必为每个新元素重新分配容器。所分配的额外内存容量的确切数目因库的实现不同而不同。比起每添加一个新元素就必须重新分配一次容器，这个分配策略带来显著的效率。事实上，其性能非常好，因此在实际应用中，比起 `list` 和 `deque` 容器，`vector` 的增长效率通常会更高。

9.4.1. `capacity` and `reserve` Members

9.4.1. `capacity` 和 `reserve` 成员

The details of how `vector` handles its memory allocation are part of its implementation. However, a portion of this implementation is supported by the interface to `vector`. The `vector` class includes two members, `capacity` and `reserve`, that let us interact with the memory-allocation part of `vector`'s implementation. The `capacity` operation tells us how many elements the container could hold before it must allocate more space. The `reserve` operation lets us tell the `vector` how many elements it should be prepared to hold.

`vector` 容器处理内存分配的细节是其实现的一部分。然而，该实现部分是由 `vector` 的接口支持的。`vector` 类提供了两个成员函数：`capacity` 和 `reserve` 使程序员可与 `vector` 容器内存分配的实现部分交互工作。`capacity` 操作获取在容器需要分配更多的存储空间之前能够存储的元素总数，而 `reserve` 操作则告诉 `vector` 容器应该预留多少个元素的存储空间。

Section 9.4. How a vector Grows



It is important to understand the difference between `capacity` and `size`. The `size` is the number of elements in the `vector`; `capacity` is how many it could hold before new space must be allocated.

弄清楚容器的 `capacity` (容量) 与 `size` (长度) 的区别非常重要。`size` 指容器当前拥有的元素个数；而 `capacity` 则指容器在必须分配新存储空间之前可以存储的元素总数。

To illustrate the interaction between `size` and `capacity`, consider the following program:

为了说明 `size` 和 `capacity` 的交互作用，考虑下面的程序：

```
vector<int> ivect;  
  
// size should be zero; capacity is implementation defined  
cout << "ivec: size: " << ivect.size()  
     << " capacity: " << ivect.capacity() << endl;  
  
// give ivect 24 elements  
for (vector<int>::size_type ix = 0; ix != 24; ++ix)  
    ivect.push_back(ix);  
  
// size should be 24; capacity will be >= 24 and is implementation defined  
cout << "ivec: size: " << ivect.size()  
     << " capacity: " << ivect.capacity() << endl;
```

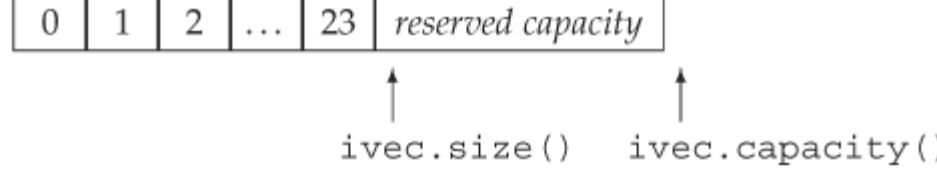
When run on our system, this program produces the following output:

在我们的系统上运行该程序时，得到以下输出结果：

```
ivec: size: 0 capacity: 0  
ivec: size: 24 capacity: 32
```

We know that the `size` of an empty `vector` is zero, and evidently our library also sets `capacity` of an empty `vector` to zero. When we add elements to the `vector`, we know that the `size` is the same as the number of elements we've added. The `capacity` must be at least as large as `size` but can be larger. Under this implementation, adding 24 elements one at a time results in a `capacity` of 32. Visually we can think of the current state of `ivec` as

由此可见，空 `vector` 容器的 `size` 是 0，而标准库显然将其 `capacity` 也设置为 0。当程序员在 `vector` 中插入元素时，容器的 `size` 就是所添加的元素个数，而其 `capacity` 则必须至少等于 `size`，但通常比 `size` 值更大。在上述程序中，一次添加一个元素，共添加了 24 个元素，结果其 `capacity` 为 32。容器的当前状态如下图所示：



We could now `reserve` some additional space:

现在，可如下预留额外的存储空间：

```
ivec.reserve(50); // sets capacity to at least 50; might be more  
// size should be 24; capacity will be >= 50 and is implementation defined  
cout << "ivec: size: " << ivect.size()  
     << " capacity: " << ivect.capacity() << endl;
```

As the output indicates, doing so changes the `capacity` but not the `size`:

正如下面的输出结果所示，该操作保改变了容器的 `capacity`，而其 `size` 不变：

```
ivec: size: 24 capacity: 50
```

We might next use up that reserved capacity as follows:

下面的程序将预留的容量用完：

```
// add elements to use up the excess capacity  
while (ivec.size() != ivec.capacity())
```

Section 9.4. How a vector Grows

```
ivec.push_back(0);
// size should be 50; capacity should be unchanged
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
```

Because we used only reserved capacity, there is no need for the `vector` to do any allocation. In fact, as long as there is excess capacity, the `vector` must not reallocate its elements.

由于在该程序中，只使用了预留的容量，因此 `vector` 不必做任何的内存分配工作。事实上，只要有剩余的容量，`vector` 就不必为其元素重新分配存储空间。

The output indicates that at this point we've used up the reserved capacity, and `size` and `capacity` are equal:

其输出结果表明：此时我们已经耗尽了预留的容量，该容器的 `size` 和 `capacity` 值相等：

```
ivec: size: 50 capacity: 50
```

If we now add another element, the `vector` will have to reallocate itself:

此时，如果要添加新的元素，`vector` 必须为自己重新分配存储空间：

```
ivec.push_back(42); // add one more element
// size should be 51; capacity will be >= 51 and is implementation defined
cout << "ivec: size: " << ivec.size()
    << " capacity: " << ivec.capacity() << endl;
```

The output from this portion of the program

这段程序的输出：

```
ivec: size: 51 capacity: 100
```

indicates that this `vector` implementation appears to follow a strategy of doubling the current capacity each time it has to allocate new storage.

表明：每当 `vector` 容器不得不分配新的存储空间时，以加倍当前容量的分配策略实现重新分配。



Each implementation of `vector` is free to choose its own allocation strategy. However, it must provide the `reserve` and `capacity` functions, and it must not allocate new memory until it is forced to do so. How much memory it allocates is up to the implementation. Different libraries will implement different strategies.

`vector` 的每种实现都可自由地选择自己的内存分配策略。然而，它们都必须提供 `vector` 和 `capacity` 函数，而且必须是到必要时才分配新的内存空间。分配多少内存取决于其实现方式。不同的库采用不同的策略实现。

Moreover, every implementation is required to follow a strategy that ensures that it is efficient to use `push_back` to populate a `vector`. Technically speaking, the execution time of creating an n -element `vector` by calling `push_back` n times on an initially empty `vector` is never more than a constant multiple of n .

此外，每种实现都要求遵循以下原则：确保 `push_back` 操作高效地在 `vector` 中添加元素。从技术上来说，在原来为空的 `vector` 容器上 n 次调用 `push_back` 函数，从而创建拥有 n 个元素的 `vector` 容器，其执行时间永远不能超过 n 的常量倍。

Exercises Section 9.4.1

Exercise 9.29: Explain the difference between a `vector`'s capacity and its size. Why is it necessary to support the notion of capacity in a container that stores elements contiguously but not, for example, in a `list`?

解释 `vector` 的容量和长度之间的区别。为什么在连续存储元素的容器中需要支持“容量”的概念？而非连续的容器，如 `list`，则不需要。

Exercise 9.30: Write a program to explore the allocation strategy followed by the library you use for `vector` objects.

Section 9.4. How a vector Grows

编写程序研究标准库为 `vector` 对象提供的内存分配策略。

Exercise Can a container have a capacity less than its size? Is a capacity equal to its size desirable?
9.31: Initially? After an element is inserted? Why or why not?

容器的容量可以比其长度小吗？在初始时或插入元素后，容量是否恰好等于所需要的长度？为什么？

Exercise Explain what the following program does:
9.32: 解释下面程序实现的功能：

```
vector<string> svec;
svec.reserve(1024);
string text_word;
while (cin >> text_word)
    svec.push_back(text_word);
svec.resize(svec.size() + svec.size() / 2);
```

If the program reads 256 words, what is its likely capacity after it is resized? What if it reads 512? 1,000? 1,048?

如果该程序读入了 256 个单词，在调整大小后，该容器可能是多少？如果读入 512，或 1000，或 1048 个单词呢？

9.5. Deciding Which Container to Use

9.5. 容器的选用

As we saw in the previous section, allocating memory to hold elements in contiguous storage has impacts on the memory allocation strategies and overhead of a container. By using clever implementation techniques, library authors minimize this allocation overhead. Whether elements are stored contiguously has other significant impacts on:

在前面的章节中可见，分配连续存储元素的内存空间会影响内存分配策略和容器对象的开销。通过巧妙的实现技巧，标准库的实现者已经最小化了内存分配的开销。元素是否连续存储还会显著地影响：

- The costs to add or delete elements from the middle of the container

在容器的中间位置添加或删除元素的代价。

- The costs to perform nonsequential access to elements of the container

执行容器元素的随机访问的代价。

The degree to which a program does these operations should be used to determine which type of container to use. The `vector` and `deque` types provide fast non-sequential access to elements at the cost of making it expensive to add or remove elements anywhere other than the ends of the container. The `list` type supports fast insertion and deletion anywhere but at the cost of making nonsequential access to elements expensive.

程序使用这些操作的程序将决定应该选择哪种类型的容器。`vector` 和 `deque` 容器提供了对元素的快速随机访问，但付出的代价是，在容器的任意位置插入或删除元素，比在容器尾部插入和删除的开销更大。`list` 类型在任何位置都能快速插入和删除，但付出的代价是元素的随机访问开销较大。

How Insertion Affects Choice of Container

插入操作如何影响容器的选择

A `list` represents noncontiguous memory and allows for both forward and backward traversal one element at a time. It is efficient to `insert` or `erase` an element at any point. Inserting or removing an element in a `list` does not move any other elements. Random access, on the other hand, is not supported. Accessing an element requires traversing the intervening elements.

`list` 容器表示不连续的内存区域，允许向前和向后逐个遍历元素。在任何位置都可高效地 `insert` 或 `erase` 一个元素。插入或删除 `list` 容器中的一个元素不需要移动任何其他元素。另一方面，`list` 容器不支持随机访问，访问某个元素要求遍历涉及的其他元素。

Inserting (or removing) anywhere except at the back of a `vector` requires that each element to the right of the inserted (or deleted) element be moved. For example, if we have a `vector` with 50 elements and we wish to `erase` element number 23, then each of the elements after 23 have to be moved forward by one position. Otherwise, there'd be a hole in the `vector`, and the `vector` elements would no longer be contiguous.

对于 `vector` 容器，除了容器尾部外，其他任何位置上的插入（或删除）操作都要求移动被插入（或删除）元素右边所有的元素。例如，假设有一个拥有 50 个元素的 `vector` 容器，我们希望删除其中的第 23 号元素，则 23 号元素后面的所有元素都必须向前移动一个位置。否则，`vector` 容器上将会留下一个空位（hole），而 `vector` 容器的元素就不再是连续存放的了。

A `deque` is a more complicated data structure. We are guaranteed that adding or removing elements from either end of the `deque` is a fast operation. Adding or removing from the middle will be more expensive. A `deque` offers some properties of both `list` and `vector`:

`deque` 容器拥有更加复杂的数据结构。从 `deque` 队列的两端插入和删除元素都非常快。在容器中间插入或删除付出的代价将更高。`deque` 容器同时提供了 `list` 和 `vector` 的一些性质：

- Like `vector`, it is inefficient to `insert` or `erase` elements in the middle of the `deque`.

与 `vector` 容器一样，在 `deque` 容器的中间 `insert` 或 `erase` 元素效率比较低。

- Unlike `vector`, a `deque` offers efficient `insert` and `erase` at the front as well as at the back.

不同于 `vector` 容器，`deque` 容器提供高效地在其首部实现 `insert` 和 `erase` 的操作，就像在容器尾部的一样。

- Unlike `list` and like `vector`, a `deque` supports fast random access to any element.

与 `vector` 容器一样而不同于 `list` 容器的是，`deque` 容器支持对所有元素的随机访问。

Section 9.5. Deciding Which Container to Use

- Inserting elements at the front or back of a `deque` does not invalidate any iterators. Erasing the front or back element invalidates only iterators referring to the element(s) erased. Inserting or erasing anywhere else in the `deque` invalidates all iterators referring to elements of the `deque`.
- 在 `deque` 容器首部或尾部插入元素不会使任何迭代器失效，而首部或尾部删除元素则只会使指向被删除元素的迭代器失效。在 `deque` 容器的任何其他位置的插入和删除操作将使指向该容器元素的所有迭代器都失效。

How Access to the Elements Affects Choice of Container

元素的访问如何影响容器的选择

Both `vector` and `deque` support efficient random access to their elements. That is, we can efficiently access element 5, then 15, then 7, and so on. Random access in a `vector` can be efficient because each access is to a fixed offset from the beginning of the `vector`. It is much slower to jump around in a `list`, the only way to move between the elements of a `list` is to sequentially follow the pointers. Moving from the 5th to the 15th element requires visiting every element between them.

`vector` 和 `deque` 容器都支持对其元素实现高效的随机访问。也就是说，我们可以高效地先访问 5 号元素，然后访问 15 号元素，接着访问 7 号元素，等等。由于 `vector` 容器的每次访问都是距离其起点的固定偏移，因此其随机访问非常有效率。在 `list` 容器中，上述跳跃访问会变得慢很多。在 `list` 容器的元素之间移动的唯一方法是顺序跟踪指针。从 5 号元素移动到 15 号元素必须遍历它们之间所有的元素。



In general, unless there is a good reason to prefer another container, `vector` is usually the right one to use.

通常来说，除非找到选择使用其他容器的更好理由，否则 `vector` 容器都是最佳选择。

Hints on Selecting Which Container to Use

选择容器的提示

There are a few rules of thumb that apply to selecting which container to use:

下面列举了一些选择容器类型的法则：

1. If the program requires random access to elements, use a `vector` or a `deque`.
如果程序要求随机访问元素，则应使用 `vector` 或 `deque` 容器。
2. If the program needs to insert or delete elements in the middle of the container, use a `list`.
如果程序必须在容器的中间位置插入或删除元素，则应采用 `list` 容器。
3. If the program needs to insert or delete elements at the front and the back, but not in the middle, of the container, use a `deque`.
如果程序不是在容器的中间位置，而是在容器首部或尾部插入或删除元素，则应采用 `deque` 容器。
4. If we need to insert elements in the middle of the container only while reading input and then need random access to the elements, consider reading them into a `list` and then reordering the `list` as appropriate for subsequent access and copying the reordered `list` into a `vector`.
如果只需在读取输入时在容器的中间位置插入元素，然后需要随机访问元素，则可考虑在输入时将元素读入到一个 `list` 容器，接着对此容器重新排序，使其适合顺序访问，然后将排序后的 `list` 容器复制到一个 `vector` 容器。

What if the program needs to randomly access and insert and delete elements in the middle of the container?

如果程序既需要随机访问又必须在容器的中间位置插入或删除元素，那应该怎么办呢？

This decision will depend on the relative cost of doing random access to `list` elements versus the cost of copying elements when inserting or deleting elements in a `vector` or `deque`. In general, the predominant operation of the application (whether it does more access or more insertion or deletion) should determine the choice of container type.

此时，选择何种容器取决于下面两种操作付出的相对代价：随机访问 `list` 容器元素的代价，以及在 `vector` 或 `deque` 容器中插入／删除元素时复制元素的代价。通常来说，应用中占优势的操作（程序中更多使用的是访问操作还是插入／删除操作）将决定应该什么类型的容器。

Deciding which container to use may require profiling the performance of each container type doing the kinds of operations the application requires.

决定使用哪种容器可能要求剖析各种容器类型完成应用所要求的各类操作的性能。



When you are not certain which container the application should use, try to write your code so that it uses only operations common to both `vectors` and `lists`: Use iterators, not subscripts, and avoid random access to elements. By writing your programs this way, it will be easier to change the container from a `vector` to a `list` if necessary.

如果无法确定某种应用应该采用哪种容器，则编写代码时尝试只使用 `vector` 和 `lists` 容器都提供的操作：使用迭代器，而不是下标，并且避免随机访问元素。这样编写，在必要时，可很方便地将程序从使用 `vector` 容器修改为使用 `list` 的容器。

Exercises Section 9.5

Exercise 9.33: Which is the most appropriate `vector`, a `deque`, or a `list` for the following program tasks? Explain the rationale for your choice. If there is no reason to prefer one or another container explain why not?

对于下列程序任务，采用哪种容器（`vector`、`deque` 还是 `list`）实现最合适？解释选择的理由。如果无法说明采用某种容器比另一种容器更好的原因，请解释为什么无法说明？

- a. Read an unknown number of words from a file for the purpose of generating English language sentences.

从一个文件中讲稿未知数目的单词，以生成英文句子。

- b. Read a fixed number of words, inserting them in the container alphabetically as they are entered. We'll see in the next chapter that associative containers are better suited to this problem.

读入固定数目的单词，在输入时将它们按字母顺序插入到容器中。下一章将更适合处理此类问题的关联容器。

- c. Read an unknown number of words. Always insert new words at the back. Remove the next value from the front.

读入未知数目的单词。总是在容器尾部插入新单词，从容器首部删除下一个值。

- d. Read an unknown number of integers from a file. Sort the numbers and then print them to standard output.

从一个文件中讲稿未知数目的整数。对这些整数排序，然后把它们输出到标准输出设备。

9.6. strings Revisited

9.6. 再谈 string 类型

We introduced the `string` type in [Section 3.2](#) (p. 80). [Table 9.12](#) (p. 337) recaps the `string` operations covered in that section.

第 3.2 节介绍了 `string` 类型, 表 9.12 概要重述了在该节中介绍的 `string` 操作。

Table 9.12. string Operations Introduced in Section 3.2

表 9.12 第 3.2 节介绍的 `string` 操作

<code>string s;</code>	Defines a new, empty <code>string</code> named <code>s</code> . 定义一个新的空 <code>string</code> 对象, 命名为 <code>s</code>
<code>string s(cp);</code>	Defines a new <code>string</code> initialized from the null-terminated C-style string pointed to by <code>cp</code> . 定义一个新的 <code>string</code> 对象, 用 <code>cp</code> 所指向的 (以空字符 <code>null</code> 结束的) C 风格字符串初始化该对象
<code>string s(s2);</code>	Defines a new <code>string</code> initialized as a copy of <code>s2</code> . 定义一个新的 <code>string</code> 对象, 并将它初始化为 <code>s2</code> 的副本
<code>is >> s;</code>	Reads a whitespace-separated string from the input stream <code>is</code> into <code>s</code> . 从输入流 <code>is</code> 中读取一个以空白字符分隔的字符串, 写入 <code>s</code>
<code>os << s;</code>	Writes <code>s</code> to the output stream <code>os</code> . 将 <code>s</code> 写到输出流 <code>os</code> 中
<code>getline(is, s)</code>	Reads characters up to the first newline from input stream <code>is</code> into <code>s</code> . 从输入流 <code>is</code> 中读取一行字符, 写入 <code>s</code>
<code>s1 + s2</code>	Concatenates <code>s1</code> and <code>s2</code> , yielding a new <code>string</code> . 把 <code>s1</code> 和 <code>s2</code> 串接起来, 产生一个新的 <code>string</code> 对象
<code>s1 += s2</code>	Appends <code>s2</code> to <code>s1</code> . 将 <code>s2</code> 拼接在 <code>s1</code> 后面
Relational Operators 关系操作符	The equality (<code>==</code> and <code>!=</code>) and relational (<code><</code> , <code><=</code> , <code>></code> , and <code>>=</code>) can be used to compare <code>strings</code> . <code>string</code> comparison is equivalent to (case-sensitive) dictionary ordering. 相等运算 (<code>==</code> 和 <code>!=</code>) 以及关系运算 (<code><</code> , <code><=</code> , <code>></code> 和 <code>>=</code>) 都可用于 <code>string</code> 对象的比较, 等效于 (区分大小写的) 字典次序的比较

In addition to the operations we've already used, `strings` also supports most of the sequential container operations. In some ways, we can think of a `string` as a container of characters. With some exceptions, `strings` support the same operations that `vectors` support: The exceptions are that `string` does not support the operations to use the container like a stack: We cannot use the `front`, `back`, and `pop_back` operations on `strings`.

除了已经使用过的操作外, `string` 类型还支持大多数顺序容器操作。在某些方面, 可将 `string` 类型视为字符容器。除了一些特殊操作, `string` 类型提供与 `vector` 容器相同的操作。`string` 类型与 `vector` 容器不同的是, 它不支持以栈方式操纵容器: 在 `string` 类型中不能使用 `front`、`back` 和 `pop_back` 操作。

The container operations that `string` supports are:

`string` 支持的容器操作有:

Section 9.6. strings Revisited

- The typedefs, including the iterator types, listed in [Table 9.5](#) (p. 316).

表 9.5 列出的 typedef，包括迭代器类型。

- The constructors listed in [Table 9.2](#) (p. 307) except for the constructor that takes a single size parameter.

表 9.2 列出的容器构造函数，但是不包括只需要一个长度参数的构造函数。

- The operations to add elements listed in [Table 9.7](#) (p. 319) that `vector` supports. Note: Neither `vector` nor `string` supports `push_front`.

表 9.7 列出的 `vector` 容器所提供的添加元素的操作。注意：无论 `vector` 容器还是 `string` 类型都不支持 `push_front` 操作。

- The size operations in [Table 9.8](#) (p. 324).

表 9.8 列出的长度操作。

- The subscript and `at` operations listed in [Table 9.9](#) (p. 325); `string` does not provide `back` or `front` operations listed in that table.

表 9.9 列出的下标和 `at` 操作；但 `string` 类型不提供该表列出的 `back` 和 `front` 操作。

- The `begin` and `end` operations of [Table 9.6](#) (p. 317).

表 9.6 列出的 `begin` 和 `end` 操作。

- The `erase` and `clear` operations of [Table 9.10](#) (p. 326); `string` does not support either `pop_back` or `pop_front`.

表 9.10 列出的 `erase` 和 `clear` 操作；但是 `string` 类型不提供 `pop_back` 或 `pop_front` 操作。

- The assignment operations in [Table 9.11](#) (p. 329).

表 9.11 列出的赋值操作。

- Like the elements in a `vector`, the characters of a `string` are stored contiguously. Therefore, `string` supports the `capacity` and `reserve` operations described in [Section 9.4](#) (p. 330).

与 `vector` 容器的元素一样，`string` 的字符也是连续存储的。因此，`string` 类型支持第 9.4 节描述的 `capacity` 和 `reserve` 操作。

When we say that `string` supports the container operations, we mean that we could take a program that manipulates a `vector` and rewrite that same program to operate on `strings`. For example, we could use iterators to print the characters of a `string` a line at a time to the standard output:

`string` 类型提供容器操作意味着可将操纵 `vector` 对象的程序改写为操纵 `string` 对象。例如，以下程序使用迭代器将一个 `string` 对象的字符以每次一行的方式输出到标准输出设备：

```
string s("Hiya!");
string::iterator iter = s.begin();
while (iter != s.end())
    cout << *iter++ << endl; // postfix increment: print old value
```

Not surprisingly, this code looks almost identical to the code from page 163 that printed the elements of a `vector<int>`.

不要奇怪，这段代码看上去几乎与第 5.5 节中输出 `vector` 容器元素的程序一模一样。

In addition to the operations that `string` shares with the containers, `string` supports other operations that are specific to `strings`. We will review these `string`-specific operations in the remainder of this section. These operations include additional versions of container-related operations as well as other, completely new functions. The additional functions that `string` provides are covered starting on page 341.

除了共享容器的操作外，`string` 类型还支持其他本类型特有的操作。在本节剩下的篇幅上，我们将回顾这些 `string` 类型特有的操作，包括与其他容器相关操作的补充版本，以及全新的函数。`string` 类型的补充函数将在第 9.6.2 节介绍。

The additional versions of the container operations that `string` provides are defined to support attributes that are unique to `strings` and not shared by the containers. For example, several operations permit us to specify arguments that are pointers to character arrays. These operations support the close interaction between library `strings` and character arrays, whether null-terminated or not. Other versions let us use indices rather than iterators. These versions operate positionally: We specify a starting position, and in some cases a count, to specify the element or range of elements which we want to manipulate.

`string` 类型为某些容器操作提供补充版本，以支持 `string` 特有的、不为其他容器共享的属性。例如，好几种操作允许指定指向字符数组的指针参数。无论字符串是否以空字符结束，这些操作都支持标准库 `string` 对象与字符数组之间的紧密交互作用。其他版本则使用程序员只能使用下标而不能使用迭代器。这些版本只能通过位置操纵元素：指定初始位置，在某些情况下还需指定一个计数器，由此指定要操纵的某个元素或一段元素。

Exercises Section 9.6

Exercise Use iterators to change the characters in a `string` to uppercase.

9.34: 使用迭代器将 `string` 对象中的字符都改为大写字母。

Exercise Use iterators to find and to erase each capital letter from a `string`.

9.35: 使用迭代器寻找和删除 `string` 对象中所有的大写字符。

Exercise Write a program that initializes a `string` from a `vector<char>`.

9.36: 编写程序用 `vector<char>` 容器初始化 `string` 对象。

Exercise Given that you want to read a character at a time into a `string`, and you know that the data you need to read is at least 100 characters long, how might you improve the performance of your program?

假设希望一次读取一个字符并写入 `string` 对象，而且已知需要读入至少 100 个字符，考虑应该如何提高程序的性能？



The `string` library defines a great number of functions, which use repeated patterns. Given the number of functions supported, this section can be mind-numbing on first reading.

`string` 库定义了大量使用重复模式的函数。由于该类型支持的函数非常多，初次阅读本节是会觉得精神疲累。

Readers might want to skim the remainder of [Section 9.6](#). Once you know what kinds of operations are available, you can return for the details when writing programs that need to use a given operation.

读者可跳过第 9.6 节剩下的内容。一旦知道了有哪些操作可以使用，就可以在编写需要使用这种操作的程序时，才回来阅读其细节。

9.6.1. Other Ways to Construct `string`s

9.6.1. 构造 `string` 对象的其他方法

The `string` class supports all but one of the constructors in [Table 9.2](#) (p. 307). The constructor that takes a single size parameter is not supported for `string`. We can create a `string`: as the empty `string`, by providing no argument; as a copy of another `string`; from a pair of iterators; or from a count and a character:

`string` 类支持表 9.2 所列出的几乎所有构造函数，只有一个例外：`string` 不支持带有单个容器长度作为参数的构造函数。创建 `string` 对象时：不提供任何参数，则得到空的 `string` 对象；也可将新对象初始化为另一个 `string` 对象的副本；或用一对迭代器初始化；或者使用一个计数器和一个字符初始化：

```
string s1;           // s1 is the empty string
string s2(5, 'a');   // s2 == "aaaaa"
string s3(s2);       // s3 is a copy of s2
string s4(s3.begin(),
         s3.begin() + s3.size() / 2); // s4 == "aa"
```

In addition to these constructors, the `string` type supports three other ways to create a `string`. We have already used the constructor that takes a pointer to the first character in a null-terminated, character array. There is another constructor that takes a pointer to an element in a character array and a count of how many characters to copy. Because the constructor takes a count, the array does not have to be null-terminated:

```
char *cp = "Hiya";           // null-terminated array
char c_array[] = "World!!!!"; // null-terminated
char no_null[] = {'H', 'i'};   // not null-terminated

string s1(cp);               // s1 == "Hiya"
string s2(c_array, 5);        // s2 == "World"
string s3(c_array + 5, 4);    // s3 == "!!!!"
```

Section 9.6. strings Revisited

```
string s4(no_null);      // runtime error: no_null not null-terminated
string s5(no_null, 2);    // ok: s5 == "Hi"
```

We define `s1` using the constructor that takes a pointer to the first character of a null-terminated array. All the characters in that array, up to but not including the terminating null, are copied into the newly created `string`.

使用只有一个指针参数的构造函数定义 `s1`, 该指针指向以空字符结束的数组中的第一字符。这个数组的所有字符, 但不包括结束符 `null`, 都被复制到新创建的 `string` 对象中。

The initializer for `s2` uses the second constructor, taking a pointer and a count. In this case, we start at the character denoted by the pointer and copy as many characters as indicated in the second argument. `s2`, therefore, is a copy of the first five characters from the array `c_array`. Remember that when we pass an array as an argument, it is automatically converted to a pointer to its first element. Of course, we are not restricted to passing a pointer to the beginning of the array. We initialize `s3` to hold four exclamation points by passing a pointer to the first exclamation point in `c_array`.

而 `s2` 的初始化式则通过第二种构造函数实现, 它的参数包括一个指针和一个计数器。在这个例子中, 从参数指针指向的那个字符开始, 连续复制第二个参数指定数目的字符。因此, `s2` 是 `c_array` 数组前 5 个字符的副本。记住, 将数组作为参数传递时, 数组将自动转换为指向其第一个元素的指针。当然, 并没有限制非得传递指向数组起点的指针不可。通过给 `s3` 的构造函数传递指向 `c_array` 数组中第一个感叹号字符的指针, `s3` 被初始化为存储 4 个感叹号字符的 `string` 对象。

The initializers for `s4` and `s5` are not C-style strings. The definition of `s4` is an error. This form of initialization may be called only with a null-terminated array. Passing an array that does not contain a null is a serious error ([Section 4.3](#), p. 130), although it is an error that the compiler cannot detect. What happens at run time is undefined.

`s4` 和 `s5` 的初始化式并不是 C 风格字符串。其中, `s4` 的定义是错误的。调用这种形式的初始化, 其参数必须是以空字符结束的数组。将不包含 `null` 的数组传递给构造函数将导致编译器无法检测的严重错误 (第 4.3 节), 此类错误在运行时将发生什么状况并未定义。

The initialization of `s5` is fine: That initializer includes a count that says how many characters to copy. As long as the count is within the size of the array, it doesn't matter whether the array is null-terminated.

`s5` 的初始化则是正确的: 初始化式包含了一个计数器, 以说明要复制多少个字符用于初始化。该计数器的值必须小于数组的长度, 此时, 无论数组是否以空字符结束, 都没什么关系。

Table 9.13. Additional Ways to Construct `string`s

9.13. 构造 `string` 对象的其他方法

<code>string s(cp, n)</code>	Create <code>s</code> as a copy of <code>n</code> characters from array pointed to by <code>cp</code> . 创建一个 <code>string</code> 对象, 它被初始化为 <code>cp</code> 所指向数组的前 <code>n</code> 个元素的副本
<code>string s(s2, pos2)</code>	Create <code>s</code> as a copy of characters in the <code>string</code> <code>s2</code> starting at index <code>pos2</code> . Undefined if <code>pos2 > s2.size()</code> . 创建一个 <code>string</code> 对象, 它被初始化为一个已存在的 <code>string</code> 对象 <code>s2</code> 中从下标 <code>pos2</code> 开始的字符的副本
<code>string s(s2, pos2, len2)</code>	Create <code>s</code> as a copy of <code>len2</code> characters from <code>s2</code> starting at index <code>pos2</code> . Undefined if <code>pos2 > s2.size()</code> . Regardless of the value of <code>len2</code> , copies at most <code>s2.size() - pos2</code> characters. 创建一个 <code>string</code> 对象, 它被初始化为 <code>s2</code> 中从下标 <code>pos2</code> 开始的 <code>len2</code> 个字符的副本。如果 <code>pos2 > s2.size()</code> , 则该操作未定义, 无论 <code>len2</code> 的值是多少, 最多只能复制 <code>s2.size() - pos2</code> 个字符

Note: `n`, `len2` and `pos2` are all `unsigned` values.

注意: `n`、`len2` 和 `pos2` 都是 `unsigned` 值

Using a Substring as the Initializer

用子串做初始化式

The other pair of constructors allow us to create a `string` as a copy of a substring of the characters in another `string`:

另一对构造函数使用程序员可以在创建 `string` 对象时将其初始化为另一个 `string` 对象的子串。

Section 9.6. strings Revisited

```
string s6(s1, 2); // s6 == "ya"
string s7(s1, 0, 2); // s7 == "Hi"
string s8(s1, 0, 8); // s8 == "Hiya"
```

The first two arguments are the `string` from which we want to copy and a starting position. In the two-argument version, the newly created `string` is initialized with the characters from that position to the end of the `string` argument. We can also provide a third argument that specifies how many characters to copy. In this case, we copy as many characters as indicated (up to the size of the `string`), starting at the specified position. For example, when we create `s7`, we copy two characters from `s1`, starting at position zero. When we create `s8`, we copy only four characters, not the requested nine. Regardless of how many characters we ask to copy, the library copies up to the size of the `string`, but not more.

第一个语句的两个参数指定了要复制的 `string` 对象及其复制的起点。在两个参数的构造函数版本中，复制 `string` 对象实参中从指定位置到其末尾的所有字符，用于初始化新创建的 `string` 对象。还可以为此类构造函数提供第三个参数，用于指定复制字符的个数。在本例中，我们从指定位置开始复制指定数目（最多为 `string` 对象的长度）的字符数。例如，创建 `s7` 时，从 `s1` 中下标为 0 的位置开始复制两个字符；而创建 `s8` 时，只复制了 4 个字符，而并不是要求的 8 个字符。无论要求复制多少个字符，标准库最多只能复制数目与 `string` 对象长度相等的字符。

9.6.2. 修改 `string` 对象的其他方法

Many of the container operations that `string` supports operate in terms of iterators. For example, `erase` takes an iterator or iterator range to specify which element(s) to remove from the container. Similarly, the first argument to each version of `insert` takes an iterator to indicate the position before which to insert the values represented by the other arguments. Although `string` supports these iterator-based operations, it also supplies operations that work in terms of an index. The index is used to indicate the starting element to `erase` or the position before which to `insert` the appropriate values. [Table 9.14](#) lists the operations that are common to both `string` and the containers; [Table 9.15](#) on the facing page lists the `string`-only operations.

`string` 类型支持的许多容器操作在操作时都以迭代器为基础。例如，`erase` 操作需要一个迭代器或一段迭代器范围作其参数，用于指定从容器中删除的元素。类似地，所有版本的 `insert` 函数的第一参数都是一个指向插入位置之后的迭代器，而新插入的元素值则由其他参数指定。尽管 `string` 类型支持这些基于迭代器的操作，它同样也提供以下基础的操作。下标用于指定 `erase` 操作的起始元素，或在其前面 `insert` 适当值的元素。表 9.14 列出了 `string` 类型和容器类型共有的操作；而表 9.15 则列出了 `string` 类型特有的操作。

Table 9.14. `string` Operations in Common with the Containers

表 9.14 与容器共有的 `string` 操作

<code>s.insert(p, t)</code>	Insert copy of value <code>t</code> before element referred to by iterator <code>p</code> . 在迭代器 <code>p</code> 指向的元素之前插入一个值为 <code>t</code> 的新元素。返回指向新插入元素的迭代器。 Returns an iterator referring to the inserted element.
<code>s.insert(p, n, t)</code>	Insert <code>n</code> copies of <code>t</code> before <code>p</code> . Returns <code>void</code> . 在迭代器 <code>p</code> 指向的元素之前插入 <code>n</code> 个值为 <code>t</code> 的新元素。返回 <code>void</code>
<code>s.insert(p, b, e)</code>	Insert elements in range denoted by iterators <code>b</code> and <code>e</code> before <code>p</code> . 在迭代器 <code>p</code> 指向的元素之前插入迭代器 <code>b</code> 和 <code>e</code> 标记范围内所有的元素。返回 <code>void</code> Returns <code>void</code> .
<code>s.assign(b, e)</code>	Replace <code>s</code> by elements in range denoted by <code>b</code> and <code>e</code> . For <code>string</code> , returns <code>s</code> , for the containers, returns <code>void</code> . 在迭代器 <code>b</code> 和 <code>e</code> 标记范围内的元素替换 <code>s</code> 。对于 <code>string</code> 类型，该操作返回 <code>s</code> ；对于容器类型，则返回 <code>void</code>
<code>s.assign(n, t)</code>	Replace <code>s</code> by <code>n</code> copies of value <code>t</code> . For <code>string</code> , returns <code>s</code> , for the containers, returns <code>void</code> . 用值为 <code>t</code> 的 <code>n</code> 个副本替换 <code>s</code> 。对于 <code>string</code> 类型，该操作返回 <code>s</code> ；对于容器类型，则返回 <code>void</code>
<code>s.erase(p)</code>	Erase element referred to by iterator <code>p</code> . 删除迭代器 <code>p</code> 指向的元素。返回一个迭代器，指向被删除元素后面的元素 Returns an iterator to the element after the one deleted.
<code>s.erase(b, e)</code>	Remove elements in range denoted by <code>b</code> and <code>e</code> .

删除迭代器 `b` 和 `e` 标记范围内所有的元素。返回一个迭代器，指向被删除元素段后面的第一个元素

Returns an iterator to the first element after the range deleted.

Table 9.15. `string`-Specific Versions表 9.15 `string` 类型特有的版本

<code>s.insert(pos, n, c)</code>	Insert <code>n</code> copies of character <code>c</code> before element at index <code>pos</code> . 在下标为 <code>pos</code> 的元素之前插入 <code>n</code> 个字符 <code>c</code>
<code>s.insert(pos, s2)</code>	Insert copy of <code>string</code> <code>s2</code> before <code>pos</code> . 在下标为 <code>pos</code> 的元素之前插入 <code>string</code> 对象 <code>s2</code> 的副本
<code>s.insert(pos, s2, pos2, len)</code>	Insert <code>len</code> characters from <code>s2</code> starting at <code>pos2</code> before <code>pos</code> . 在下标为 <code>pos</code> 的元素之前插入 <code>s2</code> 中从下标 <code>pos2</code> 开始的 <code>len</code> 个字符
<code>s.insert(pos, cp, len)</code>	Insert <code>len</code> characters from array pointed to by <code>cp</code> before <code>pos</code> . 在下标为 <code>pos</code> 打元素之前插入 <code>cp</code> 所指向数组的前 <code>len</code> 个字符
<code>s.insert(pos, cp)</code>	Insert copy of null-terminated string pointed to by <code>cp</code> before <code>pos</code> . 在下标为 <code>pos</code> 的元素之前插入 <code>cp</code> 所指向的以空字符结束的字符串副本
<code>s.assign(s2)</code>	Replace <code>s</code> by a copy of <code>s2</code> . 用 <code>s2</code> 的副本替换 <code>s</code>
<code>s.assign(s2, pos2, len)</code>	Replace <code>s</code> by a copy of <code>len</code> characters from <code>s2</code> starting at index <code>pos2</code> in <code>s2</code> . 用 <code>s2</code> 中从下标 <code>pos2</code> 开始的 <code>len</code> 个字符副本替换 <code>s</code>
<code>s.assign(cp, len)</code>	Replace <code>s</code> by <code>len</code> characters from array pointed to by <code>cp</code> . 用 <code>cp</code> 所指向数组的前 <code>len</code> 个字符副本替换 <code>s</code>
<code>s.assign(cp)</code>	Replace <code>s</code> by null-terminated array pointed to by <code>cp</code> . 用 <code>cp</code> 所指向的以空字符结束的字符串副本替换 <code>s</code>
<code>s.erase(pos, len)</code>	Erase <code>len</code> characters starting at <code>pos</code> . 删除从下标 <code>pos</code> 开始的 <code>len</code> 个字符
<i>Unless noted otherwise, all operations return a reference to <code>s</code>.</i>	
除非特别声明，上述所有操作都返回 <code>s</code> 的引用	

Position-Based Arguments

基于位置的实参

The `string`-specific versions of these operations take arguments similar to those of the additional constructors covered in the previous section. These operations let us deal with `strings` positionally and/or let us use arguments that are pointers to character arrays rather than `strings`.

Section 9.6. strings Revisited

`string` 类型为这些操作提供本类型特有的版本，它们接受的实参类似于在前一节介绍的补充构造函数。程序员可通过这些操作基于位置处理 `string` 对象，并／或使用指向字符数组的指针而不是 `string` 对象作实参。

For example, all containers let us specify a pair of iterators that denote a range of elements to `erase`. For `strings`, we can also specify the range by passing a starting position and count of the number of elements to `erase`. Assuming `s` is at least five characters long, we could erase the last five characters as follows:

例如，所有容器都允许程序员指定一对迭代器，用于标记删除 (`erase`) 的元素范围。对于 `string` 类型，还允许通过为 `erase` 函数传递一个起点位置和删除元素的数目，来指定删除的范围。假设 `s` 至少有 5 个元素，下面的语句用于删除 `s` 的最后 5 个字符：

```
s.erase(s.size() - 5, 5); // erase last five characters from s
```

Similarly, we can `insert` a given number of values in a container before the element referred to by an iterator. In the case of `strings`, we can specify the insertion point as an index rather than using an iterator:

类似地，对于容器类型，可在迭代器指向的元素之前插入 (`insert`) 指定数目的新值。而对于 `string` 类型，系统还允许使用下标而不是迭代器指定插入位置：

```
s.insert(s.size(), 5, '!'); // insert five exclamation points at end of s
```

Specifying the New Contents

指定新的内容

The characters to `insert` or `assign` into the `string` can be taken from a character array or another `string`. For example, we can use a null-terminated character array as the value to `insert` or `assign` into a `string`:

在 `string` 对象中 `insert` 或 `assign` 的字符可来自于字符数组或另一个 `string` 对象。例如，以空字符结束的字符数组可以用作 `insert` 或 `assign` 到 `string` 对象的内容：

```
char *cp = "Stately plump Buck";
string s;
s.assign(cp, 7); // s == "Stately"
s.insert(s.size(), cp + 7); // s == "Stately plump Buck"
```

Similarly, we can `insert` a copy of one `string` into another as follows:

类似地，可如下所示将一个 `string` 对象的副本插入到另一个 `string` 对象中：

```
s = "some string";
s2 = "some other string";
// 3 equivalent ways to insert all the characters from s2 at beginning of s
// insert iterator range before s.begin()
s.insert(s.begin(), s2.begin(), s2.end());
// insert copy of s2 before position 0 in s
s.insert(0, s2);
// insert s2.size() characters from s2 starting at s2[0] before s[0]
s.insert(0, s2, 0, s2.size());
```

9.6.3. `string`-Only Operations

9.6.3. 只适用于 `string` 类型的操作

The `string` type provides several other operations that the containers do not:

`string` 类型提供了容器类型不支持其他几种操作，如表 9.16 所示：

- The `substr` function that returns a substring of the current `string`
`substr` 函数，返回当前 `string` 对象的子串。
- The `append` and `replace` functions that modify the `string`
`append` 和 `replace` 函数，用于修改 `string` 对象。
- A family of `find` functions that search the `string`

Section 9.6. strings Revisited

一系列 `find` 函数，用于查找 `string` 对象。

The `substr` Operation

substr 操作

The `substr` operation lets us retrieve a substring from a given `string`. We can pass `substr` a starting position and a count. It creates a new `string` that has that many characters, (up to the end of the `string`) from the target `string`, starting at the given position:

使用 `substr` 操作可在指定 `string` 对象中检索需要的子串。我们可以给 `substr` 函数传递查找的起点和一个计数器。该函数将生成一个新的 `string` 对象，包含原目标 `string` 对象从指定位置开始的若干个字符（字符数目由计数器决定，但最多只能到原 `string` 对象的最后一个字符）：

```
string s("hello world");
// return substring of 5 characters starting at position 6
string s2 = s.substr(6, 5); // s2 = world
```

Alternatively, we could obtain the same result by writing:

可选择另一种方法实现相同的功能：

```
// return substring from position 6 to the end of s
string s3 = s.substr(6); // s3 = world
```

Table 9.16. Substring Operation

表 9.16 子串操作

<code>s.substr(pos, n)</code>	Return a <code>string</code> containing <code>n</code> characters from <code>s</code> starting at <code>pos</code> .
	返回一个 <code>string</code> 类型的字符串，它包含 <code>s</code> 中从下标 <code>pos</code> 开始的 <code>n</code> 个字符
<code>s.substr(pos)</code>	Return a <code>string</code> containing characters from <code>pos</code> to the end of <code>s</code> .
	返回一个 <code>string</code> 类型的字符串，它包含从下标 <code>pos</code> 开始到 <code>s</code> 末尾的所有字符
<code>s.substr()</code>	Return a copy of <code>s</code> .
	返回 <code>s</code> 的副本

The `append` and `replace` Functions

append 和 replace 函数

There are six overloaded versions of `append` and ten versions of `replace`. The `append` and `replace` functions are overloaded using the same set of arguments, which are listed in [Table 9.18](#) on the next page. These arguments specify the characters to add to the `string`. In the case of `append`, the characters are added at the end of the `string`. In the `replace` function, these characters are inserted in place a specified range of existing characters in the `string`.

`string` 类型提供了 6 个 `append` 重载函数版本和 10 个 `replace` 版本（见表 9.17）。`append` 和 `replace` 函数使用了相同的参数集合实现重载。这些参数如表 9.18 所示，用于指定在 `string` 对象中添加的字符。对于 `append` 操作，字符将添加在 `string` 对象的末尾。而 `replace` 函数则将这些字符插入到指定位置，从而替换 `string` 对象中一段已存在的字符。

The `append` operation is a shorthand way of inserting at the end:

```
string s("C++ Primer"); // initialize s to "C++ Primer"
s.append(" 3rd Ed."); // s == "C++ Primer 3rd Ed."
// equivalent to s.append(" 3rd Ed.")
s.insert(s.size(), " 3rd Ed.');
```

Section 9.6. strings Revisited

The `replace` operations remove an indicated range of characters and insert a new set of characters in their place. The `replace` operations have the same effect as calling `erase` and `insert`.

The ten different versions of `replace` differ from each other in how we specify the characters to remove and in how we specify the characters to insert in their place. The first two arguments specify the range of elements to remove. We can specify the range either with an iterator pair or an index and a count. The remaining arguments specify what new characters to insert.

`string` 类型为 `replace` 操作提供了 10 个不同版本，其差别在于以不同的方式指定要删除的字符和要插入的新字符。前两个参数应指定删除的元素范围，可用迭代器对实现，也可用一个下标和一个计数器实现。其他的参数则用于指定插入的新字符。

We can think of `replace` as a shorthand way of erasing some characters and inserting others in their place:

可将 `replace` 视为删除一些字符然后在同一位置插入其他内容的捷径：

Table 9.17. Operations to Modify `string`s (args defined in Table 9.18)

表 9.17 修改 `string` 对象的操作 (args 在表 9.18 中定义)

<code>s.append(args)</code>	Append <code>args</code> to <code>s</code> . Returns reference to <code>s</code> . 将 <code>args</code> 串联在 <code>s</code> 后面。返回 <code>s</code> 引用
<code>s.replace(pos, len, args)</code>	Remove <code>len</code> characters from <code>s</code> starting at <code>pos</code> and replace them by characters formed by <code>args</code> . Returns reference to <code>s</code> . 删除 <code>s</code> 中从下标 <code>pos</code> 开始的 <code>len</code> 个字符，用 <code>args</code> 指定的字符替换之。返回 <code>s</code> 的引用 This version does not take args equal to b2, e2. 在这个版本中， <code>args</code> 不能为 <code>b2, e2</code>
<code>s.replace(b, e, args)</code>	Remove characters in the range denoted by iterators <code>b</code> and <code>e</code> and replace them by <code>args</code> . Returns reference to <code>s</code> . 删除迭代器 <code>b</code> 和 <code>e</code> 标记范围内所有的字符，用 <code>args</code> 替换之。返回 <code>s</code> 的引用 This version does not take args equal to s2, pos2, len2. 在这个版本中， <code>args</code> 不能为 <code>s2, pos2, len2</code>

```
// starting at position 11, erase 3 characters and then insert "4th"
s.replace(11, 3, "4th");           // s == "C++ Primer 4th Ed."
// equivalent way to replace "3rd" by "4th"
s.erase(11, 3);                  // s == "C++ Primer Ed."
s.insert(11, "4th");             // s == "C++ Primer 4th Ed."
```

There is no requirement that the size of the text removed and inserted be the same.

`append` 操作提供了在字符串尾部插入的捷径：



In the previous call to `replace`, the text we inserted happens to be the same size as the text we removed. We could insert a larger or smaller `string`:

`replace` 操作用于删除一段指定范围的字符，然后在删除位置插入一组新字符，等效于调用 `erase` 和 `insert` 函数。

```
s.replace(11, 3, "Fourth"); // s == "C++ Primer Fourth Ed."
```

In this call we remove three characters but insert six in their place.

在这个例子中，删除了 3 个字符，但在同一个位置却插入了 6 个字符。

Table 9.18. Arguments to `append` and `replace`表 9.18 `append` 和 `replace` 操作的参数: `args`

<code>s2</code>	The <code>string</code> <code>s2</code> . string 类型的字符串 <code>s2</code>
<code>s2, pos2, len2</code>	up to <code>len2</code> characters from <code>s2</code> starting at <code>pos2</code> . 字符串 <code>s2</code> 中从下标 <code>pos2</code> 开始的 <code>len2</code> 个字符
<code>cp</code>	Null-terminated array pointed to by pointer <code>cp</code> . 指针 <code>cp</code> 指向的以空字符结束的数组
<code>cp, len2</code>	up to <code>len2</code> characters from character array pointed to by <code>cp</code> . <code>cp</code> 指向的以空字符结束的数组中前 <code>len2</code> 个字符
<code>n, c</code>	<code>n</code> copies of character <code>c</code> . 字符 <code>c</code> 的 <code>n</code> 个副本
<code>b2, e2</code>	Characters in the range formed by iterators <code>b2</code> and <code>e2</code> . 迭代器 <code>b2</code> 和 <code>e2</code> 标记的范围内所有字符

9.6.4. `string` Search Operations

9.6.4. `string` 类型的查找操作

The `string` class provides six search functions, each named as a variant of `find`. The operations all return a `string::size_type` value that is the index of where the match occurred, or a special value named `string::npos` if there is no match. The `string` class defines `npos` as a value that is guaranteed to be greater than any valid index.

`string` 类提供了 6 种查找函数 (表 9.19)，每种函数以不同形式的 `find` 命名。这些操作全都返回 `string::size_type` 类型的值，以下标形式标记查找匹配所发生的位置；或者返回一个名为 `string::npos` 的特殊值，说明查找没有匹配。`string` 类将 `npos` 定义为保证大于任何有效下标的值。

There are four versions of each of the search operations, each of which takes a different set of arguments. The arguments to the search operations are listed in [Table 9.20](#). Basically, these operations differ as to whether they are looking for a single character, another `string`, a C-style, null-terminated string, or a given number of characters from a character array.

每种查找操作都有 4 个重载版本，每个版本使用不同的参数集合。表 9.20 列出了查找操作使用的不同参数形式。基本上，这些操作的不同之处在于查找的到底是单个字符、另一个 `string` 字符串、C 风格的以空字符结束的字符串，还是用字符数组给出的特定数目的字符集合。

Table 9.19. `string` Search Operations (Arguments in [Table 9.20](#))

<code>s.find(args)</code>	Find first occurrence of <code>args</code> in <code>s</code> . 在 <code>s</code> 中查找 <code>args</code> 的第一次出现
<code>s.rfind(args)</code>	Find last occurrence of <code>args</code> in <code>s</code> . 在 <code>s</code> 中查找 <code>args</code> 的最后一次出现
<code>s.find_first_of(args)</code>	Find first occurrence of any character from <code>args</code> in <code>s</code> . 在 <code>s</code> 中查找 <code>args</code> 的任意字符的第一次出现
<code>s.find_last_of(args)</code>	Find last occurrence of any character from <code>args</code> in <code>s</code> . 在 <code>s</code> 中查找 <code>args</code> 的任意字符的最后一次出现
<code>s.find_first_not_of(args)</code>	Find first character in <code>s</code> that is not in <code>args</code> . 在 <code>s</code> 中查找第一个不属于 <code>args</code> 的字符
<code>s.find_last_not_of(args)</code>	Find last character in <code>s</code> that is not in <code>args</code> .

在 `s` 中查找最后一个不属于 `args` 的字符

Table 9.20. Arguments to `string` `find` Operations

`string` 类型提供的 `find` 操作的参数

<code>c, pos</code>	Look for the character <code>c</code> starting at position <code>pos</code> in <code>s</code> . <code>pos</code> defaults to 0. 在 <code>s</code> 中, 从下标 <code>pos</code> 标记的位置开始, 查找字符 <code>c</code> 。 <code>pos</code> 的默认值为 0
<code>s2, pos</code>	Look for the <code>string</code> <code>s2</code> starting at position <code>pos</code> in <code>s</code> . <code>pos</code> defaults to 0. 在 <code>s</code> 中, 从下标 <code>pos</code> 标记的位置开始, 查找 <code>string</code> 对象 <code>s2</code> 。 <code>pos</code> 的默认值为 0
<code>cp, pos</code>	Look for the C-style null-terminated string pointed to by the pointer <code>cp</code> . 在 <code>s</code> 中, 从下标 <code>pos</code> 标记的位置形参, 查找指针 <code>cp</code> 所指向的 C 风格的以空字符结束的字符串。 <code>pos</code> 的默认值为 0
<code>cp, pos, n</code>	Start looking starting at position <code>pos</code> in <code>s</code> . <code>pos</code> defaults to 0. Look for the first <code>n</code> characters in the array pointed to by the pointer <code>cp</code> . 在 <code>s</code> 中, 从下标 <code>pos</code> 标记的位置开始, 查找指针 <code>cp</code> 所指向数组的前 <code>n</code> 个字符。 <code>pos</code> 和 <code>n</code> 都没有默认值 Start looking starting at position <code>pos</code> in <code>s</code> . No default for <code>pos</code> or <code>n</code> .

Finding an Exact Match

精确匹配的查找

The simplest of the search operations is the `find` function. It looks for its argument and returns the index of the first match that is found, or `npos` if there is no match:

最简单的查找操作是 `find` 函数, 用于寻找实参指定的内容。如果找到的话, 则返回第一次匹配的下标值; 如果找不到, 则返回 `npos`:

```
string name("AnnaBelle");
string::size_type pos1 = name.find("Anna"); // pos1 == 0
```

Returns 0, the index at which the substring "Anna" is found in "AnnaBelle".

返回 0, 这是子串"Anna"位于字符串"AnnaBelle"中的下标。



By default, the `find` operations (and other `string` operations that deal with characters) use the built-in operators to compare characters in the `string`. As a result, these operations (and other `string` operations) are case sensitive.

默认情况下, `find` 操作 (以及其他处理字符的 `string` 操作) 使用内置操作符比较 `string` 字符串中的字符。因此, 这些操作 (以及其他 `string` 操作) 都区分字母的大小写。

When we look for a value in the `string`, case matters:

以下程序寻找 `string` 对象中的某个值, 字母的大小写影响了程序结果:

```
string lowercase("annabelle");
pos1 = lowercase.find("Anna"); // pos1 == npos
```

This code will set `pos2` to `npos` the `string` `Anna` does not match `anna`.

Section 9.6. strings Revisited

这段代码使 `pos1` 的值为 `npos` ——字符串 `Anna` 与 `anna` 不匹配



The `find` operations return a `string::size_type`. Use an object of that type to hold the return from `find`.

`find` 操作的返回类型是 `string::size_type`, 请使用该类型的对象存储 `find` 的返回值。

Find Any Character

查找任意字符

A slightly more complicated problem would be if we wanted to match any character in our search string. For example, the following locates the first digit within `name`:

如果在查找字符串时希望匹配任意指定的字符，则实现起来稍微复杂一点。例如，下面的程序要在 `name` 中寻找并定位第一个数字：

```
string numerics("0123456789");
string name("r2d2");
string::size_type pos = name.find_first_of(numerics);
cout << "found number at index: " << pos
    << " element is " << name[pos] << endl;
```

In this example, `pos` is set to a value of 1 (the elements of a `string`, remember, are indexed beginning at 0).

在这个例子中，`pos` 的值被设置为 1（记住，`string` 对象的元素下标从 0 开始计数）。

Specifying Where to Start the Search

指定查找的起点

We can pass an optional starting position to the `find` operations. This optional argument indicates the index position from which to start the search. By default, that position is set to zero. One common programming pattern uses this optional argument to loop through a `string` finding all occurrences. We could rewrite our search of `"r2d2"` to find all the numbers in `name`:

程序员可以给 `find` 操作传递一个可选的起点位置实参，用于指定开始查找的下标位置，该位置实参的默认值为 0。通常的编程模式是使用这个可选的实参循环查找 `string` 对象中所有的匹配。下面的程序重写了查找“r2d2”的程序，以便找出 `name` 字符串中出现的所有数字：

```
string::size_type pos = 0;
// each trip reset pos to the next instance in name
while ((pos = name.find_first_of(numerics, pos))
       != string::npos) {
    cout << "found number at index: " << pos
        << " element is " << name[pos] << endl;
    ++pos; // move to the next character
}
```

In this case, we initialize `pos` to zero so that on the first trip through the `while` `name` is searched, beginning at position 0. The condition in the `while` resets `pos` to the index of the first number encountered, starting from the current value of `pos`. As long as the return from `find_first_of` is a valid index, we print our result and increment `pos`.

在这个例子中，首先将 `pos` 初始化为 0，使第一次循环从 0 号元素开始查找 `name`。`while` 的循环条件实现两个功能：从当前 `pos` 位置开始查找，并将找到的第一个数字出现的下标值赋给 `pos`。当 `find_first_of` 函数返回有效的下标值时，输出此次查找的结果，并且让 `pos` 加 1。

Had we neglected to increment `pos` at the end of this loop, then it would never terminate. To see why, consider what would happen if we didn't. On the second trip through the loop, we start looking at the character indexed by `pos`. That character would be a number, so `find_first_of` would (repeatedly) returns `pos`!

如果漏掉了循环体末尾让 `pos` 加 1 的语句，那么循环永远都不会结束。考虑没有该操作时，会发生什么情况？第二次循环时，从 `pos` 标记的位置开始查找，而此时 `pos` 标记的就是一个数字，于是 `find_first_of` 函数将（不断重复地）返回同一个 `pos` 值。

It is essential that we increment `pos`. Doing so ensures that we start looking for the next number at a point after the number we just found.



Looking for a Nonmatch

寻找不匹配点

Instead of looking for a match, we might call `find_first_not_of` to find the first position that is *not* in the search argument. For example, to find the first non-numeric character of a `string`, we can write

除了寻找匹配的位置外，还可以调用 `find_first_not_of` 函数查找第一个与实参不匹配的位置。例如，如果要在 `string` 对象中寻找第一个非数字字符，可以如下编写程序：

```
string numbers("0123456789");
string dept("03714p3");

// returns 5, which is the index to the character 'p'
string::size_type pos = dept.find_first_not_of(numbers);
```

Searching Backward

反向查找

Each of the `find` operations that we've seen so far executes left to right. The library provides an analogous set of operations that look through the `string` from right to left. The `rfind` member searches for the last that is, rightmost occurrence of the indicated substring:

迄今为止，我们使用的所有 `find` 操作都是从左向右查找的。除此之外，标准库还提供了一组类似的从右向左查找 `string` 对象的操作。`rfind` 成员函数用于寻找最后一个——也就是最右边的——指定子串出现的位置：

```
string river("Mississippi");
string::size_type first_pos = river.find("is"); // returns 1
string::size_type last_pos = river.rfind("is"); // returns 4
```

`find` returns an index of 1, indicating the start of the first "is", while `rfind` returns an index of 4, indicating the start of the last occurrence of "is".

`find` 函数返回下标 1，标记 `river` 字符串中第一个"is"的出现位置；而 `rfind` 函数返回最后一个匹配的位置，而并不是第一个。

The `find_last` Functions

The `find_last` functions operate like the corresponding `find_first` functions, except that they return the *last* match rather than the first:

- `find_last_of` searches for the last character that matches any element of the search `string`.
`find_last_of` 函数查找与目标字符串的任意字符匹配的最后一个字符。
- `find_last_not_of` searches for the last character that does not match any element of the search `string`.
`find_last_not_of` 函数查找最后一个不能跟目标字符串的任何字符匹配的字符。

Each of these operations takes an optional second argument indicating the position within the `string` to begin searching.

这两个操作都提供第二个参数，这个参数是可选的，用于指定在 `string` 对象中开始查找的位置。

9.6.5. Comparing `string`s

9.6.5.

As we saw in [Section 3.2.3](#) (p. 85), the `string` type defines all the relational operators so that we can compare two `strings` for equality (`==`), inequality (`!=`), and the less- or greater-than operations (`<`, `<=`, `>`, `>=`). Comparison between `strings` is lexicographical that is, `string` comparison is the same as a case-sensitive, dictionary ordering:

正如在第 3.2.3 节所看到的, `string` 类型定义了所有关系操作符, 使程序员可以比较两个 `string` 对象是否相等 (`==`)、不等 (`!=`) , 以及实现小于或大于 (`<`、`<=`、`>`、`>=`) 运算。`string` 对象采用字典顺序比较, 也就是说, `string` 对象的比较与大小写敏感的字典顺序比较相同:

```
string cobol_program_crash("abend");
string cplus_program_crash("abort");
```

Exercises Section 9.6.4

Exercise

9.38: Write a program that, given the `string`

已知有如下 `string` 对象:

```
"ab2c3d7R4E6"
```

finds each numeric character and then each alphabetic character. Write two versions of the program. The first should use `find_first_of`, and the second `find_first_not_of`.

编写程序寻找该字符串中所有的数字字符, 然后再寻找所有的字母字符。以两种版本编写该程序: 第一个版本使用 `find_first_of` 函数, 而第二个版本则使用 `find_first_not_of` 函数。

Exercise

9.39: Write a program that, given the `strings`

已知有如下 `string` 对象:

```
string line1 = "We were her pride of 10 she named us:";
string line2 = "Benjamin, Phoenix, the Prodigal"
string line3 = "and perspicacious pacific Suzanne";

string sentence = line1 + ' ' + line2 + ' ' + line3;
```

counts the number of words in `sentence` and identifies the largest and smallest words. If several words have the largest or smallest length, report all of them.

编写程序计算 `sentence` 中有多少个单词, 并指出其中最长和最短的单词。如果有多个最长或最短的单词, 则将它们全部输出。

Here `cobol_program_crash` is less than the `cplus_program_crash`. The relational operators compare two `strings` character by character until reaching a position where the two `strings` differ. The overall comparison of the `strings` depends on the comparison between these unequal characters. In this case, the first unequal characters are '`e`' and '`o`'. The letter '`e`' occurs before (is less than) '`o`' in the English alphabet and so "`abend`" is less than "`abort`". If the `strings` are of different length, and one `string` is a substring of the other, then the shorter `string` is less than the longer.

操作符逐个字符地进行比较, 直到比较到某个位置上, 两个 `string` 对象对应的字符不相同为止。`string` 对象的整个比较依赖于不相同字符之间的比较。在本例中, 第一个不相等的字符是'`e`'和'`o`'。由于在英文字母表中, '`e`'出现得比'`o`'早(即'`e`'小于'`o`') , 于是"`abend`"小于"`abort`"。如果要比较的两个 `string` 对象长度不相同, 而且一个 `string` 对象是另一个 `string` 对象的子串, 则较短的 `string` 对象小于较长的 `string` 对象。

The `compare` Functions

compare 函数

In addition to the relational operators, `string` provides a set of `compare` operations that perform lexicographical comparisons. The results of these operations are similar to the C library `strcmp` function ([Section 4.3](#), p. 132). Given

除了关系操作符, `string` 类型还提供了一组 `compare` 操作 (表 9.21) , 用于实现字典顺序的比较。这些操作的结果类似于 C 语言中的库函数 `strcmp` (第 4.3 节) 。假设有关语句:

```
s1.compare (args);
```

`compare` returns one of three possible values:

`compare` 函数返回下面列出的三种可能值之一：

1. A positive value if `s1` is greater than the `string` represented by `args`

正数，此时 `s1` 大于 `args` 所代表的 `string` 对象。

2. A negative value if `s1` is less than the `string` represented by `args`

负数，此时 `s1` 小于 `args` 所代表的 `string` 对象。

3. 0 if `s1` is equal to the `string` represented by `args`

0，此时 `s1` 恰好等于 `args` 所代表的 `string` 对象。

For example

例如：

```
// returns a negative value
cobol_program_crash.compare(cplus_program_crash);
// returns a positive value
cplus_program_crash.compare(cobol_program_crash);
```

Table 9.21. `string` compare Operations

表 9.21 `string` 类型 `compare` 操作

<code>s.compare(s2)</code>	Compare <code>s</code> to <code>s2</code> . 比较 <code>s</code> 和 <code>s2</code>
<code>s.compare(pos1, n1, s2)</code>	Compares <code>n1</code> characters starting at <code>pos1</code> from <code>s</code> to <code>s2</code> . 让 <code>s</code> 中从 <code>pos</code> 下标位置开始的 <code>n1</code> 个字符与 <code>s2</code> 做比较
<code>s.compare(pos1, n1, s2, pos2, n2)</code>	Compares <code>n1</code> characters starting at <code>pos1</code> from <code>s</code> to the <code>n2</code> characters starting at <code>pos2</code> in <code>s2</code> . 让 <code>s</code> 中从 <code>pos1</code> 下标位置开始的 <code>n1</code> 个字符与 <code>s2</code> 中从 <code>pos2</code> 下标位置开始的 <code>n2</code> 个字符做比较
<code>s.compare(cp)</code>	Compares <code>s</code> to the null-terminated string pointed to by <code>cp</code> . 比较 <code>s</code> 和 <code>cp</code> 所指向的以空字符结束的字符串
<code>s.compare(pos1, n1, cp)</code>	Compares <code>n1</code> characters starting at <code>pos1</code> from <code>s</code> to <code>cp</code> . 让 <code>s</code> 中从 <code>pos1</code> 下标位置开始的 <code>n1</code> 个字符与 <code>cp</code> 所指向的字符串做比较
<code>s.compare(pos1, n1, cp, n2)</code>	Compares <code>n1</code> characters starting at <code>pos1</code> from <code>s</code> to <code>n2</code> characters starting from the pointer <code>cp</code> . 让 <code>s</code> 中从 <code>pos1</code> 下标位置开始的 <code>n1</code> 个字符与 <code>cp</code> 所指向的字符串的前 <code>n2</code> 个字符做比较

The overloaded set of six `compare` operations allows us to compare a substring of either one or both `strings` for comparison. They also let us compare a `string` to a character array or portion thereof:

`compare` 操作提供了 6 种重载函数版本，以方便程序员实现一个或两个 `string` 对象的子串的比较，以及 `string` 对象与字符数组或其中某一部分的比较：

```
char second_ed[] = "C++ Primer, 2nd Edition";
```

Section 9.6. strings Revisited

```
string third_ed("C++ Primer, 3rd Edition");
string fourth_ed("C++ Primer, 4th Edition");

// compares C++ library string to C-style string
fourth_ed.compare(second_ed); // ok, second_ed is null-terminated

// compare substrings of fourth_ed and third_ed
fourth_ed.compare(fourth_ed.find("4th"), 3,
                   third_ed, third_ed.find("3rd"), 3);
```

The second call to `compare` is the most interesting. This call uses the version of `compare` that takes five arguments. We use `find` to locate the position of the beginning of the substring "4th". We compare three characters starting at that position to a substring from `third_ed`. That substring begins at the position returned from `find` when looking for "3rd" and again we compare three characters. Essentially, this call compares "4th" to "3rd".

我们对第二个 `compare` 函数的调用更感兴趣。这个调用使用了具有 5 个参数的 `compare` 函数版本。先调用 `find` 函数找到子串"4th"的起点。让从此起点开始的 3 个字符与 `third_ed` 的子串做比较。而 `third_ed` 的子串起始于 `find` 函数找到的"3rd"的起点，同样取其随后的 3 个字符参加比较。可见这个语句本质上比较的是"4th"和"3rd"。

Exercises Section 9.6.5

Exercise 9.40: Write a program that accepts the following two `strings`:

编写程序接收下列两个 `string` 对象：

```
string q1("When lilacs last in the dooryard bloom'd");
string q2("The child is father of the man");
```

Using the `assign` and `append` operations, create the `string`

然后使用 `assign` 和 `append` 操作，创建 `string` 对象：

```
string sentence("The child is in the dooryard");
```

Exercise 9.41: Write a program that, given the `strings`

已知有如下 `string` 对象：

```
string generic1("Dear Ms Daisy:");
string generic2("MrsMsMissPeople");
```

implements the function

编写程序实现下面函数：

```
string greet(string form, string lastname, string title,
            string::size_type pos, int length);
```

using the `replace` operations, where `lastname` replaces `Daisy` and `pos` indexes into `generic2` of `length` characters replacing `Ms`. For example, the following

该函数使用 `replace` 操作实现以下功能：对于字符串 `form`，将其中的 `Daisy` 替换为 `lastname`，将其中的 `Ms` 替换为字符串 `generic2` 中从 `pos` 下标开始的 `length` 个字符。例如，下面的语句：

```
string lastName("AnnaP");
string salute = greet(generic1, lastName, generic2, 5, 4);
```

returns the `string`

将返回字符串：

```
Dear Miss AnnaP:
```

Team LiB

◀ PREVIOUS NEXT ▶

9.7. Container Adaptors

9.7. 容器适配器

In addition to the sequential containers, the library provides three sequential container adaptors: `queue`, `priority_queue`, and `stack`. An **adaptor** is a general concept in the library. There are container, iterator, and function adaptors. Essentially, an adaptor is a mechanism for making one thing act like another. A container adaptor takes an existing container type and makes it act like a different abstract type. For example, the `stack` adaptor takes any of the sequential containers and makes it operate as if it were a `stack`. [Table 9.22](#) (p. 350) lists the operations and types that are common to all the container adaptors.

除了顺序容器，标准库还提供了三种顺序容器适配器：`queue`、`priority_queue` 和 `stack`。**适配器（adaptor）**是标准库中通用的概念，包括容器适配器、迭代器适配器和函数适配器。本质上，适配器是使一事物的行为类似于另一事物的一种机制。容器适配器让一种已存在的容器类型采用另一种不同的抽象类型的工作方式实现。例如，`stack`（栈）适配器可使任何一种顺序容器以栈的方式工作。[表 9.22](#) 列出了所有容器适配器通用的操作和类型。

Table 9.22. Common Adaptor Operations and Types

表 9.22. 适配器通用的操作和类型

<code>size_type</code>	Type large enough to hold size of largest object of this type. 一种类型，足以存储此适配器类型最大对象的长度
<code>value_type</code>	Element type. 元素类型
<code>container_type</code>	Type of the underlying container on which the adaptor is implemented. 基础容器的类型，适配器在此容器类型上实现
<code>A a;</code>	Create a new empty adaptor named <code>a</code> . 创建一个新空适配器，命名为 <code>a</code>
<code>A a(c);</code>	Create a new adaptor named <code>a</code> with a copy of the container <code>c</code> . 创建一个名为 <code>a</code> 的新适配器，初始化为容器 <code>c</code> 的副本
<i>Relational Operators</i>	Each adaptor supports all the relational operators: <code>==</code> , <code>!=</code> , <code><</code> , <code><=</code> , <code>></code> , <code>>=</code> .
关系操作符	所有适配器都支持全部关系操作符： <code>==</code> 、 <code>!=</code> 、 <code><</code> 、 <code><=</code> 、 <code>></code> 、 <code>>=</code>

To use an adaptor, we must include its associated header:

使用适配器时，必须包含相关的头文件：

```
#include <stack>    // stack adaptor
#include <queue>   // both queue and priority_queue adaptors
```

Initializing an Adaptor

适配器的初始化

Each adaptor defines two constructors: the default constructor that creates an empty object and a constructor that takes a container and makes a copy of that container as its underlying value. For example, assuming that `deq` is a `deque<int>`, we could use `deq` to initialize a new `stack` as follows:

所有适配器都定义了两个构造函数：默认构造函数用于创建空对象，而带一个容器参数的构造函数将参数容器的副本作为其基础值。例如，假设 `deq` 是 `deque<int>` 类型的容

Section 9.7. Container Adaptors

器，则可用 `deq` 初始化一个新的栈，如下所示：

```
stack<int> stk(deq); // copies elements from deq into stk
```

Overriding the Underlying Container Type

覆盖基础容器类型

By default both `stack` and `queue` are implemented in terms of `deque`, and a `priority_queue` is implemented on a `vector`. We can override the default container type by naming a sequential container as a second type argument when creating the adaptor:

默认的 `stack` 和 `queue` 都基于 `deque` 容器实现，而 `priority_queue` 则在 `vector` 容器上实现。在创建适配器时，通过将一个顺序容器指定为适配器的第二个类型实参，可覆盖其关联的基础容器类型：

```
// empty stack implemented on top of vector  
stack<string, vector<string>> str_stk;  
  
// str_stk2 is implemented on top of vector and holds a copy of svec  
stack<string, vector<string>> str_stk2(svec);
```

There are constraints on which containers can be used for a given adaptor. We can use any of the sequential containers as the underlying container for a `stack`. Thus, a `stack` can be built on a `vector`, `list`, or `deque`. The `queue` adaptor requires `push_front` in its underlying container, and so could be built on a `list` but not on a `vector`. A `priority_queue` requires random access and so can be built on a `vector` or a `deque` but not on a `list`.

对于给定的适配器，其关联的容器必须满足一定的约束条件。`stack` 适配器所关联的基础容器可以是任意一种顺序容器类型。因此，`stack` 栈可以建立在 `vector`、`list` 或者 `deque` 容器之上。而 `queue` 适配器要求其关联的基础容器必须提供 `push_front` 运算，因此只能建立在 `list` 容器上，而不能建立在 `vector` 容器上。`priority_queue` 适配器要求提供随机访问功能，因此可建立在 `vector` 或 `deque` 容器上，但不能建立在 `list` 容器上。

Relational Operations on Adaptors

适配器的关系运算

Two adaptors of the same type can be compared for equality, inequality, less-than, greater-than, less-than-equal, and greater-than-equal relationships, provided that the underlying element type supports the equality and less-than operators. For these operations, the elements are compared in turn. The first pair of unequal elements determines the less-than or greater-than relationship.

两个相同类型的适配器可以做相等、不等、小于、大于、小于等于以及大于等于关系比较，只要基础元素类型支持等于和小于操作符既可。这些关系运算由元素依次比较来实现。第一对不相等的元素将决定两者之间的小于或大于关系。

9.7.1. Stack Adaptor

9.7.1. 栈适配器

The operations provided by a `stack` are listed in [Table 9.23](#) on the facing page. The following program exercises this set of five `stack` operations:

[表 9.23](#) 列出了栈提供的所有操作。

Table 9.23. Operations Supported by the Stack Container Adaptor

表 9.23. 栈容器适配器支持的操作

<code>s.empty()</code>	Returns <code>true</code> if the <code>stack</code> is empty; <code>false</code> otherwise. 如果栈为空，则返回 <code>true</code> ，否则返回 <code>stack</code>
<code>s.size()</code>	Returns a count of the number of elements on the <code>stack</code> . 返回栈中元素的个数

Section 9.7. Container Adaptors

<code>s.pop()</code>	Removes, but does not return, the top element from the <code>stack</code> . 删除栈顶元素的值，但不返回其值
<code>s.top()</code>	Returns, but does not remove, the top element on the <code>stack</code> . 返回栈顶元素的值，但不删除该元素
<code>s.push(item)</code>	Places a new top element on the <code>stack</code> . 在栈顶压入新元素

```
// number of elements we'll put in our stack
const stack<int>::size_type stk_size = 10;
stack<int> intStack; // empty stack
// fill up the stack
int ix = 0;
while (intStack.size() != stk_size)
    // use postfix increment; want to push old value onto intStack
    intStack.push(ix++); // intStack holds 0...9 inclusive

int error_cnt = 0;
// look at each value and pop it off the stack
while (intStack.empty() == false) {
    int value = intStack.top();
    // read the top element of the stack
    if (value != --ix) {
        cerr << "oops! expected " << ix
            << " received " << value << endl;
        ++error_cnt;
    }
    intStack.pop(); // pop the top element, and repeat
}
cout << "Our program ran with "
    << error_cnt << " errors!" << endl;
```

The declaration

声明语句:

```
stack<int> intStack; // empty stack
```

defines `intStack` to be an empty `stack` that holds integer elements. The `for` loop adds `stk_size` elements initializing each to the next integer in sequence starting from zero. The `while` loop iterates through the entire `stack`, examining the `top` value and `popping` it from the `stack` until the `stack` is empty.

将 `intStack` 定义为一个存储整型元素的空栈。第一个 `while` 循环在该栈中添加了 `stk_size` 个元素，元素初值是从 0 开始依次递增 1 的整数。第二个 `while` 循环迭代遍历整个栈，检查其栈顶 (`top`) 的元素值，然后栈顶元素出栈，直到栈变空为止。

Each container adaptor defines its own operations in terms of operations provided by the underlying container type. By default, this `stack` is implemented using a `deque` and uses `deque` operations to implement the operations of a `stack`. For example, when we execute

所有容器适配器都根据其基础容器类型所支持的操作来定义自己的操作。默认情况下，栈适配器建立在 `deque` 容器上，因此采用 `deque` 提供的操作来实现栈功能。例如，执行下面的语句：

```
// use postfix increment; want to push old value onto intStack
intStack.push(ix++); // intStack holds 0...9 inclusive
```

this operation executes by calling the `push_back` operation of the `deque` object on which `intStack` is based. Although `stack` is implemented by using a `deque`, we have no direct access to the `deque` operations. We cannot call `push_back` on a `stack`; instead, we must use the `stack` operation named `push`.

这个操作通过调用 `push_back` 操作实现，而该 `intStack` 所基于的 `deque` 对象提供。尽管栈是以 `deque` 容器为基础实现的，但是程序员不能直接访问 `deque` 所提供的操作。例如，不能在栈上调用 `push_back` 函数，而是必须使用栈所提供的名为 `push` 的操作。

9.7.2. Queue and Priority Queue

9.7.2. 队列和优先级队列

The library `queue` uses a first-in, first-out (FIFO) storage and retrieval policy. Objects entering the queue are placed in the back. The next object

Section 9.7. Container Adaptors

retrieved is taken from the front of the queue. There are two kinds of queues: the FIFO queue, which we will speak of simply as a `queue`, and a priority queue.

标准队列使用了先进先出 (FIFO) 的存储和检索策略。进入队列的对象被放置在尾部，下一个被取出的元素则取自队列的首部。标准库提供了两种风格的队列：FIFO 队列 (FIFO queue, 简称 `queue`)，以及优先级队列 (priority queue)。

A `priority_queue` lets us establish a priority among the elements held in the queue. Rather than place a newly entered item at the back of the queue, the item is placed ahead of all those items with a lower priority. By default, the library uses the `<` operator on the element type to determine relative priorities.

`priority_queue` 允许用户为队列中存储的元素设置优先级。这种队列不是直接将新元素放置在队列尾部，而是放在比它优先级低的元素前面。标准库默认使用元素类型的 `<` 操作符来确定它们之间的优先级关系。

A real-world example of a priority queue is the line to check luggage at an airport. Those whose flight is going to leave within the next 30 minutes are generally moved to the front of the line so that they can finish the check-in process before their plane takes off. A programming example of a priority queue is the scheduler of an operating system determining which, of a number of waiting processes, should execute next.

优先级队列的一个实例是机场行李检查队列。30 分钟后即将离港的航班的乘客通常会被移到队列前面，以便他们能在飞机起飞前完成检查过程。使用优先级队列的程序示例是操作系统的调试表，它决定在大量等待进程中下一个要执行的进程。

To use either `queue` or `priority_queue`, we must include the `queue` header. [Table 9.24](#) lists the operations supported by `queue` and `priority_queue`.

要使用这两种队列，必须包含 `queue` 头文件。[表 9.24](#) 列出了队列和优先级队列所提供的所有操作。

Table 9.24. Operations Supported by Queues and Priority Queues

表 9.24. 队列和优先级队列支持的操作

<code>q.empty()</code>	Returns <code>true</code> if the <code>queue</code> is empty; <code>false</code> otherwise. 如果队列为空，则返回 <code>true</code> ，否则返回 <code>false</code>
<code>q.size()</code>	Returns a count of the number of elements on the <code>queue</code> . 返回队列中元素的个数
<code>q.pop()</code>	Removes, but does not return, the front element from the <code>queue</code> . 删除队首元素，但不返回其值
<code>q.front()</code>	Returns, but does not remove, the front element on the <code>queue</code> . 返回队首元素的值，但不删除该元素 This operation can be applied only to a <code>queue</code>. 该操作只适用于队列
<code>q.back()</code>	Returns, but does not remove, the back element on the <code>queue</code> . 返回队尾元素的值，但不删除该元素 This operation can be applied only to a <code>queue</code>. 该操作只适用于队列
<code>q.top()</code>	Returns, but does not remove, the highest-priority element. 返回具有最高优先级的元素值，但不删除该元素 This operation can be applied only to a <code>priority_queue</code>. 该操作只适用于优先级队列
<code>q.push(item)</code>	Places a new element at the end of the <code>queue</code> or at its appropriate position based on priority in a <code>priority_queue</code> . 对于 <code>queue</code> ，在队尾压入一个新元素，对于 <code>priority_queue</code> ，在基于优先级的适当位置插入新元素

Exercises Section 9.7.2

Exercise Write a program to read a series of words into a `stack`.

9.42:

编写程序读入一系列单词，并将它们存储在 `stack` 对象中。

Exercise Use a `stack` to process parenthesized expressions. When you see an open parenthesis, note

9.43:

that it was seen. When you see a close parenthesis after an open parenthesis, `pop` elements down to and including the open parenthesis off the `stack`. `push` a value onto the `stack` to indicate that a parenthesized expression was replaced.

使用 `stack` 对象处理带圆括号的表达式。遇到左圆括号时，将其标记下来。然后在遇到右加括号时，弹出 `stack` 对象中这两边括号之间的相关元素（包括左圆括号）。接着在 `stack` 对象中压入一个值，用以表明这个用一对圆括号括起来的表达式已经被替换。

Team LiB

◀ PREVIOUS NEXT ▶

Chapter Summary

小结

The C++ library defines a number of sequential container types. A container is a template type that holds objects of a given type. In a sequential container, elements are ordered and accessed by position. The sequential containers share a common, standardized interface: If two sequential containers offer a particular operation, then the operation has the same interface and meaning for both containers. All the containers provide (efficient) dynamic memory management. We may add elements to the container without worrying about where to store the elements. The container itself manages its storage.

C++ 标准库定义了一系列顺序容器类型。容器是用于存储某种给定类型对象的模板类型。在顺序容器中，所有元素根据其位置排列和访问。顺序容器共享一组通用的已标准化的接口：如果两种顺序容器都提供某一操作，那么该操作具有相同的接口和含义。所有容器都提供（有效的）动态内存管理。程序员在容器中添加元素时，不必操心元素存放在哪里。容器自己实现其存储管理。

The most commonly used container, `vector`, supports fast, random access to elements. Elements can be added and removed efficiently from the end of a `vector`. Inserting or deleting elements elsewhere can be expensive. The `deque` class is like a `vector`, but also supports fast insertion and deletion at the front of the `deque`. The `list` class supports only sequential access to elements, but it can be quite fast to insert or remove elements anywhere within the list.

最经常使用的容器类型是 `vector`，它支持对元素的快速随机访问。可高效地在 `vector` 容器尾部添加和删除元素，而在其他任何位置上的插入或删除运算则要付出比较昂贵的代价。`deque` 类与 `vector` 相似，但它还支持在 `deque` 首部的快速插入和删除运算。`list` 类只支持元素的顺序访问，但在 `list` 内部任何位置插入和删除元素都非常快速。

The containers define surprisingly few operations. Containers define constructors, operations to add or remove elements, operations to determine the size of the container, and operations to return iterators to particular elements. Other useful operations, such as sorting or searching, are defined not by the container types but by the standard algorithms, which we shall cover in [Chapter 11](#).

容器定义的操作非常少，只定义了构造函数、添加或删除元素的操作、设置容器长度的操作以及返回指向特殊元素的迭代器的操作。其他一些有用的操作，如排序、查找，则不是由容器类型定义，而是由[第十一章](#)介绍的标准算法定义。

Container operations that add or remove elements can invalidate existing iterators. When mixing actions on iterators and container operations, it is essential to keep in mind whether a given container operation could invalidate the iterators. Many operations that invalidate an iterator, such as `insert` or `erase`, return a new iterator that allows the programmer to maintain a position within the container. Loops that use container operations that change the size of a container should be particularly careful in their use of iterators.

在容器中添加或删除元素可能会使已存在的迭代器失效。当混合使用迭代器操作和容器操作时，必须时刻留意给定的容器操作是否会使迭代器失效。许多使一个迭代器失效的操作，例如 `insert` 或 `erase`，将返回一个新的迭代器，让程序员保留容器中的一个位置。使用改变容器长度的容器操作的循环必须非常小心其迭代器的使用。

Defined Terms

术语

adaptor (适配器)

A library type, function, or iterator that given a type, function, or iterator, makes it act like another. There are three sequential container adaptors: `stack`, `queue`, and `priority_queue`. Each of these adaptors defines a new interface on top of an underlying sequential container.

一种标准库类型、函数或迭代器，使某种标准库类型、函数或迭代器的行为类似于另外一种标准库类型、函数或迭代器。系统提供了三种顺序容器适配器：`stack`（栈）、`queue`（队列）以及 `priority_queue`（优先级队列）。所有的适配器都会在其基础顺序容器上定义一个新接口。

begin (begin 操作)

Container operation that returns an iterator referring to the first element in the container, if there is one, or the off-the-end iterator if the container is empty.

一种容器操作。如果容器中有元素，该操作返回指向容器中第一个元素的迭代器；如果容器为空，则返回超出末端迭代器。

container (容器)

A type that holds a collection of objects of a given type. Each library container type is a template type. To define a container, we must specify the type of the elements stored in the container. The library containers are variable-sized.

一种存储给定类型对象集合的类型。所有标准库容器类型都是模板类型。定义容器时，必须指定在该容器中存储的元素是什么类型。标准库容器具有可变的长度。

deque (双端队列)

Sequential container. Elements in a `deque` are accessed by their positional index. Like a `vector` in all respects except that it supports fast insertion at the front of the container as well as at the end and does not relocate elements as a result of insertions or deletions at either end.

一种顺序容器。`deque` 中存储的元素通过其下标位置访问。该容器类型在很多方面与 `vector` 一样，唯一的不同是 `deque` 类型支持在容器首部快速地插入新元素，就像在尾部插入一样，而且无论在容器的哪一端插入或删除都不会引起元素的重新定位。

end (end 操作)

Container operation that returns an iterator referring to the element one past the end of the container.

一种容器操作，返回指向容器的超出末端的下一位置的迭代器。

invalidated iterator (无效迭代器)

An iterator that refers to an element that no longer exists. Using an invalidated iterator is undefined and can cause serious run-time problems.

指向不再存在的元素的迭代器。无效迭代器的使用未定义，可能会导致严重的运行时错误。

iterator (迭代器)

A type whose operations support navigating among the elements of a container and examining values in the container. Each of the library containers has four companion iterator types listed in [Table 9.5](#) (p. 316). The library iterators all support the dereference (*) and arrow (->) operators to examine the value to which the iterator refers. They also support prefix and postfix increment (++) and decrement (--) and the equality (==) and inequality (!=) operators.

一种类型，其操作支持遍历和检查容器元素的操作。所有标准库容器都定义了表 9.5 列出的 4 种迭代器，与之共同工作。标准库迭代器都支持解引用 (*) 操作符和箭头 (->) 操作符，用于检查迭代器指向的元素值。它们还支持前置和后置的自增 (++)、自减操作符 (--)，以及相等 (==) 和不等 (!=) 操作符。

iterator range (迭代器范围)

A range of elements denoted by a pair of iterators. The first iterator refers to the first element in the sequence, and the second iterator refers one past the last element. If the range is empty, then the iterators are equal (and vice versa if the iterators are equal, they denote an empty range). If the range is non-empty, then it must be possible to reach the second iterator by repeatedly incrementing the first iterator. By incrementing the iterator, each element in the sequence can be processed.

Keyterm Defined Terms

由一对迭代器标记的一段元素范围。第一个迭代器指向序列中的第一个元素，而第二个迭代器则指向该范围中的最后一个元素的下一位置。如果这段范围为空，则这两个迭代器相等（反之亦然——如果这两个迭代器相等，则它们标记一个空范围）。如果这段范围非空，则对第一个空范围）。如果这段范围非空，则对第一个迭代器重複做自增运算，必然能达第二个迭代器。通过这个对迭代器进行自增的过程，即可处理该序列中所有的元素。

left-inclusive interval (左闭合区间)

A range of values that includes its first element but not its last. Typically denoted as `[i, j)` meaning the sequence starting at and including `i` up to but excluding `j`.

一段包含第一个元素但不包含最后一个元素的范围。一般表示为 `[i, j)`，意味着该序列从 `i` 开始（包括 `i`）一直到 `j`，但不包含 `j`。

list (列表)

Sequential container. Elements in a `list` may be accessed only sequentially starting from a given element, we can get to another element by incrementing or decrementing across each element between them. Supports fast insertion (or deletion) anywhere in the `list`. Adding elements does not affect other elements in the `list`; iterators remain valid when new elements are added. When an element is removed, only the iterators to that element are invalidated.

一种顺序容器。`list` 中的元素只能顺序访问——从给定元素开始，要获取另一个元素，则必须通过自增或自减迭代器的操作遍历这两个元素之间的所有元素。`list` 容器支持在容器的任何位置实现快速插入（或删除）运算。新元素的插入不会影响 `list` 中的其他元素。插入元素时，迭代器保持有效；删除元素时，只有指向该元素的迭代器失效。

priority queue (优先级队列)

Adaptor for the sequential containers that yields a queue in which elements are inserted, not at the end but according to a specified priority level. By default, priority is determined by using the less-than operator for the element type.

一种顺序容器适配器。在这种队列中，新元素不是在队列尾部插入，而是根据指定的优先级级别插入。默认情况下，元素的优先级由元素类型的小于操作符决定。

queue (队列)

Adaptor for the sequential containers that yields a type that lets us add elements to the back and remove elements from the front.

一种顺序容器适配器。在这种队列中，保证只在队尾插入新元素，而且只在队首删除元素。

sequential container (顺序容器)

A type that holds an ordered collection of objects of a single type. Elements in a sequential container are accessed by position.

以有序集合的方式存储单一类型对象的类型。顺序容器中的元素可通过下标访问。

stack (栈)

Adaptor for the sequential containers that yields a type that lets us add and remove elements from one end only.

一种顺序容器适配器，这种类型只能在一端插入和删除元素。

vector (向量)

Sequential container. Elements in a `vector` are accessed by their positional index. We add elements to a `vector` by calling `push_back` or `insert`. Adding elements to a `vector` might cause it to be reallocated, invalidating all iterators into the `vector`. Adding (or removing) an element in the middle of a `vector` invalidates all iterators to elements after the insertion (or deletion) point.

一种顺序容器。`vector` 中的元素通过其位置下标访问。可通过调用 `push_back` 或 `insert` 函数在 `vector` 中添加元素。在 `vector` 中添加元素可能会导致重新为容器分配内存空间，也可能使所有的迭代器失效。在 `vector` 容器中间添加（或删除）元素将使所有指向插入（或删除）点后面的元素的迭代器失效。

Chapter 10. Associative Containers

第十章 关联容器

CONTENTS

Section 10.1 Preliminaries: the <code>pair</code> Type	356
Section 10.2 Associative Containers	358
Section 10.3 The <code>map</code> Type	360
Section 10.4 The <code>set</code> Type	372
Section 10.5 The <code>multimap</code> and <code>multiset</code> Types	375
Section 10.6 Using Containers: Text-Query Program	379
Chapter Summary	388
Defined Terms	388

This chapter completes our review of the standard library container types by looking at the associative containers. Associative containers differ in a fundamental respect from the sequential containers: Elements in an associative container are stored and retrieved by a key, in contrast to elements in a sequential container, which are stored and accessed sequentially by their position within the container.

本章将继续介绍标准库容器类型的另一项内容——关联容器。关联容器和顺序容器的本质差别在于：关联容器通过键（key）存储和读取元素，而顺序容器则通过元素在容器中的位置顺序存储和访问元素。

Although the associative containers share much of the behavior of the sequential containers, they differ from the sequential containers in ways that support the use of keys. This chapter covers the associative containers and closes with an extended example that uses both associative and sequential containers.

虽然关联容器的大部分行为与顺序容器相同，但其独特之处在于支持键的使用。本章涵盖了关联容器的相关内容，并完善和扩展了一个使用顺序容器和关联容器的例子。

Associative containers support efficient lookup and retrieval by a key. The two primary associative-container types are `map` and `set`. The elements in a `map` are key-value pairs: The key serves as an index into the `map`, and the value represents the data that are stored and retrieved. A `set` contains only a key and supports efficient queries to whether a given key is present.

关联容器（**Associative containers**）支持通过键来高效地查找和读取元素。两个基本的关联容器类型是 `map` 和 `set`。`map` 的元素以键—值（key-value）对的形式组织：键用作元素在 `map` 中的索引，而值则表示所存储和读取的数据。`set` 仅包含一个键，并有效地支持关于某个键是否存在查询。

In general, a `set` is most useful when we want to store a collection of distinct values efficiently, and a `map` is most useful when we wish to store (and possibly modify) a value associated with each key. We might use a `set` to hold words that we want to ignore when doing some kind of text processing. A dictionary would be a good use for a `map`: The word would be the key, and its definition would be the value.

一般来说，如果希望有效地存储不同值的集合，那么使用 `set` 容器比较合适，而 `map` 容器则更适用于需要存储（乃至修改）每个键所关联的值的情况。在做某种文本处理时，可使用 `set` 保存要忽略的单词。而字典则是 `map` 的一种很好的应用：单词本身是键，而它的解释说明则是值。

An object of the `map` or `set` type may contain only a single element with a given key. There is no way to add a second element with the same key. If we need to have multiple instances with a single key, then we can use `multimap` or `multi set`, which do allow multiple elements with a given key.

`set` 和 `map` 类型的对象所包含的元素都具有不同的键，不允许为同一个键添加第二个元素。如果一个键必须对应多个实例，则需使用 `multimap` 或 `multi set`，这两种类型允许多个元素拥有相同的键。

The associative containers support many of the same operations as do the sequential containers. They also provide specialized operations that manage or use the key. In the sections that follow, we look at the associative container types and their operations in detail. We'll conclude the chapter by using the containers to implement a small text-query program.

关联容器支持很多顺序容器也提供的相同操作，此外，还提供管理或使用键的特殊操作。下面的小节将详细讨论关联容器类型及其操作，最后以一个用容器实现的小型文本查询程序结束本章。

Table 10.1. Associative Container Types

表 10.1. 关联容器类型

<code>map</code>	Associative array; elements stored and retrieved by key 关联数组：元素通过键来存储和读取
<code>set</code>	Variable-sized collection with fast retrieval by key 大小可变的集合，支持通过键实现的快速读取
<code>multimap</code>	<code>map</code> in which a key can appear multiple times 支持同一个键多次出现的 <code>map</code> 类型
<code>multiset</code>	<code>set</code> in which a key can appear multiple times 支持同一个键多次出现的 <code>set</code> 类型

Team LiB**◀ PREVIOUS** **NEXT ▶**

10.1. Preliminaries: the `pair` Type

10.1. 引言: `pair` 类型

Before we look at the associative containers, we need to know about a simple companion library type named `pair`, which is defined in the `utility` header.

在开始介绍关联容器之前, 必须先了解一种与之相关的简单的标准库类型——`pair` (表 10.2), 该类型在 `utility` 头文件中定义。

Table 10.2. Operations on `pairs`

表 10.2 `pairs` 类型提供的操作

<code>pair<T1, T2> p1;</code>	Create an empty <code>pair</code> with two elements of types <code>T1</code> and <code>T2</code> . The elements are value-initialized (Section 3.3.1, p. 92).
	创建一个空的 <code>pair</code> 对象, 它的两个元素分别是 <code>T1</code> 和 <code>T2</code> 类型, 采用值初始化 (第 3.3.1 节)
<code>pair<T1, T2> p1(v1, v2);</code>	Create a <code>pair</code> with types <code>T1</code> and <code>T2</code> initializing the <code>first</code> member from <code>v1</code> and the <code>second</code> from <code>v2</code> .
	创建一个 <code>pair</code> 对象, 它的两个元素分别是 <code>T1</code> 和 <code>T2</code> , 其中 <code>first</code> 成员初始化为 <code>v1</code> , 而 <code>second</code> 成员初始化为 <code>v2</code>
<code>make_pair(v1, v2)</code>	Creates a new <code>pair</code> from the values <code>v1</code> and <code>v2</code> . The type of the <code>pair</code> is inferred from the types of <code>v1</code> and <code>v2</code> .
	以 <code>v1</code> 和 <code>v2</code> 值创建一个新 <code>pair</code> 对象, 其元素类型分别是 <code>v1</code> 和 <code>v2</code> 的类型
<code>p1 < p2</code>	Less than between two pair objects. Less than is defined as dictionary ordering: Returns <code>true</code> if <code>p1.first < p2.first</code> or if <code>!(p2.first < p1.first) && p1.second < p2.second</code> .
	两个 <code>pair</code> 对象之间的小于运算, 其定义遵循字典次序: 如果 <code>p1.first < p2.first</code> 或者 <code>!(p2.first < p1.first) && p1.second < p2.second</code> , 则返回 <code>true</code>
<code>p1 == p2</code>	Two <code>pairs</code> are equal if their <code>first</code> and <code>second</code> members are respectively equal. Uses the underlying element <code>==</code> operator.
	如果两个 <code>pair</code> 对象的 <code>first</code> 和 <code>second</code> 成员依次相等, 则这两个对象相等。该运算使用其元素的 <code>==</code> 操作符
<code>p.first</code>	Returns the (public) data member of <code>p</code> named <code>first</code> .
	返回 <code>p</code> 中名为 <code>first</code> 的 (公有) 数据成员
<code>p.second</code>	Returns the (public) data member of <code>p</code> named <code>second</code> .
	返回 <code>p</code> 的名为 <code>second</code> 的 (公有) 数据成员

Creating and Initializing pairs

`pair` 的创建和初始化

A `pair` holds two data values. Like the containers, `pair` is a template type. Unlike the containers we've seen so far, we must supply two type names when we create a `pair`: A `pair` holds two data members, each of which has the corresponding named type. There is no requirement that the

Section 10.1. Preliminaries: the pair Type

two types be the same.

`pair` 包含两个数据值。与容器一样，`pair` 也是一种模板类型。但又与之前介绍的容器不同，在创建 `pair` 对象时，必须提供两个类型名：`pair` 对象所包含的两个数据成员各自对应的类型名字，这两个类型必相同。

```
pair<string, string> anon;           // holds two strings
pair<string, int> word_count;        // holds a string and an int
pair<string, vector<int>> line; // holds string and vector<int>
```

When we create `pair` objects with no initializer, the default constructor value-initializes the members. Thus, `anon` is a `pair` of two empty `strings`, and `line` holds an empty `string` and an empty `vector`. The `int` value in `word_count` gets the value 0 and the `string` member is initialized to the empty `string`.

如果在创建 `pair` 对象时不提供初始化式，则调用默认构造函数对其成员采用值初始化。于是，`anon` 是包含两空 `string` 类型成员的 `pair` 对象，`line` 则存储一个空的 `string` 类型对象和一个空的 `vector` 类型对象。`word_count` 中的 `int` 成员获得 0 值，而 `string` 成员则初始化为空 `string` 对象。

We can also provide initializers for each member:

当然，也可在定义时为每个成员提供初始化式：

```
pair<string, string> author("James", "Joyce");
```

creates a `pair` named `author`, in which each member has type `string`. The object named `author` is initialized to hold two `strings` with the values "James" and "Joyce".

创建一个名为 `author` 的 `pair` 对象，它的两个成员都是 `string` 类型，分别初始化为字符串 "James" 和 "Joyce"。

The `pair` type can be unwieldy to type, so when we wish to define a number of objects of the same `pair` type, it is convenient to use a `typedef` ([Section 2.6](#), p. 61):

`pair` 类型的使用相当繁琐，因此，如果需要定义多个相同的 `pair` 类型对象，可考虑利用 `typedef` 简化其声明：

```
typedef pair<string, string> Author;
Author proust("Marcel", "Proust");
Author joyce("James", "Joyce");
```

Operations on pairs

pairs 对象的操作

Unlike other library types, the `pair` class gives us direct access to its data members: Its members are `public`. These members are named `first` and `second`, respectively. We can access them using the normal dot operator ([Section 1.5.2](#), p. 25) member access notation:

与其他标准库类型不同，对于 `pair` 类，可以直接访问其数据成员：其成员都是仅有的，分别命名为 `first` 和 `second`。只需使用普通的点操作符——成员访问标志即可访问其成员：

```
string firstBook;
// access and test the data members of the pair
if (author.first == "James" && author.second == "Joyce")
    firstBook = "Stephen Hero";
```

The library defines only a limited number of operations on `pairs`, which are listed in [Table 10.2](#) on the preceding page.

标准库只为 `pair` 类型定义了 [表 10.2](#) 所列出的数量有限的操作。

Generating a New pair

生成新的 `pair` 对象

In addition to the constructors, the library defines the `make_pair` function, which generates a new `pair` from its two arguments. We might use this function to make a new `pair` to assign to an existing `pair`:

除了构造函数，标准库还定义了一个 `make_pair` 函数，由传递给它的两个实参生成一个新的 `pair` 对象。可如下使用该函数创建新的 `pair` 对象，并赋给已存在的 `pair` 对象：

Section 10.1. Preliminaries: the pair Type

```
pair<string, string> next_auth;
string first, last;
while (cin >> first >> last) {
    // generate a pair from first and last
    next_auth = make_pair(first, last);
    // process next_auth...
}
```

This loop processes a sequence of authors. The call to `make_pair` generates a new `pair` from the names read in the `while` condition. It is equivalent to the somewhat more complicated

这个循环处理一系列的作者信息：在 `while` 循环条件中读入的作者名字作为实参，调用 `make_pair` 函数生成一个新的 `pair` 对象。此操作等价于下面更复杂的操作：

```
// use pair constructor to make first and last into a pair
next_auth = pair<string, string>(first, last);
```

Because the data members of `pair` are `public`, we could read the input even more directly as

由于 `pair` 的数据成员是公有的，因而可如下直接地读取输入：

```
pair<string, string> next_auth;
// read directly into the members of next_auth
while (cin >> next_auth.first >> next_auth.second) {
    // process next_auth...
}
```

Exercises Section 10.1

Exercise 10.1: Write a program to read a sequence of `strings` and `ints`, storing each into a `pair`. Store the `pairs` in a `vector`.

编写程序读入一系列 `string` 和 `int` 型数据，将每一组存储在一个 `pair` 对象中，然后将这些 `pair` 对象存储在 `vector` 容器里。

Exercise 10.2: There are at least three ways to create the `pairs` in the program for the previous exercise. Write three versions of the program creating the `pairs` in each way. Indicate which form you think is easier to write and understand and why.

在前一题中，至少可使用三种方法创建 `pair` 对象。编写三个版本的程序，分别采用不同的方法来创建 `pair` 对象。你认为哪一种方法更易于编写和理解，为什么？

10.2. Associative Containers

10.2. 关联容器

Associative containers share many, but not all, of the operations on sequential containers. Associative containers do not have the `front`, `push_front`, `pop_front`, `back`, `push_back`, or `pop_back` operations.

关联容器共享大部分——但并非全部——的顺序容器操作。关联容器不提供 `front`、`push_front`、`pop_front`、`back`、`push_back` 以及 `pop_back` 操作。

The operations common to sequential and associative containers are:

顺序容器和关联容器公共的操作包括下面的几种：

- The first three constructors described in [Table 9.2](#) (p. 307):

[表 9.2](#) 描述的前三种构造函数：

```
C<T> c;           // creates an empty container
// c2 must be same type as c1
C<T> c1(c2);    // copies elements from c2 into c1
// b and e are iterators denoting a sequence
C<T> c(b, e);   // copies elements from the sequence into c
```

Associative containers cannot be defined from a size, because there would be no way to know what values to give the keys.

关联容器不能通过容器大小来定义，因为这样的话就无法知道键所对应的值是什么。

- The relational operations described in [Section 9.3.4](#) (p. 321).

[第 9.3.4 节](#) 中描述的关系运算。

- The `begin`, `end`, `rbegin`, and `rend` operations of [Table 9.6](#) (p. 317).

[表 9.6](#) 列出的 `begin`、`end`、`rbegin` 和 `rend` 操作。

- The typedefs listed in [Table 9.5](#) (p. 316). Note that for `map`, the `value_type` is not the same as the element type. Instead, `value_type` is a `pair` representing the types of the keys and associated values. [Section 10.3.2](#) (p. 361) explains the typedefs for `maps` in more detail.

[表 9.5](#) 列出的类型别名 (typedef)。注意，对于 `map` 容器，`value_type` 并非元素的类型，而是描述键及其关联值类型的 `pair` 类型。[第 10.3.2 节](#) 将详细解释 `map` 中的类型别名。

- The `swap` and assignment operator described in [Table 9.11](#) (p. 329). Associative containers do not provide the `assign` functions.

[表 9.11](#) 中描述的 `swap` 和赋值操作。但关联容器不提供 `assign` 函数。

- The `clear` and `erase` operations from [Table 9.10](#) (p. 326), except that the `erase` operation on an associative container returns `void`.

[表 9.10](#) 列出的 `clear` 和 `erase` 操作，但关联容器的 `erase` 运算返回 `void` 类型。

- The size operations in [Table 9.8](#) (p. 324) except for `resize`, which we cannot use on an associative container.

[表 9.8](#) 列出的关于容器大小的操作。但 `resize` 函数不能用于关联容器。

Elements Are Ordered by Key

根据键排列元素

The associative container types define additional operations beyond the ones just listed. They also redefine the meaning or return type of operations that are in common with the sequential containers. The differences in these common operations reflect the use of keys in the associative containers.

除了上述列出的操作之外，关联容器还提供了其他的操作。而对于顺序容器也提供的相同操作，关联容器也重新定义了这些操作的含义或返回类型，其中的差别在于关联容器中使用了键。



There is one important consequence of the fact that elements are ordered by key: When we iterate across an associative container, we are guaranteed that the elements are accessed in key order, irrespective of the order in which the elements were placed in the container.

“容器元素根据键的次序排列”这一事实就是一个重要的结论：在迭代遍历关联容器时，我们可确保按键的顺序的访问元素，而与元素在容器中的存放位置完全无关。

Exercises Section 10.2

Exercise Describe the differences between an associative container and a sequential container.

10.3:

描述关联容器和顺序容器的差别。

Exercise Give illustrations on when a `list`, `vector`, `deque`, `map`, and `set` might be most useful.

10.4:

举例说明 `list`、`vector`、`deque`、`map` 以及 `set` 类型分别适用的情况。

10.3. The `map` Type

10.3. `map` 类型

A `map` is a collection of keyvalue pairs. The `map` type is often referred to as an [associative array](#): It is like the built-in array type, in that the key can be used as an index to fetch a value. It is associative in that values are associated with a particular key rather than being fetched by position in the array.

`map` 是键-值对的集合。`map` 类型通常可理解为[关联数组 \(associative array\)](#)：可使用键作为下标来获取一个值，正如内置数组类型一样。而关联的本质在于元素的值与某个特定的键相关联，而并非通过元素在数组中的位置来获取。

10.3.1. Defining a `map`

10.3.1. `map` 对象的定义

To use a `map`, we must include the `map` header. When we define a `map` object, we must indicate both the key and [value type](#):

要使用 `map` 对象，则必须包含 `map` 头文件。在定义 `map` 对象时，必须分别指明键和值的类型 ([value type](#)) (表 10.4)：

```
// count number of times each word occurs in the input
map<string, int> word_count; // empty map from string to int
```

defines a `map` object `word_count` that is indexed by a `string` and that holds an associated `int` value.

这个语句定义了一个名为 `word_count` 的 `map` 对象，由 `string` 类型的键索引，关联的值则 `int` 型。

Table 10.3. Constructors for `map`

表 10.3. `map` 的构造函数

<code>map<k, v> m;</code>	Create an empty <code>map</code> named <code>m</code> with key and value types <code>k</code> and <code>v</code> . 创建一个名为 <code>m</code> 的空 <code>map</code> 对象，其键和值的类型分别为 <code>k</code> 和 <code>v</code>
<code>map<k, v> m(m2);</code>	Create <code>m</code> as a copy of <code>m2</code> ; <code>m</code> and <code>m2</code> must have the same key and value types. 创建 <code>m2</code> 的副本 <code>m</code> , <code>m</code> 与 <code>m2</code> 必须有相同的键类型和值类型
<code>map<k, v> m(b, e);</code>	Create <code>m</code> as a copy of the elements from the range denoted by iterators <code>b</code> and <code>e</code> . Elements must have a type that can be converted to <code>pair<const k, v></code> . 创建 <code>map</code> 类型的对象 <code>m</code> , 存储迭代器 <code>b</code> 和 <code>e</code> 标记的范围内所有元素的副本。元素的类型必须能转换为 <code>pair<const k, v></code>

Constraints on the Key Type

键类型的约束

Whenever we use an associative container, its keys have not only a type, but also an associated comparison function. By default, the library uses the `<` operator for the [key type](#) to compare the keys. [Section 15.8.3](#) (p. 605) will show how we can override the default and provide our own function.

Section 10.3. The map Type

在使用关联容器时，它的键不但有一个类型，而且还有一个相关的比较函数。默认情况下，标准库使用键类型定义的 `<` 操作符来实现键 (key type) 的比较。[第 15.8.3 节](#) 将介绍如何重写默认的操作符，并提供自定义的操作符函数。

Whichever comparison function we use must define a **strict weak ordering** over the key type. We can think of a strict weak ordering as "less than," although we might choose to define a more complicated comparison function. However we define it, such a comparison function must always yield false when we compare a key with itself. Moreover, if we compare two keys, they cannot both be "less than" each other, and if `k1` is "less than" `k2`, which in turn is "less than" `k3`, then `k1` must be "less than" `k3`. If we have two keys, neither of which is "less than" the other, the container will treat them as equal. When used as a key to a `map`, either value could be used to access the corresponding element.

所用的比较函数必须在键类型上定义**严格弱排序 (strict weak ordering)**。所谓的严格弱排序可理解为键类型数据上的“小于”关系，虽然实际上可以选择将比较函数设计得更复杂。但无论这样的比较函数如何定义，当用于一个键与自身的比较时，肯定会导致 `false` 结果。此外，在比较两个键时，不能出现相互“小于”的情况，而且，如果 `k1`“小于”`k2`, `k2`“小于”`k3`，则 `k1` 必然“小于”`k3`。对于两个键，如果它们相互之间都不存在“小于”关系，则容器将之视为相同的键。用做 `map` 对象的键时，可使用任意一个键值来访问相应的元素。



In practice, what's important is that the key type must define the `<` operator and that that operator should "do the right thing."

在实际应用中，键类型必须定义 `<` 操作符，而且该操作符应能“正确地工作”，这一点很重要。

例如，在书店问题中，可增加一个名为 `ISBN` 的类型，封装与国际标准图书编号 (`ISBN`) 相关的规则。在我们的实现中，国际标准图书编号是 `string` 类型，可做比较运算以确定编号之间的大小关系。因此，`ISBN` 类型可以支持 `<` 运算。假设我们已经定义了这样的类型，则可定义一个 `map` 容器对象，以便高效地查找书店中存放的某本书。

```
map<ISBN, Sales_item> bookstore;
```

defines a `map` object named `bookstore` that is indexed by an `ISBN`. Each element in the `map` holds an associated instance of our `Sales_item` class.

该语句定义了一个名为 `bookstore` 的 `map` 对象，以 `ISBN` 类型的对象为索引，其所有元素都存储了一个关联的 `Sales_item` 类类型实例。



The key type needs to support only the `<` operator. There is no requirement that it support the other relational or equality operators.

对于键类型，唯一的约束就是必须支持 `<` 操作符，至于是否支持其他的关系或相等运算，则不作要求。

Exercises Section 10.3.1

Exercise 10.5: Define a `map` that associates words with a `list` of line numbers on which the word might occur.

定义一个 `map` 对象，将单词与一个 `list` 对象关联起来，该 `list` 对象存储对应的单词可能的行号。

Exercise 10.6: Could we define a `map` from `vector<int>::iterator` to `int`? What about from `list<int>::iterator` to `int`? What about from `pair<int, string>` to `int`? In each case, if not, explain why not.

可否定义一个 `map` 对象以 `vector<int>::iterator` 为键关联 `int` 型对象？如果以 `list<int>::iterator` 关联 `int`？对于每种情况，如果不允许，请解释其原因。

10.3.2. Types Defined by `map`

10.3.2. `map` 定义的类型

The elements of a `map` are keyvalue pairs. That is, each element has two parts: its key and the value associated with that key. The `value_type` for a `map` reflects this fact. This type is more complicated than those we've seen for other containers: `value_type` is a `pair` that holds the key and value of a given element. Moreover, the key is `const`. For example, the `value_type` of the `word_count` array is `pair<const string, int>`.

Section 10.3. The map Type

`map` 对象的元素是键—值对，也即每个元素包含两个部分：键以及由键关联的值。`map` 的 `value_type` 就反映了这个事实。该类型比前面介绍的容器所使用的元素类型要复杂得多：`value_type` 是存储元素的键以及值的 `pair` 类型，而且键为 `const`。例如，`word_count` 数组的 `value_type` 为 `pair<const string, int>` 类型。

Table 10.4. Types Defined by the `map` Class

表 10.4. `map` 类定义的类型

<code>map<K, V>::key_type</code>	The type of the keys used to index the <code>map</code> . 在 <code>map</code> 容器中，用做索引的键的类型
<code>map<K, V>::mapped_type</code>	The type of the values associated with the keys in the <code>map</code> . 在 <code>map</code> 容器中，键所关联的值的类型
<code>map<K, V>::value_type</code>	A pair whose first element has type <code>const map<K, V>::key_type</code> and <code>second</code> has type <code>map<K, V>::mapped_type</code> . 一个 <code>pair</code> 类型，它的 <code>first</code> 元素具有 <code>const map<K, V>::key_type</code> 类型，而 <code>second</code> 元素则为 <code>map<K, V>::mapped_type</code> 类型



When learning the `map` interface, it is essential to remember that the `value_type` is a `pair` and that we can change the value but not the key member of that `pair`.

在学习 `map` 的接口时，需谨记 `value_type` 是 `pair` 类型，它的值成员可以修改，但键成员不能修改。

Dereferencing a `map` Iterator Yields a `pair`

`map` 迭代器进行解引用将产生 `pair` 类型的对象

When we dereference an iterator, we get a reference to a value of the container's `value_type`. In the case of `map`, the `value_type` is a `pair`:

对迭代器进行解引用时，将获得一个引用，指向容器中一个 `value_type` 类型的值。对于 `map` 容器，其 `value_type` 是 `pair` 类型：

```
// get an iterator to an element in word_count
map<string, int>::iterator map_it = word_count.begin();

// *map_it is a reference to a pair<const string, int> object
cout << map_it->first;           // prints the key for this element
cout << " " << map_it->second;    // prints the value of the element
map_it->first = "new key";        // error: key is const
++map_it->second;               // ok: we can change value through an iterator
```

Dereferencing the iterator yields a `pair` object in which `first` member holds the `const` key and `second` member holds the value.

对迭代器进行解引用将获得一个 `pair` 对象，它的 `first` 成员存放键，为 `const`，而 `second` 成员则存放值。

Additional `map` Typedefs

`map` 容器额外定义的类型别名 (typedef)

The `map` class defines two additional types, `key_type` and `mapped_type`, that let us access the type of either the key or the value. For `word_count`, the `key_type` is `string` and `mapped_type` is `int`. As with the sequential containers (Section 9.3.1, p. 317), we use the scope operator to fetch a type member for example, `map<string, int>::key_type`.

`map` 类额外定义了两种类型：`key_type` 和 `mapped_type`，以获得键或值的类型。对于 `word_count`，其 `key_type` 是 `string` 类型，而 `mapped_type` 则是 `int` 型。如同顺

Section 10.3. The map Type

序容器（第 9.3.1 节）一样，可使用作用域操作符（scope operator）来获取类型成员，如 `map<string, int>::key_type`。

Exercises Section 10.3.2

Exercise 10.7: What are the `mapped_type`, `key_type`, and `value_type` of a `map` from `int` to `vector<int>`?

对于以 `int` 型对象为索引关联 `vector<int>` 型对象的 `map` 容器，它的 `mapped_type`、`key_type` 和 `value_type` 分别是什么？

Exercise 10.8: Write an expression using a `map` iterator to assign a value to an element.

编写一个表达式，使用 `map` 的迭代器给其元素赋值。

10.3.3. Adding Elements to a `map`

10.3.3. 给 `map` 添加元素

Once the `map` is defined, the next step is to populate it with the keyvalue element pairs. We can do so either by using the `insert` member or by fetching an element using the subscript operator and then assigning a value to the element returned. In both cases, the fact that there can be only a single element for a given key affects the behavior of these operations.

定义了 `map` 容器后，下一步工作就是在容器中添加键—值元素对。该项工作可使用 `insert` 成员实现；或者，先用下标操作符获取元素，然后给获取的元素赋值。在这两种情况下，一个给定的键只能对应于一个元素这一事实影响了这些操作的行为。

10.3.4. Subscripting a `map`

10.3.4. 使用下标访问 `map` 对象

When we write

如下编写程序时：

```
map <string, int> word_count; // empty map  
// insert default initialized element with key Anna; then assign 1 to its value  
word_count["Anna"] = 1;
```

the following steps take place:

将发生以下事情：

1. `word_count` is searched for the element whose key is `Anna`. The element is not found.
在 `word_count` 中查找键为 `Anna` 的元素，没有找到。
2. A new keyvalue pair is inserted into `word_count`. The key is a `const string` holding `Anna`. The value is value initialized, meaning in this case that the value is 0.
将一个新的键—值对插入到 `word_count` 中。它的键是 `const string` 类型的对象，保存 `Anna`。而它的值则采用值初始化，这就意味着在本例中值为 0。
3. The new keyvalue pair is inserted into `word_count`.
将这个新的键—值对插入到 `word_count` 中。
4. The newly inserted element is fetched and is given the value 1.
读取新插入的元素，并将它的值赋为 1。



Subscripting a `map` behaves quite differently from subscripting an array or `vector`: Using an index that is not already present adds an element with that index to the `map`.

使用下标访问 `map` 与使用下标访问数组或 `vector` 的行为截然不同：用下标访问不存在的元素将导致在 `map` 容器中添加一个新元素，它的键即为该下标值。

Section 10.3. The map Type

As with other subscript operators, the `map` subscript takes an index (that is, a key) and fetches the value associated with that key. When we look for a key that is already in the `map`, then the behavior is the same for a `map` subscript or a `vector` subscript: The value associated with the key is returned. For `maps` only, if the key is not already present, a new element is created and inserted into the `map` for that key. The associated value is value-initialized: An element of class type is initialized using the default constructor for the element type; a built-in type is initialized to 0.

如同其他下标操作符一样，`map` 的下标也使用索引（其实就是键）来获取该键所关联的值。如果该键已在容器中，则 `map` 的下标运算与 `vector` 的下标运算行为相同：返回该键所关联的值。只有在所查找的键不存在时，`map` 容器才为该键创建一个新的元素，并将它插入到此 `map` 对象中。此时，所关联的值采用值初始化：类类型的元素用默认构造函数初始化，而内置类型的元素初始化为 0。

Using the Value Returned from a Subscript Operation

下标操作符返回值的使用

As usual, the subscript operator returns an lvalue. The lvalue it returns is the value associated with the key. We can read or write the element:

通常来说，下标操作符返回左值。它返回的左值是特定键所关联的值。可如下读或写元素：

```
cout << word_count[ "Anna" ]; // fetch element indexed by Anna; prints 1
++word_count[ "Anna" ];      // fetch the element and add one to it
cout << word_count[ "Anna" ]; // fetch the element and print it; prints 2
```



Unlike `vector` or `string`, the type returned by `map` subscript operator differs from the type obtained by dereferencing a `map` iterator.

有别于 `vector` 或 `string` 类型，`map` 下标操作符返回的类型与对 `map` 迭代器进行解引用获得的类型不相同。

As we've seen, a `map` iterator returns a `value_type`, which is a `pair` that contains a `const key_type` and `mapped_type`; the subscript operator returns a value of type `mapped_type`.

显然，`map` 迭代器返回 `value_type` 类型的值——包含 `const key_type` 和 `mapped_type` 类型成员的 `pair` 对象；下标操作符则返回一个 `mapped_type` 类型的值。

Programming Implications of the Subscript Behavior

下标行为的编程意义

The fact that subscript adds an element if it is not already in the `map` allows us to write surprisingly succinct programs:

对于 `map` 容器，如果下标所表示的键在容器中不存在，则添加新元素，这一特性可使程序惊人地简练：

```
// count number of times each word occurs in the input
map<string, int> word_count; // empty map from string to int
string word;
while (cin >> word)
    ++word_count[word];
```

This program creates a `map` that keeps track of how many times each word occurs. The `while` loop reads the standard input one word at a time. Each time it reads a new word, it uses that word to index `word_count`. If `word` is already in the `map`, then its value is incremented.

这段程序创建一个 `map` 对象，用来记录每个单词出现的次数。`while` 循环每次从标准输入读取一个单词。如果这是一个新的单词，则在 `word_count` 中添加以该单词为索引的新元素。如果读入的单词已在 `map` 对象中，则将它所对应的值加 1。

The interesting part is what happens when a word is encountered for the first time: A new element indexed by `word`, with an initial value of zero, is created and inserted into `word_count`. The value of that element is immediately incremented so that each time we insert a new word into the `map` it starts off with an occurrence count of one.

其中最有趣的是，在单词第一次出现时，会在 `word_count` 中创建并插入一个以该单词为索引的新元素，同时将它的值初始化为 0。然后其值立即加 1，所以每次在 `map` 中添加新元素时，所统计的出现次数正好从 1 开始。

Exercises Section 10.3.4

Exercise 10.9: Write a program to count and print the number of times each word occurs in the input.
编写程序统计并输出所读入的单词出现的次数。

Exercise 10.10: What does the following program do?
解释下面程序的功能：

```
map<int, int> m;
m[0] = 1;
```

Contrast the behavior of the previous program with this one:

比较上一程序和下面程序的行为

```
vector<int> v;
v[0] = 1;
```

Exercise 10.11: What type can be used to subscript a `map`? What type does the sub-script operator return? Give a concrete example that is, define a `map` and then write the types that could be used to subscript the `map` and the type that would be returned from the subscript operator.

哪些类型可用做 `map` 容器对象的下标？下标操作符返回的又是什么类型？给出一个具体例子说明，即定义一个 `map` 对象，指出哪些类型可用作其下标，以及下标操作符返回的类型。

10.3.5. Using `map::insert`

10.3.5. `map::insert` 的使用

The `insert` members operate similarly to the operations on sequential containers (Section 9.3.3, p. 318), with one important caveat: We must account for the effect of the key. The key impacts the argument types: The versions that insert a single element take a value that is a keyvalue pair. Similarly, for the version that takes an iterator pair, the iterators must refer to elements that are keyvalue pairs. The other difference is the return type from the version of `insert` that takes a single value, which we will cover in the remainder of this section.

`map` 容器的 `insert` 成员与顺序容器的类似，但有一点要注意：必须考虑键的作用。键影响了实参的类型：插入单个元素的 `insert` 版本使用键-值 `pair` 类型的参数。类似地，对于参数为一对迭代器的版本，迭代器必须指向键-值 `pair` 类型的元素。另一个差别则是：`map` 容器的接受单个值的 `insert` 版本的返回类型。本节的后续部分将详细阐述这一特性。

Table 10.5. `insert` Operations on maps

表 10.5. `map` 容器提供的 `insert` 操作

<code>m.insert(e)</code>	<code>e</code> 是一个用在 <code>m</code> 上的 <code>value_type</code> 类型的值。如果键 (<code>e.first</code>) 不在 <code>m</code> 中，则插入一个值为 <code>e.second</code> 的新元素；如果该键在 <code>m</code> 中已存在，则保持 <code>m</code> 不变。该函数返回一个 <code>pair</code> 类型对象，包含指向键为 <code>e.first</code> 的元素的 <code>map</code> 迭代器，以及一个 <code>bool</code> 类型的对象，表示是否插入了该元素
<code>m.insert(beg, end)</code>	<code>beg</code> 和 <code>end</code> 是迭代器，表示一个范围，其中包含与 <code>m</code> 的 <code>value_type</code> 相同类型的键值对。对于范围内的每个元素，如果给定的键不在 <code>m</code> 中，它将插入键及其关联的值。返回 <code>void</code> 。

Section 10.3. The map Type

	<p><code>beg</code> 和 <code>end</code> 是标记元素范围的迭代器，其中的元素必须为 <code>m.value_type</code> 类型的键—值对。对于该范围内的所有元素，如果它的键在 <code>m</code> 中不存在，则将该键及其关联的值插入到 <code>m</code>。返回 <code>void</code> 类型</p>
<code>m.insert(iter, e)</code>	<p><code>e</code> is a value of the <code>value_type</code> for <code>m</code>. If the key(<code>e.first</code>) is not in <code>m</code>, inserts the new element using the iterator <code>iter</code> as a hint for where to begin the search for where the new element should be stored. Returns an iterator that refers to the element in <code>m</code> with given key.</p> <p><code>e</code> 是一个用在 <code>m</code> 上的 <code>value_type</code> 类型的值。如果键 (<code>e.first</code>) 不在 <code>m</code> 中，则创建新元素，并以迭代器 <code>iter</code> 为起点搜索新元素存储的位置。返回一个迭代器，指向 <code>m</code> 中具有给定键的元素</p>

Using `insert` Instead of Subscripting

以 `insert` 代替下标运算

When we use a subscript to add an element to a `map`, the value part of the element is value-initialized. Often we immediately assign to that value, which means that we've initialized and assigned the same object. Alternatively, we could insert the element directly by using the syntactically more intimidating `insert` member:

使用下标给 `map` 容器添加新元素时，元素的值部分将采用值初始化。通常，我们会立即为其赋值，其实就是对同一个对象进行初始化并赋值。而插入元素的另一个方法是：直接使用 `insert` 成员，其语法更紧凑：

```
// if Anna not already in word_count, inserts new element with value 1
word_count.insert(map<string, int>::value_type("Anna", 1));
```

The argument to this version of `insert`

这个 `insert` 函数版本的实参：

```
map<string, int>::value_type(anna, 1)
```

is a newly created `pair` that is directly inserted into the `map`. Remember that `value_type` is a synonym for the type `pair<const K, V>`, where `K` is the key type and `V` is the type of the associated value. The argument to `insert` constructs a new object of the appropriate `pair` type to insert into the `map`. By using `insert`, we can avoid the extraneous initialization of the value that happens when we insert a new `map` element as a side-effect of using a subscript.

是一个新创建的 `pair` 对象，将直接插入到 `map` 容器中。谨记 `value_type` 是 `pair<const K, V>` 类型的同义词，`K` 为键类型，而 `V` 是键所关联的值的类型。`insert` 的实参创建了一个适当的 `pair` 类型新对象，该对象将插入到 `map` 容器。在添加新 `map` 元素时，使用 `insert` 成员可避免使用下标操作符所带来的副作用：不必要的初始化。

The argument to `insert` is fairly unwieldy. There are two ways to simplify it. We might use `make_pair`:

传递给 `insert` 的实参相当笨拙。可用两种方法简化：使用 `make_pair`：

```
word_count.insert(make_pair("Anna", 1));
```

Or use a `typedef`:

或使用 `typedef`

```
typedef map<string,int>::value_type valType;
word_count.insert(valType("Anna", 1));
```

Either approach improves readability by making the call less complicated.

这两种方法都使用调用变得简单，提高了程序的可读性。

Testing the Return from `insert`

检测 `insert` 的返回值

Section 10.3. The map Type

There can be only one element with a given key in a `map`. If we attempt to `insert` an element with a key that is already in the `map`, then `insert` does nothing. The versions of `insert` that take an iterator or iterator pair do not indicate whether or how many elements were inserted.

`map` 对象中一个给定键只对应一个元素。如果试图插入的元素所对应的键已在容器中，则 `insert` 将不做任何操作。含有一个或一对迭代器形参的 `insert` 函数版本并不说明是否有或有多少个元素插入到容器中。

However, the version of `insert` that takes a single keyvalue `pair` does return a value. That value is a `pair` that contains an iterator that refers to the element in the `map` with the corresponding key, and a `bool` that indicates whether the element was inserted. If the key is already in the `map`, then the value is unchanged, and the `bool` portion of the return is `false`. If the key isn't present, then the element is inserted and the `bool` is `true`. In either case, the iterator refers to the element with the given key. We could rewrite our word count program to use `insert`:

但是，带有一个键—值 `pair` 形参的 `insert` 版本将返回一个值：包含一个迭代器和一个 `bool` 值的 `pair` 对象，其中迭代器指向 `map` 中具有相应键的元素，而 `bool` 值则表示是否插入了该元素。如果该键已在容器中，则其关联的值保持不变，返回的 `bool` 值为 `true`。在这两种情况下，迭代器都将指向具有给定键的元素。下面是使用 `insert` 重写的单词统计程序：

```
// count number of times each word occurs in the input
map<string, int> word_count; // empty map from string to int
string word;
while (cin >> word) {
    // inserts element with key equal to word and value 1;
    // if word already in word_count, insert does nothing
    pair<map<string, int>::iterator, bool> ret =
        word_count.insert(make_pair(word, 1));
    if (!ret.second)           // word already in word_count
        ++ret.first->second; // increment counter
}
```

For each `word`, we attempt to `insert` it with a value 1. The `if` test examines the `bool` in the return from the `insert`. If it is `false`, then the insertion didn't happen and an element indexed by `word` was already in `word_count`. In this case we increment the value associated with that element.

对于每个单词，都尝试 `insert` 它，并将它的值赋 1。`if` 语句检测 `insert` 函数返回值中的 `bool` 值。如果该值为 `false`，则表示没有做插入操作，按 `word` 索引的元素已在 `word_count` 中存在。此时，将该元素所关联的值加 1。

Unwinding the Syntax

语法展开

The definition of `ret` and the increment may be hard to decipher:

`ret` 的定义和自增运算可能比较难解释：

```
pair<map<string, int>::iterator, bool> ret =
    word_count.insert(make_pair(word, 1));
```

It should be easy to see that we're defining a `pair` and that the second type of the `pair` is `bool`. The first type of that `pair` is a bit harder to understand. It is the `iterator` type defined by the `map<string, int>` type.

首先，应该很容易看出我们定义的是一个 `pair` 对象，它的 `second` 成员为 `bool` 类型。而它的 `first` 成员则比较难理解，这是 `map<string, int>` 容器所定义的迭代器类型。

We can understand the increment by first parenthesizing it to reflect the precedence ([Section 5.10.1](#), p. 168) of the operators:

根据操作符的优先级次序 ([第 5.10.1 节](#))，可如下从添加圆括号开始理解自增操作：

```
++(ret.first)->second; // equivalent expression
```

Explaining this expression step by step, we have

下面对这个表达式一步步地展开解释：

- `ret` holds return value from `insert`, which is a `pair`. The `first` member of that `pair` is a `map` iterator referring to the key that was inserted.
`ret` 存储 `insert` 函数返回的 `pair` 对象。该 `pair` 的 `first` 成员是一个 `map` 迭代器，指向插入的键。
- `ret.first` fetches the `map` iterator from the `pair` returned by `insert`.
`ret.first` 从 `insert` 返回的 `pair` 对象中获取 `map` 迭代器。

Section 10.3. The map Type

- `ret.first->second` dereferences that iterator obtaining a `value_type` object. That object is also a `pair`, in which the `second` member is the value part of the element we added.
`ret.first->second` 对该迭代器进行解引用，获得一个 `value_type` 类型的对象。这个对象同样是 `pair` 类型的，它的 `second` 成员即为我们所添加的元素的值部分。
• `++ret.first->second` increments that value.
`++ret.first->second` 实现该值的自增运算。

Putting it back together, the increment statement fetches the iterator for the element indexed by `word` and increments the value part of that element.

归结起来，这个自增语句获取指向按 `word` 索引的元素的迭代器，并将该元素的值加 1。

Exercises Section 10.3.5

Exercise 10.12: Rewrite the word-count program that you wrote in the exercises for [Section 10.3.4](#) (p. 364) to use `insert` instead of subscripting. Explain which program you think is easier to write and read. Explain your reasoning.

重写第 10.3.4 节习题的单词统计程序，要求使用 `insert` 函数代替下标运算。你认为哪个程序更容易编写和阅读？请解释原因。

Exercise 10.13: Given a `map<string, vector<int> >`, write the types used as an argument and as the return value for the version of `insert` that inserts one element.

假设有 `map<string, vector<int> >` 类型，指出在该容器中插入一个元素的 `insert` 函数应具有的参数类型和返回值类型。

10.3.6. Finding and Retrieving a `map` Element

10.3.6. 查找并读取 `map` 中的元素

The subscript operator provides the simplest method of retrieving a value:

下标操作符给出了读取一个值的最简单方法：

```
map<string,int> word_count;
int occurs = word_count["foobar"];
```

As we've seen, using a subscript has an important side effect: If that key is not already in the `map`, then subscript inserts an element with that key.

但是，使用下标存在一个很危险的副作用：如果该键不在 `map` 容器中，那么下标操作会插入一个具有该键的新元素。

Whether this behavior is correct depends on our expectations. In this example, if "foobar" weren't already present, it would be added to the `map` with an associated value of 0. In this case, `occurs` gets a value of 0.

这样的行为是否正确取决于程序员的意愿。在这个例子中，如果“foobar”不存在，则在 `map` 中插入具有该键的新元素，其关联的值为 0。在这种情况下，`occurs` 获得 0 值。

Our word-counting programs relied on the fact that subscripting a nonexistent element inserts that element and initializes the value to 0. There are times, though, when we want to know if an element is present but do not want the element inserted if it is not present. In such cases, we cannot use the subscript operator to determine whether the element is present.

我们的单词统计程序的确是要通过下标引用一个不存在的元素来实现新元素的插入，并将其关联的值初始化为 0。然而，大多数情况下，我们只想知道某元素是否存在，而当该元素不存在时，并不想做插入运算。对于这种应用，则不能使用下标操作符来判断元素是否存在。

There are two operations, `count` and `find`, that we can use to determine if a key is present without causing it to be inserted.

`map` 容器提供了两个操作：`count` 和 `find`，用于检查某个键是否存在而不会插入该键。

Table 10.6. Interrogating a `map` Without Changing It

表 10.6. 不修改 `map` 对象的查询操作

<code>m.count(k)</code>	Returns the number of occurrences of <code>k</code> within <code>m</code> . 返回 <code>m</code> 中 <code>k</code> 的出现次数
<code>m.find(k)</code>	Returns an iterator to the element indexed by <code>k</code> , if there is one, or returns an off-the-end iterator (Section 3.4 , p. 97) if the key is not present. 如果 <code>m</code> 容器中存在按 <code>k</code> 索引的元素，则返回指向该元素的迭代器。如果不存在，则返回超出末端迭代器 (第 3.4 节)

Using `count` to Determine Whether a Key is in the `map`

使用 `count` 检查 `map` 对象中某键是否存在

The `count` member for a `map` always returns either 0 or 1. A `map` may have only one instance of any given key, so `count` effectively indicates whether the key is present. The return from `count` is more useful for `multimaps`, which we cover in [Section 10.5](#) (p. 375). If the return value is nonzero, we can use the subscript operator to fetch the value associated with the key without worrying that doing so will insert the element into the `map`:

对于 `map` 对象，`count` 成员的返回值只能是 0 或 1。`map` 容器只允许一个键对应一个实例，所以 `count` 可有效地表明一个键是否存在。而对于 `multimaps` 容器，`count` 的返回值将有更多的用途，相关内容将会在[第 10.5 节](#)中介绍。如果返回值非 0，则可以使用下标操作符来获取该键所关联的值，而不必担心这样做会在 `map` 中插入新元素：

```
int occurs = 0;
if (word_count.count("foobar"))
    occurs = word_count["foobar"];
```

Of course, executing `count` followed by the subscript effectively looks for the element twice. If we want to use the element if it is present, we should use `find`.

当然，在执行 `count` 后再使用下标操作符，实际上是对元素作了两次查找。如果希望当元素存在时就使用它，则应该用 `find` 操作。

Retrieving an Element Without Adding It

读取元素而不插入该元素

The `find` operation returns an iterator to the element or the `end` iterator if the element is not present:

`find` 操作返回指向元素的迭代器，如果元素不存在，则返回 `end` 迭代器：

```
int occurs = 0;
map<string,int>::iterator it = word_count.find("foobar");
if (it != word_count.end())
    occurs = it->second;
```

We should use `find` when we want to obtain a reference to the element with the specified key if it exists, and do not want to create the element if it does not exist.

如果希望当具有指定键的元素存在时，就获取该元素的引用，否则就不在容器中创建新元素，那么应该使用 `find`。

Exercises Section 10.3.6

Exercise What is the difference between the `map` operations `count` and `find`?

10.14:

`map` 容器的 `count` 和 `find` 运算有何区别？

Exercise What kinds of problems would you use `count` to solve? When might you use `find` instead?

10.15:

你认为 `count` 适合用于解决哪一类问题？而 `find` 呢？

Exercise 10.16: Define and initialize a variable to hold the result of a call to `find` on a `map` from `string` to `vector<int>`.

定义并初始化一个变量，用来存储调用键为 `string`、值为 `vector<int>` 的 `map` 对象的 `find` 函数的返回结果。

10.3.7. Erasing Elements from a `map`

10.3.7. 从 `map` 对象中删除元素

There are three variants of the `erase` operation to remove elements from a `map`. As with the sequential containers, we can `erase` a single element or a range of elements by passing `erase` an iterator or an iterator pair. These versions of `erase` are similar to the corresponding operations on sequential containers with one exception: The `map` operations return `void`, whereas those on the sequential containers return an iterator to the element following the one that was removed.

从 `map` 容器中删除元素的 `erase` 操作有三种变化形式（表 10.7）。与顺序容器一样，可向 `erase` 传递一个或一对迭代器，来删除单个元素或一段范围内的元素。其删除功能类似于顺序容器，但有一点不同：`map` 容器的 `erase` 操作返回 `void`，而顺序容器的 `erase` 操作则返回一个迭代器，指向被删除元素后面的元素。

The `map` type supplies an additional `erase` operation that takes a value of the `key_type` and removes the element with the given key if the element exists. We could use this version to remove a specific word from `word_count` before printing the results:

除此之外，`map` 类型还提供了一种额外的 `erase` 操作，其参数是 `key_type` 类型的值，如果拥有该键的元素存在，则删除该元素。对于单词统计程序，可使用这个版本的 `erase` 函数来删除 `word_count` 中指定的单词，然后输出被删除的单词：

```
// erase of a key returns number of elements removed
if (word_count.erase(removal_word))
    cout << "ok: " << removal_word << " removed\n";
else cout << "oops: " << removal_word << " not found!\n";
```

The `erase` function returns a count of how many elements were removed. In the case of a `map`, that number is either zero or one. If the return value is zero, then the element we wanted to erase was not in the `map`.

`erase` 函数返回被删除元素的个数。对于 `map` 容器，该值必然是 0 或 1。如果返回 0，则表示欲删除的元素在 `map` 不存在。

Table 10.7. Removing Elements from a `map`

表 10.7. 从 `map` 对象中删除元素

<code>m.erase(k)</code>	Removes the element with key <code>k</code> from <code>m</code> . Returns <code>size_type</code> indicating the number of elements removed. 删除 <code>m</code> 中键为 <code>k</code> 的元素。返回 <code>size_type</code> 类型的值，表示删除的元素个数
<code>m.erase(p)</code>	Removes element referred to by the iterator <code>p</code> from <code>m</code> . <code>p</code> must refer to an actual element in <code>m</code> ; it must not be equal to <code>m.end()</code> . Returns <code>void</code> . 从 <code>m</code> 中删除迭代器 <code>p</code> 所指向的元素。 <code>p</code> 必须指向 <code>m</code> 中确实存在的元素，而且不能等于 <code>m.end()</code> 。返回 <code>void</code>
<code>m.erase(b, e)</code>	Removes the elements in the range denoted by the iterator pair <code>b, e</code> . <code>b</code> and <code>e</code> must be a valid range of elements in <code>m</code> : <code>b</code> and <code>e</code> must refer to elements in <code>m</code> or one past the last element in <code>m</code> . <code>b</code> and <code>e</code> must either be equal in which case the range is empty or the element to which <code>b</code> refers must occur before the element referred to by <code>e</code> . Returns <code>void</code> . 从 <code>m</code> 中删除一段范围内的元素，该范围由迭代器对 <code>b</code> 和 <code>e</code> 标记。 <code>b</code> 和 <code>e</code> 必须标记 <code>m</code> 中的一段有效范围：即 <code>b</code> 和 <code>e</code> 都必须指向 <code>m</code> 中的元素或最后一个元素的下一个位置。而且， <code>b</code> 和 <code>e</code> 要么相等（此时删除的范围为空），要么 <code>b</code> 所指向的元素必须出现在 <code>e</code> 所指向的元素之前。返回 <code>void</code> 类型

10.3.8. Iterating across a `map`

10.3.8. `map` 对象的迭代遍历

Like any other container, `map` provides `begin` and `end` operations that yield iterators that we can use to traverse the `map`. For example, we could print the `map` named `word_count` that we built on page [363](#) as follows:

与其他容器一样, `map` 同样提供 `begin` 和 `end` 运算, 以生成用于遍历整个容器的迭代器。例如, 可如下将 `map` 容器 `word_count` 的内容输出:

```
// get iterator positioned on the first element
map<string, int>::const_iterator
    map_it = word_count.begin();
// for each element in the map
while (map_it != word_count.end()) {
    // print the element key, value pairs
    cout << map_it->first << " occurs "
        << map_it->second << " times" << endl;
    ++map_it; // increment iterator to denote the next element
}
```

The `while` condition and increment for the iterator in this loop look a lot like the programs we wrote that printed the contents of a `vector` or a `string`. We initialize an iterator, `map_it`, to refer to the first element in `word_count`. As long as the iterator is not equal to the `end` value, we print the current element and then increment the iterator. The body of the loop is more complicated than those earlier programs because we must print both the key and value for each element.

`while` 循环的条件判断以及循环体中迭代器的自增都与输出 `vector` 或 `string` 容器内容的程序非常相像。首先, 初始化 `map_it` 迭代器, 使之指向 `word_count` 的第一元素。只要该迭代器不等于 `end` 的值, 就输出当前元素并给迭代器加 1。这段程序的循环体要比前面类似的程序更加复杂, 原因在于对于 `map` 的每个元素都必须分别输出它的键和值。



The output of our word-count program prints the words in alphabetical order. When we use an iterator to traverse a `map`, the iterators yield elements in ascending key order.

这个单词统计程序依据字典顺序输出单词。在使用迭代器遍历 `map` 容器时, 迭代器指向的元素按键的升序排列。

10.3.9. A Word Transformation Map

10.3.9. “单词转换” `map` 对象

We'll close this section with a program to illustrate creating, searching, and iterating across a `map`. Our problem is to write a program that, given one `string`, transforms it into another. The input to our program is two files. The first file contains several word pairs. The first word in the pair is one that might be in the input string. The second is the word to use in the output. Essentially, this file provides a set of word transformations when we find the first word, we should replace it by the second. The second file contains the text to transform. If the contents of the word transformation file is

下面的程序说明如何创建、查找和迭代遍历一个 `map` 对象, 我们将以此结束本节内容。这个程序求解的问题是: 给出一个 `string` 对象, 把它转换为另一个 `string` 对象。本程序的输入是两个文件。第一个文件包括了若干单词对, 每对的第一个单词将出现在输入的字符串中, 而第二个单词则是用于输出。本质上, 这个文件提供的是单词转换的集合——在遇到第一个单词时, 应该将之替换为第二个单词。第二个文件则提供了需要转换的文本。如果单词转换文件的内容是:

'em	them
cuz	because
gratz	grateful
i	I
nah	no
pos	supposed
sez	said
tanx	thanks
wuz	was

and the text we are given to transform is

Section 10.3. The map Type

而要转换的文本是：

```
nah i sez tanx cuz i wuz pos to  
not cuz i wuz gratz
```

then the program should generate the following output:

则程序将产生如下输出结果：

```
no I said thanks because I was supposed to  
not because I was grateful
```

The Word Transformation Program

单词转换程序

Our solution, which appears on the next page, stores the word transformation file in a `map`, using the word to be replaced as the key and the word to use as the replacement as its corresponding value. We then read the input, looking up each word to see if it has a transformation. If so, we do the transformation and then print the transformed word. If not, we print the original word.

下面给出的解决方案是将单词转换文件的内容存储在一个 `map` 容器中，将被替换的单词作为键，而用作替换的单词则作为其相应的值。接着读取输入，查找输入的每个单词是否对应有转换。若有，则实现转换，然后输出其转换后的单词，否则，直接输出原词。

Our `main` program takes two arguments ([Section 7.2.6](#), p. 243): the name of the word transformation file and the name of the file to transform. We start by checking the number of arguments. The first argument, `argv[0]`, is always the name of the command. The file names will be in `argv[1]` and `argv[2]`.

该程序的 `main` 函数需要两个实参 ([第 7.2.6 节](#))：单词转换文件的名字以及需要转换的文件名。程序执行时，首先检查实参的个数。第一个实参 `argv[0]` 是命令名，而执行该程序所需要的两个文件名参数则分别存储在 `argv[1]` 及 `argv[2]` 中。

Once we know that `argv[1]` is valid, we call `open_file` ([Section 8.4.3](#), p. 299) to `open` the word transformation file. Assuming the `open` succeeded, we read the transformation pairs. We call `insert` using the first word as the key and the second as the value. When the `while` concludes, `trans_map` contains the data we need to transform the input. If there's a problem with the arguments, we `throw` ([Section 6.13](#), p. 215) an exception and exit the program.

如果 `argv[1]` 的值合法，则调用 `open_file` ([第 8.4.3 节](#)) 打开单词转换文件。假设 `open` 操作成功，则读入“转换对”中的第一个单词为键，第二个为值，调用 `insert` 函数在容器中插入新元素。`while` 循环结束后，`trans_map` 容器对象包含了转换输入文本所需的数据。而如果该实参有问题，则抛出异常 ([第 6.13 节](#)) 并结束程序的运行。

Next, we call `open_file` to open the file we want to transform. The second `while` uses `getline` to read that file a line at a time. We read by line so that our output will have line breaks at the same position as our input file. To get the words from each line we use a nested `while` loop that uses an `istringstream`. This part of the program is similar to the sketch we wrote on page 300.

接下来，调用 `open_file` 打开要转换的文件。第二个 `while` 循环使用 `getline` 函数逐行读入文件。因为程序每次读入一行，从而可在输出文件的相同位置进行换行。然后在内嵌的 `while` 循环中使用 `istringstream` 将每一行中的单词提取出来。这部分程序与[第 8.5 节](#)的程序框架类似。

The inner `while` checks each word to see if it is in the transformation map. If it is, then we replace the word by its corresponding value from the `map`. Finally, we print the word, transformed or not. We use the `bool firstword` to determine whether to print a space. If it is the first word in the line, we don't print a space.

内层的 `while` 循环检查每个单词，判断它是否在转换的 `map` 中出现。如果在，则从该 `map` 对象中取出对应的值替代此单词。最后，无论是否做了转换，都输出该单词。同时，程序使用 `bool` 值 `firstword` 判断是否需要输出空格。如果当前处理的是这一行的第一个单词，则无须输出空格。

```
/*  
 * A program to transform words.  
 * Takes two arguments: The first is name of the word transformation file  
 *                      The second is name of the input to transform  
 */  
int main(int argc, char **argv)  
{  
    // map to hold the word transformation pairs:  
    // key is the word to look for in the input; value is word to use in the output  
    map<string, string> trans_map;  
    string key, value;  
    if (argc != 3)  
        throw runtime_error("wrong number of arguments");  
    // open transformation file and check that open succeeded  
    ifstream map_file;  
    if (!open_file(map_file, argv[1]))  
        throw runtime_error("no transformation file");  
    // read the transformation map and build the map  
    while (map_file >> key >> value)  
        trans_map.insert(make_pair(key, value));  
    // ok, now we're ready to do the transformations
```

Section 10.3. The map Type

```
// open the input file and check that the open succeeded
ifstream input;
if (!open_file(input, argv[2]))
    throw runtime_error("no input file");
string line; // hold each line from the input
// read the text to transform it a line at a time
while (getline(input, line)) {
    istringstream stream(line); // read the line a word at a time
    string word;
    bool firstword = true; // controls whether a space is printed
    while (stream >> word) {
        // ok: the actual mapwork, this part is the heart of the program
        map<string, string>::const_iterator map_it =
            trans_map.find(word);
        // if this word is in the transformation map
        if (map_it != trans_map.end())
            // replace it by the transformation value in the map
            word = map_it->second;
        if (firstword)
            firstword = false;
        else
            cout << " "; // print space between words
        cout << word;
    }
    cout << endl; // done with this line of input
}
return 0;
}
```

Exercises Section 10.3.9

Exercise

10.17: Our transformation program uses `find` to look for each word:

上述转换程序使用了 `find` 函数来查找单词：

```
map<string, string>::const_iterator map_it =
    trans_map.find(word);
```

Why do you suppose the program uses `find`? What would happen if it used the subscript operator instead?

你认为这个程序为什么要使用 `find` 函数？如果使用下标操作符又会怎么样？

Exercise

10.18: Define a `map` for which the key is the family surname and the value is a `vector` of the children's names. Populate the `map` with at least six entries. Test it by supporting user queries based on a surname, which should list the names of children in that family.

定义一个 `map` 对象，其元素的键是家庭姓氏，而值则是存储该家庭孩子名字的 `vector` 对象。为这个 `map` 容器输入至少六个条目。通过基于家庭姓氏的查询检测你的程序，查询应输出该家庭所有孩子的名字。

Exercise

10.19: Extend the `map` from the previous exercise by having the `vector` store a `pair` that holds a child's name and birthday. Revise the program accordingly. Test your modified test program to verify its correctness.

把上一题的 `map` 对象再扩展一下，使其 `vector` 对象存储 `pair` 类型的对象，记录每个孩子的名字和生日。相应地修改程序，测试修改后测试程序以检查所编写的 `map` 是否正确。

Exercise

10.20: List at least three possible applications in which the `map` type might be of use. Write the definition of each `map` and indicate how the elements are likely to be inserted and retrieved.

列出至少三种可以使用 `map` 类型的应用。为每种应用定义 `map` 对象，并指出如何插入和读取元素。

10.4. The `set` Type

10.4. `set` 类型

A `map` is a collection of key-value pairs, such as an address and phone number keyed to an individual's name. In contrast, a `set` is simply a collection of keys. For example, a business might define a `set` named `bad_checks`, to hold the names of individuals who have issued bad checks to the company. A `set` is most useful when we simply want to know whether a value is present. Before accepting a check, for example, that business would query `bad_checks` to see whether the customer's name was present.

`map` 容器是键-值对的集合，好比以人名为键的地址和电话号码。相反地，`set` 容器只是单纯的键的集合。例如，某公司可能定义了一个名为 `bad_checks` 的 `set` 容器，用于记录曾经给本公司发空头支票的客户。当只想知道一个值是否存在时，使用 `set` 容器是最适合的。例如，在接收一张支票前，该公司可能想查询 `bad_checks` 对象，看看该客户的名字是否存在。

With two exceptions, `set` supports the same operations as `map`:

除了两种例外情况，`set` 容器支持大部分的 `map` 操作，包括下面几种：

- All the common container operations listed in [Section 10.2](#) (p. 358).

[第 10.2 节](#)列出的所有通用的容器操作。

- The constructors described in [Table 10.3](#) (p. 360).

[表 10.3](#) 描述的构造函数。

- The `insert` operations described in [Table 10.5](#) (p. 365).

[表 10.5](#) 描述的 `insert` 操作。

- The `count` and `find` operations described in [Table 10.6](#) (p. 367).

[表 10.6](#) 描述的 `count` 和 `find` 操作。

- The `erase` operations described in [Table 10.7](#) (p. 369).

[表 10.7](#) 描述的 `erase` 操作。

The exceptions are that `set` does not provide a subscript operator and does not define `mapped_type`. In a `set`, the `value_type` is not a `pair`; instead it and `key_type` are the same type. They are each the type of the elements stored in the `set`. These differences reflect the fact that `set` holds only keys; there is no value associated with the key. As with `map`, the keys of a `set` must be unique and may not be changed.

两种例外包括：`set` 不支持下标操作符，而且没有定义 `mapped_type` 类型。在 `set` 容器中，`value_type` 不是 `pair` 类型，而是与 `key_type` 相同的类型。它们指的都是 `set` 中存储的元素类型。这一差别也体现了 `set` 存储的元素仅仅是键，而没有所关联的值。与 `map` 一样，`set` 容器存储的键也必须唯一，而且不能修改。

Exercises Section 10.4

Exercise 10.21: Explain the difference between a `map` and a `set`. When might you use one or the other?

解释 `map` 和 `set` 容器的差别，以及它们各自适用的情况。

Exercise 10.22: Explain the difference between a `set` and a `list`. When might you use one or the other?

解释 `set` 和 `list` 容器的差别，以及它们各自适用的情况。

10.4.1. Defining and Using `sets`

10.4.1. `set` 容器的定义和使用

To use a `set`, we must include the `set` header. The operations on `sets` are essentially identical to those on `maps`.

为了使用 `set` 容器，必须包含 `set` 头文件。`set` 支持的操作基本上与 `map` 提供的相同。

Section 10.4. The set Type

As with `map`, there can be only one element with a given key in a `set`. When we initialize a `set` from a range of elements or `insert` a range of elements, only one element with a given key is actually added:

与 `map` 容器一样, `set` 容器的每个键都只能对应一个元素。以一段范围的元素初始化 `set` 对象, 或在 `set` 对象中插入一组元素时, 对于每个键, 事实上都只添加了一个元素:

```
// define a vector with 20 elements, holding two copies of each number from 0 to 9
vector<int> ivec;
for (vector<int>::size_type i = 0; i != 10; ++i) {
    ivec.push_back(i);
    ivec.push_back(i); // duplicate copies of each number
}
//iset holds unique elements from ivec
set<int> iset(ivec.begin(), ivec.end());
cout << ivec.size() << endl;      // prints 20
cout << iset.size() << endl;      // prints 10
```

We first create a `vector` of `ints` named `ivec` that has 20 elements: two copies of each of the integers from 0 through 9 inclusive. We then use all the elements from `ivec` to initialize a `set` of `ints`. That `set` has only ten elements: one for each distinct element in `ivec`.

首先创建了一个名为 `ivec` 的 `int` 型 `vector` 容器, 存储 20 个元素: 0-9 (包括 9) 中每个整数都出现了两次。然后用 `ivec` 中所有的元素初始化一个 `int` 型的 `set` 容器。则这个 `set` 容器仅有 10 个元素: `ivec` 中不相同的各个元素。

Adding Elements to a `set`

在 `set` 中添加元素

We can add elements to a `set` by using the `insert` operation:

可使用 `insert` 操作在 `set` 中添加元素:

```
set<string> set1;           // empty set
set1.insert("the");          // set1 now has one element
set1.insert("and");          // set1 now has two elements
```

Alternatively, we can insert a range of elements by providing a pair of iterators to `insert`. This version of `insert` works similarly to the constructor that takes an iterator pair only one element with a given key is inserted:

另一种用法是, 调用 `insert` 函数时, 提供一对迭代器实参, 插入其标记范围内所有的元素。该版本的 `insert` 函数类似于形参为一对迭代器的构造函数——对于一个键, 仅插入一个元素:

```
set<int> iset2; // empty set
iset2.insert(ivec.begin(), ivec.end()); // iset2 has 10 elements
```

Like the `map` operations, the version of `insert` that takes a key returns a `pair` containing an iterator to the element with this key and a `bool` indicating whether the element was added. The one that takes an iterator pair returns `void`.

与 `map` 容器的操作一样, 带有一个键参数的 `insert` 版本返回 `pair` 类型对象, 包含一个迭代器和一个 `bool` 值, 迭代器指向拥有该键的元素, 而 `bool` 值表明是否添加了元素。使用迭代器对的 `insert` 版本返回 `void` 类型。

Fetching an Element from a `set`

从 `set` 中获取元素

There is no subscript operator on `sets`. To fetch an element from a `set` by its key, we use the `find` operation. If we just want to know whether the element is present, we could also use `count`, which returns the number of elements in the `set` with a given key. Of course, for `set` that value can be only one (if the element is present) or zero (if it is not):

`set` 容器不提供下标操作符。为了通过键从 `set` 中获取元素, 可使用 `find` 运算。如果只需简单地判断某个元素是否存在, 同样可以使用 `count` 运算, 返回 `set` 中该键对应的元素个数。当然, 对于 `set` 容器, `count` 的返回值只能是 1 (该元素存在) 或 0 (该元素不存在) :

```
iset.find(1)      // returns iterator that refers to the element with key == 1
iset.find(11)     // returns iterator == iset.end()

iset.count(1)     // returns 1
iset.count(11)    // returns 0
```

Just as we cannot change the key part of a `map` element, the keys in a `set` are also `const`. If we have an iterator to an element of the `set`, all we can do is read it; we cannot write through it:

Section 10.4. The set Type

正如不能修改 `map` 中元素的键部分一样，`set` 中的键也为 `const`。在获得指向 `set` 中某元素的迭代器后，只能对其做读操作，而不能做写操作：

```
// set_it refers to the element with key == 1
set<int>::iterator set_it = iset.find(1);
*set_it = 11;           // error: keys in a set are read-only
cout << *set_it << endl; // ok: can read the key
```

10.4.2. Building a Word-Exclusion Set

10.4.2. 创建“单词排除”集

On page 369 we removed a given word from our `word_count` `map`. We might want to extend this approach to remove all the words in a specified file. That is, our word-count program should count only words that are not in a set of excluded words. Using `set` and `map`, this program is fairly straightforward:

第 10.3.7 节的程序从 `map` 对象 `word_count` 中删除一个指定的单词。可将这个操作扩展为删除指定文件中所有的单词（即该文件记录的是排除集）。也即，我们的单词统计程序只对那些不在排除集中的单词进行统计。使用 `set` 和 `map` 容器，可以简单而直接地实现该功能：

```
void restricted_wc(ifstream &remove_file,
                    map<string, int> &word_count)
{
    set<string> excluded; // set to hold words we'll ignore
    string remove_word;
    while (remove_file >> remove_word)
        excluded.insert(remove_word);
    // read input and keep a count for words that aren't in the exclusion set
    string word;
    while (cin >> word)
        // increment counter only if the word is not in excluded
        if (!excluded.count(word))
            ++word_count[word];
}
```

This program is similar to the word-count program on page 363. The difference is that we do not bother to count the common words.

这个程序类似第 10.3.4 节的单词统计程序。其差别在于不需要费力地统计常见的单词。

The function starts by reading the file it was passed. That file contains the list of excluded words, which we store in the `set` named `excluded`. When the first `while` completes, that `set` contains an entry for each word in the input file.

该函数首先读取传递进来的文件，该文件列出了所有被排除的单词。读入这些单词并存储在一个名为 `excluded` 的 `set` 容器中。第一个 `while` 循环完成时，该 `set` 对象包含了输入文件中的所有单词。

The next part of the program looks a lot like our original word-count program. The important difference is that before counting each word, we check whether the word is in the exclusion set. We do this check in the `if` inside the second `while`:

接下来的程序类似原来的单词统计程序。关键的区别在于：在统计每个单词之前，先检查该单词是否出现在排除集中。第二个 `while` 循环里的 `if` 语句实现了该功能：

```
// increment counter only if the word is not in excluded
if (!excluded.count(word))
```

The call to `count` returns one if `word` is in `excluded` and zero otherwise. We negate the return from `count` so that the test succeeds if `word` is not in `excluded`. If `word` is not in `excluded`, we update its value in the `map`.

如果该单词出现在排除集 `excluded` 中，则调用 `count` 将返回 1，否则返回 0。对 `count` 的返回值做“非”运算，则当该 `word` 不在 `excluded` 中时，条件测试成功，此时修改该单词在 `map` 中对应的值。

As in the previous version of our word count program, we rely on the fact that subscripting a `map` inserts an element if the key is not already in the `map`. Hence, the effect of

与单词统计程序原来的版本一样，需要使用下标操作符的性质：如果某键尚未在 `map` 容器中出现，则将该元素插入容器。所以语句

```
++word_count[word];
```

is to insert `word` into `word_count` if it wasn't already there. If the element is inserted, its value is initially set to 0. Regardless of whether the element had to be created, the value is then incremented.

的效果是：如果 `word` 还没出现过，则将它插入到 `word_count` 中，并在插入元素后，将它关联的值初始化为 0。然后不管是否插入了新元素，相应元素的值都加 1。

Exercises Section 10.4.2

Exercise 10.23: Write a program that stores the excluded words in a `vector` instead of in a `set`. What are the advantages to using a `set`?

编写程序将被排除的单词存储在 `vector` 对象中，而不是存储在 `set` 对象中。请指出使用 `set` 的好处。

Exercise 10.24: Write a program that generates the non-plural version of a word by stripping the '`s`' off the end of the word. Build an exclusion set to recognize words in which the trailing '`s`' should not be removed. Two examples of words to place in this set are `success`, `class`. Use this exclusion set to write a program that strips plural suffixes from its input but leaves words in the exclusion set unchanged.

编写程序通过删除单词尾部的'`s`'生成该单词的非复数版本。同时，建立一个单词排除集，用于识别以'`s`'结尾、但这个结尾的'`s`'又不能删除的单词。例如，放在该排除集中的单词可能有 `success` 和 `class`。使用这个排除集编写程序，删除输入单词的复数后缀，而如果输入的是排除集中的单词，则保持该单词不变。

Exercise 10.25: Define a `vector` of books you'd like to read within the next six months and a set of titles that you've read. Write a program that chooses a next book for you to read from the `vector`, provided that you have not yet read it. When it returns the selected title to you, it should enter the title in the set of books read. If in fact you end up putting the book aside, provide support for removing the title from the set of books read. At the end of our virtual six months, print the set of books read and those books that were not read.

定义一个 `vector` 的容器，存储你在未来六个月里要阅读的书，再定义一个 `set`，用于记录你已经看过的书名。编写程序从 `vector` 中为你选择一本没有读过而现在要读的书。当它为你返回选中的书名后，应该将该书名放入记录已读书目的 `set` 中。如果实际上你把这本书放在一边没有看，则本程序应该支持从已读书目的 `set` 中删除该书的记录。在虚拟的六个月后，输出已读书目和还没有读的书目。

10.5. The `multimap` and `multiset` Types

10.5. `multimap` 和 `multiset` 类型

Both `map` and `set` can contain only a single instance of each key. The types `multiset` and a `multimap` allow multiple instances of a key. In a phone directory, for example, someone might wish to provide a separate listing for each phone number associated with an individual. A listing of available texts by an author might provide a separate listing for each title. The `multimap` and `multiset` types are defined in the same headers as the corresponding single-element versions: the `map` and `set` headers, respectively.

`map` 和 `set` 容器中，一个键只能对应一个实例。而 `multiset` 和 `multimap` 类型则允许一个键对应多个实例。例如，在电话簿中，每个人可能有单独的电话号码列表。在作者的文章集中，每位作者可能有单独的文章标题列表。`multimap` 和 `multiset` 类型与相应的单元素版本具有相同的头文件定义：分别是 `map` 和 `set` 头文件。

The operations supported by `multimap` and `multiset` are the same as those on `map` and `set`, respectively, with one exception: `multimap` does not support subscripting. We cannot subscript a `multimap` because there may be more than one value associated with a given key. The operations that are common to both `map` and `multimap` or `set` and `multiset` change in various ways to reflect the fact that there can be multiple keys. When using either a `multimap` or `multiset`, we must be prepared to handle multiple values, not just a single value.

`multimap` 和 `multiset` 所支持的操作分别与 `map` 和 `set` 的操作相同，只有一个例外：`multimap` 不支持下标运算。不能对 `multimap` 对象使用下标操作，因为在这类容器中，某个键可能对应多个值。为了顺应一个键可以对应多个值这一性质，`map` 和 `multimap`，或 `set` 和 `multiset` 中相同的操作都以不同的方式做出了一定的修改。在使用 `multimap` 或 `multiset` 时，对于某个键，必须做好处理多个值的准备，而非只有单一的值。

10.5.1. Adding and Removing Elements

10.5.1. 元素的添加和删除

The `insert` operations described in [Table 10.5](#) (p. 365) and the `erase` operations described in [Table 10.7](#) (p. 369) are used to add and remove elements of a `multimap` or `multiset`.

[表 10.5](#) 描述的 `insert` 操作和 [表 10.7](#) 描述的 `erase` 操作同样适用于 `multimap` 以及 `multiset` 容器，实现元素的添加和删除。

Because keys need not be unique, `insert` always adds an element. As an example, we might define a `multimap` to map authors to titles of the books they have written. The `map` might hold multiple entries for each author:

由于键不要求是唯一的，因此每次调用 `insert` 总会添加一个元素。例如，可如下定义一个 `multimap` 容器对象将作者映射到他们所写的书的书名上。这样的映射可为一个作者存储多个条目：

```
// adds first element with key Barth
authors.insert(make_pair(
    string("Barth, John"),
    string("Sot-Weed Factor")));

// ok: adds second element with key Barth
authors.insert(make_pair(
    string("Barth, John"),
    string("Lost in the Funhouse")));
```

The version of `erase` that takes a key removes *all* elements with that key. It returns a count of how many elements were removed. The versions that take an iterator or an iterator pair remove only the indicated element(s). These versions return `void`:

带有一个键参数的 `erase` 版本将删除拥有该键的所有元素，并返回删除元素的个数。而带有一个或一对迭代器参数的版本只删除指定的元素，并返回 `void` 类型：

```
multimap<string, string> authors;
string search_item("Kazuo Ishiguro");

// erase all elements with this key; returns number of elements removed
multimap<string, string>::size_type cnt =
    authors.erase(search_item);
```

10.5.2. Finding Elements in a `multimap` or `multiset`

10.5.2. 在 `multimap` 和 `multiset` 中查找元素

Section 10.5. The multimap and multiset Types

We noted that `maps` and `sets` store their elements in order. The `multimap` and `multiset` types do so as well. As a result, when a `multimap` or `multiset` has multiple instances of a given key, those instances will be adjacent elements within the container.

注意到，关联容器 `map` 和 `set` 的元素是按顺序存储的。而 `multimap` 和 `multiset` 也一样。因此，在 `multimap` 和 `multiset` 容器中，如果某个键对应多个实例，则这些实例在容器中将相邻存放。



We are guaranteed that iterating across a `multimap` or `multiset` returns all the elements with a given key in sequence.

迭代遍历 `multimap` 或 `multiset` 容器时，可保证依次返回特定键所关联的所有元素。

Finding an element in a `map` or a `set` is a simple matter—the element is or is not in the container. For `multimap` and `multiset` the process is more complicated: the element may be present many times. For example, given our map from author to titles, we might want to find and print all the books by a particular author.

在 `map` 或 `set` 容器中查找一个元素很简单——该元素要么在要么不在容器中。但对于 `multimap` 或 `multiset`，该过程就复杂多了：某键对应的元素可能出现多次。例如，假设有作者与书名的映射，我们可能希望找到并输出某个作者写的所有书的书名。

It turns out that there are three strategies we might use to find and print all the books by a given author. Each of these strategies relies on the fact that we know that all the entries for a given author will be adjacent within the `multimap`.

事实证明，上述问题可用三种策略解决。而且三种策略都基于一个事实——在 `multimap` 中，同一个键所关联的元素必然相邻存放。

We'll start by presenting a strategy that uses only functions we've already seen. This version turns out to require the most code, so we will continue by exploring more compact alternatives.

首先介绍第一种策略：仅使用前面介绍过的函数。但这种方法要编写比较多的代码，所以我们将继续探索更简洁的方法。

Using `find` and `count`

使用 `find` 和 `count` 操作

We could solve our problem using `find` and `count`. The `count` function tells us how many times a given key occurs, and the `find` operation returns an iterator that refers to the first instance of the key we're looking for:

使用 `find` 和 `count` 可有效地解决刚才的问题。`count` 函数求出某键出现的次数，而 `find` 操作则返回一个迭代器，指向第一个拥有正在查找的键的实例：

```
// author we'll look for
string search_item("Alain de Botton");

// how many entries are there for this author
typedef multimap<string, string>::size_type sz_type;
sz_type entries = authors.count(search_item);

// get iterator to the first entry for this author
multimap<string, string>::iterator iter =
    authors.find(search_item);

// loop through the number of entries there are for this author
for (sz_type cnt = 0; cnt != entries; ++cnt, ++iter) cout <<
    iter->second << endl; // print each title
```

We start by determining how many entries there are for the author by calling `count` and getting an iterator to the first element with this key by calling `find`. The number of iterations of the `for` loop depends on the number returned from `count`. In particular, if the `count` was zero, then the loop is never executed.

首先，调用 `count` 确定某作者所写的书籍数目，然后调用 `find` 获得指向第一个该键所关联的元素的迭代器。`for` 循环迭代的次数依赖于 `count` 返回的值。在特殊情况下，如果 `count` 返回 0 值，则该循环永不执行。

A Different, Iterator-Oriented Solution

与众不同的面向迭代器的解决方案

Another, more elegant strategy uses two associative container operations that we haven't seen yet: `lower_bound` and `upper_bound`. These operations,

Section 10.5. The multimap and multiset Types

listed in [Table 10.8](#) (p. 379), apply to all associative containers. They can be used with (plain) `maps` or `sets` but are most often used with `multimaps` or `multisets`. Each of these operations takes a key and returns an iterator.

另一个更优雅简洁的方法是使用两个未曾见过的关联容器的操作：`lower_bound` 和 `upper_bound`。[表 10.8](#) 列出的这些操作适用于所有的关联容器，也可用于普通的 `map` 和 `set` 容器，但更常用于 `multimap` 和 `multiset`。所有这些操作都需要传递一个键，并返回一个迭代器。

Table 10.8. Associative Container Operations Returning Iterators

表 10.8. 返回迭代器的关联容器操作

<code>m.lower_bound(k)</code>	Returns an iterator to the first element with key not less than <code>k</code> .
	返回一个迭代器，指向键不小于 <code>k</code> 的第一个元素
<code>m.upper_bound(k)</code>	Returns an iterator to the first element with key greater than <code>k</code> .
	返回一个迭代器，指向键大于 <code>k</code> 的第一个元素
<code>m.equal_range(k)</code>	Returns a <code>pair</code> of iterators.
	返回一个迭代器的 <code>pair</code> 对象
	The <code>first</code> member is equivalent to <code>m.lower_bound(k)</code> and <code>second</code> is equivalent to <code>m.upper_bound(k)</code> .
	它的 <code>first</code> 成员等价于 <code>m.lower_bound(k)</code> 。而 <code>second</code> 成员则等价于 <code>m.upper_bound(k)</code>

Calling `lower_bound` and `upper_bound` on the same key yields an iterator range ([Section 9.2.1](#), p. 314) that denotes all the elements with that key. If the key is in the container, the iterators will differ: the one returned from `lower_bound` will refer to the first instance of the key, whereas `upper_bound` will return an iterator referring just after the last instance. If the element is not in the `multimap`, then `lower_bound` and `upper_bound` will return equal iterators; both will refer to the point at which the key could be inserted without disrupting the order.

在同一个键上调用 `lower_bound` 和 `upper_bound`，将产生一个迭代器范围（[第 9.2.1 节](#)），指示出该键所关联的所有元素。如果该键在容器中存在，则会获得两个不同的迭代器：`lower_bound` 返回的迭代器指向该键关联的第一个实例，而 `upper_bound` 返回的迭代器则指向最后一个实例的下一位置。如果该键不在 `multimap` 中，这两个操作将返回同一个迭代器，指向依据元素的排列顺序该键应该插入的位置。

Of course, the iterator returned from these operations might be the off-the-end iterator for the container itself. If the element we look for has the largest key in the `multimap`, then `upper_bound` on that key returns the off-the-end iterator. If the key is not present and is larger than any key in the `multimap`, then the return from `lower_bound` will also be the off-the-end iterator.

当然，这些操作返回的也可能是容器自身的超出末端迭代器。如果所查找的元素拥有 `multimap` 容器中最大的键，那么的该键上调用 `upper_bound` 将返回超出末端迭代器。如果所查找的键不存在，而且比 `multimap` 容器中所有的键都大，则 `lower_bound` 也将返回超出末端迭代器。



The iterator returned from `lower_bound` may or may not refer to an element with the given key. If the key is not in the container, then `lower_bound` refers to the first point at which this key could be inserted while preserving the element order within the container.

`lower_bound` 返回的迭代器不一定指向拥有特定键的元素。如果该键不在容器中，则 `lower_bound` 返回在保持容器元素顺序的前提下该键应被插入的第一个位置。

Using these operations, we could rewrite our program as follows:

使用这些操作，可如下重写程序：

```
// definitions of authors and search_item as above

// beg and end denote range of elements for this author
typedef multimap<string, string>::iterator authors_it;
authors_it beg = authors.lower_bound(search_item),
           end = authors.upper_bound(search_item);

// loop through the number of entries there are for this author
while (beg != end) {
    cout << beg->second << endl; // print each title
    ++beg;
```

Section 10.5. The multimap and multiset Types

}

This program does the same work as the previous one that used `count` and `find` but accomplishes its task more directly. The call to `lower_bound` positions `beg` so that it refers to the first element matching `search_item` if there is one. If there is no such element, then `beg` refers to first element with a key larger than `search_item`. The call to `upper_bound` sets `end` to refer to the element with the key just beyond the last element with the given key.

这个程序实现的功能与前面使用 `count` 和 `find` 的程序相同，但任务的实现更直接。调用 `lower_bound` 定位 `beg` 迭代器，如果键 `search_item` 在容器中存在，则使 `beg` 指向第一个与之匹配的元素。如果容器中没有这样的元素，那么 `beg` 将指向第一个键比 `search_item` 大的元素。调用 `upper_bound` 设置 `end` 迭代器，使之指向拥有该键的最后一个元素的下一位。



These operations say nothing about whether the key is present. The important point is that the return values act like an iterator range.

这两个操作不会说明键是否存在，其关键之处在于返回值给出了迭代器范围。

If there is no element for this key, then `lower_bound` and `upper_bound` will be equal: They will both refer to the same element or they will both point one past the end of the `multimap`. They both will refer to the point at which this key could be inserted while maintaining the container order.

若该键没有关联的元素，则 `lower_bound` 和 `upper_bound` 返回相同的迭代器：都指向同一个元素或同时指向 `multimap` 的超出末端位置。它们都指向在保持容器元素顺序的前提下该键应被插入的位置。

If there are elements with this key, then `beg` will refer to the first such element. We can increment `beg` to traverse the elements with this key. The iterator in `end` will signal when we've seen all the elements. When `beg` equals `end`, we have seen every element with this key.

如果该键所关联的元素存在，那么 `beg` 将指向满足条件的元素中的第一个。可对 `beg` 做自增运算遍历拥有该键的所有元素。当迭代器累加至 `end` 标志时，表示已遍历了所有这些元素。当 `beg` 等于 `end` 时，表示已访问所有与该键关联的元素。

Given that these iterators form a range, we can use the same kind of `while` loop that we've used to traverse other ranges. The loop is executed zero or more times and prints the entries, if any, for the given author. If there are no elements, then `beg` and `end` are equal and the loop is never executed. Otherwise, we know that the increment to `beg` will eventually reach `end` and that in the process we will print each record associated with this author.

假设这些迭代器标记某个范围，可使用同样的 `while` 循环遍历该范围。该循环执行 0 次或多次，输出指定作者所写的所有书的书名（如果有的话）。如果没有相关的元素，那么 `beg` 和 `end` 相等，循环永不执行。否则，不断累加 `beg` 将最终到达 `end`，在这个过程中可输出该作者所关联的记录。

The `equal_range` Function

`equal_range` 函数

It turns out that there is an even more direct way to solve this problem: Instead of calling `upper_bound` and `lower_bound`, we can call `equal_range`. This function returns a `pair` of iterators. If the value is present, then the first iterator refers to the first instance of the key and the second iterator refers one past the last instance of the key. If no matching element is found, then both the first and second iterator refer to the position where this key could be inserted.

事实上，解决上述问题更直接的方法是：调用 `equal_range` 函数来取代调用 `upper_bound` 和 `lower_bound` 函数。`equal_range` 函数返回存储一对迭代器的 `pair` 对象。如果该值存在，则 `pair` 对象中的第一个迭代器指向该键关联的第一个实例，第二个迭代器指向该键关联的最后一个实例的下一位。如果找不到匹配的元素，则 `pair` 对象中的两个迭代器都将指向此键应该插入的位置。

We could use `equal_range` to modify our program once again:

使用 `equal_range` 函数再次修改程序：

```
// definitions of authors and search_item as above

// pos holds iterators that denote range of elements for this key
pair<authors_it, authors_it>
pos = authors.equal_range(search_item);

// loop through the number of entries there are for this author
while (pos.first != pos.second) {
    cout << pos.first->second << endl; // print each title
    ++pos.first;
}
```

This program is essentially identical to the previous one that used `upper_bound` and `lower_bound`. Instead of using local variables, `beg` and `end`, to hold the iterator range, we use the `pair` returned by `equal_range`. The `first` member of that `pair` holds the same iterator as the one `lower_bound` would have returned. The iterator that `upper_bound` would have returned is in the `second` member.

这个程序段与前面使用 `upper_bound` 和 `lower_bound` 的程序基本上是相同的。本程序不用局部变量 `beg` 和 `end` 来记录迭代器范围，而是直接使用 `equal_range` 返回的 `pair` 对象。该 `pair` 对象的 `first` 成员存储 `lower_bound` 函数返回的迭代器，而 `second` 成员则记录 `upper_bound` 函数返回的迭代器。

Section 10.5. The multimap and multiset Types

Thus, in this program `pos.first` is equivalent to `beg`, and `pos.second` is equivalent to `end`.

因此, 本程序的 `pos.first` 等价于前一方法中的 `beg`, 而 `pos.second` 等价于 `end`。

Exercises Section 10.5.2

- Exercise 10.26:** Write a program that populates a `multimap` of authors and their works. Use `find` to find an element in the `multimap` and `erase` that element. Be sure your program works correctly if the element you look for is not in the `map`.

编写程序建立作者及其作品的 `multimap` 容器。使用 `find` 函数在 `multimap` 中查找元素，并调用 `erase` 将其删除。当所寻找的元素不存在时，确保你的程序依然能正确执行。

- Exercise 10.27:** Repeat the program from the previous exercise, but this time use `equal_range` to get iterators so that you can `erase` a range of elements.

重复上一题所编写的程序，但这一次要求使用 `equal_range` 函数获取迭代器，然后删除一段范围内的元素。

- Exercise 10.28:** Using the `multimap` from the previous exercise, write a program to generate the list of authors whose name begins with the each letter in the alphabet. Your output should look something like:

沿用上题中的 `multimap` 容器，编写程序以下面的格式按姓名首字母的顺序输出作者名字：

```
Author Names Beginning with 'A':  
Author, book, book, ...  
...  
Author Names Beginning with 'B':  
...
```

- Exercise 10.29:** Explain the meaning of the operand `pos.first->second` used in the output expression of the final program in this section.

解释本节最后一个程序的输出表达式使用操作数 `pos.first->second` 的含义。

10.6. Using Containers: Text-Query Program

10.6. 容器的综合应用：文本查询程序

To conclude this chapter, we'll implement a simple text-query program.

我们将实现一个简单的文本查询程序来结束本章。

Our program will read a file specified by the user and then allow the user to search the file for words that might occur in it. The result of a query will be the number of times the word occurs and a list of lines on which it appears. If a word occurs more than once on the same line, our program should be smart enough to display that line only once. Lines should be displayed in ascending order that is, line 7 should be displayed before line 9, and so on.

我们的程序将读取用户指定的任意文本文件，然后允许用户从该文件中查找单词。查询的结果是该单词出现的次数，并列出每次出现所在的行。如果某单词在同一行中多次出现，程序将只显示该行一次。行号按升序显示，即第 7 行应该在第 9 行之前输出，依此类推。

For example, we might read the file that contains the input for this chapter and look for the word "element." The first few lines of the output would be:

例如，以本章的内容作为文件输入，然后查找单词"element"。输出的前几行应为：

```
element occurs 125 times
(line 62) element with a given key.
(line 64) second element with the same key.
(line 153) element [=] operator.
(line 250) the element type.
(line 398) corresponding element.
```

followed by the remaining 120 or so lines in which the word "element" occurs.

后面省略了大约 120 行。

10.6.1. Design of the Query Program

10.6.1. 查询程序的设计

A good way to start the design of a program is to list the program's operations. Knowing what operations we need to provide can then help us see what data structures we'll need and how we might implement those actions. Starting from requirements, the tasks our program needs to support include:

设计程序的一个良好习惯是首先将程序所涉及的操作列出来。明确需要提供的操作有助于建立需要的数据结构和实现这些行为。从需求出发，我们的程序需要支持如下任务：

1. It must allow the user to indicate the name of a file to process. The program will store the contents of the file so that we can display the original line in which each word appears.
它必须允许用户指明要处理的文件名字。程序将存储该文件的内容，以便输出每个单词所在的原始行。
2. The program must break each line into words and remember all the lines in which each word appears. When it prints the line numbers, they should be presented in ascending order and contain no duplicates.
它必须将每一行分解为各个单词，并记录每个单词所在的所有行。在输出行号时，应保证以升序输出，并且不重复。
3. The result of querying for a particular word should be the line numbers on which that word appeared.
对特定单词的查询将返回出现该单词的所有行的行号。
4. To print the text in which the word appeared, it must be possible to fetch the line from the input file corresponding to a given line number.
输出某单词所在的行文本时，程序必须能根据给定的行号从输入文件中获取相应的行。

Data Structure

数据结构

Section 10.6. Using Containers: Text-Query Program

We'll implement our program as a simple class that we'll name `TextQuery`. Our requirements can be met quite neatly by using various containers:

我们将用一个简单的类 `TextQuery` 实现这个程序。再加几种容器的配合使用，就可相当巧妙地满足上述要求。

1. We'll use a `vector<string>` to store a copy of the entire input file. Each line in the input file will be an element in this `vector`. That way, when we want to print a line, we can fetch it by using the line number as an index.

使用一个 `vector<string>` 类型的对象存储整个输入文件的副本。输入文件的每一行是该 `vector` 对象的一个元素。因而，在希望输出某一行时，只需以行号为下标获取该行所在的元素即可。

2. We'll store each word's line numbers in a `set`. Using a `set` will guarantee that there is only one entry per line and that the line numbers will be automatically stored in ascending order.

将每个单词所在的行号存储在一个 `set` 容器对象中。使用 `set` 就可确保每行只有一个条目，而且行号将自动按升序排列。

3. We'll use a `map` to associate each word with the `set` of line numbers on which the word appears.

使用一个 `map` 容器将每个单词与一个 `set` 容器对象关联起来，该 `set` 容器对象记录此单词所在的行号。

Our class will have two data members: a `vector` to hold the input file and a `map` to associate each input word with the `set` of line numbers on which it appears.

综上所述，我们定义的 `TextQuery` 类将有两个数据成员：储存输入文件的 `vector` 对象，以及一个 `map` 容器对象，该对象关联每个输入的单词以及记录该单词所在行号的 `set` 容器对象。

Operations

操作

The requirements also lead fairly directly to an interface for our class. However, we have one important design decision to make first: The function that does the query will need to return a `set` of line numbers. What type should it use to do so?

对于类还要求有良好的接口。然而，一个重要的设计策略首先要确定：查询函数需返回存储一组行号的 `set` 对象。这个返回类型应该如何设计呢？

We can see that doing the query will be simple: We'll index into the `map` to obtain the associated `set`. The only question is how to return the `set` that we find. The safest design is to return a copy of that `set`. However, doing so means that each element in the `set` must be copied. Copying the `set` could be expensive if we process a very large file. Other possible return values are a `pair` of iterators into the `set`, or a `const` reference to the `set`. For simplicity, we'll return a copy, noting that this decision is one that we might have to revisit if the copy is too expensive in practice.

事实上，查询的过程相当简单：使用下标访问 `map` 对象获取关联的 `set` 对象即可。唯一的问题是如何返回所找到的 `set` 对象。安全的设计方案是返回该 `set` 对象的副本。但如此一来，就意味着要复制 `set` 中的每个元素。如果处理的是一个相当庞大的文件，则复制 `set` 对象的代价会非常昂贵。其他可行的方法包括：返回一个 `pair` 对象，存储一对指向 `set` 中元素的迭代器；或者返回 `set` 对象的 `const` 引用。为简单起见，我们在这里采用返回副本的方法，但注意：如果在实际应用中复制代价太大，需要新考虑其实现方法。

The first, third, and fourth tasks are actions programmers using our class will perform. The second task is internal to the class. Mapping these tasks to member functions, we'll have three `public` functions in the class interface:

第一、第三和第四个任务是使用这个类的程序员将执行的动作。第二个任务则是类的内部任务。将这四任务映射为类的成员函数，则类的接口需提供下列三个 `public` 函数：

- `read_file` takes an `ifstream&`, which it reads a line at a time, storing the lines in the `vector`. Once it has read all the input, `read_file` will create the `map` that associates each word to the line numbers on which it appears.
`read_file` 成员函数，其形参为一个 `ifstream&` 类型对象。该函数每次从文件中读入一行，并将它保存在 `vector` 容器中。输入完毕后，`read_file` 将创建关联每个单词及其所在行号的 `map` 容器。
- `run_query` takes a `string` and returns the `set` of line numbers on which that `string` appears.
`run_query` 成员函数，其形参为一个 `string` 类型对象，返回一个 `set` 对象，该 `set` 对象包含出现该 `string` 对象的所有行的行号。
- `text_line` takes a line number and returns the corresponding text for that line from the input file.
`text_line` 成员函数，其形参为一个行号，返回输入文本中该行号对应的文本行。

Neither `run_query` nor `text_line` changes the object on which it runs, so we'll define these operations as `const` member functions ([Section 7.7.1, p. 260](#)).

无论 `run_query` 还是 `text_line` 都不会修改调用此函数的对象，因此，可将这两个操作定义为 `const` 成员函数（[第 7.7.1 节](#)）。

To do the work of `read_file`, we'll also define two `private` functions to read the input file and build the `map`:

为实现 `read_file` 功能，还需定义两个 `private` 函数来读取输入文本和创建 `map` 容器：

Section 10.6. Using Containers: Text-Query Program

- `store_file` will read the file and store the data in our `vector`.
`store_file` 函数读入文件，并将文件内容存储在 `vector` 容器对象中。
- `build_map` will break each line into words and build the `map`, remembering the line number on which each word appeared.
`build_map` 函数将每一行分解为各个单词，创建 `map` 容器对象，同时记录每个单词出现行号。

10.6.2. `TextQuery` Class

10.6.2. `TextQuery` 类

Having worked through our design, we can now write our `TextQuery` class:

经过前面的设计后，现在可以编写 `TextQuery` 类了：

```
class TextQuery {  
public:  
    // typedef to make declarations easier  
    typedef std::vector<std::string>::size_type line_no;  
    /* interface:  
     * read_file builds internal data structures for the given file  
     * run_query finds the given word and returns set of lines on which it appears  
     * text_line returns a requested line from the input file  
     */  
    void read_file(std::ifstream &is)  
    { store_file(is); build_map(); }  
    std::set<line_no> run_query(const std::string&) const;  
    std::string text_line(line_no) const;  
private:  
    // utility functions used by read_file  
    void store_file(std::ifstream&); // store input file  
    void build_map(); // associated each word with a set of line numbers  
    // remember the whole input file  
    std::vector<std::string> lines_of_text;  
    // map word to set of the lines on which it occurs  
    std::map<std::string, std::set<line_no>> word_map;  
};
```

The class directly reflects our design decisions. The only part we hadn't described is the `typedef` that defines an alias for `size_type` of `vector`.

这个类直接反映了我们的设计策略。唯一提及的是使用 `typedef` 为 `vector` 的 `size_type` 定义了一个别名。



For the reasons described on page 80, this class definition uses fully qualified `std::` names for all references to library entities.

基于第 3.1 节所提及的原因，这个类的定义在引用标准库内容时都必须完整地使用 `std::` 限定符。

The `read_file` function is defined inside the class. It calls `store_file` to read and store the input file and `build_map` to build the `map` from words to line numbers. We'll define the other functions in [Section 10.6.4](#) (p. 385). First, we'll write a program that uses this class to solve our text-query problem.

`read_file` 函数在类的内部定义。该函数首先调用 `store_file` 读取并保存输入文件，然后调用 `build_map` 创建关联单词与行号的 `map` 容器。该类的其他函数将在[第 10.6.4 节](#) 定义。首先，我们编写一个程序，使用这个类来解决文本查询问题。

Exercises Section 10.6.2

Exercise 10.30: The member functions of `TextQuery` use only capabilities that we have already covered. Without looking ahead, write your own versions of the member functions. Hint: The only tricky part is what to return from `run_query` if the line number set is empty. The solution is to construct and return a new (temporary) `set`.

`TextQuery` 类的成员函数仅使用了前面介绍过的内容。先别查看后面章节，请自己编写这些成员函数。提示：唯一

棘手的是 `run_query` 函数在行号集合 `set` 为空时应返回什么值？解决方法是构造并返回一个新的（临时）`set` 对象。

10.6.3. Using the `TextQuery` Class

10.6.3. `TextQuery` 类的使用

The following `main` program uses a `TextQuery` object to perform a simple query session with the user. Most of the work in this program involves managing the interaction with the user: prompting for the next search word and calling the `print_results` function which we shall write next to print the results.

下面的主程序 `main` 使用 `TextQuery` 对象实现简单的用户查询会话。这段程序的主要工作是实现与用户的互动：提示输入下一个要查询的单词，然后调用 `print_results` 函数（将在下面定义）输出结果。

```
// program takes single argument specifying the file to query
int main(int argc, char **argv)
{
    // open the file from which user will query words
    ifstream infile;
    if (argc < 2 || !open_file(infile, argv[1])) {
        cerr << "No input file!" << endl;
        return EXIT_FAILURE;
    }
    TextQuery tq;
    tq.read_file(infile); // builds query map
    // iterate with the user: prompt for a word to find and print results
    // loop indefinitely; the loop exit is inside the while
    while (true) {
        cout << "enter word to look for, or q to quit: ";
        string s;
        cin >> s;
        // stop if hit eof on input or a 'q' is entered
        if (!cin || s == "q") break;
        // get the set of line numbers on which this word appears
        set<TextQuery::line_no> locs = tq.run_query(s);
        // print count and all occurrences, if any
        print_results(locs, s, tq);
    }
    return 0;
}
```

Preliminaries

引子

This program checks that `argv[1]` is valid and then uses the `open_file` function ([Section 8.4.3, p. 299](#)) to open the file we're given as an argument to `main`. We test the stream to determine whether the input file is okay. If not, we generate an appropriate message and exit, returning `EXIT_FAILURE` ([Section 7.3.2, p. 247](#)) to indicate that an error occurred.

程序首先检查 `argv[1]` 是否合法，然后调用 `open_file` 函数 ([第 8.4.3 节](#)) 打开以 `main` 函数实参形式给出的文件。检查流以判断输入文件是否正确。如果不正确，就给出适当的提示信息结束程序的运行，返回 `EXIT_FAILURE` ([第 7.3.2 节](#)) 说明发生了错误。

Once we have opened the file, it is a simple matter to build up the `map` that will support queries. We define a local variable named `tq` to hold the file and associated data structures:

一旦文件成功打开，建立支持查询的 `map` 容器就相当简单。定义一个局部变量 `tq` 来保存该输入文件和所关联的数据结构：

```
TextQuery tq;
tq.read_file(infile); // builds query map
```

We call the `read_file` operation on `tq`, passing it the file opened by `open_file`.

`tq` 调用 `read_file` 操作，并将由 `open_file` 打开的文件传递给此函数。

After `read_file` completes, `tq` holds our two data structures: the `vector` that corresponds to the input file and the `map` from word to set of line numbers. That `map` contains an entry for each unique word in the input file. The `map` associates with each word the `set` of line numbers on which that word appeared.

Section 10.6. Using Containers: Text-Query Program

`read_file` 完成后，`tq` 存储了两个数据结构：保存输入文件的 `vector` 对象，以及关联单词和行号的 `map` 容器对象。`map` 容器为输入文件中的每个单词建立唯一的元素，由每个单词关联的 `set` 容器记录了该单词出现的行号。

Doing the Search

实现查询

We want the program to let the user look for more than one word in each session, so we wrap the prompt in a `while` loop:

为了使用户在每次会话时都能查询多个单词，我们将提示语句也置于 `while` 循环中：

```
// iterate with the user: prompt for a word to find and print results
// loop indefinitely; the loop exit is inside the while
while (true) {
    cout << "enter word to look for, or q to quit: ";
    string s;
    cin >> s;
    // stop if hit eof on input or a 'q' is entered
    if (!cin || s == "q") break;
    // get the set of line numbers on which this word appears
    set<TextQuery::line_no> locs = tq.run_query(s);
    // print count and all occurrences, if any
    print_results(locs, s, tq);
}
```

The test in the `while` is the boolean literal `true`, which means that the test always succeeds. We exit the loop through the `break` after the test on `cin` and the value read into `sought`. The loop exits when `cin` hits an error or end-of-file or when the user enters a '`q`' to quit.

`while` 循环条件为布尔字面值 `true`，这就意味着循环条件总是成立。在检查 `cin` 和读入 `s` 值后，由紧跟的 `break` 语句跳出循环。具体说来，当 `cin` 遇到错误或文件结束，或者用户输入 `q` 时，循环结束。

Once we have a word to look for, we ask `tq` for the `set` of line numbers on which that word appears. We pass that `set` along with the word we are looking for and the `TextQuery` object to the `print_results` function. That function will write the output of our program.

每次要查找一个单词时，访问 `tq` 获取记录该单词出现的行号的 `set` 对象。将 `set` 对象、要查找的单词和 `TextQuery` 对象作为参数传递给 `print_results` 函数，该函数输出查询结果。

Printing the Results

输出结果

What remains is to define the `print_results` function:

现在只剩下 `print_results` 函数的定义：

```
void print_results(const set<TextQuery::line_no>& locs,
                   const string& sought, const TextQuery &file)
{
    // if the word was found, then print count and all occurrences
    typedef set<TextQuery::line_no> line_nums;
    line_nums::size_type size = locs.size();
    cout << "\n" << sought << " occurs "
        << size << " "
        << make_plural(size, "time", "s") << endl;

    // print each line in which the word appeared
    line_nums::const_iterator it = locs.begin();
    for ( ; it != locs.end(); ++it) {
        cout << "\t(line "
            // don't confound user with text lines starting at 0
            << (*it) + 1 << " )"
            << file.text_line(*it) << endl;
    }
}
```

The function starts by defining a `typedef` to simplify the use of the line number `set`. Our output first reports how many matches were found, which we know from the `size` of the `set`. We call `make_plural` ([Section 7.3.2](#), p. 248) to print `time` or `times`, depending on whether that `size` is equal to one.

函数首先使用 `typedef` 简化记录行号的 `set` 容器对象的使用。输出时，首先给出查询到的匹配个数，即 `set` 对象的大小。然后调用 `make_plural` ([第 7.3.2 节](#))，根据 `size` 是否为 1 输出“`time`”或“`times`”。

Section 10.6. Using Containers: Text-Query Program

The messiest part of the program is the `for` loop that processes `locs` to print the line numbers on which the word was found. The only subtlety here is remembering to change the line numbers into more human-friendly counting. When we stored the text, we stored the first line as line number zero, which is consistent with how C++ containers and arrays are numbered. Most users think of the first line as line number 1, so we systematically add one to our stored line numbers to convert to this more common notation.

这段程序最复杂的部分是处理 `locs` 对象的 `for` 循环，用于输出找到该单词的行号。其唯一的微妙之处是记得将行号修改为更友好的形式输出。为了与 C++ 的容器和数组下标编号匹配，在储存文本时，我们以行号 0 存储第一行。但考虑到很多用户会默认第一行的行号为 1，所以输出行号时，相应地所存储的行号上加 1 使之转换为更通用的形式。

Exercises Section 10.6.3

Exercise What is the output of `main` if we look for a word that is not found?
10.31: 如果没有找到要查询的单词，`main` 函数输出什么？

10.6.4. Writing the Member Functions

10.6.4. 编写成员函数

What remains is to write the definitions of the member functions that were not defined inside the class.

现在给没有在类内定义的成员函数编写定义。

Storing the Input File

存储输入文件

Our first task is to read the file that our user wishes to query. Using `string` and `vector` operations, this task is handled easily:

第一个任务是读入需要查询的文件。使用 `string` 和 `vector` 容器提供的操作，可以很简便地实现这个任务：

```
// read input file: store each line as element in lines_of_text
void TextQuery::store_file(ifstream &is)
{
    string textline;
    while (getline(is, textline))
        lines_of_text.push_back(textline);
}
```

Because we want to store the file a line at a time, we use `getline` to read our input. We push each line we read onto the `lines_of_text` vector.

由于我们希望每次存储文件的一行内容，因此使用 `getline` 读取输入，每读入一行就将它添加到名为 `lines_of_text` 的 `vector` 对象中。

Building the Word `map`

建立单词 `map` 容器

Each element in the `vector` holds a line of text. To build the `map` from words to line numbers, we need to break each line into its individual words. We again use an `istringstream` in ways outlined in the program on page 300:

`vector` 容器中的每个元素就是一行文本。要建立一个从单词关联到行号的 `map` 容器，必须将每行分解为各个单词。再次使用第 8.5 节描述的 `istringstream`：

```
// finds whitespace-separated words in the input vector
// and puts the word in word_map along with the line number
void TextQuery::build_map()
{
    // process each line from the input vector
    for (line_no line_num = 0;
         line_num != lines_of_text.size();
         ++line_num)
```

Section 10.6. Using Containers: Text-Query Program

```
{  
    //we'll use line to read the text a word at a time  
    istringstream line(lines_of_text[line_num]);  
    string word;  
    while (line >> word)  
        // add this line number to the set;  
        // subscript will add word to the map if it's not already there  
        word_map[word].insert(line_num);  
}  
}
```

The `for` loop marches through `lines_of_text` a line at a time. We start by binding an `istringstream` object named `line` to the current line and use the `istringstream` input operator to read each word on the line. Remember that that operator, like the other `istream` operators, ignores whitespace. Thus, the `while` reads each whitespace-separated word from `line` into `word`.

`for` 循环以每次一行的迭代过程遍历 `lines_of_text`。首先将 `istringstream` 对象 `line` 与当前行绑定起来，然后使用 `istringstream` 的输入操作符读入该行中的每个单词。回顾此类输入操作符，与其他 `istream` 操作符一样，将忽略空白符号。因此，`while` 循环将 `line` 中以空白符分隔的单词读取出来。

The last part of this function is similar to our word-count programs. We use `word` to subscript the `map`. If `word` was not already present, then the subscript operator adds `word` to the `word_map`, giving it an initial value that is the empty `set`. Regardless of whether `word` was added, the return value from the subscript is a `set`. We then call `insert` to add the current line number. If the word occurs more than once in the same line, then the call to `insert` does nothing.

这个函数的结尾部分类似前面的单词统计程序。将 `word` 用做 `map` 容器的下标。如果 `word` 在 `word_map` 容器对象中不存在，那么下标操作符将该 `word` 添加到此容器中，将其关联的值初始化为空的 `set`。不管是否添加了 `word`，下标运算都返回一个 `set` 对象，然后调用 `insert` 函数在该 `set` 对象中添加当前行号。如果某个单词在同一行中重复出现，那么 `insert` 函数的调用将不做任何操作。

Supporting Queries

支持查询

The `run_query` function handles the actual queries:

`run_query` 函数实现真正的查询功能：

```
set<TextQuery::line_no>  
TextQuery::run_query(const string &query_word) const  
{  
    //Note: must use find and not subscript the map directly  
    //to avoid adding words to word_map!  
    map<string, set<line_no>>::const_iterator  
        loc = word_map.find(query_word);  
    if (loc == word_map.end())  
        return set<line_no>(); // not found, return empty set  
    else  
        // fetch and return set of line numbers for this word  
        return loc->second;  
}
```

The `run_query` function takes a reference to a `const string` and uses that value to index the `word_map`. Assuming the `string` is found, it returns the `set` of line numbers associated with the `string`. Otherwise, it returns an empty `set`.

`run_query` 函数带有指向 `const string` 类型对象的引用参数，并以这个参数作为下标来访问 `word_map` 对象。假设成功找到这个 `string`，那么该函数返回关联此 `string` 的 `set` 对象，否则返回一个空的 `set` 对象。

Using the Return from `run_query`

`run_query` 返回值的使用

Once we've run the `run_query` function, we get back a set of line numbers on which the word we sought appears. In addition to printing how many times each word appears, we also want to print the line on which the word appeared. The `text_line` function lets us do so:

运行 `run_query` 函数后，将获得一组所查找的单词出现的行号。除了输出该单词的出现次数之外，还需要输出出现该单词的每一行。这就是 `text_line` 函数实现的功能：

```
string TextQuery::text_line(line_no line) const  
{  
    if (line < lines_of_text.size())  
        return lines_of_text[line];  
    throw std::out_of_range("line number out of range");  
}
```

This function takes a line number and returns the input text line corresponding to that line number. Because the code using our `TextQuery` class cannot do so `lines_of_text` is `private` we first check that the line we are asked for is in range. If it is, we return the corresponding line. If it is not, we `throw` an `out_of_range` exception.

该函数带有一个行号参数，返回该行号所对应的输入文本行。由于上述代码使用了 `TextQuery` 类，因此不能直接输出（因为 `lines_of_text` 是私有的），应该首先检查我们要查询的行是否位于合法范围内。如果是，则返回相应的行，否则，抛出 `out_of_range` 异常。

Exercises Section 10.6.4

Exercise 10.32: Reimplement the text-query program to use a `vector` instead of a `set` to hold the line numbers. Note that because lines appear in ascending order, we can append a new line number to the `vector` only if the last element of the `vector` isn't that line number. What are the performance and design characteristics of the two solutions? Which do you feel is the preferred design solution? Why?

重新实现文本查询程序，使用 `vector` 容器代替 `set` 对象来存储行号。注意，由于行以升序出现，因此只有在当前行号不是 `vector` 容器对象中的最后一个元素时，才能将新行号添加到 `vector` 中。这两种实现方法的性能特点和设计特点分别是什么？你觉得哪一种解决方法更好？为什么？

Exercise 10.33: Why doesn't the `TextQuery::text_line` function check whether its argument is negative?
`TextQuery::text_line` 函数为什么不检查它的参数是否为负数？

Chapter Summary

小结

The elements in an associative container are ordered and accessed by key. The associative containers support efficient lookup and retrieval of elements by key. The use of a key distinguishes them from the sequential containers, in which elements are accessed positionally.

关联容器的元素按键排序和访问。关联容器支持通过键高效地查找和读取元素。键的使用，使关联容器区别于顺序容器，顺序容器的元素是根据位置访问的。

The `map` and `mymap` types store elements that are keyvalue pairs. These types use the library `pair` class, defined in the `utility` header, to represent these pairs. Dereferencing a `map` or `mymap` iterator yields a value that is a `pair`. The `first` member of the `pair` is a `const` key, and the `second` member is a value associated with that key. The `set` and `multiset` types store keys. The `map` and `set` types allow only one element with a given key. The `mymap` and `multiset` types allow multiple elements with the same key.

`map` 和 `mymap` 类型存储的元素是键—值对。它们使用在 `utility` 头文件中定义的标准库 `pair` 类，来表示这些键—值对元素。对 `map` 或 `mymap` 迭代器进行解引用将获得 `pair` 类型的值。`pair` 对象的 `first` 成员是一个 `const` 键，而 `second` 成员则是该键所关联的值。`set` 和 `multiset` 类型则专门用于存储键。在 `map` 和 `set` 类型中，一个键只能关联一个元素。而 `mymap` 和 `multiset` 类型则允许多个元素拥有相同的键。

The associative containers share many operations with the sequential containers. However, the associative containers define some new operations and redefine the meaning or return types of some operations that are in common with the sequential containers. The differences in the operations reflect the use of keys in associative containers.

关联容器共享了顺序容器的许多操作。除此之外，关联容器还定义一些新操作，并对某些顺序容器同样提供的操作重新定义了其含义或返回类型，这些操作的差别体现了关联容器中键的使用。

Elements in an associative container can be accessed by iterators. The library guarantees that iterators access elements in order by key. The `begin` operation yields the element with the lowest key. Incrementing that iterator yields elements in nondescending order.

关联容器的元素可用迭代器访问。标准库保证迭代器按照键的次序访问元素。`begin` 操作将获得拥有最小键的元素，对此迭代器作自增运算则可以按非降序依次访问各个元素。

Defined Terms

术语

associative array (关联数组)

Array whose elements are indexed by key rather than positionally. We say that the array maps a key to its associated value.

由键而不是位置来索引元素的数组。通常描述为：此类数组将键映射到其关联的值上。

associative container (关联容器)

A type that holds a collection of objects that supports efficient lookup by key.

存储对象集合的类型，支持通过键的高效查询。

key type

Type defined by the associative containers that is the type for the keys used to store and retrieve values. For a `map`, `key_type` is the type used to index the `map`. For `set`, `key_type` and `value_type` are the same.

关联容器定义的类型，表示该容器在存储或读取值时所使用的键的类型。对于 `map` 容器，`key_type` 是用于索引该容器的类型。对于 `set` 容器，`key_type` 与 `value_type` 相同。

map

Associative container type that defines an associative array. Like `vector`, `map` is a class template. A `map`, however, is defined with two types: the type of the key and the type of the associated value. In a `map` a given key may appear only once. Each key is associated with a particular value. Dereferencing a `map` iterator yields a `pair` that holds a `const` key and its associated value.

定义关联数组的关联容器类型。与 `vector` 容器一样，`map` 也是类模板。但是，`map` 容器定义了两种类型：键类型及其关联的值类型。在 `map` 中，每个键只能出现一次，并关联某一具体的值。对 `map` 容器的迭代器进行解引用将获得一个 `pair` 对象，该对象存储了一个 `const` 键和它所关联的值。

mapped type

Type defined by `map` and `multimap` that is the type for the values stored in the `map`.

`map` 或 `multimap` 容器定义的类型，表示在 `map` 容器中存储的值的类型。

multimap

Associative container similar to `map` except that in a `multimap`, a given key may appear more than once.

类似 `map` 的关联容器。在 `multimap` 容器中，一个键可以出现多次。

multiset

Associative container type that holds keys. In a `multiset`, a given key may appear more than once.

只存储键的关联容器类型。在 `multiset` 容器中，一个键可以出现多次。

pair

Type that holds two `public` data members named `first` and `second`. The `pair` type is a template type that takes two type parameters that are used as the types of these members.

一种类型，有两个 `public` 数据成员，分别名为 `first` 和 `second`。`pair` 类型是带有两个类型形参的模板类型，它的类型形参用作数据成员的类型。

set

Associative container that holds keys. In a `set`, a given key may appear only once.

只存储键的关联容器。在 `set` 容器中，一个键只能出现一次。

strict weak ordering (严格弱排序)

Relationship among the keys used in an associative container. In a strict weak ordering, it is possible to compare any two values and determine which of the two is less than the other. If neither value is less than the other, then the two values are considered equal. See [Section 10.3.1 \(p. 360\)](#).

关联容器所使用的键之间的比较关系。在这种关系下，任意两个元素都可比较，并能确定两者之间谁比谁小。如果两个值都不比对方小，则这两个值相等，详见[第10.3.1节](#)。

value_type

The type of the element stored in a container. For `set` and `multiset`, `value_type` and `key_type` are the same. For `map` and `multimap`, this type is a `pair` whose `first` member has type `const key_type` and whose `second` member has type `mapped_type`.

存储在容器中的元素的类型。对于 `set` 和 `multiset` 容器，`value_type` 与 `key_type` 相同。而对于 `map` 和 `multimap` 容器，该类型为 `pair` 类型，它的 `first` 成员是 `const key_type` 类型，`second` 成员则是 `mapped_type` 类型。

* operator (解引用操作符)

The dereference operator when applied to a `map`, `set`, `multimap`, or `multiset` iterator yields a `value_type`. Note, that for `map` and `multimap` the `value_type` is a `pair`.

用于 `map`、`set`、`multimap` 或 `multiset` 迭代器时，解引用操作符将生成一个 `value_type` 类型的值。注意，对于 `map` 和 `multimap` 容器，`value_type` 是 `pair` 类型。

[] operator (下标操作符)

Subscript operator. When applied to a `map`, `[]` takes an index that must be a `key_type` (or type that can be converted to `key_type`) value. Yields a `mapped_type` value.

下标操作符。对 `map` 容器使用下标操作符时，`[]` 中的索引必须是 `key_type` 类型（或者是可以转换为 `key_type` 的类型）的值；该运算生成 `mapped_type` 类型的值。

Chapter 11. Generic Algorithms

第十一章 泛型算法

CONTENTS

<u>Section 11.1</u> Overview	392
<u>Section 11.2</u> A First Look at the Algorithms	395
<u>Section 11.3</u> Revisiting Iterators	405
<u>Section 11.4</u> Structure of Generic Algorithms	419
<u>Section 11.5</u> Container-Specific Algorithms	421
<u>Chapter Summary</u>	424
<u>Defined Terms</u>	424

The library containers define a surprisingly small set of operations. Rather than adding lots of functionality to the containers, the library instead provides a set of algorithms, most of which are independent of any particular container type. These algorithms are "generic:" They operate on different types of containers and on elements of various types.

标准库容器定义的操作非常少。标准库没有给容器添加大量的功能函数，而是选择提供一组算法，这些算法大都不依赖特定的容器类型，是“泛型”的，可作用在不同类型的容器和不同类型的元素上。

The generic algorithms, and a more detailed look at iterators, form the subject matter of this chapter.

泛型算法以及对迭代器更详尽的描述，组成了本章的主题。

The standard containers define few operations. For the most part they allow us to add and remove elements; to access the first or last element; to obtain and in some cases reset the size of the container; and to obtain iterators to the first and one past the last element.

标准容器（the standard container）定义了很多的操作。大部分容器都支持添加和删除元素；访问第一个和最后一个元素；获取容器的大小，并在某些情况下重设容器的大小；以及获取指向第一个元素和最后一个元素的下一位位置的迭代器。

We can imagine many other useful operations one might want to do on the elements of a container. We might want to sort a sequential container, or find a particular element, or find the largest or smallest element, and so on. Rather than define each of these operations as members of each container type, the standard library defines a set of [generic algorithms](#): "algorithms" because they implement common operations; and "generic" because they operate across multiple container types not only library types such as `vector` or `list`, but also the built-in array type, and, as we shall see, over other kinds of sequences as well. The algorithms also can be used on container types we might define ourselves, as long as our types conform to the standard library conventions.

可以想像，用户可能还希望对容器元素进行更多其他有用的操作：也许需要给顺序容器排序，或者查找某个特定的元素，或者查找最大或最小的元素，等等。标准库并没有为每种容器类型都定义实现这些操作的成员函数，而是定义了一组**泛型算法**：因为它们实现共同的操作，所以称之为“算法”；而“泛型”指的是它们可以操作在多种容器类型上——不但可作用于 `vector` 或 `list` 这些标准库类型，还可用在内置数组类型、甚至其他类型的序列上，这些我们将在本章的后续内容中了解。自定义的容器类型只要与标准库兼容，同样可以使用这些泛型算法。

Most of the algorithms operate by traversing a range of elements bounded by two iterators. Typically, as the algorithm traverses the elements in the range, it operates on each element. The algorithms obtain access to the elements through the iterators that denote the range of elements to traverse.

大多数算法是通过遍历由两个迭代器标记的一段元素来实现其功能。典型情况下，算法在遍历一段元素范围时，深灰其中的每一个元素。算法通过这两个迭代器访问元素，这两个迭代器标记了要遍历的元素范围。

11.1. Overview

11.1. 概述

Suppose we have a `vector` of `ints`, named `vec`, and we want to know if it holds a particular value. The easiest way to answer this question is to use the library `find` operation:

假设有一个 `int` 的 `vector` 对象，名为 `vec`，我们想知道其中包含某个特定值。解决这个问题最简单的方法是使用标准库提供的 `find` 运算：

```
// value we'll look for
int search_value = 42;

// call find to see if that value is present
vector<int>::const_iterator result =
    find(vec.begin(), vec.end(), search_value);

// report the result
cout << "The value " << search_value
    << (result == vec.end()
        ? " is not present" : " is present")
    << endl;
```

The call to `find` takes two iterators and a value. It tests each element in the range ([Section 9.2.1](#), p. 314) denoted by its iterator arguments. As soon as it sees an element that is equal to the given value, `find` returns an iterator referring to that element. If there is no match, then `find` returns its second iterator to indicate failure. We can test whether the return is equal to the second argument to determine whether the element was found. We do this test in the output statement, which uses the conditional operator ([Section 5.7](#), p. 165) to report whether the value was found.

使用两个迭代器和一个值调用 `find` 函数，检查两个迭代器实参标记范围内的每一个元素。只要找到与给定值相等的元素，`find` 就会返回指向该元素的迭代器。如果没有匹配的元素，`find` 就返回它的第二个迭代器实参，表示查找失败。于是，只要检查该函数的返回值是否与它的第二个实参相等，就可得知元素是否找到了。我们在输出语句中使用条件操作符（[第 5.7 节](#)）实现这个检查并报告是否找到了给定值。

Because `find` operates in terms of iterators, we can use the same `find` function to look for values in any container. For example, we can use `find` to look for a value in a `list` of `int` named `lst`:

由于 `find` 运算是基于迭代器的，因此可在任意容器中使用相同的 `find` 函数查找值。例如，可在名为 `lst` 的 `int` 型 `list` 对象上，使用 `find` 函数查找一个值：

```
// call find to look through elements in a list
list<int>::const_iterator result =
    find(lst.begin(), lst.end(), search_value);
cout << "The value " << search_value
    << (result == lst.end()
        ? " is not present" : " is present")
    << endl;
```

Except for the type of `result` and the iterators passed to `find`, this code is identical to the program that used `find` to look at elements of a `vector`.

除了 `result` 的类型和传递给 `find` 的迭代器类型之外，这段代码与使用 `find` 在 `vector` 对象中查找元素的程序完全相同。

Similarly, because pointers act like iterators on built-in arrays, we can use `find` to look in an array:

类似地，由于指针的行为与作用在内置数组上的迭代器一样，因此也可以使用 `find` 来搜索数组：

```
int ia[6] = {27, 210, 12, 47, 109, 83};
int search_value = 83;

int *result = find(ia, ia + 6, search_value);

cout << "The value " << search_value
    << (result == ia + 6
        ? " is not present" : " is present")
    << endl;
```

Here we pass a pointer to the first element in `ia` and another pointer that is six elements past the start of `ia` (that is, one past the last element of `ia`). If the pointer returned is equal to `ia + 6` then the search is unsuccessful; otherwise, the pointer points to the value that was found.

这里给 `find` 函数传递了两个指针：指向 `ia` 数组中第一个元素的指针，以及指向 `ia` 数组起始位置之后第 6 个元素的指针（即 `ia` 的最后一个元素的下一位置）。如果返回的指针等于 `ia + 6`，那么搜索不成功；否则，返回的指针指向找到值。

If we wish to pass a subrange, we pass iterators (or pointers) to the first and one past the last element of that subrange. For example, in this

Section 11.1. Overview

invocation of `find`, only the elements `ia[1]` and `ia[2]` are searched:

如果需要传递一个子区间，则传递指向这个子区间的第一个元素以及最后一个元素的下一位置的迭代器（或指针）。例如，在下面对 `find` 函数的调用中，只搜索了 `ia[1]` 和 `ia[2]`：

```
// only search elements ia[1] and ia[2]
int *result = find(ia + 1, ia + 3, search_value);
```

How the Algorithms Work

算法如何工作

Each generic algorithm is implemented independently of the individual container types. The algorithms are also largely, but not completely, independent of the types of the elements that the container holds. To see how the algorithms work, let's look a bit more closely at `find`. Its job is to find a particular element in an unsorted collection of elements. Conceptually the steps that `find` must take include:

每个泛型算法的实现都独立于单独的容器。这些算法还是大而不全的，并且不依赖于容器存储的元素类型。为了知道算法如何工作，让我们深入了解 `find` 操作。该操作的任务是在一个未排序的元素集合中查找特定的元素。从概念上看，`find` 必须包含以下步骤：

1. Examine each element in turn.

顺序检查每个元素。

2. If the element is equal to the value we want, then return an iterator that refers to that element.

如果当前元素等于要查找的值，那么返回指向该元素的迭代器。

3. Otherwise, examine the next element, repeating step 2 until either the value is found or all the elements have been examined.

否则，检查下一个元素，重复步骤 2，直到找到这个值，或者检查完所有的元素为止。

4. If we have reached the end of the collection and we have not found the value, return a value that indicates that the value was not found.

如果已经到达集合末尾，而且还未找到该值，则返回某个值，指明要查找的值在这个集合中不存在。

The Standard Algorithms Are Inherently Type-Independent

标准算法固有地独立于类型

The algorithm, as we've stated it, is independent of the type of the container: Nothing in our description depends on the container type. Implicitly, the algorithm does have one dependency on the element type: We must be able to compare elements. More specifically, the requirements of the algorithm are:

这种算法，正如我们所指出的，与容器的类型无关：在前面的描述中，没有任何内容依赖于容器类型。这种算法只在一点上隐式地依赖元素类型：必须能够对元素做比较运算。该算法的明确要求如下：

1. We need a way to traverse the collection: We need to be able to advance from one element to the next.

需要某种遍历集合的方式：能够从一个元素向前移到下一个元素。

2. We need to be able to know when we have reached the end of the collection.

必须能够知道是否到达了集合的末尾。

3. We need to be able to compare each element to the value we want.

必须能够对容器中的每一个元素与被查找的元素进行比较。

4. We need a type that can refer to an element's position within the container or that can indicate that the element was not found.

需要一个类型指出元素在容器中的位置，或者表示找不到该元素。

Iterators Bind Algorithms to Containers

迭代器将算法和容器绑定起来

The generic algorithms handle the first requirement, container traversal, by using iterators. All iterators support the increment operator to navigate from one element to the next, and the dereference operator to access the element value. With one exception that we'll cover in [Section 11.3.5 \(p. 416\)](#), the iterators also support the equality and inequality operators to determine whether two iterators are equal.

泛型算法用迭代器来解决第一个要求：遍历容器。所有迭代器都支持自增操作符，从一个元素定位下一个元素，并提供解引用操作符访问元素的值。除了 [第 11.3.5 节](#) 将介绍的一个例外情况之外，迭代器还支持相等和不等操作符，用于判断两个迭代器是否相等。

For the most part, the algorithms each take (at least) two iterators that denote the range of elements on which the algorithm is to operate. The first iterator refers to the first element, and the second iterator marks one past the last element. The element addressed by the second iterator, sometimes referred to as the [off-the-end iterator](#), is not itself examined; it serves as a sentinel to terminate the traversal.

大多数情况下，每个算法都需要使用（至少）两个迭代器指出该算法操纵的元素范围。第一个迭代器指向第一个元素，而第二个迭代器则指向最后一个元素的下一位置。第二个迭代器所指向的元素 [有时被称为[超出末端迭代器](#)] 本身不是要操作的元素，而被用作终止遍历的哨兵（sentinel）。

The off-the-end iterator also handles requirement 4 by providing a convenient return value that indicates that the search element wasn't found. If the value isn't found, then the off-the-end iterator is returned; otherwise, the iterator that refers to the matching element is returned.

使用超出末端迭代器还可以很方便地处理第四个要求，只要以此迭代器为返回值，即可表示没有找到要查找的元素。如果要查找的值未找到，则返回超出末端迭代器；否则，返回的迭代器指向匹配的元素。

Requirement 3, value comparison, is handled in one of two ways. By default, the `find` operation requires that the element type define operator `==`. The algorithm uses that operator to compare elements. If our type does not support the `==` operator, or if we wish to compare elements using a different test, we can use a second version of `find`. That version takes an extra argument that is the name of a function to use to compare the elements.

第三个要求——元素值的比较，有两种解决方法。默认情况下，`find` 操作要元素类型定义了相等（`==`）操作符，算法使用这个操作符比较元素。如果元素类型不支持相等（`==`）操作符，或者打算用不同的测试方法来比较元素，则可使用第二个版本的 `find` 函数。这个版本需要一个额外的参数：实现元素比较的函数名字。

The algorithms achieve type independence by never using container operations; rather, all access to and traversal of the elements is done through the iterators. The actual container type (or even whether the elements are stored in a container) is unknown.

这些算法从不使用容器操作，因而其实现与类型无关，元素的所有访问和遍历都通过迭代器实现。实际的容器类型未知（甚至所处理的元素是否存储在容器中也是未知的）。

The library provides more than 100 algorithms. Like the containers, the algorithms have a consistent architecture. Understanding the design of the algorithms makes learning and using them easier than memorizing all 100+ algorithms. In this chapter we'll both illustrate the use of the algorithms and describe the unifying principles used by the library algorithms. [Appendix A](#) lists all the algorithms classified by how they operate.

标准库提供了超过 100 种算法。与容器一样，算法有着一致的结构。比起死记全部一百多种算法，了解算法的设计可使我们更容易学习和使用它们。本章除了举例说明这些算法的使用之外，还将描述标准库算法的统一原理。[附录 A](#)根据操作分类列出了所有的算法。

Exercises Section 11.1

Exercise 11.1: The `algorithm` header defines a function named `count` that is similar to `find`. It takes a pair of iterators and a value and returns a count of how often that value appears. Read a sequence of `ints` into a `vector`. Count how many elements have a given value.

`algorithm` 头文件定义了一个名为 `count` 的函数，其功能类似于 `find`。这个函数使用一对迭代器和一个值做参数，返回这个值出现次数的统计结果。编写程序读取一系列 `int` 型数据，并将它们存储到 `vector` 对象中，然后统计某个指定的值出现了多少次。

Exercise 11.2: Repeat the previous program, but read values into a `list` of `strings`.

重复前面的程序，但是，将读入的值存储到一个 `string` 类型的 `list` 对象中。

Key Concept: Algorithms **Never** Execute Container Operations

关键概念：算法永不执行容器提供的操作

The generic algorithms do not themselves execute container operations. They operate solely in terms of iterators and iterator operations. The fact that the algorithms operate in terms of iterators and not container operations has a perhaps surprising but essential implication: When used on "ordinary" iterators, algorithms never change the size of the underlying container. As we'll see, algorithms may change the values of the elements stored in the container, and they may move elements around within the container. They do not, however, ever add or remove elements directly.

泛型算法本身从不执行容器操作，只是单独依赖迭代器和迭代器操作实现。算法基于迭代器及其操作实现，而并非基于容器操作。这个事实也许比较意外，但本质上暗示了：使用“普通”的迭代器时，算法从不修改基础容器的大小。正如我们所看到的，算法也许会改变存储在容器中的元素的值，也许会在容器内移动元素，但是，算法从不直接添加或删除元素。

[第 11.3.1 节](#)将介绍标准库提供的另一种特殊的迭代器类：插入器（**inserter**），除了用于遍历其所绑定的序列之外，还可实现更多的功能。在给这类迭代器赋值时，在基础容器上将执行插入运算。如果算法操纵这类迭代器，迭代器将可能导致在容器中添加元素。但是，算法本身从不这么做。

Team LiB

◀ PREVIOUS NEXT ▶

11.2. A First Look at the Algorithms

11.2. 初窥算法

Before covering the structure of the algorithms library, let's look at a couple of examples. We've already seen the use of `find`; in this section we'll use a few additional algorithms. To use a generic algorithm, we must include the `algorithm` header:

在研究算法标准库的结构之前，先看一些例子。上一节已经介绍了 `find` 函数的用法；本节将要使用其他的一些算法。使用泛型算法必须包含 `algorithm` 头文件：

```
#include <algorithm>
```

The library also defines a set of generalized numeric algorithms, using the same conventions as the generic algorithms. To use these algorithms we include the `numeric` header:

标准库还定义了一组泛化的算术算法（generalized numeric algorithm），其命名习惯与泛型算法相同。使用这些算法则必须包含 `numeric` 头文件：

```
#include <numeric>
```

With only a handful of exceptions, all the algorithms operate over a range of elements. We'll refer to this range as the "input range." The algorithms that take an input range always use their first two parameters to denote that range. These parameters are iterators used to denote the first and one past the last element that we want to process.

除了少数例外情况，所有算法都在一段范围内的元素上操作，我们将这段范围称为“输出范围（input range）”。带有输入范围参数的算法总是使用头两个形参标记该范围。这两个形参是分别指向要处理的第一个元素和最后一个元素的下一位置的迭代器。

Although most algorithms are similar in that they operate over an input range, they differ in how they use the elements in that range. The most basic way to understand the algorithms is to know whether they read elements, write elements, or rearrange the order of the elements. We'll look at samples of each kind of algorithm in the remainder of this section.

尽管大多数算法对算法对输入范围的操作是类似的，但在该范围内如何操纵元素却有所不同。理解算法的最基本方法是了解该算法是否读元素、写元素或者对元素进行重新排序。在本节的余下内容中，将会观察到每种算法的例子。

11.2.1. Read-Only Algorithms

11.2.1. 只读算法

A number of the algorithms read, but never write to, the elements in their input range. `find` is one such algorithm. Another simple, read-only algorithm is `accumulate`, which is defined in the `numeric` header. Assuming `vec` is a `vector` of `int` values, the following code

许多算法只会读取其输入范围内的元素，而不会写这些元素。`find` 就是一个这样的算法。另一个简单的只读算法是 `accumulate`，该算法在 `numeric` 头文件中定义。假设 `vec` 是一个 `int` 型的 `vector` 对象，下面的代码：

```
// sum the elements in vec starting the summation with the value 42
int sum = accumulate(vec.begin(), vec.end(), 42);
```

sets `sum` equal to the sum of the elements in `vec` plus 42. `accumulate` takes three parameters. The first two specify a range of elements to sum. The third is an initial value for the sum. The `accumulate` function sets an internal variable to the initial value. It then adds the value of each element in the range to that initial value. The algorithm returns the result of the summation. The return type of `accumulate` is the type of its third argument.

将 `sum` 设置为 `vec` 的元素之和再加上 42。`accumulate` 带有三个形参。头两个形参指定要累加的元素范围。第三个形参则是累加的初值。`accumulate` 函数将它的一个内部变量设置为指定的初值，然后在此初值上累加输入范围 `accumulate`



The third argument, which specifies the starting value, is necessary because `accumulate` knows nothing about the element types that it is accumulating. Therefore, it has no way to invent an appropriate starting value or associated type.

用于指定累加起始值的第三个实参是必要的，因为 `accumulate` 对将要累加的元素类型一无所知，因此，除此之外，没有别的办法创建合适的起始值或者关联的类型。

There are two implications of the fact that `accumulate` doesn't know about the types over which it sums. First, we must pass an initial starting value because otherwise `accumulate` cannot know what starting value to use. Second, the type of the elements in the container must match or be convertible to the type of the third argument. Inside `accumulate`, the third argument is used as the starting point for the summation; the elements in the container are successively added into this sum. It must be possible to add the element type to the type of the sum.

`accumulate` 对要累加的元素类型一无所知，这个事实有两层含义。首先，调用该函数时必须传递一个起始值，否则，`accumulate` 将不知道使用什么起始值。其次，容器内的

Section 11.2. A First Look at the Algorithms

元素类型必须与第三个实参的类型匹配，或者可转换为第三个实参的类型。在 `accumulate` 内部，第三个实参用作累加的起点；容器内的元素按顺序连续累加到总和之中。因此，必须能够将元素类型加到总和类型上。

As an example, we could use `accumulate` to concatenate the elements of a `vector` of `strings`:

考虑下面的例子，可以使用 `accumulate` 把 `string` 型的 `vector` 容器中的元素连接起来：

```
// concatenate elements from v and store in sum
string sum = accumulate(v.begin(), v.end(), string("") );
```

The effect of this call is to concatenate each element in `vec` onto a `string` that starts out as the empty string. Note that we explicitly create a `string` as the third parameter. Passing a character-string literal would be a compile-time error. If we passed a `string` literal, the summation type would be `const char*` but the `string` addition operator ([Section 3.2.3](#), p. 86) for operands of type `string` and `const char*` yields a `string` not a `const char*`.

这个函数调用的效果是：从空字符串开始，把 `vec` 里的每个元素连接成一个字符串。注意：程序显式地创建了一个 `string` 对象，用该函数调用的第三个实参。传递一个字符串面值，将会导致编译时错误。因为此时，累加和的类型将是 `const char*`，而 `string` 的加法操作符（[第 3.2.3 节](#)）所使用的操作数则分别是 `string` 和 `const char*` 类型，加法的结果将产生一个 `string` 对象，而不是 `const char*` 指针。

Using `find_first_of`

`find_first_of` 的使用

In addition to `find`, the library defines several other, more complicated searching algorithms. Several of these are similar to the `find` operations of the `string` class ([Section 9.6.4](#), p. 343). One such is `find_first_of`. This algorithm takes two pairs of iterators that denote two ranges of elements. It looks in the first range for a match to any element from the second range and returns an iterator denoting the first element that matches. If no match is found, it returns the end iterator of the first range. Assuming that `roster1` and `roster2` are two `lists` of names, we could use `find_first_of` to count how many names are in both lists:

除了 `find` 之外，标准库还定义了其他一些更复杂的查找算法。当中的一部分类似 `string` 类的 `find` 操作（[第 9.6.4 节](#)），其中一个是 `find_first_of` 函数。这个算法带有两对迭代器参数来标记两段元素范围，在第一段范围内查找与第二段范围内任意元素匹配的元素，然后返回一个迭代器，指向第一个匹配的元素。如果找不到元素，则返回第一个范围的 `end` 迭代器。假设 `roster1` 和 `roster2` 是两个存放名字的 `list` 对象，可使用 `find_first_of` 统计有多少个名字同时出现在这两个列表中：

```
// program for illustration purposes only:
// there are much faster ways to solve this problem
size_t cnt = 0;
list<string>::iterator it = roster1.begin();
// look in roster1 for any name also in roster2
while ((it = find_first_of(it, roster1.end(),
                           roster2.begin(), roster2.end()))
       != roster1.end()) {
    ++cnt;
    // we got a match, increment it to look in the rest of roster1
    ++it;
}
cout << "Found " << cnt
    << " names on both rosters" << endl;
```

The call to `find_first_of` looks for any element in `roster2` that matches an element from the first range—that is, it looks for an element in the range from `it` to `roster1.end()`. The function returns the first element in that range that is also in the second range. On the first trip through the `while`, we look in the entire range of `roster1`. On second and subsequent trips, we look in that part of `roster1` that has not already been matched.

调用 `find_first_of` 查找 `roster2` 中的每个元素是否与第一个范围内的元素匹配，也就是在 `it` 到 `roster1.end()` 范围内查找一个元素。该函数返回此范围内第一个同时存在于第二个范围中的元素。在 `while` 的第一次循环中，遍历整个 `roster1` 范围。第二次以及后续的循环迭代则只考虑 `roster1` 中尚未匹配的部分。

In the condition, we check the return from `find_first_of` to see whether we found a matching name. If we got a match, we increment our counter. We also increment `it` so that it refers to the next element in `roster1`. We know we're done when `find_first_of` returns `roster1.end()`, which it does if there is no match.

循环条件检查 `find_first_of` 的返回值，判断是否找到匹配的名字。如果找到一个匹配，则使计数器加 1，同时给 `it` 加 1，使它指向 `roster1` 中的下一个元素。很明显可知，当不再有任何匹配时，`find_first_of` 返回 `roster1.end()`，完成统计。

Key Concept: Iterator Argument Types

关键概念：迭代器实参类型

In general, the generic algorithms operate on iterator pairs that denote a range of elements in a container (or other sequence). The types of the arguments that denote the range must match exactly, and the iterators themselves must denote a range: They must refer to elements in the same container (or to the element just past the end of that container), and if they are unequal, then it must be possible to reach the second iterator by repeatedly incrementing the first iterator.

通常，泛型算法都是在标记容器（或其他序列）内的元素范围的迭代器上操作的。标记范围的两个实参类型必须精确匹配，而迭代器本身必须标记一个范

Section 11.2. A First Look at the Algorithms

围：它们必须指向同一个容器中的元素（或者超出容器末端的下一位置），并且如果两者不相等，则第一个迭代器通过不断地自增，必须可以到达第二个迭代器。

Some algorithms, such as `find_first_of`, take two pairs of iterators. The iterator types in each pair must match exactly, but there is no requirement that the type of the two pairs match each other. In particular, the elements can be stored in different kinds of sequences. What is required is that we be able to compare elements from the two sequences.

有些算法，例如 `find_first_of`，带有两对迭代器参数。每对迭代器中，两个实参的类型必须精确匹配，但不要求两对之间的类型匹配。特别是，元素可存储在不同类型序列中，只要这两序列的元素可以比较即可。

In our program, the types of `roster1` and `roster2` need not match exactly: `roster1` could be a `list` while `roster2` was a `vector`, `deque`, or other sequence that we'll learn about later in this chapter. What is required is that we be able to compare the elements from these two sequences using the `==` operator. If `roster1` is a `list<string>`, then `roster2` could be a `vector<char*>` because the `string` library defines `==` on a `string` and a `char*`.

在上述程序中，`roster1` 和 `roster2` 的类型不必精确匹配：`roster1` 可以是 `list` 对象，而 `roster2` 则可以是 `vector` 对象、`deque` 对象或者是其他后面要学到的序列。只要这两个序列的元素可使用相等（`==`）操作符进行比较即可。如果 `roster1` 是 `list<string>` 对象，则 `roster2` 可以是 `vector<char*>` 对象，因为 `string` 标准库为 `string` 对象与 `char*` 对象定义了相等（`==`）操作符。

Exercises Section 11.2.1

Exercise Use `accumulate` to sum the elements in a `vector<int>`.

11.3: 用 `accumulate` 统计 `vector<int>` 容器对象中的元素之和

Exercise Assuming `v` is a `vector<double>` what, if anything, is wrong with calling `accumulate<v.begin(), v.end(), 0>`?

11.4: 假定 `v` 是 `vector<double>` 类型的对象，则调用 `accumulate<v.begin(), v.end(), 0>` 是否有错？如果有的话，错在哪里？

Exercise What would happen if the program that called `find_first_of` did not increment `it`?

11.5: 对于本节调用 `find_first_of` 的例程，如果不给 `it` 加 1，将会如何？

11.2.2. Algorithms that Write Container Elements

11.2.2. 写容器元素的算法

Some algorithms write element values. When using algorithms that write elements, we must take care to ensure that the sequence into which the algorithm writes is at least as large as the number of elements being written.

一些算法写入元素值。在使用这些算法写元素时要当心，必须确保算法所写的序列至少足以存储要写入的元素。

Some algorithms write directly into the input sequence. Others take an additional iterator that denotes a destination. Such algorithms use the destination iterator as a place in which to write output. Still a third kind writes a specified number of elements to some sequence.

有些算法直接将数据写入到输入序列，另外一些则带有一个额外的迭代器参数指定写入目标。这类算法将目标迭代器用作输出的位置。还有第三种算法将指定数目的元素写入某个序列。

Writing to the Elements of the Input Sequence

写入输入序列的元素

The algorithms that write to their input sequence are inherently safe—they write only as many elements as are in the specified input range.

写入到输入序列的算法本质上是安全的——只会写入与指定输入范围数量相同的元素。

A simple example of an algorithm that writes to its input sequence is `fill`:

写入到输入序列的一个简单算法是 `fill` 函数，考虑如下例子：

```
fill(vec.begin(), vec.end(), 0); // reset each element to 0  
// set subsequence of the range to 10
```

Section 11.2. A First Look at the Algorithms

```
fill(vec.begin(), vec.begin() + vec.size()/2, 10);
```

`fill` takes a pair of iterators that denote a range in which to write copies of its third parameter. It executes by setting each element in the range to the given value. Assuming that the input range is valid, then the writes are safe. The algorithm writes only to elements known to exist in the input range itself.

`fill` 带有一对迭代器形参，用于指定要写入的范围，而所写的值是它的第三个形参的副本。执行时，将该范围内的每个元素都设为给定的值。如果输入范围有效，则可安全写入。这个算法只会对输入范围内已存在的元素进行写入操作。

Algorithms Do Not Check Write Operations

不检查写入操作的算法

The `fill_n` function takes an iterator, a count, and a value. It writes the specified number of elements with the given value starting at the element referred to by the iterator. The `fill_n` function assumes that it is safe to write the specified number of elements. It is a fairly common beginner mistake to call `fill_n` (or similar algorithms that write to elements) on a container that has no elements:

`fill_n` 函数带有的参数包括：一个迭代器、一个计数器以及一个值。该函数从迭代器指向的元素开始，将指定数量的元素设置为给定的值。`fill_n` 函数假定对指定数量的元素做写操作是安全的。初学者常犯的错误的是：在没有元素的空容器上调用 `fill_n` 函数（或者类似的写元素算法）。

```
vector<int> vec; // empty vector
// disaster: attempts to write to 10 (nonexistent) elements in vec
fill_n(vec.begin(), 10, 0);
```

This call to `fill_n` is a disaster. We specified that ten elements should be written, but there are no such elements `vec` is empty. The result is undefined and will probably result in a serious run-time failure.

这个 `fill_n` 函数的调用将带来灾难性的后果。我们指定要写入 10 个元素，但这些元素却不存在——`vec` 是空的。其结果未定义，很可能导致严重的运行时错误。



Algorithms that write a specified number of elements or that write to a destination iterator do not check whether the destination is large enough to hold the number of elements being written.

对指定数目的元素做写入运算，或者写到目标迭代器的算法，都不检查目标的大小是否足以存储要写入的元素。

Introducing `back_inserter`

引入 `back_inserter`

One way to ensure that an algorithm has enough elements to hold the output is to use an `insert iterator`. An insert iterator is an iterator that adds elements to the underlying container. Ordinarily, when we assign to a container element through an iterator, we assign to the element to which the iterator refers. When we assign through an insert iterator, a new element equal to the right-hand value is added to the container.

确保算法有足够的元素存储输出数据的一种方法是使用 `插入迭代器`。插入迭代器是可以给基础容器添加元素的迭代器。通常，用迭代器给容器元素赋值时，被赋值的是迭代器所指向的元素。而使用插入迭代器赋值时，则会在容器中添加一个新元素，其值等于赋值运算的右操作数的值。

We'll have more to say about insert iterators in [Section 11.3.1](#) (p. 406). However, in order to illustrate how to safely use algorithms that write to a container, we will use `back_inserter`. Programs that use `back_inserter` must include the `iterator` header.

[第 11.3.1 节](#) 将会讨论更多关于插入迭代器的内容。然而，为了说明如何安全使用写容器的算法，下面将使用 `back_inserter`。使用 `back_inserter` 的程序必须包含 `iterator` 头文件。

The `back_inserter` function is an iterator adaptor. Like the container adaptors ([Section 9.7](#), p. 348), an iterator adaptor takes an object and generates a new object that adapts the behavior of its argument. In this case, the argument to `back_inserter` is a reference to a container. `back_inserter` generates an insert iterator bound to that container. When we attempt to assign to an element through that iterator, the assignment calls `push_back` to add an element with the given value to the container. We can use `back_inserter` to generate an iterator to use as the destination in `fill_n`:

`back_inserter` 函数是迭代器适配器。与容器适配器（[第 9.7 节](#)）一样，迭代器适配器使用一个对象作为实参，并生成一个适应其实参行为的新对象。在本例中，传递给 `back_inserter` 的实参是一个容器的引用。`back_inserter` 生成一个绑定在该容器上的插入迭代器。在试图通过这个迭代器给元素赋值时，赋值运算将调用 `push_back` 在容器中添加一个具有指定值的元素。使用 `back_inserter` 可以生成一个指向 `fill_n` 写入目标的迭代器：

```
vector<int> vec; // empty vector
// ok: back_inserter creates an insert iterator that adds elements to vec
fill_n (back_inserter(vec), 10, 0); // appends 10 elements to vec
```

Section 11.2. A First Look at the Algorithms

Now, each time `fill_n` writes a value, it will do so through the insert iterator generated by `back_inserter`. The effect will be to call `push_back` on `vec`, adding ten elements to the end of `vec`, each of which has the value 0.

现在, `fill_n` 函数每写入一个值, 都会通过 `back_inserter` 生成的插入迭代器实现。效果相当于在 `vec` 上调用 `push_back`, 在 `vec` 末尾添加 10 个元素, 每个元素的值都是 0。

Algorithms that Write to a Destination Iterator

写入到目标迭代器的算法

A third kind of algorithm writes an unknown number of elements to a destination iterator. As with `fill_n`, the destination iterator refers to the first element of a sequence that will hold the output. The simplest such algorithm is `copy`. This algorithm takes three iterators: The first two denote an input range and the third refers to an element in the destination sequence. It is essential that the destination passed to `copy` be at least as large as the input range. Assuming `ilst` is a `list` holding `ints`, we might `copy` it into a `vector`:

第三类算法向目标迭代器写入未知个数的元素。正如 `fill_n` 函数一样, 目标迭代器指向存放输出数据的序列中第一个元素。这类算法中最简单的是 `copy` 函数。`copy` 带有三个迭代器参数: 头两个指定输入范围, 第三个则指向目标序列的一个元素。传递给 `copy` 的目标序列必须至少要与输入范围一样大。假设 `ilst` 是一个存放 `int` 型数据的 `list` 对象, 可如下将它 `copy` 给一个 `vector` 对象:

```
vector<int> ivec; // empty vector
// copy elements from ilst into ivec
copy (ilst.begin(), ilst.end(), back_inserter(ivec));
```

`copy` reads elements from the input range, copying them to the destination.

`copy` 从输入范围中读取元素, 然后将它们复制给目标 `ivec`。

Of course, this example is a bit inefficient: Ordinarily if we want to create a new container as a copy of an existing container, it is better to use an input range directly as the initializer for a newly constructed container:

当然, 这个例子的效率比较差: 通常, 如果要以一个已存在的容器为副本创建新容器, 更好的方法是直接用输入范围作为新构造容器的初始化式:

```
// better way to copy elements from ilst
vector<int> ivec(ilst.begin(), ilst.end());
```

Algorithm `_copy` Versions

算法的 `_copy` 版本

Several algorithms provide so-called "copying" versions. These algorithms do some processing on the elements of their input sequence but do not change the original elements. Instead, a new sequence is written that contains the result of processing the elements of the original.

有些算法提供所谓的“复制 (copying) ”版本。这些算法对输入序列的元素做出处理, 但不修改原来的元素, 而是创建一个新序列存储元素的处理结果。但不修改原来的元素, 而是创建一个新序列存储元素的处理结果。

The `replace` algorithm is a good example. This algorithm reads and writes to an input sequence, replacing a given value by a new value. The algorithm takes four parameters: a pair of iterators denoting the input range and a pair of values. It substitutes the second value for each element that is equal the first:

`replace` 算法就是一个很好的例子。该算法对输入序列做读写操作, 将序列中特定的值替换为新的值。该算法带有四个形参: 一对指定输入范围的迭代器和两个值。每一个等于第一值的元素替换成第二个值。

```
// replace any element with value of 0 by 42
replace(ilst.begin(), ilst.end(), 0, 42);
```

This call replaces all instances of 0 by the 42. If we wanted to leave the original sequence unchanged, we would call `replace_copy`. That algorithm takes a third iterator argument denoting a destination in which to write the adjusted sequence:

这个调用将所有值为 0 的实例替换成 42。如果不改变原来的序列, 则调用 `replace_copy`。这个算法接受第三个迭代器实参, 指定保存调整后序列的目标位置。

```
// create empty vector to hold the replacement
vector<int> ivec;
// use back_inserter to grow destination as needed
replace_copy (ilst.begin(), ilst.end(),
             back_inserter(ivec), 0, 42);
```

After this call, `ilst` is unchanged, and `ivec` contains a copy of `ilst` with the exception that every element in `ilst` with the value 0 has the value 42 in `ivec`.

调用该函数后, `ilst` 没有改变, `ivec` 存储 `ilst` 一份副本, 而 `ilst` 内所有的 0 在 `ivec` 中都变成了 42。

Exercises Section 11.2.2

Exercise

11.6: Using `fill_n`, write a program to set a sequence of `int` values to 0.

使用 `fill_n` 编写程序，将一个 `int` 序列的值设为 0。

Exercise

11.7: Determine if there are any errors in the following programs and, if so, correct the error(s):

判断下面的程序是否有错，如果有，请改正之：

```
(a) vector<int> vec; list<int> lst; int i;
    while (cin >> i)
        lst.push_back(i);
    copy(lst.begin(), lst.end(), vec.begin());
```

```
(b) vector<int> vec;
    vec.reserve(10);
    fill_n(vec.begin(), 10, 0);
```

Exercise

11.8: We said that algorithms do not change the size of the containers over which they operate. Why doesn't the use of `back_inserter` invalidate this claim?

前面说过，算法不改变它所操纵的容器的大小，为什么使用 `back_inserter` 也不能突破这个限制？

11.2.3. Algorithms that Reorder Container Elements

11.2.3. 对容器元素重新排序的算法

Suppose we want to analyze the words used in a set of children's stories. For example, we might want know how many words contain six or more characters. We want to count each word only once, regardless of how many times it appears or whether it appears in multiple stories. We'd like to be able to print the words in size order, and we want the words to be in alphabetic order within a given size.

假设我们要分析一组儿童故事中所使用的单词。例如，可能想知道它们使用了多少个由六个或以上字母组成的单词。每个单词只统计一次，不考虑它出现的次数，也不考虑它是否在多个故事中出现。要求以长度的大小输出这些单词，对于同样长的单词，则以字典顺序输出。

We'll assume that we have read our input and stored the text of each book in a `vector` of `strings` named `words`. How might we solve the part of the problem that involves counting word occurrences? To solve this problem, we'd need to:

1. Eliminate duplicate copies of each word

去掉所有重复的单词。

2. Order the words based on size

按单词的长度排序。

3. Count the words whose size is 6 or greater

统计长度等于或超过 6 个字符的单词个数。

We can use generic algorithms in each of these steps.

上述每一步都可使用泛型算法实现。

For purposes of illustration, we'll use the following simple story as our input:

为了说清楚，使用下面这个简单的故事作为我们的输入：

```
the quick red fox jumps over the slow red turtle
```

Given this input, our program should produce the following output:

对于这个输入，我们的程序应该产生如下输出：

```
1 word 6 characters or longer
```

Eliminating Duplicates

去除重复

Assuming our input is in a `vector` named `words`, our first subproblem is to eliminate duplicates from the `words`:

假设我们的输入存储在一个名为 `words` 的 `vector` 对象中，第一个子问题是将 `words` 中重复出现的单词去除掉：

```
// sort words alphabetically so we can find the duplicates
sort(words.begin(), words.end());
/* eliminate duplicate words:
 * unique reorders words so that each word appears once in the
 * front portion of words and returns an iterator one past the unique range;
 * erase uses a vector operation to remove the nonunique elements
 */
vector<string>::iterator end_unique =
    unique(words.begin(), words.end());
words.erase(end_unique, words.end());
```

Our input `vector` contains a copy of every word used in each story. We start by sorting this `vector`. The `sort` algorithm takes two iterators that denote the range of elements to sort. It uses the `<` operator to compare the elements. In this call we ask that the entire `vector` be sorted.

`vector` 对象包含每个故事中使用的所有单词。首先对此 `vector` 对象排序。`sort` 算法带有两个迭代器实参，指出要排序的元素范围。这个算法使用小于 (`<`) 操作符比较元素。在本次调用中，要求对整个 `vector` 对象排序。

After the call to `sort`, our `vector` elements are ordered:

调用 `sort` 后，此 `vector` 对象的元素按次序排列：

```
fox jumps over quick red red slow the the turtle
```

Note that the words `red` and `the` are duplicated.

注意，单词 `red` 和 `the` 重复出现了。

Using `unique`

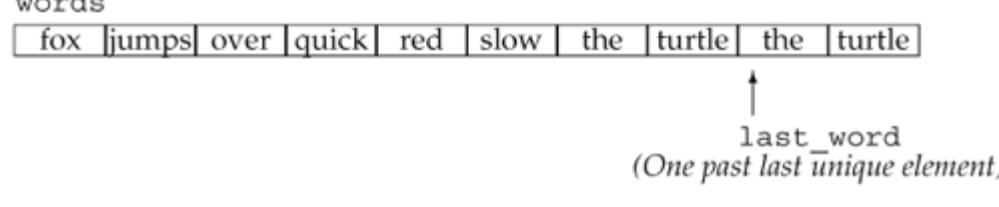
`unique` 的使用

Once `words` is sorted, our problem is to keep only one copy of each word that is used in our stories. The `unique` algorithm is well suited to this problem. It takes two iterators that denote a range of elements. It rearranges the elements in the input range so that adjacent duplicated entries are eliminated and returns an iterator that denotes the end of the range of the unique values.

单词按次序排列后，现在的问题是：让故事中所用到的每个单词都只保留一个副本。`unique` 算法很适合用于解决这个问题，它带有两个指定元素范围的迭代器参数。该算法删除相邻的重复元素，然后重新排列输入范围内的元素，并且返回一个迭代器，表示无重复的值范围的结束。

After the call to `unique`, the `vector` holds

调用 `unique` 后，`vector` 中存储内容是：



Note that the size of `words` is unchanged. It still has ten elements; only the order of these elements has changed. The call to `unique` "removes" adjacent duplicates. We put `remove` in quotes because `unique` doesn't remove any elements. Instead, it overwrites adjacent duplicates so that the unique elements are copied into the front of the sequence. The iterator returned by `unique` denotes one past the end of the range of unique elements.

注意，`words` 的大小并没有改变，依然保存着 10 个元素；只是这些元素的顺序改变了。调用 `unique` “删除”了相邻的重复值。给“删除”加上引号是因为 `unique` 实际上并没有删除任何元素，而是将无重复的元素复制到序列的前端，从而覆盖相邻的重复元素。`unique` 返回的迭代器指向超出无重复的元素范围末端的下一位置。

Using Container Operations to Remove Elements

使用容器操作删除元素

If we want to eliminate the duplicate items, we must use a container operation, which we do in the call to `erase`. This call erases the elements starting with the one to which `end unique` refers through the end of `words`. After this call, `words` contains the eight unique words from the input.

Section 11.2. A First Look at the Algorithms

如果要删除重复的项，必须使用容器操作，在本例中调用 `erase` 实现该功能。这个函数调用从 `end_unique` 指向的元素开始删除，直到 `words` 的最后一个元素也删除掉为止。调用之后，`words` 存储输入的 8 个不相同的元素。



Algorithms never directly change the size of a container. If we want to add or remove elements, we must use a container operation.

算法不直接修改容器的大小。如果需要添加或删除元素，则必须使用容器操作。

It is worth noting that this call to `erase` would be safe even if there were no duplicated words in our `vector`. If there were no duplicates, then `unique` would return `words.end()`. Both arguments in the call to `erase` would have the same value, `words.end()`. The fact that the iterators are equal would mean that the range to `erase` would be empty. Erasing an empty range has no effect, so our program is correct even if the input has no duplicates.

值得注意的是，对没有重复元素的 `vector` 对象，调用 `erase` 也是安全的。如果不存在重复的元素，`unique` 就会返回 `words.end()`，此时，调用 `erase` 的两个实参值相同，都是 `words.end()`。两个迭代器相等这个事实意味着 `erase` 函数要删除的范围是空的。删除一段空的范围没有任何作用，所以即使输入中没有重复的元素，我们的程序仍然正确。

Defining Needed Utility Functions

定义需要的实用函数

Our next subproblem is to count how many words are of length six or greater. To solve this problem, we'll use two additional generic algorithms: `stable_sort` and `count_if`. To use each of these algorithms we'll need a companion utility function, known as a [predicates](#). A predicate is a function that performs some test and returns a type that can be used in a condition to indicate success or failure.

下一个子问题统计长度不小于 6 的单词个数。为了解决这个问题，需要用到另外两个泛型算法：`stable_sort` 和 `count_if`。使用这些算法，还需要一个配套的实用函数，称为[谓词](#)。谓词是做某些检测的函数，返回用于条件判断的类型，指出条件是否成立。

The first predicate we need will be used to sort the elements based on size. To do this sort, we need to define a predicate function that compares two `strings` and returns a `bool` indicating whether the first is shorter in length than the second:

我们需要的第一个谓词将用在基于大小的元素排序中。为了实现排序，必须定义一个谓词函数来实现两个 `string` 对象的比较，并返回一个 `bool` 值，指出第一个字符串是否比第二个短：

```
// comparison function to be used to sort by word length
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

The other function we need will determine whether a given `string` is of length six or greater:

另一个所需的谓词函数将判断给出的 `string` 对象的长度是否不小于 6：

```
// determine whether a length of a given word is 6 or more
bool GT6(const string &s)
{
    return s.size() >= 6;
}
```

Although this function solves our problem, it is unnecessarily limited—the function hardwires the size into the function itself. If we wanted to find out how many words were of another length, we'd have to write another function. We could easily write a more general comparison function that took two parameters, the `string` and the size. However, the function we pass to `count_if` takes a single argument, so we cannot use the more general approach in this program. We'll see a better way to write this part of our solution in [Section 14.8.1](#) (p. 531).

尽管这个函数能解决问题，但存在不必要的限制——函数内部硬性规定了对长度大小的要求。如果要统计其他长度的单词个数，则必须编写另一个函数。其实很容易写出更通用的比较函数，使它带有两个形参，分别是 `string` 对象和一个长度大小值即可。但是，传递给 `count_if` 算法的函数只能带有一个实参，因此本程序不能使用上述更通用的方法。[第 14.8.1 节](#)将为这个问题提供更好的解决方案。

Sorting Algorithms

排序算法

The library defines four different sort algorithms, of which we've used the simplest, `sort`, to sort `words` into alphabetical order. In addition to `sort`, the library also defines a `stable_sort` algorithm. A `stable_sort` maintains the original order among equal elements. Ordinarily, we don't care about the relative order of equal elements in a sorted sequence. After all, they're equal. However, in this case, we have defined "equal" to mean "the same length." Elements that have the same length can still be distinct when viewed alphabetically. By calling `stable_sort`, we maintain alphabetic order among those elements that have the same length.

Section 11.2. A First Look at the Algorithms

标准库定义了四种不同的排序算法，上面只使用了最简单的 `sort` 算法，使 `words` 按字典次序排列。除了 `sort` 之外，标准库还定义了 `stable_sort` 算法，`stable_sort` 保留相等元素的原始相对位置。通常，对于已排序的序列，我们并不关心其相等元素的相对位置，毕竟，这些元素是相等的。但是，在这个应用中，我们将“相等”定义为“相同的长度”，有着相同长度的元素还能以字典次序的不同而区分。调用 `stable_sort` 后，对于长度相同的元素，将保留其字典顺序。

Both `sort` and `stable_sort` are overloaded functions. One version uses the `<` operator for the element type to do the comparison. We used this version of `sort` to sort `words` before looking for duplicate elements. The second version takes a third parameter: the name of a predicate to use when comparing elements. That function must take two arguments of the same type as the element type and return a value that can be tested in a condition. We will use this second version, passing our `isShorter` function to compare elements:

`sort` 和 `stable_sort` 都是重载函数。其中一个版本使用元素类型提供的小于 (`<`) 操作符实现比较。在查找重复元素之前，我们就是用这个 `sort` 版本对元素排序。第二个重载版本带有第三个形参：比较元素所使用的谓词函数的名字。这个谓词函数必须接受两个实参，实参的类型必须与元素类型相同，并返回一个可用作条件检测的值。下面将比较元素的 `isShorter` 函数作为实参，调用第二个版本的排序函数：

```
// sort words by size, but maintain alphabetic order for words of the same size
stable_sort(words.begin(), words.end(), isShorter);
```

After this call, `words` is sorted by element size, but the words of each length are also still in alphabetical order:

调用后，`words` 中的元素按长度大小排序，而长度相同的单词则仍然保持字典顺序：

`words`

fox	red	the	over	slow	jumps	quick	turtle
-----	-----	-----	------	------	-------	-------	--------

Counting Words of Length Six or More

统计长度不小于 6 的单词

Now that we've reordered our `vector` by word size, our remaining problem is to count how many words are of length six or greater. The `count_if` algorithm handles this problem:

现在此 `vector` 对象已经按单词长度排序，剩下的问题是统计长度不小于 6 的单词个数。使用 `count_if` 算法处理这个问题：

```
vector<string>::size_type wc =
    count_if(words.begin(), words.end(), GT6);
```

`count_if` executes by reading the range denoted by its first two parameters. It passes each value that it reads to the predicate function represented by its third argument. That function must take a single argument of the element type and must return a value that can be tested as a condition. The algorithm returns a count of the number of elements for which the function succeeded. In this case, `count_if` passes each word to `GT6`, which returns the `bool` value `true` if the word's length is six or more.

执行 `count_if` 时，首先读取它的头两个实参所标记的范围内的元素。每读出一个元素，就将它传递给第三个实参表示的谓词函数。此谓词函数。此谓词函数需要单个元素类型的实参，并返回一个可用作条件检测的值。`count_if` 算法返回使谓词函数返回条件成立的元素个数。在这个程序中，`count_if` 将每个单词传递给 `GT6`，而 `GT6` 返回一个 `bool` 值，如果单词长度不小于 6，则该 `bool` 值为 `true`。

Putting It All Together

将全部程序段放在一起

Having looked at the program in detail, here is the program as a whole:

了解程序的细节之后，下面是完整的程序：

```
// comparison function to be used to sort by word length
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
// determine whether a length of a given word is 6 or more
bool GT6(const string &s)
{
    return s.size() >= 6;
}
int main()
{
    vector<string> words;
    // copy contents of each book into a single vector
    string next_word;
    while (cin >> next_word) {
        // insert next book's contents at end of words
        words.push_back(next_word);
    }
    // sort words alphabetically so we can find the duplicates
    sort (words.begin(), words.end());
    /* eliminate duplicate words:
```

Section 11.2. A First Look at the Algorithms

```
* unique reorders words so that each word appears once in the
*         front portion of words and returns an iterator one past the unique range;
* erase uses a vector operation to remove the nonunique elements
*/
vector<string>::iterator end_unique =
    unique(words.begin(), words.end());
words.erase(end_unique, words.end());
// sort words by size, but maintain alphabetic order for words of the same size
stable_sort(words.begin(), words.end(), isShorter);
vector<string>::size_type wc =
    count_if (words.begin(), words.end(), GT6);
cout << wc << " " << make_plural(wc, "word", "s")
    << " 6 characters or longer" << endl;
return 0;
}
```

We leave as an exercise the problem of printing the words in size order.

最后，我们留下按长度顺序输出单词这个问题作为习题。

Exercises Section 11.2.3

Exercise 11.9: Implement the program to count words of size 4 or greater, including printing the list of unique words in the input. Test your program by running it on the program's source file.

编写程序统计长度不小于 4 的单词，并输出输入序列中不重复的单词。在程序源文件上运行和测试你自己编写的程序。

Exercise 11.10: The library defines a `find_if` function. Like `find`, the `find_if` function takes a pair of iterators that indicates a range over which to operate. Like `count_if`, it also takes a third parameter that names a predicate that can be used to test each element in the range. `find_if` returns an iterator that refers to the first element for which the function returns a nonzero value. It returns its second iterator argument if there is no such element. Use the `find_if` function to rewrite the portion of our program that counted how many words are greater than length six.

标准库定义了一个 `find_if` 函数。与 `find` 一样，`find_if` 函数带有一对迭代器形参，指定其操作的范围。与 `count_if` 一样，该函数还带有第三个形参，表明用于检查范围内每个元素的谓词函数。`find_if` 返回一个迭代器，指向第一个谓词函数返回非零值的元素。如果这样的元素不存在，则返回第二个迭代器实参。使用 `find_if` 函数重写上述例题中统计长度大于 6 的单词个数的程序部分。

Exercise 11.11: Why do you think the algorithms don't change the size of containers?

你认为为什么算法不改变容器的大小？

Exercise 11.12: Why was it necessary to use `erase` rather than define a generic algorithm that could remove elements from the container?

为什么必须使用 `erase`，而不是定义一个泛型算法来删除容器中的元素？

11.3. Revisiting Iterators

11.3. 再谈迭代器

In [Section 11.2.2](#) (p. 398) we saw that the library defines iterators that are independent of a particular container. In fact, there are three additional kinds of iterators:

[第 11.2.2 节](#)已强调标准库所定义的迭代器不依赖于特定的容器。事实上，C++ 语言还提供了另外三种迭代器：

- **insert iterators:** These iterators are bound to a container and can be used to insert elements to the container.
插入迭代器：这类迭代器与容器绑定在一起，实现在容器中插入元素的功能。
- **iostream iterators:** These iterators can be bound to input or output streams and used to iterate through the associated IO stream.
iostream 迭代器：这类迭代器可与输入或输出流绑定在一起，用于迭代遍历所关联的 IO 流。
- **reverse iterators:** These iterators move backward, rather than forward. Each container type defines its own `reverse_iterator` types, which are returned by the `rbegin` and `rend` functions.
反向迭代器：这类迭代器实现向后遍历，而不是向前遍历。所有容器类型都定义了自己的 `reverse_iterator` 类型，由 `rbegin` 和 `rend` 成员函数返回。

These iterator types are defined in the `iterator` header.

上述迭代器类型都在 `iterator` 头文件中定义。

This section will look at each of these kinds of iterators and show how they can be used with the generic algorithms. We'll also take a look at how and when to use the container `const_iterators`.

本节将详细分析上述每种迭代器，并介绍在泛型算法中如何使用这些迭代器，还会了解什么时候应该使用和如何使用 `const_iterator` 容器。

11.3.1. Insert Iterators

11.3.1. 插入迭代器

In [Section 11.2.2](#) (p. 398) we saw that we can use `back_inserter` to create an iterator that adds elements to a container. The `back_inserter` function is an example of an **inserter**. An inserter is an iterator adaptor ([Section 9.7](#), p. 348) that takes a container and yields an iterator that inserts elements into the specified container. When we assign through an insert iterator, the iterator inserts a new element. There are three kinds of inserters, which differ as to where elements are inserted:

[第 11.2.2 节](#)使用 `back_insert` 创建一个迭代器，用来给容器添加元素。`back_inserter` 函数是一种插入器。插入器是一种迭代器适配器（[第 9.7 节](#)），带有一个容器参数，并生成一个迭代器，用于在指定容器中插入元素。通过插入迭代器赋值时，迭代器将会插入一个新的元素。C++ 语言提供了三种插入器，其差别在于插入元素的位置不同。

- `back_inserter`, which creates an iterator that uses `push_back`.
`back_inserter`, 创建使用 `push_back` 实现插入的迭代器。
- `front_inserter`, which uses `push_front`.
`front_inserter`, 使用 `push_front` 实现插入。
- `inserter`, which uses `insert`. In addition to a container, `inserter` takes a second argument: an iterator indicating the position ahead of which insertion should begin.
`inserter`, 使用 `insert` 实现插入操作。除了所关联的容器外，`inserter` 还带有第二实参：指向插入起始位置的迭代器。

`front_inserter` Requires `push_front`

Section 11.3. Revisiting Iterators

`front_inserter` 需要使用 `push_front`

`front_inserter` operates similarly to `back_inserter`: It creates an iterator that treats assignment as a call to `push_front` on its underlying container.

`front_inserter` 的操作类似于 `back_inserter`: 该函数将创建一个迭代器，调用它所关联的基础容器的 `push_front` 成员函数代替赋值操作。



We can use `front_inserter` only if the container has a `push_front` operation. Using `front_inserter` on a `vector`, or other container that does not have `push_front`, is an error.

只有当容器提供 `push_front` 操作时，才能使用 `front_inserter`。在 `vector` 或其他没有 `push_front` 运算的容器上使用 `front_inserter`，将产生错误。

`inserter` Yields an Iterator that Inserts at a Given Place

`inserter` 将产生在指定位置实现插入的迭代器

The `inserter` adaptor provides a more general form. This adaptor takes both a container and an iterator denoting a position at which to do the insertion:

`inserter` 适配器提供更普通的插入形式。这种适配器带有两个实参：所关联的容器和指示起始插入位置的迭代器。

```
// position an iterator into ilst
list<int>::iterator it =
    find(ilst.begin(), ilst.end(), 42);
// insert replaced copies of ivec at that point in ilst
replace_copy(ivec.begin(), ivec.end(),
    inserter(ilst, it), 100, 0);
```

We start by using `find` to locate an element in `ilst`. The call to `replace_copy` uses an `inserter` that will insert elements into `ilst` just before of the element denoted by the iterator returned from `find`. The effect of the call to `replace_copy` is to copy the elements from `ivec`, replacing each value of `100` by `0`. The elements are inserted just ahead of the element denoted by `it`.

首先用 `find` 定位 `ilst` 中的某个元素。使用 `inserter` 作为实参调用 `replace_copy`，`inserter` 将会在 `ilst` 中由 `find` 返回的迭代器所指向的元素前面插入新元素。而调用 `replace_copy` 的效果是从 `ivec` 中复制元素，并将其中值为 `100` 的元素替换为 `0` 值。`ilst` 的新元素在 `it` 所标明的元素前面插入。

When we create an `inserter`, we say where to insert new elements. Elements are always inserted in *front* of the position denoted by the iterator argument to `inserter`.

在创建 `inserter` 时，应指明新元素在何处插入。`inserter` 函数总是在它的迭代器实参所标明的位置前面插入新元素。

We might think that we could simulate the effect of `front_inserter` by using `inserter` and the `begin` iterator for the container. However, an `inserter` behaves quite differently from `front_inserter`. When we use `front_inserter`, the elements are always inserted ahead of the then first element in the container. When we use `inserter`, elements are inserted ahead of a specific position. Even if that position initially is the first element, as soon as we insert an element in front of that element, it is no longer the one at the beginning of the container:

也许我们会认为可使用 `inserter` 和容器的 `begin` 迭代器来模拟 `front_inserter` 的效果。然而，`inserter` 的行为与 `front_inserter` 的有很大差别。在使用 `front_inserter` 时，元素始终在容器的第一个元素前面插入。而使用 `inserter` 时，元素则在指定位置前面插入。即使此指定位置初始化为容器中的第一个元素，但是，一旦在该位置前插入一个新元素后，插入位置就不再是容器的首元素了：

```
list<int> ilst, ilst2, ilst3;      // empty lists
// after this loop ilst contains: 3 2 1 0
for (list<int>::size_type i = 0; i != 4; ++i)
    ilst.push_front(i);
// after copy ilst2 contains: 0 1 2 3
copy(ilst.begin(), ilst.end(), front_inserter(ilst2));
// after copy, ilst3 contains: 3 2 1 0
copy(ilst.begin(), ilst.end(),
    inserter(ilst3, ilst3.begin()));
```

When we copy into `ilst2`, elements are always inserted ahead of any other element in the `list`. When we copy into `ilst3`, elements are inserted at a fixed point. That point started out as the head of the `list`, but as soon as even one element is added, it is no longer the first element.

在复制并创建 `ilst2` 的过程中，元素总是在这个 `list` 对象的所有元素之前插入。而在复制创建 `ilst3` 的过程中，元素则在 `ilst3` 中的固定位置插入。刚开始时，这个插入位置是此 `list` 对象的头部，但插入一个元素后，就不再是首元素了。

Recalling the discussion in [Section 9.3.3](#) (p. 318), it is important to understand that using `front_inserter` results in the elements appearing in the destination in reverse order.

回顾[第 9.3.3 节](#)的讨论，应该清楚理解 `front_inserter` 的使用将导致元素以相反的次序出现在目标对象中，这点非常重要。



Exercises Section 11.3.1

Exercise 11.13: Explain the differences among the three insert iterators.

解释三种插入迭代器的区别。

Exercise 11.14: Write a program that uses `replace_copy` to copy a sequence from one container to another, replacing elements with a given value in the first sequence by the specified new value. Write the program to use an `inserter`, a `back_inserter` and a `front_inserter`. Discuss how the output sequence varies in each case.

编写程序使用 `replace_copy` 将一个容器中的序列复制给另一个容器，并将前一个序列中给定的值替换为指定的新值。分别使用 `inserter`、`back_inserter` 和 `front_inserter` 实现这个程序。讨论在不同情况下输出序列如何变化。

Exercise 11.15: The algorithms library defines a function named `unique_copy` that operates like `unique`, except that it takes a third iterator denoting a sequence into which to copy the unique elements. Write a program that uses `unique_copy` to copy the unique elements from a `list` into an initially empty `vector`.

算法标准库定义了一个名为 `unique_copy` 的函数，其操作与 `unique` 类似，唯一的区别在于：前者接受第三个迭代器实参，用于指定复制不重复元素的目标序列。编写程序使用 `unique_copy` 将一个 `list` 对象中不重复的元素复制到一个空的 `vector` 对象中。

11.3.2. `iostream` Iterators

11.3.2. `iostream` 迭代器

Even though the `iostream` types are not containers, there are iterators that can be used with `iostream` objects: An `istream_iterator` reads an input stream, and an `ostream_iterator` writes an output stream. These iterators treat their corresponding stream as a sequence of elements of a specified type. Using a `stream_iterator`, we can use the generic algorithms to read (or write) data to (or from) stream objects.

虽然 `iostream` 类型不是容器，但标准库同样提供了在 `iostream` 对象上使用的迭代器：`istream_iterator` 用于读取输入流，而 `ostream_iterator` 则用于写输出流（表 11.1）。这些迭代器将它们所对应的流视为特定类型的元素序列。使用流迭代器时，可以用泛型算法从流对象中读数据（或将数据写到流对象中）。

Table 11.1. `iostream` Iterator Constructors

表 11.1 `iostream` 迭代器的构造函数

<code>istream_iterator<T> in(strm);</code>	Create <code>istream_iterator</code> that reads objects of type <code>T</code> from input stream <code>strm</code> .
<code>istream_iterator<T> in;</code>	Create an off-the-end iterator for <code>istream_iterator</code> .
<code>ostream_iterator<T> in(strm);</code>	Create <code>ostream_iterator</code> that writes objects of type <code>T</code> to the output stream <code>strm</code> .

Section 11.3. Revisiting Iterators

<pre>ostream_iterator<T> in(strm, delim);</pre>	创建将 <code>T</code> 类型的对象写到输出流 <code>strm</code> 的 <code>ostream_iterator</code> 对象 Create <code>ostream_iterator</code> that writes objects of type <code>T</code> to the output stream <code>strm</code> using <code>delim</code> as a separator between elements. <code>delim</code> is a null-terminated character array.
	创建将 <code>T</code> 类型的对象写到输出流 <code>strm</code> 的 <code>ostream_iterator</code> 对象，在写入过程中使用 <code>delim</code> 作为元素的分隔符。 <code>delim</code> 是以空字符结束的字符数组

The stream iterators define only the most basic of the iterator operations: increment, dereference, and assignment. In addition, we can compare two `istream` iterators for equality (or inequality). There is no comparison for `ostream` iterators.

流迭代器只定义了最基本的迭代器操作：自增、解引用和赋值。此外，可比较两个 `istream` 迭代器是否相等（或不等）。而 `ostream` 迭代器则不提供比较运算（表 11.2）。

Table 11.2. `istream_iterator` Operations

表 11.2. `istream_iterator` 的操作

<code>it1 == it2</code> <code>it1 != it2</code>	Equality (inequality) between two <code>istream_iterator</code> s. The iterators must read the same type. Two iterators are equal if they are both the end value. Two non-end-of-stream iterators are equal if they are constructed using the same input stream.
	比较两上 <code>istream_iterator</code> 对象是否相等（不等）。迭代器读取的必须是相同的类型。如果两个迭代器都是 <code>end</code> 值，则它们相等。对于两个都不指向流结束位置的迭代器，如果它们使用同一个输入流构造，则它们也相等
<code>*it</code>	Returns the value read from the stream. 返回从流中读取的值
<code>it->mem</code>	Synonym for <code>(*it).mem</code> . Returns member, <code>mem</code> , of the object read from the stream. 是 <code>(*it).mem</code> 的同义词。返回从流中读取的对象的 <code>mem</code> 成员
<code>++it</code> <code>it++</code>	Advances the iterator by reading the next value from the input stream using the <code>>></code> operator for the element type. As usual, the prefix version advances the stream and returns a reference to the incremented iterator. The postfix version advances the stream but returns the old value. 通过使用元素类型提供的 <code>>></code> 操作从输入流中读取下一个元素值，使迭代器向前移动。通常，前缀版本使用迭代器在流中向前移动，并返回对加 1 后的迭代器的引用。而后缀版本使迭代器在流中向前移动后，返回原值

Defining Stream Iterators

流迭代器的定义

The stream iterators are class templates: An `istream_iterator` can be defined for any type for which the input operator (the `>>` operator) is defined. Similarly, an `ostream_iterator` can be defined for any type that has an output operator (the `<<` operator).

流迭代器都是类模板：任何已定义输入操作符（`>>` 操作符）的类型都可以定义 `istream_iterator`。类似地，任何已定义输出操作符（`<<` 操作符）的类型也可定义 `ostream_iterator`。

When we create a stream iterator, we must specify the type of objects that the iterator will read or write:

在创建流迭代器时，必须指定迭代器所读写的对象类型：

```
istream_iterator<int> cin_it(cin); // reads ints1 from cin
istream_iterator<int> end_of_stream; // end iterator value
// writes Sales_items from the ofstream named outfile
// each element is followed by a space
ofstream outfile;
```

Section 11.3. Revisiting Iterators

```
ostream_iterator<Sales_item> output(outfile, " ");
```

We must bind an `ostream_iterator` to a specific stream. When we create an `istream_iterator`, we can bind it to a stream. Alternatively, we can supply no argument, which creates an iterator that we can use as the off-the-end value. There is no off-the-end iterator for `ostream_iterator`.

`ostream_iterator` 对象必须与特定的流绑定在一起。在创建 `istream_iterator` 时，可直接将它绑定到一个流上。另一种方法是在创建时不提供实参，则该迭代器指向超出末端位置。`ostream_iterator` 不提供超出末端迭代器。

When we create an `ostream_iterator`, we may (optionally) provide a second argument that specifies a delimiter to use when writing elements to the output stream. The delimiter must be a C-style character string. Because it is a C-style string, it must be null-terminated; otherwise, the behavior is undefined.

在创建 `ostream_iterator` 对象时，可提供第二个（可选的）实参，指定将元素写入输出流时使用的分隔符。分隔符必须是 C 风格字符串。因为它是 C 风格字符串，所以必须以空字符结束；否则，其行为将是未定义的。

Operations on `istream_iterators`

`istream_iterator` 对象上的操作

Constructing an `istream_iterator` bound to a stream positions the iterator so that the first dereference reads the first value from the stream.

构造与流绑定在一起的 `istream_iterator` 对象时将对迭代器定位，以便第一次对该迭代器进行解引用时即可从流中读取第一个值。

As an example, we could use an `istream_iterator` to read the standard input into a `vector`:

考虑下面例子，可使用 `istream_iterator` 对象将标准输入读到 `vector` 对象中。

```
istream_iterator<int> in_iter(cin); // read ints from cin
istream_iterator<int> eof; // istream "end" iterator
// read until end of file, storing what was read in vec
while (in_iter != eof)
    // increment advances the stream to the next value
    // dereference reads next value from the istream
    vec.push_back(*in_iter++);
```

This loop reads `ints` from `cin`, and stores what was read in `vec`. On each trip the loop checks whether `in_iter` is the same as `eof`. That iterator was defined as the empty `istream_iterator`, which is used as the end iterator. An iterator bound to a stream is equal to the end iterator once its associated stream hits end-of-file or encounters another error.

这个循环从 `cin` 中读取 `int` 型数据，并将读入的内容保存在 `vec` 中。每次循环都检查 `in_iter` 是否为 `eof`。其中 `eof` 迭代器定义为空的 `istream_iterator` 对象，用作结束迭代器。绑在流上的迭代器在遇到文件结束或某个错误时，将等于结束迭代器的值。

The hardest part of this program is the argument to `push_back`, which uses the dereference and postfix increment operators. Precedence rules ([Section 5.5](#), p. 163) say that the result of the increment is the operand to the dereference. Incrementing an `istream_iterator` advances the stream. However, the expression uses the postfix increment, which yields the *old* value of the iterator. The effect of the increment is to read the next value from the stream but return an iterator that refers to the previous value read. We dereference that iterator to obtain that value.

本程序最难理解的部分是传递给 `push_back` 的实参，该实参使用解引用和后自增操作符。根据优先级规则（[第 5.5 节](#)），自增运算的结果将是解引用运算的操作数。对 `istream_iterator` 对象做自增运算使该迭代器在流中向前移动。然而，使用后自增运算的表达式，其结果是迭代器原来的价值。自增的效果是使迭代器的流中移动到下一个值，但返回指向一个值的迭代器。对该迭代器进行解引用获取该值。

What is more interesting is that we could rewrite this program as:

更有趣的是可以这样重写程序：

```
istream_iterator<int> in_iter(cin); // read ints from cin
istream_iterator<int> eof; // istream "end" iterator
vector<int> vec(in_iter, eof); // construct vec from an iterator range
```

Here we construct `vec` from a pair of iterators that denote a range of elements. Those iterators are `istream_iterators`, which means that the range is obtained by reading the associated stream. The effect of this constructor is to read `cin` until it hits end-of-file or encounters an input that is not an `int`. The elements that are read are used to construct `vec`.

这里，用一对标记元素范围的迭代器构造 `vec` 对象。这些迭代器是 `istream_iterator` 对象，这就意味着这段范围的元素是通过读取所关联的流来获得的。这个构造函数的效果是读 `cin`，直到到达文件结束或输入的不是 `int` 型数值为止。读取的元素将用于构造 `vec` 对象。

Using `ostream_iterators` and `ostream_iterators`

`ostream_iterator` 对象和 `ostream_iterator` 对象的使用

We can use an `ostream_iterator` to write a sequence of values to a stream in much the same way that we might use an iterator to assign a sequence of values to the elements of a container:

Section 11.3. Revisiting Iterators

可使用 `ostream_iterator` 对象将一个值序列写入流中，其操作的过程与使用迭代器将一组值逐个赋给容器中的元素相同：

```
// write one string per line to the standard output
ostream_iterator<string> out_iter(cout, "\n");
// read strings from standard input and the end iterator
istream_iterator<string> in_iter(cin), eof;
// read until eof and write what was read to the standard output
while (in_iter != eof)
    // write value of in_iter to standard output
    // and then increment the iterator to get the next value from cin
    *out_iter++ = *in_iter++;
```

This program reads `cin`, writing each word it reads on separate line on `cout`.

这个程序读 `cin`，并将每个读入的值依次写到 `cout` 中不同的行中。

We start by defining an `ostream_iterator` to write `strings` to `cout`, following each `string` by a newline. We define two `istream_iterators` that we'll use to read `strings` from `cin`. The `while` loop works similarly to our previous example. This time, instead of storing the values we read into a `vector`, we print them to `cout` by assigning the values we read to `out_iter`.

首先，定义一个 `ostream_iterator` 对象，用于将 `string` 类型的数据写到 `cout` 中，每个 `string` 对象后跟一个换行符。定义两个 `istream_iterator` 对象，用于从 `cin` 中读取 `string` 对象。`while` 循环类似前一个例子。但是这次不是将读取的数据存储在 `vector` 对象中，而是将读取的数据赋给 `out_iter`，从而输出到 `cout` 上。

The assignment works similarly to the one in the program on page 205 that copied one array into another. We dereference both iterators, assigning the right-hand value into the left, incrementing each iterator. The effect is to write what was read to `cout` and then increment each iterator, reading the next value from `cin`.

这个赋值类似于第 6.7 节将一个数组复制给另一个数组的程序。对这两个迭代器进行解引用，将右边的值赋给左边的元素，然后两个迭代器都自增 1。其效果就是：将读取的数据输出到 `cout` 上，然后两个迭代器都加 1，再从 `cin` 中读取下一个值。

Using `istream_iterator` with Class Types

在类类型上使用 `istream_iterator`

We can create an `istream_iterator` for any type for which an input operator (`>>`) exists. For example, we might use an `istream_iterator` to read a sequence of `Sales_item` objects to sum:

提供了输入操作符 (`>>`) 的任何类型都可以创建 `istream_iterator` 对象。例如，可如下使用 `istream_iterator` 对象读取一系列的 `Sales_item` 对象，并求和：

```
istream_iterator<Sales_item> item_iter(cin), eof;
Sales_item sum; // initially empty sales_item
sum = *item_iter++; // read first transaction into sum and get next record
while (item_iter != eof) {
    if (item_iter->same_isbn(sum))
        sum = sum + *item_iter;
    else {
        cout << sum << endl;
        sum = *item_iter;
    }
    ++item_iter; // read next transaction
}
cout << sum << endl; // remember to print last set of records
```

This program binds `item_iter` to `cin` and says that the iterator will read objects of type `Sales_item`. The program next reads the first record into `sum`:

该程序将迭代器 `item_iter` 与 `cin` 绑在一起，意味着迭代器将读取 `Sales_item` 类型的对象。然后给迭代器加 1，使流从标准输入中读取下一记录。

```
sum = *item_iter++; // read first transaction into sum and get next record
```

This statement uses the dereference operator to fetch the first record from the standard input and assigns that value to `sum`. It increments the iterator, causing the stream to read the next record from the standard input.

这个语句使用解引用操作符获取标准输入的第一个记录，并将这个值赋给 `sum`。然后给迭代器加 1，使流从标准输入中读取下一记录。

The `while` loop executes until we hit end-of-file on `cin`. Inside the `while`, we compare the `isbn` of the record we just read with `sum`'s `isbn`. The first statement in the `while` uses the arrow operator to dereference the `istream` iterator and obtain the most recently read object. We then run the `same_isbn` member on that object and the object in `sum`.

`while` 循环反复执行直到到达 `cin` 的结束位置为止。在 `while` 循环中，将刚读入记录的 `isbn` 与 `sum` 的 `isbn` 比较。`while` 中的第一个语句使用了箭头操作符对 `istream` 迭代器进行解引用，获得最近读入的对象。然后在该对象和 `sum` 对象上调用 `same_isbn` 成员。

If the `isbns` are the same, we increment the totals in `sum`. Otherwise, we print the current value of `sum` and reset it as a copy of the most recently read transaction. The last step in the loop is to increment the iterator, which in this case causes the next transaction to be read from the standard input. The loop continues until an error or end-of-file is encountered. Before exiting we remember to print the values associated with the last ISBN in the input.

Section 11.3. Revisiting Iterators

如果 `isbn` 值相同，则增加总和 `sum`。否则，输出 `sum` 的当前值，并将它重设为最近读取对象的副本。循环的最后一步是给迭代器加 1，在本例中，将导致从标准输入中读入下一个 `Sales_item` 对象。循环持续直到遇到错误或结束位置为止。在结束程序之前，记住输出从输入中读入的最后一个 `ISBN` 所关联的值。

Limitations on Stream Iterators

流迭代器的限制

The stream iterators have several important limitations:

流迭代器有下面几个重要的限制：

- It is not possible to read from an `ostream_iterator`, and it is not possible to write to an `istream_iterator`.
不可能从 `ostream_iterator` 对象读入，也不可能写到 `istream_iterator` 对象中。
- Once we assign a value to an `ostream_iterator`, the write is committed. There is no way to subsequently change a value once it is assigned. Moreover, each distinct value of an `ostream_iterator` is expected to be used for output exactly once.
一旦给 `ostream_iterator` 对象赋了一个值，写入就提交了。赋值后，没有办法再改变这个值。此外，`ostream_iterator` 对象中每个不同的值都只能正好输出一次。
- There is no `->` operator for `ostream_iterator`.
`ostream_iterator` 没有 `->` 操作符。

Using Stream Iterators with the Algorithms

与算法一起使用流迭代器

As we know, the algorithms operate in terms of iterator operations. And as we've seen, stream iterators define at least some of the iterator operations. Because the stream iterators support iterator operations, we can use them with at least some of the generic algorithms. As an example, we could read numbers from the standard input and write the unique numbers we read on the standard output:

```
istream_iterator<int> cin_it(cin);      // reads ints from cin
istream_iterator<int> end_of_stream;    // end iterator value
// initialize vec from the standard input:
vector<int> vec(cin_it, end_of_stream);
sort(vec.begin(), vec.end());

// writes ints to cout using " " as the delimiter
ostream_iterator<int> output(cout, " ");

// write only the unique elements in vec to the standard output
unique_copy(vec.begin(), vec.end(), output);
```

If the input to this program is

如果程序的输入是：

```
23 109 45 89 6 34 12 90 34 23 56 23 8 89 23
```

then the output would be

输出则是：

```
6 8 12 23 34 45 56 89 90 109
```

The program creates `vec` from the iterator pair, `input` and `end_of_stream`. The effect of this initializer is to read `cin` until end-of-file or an error occurs. The values read are stored in `vec`.

程序用一对迭代器 `input` 和 `end_of_stream` 创建了 `vec` 对象。这个初始化的效果是读取 `cin` 直到文件结束或者出现错误为止。读取的值保存在 `vec` 里。

Once the input is read and `vec` initialized, we call `sort` to sort the input. Duplicated items from the input will be adjacent after the call to `sort`.

读取输入和初始化 `vec` 后，调用 `sort` 对输入的数排序。`sort` 调用完成后，重复输入的数就会相邻存储。

The program uses `unique_copy`, which is a copying version of `unique`. It copies the unique values in its input range to the destination iterator. This

Section 11.3. Revisiting Iterators

call uses our output iterator as the destination. The effect is to copy the unique values from `vec` to `cout`, following each value by a space.

程序再使用 `unique_copy` 算法，这是 `unique` 的“复制”版本。该算法将输入范围中不重复的值复制到目标迭代器。该调用将输出迭代器用作目标。其效果是将 `vec` 中不重复的值复制给 `cout`，每个复制的值后面输出一个空格。

Exercises Section 11.3.2

Exercise 11.16: Rewrite the program on 410 to use the `copy` algorithm to write the contents of a file to the standard output.

重写（第 11.3.2 节第 3 小节）的程序，使用 `copy` 算法将一个文件的内容写到标准输出中。

Exercise 11.17: Use a pair of `istream_iterators` to initialize a `vector` of `int`s.

使用一对 `istream_iterator` 对象初始化一个 `int` 型的 `vector` 对象。

Exercise 11.18: Write a program to read a sequence of integer numbers from the standard input using an `istream_iterator`. Write the odd numbers into one file, using an `ostream_iterator`. Each value should be followed by a space. Write the even numbers into a second file, also using an `ostream_iterator`. Each of these values should be placed on a separate line.

编写程序使用 `istream_iterator` 对象从标准输入读入一系列整数。使用 `ostream_iterator` 对象将其中的奇数写到一个文件中，并在每个写入的值后面加一个空格。同样使用 `ostream_iterator` 对象将偶数写到第二个文件，每个写入的值都存放在单独的行中。

11.3.3. Reverse Iterators

11.3.3. 反向迭代器

A reverse iterator is an iterator that traverses a container backward. That is, it traverses from the last element toward the first. A reverse iterator inverts the meaning of increment (and decrement): `++` on a reverse iterator accesses the previous element; `--` accesses the next element.

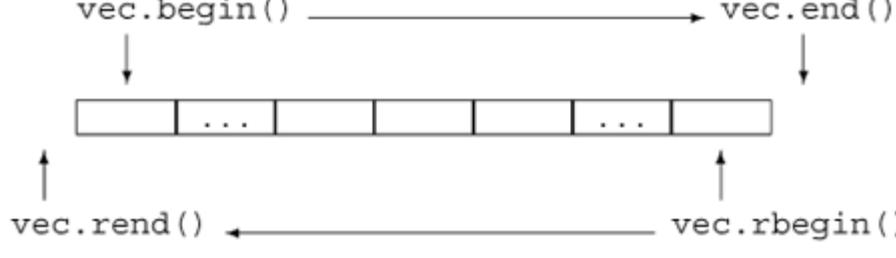
反向迭代器是一种反向遍历容器的迭代器。也就是，从最后一个元素到第一个元素遍历容器。反向迭代器将自增（和自减）的含义反过来：对于反向迭代器，`++` 运算将访问前一个元素，而 `--` 运算则访问下一个元素。

Recall that each container defines `begin` and `end` members. These members return respectively an iterator to the first element of the container and an iterator one past the last element of the container. The containers also define `rbegin` and `rend`, which return reverse iterators to the last element in the container and one “past” (that is, one before) the beginning of the container. As with ordinary iterators, there are both `const` and `nonconst` reverse iterators. [Figure 11.1](#) on the facing page illustrates the relationship between these four iterators on a hypothetical `vector` named `vec`.

回想一下，所有容器都定义了 `begin` 和 `end` 成员，分别返回指向容器首元素和尾元素下一位置的迭代器。容器还定义了 `rbegin` 和 `rend` 成员，分别返回指向容器尾元素和首元素前一位置的反向迭代器。与普通迭代器一样，反向迭代器也有常量 (`const`) 和非常量 (`nonconst`) 类型。[图 11.1](#) 使用一个假设名为 `vec` 的 `vector` 类型对象阐明了这四种迭代器之间的关系。

Figure 11.1. Comparing `begin/end` and `rbegin/rend` Iterators

图 11.1 比较 `begin/end` 和 `rbegin/rend` 迭代器



Given a `vector` that contains the numbers from 0 to 9 in ascending order

假设有一个 `vector` 容器对象，存储 0-9 这 10 个以升序排列的数字：

```
vector<int> vec;
for (vector<int>::size_type i = 0; i != 10; ++i)
```

Section 11.3. Revisiting Iterators

```
vec.push_back(i); // elements are 0,1,2,...9
```

the following `for` loop prints the elements in reverse order:

下面的 `for` 循环将以逆序输出这些元素：

```
// reverse iterator of vector from back to front
vector<int>::reverse_iterator r_iter;
for (r_iter = vec.rbegin(); // binds r_iter to last element
     r_iter != vec.rend(); // rend refers to before 1st element
     ++r_iter)           // decrements iterator one element
    cout << *r_iter << endl; // prints 9,8,7,...0
```

Although it may seem confusing to have the meaning of the increment and decrement operators reversed, doing so lets us use the algorithms transparently to process a container forward or backward. For example, we could sort our `vector` in descending order by passing `sort` a pair of reverse iterators:

虽然颠倒自增和自减这两个操作符的意义似乎容易使人迷惑，但是它让程序员可以透明地向前或向后处理容器。例如，为了以降序排列 `vector`，只需向 `sort` 传递一对反向迭代器：

```
// sorts vec in "normal" order
sort(vec.begin(), vec.end());
// sorts in reverse: puts smallest element at the end of vec
sort(vec.rbegin(), vec.rend());
```

Reverse Iterators Require Decrement Operators

反向迭代器需要使用自减操作符

Not surprisingly, we can define a reverse iterator only from an iterator that supports `--` as well as `++`. After all, the purpose of a reverse iterator is to move the iterator backward through the sequence. The iterators on the standard containers all support decrement as well as increment. However, the stream iterators do not, because it is not possible to move backward through a stream. Therefore, it is not possible to create a reverse iterator from a stream iterator.

从一个既支持 `--` 也支持 `++` 的迭代器就可以定义反向迭代器，这不用感到吃惊。毕竟，反向迭代器的目的是移动迭代器反向遍历序列。标准容器上的迭代器既支持自增运算，也支持自减运算。但是，流迭代器却不然，由于不能反向遍历流，因此流迭代器不能创建反向迭代器。

Relationship between Reverse Iterators and Other Iterators

反向迭代器与其他迭代器之间的关系

Suppose we have a `string` named `line` that contains a comma-separated list of words, and we want to print the first word in `line`. Using `find`, this task is easy:

假设有一个名为 `line` 的 `string` 对象，存储以逗号分隔的单词列表。我们希望输出 `line` 中的第一个单词。使用 `find` 可很简单地实现这个任务：

```
// find first element in a comma-separated list
string::iterator comma = find(line.begin(), line.end(), ',');
cout << string(line.begin(), comma) << endl;
```

If there is a comma in `line`, then `comma` refers to that comma; otherwise it is `line.end()`. When we print the `string` from `line.begin()` to `comma` we print characters up to the comma, or the entire `string` if there is no comma.

如果在 `line` 中有一个逗号，则 `comma` 指向这个逗号；否则，`comma` 的值为 `line.end()`。在输出 `string` 对象中从 `line.begin()` 到 `comma` 的内容时，从头开始输出字符直到遇到逗号为止。如果该 `string` 对象中没有逗号，则输出整个 `string` 字符串。

If we wanted the last word in the list, we could use reverse iterators instead:

如果要输出列表中最后一个单词，可使用反向迭代器：

```
// find last element in a comma-separated list
string::reverse_iterator rcomma =
    find(line.rbegin(), line.rend(), ',');
```

Because we pass `rbegin()` and `rend()`, this call starts with the last character in `line` and searches backward. When `find` completes, if there is a comma, then `rcomma` refers to the last comma in the line that it refers to the first comma found in the backward search. If there is no comma, then `rcomma` is `line.rend()`.

因为此时传递的是 `rbegin()` 和 `rend()`，这个函数调用从 `line` 的最后一个字符开始往回搜索。当 `find` 完成时，如果列表中有逗号，那么 `rcomma` 指向其最后一个逗号，即指向反向搜索找到的第一个逗号。如果没有逗号，则 `rcomma` 的值为 `line.rend()`。

Section 11.3. Revisiting Iterators

The interesting part comes when we try to print the word we found. The direct attempt

在尝试输出所找到的单词时，有趣的事情发生了。直接尝试：

```
// wrong: will generate the word in reverse order  
cout << string(line.rbegin(), rcomma) << endl;
```

generates bogus output. For example, had our input been

会产生假的输出。例如，如果输入是：

`FIRST,MIDDLE,LAST`

then this statement would print `TSAL!`

则将输出 `TSAL!`

[Figure 11.2](#) illustrates the problem: We are using reverse iterators, and such iterators process the `string` backward. To get the right output, we need to transform the reverse iterators `line.rbegin()` and `rcomma` into normal iterators that go forward. There is no need to transform `line.rbegin()` as we already know that the result of that transformation would be `line.end()`. We can transform `rcomma` by calling `base`, which is a member of each reverse iterator type:

[图 11.2](#) 阐明了这个问题：使用反向迭代器时，以逆序从后向前处理 `string` 对象。为了得到正确的输出，必须将反向迭代器 `line.rbegin()` 和 `rcomma` 转换为从前向后移动的普通迭代器。其实没必要转换 `line.rbegin()`，因为我们知道转换的结果必定是 `line.end()`。只需调用所有反向迭代器类型都提供的成员函数 `base` 转换 `rcomma` 即可：

Figure 11.2. Relationship between Reverse and Ordinary Iterators

图 11.2. 反向迭代器与普通迭代器之间的区别



```
// ok: get a forward iterator and read to end of line  
cout << string(rcomma.base(), line.end()) << endl;
```

Given the same preceding input, this statement prints `LAST` as expected.

假设还是前面给出的输入，该语句将如愿输出 `LAST`。

The objects shown in [Figure 11.2](#) visually illustrate the relationship between ordinary and reverse iterators. For example, `rcomma` and `rcomma.base()` refer to different elements, as do `line.rbegin()` and `line.end()`. These differences are needed to ensure that the range of elements whether processed forward or backward is the same. Technically speaking, the relationship between normal and reverse iterators is designed to accommodate the properties of a left-inclusive range ([Section 9.2.1](#), p. 314), so that `[line.rbegin(), rcomma]` and `[rcomma.base(), line.end()]` refer to the same elements in `line`.

[图 11.2](#) 显示的对象直观地解释了普通迭代器与反向迭代器之间的关系。例如，正如 `line.rbegin()` 和 `line.end()` 一样，`rcomma` 和 `rcomma.base()` 也指向不同的元素。为了确保正向和反向处理元素的范围相同，这些区别必要的。从技术上来说，设计普通迭代器与反向迭代器之间的关系是为了适应左闭合范围（[第 9.2.1 节](#)）这个性质的，所以，`[line.rbegin(), rcomma]` 和 `[rcomma.base(), line.end()]` 标记的是 `line` 中的相同元素。



The fact that reverse iterators are intended to represent ranges and that these ranges are asymmetric has an important consequence. When we initialize or assign a reverse iterator from a plain iterator, the resulting iterator does not refer to the same element as the original.

反向迭代器用于表示范围，而所表示的范围是不对称的，这个事实可推导出一个重要的结论：使用普通的迭代器对反向迭代器进行初始化或赋值时，所得到的迭代器并不是指向原迭代器所指向的元素。

Exercises Section 11.3.3

Exercise Write a program that uses `reverse_iterators` to print the contents of a `vector` in reverse order.

11.19:

编写程序使用 `reverse_iterator` 对象以逆序输出 `vector` 容器对象的内容。

Section 11.3. Revisiting Iterators

Exercise Now print the elements in reverse order using ordinary iterators.

11.20: 现在，使用普通的迭代器逆序输出上题中对象的元素。

Exercise Use `find` to find the last element in a `list` of `ints` with value 0.

11.21: 使用 `find` 在一个 `int` 型的 `list` 中寻找值为 0 的最后一个元素。

Exercise Given a `vector` that has 10 elements, copy the elements from position 3 through 7 in reverse order to a `list`.

假设有一个存储了 10 个元素的 `vector` 对象，将其中第 3 个至第 7 个位置上的元素以逆序复制给 `list` 对象。

11.3.4. `const` Iterators

11.3.4. `const` 迭代器

Careful readers will have noted that in the program on page 392 that used `find`, we defined `result` as a `const_iterator`. We did so because we did not intend to use the iterator to change a container element.

细心的读者可能已经注意到，在第 11.1 节使用 `find` 的程序中，我们将 `result` 定义为 `const_iterator` 类型。这样做是因为我们不希望使用这个迭代器来修改容器中的元素。

On the other hand, we used a plain, non`const` iterator to hold the return from `find_first_of` on page 397, even though we did not intend to change any container elements in that program either. The difference in treatment is subtle and deserves an explanation.

另一方面，虽然第 11.2.1 节的程序也不打算改变容器内的任何元素，但是它却使用了普通的非 `const` 迭代器来保存 `find_first_of` 的返回值。这两种处理存在细微的差别，值得解释一下。

The reason is that in the second case, we use the iterator as an argument to `find_first_of`:

原因是，在第二个例子中，程序将迭代器用作 `find_first_of` 的实参：

```
find_first_of(it, roster1.end(),
              roster2.begin(), roster2.end())
```

The input range for this call is specified by `it` and the iterator returned from a call to `roster1.end()`. Algorithms require the iterators that denote a range to have *exactly* the same type. The iterator returned by `roster1.end()` depends on the type of `roster1`. If that container is a `const` object, then the iterator is `const_iterator`; otherwise, it is the plain `iterator` type. In this program, `roster1` was not `const`, and so `end` returns an `iterator`.

该函数调用的输入范围由 `it` 和调用 `roster1.end()` 返回的迭代器指定。算法要求用于指定范围的两个迭代器必须具有完全一样的类型。`roster1.end()` 返回的迭代器依赖于 `roster1` 的类型。如果该容器是 `const` 对象，则返回的迭代器是 `const_iterator` 类型；否则，就是普通的 `iterator` 类型。在这个程序中，`roster1` 不是 `const` 对象，因而 `end` 返回的只是一个普通的迭代器。

If we defined `it` as a `const_iterator`, the call to `find_first_of` would not compile. The types of the iterators used to denote the range would not have been identical. `it` would have been a `const_iterator`, and the iterator returned by `roster1.end()` would be `iterator`.

如果我们将 `it` 定义为 `const_iterator`，那么 `find_first_of` 的调用将无法编译。用来指定范围的两个迭代器的类型不相同。`it` 是 `const_iterator` 类型的对象，而 `roster1.end()` 返回的则是一个 `iterator` 对象。

11.3.5. The Five Iterator Categories

11.3.5. 五种迭代器

Iterators define a common set of operations, but some iterators are more powerful than other iterators. For example, `ostream_iterators` support only increment, dereference, and assignment. Iterators on `vectors` support these operations and the decrement, relational, and arithmetic operators as well. As a result, we can classify iterators based on the set of operations they provide.

迭代器定义了常用的操作集，但有些迭代器具有比其他迭代器更强大的功能。例如 `ostream_iterator` 只支持自增、解引用和赋值运算，而 `vector` 容器提供的迭代器除了这些运算，还支持自减、关系和算术运算。因此，迭代器可根据所提供的操作集进行分类。

Similarly, we can categorize algorithms by the kinds of operations they require from their iterators. Some, such as `find`, require only the ability to read through the iterator and to increment it. Others, such as `sort`, require the ability to read, write, and randomly access elements. The iterator operations required by the algorithms are grouped into five categories. These five categories correspond to five categories of iterators, which are summarized in [Table 11.3](#).

Section 11.3. Revisiting Iterators

类似地，还可根据算法要求它的迭代器提供什么类型的操作，对算法分类。有一些算法，例如 `find`，只要求迭代器提供读取所指向内容和自增的功能。另一些算法，，比如 `sort`，则要求其迭代器有读、写和随机访问元素的能力。算法要求的迭代器操作分为五个类别，分别对应表 11.3 列出的五种迭代器。

Table 11.3. Iterator Categories

表 11.3. 迭代器种类

Input iterator (输入迭代器)	Read, but not write; increment only 读，不能写；只支持自增运算
Output iterator (输出迭代器)	Write, but not read; increment only 写，不能读；只支持自增运算
Forward iterator (前向迭代器)	Read and write; increment only 读和写；只支持自增运算
Bidirectional iterator (双向迭代器)	Read and write; increment and decrement 读和写；支持自增和自减运算
Random access iterator (随机访问迭代器)	Read and write; full iterator arithmetic 读和写；支持完整的迭代器算术运算

1. **Input iterators** can read the elements of a container but are not guaranteed to be able to write into a container. An input iterator must provide the following minimum support:

输入迭代器可用于读取容器中的元素，但是不保证能支持容器的写入操作。输入迭代器必须至少提供下列支持。

- Equality and inequality operators (`==`, `!=`) to compare two iterators.
相等和不等操作符 (`==`, `!=`)，比较两个迭代器。
- Prefix and postfix increment (`++`) to advance the iterator.
前置和后置的自增运算 (`++`)，使迭代器向前递进指向下一个元素。
- Dereference operator (`*`) to read an element; dereference may appear only on the right-hand side of an assignment.
用于读取元素的解引用操作符 (`*`)，此操作符只能出现在赋值运算的右操作数上。
- The arrow operator (`->`) as a synonym for `(*it).member` that is, dereference the iterator and fetch a member from the underlying object.
箭头操作符 (`->`)，这是 `(*it).member` 的同义语，也就是说，对迭代器进行解引用来获取其所关联的对象的成员。

Input iterators may be used only sequentially; there is no way to examine an element once the input iterator has been incremented. Generic algorithms requiring only this level of support include `find` and `accumulate`. The library `istream_iterator` type is an input iterator.

输入迭代器只能顺序使用；一旦输入迭代器自增了，就无法再用它检查之前的元素。要求在这个层次上提供支持的泛型算法包括 `find` 和 `accumulate`。标准库 `istream_iterator` 类型输入迭代器。

2. **Output iterators** can be thought of as having complementary functionality to input iterators; An output iterator can be used to write an element but it is not guaranteed to support reading. Output iterators require:

输出迭代器 可视为与输入迭代器功能互补的迭代器；输出迭代器可用于向容器写入元素，但是不保证能支持读取容器内容。输出迭代器要求：

- Prefix and postfix increment (`++`) to advance the iterator.
前置和后置的自增运算 (`++`)，使迭代器向前递进指向下一个元素。
- Dereference (`*`), which may appear only as the left-hand side of an assignment. Assigning to a dereferenced output iterator writes to the underlying element.
解引用操作符 (`*`)，引操作符只能出现在赋值运算的左操作数上。给解引用的输出迭代器赋值，将对该迭代器所指向的元素做写入操作。

Output iterators may require that each iterator value must be written exactly once. When using an output iterator, we should use `*` once and only once on a given iterator value. Output iterators are generally used as a third argument to an algorithm and mark the position where writing should begin. For example, the `copy` algorithm takes an output iterator as its third parameter and copies elements from its input range to the destination indicated by the output iterator. The `ostream_iterator` type is an output iterator.

Section 11.3. Revisiting Iterators

输出迭代器可以要求每个迭代器的值必须正好写入一次。使用输出迭代器时，对于指定的迭代器值应该使用一次 `*` 运算，而且只能用一次。输出迭代器一般用作算法的第三个实参，标记起始写入的位置。例如，`copy` 算法使用一个输出迭代器作为它的第三个实参，将输入范围内的元素复制到输出迭代器指定的目标位置。标准库 `ostream_iterator` 类型输出迭代器。

3. **Forward iterators** read from and write to a given container. They move in only one direction through the sequence. Forward iterators support all the operations of both input iterators and output iterators. In addition, they can read or write the same element multiple times. We can copy a forward iterator to remember a place in the sequence so as to return to that place later. Generic algorithms that require a forward iterator include `replace`.

前向迭代器 用于读写指定的容器。这类迭代器只会以一个方向遍历序列。前向迭代器支持输入迭代器和输出迭代器提供的所有操作，除此之外，还支持对同一个元素的多次读写。可复制前向迭代器来记录序列中的一个位置，以便将来返回此处。需要前向迭代器的泛型算法包括 `replace`。

4. **Bidirectional iterators** read from and write to a container in both directions. In addition to supporting all the operations of a forward iterator, a bidirectional iterator also supports the prefix and postfix decrement (`--`) operators. Generic algorithms requiring a bidirectional iterator include `reverse`. All the library containers supply iterators that at a minimum meet the requirements for a bidirectional iterator.

双向迭代器 从两个方向读写容器。除了提供前向迭代器的全部操作之外，双向迭代器还提供前置和后置的自减运算 (`--`)。需要使用双向迭代器的泛型算法包括 `reverse`。所有标准库容器提供的迭代器都至少达到双向迭代器的要求。

5. **Random-access iterators** provide access to any position within the container in constant time. These iterators support all the functionality of bidirectional iterators. In addition, random-access iterators support:

随机访问迭代器 提供在常量时间内访问容器任意位置的功能。这种迭代器除了支持双向迭代器的所有功能之外，还支持下面的操作：

- The relational operators `<`, `<=`, `>`, and `>=` to compare the relative positions of two iterators.
关系操作符 `<`、`<=`、`>` 和 `>=`，比较两个迭代器的相对位置。
- Addition and subtraction operators `+`, `+=`, `-`, and `-=` between an iterator and an integral value. The result is the iterator advanced (or retreated) the integral number of elements within the container.
迭代器与整型数值 `n` 之间的加法和减法操作符 `+`、`+=`、`-` 和 `-=`，结果是迭代器在容器中向前（或退回）`n` 个元素。
- The subtraction operator `-` when applied to two iterators, which yields the distance between two iterators.
两个迭代器之间的减法操作符 (`--`)，得到两个迭代器间的距离。
- The subscript operator `iter[n]` as a synonym for `*(iter + n)`.
下标操作符 `iter[n]`，这是 `*(iter + n)` 的同义词。

Generic algorithms requiring a random-access iterator include the `sort` algorithms. The `vector`, `deque`, and `string` iterators are random-access iterators, as are pointers when used to access elements of a built-in array.

需要随机访问迭代器的泛型算法包括 `sort` 算法。`vector`、`deque` 和 `string` 迭代器是随机访问迭代器，用作访问内置数组元素的指针也是随机访问迭代器。

With the exception of output iterators, the iterator categories form a sort of hierarchy: Any iterator of a higher category can be used where an iterator of lesser power is required. We can call an algorithm requiring an input iterator with an input iterator or a forward, bidirectional, or random-access iterator. Only a random-access iterator may be passed to an algorithm requiring a random-access iterator.

除了输出迭代器，其他类别的迭代器形成了一个层次结构：需要低级类别迭代器的地方，可使用任意一种更高级的迭代器。对于需要输入迭代器的算法，可传递前向、双向或随机访问迭代器调用该算法。调用需要随机访问迭代器的算法时，必须传递随机访问迭代器。

The `map`, `set`, and `list` types provide bidirectional iterators. Iterators on `string`, `vector`, and `deque` are random-access iterators, as are pointers bound to arrays. An `istream_iterator` is an input iterator, and an `ostream_iterator` is an output iterator.

`map`、`set` 和 `list` 类型提供双向迭代器，而 `string`、`vector` 和 `deque` 容器上定义的迭代器都是随机访问迭代器都是随机访问迭代器，用作访问内置数组元素的指针也是随机访问迭代器。`istream_iterator` 是输入迭代器，而 `ostream_iterator` 则是输出迭代器。

Key Concept: Associative Containers and the Algorithms

关键概念：关联容器与算法

Although the `map` and `set` types provide bidirectional iterators, we can use only a subset of the algorithms on associative containers. The problem is that the key in an associative container is `const`. Hence, any algorithm that writes to elements in the sequence cannot be used on an associative container. We may use iterators bound to associative containers only to supply arguments that will be read.

尽管 `map` 和 `set` 类型提供双向迭代器，但关联容器只能使用算法的一个子集。问题在于：关联容器的键是 `const` 对象。因此，关联容器不能使用任何写序列元素的算法。只能使用与关联容器绑在一起的迭代器来提供用于读操作的实参。

When dealing with the algorithms, it is best to think of the iterators on associative containers as if they were input iterators that also support decrement, not as full bidirectional iterators.



在处理算法时，最好将关联容器上的迭代器视为支持自减运算的输入迭代器，而不是完整的双向迭代器。

The C++ standard specifies the minimum iterator category for each iterator parameter of the generic and numeric algorithms. For example, `find` which implements a one-pass, read-only traversal over a container minimally requires an input iterator. The `replace` function requires a pair of iterators that are at least forward iterators. The first two iterators to `replace_copy` must be at least forward. The third, which represents a destination, must be at least an output iterator.

C++ 标准为所有泛型和算术算法的每一个迭代器形参指定了范围最小的迭代器种类。例如，`find`（以只读方式单步遍历容器）至少需要一个输入迭代器。`replace` 函数至少需要一对前向迭代器。`replace_copy` 函数的头两个迭代器必须至少是前向迭代器，第三个参数代表输出目标，必须至少是输出迭代器。

For each parameter, the iterator must be at least as powerful as the stipulated minimum. Passing an iterator of a lesser power results in an error; passing an stronger iterator type is okay.

对于每一个形参，迭代器必须保证最低功能。将支持更少功能的迭代器传递给函数是错误的；而传递更强功能的迭代器则没问题。

 Errors in passing an invalid category of iterator to an algorithm are not guaranteed to be caught at compile-time.

向算法传递无效的迭代器类别所引起的错误，无法保证会在编译时被捕获到。

Exercises Section 11.3.5

Exercise 11.23: List the five iterator categories and the operations that each supports.

列出五种迭代器类型及其各自支持的操作。

Exercise 11.24: What kind of iterator does a `list` have? What about a `vector`?

`list` 容器拥有什么类型的迭代器？而 `vector` 呢？

Exercise 11.25: What kinds of iterators do you think `copy` requires? What about `reverse` or `unique`?

你认为 `copy` 算法需要使用哪种迭代器？而 `reverse` 和 `unique` 呢？

Exercise 11.26: Explain why each of the following is incorrect. Identify which errors should be caught during compilation.

解释下列代码错误的原因，指出哪些错误可以在编译时捕获。

- (a) `string sa[10];
const vector<string> file_names(sa, sa+6);
vector<string>::iterator it = file_names.begin() + 2;`
- (b) `const vector<int> ivec;
fill(ivec.begin(), ivec.end(), ival);`
- (c) `sort(ivec.begin(), ivec.rend());`
- (d) `sort(ivec1.begin(), ivec2.end());`

11.4. Structure of Generic Algorithms

11.4. 泛型算法的结构

Just as there is a consistent design pattern behind the containers, there is a common design underlying the algorithms. Understanding the design behind the library makes it easier to learn and easier to use the algorithms. Because there are more than 100 algorithms, it is much better to understand their structure than to memorize the whole list of algorithms.

正如所有的容器都建立在一致的设计模式上一样，算法也具有共同的设计基础。理解标准算法库的设计基础有利于学习和使用算法。C++ 提供了超过一百个算法，了解它们的结构显然要比死记所有的算法更好。

The most fundamental property of any algorithm is the kind(s) of iterators it expects. Each algorithm specifies for each of its iterator parameters what kind of iterator can be supplied. If a parameter must be a random-access iterator, then we can provide an iterator for a `vector` or a `deque`, or we can supply a pointer into an array. Iterators on the other containers cannot be used with such algorithms.

算法最基本的性质是需要使用的迭代器种类。所有算法都指定了它的每个迭代器形参可使用的迭代器类型。如果形参必须为随机访问迭代器则可提供 `vector` 或 `deque` 类型的迭代器，或者提供指向数组的指针。而其他容器的迭代器不能用在这类算法上。

A second way is to classify the algorithms is as we did in the beginning of this chapter. We can categorize them by what actions they take on the elements:

另一种算法分类的方法，则如本章开头介绍的一样，根据对元素的操作将算法分为下面几种：

- Some are read-only and leave element values and ordering unchanged.
只读算法，不改变元素的值顺序。
- Others assign new values to specific elements.
给指定元素赋新值的算法。
- Others move values from one element to another.
将一个元素的值移给另一个元素的算法。

As we'll see in the remainder of this section, there are two additional patterns to the algorithms: One pattern is defined by the parameters the algorithms take; the other is defined by two function naming and overloading conventions.

正如本节后续部分所介绍的，C++ 还提供了另外两种算法模式：一种模式由算法所带的形参定义；另一种模式则通过两种函数命名和重载的规范定义。

11.4.1. Algorithm Parameter Patterns

11.4.1. 算法的形参模式

Superimposed on any other classification of the algorithms is a set of parameter conventions. Understanding these parameter conventions can aid in learning new algorithms by knowing what the parameters mean, you can concentrate on understanding the operation the algorithm performs. Most of algorithms take one of the following four forms:

任何其他的算法分类都含有一组形参规范。理解这些形参规范有利于学习新的算法——只要知道形参的含义，就可专注于了解算法实现的操作。大多数算法采用下面四种形式之一：

```
alg (beg, end, other parms);
alg (beg, end, dest, other parms);
alg (beg, end, beg2, other parms);
alg (beg, end, beg2, end2, other parms);
```

where `alg` is the name of the algorithm, and `beg` and `end` denote the range of elements on which the algorithm operates. We typically refer to this range as the "input range" of the algorithm. Although nearly all algorithms take an input range, the presence of the other parameters depends on the work being performed. The common ones listed here `dest`, `beg2` and `end2` are all iterators. When used, these iterators fill similar roles. In addition to these iterator parameters, some algorithms take additional, noniterator parameters that are algorithm-specific.

其中，`alg` 是算法的名字，`beg` 和 `end` 指定算法操作的元素范围。我们通常将该范围称为算法的“输入范围”。尽管几乎所有算法都有输入范围，但算法是否使用其他形参取决于它所执行的操作。这里列出了比较常用的其他形参：`dest`、`beg2` 和 `end2`，它们都是迭代器。这些迭代器在使用时，充当类似的角色。除了这些迭代器形参之外，有些算法还带有其他的非迭代器形参，它们是这些算法特有的。

Algorithms with a Single Destination Iterator

带有单个目标迭代器的算法

A `dest` parameter is an iterator that denotes a destination used to hold the output. Algorithms assume that it is safe to write as many elements as needed.

`dest` 形参是一个迭代器，用于指定存储输出数据的目标对象。算法假定无论需要写入多少个元素都是安全的。



When calling these algorithms, it is essential to ensure that the output container is sufficiently large to hold the output, which is why they are frequently called with insert iterators or `ostream_iterator`. If we call these algorithms with a container iterator, the algorithm assumes there are as many elements as needed in that container.

调用这些算法时，必须确保输出容器有足够大的容量存储输出数据，这正是通常要使用插入迭代器或者 `ostream_iterator` 来调用这些算法的原因。如果使用容器迭代器调用这些算法，算法将假定容器里有足够的元素。

If `dest` is an iterator on a container, then the algorithm writes its output to existing elements within the container. More commonly, `dest` is bound to an insert iterator ([Section 11.3.1, p. 406](#)) or an `ostream_iterator`. An insert iterator adds elements to the container, ensuring that there is enough space. An `ostream_iterator` writes to an output stream, again presenting no problem regardless of how many elements are written.

如果 `dest` 是容器上的迭代器，则算法将输出内容写到容器中已存在的元素上。更普遍的用法是，将 `dest` 与某个插入迭代器（[第 11.3.1 节](#)）或者 `ostream_iterator` 绑定在一起。插入迭代器在容器中添加元素，以确保容器有足够的空间存储输出。`ostream_iterator` 则实现写输出流的功能，无需要考虑所写的元素个数。

Algorithms with a Second Input Sequence

带第二个输入序列的算法

Algorithms that take either `beg2` alone or `beg2` and `end2` use these iterators to denote a second input range. These algorithms typically use the elements from the second range in combination with the input range to perform a computation. When an algorithm takes both `beg2` and `end2`, these iterators are used to denote the entire second range. That is, the algorithm takes two completely specified ranges: the input range denoted by `beg` and `end`, and a second input range denoted by `beg2` and `end2`.

有一些算法带有一个 `beg2` 迭代器形参，或者同时带有 `beg2` 和 `end2` 迭代器形参，来指定它的第二个输入范围。这类算法通常将联合两个输入范围的元素来完成计算功能。算法同时使用 `beg2` 和 `end2` 时，这些迭代器用于标记完整的第二个范围。也就是说，此时，算法完整地指定了两个范围：`beg` 和 `end` 标记第一个输入范围，而 `beg2` 和 `end2` 则标记第二个输入范围。

Algorithms that take `beg2` but not `end2` treat `beg2` as the first element in the second input range. The end of this range is not specified. Instead, these algorithms assume that the range starting at `beg2` is at least as large as the one denoted by `beg, end`.

带有 `beg2` 而不带 `end2` 的算法将 `beg2` 视为第二个输入范围的首元素，但没有指定该范围的最后一个元素。这些算法假定以 `beg2` 开始的范围至少与 `beg` 和 `end` 指定的范围一样大。



As with algorithms that write to `dest`, algorithms that take `beg2` alone assume that the sequence beginning at `beg2` is as large as the range denoted by `beg` and `end`.

与写入 `dest` 的算法一样，只带有 `beg2` 的算法也假定以 `beg2` 开始的序列与 `beg` 和 `end` 标记的序列一样大。

11.4.2. Algorithm Naming Conventions

11.4.2. 算法的命名规范

The library uses a set of consistent naming and overload conventions that can simplify learning the library. There are two important patterns. The first involves algorithms that test the elements in the input range, and the second applies to those that reorder elements within the input range.

标准库使用一组相同的命名和重载规范，了解这些规范有助于更容易地学习标准库。它们包括两种重要模式：第一种模式包括测试输入范围内元素的算法，第二种模式则应用于对输入范围内元素重新排序的算法。

Distinguishing Versions that Take a Value or a Predicate

区别带有一个值或一个谓词函数参数的算法版本

Many algorithms operate by testing elements in their input range. These algorithms typically use one of the standard relational operators, either `==` or `<`. Most of the algorithms provide a second version that allows the programmer to override the use of the operator and instead to supply a comparison or test function.

很多算法通过检查其输入范围内的元素实现其功能。这些算法通常要用到标准关系操作符：`==` 或 `<`。其中的大部分算法会提供第二个版本的函数，允许程序员提供比较或测试函数取代操作符的使用。

Algorithms that reorder the container elements use the `<` operator. These algorithms define a second, overloaded version that takes an additional parameter representing a different operation to use to order the elements:

重新对容器元素排序的算法要使用 `<` 操作符。这些算法的第二个重载版本带有一个额外的形参，表示用于元素排序的不同运算：

```
sort(beg, end);           // use < operator to sort the elements
sort(beg, end, comp);    // use function named comp to sort the elements
```

Algorithms that test for a specific value use the `==` operator by default. These algorithms provide a second named not overloaded version with a parameter that is a predicate ([Section 11.2.3, p. 402](#)). Algorithms that take a predicate have the suffix `_if` appended:

检查指定值的算法默认使用 `==` 操作符。系统为这类算法提供另外命名的（而非重载的）版本，带有谓词函数（[第 11.2.3 节](#)）形参。带有谓词函数形参的算法，其名字带有后缀 `_if`：

```
find(beg, end, val);      // find first instance of val in the input range
find_if(beg, end, pred);  // find first instance for which pred is true
```

These algorithms both find the first instance of a specific element in the input range. The `find` algorithm looks for a specific value; the `find_if` algorithm looks for a value for which `pred` returns a nonzero value.

上述两个算法都在输入范围内寻找指定元素的第一个实例。其中，`find` 算法查找一个指定的值，而 `find_if` 算法则用于查找一个使谓词函数 `pred` 返回非零值的元素。

The reason these algorithms provide a named version rather than an over-loaded one is that both versions take the same number of parameters. In the case of the ordering algorithms, it is easy to disambiguate a call based solely on the number of parameters. In the case of algorithms that look for a specific element, the number of parameters is the same whether testing for a value or testing a predicate. Overloading ambiguities ([Section 7.8.2, p. 269](#)) would therefore be possible, albeit rare, and so the library provides two named versions for these algorithms rather than relying on overloading.

标准库为这些算法提供另外命名的版本，而非重载版本，其原因在于这个两种版本的算法带有相同数目的形参。对于排序算法，只要根据参数的个数就很容易消除函数调用的歧义。而对于查找指定元素的算法，不管检查的是一个值还是谓词函数，函数调用都需要相同个数的参数。此时，如果使用重载版本，则可能导致二义性（[第 7.8.2 节](#)），尽管这个可能出现的几率很低。因此，标准库为这些算法提供两种不同名字的版本，而没有使用重载。

Distinguishing Versions that Copy from Those that Do Not

区别是否实现复制的算法版本

Independently of whether an algorithm tests its elements, the algorithm may re-arrange elements within the input range. By default, such algorithms write the rearranged elements back into their input range. These algorithms also provide a second, named version that writes to a specified output destination. These algorithms append `_copy` to their names:

```
reverse(beg, end);
reverse_copy(beg, end, dest);
```

The `reverse` function does what its name implies: It reverses the order of the elements in the input sequence. The first version reverses the elements in the input sequence itself. The second version, `reverse_copy`, makes a copy of the elements, placing them in reverse order in the sequence that begins at `dest`.

`reverse` 函数的功能就如它的名字所意味的：将输入序列中的元素反射重新排列。其中，第一个函数版本将自己的输入序列中的元素反向重排。而第二个版本，`reverse_copy`，则复制输入序列的元素，并将它们逆序存储到 `dest` 开始的序列中。

Exercises Section 11.4.2

Exercise The library defines the following algorithms:

11.27:

标准库定义了下面的算法：

```
replace(beg, end, old_val, new_val);
replace_if(beg, end, pred, new_val);
replace_copy(beg, end, dest, old_val, new_val);
```

Section 11.4. Structure of Generic Algorithms

```
replace_copy_if(beg, end, dest, pred, new_val);
```

Based only on the names and parameters to these functions, describe the operation that these algorithms perform.

只根据这些函数的名字和形参，描述这些算法的功能。

Exercise 11.28: Assume `lst` is a container that holds 100 elements. Explain the following program fragment and fix any bugs you think are present.

假设 `lst` 是存储了 100 个元素的容器。请解释下面的程序段，并修正你认为的错误。

```
vector<int> vec1;
reverse_copy(lst.begin(), lst.end(), vec1.begin());
```

Team LiB

◀ PREVIOUS NEXT ▶

11.5. Container-Specific Algorithms

11.5. 容器特有的算法

The iterators on `list` are bidirectional, not random access. Because the `list` container does not support random access, we cannot use the algorithms that require random-access iterators. These algorithms include the `sort`-related algorithms. There are other algorithms, defined generically, such as `merge`, `remove`, `reverse`, and `unique`, that can be used on `lists` but at a cost in performance. These algorithms can be implemented more efficiently if they can take advantage of how `lists` are implemented.

`list` 容器上的迭代器是双向的，而不是随机访问类型。由于 `list` 容器不支持随机访问，因此，在此容器上不能使用需要随机访问迭代器的算法。这些算法包括 `sort` 及其相关的算法。还有一些其他的泛型算法，如 `merge`、`remove`、`reverse` 和 `unique`，虽然可以用在 `list` 上，但却付出了性能上的代价。如果这些算法利用 `list` 容器实现的特点，则可以更高效地执行。

It is possible to write much faster algorithms if the internal structure of the `list` can be exploited. Rather than relying solely on generic operations, the library defines a more elaborate set of operations for `list` than are supported for the other sequential containers. These `list`-specific operations are described in [Table 11.4](#) on the next page. Generic algorithms not listed in the table that take bidirectional or weaker iterators execute equally efficiently on `lists` as on other containers.

如果可以结合利用 `list` 容器的内部结构，则可能编写出更快的算法。与其他顺序容器所支持的操作相比，标准库为 `list` 容器定义了更精细的操作集合，使它不必只依赖于泛型操作。[表 11.4](#) 列出了 `list` 容器特有的操作，其中不包括要求支持双向或更弱的迭代器类型的泛型算法，这类泛型算法无论是用在 `list` 容器上，还是用在其他容器上，都具有相同的效果。

Table 11.4. `list`-Specific Operations

表 11.4. `list` 容器特有的操作

```
lst.merge(lst2) lst.merge(lst2, comp)
```

Merges elements from `lst2` onto `lst`. Both lists must be sorted. Elements are removed from `lst2`. After the `merge`, `lst2` is empty. Returns `void`. The first version uses the `<` operator; the second version uses the specified comparison.

将 `lst2` 的元素合并到 `lst` 中。这两个 `list` 容器对象都必须排序。`lst2` 中的元素将被删除。合并后，`lst2` 为空。返回 `void` 类型。第一个版本使用 `<` 操作符，而第二个版本则使用 `comp` 指定的比较运算

```
lst.remove(val) lst.remove_if(unaryPred)
```

Removes, by calling `lst.erase`, each element that equals a specified value or for which the specified predicate returns a nonzero value. Returns `void`.

调用 `lst.erase` 删除所有等于指定值或使指定的谓词函数返回非零值的元素。返回 `void` 类型

```
lst.reverse()
```

Reverses the order of the elements in `lst`.

反向排列 `lst` 中的元素

```
lst.sort
```

Sorts the elements of the `lst`.

对 `lst` 中的元素排序

```
lst.splice(iter, lst2)
```

Moves element(s) from `lst2` into `lst` just before the element (in `lst`) referred to by the iterator `iter`. Removes element(s) that are moved from `lst2`. The first version moves all elements from `lst2` into `lst`; after the `splice`, `lst2` is empty. `lst` and `lst2` may not be the same `list`. The second version moves only the element referred to by `iter2`, which must refer to an element in `lst2`. In this case, `lst2` and `lst` could be the same `list`. That is, `splice` can be used to move an element within a single `list`. The third version moves the elements in the range denoted by

Section 11.5. Container-Specific Algorithms

the iterators `beg` and `end`. As usual, `beg` and `end` must refer to a valid range. The iterators can refer to a range in any `list`, including `lst`. If the iterators refer to `lst`, the operation is undefined if `iter` refers to an element in the range.

将 `lst2` 的元素移到 `lst` 中迭代器 `iter` 指向的元素前面。在 `lst2` 中删除移出的元素。第一个版本将 `lst2` 的所有元素移到 `lst` 中；合并后，`lst2` 为空。`lst` 和 `lst2` 不能是同一个 `list` 对象。第二个版本只移动 `iter2` 所指向的元素，这个元素必须是 `lst2` 中的元素。在这种情况下，`lst` 和 `lst2` 可以是同一个 `list` 对象。也就是说，可在同一个 `list` 对象中使用 `splice` 运算移动一个元素。第三个版本移动迭代器 `beg` 和 `end` 标记的范围内的元素。`beg` 和 `end` 照例必须指定一个有效的范围。这两个迭代器可标记任意 `list` 对象内的范围，包括 `lst`。当它们指定 `lst` 的一段范围时，如果 `iter` 也指向这个范围的一个元素，则该运算未定义。

```
lst.unique() lst.unique(binaryPred)
```

Deletes, by calling `erase`, consecutive copies of the same value. The first version uses `==` to determine if elements are equal; the second uses the specified predicate.

调用 `erase` 删除同一个值的团结副本。第一个版本使用 `==` 操作符判断元素是否相等；第二个版本则使用指定的谓词函数实现判断



The `list` member versions should be used in preference to the generic algorithms when applied to a `list` object.

对于 `list` 对象，应该优先使用 `list` 容器特有的成员版本，而不是泛型算法。

Most of the `list`-specific algorithms are similar but not identical to their counterparts that we have already seen in their generic forms:

大多数 `list` 容器特有的算法类似于其泛型形式中已经见过的相应的算法，但并不相同：

```
l.remove(val);      // removes all instances of val from l
l.remove_if(pred); // removes all instances for which pred is true from l
l.reverse();        // reverses the order of elements in l
l.sort();           // use element type < operator to compare elements
l.sort(comp);       // use comp to compare elements
l.unique();         // uses element == to remove adjacent duplicates
l.unique(comp);    // uses comp to remove duplicate adjacent copies
```

There are two crucially important differences between the `list`-specific operations and their generic counterparts. One difference is that the `list` versions of `remove` and `unique` change the underlying container; the indicated elements are actually removed. For example, second and subsequent duplicate elements are removed from the list by `list::unique`.

`list` 容器特有的算法与其泛型算法版本之间有两个至关重要的差别。其中一个差别是 `remove` 和 `unique` 的 `list` 版本修改了其关联的基础容器：真正删除了指定的元素。例如，`list::unique` 将 `list` 中第二个和后续重复的元素删除出该容器。



Unlike the corresponding generic algorithms, the `list`-specific operations do add and remove elements.

与对应的泛型算法不同，`list` 容器特有的操作能添加和删除元素。

The other difference is that the `list` operations, `merge` and `splice`, are destructive on their arguments. When we use the generic version of `merge`, the merged sequence is written to a destination iterator, and the two input sequences are left unchanged. In the case of the `merge` function that is a member of `list`, the argument `list` is destroyed—elements are moved from the argument and removed as they are merged into the `list` object on which `merge` was called.

另一个差别是 `list` 容器提供的 `merge` 和 `splice` 运算会破坏它们的实参。使用 `merge` 的泛型算法版本时，合并的序列将写入目标迭代器指向的对象，而它的两个输入序列保持不变。但是，使用 `list` 容器的 `merge` 成员函数时，则会破坏它的实参 `list` 对象——当实参对象的元素合并到调用 `merge` 函数的 `list` 对象时，实参对象的元素被移出并删除。

Exercises Section 11.5

Exercise Reimplement the program that eliminated duplicate words that we wrote in [Section 11.2.3](#) (p.

11.29: [400](#)) to use a `list` instead of a `vector`.

用 `list` 容器取代 `vector` 重新实现eliminated duplicate words that we wrote in [第 11.2.3 节](#)编写的排除重复单词的程序。

Chapter Summary

小结

One of the more important contributions from the standardization process for C++ was the creation and expansion of the standard library. The containers and algorithms libraries are a cornerstone of the standard library. The library defines more than 100 algorithms. Fortunately, the algorithms have a consistent architecture, which makes learning and using them easier.

C++ 标准化过程做出的更重要的贡献之一是：创建和扩展了标准库。容器和算法库是标准库的基础。标准库定义了超过一百个算法。幸运的是，这些算法具有相同的结构，使它们更容易学习和使用。

The algorithms are type independent: They generally operate on a sequence of elements that can be stored in a library container type, a built-in array, or even a generated sequence, such as by reading or writing to a stream. Algorithms achieve their type independence by operating in terms of iterators. Most algorithms take a pair of iterators denoting a range of elements as the first two arguments. Additional iterator arguments might include an output iterator denoting a destination, or another iterator or iterator pair denoting a second input sequence.

算法与类型无关：它们通常在一个元素序列上操作，这些元素可以存储在标准库容器类型、内置数组甚至是生成的序列（例如读写流所生成的序列）上。算法基于迭代器操作，从而实现类型无关性。大多数算法使用一对指定元素范围的迭代器作为其头两个实参。其他的迭代器实参包括指定输出目标的输出迭代器，或者用于指定第二个输入序列的另一个或一对迭代器。

Iterators can be categorized by the operations that they support. There are five iterator categories: input, output, forward, bidirectional, and random access. An iterator belongs to a particular category if it supports the operations required for that iterator category.

迭代器可通过其所支持的操作来分类。标准库定义了五种迭代器类别：输入、输出、前向、双向和随机访问迭代器。如果一个迭代器支持某种迭代器类别要求的运算，则该迭代器属于这个迭代器类别。

Just as iterators are categorized by their operations, iterator parameters to the algorithms are categorized by the iterator operations they require. Algorithms that only read their sequences often require only input iterator operations. Those that write to a destination iterator often require only the actions of an output iterator, and so on.

正如迭代器根据操作来分类一样，算法的迭代器形参也通过其所要求的迭代器操作来分类。只需要读取其序列的算法通常只要求输入迭代器的操作。而写目标迭代器的算法则通常只要求输出迭代器的操作，依此类推。

Algorithms that look for a value often have a second version that looks for an element for which a predicate returns a nonzero value. For such algorithms, the name of the second version has the suffix `_if`. Similarly, many algorithms provide so-called copying versions. These write the (transformed) elements to an output sequence rather than writing back into the input range. Such versions have names that end with `_copy`.

查找某个值的算法通常提供第二个版本，用于查找使谓词函数返回非零值的元素。对于这种算法，第二个版本的函数名字以 `_if` 后缀标识。类似地，很多算法提供所谓的复制版本，将（修改过的）元素写到输出序列，而不是写回输入范围。这种版本的名字以 `_copy` 结束。

A third pattern is whether algorithms read, write, or reorder elements. Algorithms never directly change the size of the sequences on which they operate. (If an argument is an insert iterator, then that iterator might add elements, but the algorithm does not do so directly.) They may copy elements from one position to another but cannot directly add or remove elements.

第三种模式是考虑算法是否对元素读、写或者重新排序。算法从不直接改变它所操纵的序列的大小。（如果算法的实参是插入迭代器，则该迭代器会添加新元素，但算法并不直接这么做。）算法可以从一个位置将元素复制到另一个位置，但不直接添加或删除元素。

Defined Terms

术语

back inserter

Iterator adaptor that takes a reference to a container and generates an insert iterator that uses `push_back` to add elements to the specified container.

形参为指向容器的引用的迭代器适配器，生成使用 `push_back` 为指定容器添加元素的插入迭代器。

bidirectional iterator (双向迭代器)

Same operations as forward iterators plus the ability to use to move backward through the sequence.

除了提供前向迭代器相同的操作之外，还支持使用——操作符向后遍历序列。

forward iterator (前向迭代器)

Iterator that can read and write elements, but does not support `--`.

可读写元素的迭代器，但不支持——操作符。

front inserter

Iterator adaptor that given a container, generates an insert iterator that uses `push_front` to add elements to the beginning of that container.

一种迭代器适配器，生成使用 `push_front` 在指定容器的开始位置添加新元素的插入迭代器。

generic algorithms (泛型算法)

Type-independent algorithms.

与类型无关的算法。

input iterator (输入迭代器)

Iterator that can read but not write elements.

只能读不能写元素的迭代器。

insert iterator (插入迭代器)

Iterator that uses a container operation to insert elements rather than overwrite them. When a value is assigned to an insert iterator, the effect is to insert the element with that value into the sequence.

使用容器操作插入元素而不是覆写元素的迭代器。给插入迭代器赋值，等效于将具有所赋值的新元素插入到序列中。

inserter (插入器)

Iterator adaptor that takes an iterator and a reference to a container and generates an insert iterator that uses `insert` to add elements just ahead of the element referred to by the given iterator.

一种迭代器适配器，形参为一个迭代器和一个指向容器的引用，生成使用 `insert` 为容器添加元素的插入迭代器，新元素插入在该适配器的迭代器形参所指向的元素前面。

istream iterator

Stream iterator that reads an input stream.

读输入流的流迭代器。

iterator categories (迭代器种类)

Conceptual organization of iterators based on the operations that an iterator supports. Iterator categories form a hierarchy, in which the more powerful categories offer the same operations as the lesser categories. The algorithms use iterator categories to specify what operations the iterator arguments must support. As long as the iterator provides at least that level of operation, it can be used. For Example, some algorithms require only input iterators. Such algorithms can be called on any iterator other than one that meets only the output iterator requirements. Algorithms that require random-access iterators can be used only on iterators that support random-access operations.

基于迭代器所支持的操作，在概念上对迭代器进行分类。迭代器种类形成了一个层次结构，功能较强的迭代器种类提供比它弱的迭代器的所有操作。算法使用迭代器种类来指定它的迭代器实参必须支持什么操作。只要迭代器至少提供这个层次的操作，就可以用于该算法。例如，一些算法只要求输入迭代器，则可以使用除了输出迭代器之外的任意迭代器调用这样的算法。而要求使用随机访问迭代器的算法只能用在支持随机访问运算的迭代器上。

off-the-end iterator (超出末端迭代器)

An iterator that marks the end of a range of elements in a sequence. The off-the-end iterator is used as a sentinel and refers to an element one past the last element in the range. The off-the-end iterator may refer to a nonexistent element, so it must never be dereferenced.

一种迭代器，用于标记序列中一个元素范围的结束位置。超出末端迭代器用作结束遍历的“哨兵”，指向范围内最后一个元素的下一位置。超出末端迭代器可能指向不存在的元素，因此永远不能做解引用运算。

ostream iterator

Iterator that writes to an output stream.

写输出流的迭代器。

output iterator (输出迭代器)

Iterator that can write but not read elements.

只能写不能读元素的迭代器。

predicate (谓词)

Function that returns a type that can be converted to `bool`. Often used by the generic algorithms to test elements. Predicates used by the library are either unary (taking one argument) or binary (taking two).

其返回类型可转换为 `bool` 值的函数。通常被泛型算法用于检查元素。标准库所使用的谓词函数不是一元（需要一个实参）的就是二元的（需要两个实参）。

random-access iterator (随机访问迭代器)

Same operations as bidirectional iterators plus the ability to use the relational operators to compare iterator values and the ability to do arithmetic on iterators, thus supporting random access to elements.

除了支持双向迭代器相同的操作之外，还提供了使用关系运算比较迭代器值的能力，以及在迭代器上做算术运算的能力。因此，这类迭代器支持随机访问元素。

reverse iterator (反向迭代器)

Iterator that moves backward through a sequence. These iterators invert the meaning of `++` and `--`.

向后遍历序列的迭代器。这些迭代器颠倒了 `++` 和 `--` 的含义。

stream iterator (流迭代器)

Iterator that can be bound to a stream.

可与流绑定在一起的迭代器。

Chapter 12. Classes

第十二章 类

CONTENTS

目录

Section 12.1	Class Definitions and Declarations	430
Section 12.2	The Implicit <code>this</code> Pointer	440
Section 12.3	Class Scope	444
Section 12.4	Constructors	451
Section 12.5	Friends	461
Section 12.6	<code>static</code> Class Members	461
Chapter Summary		473
Defined Terms		473

In C++ we use classes to define our own **abstract data types**. By defining types that mirror concepts in the problems we are trying to solve, we can make our programs easier to write, debug, and modify.

在 C++ 中，用类来定义自己的抽象数据类型 (abstract data types)。通过定义类型来对应所要解决的问题中的各种概念，可以使我们更容易编写、调试和修改程序。

This chapter continues the coverage of classes begun in [Chapters 2](#) and [5](#). We'll cover in more detail the importance of data abstraction, which lets us hide the internal representation of an object while still allowing `public` operations to be performed on the object.

本章进一步讨论类，并将更详细地阐述数据抽象的重要性。数据抽象能够隐藏对象的内部表示，同时仍然允许执行对象的公有（public）操作。

We'll also explain more about class scope, constructors, and the `this` pointer. We also introduce three new class-related features: `friends`, and `mutable` and `static` members.

我们也将进一步解释类作用域、构造函数以及 `this` 指针。此外，还要介绍与类有关的三个新特征：友元 (`friend`)、可变成员 (`mutable`)、和静态成员 (`static`)。

Classes are the most important feature in C++. Early versions of the language were named “C with Classes,” emphasizing the central role of the class facility. As the language evolved, support for building classes increased. A primary goal of the language design has been to provide features that allow programmers to define their own types that are as easy and intuitive to use as the built-in types. This chapter presents many of the basic features of classes.

是 C++ 最重要的特征。C++ 语言的早期版本被命名为“带类的 C (C With Classes)”，以强调类机制对于工作的作用。随着语言的发展，创建类的配套设施也在不断增加。语言设计的主要目标也变成提供这样一些特性：允许程序定义自己的类型，它们用起来与内置类型一样容易和直观。本章将介绍类的许多基本特征。

12.1. Class Definitions and Declarations

12.1. 类的定义和声明

Starting from [Chapter 1](#), our programs have used classes. The library types we've used `vector`, `istream`, `string` are all class types. We've also defined some simple classes of our own, such as the `Sales_item` and `TextQuery` classes. To recap, let's look again at the `Sales_item` class:

从第一章开始，程序中就已经使用了类。已经用过的标准库类型，比如 `vector`, `istream` 和 `string`，都是类类型。还定义了一些简单的类，如 `Sales_item` 和 `TextQuery` 类。为了扼要袜，再来看年 `Sales_item` 类：

```
class Sales_item {
public:
    // operations on Sales_item objects
    double avg_price() const;
    bool same_isbn(const Sales_item &rhs) const
        { return isbn == rhs.isbn; }
    // default constructor needed to initialize members of built-in type
    Sales_item(): units_sold(0), revenue(0.0) { }
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};

double Sales_item::avg_price() const
{
    if (units_sold)
        return revenue/units_sold;
    else
        return 0;
}
```

12.1.1. Class Definitions: A Recap

12.1.1. 类定义：扼要重述

In writing this class in [Section 2.8](#) (p. 63) and [Section 7.7](#) (p. 258), we already learned a fair bit about classes.

在第 2.8 节和第 7.7 节中编写这个类时，已经学习了有关类的一些知识。



Most fundamentally, a class defines a new type and a new scope.

最简单地说，类就是定义了一个新的类型和一个新作用域。

Class Members

类成员

Each class defines zero or more members. Members can be either data, functions, or type definitions.

每个类可以没有成员，也可以定义多个成员，成员可以是数据、函数或类型别名。

A class may contain multiple `public`, `private`, and `protected` sections. We've already used the `public` and `private` access labels: Members defined in the `public` section are accessible to all code that uses the type; those defined in the `private` section are accessible to other class members. We'll have more to say about `protected` when we discuss inheritance in [Chapter 15](#).

一个类可以包含若干公有的、私有的和受保护的部分。我们已经使用过 `public` 和 `private` 访问标号：在 `public` 部分定义的成员可被使用该类型的所有代码访问；在 `private` 部分定义的成员可被其他类成员访问。在[第十五章](#)讨论继承时将进一步探讨 `protected`。

Section 12.1. Class Definitions and Declarations

All members must be declared inside the class; there is no way to add members once the class definition is complete.

所有成员必须在类的内部声明，一旦类定义完成后，就没有任何方式可以增加成员了。

Constructors

构造函数

When we create an object of a class type, the compiler automatically uses a constructor ([Section 2.3.3](#), p. 49) to initialize the object. A constructor is a special member function that has the same name as the class. Its purpose is to ensure that each data member is set to sensible initial values.

创建一个类类型的对象时，编译器会自动使用一个构造函数（[第 2.3.3 节](#)）来初始化该对象。构造函数是一个特殊的、与类同名的成员函数，用于给每个数据成员设置适当的初始值。

A constructor generally should use a [constructor initializer list](#) ([Section 7.7.3](#), p. 263), to initialize the data members of the object:

构造函数一般就使用一个构造函数初始化列表（[第 7.7.3 节](#)），来初始化对象的数据成员：

```
// default constructor needed to initialize members of built-in type
Sales_item(): units_sold(0), revenue(0.0) { }
```

The constructor initializer list is a list of member names and parenthesized initial values. It follows the constructor's parameter list and begins with a colon.

Member Functions

成员函数

Member functions must be declared, and optionally may be defined, inside the class; functions defined inside the class are [inline](#) ([Section 7.6](#), p. 256) by default.

在类内部，声明成员函数是必需的，而定义成员函数则是可选的。在类内部定义的函数默认为 [inline](#)（[第 7.6 节](#)）。

Member functions defined outside the class must indicate that they are in the scope of the class. The definition of `Sales_item::avg_price` uses the scope operator ([Section 1.2.2](#), p. 8) to indicate that the definition is for the `avg_price` function of the `Sales_item` class.

在类外部定义的成员函数必须指明它们是在类的作用域中。`Sales_item::avg_price` 的定义使用作用域操作符（[第 1.2.2 节](#)）来指明这是 `Sales_item` 类中 `avg_price` 函数的定义。

Member functions take an extra implicit argument that binds the function to the object on behalf of which the function is called when we write

成员函数有一个附加的隐含实参，将函数绑定到调用函数的对象——当我们编写下面的函数时：

```
trans.avg_price()
```

we are calling the `avg_price` function on the object named `trans`. If `trans` is a `Sales_item` object, then references to a member of the `Sales_item` class inside the `avg_price` function are to the members in `trans`.

就是在调用名 `trans` 的对象的 `avg_price` 函数。如果 `trans` 是一个 `Sales_item` 对象，则在 `avg_price` 函数内部对 `Sales_item` 类成员引用就是对 `trans` 成员的引用。

Member functions may be declared `const` by putting the `const` keyword following the parameter list:

将关键字 `const` 加在形参表之后，就可以将成员函数声明为常量：

```
double avg_price() const;
```

A `const` member may not change the data members of the object on which it operates. The `const` must appear in both the declaration and definition. It is a compile-time error for the `const` to be indicated on one but not the other.

`const` 成员不能改变其所操作的对象的数据成员。`const` 必须同时出现在声明和定义中，若只出现在其中一处，就会出现一个编译时错误。

Exercises Section 12.1.1

Section 12.1. Class Definitions and Declarations

Exercise

12.1: Write a class named `Person` that represents the name and address of a person. Use a `string` to hold each of these elements.

编写一个名为 `Person` 的类，表示人的名字和地址。使用 `string` 来保存每个元素。

Exercise

12.2: Provide a constructor for `Person` that takes two `strings`.

为 `Person` 提供一个接受两个 `string` 参数的构造函数。

Exercise

12.3: Provide operations to return the name and address. Should these functions be `const`? Explain your choice.

提供返回名字和地址的操作。这些函数应为 `const` 吗？解释你的选择。

Exercise

12.4: Indicate which members of `Person` you would declare as `public` and which you would declare as

`private`. Explain your choice.

指明 `Person` 的哪个成员应声明为 `public`，哪个成员应声明为 `private`。解释你的选择。

12.1.2. Data Abstraction and Encapsulation

12.1.2. 数据抽象和封装

The fundamental ideas behind classes are [data abstraction](#) and [encapsulation](#).

类背后蕴涵的基本思想是[数据抽象](#)和[封装](#)。

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation. The class designer must worry about how a class is implemented, but programmers that use the class need not know about these details. Instead, programmers who use a type need to know only the type's interface; they can think *abstractly* about what the type does rather than concretely about how the type works.

数据抽象是一种依赖于接口和实现分离的编程（和设计）技术。类设计者必须关心类是如何实现的，但使用该类的程序员不必了解这些细节。相反，使用一个类型的程序员仅需了解类型的接口，他们可以抽象地考虑该类型做什么，而不必具体地考虑该类型如何工作。

Encapsulation is a term that describes the technique of combining lower-level elements to form a new, higher-level entity. A function is one form of encapsulation: The detailed actions performed by the function are *encapsulated* in the larger entity that is the function itself. Encapsulated elements hide the details of their implementation—we may call a function but have no access to the statements that it executes. In the same way, a class is an encapsulated entity: It represents an aggregation of several members, and most (well-designed) class types hide the members that implement the type.

封装是一项低层次的元素组合起来的形成新的、高层次实体的技术。函数是封装的一种形式：函数所执行的细节行为被封装在函数本身这个更大的实体中。被封装的元素隐藏了它们的实现细节——可以调用一个函数但不能访问它所执行的语句。同样地，类也是一个封装的实体：它代表若干成员的聚类，大多数（良好设计的）类类型隐藏了实现该类型的成员。

If we think about the library `vector` type, it is an example of both data abstraction and encapsulation. It is abstract in that to use it, we think about its interface about the operations that it can perform. It is encapsulated because we have no access to the details of how the type is represented nor to any of its implementation artifacts. An array, on the other hand, is similar in concept to a `vector` but is neither abstract nor encapsulated. We manipulate an array directly by accessing the memory in which the array is stored.

标准库类型 `vector` 同时具备数据抽象和封装的特性。在使用方面它是抽象的，只需考虑它的接口，即它能执行的操作。它又是封装的，因为我们既无法了解该类型如何表示的细节，也无法访问其任意的实现制品。另一方面，数组在概念上类似于 `vector`，但既不是抽象的，也不是封装的。可以通过访问存放数组的内存来直接操纵数组。

Access Labels Enforce Abstraction and Encapsulation

访问标号实施抽象和封装

In C++ we use [access labels](#) ([Section 2.8](#), p. 65) to define the abstract interface to the class and to enforce encapsulation. A class may contain zero or more access labels:

在 C++ 中，使用[访问标号](#) ([第 2.8 节](#)) 来定义类的抽象接口和实施封装。一个类可以没有访问标号，也可以包含多个访问标号：

Section 12.1. Class Definitions and Declarations

Members defined after a `public` label are accessible to all parts of the program. The data-abstraction view of a type is defined by its [public members](#).

程序的所有部分都可以访问带有 `public` 标号的成员。类型的数据抽象视图由其 [public 成员](#) 定义。

- Members defined after a `private` label are not accessible to code that uses the class. The `private` sections encapsulate (e.g., hide) the implementation from code that uses the type.

使用类的代码不可以访问带有 `private` 标号的成员。`private` 封装了类型的实现细节。

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

一个访问标号可以出现的次数通常是没有限制的。每个访问标号指定了随后的成员定义的访问级别。这个指定的访问级别持续有效，直到遇到下一个访问标号或看到类定义体的右花括号为止。

A class may define members before any access label is seen. The access level of members defined after the open curly of the class and before the first access label depend on how the class is defined. If the class is defined with the [struct keyword](#), then members defined before the first access label are `public`; if the class is defined using the [class keyword](#), then the members are `private`.

可以在任意的访问标号出现之前定义类成员。在类的左花括号之后、第一个访问标号之前定义成员的访问级别，其值依赖于类是如何定义的。如果类是用 `struct` 关键字定义的，则在第一个访问标号之前的成员是公有的；如果类是用 `class` 关键字是定义的，则这些成员是私有的。

Advice: Concrete and Abstract Types

建议：具体类型和抽象类型

Not all types need to be abstract. The library `pair` class is a good example of a useful, well-designed class that is concrete rather than abstract. A [concrete class](#) is a class that exposes, rather than hides, its implementation.

并非所有类型都必须是抽象的。标准库中的 `pair` 类就是一个实用的、设计良好的具体类而不是抽象类。[具体类](#)会暴露而非隐藏其实现细节。

Some classes, such as `pair`, really have no abstract interface. The `pair` type exists to bundle two data members into a single object. There is no need or advantage to hiding the data members. Hiding the members in a class like `pair` would only complicate the use of the type.

一些类，例如 `pair`，确实没有抽象接口。`pair` 类型只是将两个数据成员捆绑成单个对象。在这种情况下，隐藏数据成员没有必要也没有明显的好处。在像 `pair` 这样的类中隐藏数据成员只会造成类型使用的复杂化。

Even so, such types often have member functions. In particular, it is a good idea for any class that has data members of built-in or compound type to define constructor(s) to initialize those members. The user of the class could initialize or assign to the data members but it is less error-prone for the class to do so.

尽管如此，这样的类型通常还是有成员函数。特别地，如果类具有内置类型或复合类型数据成员，那么定义构造函数来初始化这些成员就是一个好主意。类的使用都可以初始化或赋值数据成员，但由类来做更不易出错。

Different Kinds of Programming Roles

编程角色的不同类别

Programmers tend to think about the people who will run their applications as "users." Applications are designed for and evolve in response to feedback from those who ultimately "use" the applications. Classes are thought of in a similar way: A class designer designs and implements a class for "users" of that class. In this case, the "user" is a programmer, not the ultimate user of the application.

程序员经常会将运行应用程序的人看作“用户”。应用程序为最终“使用”它的用户而设计，并响应用户的反馈而完善。类也类似：类的设计者为类的“用户”设计并实现类。在这种情况下，“用户”是程序员，而不是应用程序的最终用户。

Authors of successful applications do a good job of understanding and implementing the needs of the application's users. Similarly, well-designed, useful classes are designed with a close attention to the needs of the users of the class.

成功的应用程序的创建者会很好地理解和实现用户的需求。同样地，良好设计的、实用的类，其设计也要贴近类用户的需求。

In another way, the division between class designer and class user reflects the division between users of an application and the designers and implementors of the application. Users care only if the application meets their needs in a cost-effective way. Similarly, users of a class care only about its interface. Good class designers define a class interface that is intuitive and easy to use. Users care about the implementation only in so

Section 12.1. Class Definitions and Declarations

far as the implementation affects their use of the class. If the implementation is too slow or puts burdens on users of the class, then the users must care. In well-designed classes, only the class designer worries about the implementation.

另一方面，类的设计者与实现者之间的区别，也反映了应用程序的用户与设计和实现者之间的区分。用户只关心应用程序能否以合理的费用满足他们的需求。同样地，类的使用者只关心它的接口。好的类设计者会定义直观和易用的类接口，而使用者只关心类中影响他们使用的部分实现。如果类的实现速度太慢或给类的使用者加上负担，则必然引起使用者的关注。在良好设计的类中，只有类的设计者会关心实现。

In simple applications, the user of a class and the designer of the class might be one and the same person. Even in such cases, it is useful to keep the roles distinct. When designing the interface to a class, the class designer should think about how easy it will be to use the class. When using the class, the designer shouldn't think about how the class works.

在简单的应用程序中，类的使用者和设计者也许是同一个人。即使在这种情况下，保持角色区分也是有益的。设计类的接口时，设计者应该考虑的是如何方便类的使用；使用类的时候，设计者就不应该考虑类如何工作。

C++ programmers tend to speak of "users" interchangably as users of the application or users of a class.

注意，C++ 程序员经常会将应用程序的用户和类的使用者都称为“用户”。



When referring to a "user," the context makes it clear which kind of user is meant. If we speak of "user code" or the "user" of the `Sales_item` class, we mean a programmer who is using a class in writing an application. If we speak of the "user" of the bookstore application, we mean the manager of the store who is running the application.

提到“用户”时，应该由上下文清楚地标明所指的是哪类用户。如果提到“用户代码”或 `Sales_item` 类的“用户”，指的就是使用类编写应用程序的程序员。如果提到书店应用程序的“用户”，那么指的是运行应用程序的书店管理人员。

Key Concept: Benefits of Data Abstraction and Encapsulation

关键概念：数据抽象和封装的好处

Data abstraction and encapsulation provide two important advantages:

数据抽象和封装提供了两个重要优点：

- **Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.**
避免类内部出现无意的、可能破坏对象状态的用户级错误。
- **The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.**
随时间推移可以根据需求改变或缺陷（bug）报告来完美类实现，而无须改变用户级代码。

By defining data members only in the `private` section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data are `public`, then any function that directly accesses the data members of the old representation might be broken. It would be necessary to locate and rewrite all those portions of code that relied on the old representation before the program could be used again.

仅在类的私有部分定义数据成员，类的设计者就可以自由地修改数据。如果实现改变了，那么只需检查类代码来了解此变化可能造成的影响。如果数据为仅有的，则任何直接访问原有数据成员的函数都可能遭到破坏。在程序可重新使用之前，有必要定位和重写依赖原有表示的那部分代码。

Similarly, if the internal state of the class is `private`, then changes to the member data can happen in only a limited number of places. The data is protected from mistakes that users might introduce. If there is a bug that corrupts the object's state, the places to look for the bug are localized: When data are `private`, only a member function could be responsible for the error. The search for the mistake is limited, greatly easing the problems of maintenance and program correctness.

同样地，如果类的内部状态是私有的，则数据成员的改变只可能在有限的地方发生。避免数据中出现用户可能引入的错误。如果有缺陷会破坏对象的状态，就在局部位置搜寻缺陷：如果数据是私有的，那么只有成员函数可能对该错误负责。对错误的搜寻是有限的，从而大大方便了程序的维护和修正。

If the data are `private` and if the interface to the member functions does not change, then user functions that

Section 12.1. Class Definitions and Declarations

manipulate class objects require no change.

如果数据是私有的并且没有改变成员函数的接口，则操纵类对象的用户函数无须改变。



Because changing a class definition in a header file effectively changes the text of every source file that includes that header, code that uses a class must be recompiled when the class changes.

改变头文件中的类定义可有效地改变包含该头文件的每个源文件的程序文本，所以，当类发生改变时，使用该类的代码必须重新编译。

Exercises Section 12.1.2

Exercise

12.5: What are the access labels supported by C++ classes? What kinds of members should be defined after each access label? What, if any, are the constraints on where and how often an access label may appear inside a class definition?

C++ 类支持哪些访问标号？在每个访问标号之后应定义哪种成员？如果有的话，在类的定义中，一个访问标号可以出现在何处以及可出现多少次？约束条件是什么？

Exercise

12.6: How do classes defined with the `class` keyword differ from those defined as `struct`?

有 `class` 关键字定义的类和用 `struct` 定义的类有什么不同？

Exercise

12.7: What is encapsulation? Why it is useful?

什么是封装？为什么封装是有用的？

12.1.3. More on Class Definitions

12.1.3. 关于类定义的更多内容

The classes we've defined so far have been simple; yet they have allowed us to explore quite a bit of the language support for classes. There remain a few more details about the basics of writing a class that we shall cover in the remainder of this section.

迄今为止，所定义的类都是简单的，然而通过这些类我们已经了解到 C++ 语言为类所提供的相当多的支持。本节的其余部分将阐述编写类的更多基础知识。

Multiple Data Members of the Same Type

同一类型的多个数据成员

As we've seen, class data members are declared similarly to how ordinary variables are declared. One way in which member declarations and ordinary declarations are the same is that if a class has multiple data members with the same type, these members can be named in a single member declaration.

正如我们所见，类的数据成员的声明类似于普通变量的声明。如果一个类具有多个同一类型的数据成员，则这些成员可以在一个成员声明中指定，这种情况下，成员声明和普通变量声明是相同的。

For example, we might define a type named `Screen` to represent a window on a computer. Each `Screen` would have a `string` member that holds the contents of the window, and three `string::size_type` members: one that specifies the character on which the cursor currently rests, and two others that specify the height and width of the window. We might define the members of this class as:

例如，可以定义一个名为 `Screen` 的类型表示计算机上的窗口。每个 `Screen` 可以有一个保存窗口内容的 `string` 成员，以及三个 `string::size_type` 成员：一个指定光标当前停留的字符，另外两个指定窗口的高度和宽度。可以用如下方式这个类的成员：

Section 12.1. Class Definitions and Declarations

```
class Screen {
public:
    // interface member functions
private:
    std::string contents;
    std::string::size_type cursor;
    std::string::size_type height, width;
};
```

Using Typedefs to Streamline Classes

使用类型别名来简化类

In addition to defining data and function members, a class can also define its own local names for types. Our `Screen` will be a better abstraction if we provide a `typedef` for `std::string::size_type`:

除了定义数据和函数成员之外，类还可以定义自己的局部类型名字。如果为 `std::string::size_type` 提供一个类型别名，那么 `Screen` 类将是一个更好的抽象：

```
class Screen {
public:
    // interface member functions
    typedef std::string::size_type index;
private:
    std::string contents;
    index cursor;
    index height, width;
};
```

Type names defined by a class obey the standard access controls of any other member. We put the definition of `index` in the `public` part of the class because we want users to use that name. Users of class `Screen` need not know that we use a `string` as the underlying implementation. By defining `index`, we hide this detail of how `Screen` is implemented. By making the type `public`, we let our users use this name.

类所定义的类型名遵循任何其他成员的标准访问控制。将 `index` 的定义放在类的 `public` 部分，是因为希望用户使用这个名字。`Screen` 类的使用者不必了解用 `string` 实现的底层细节。定义 `index` 来隐藏 `Screen` 的实现细节。将这个类型设为 `public`，就允许用户使用这个名字。

Member Functions May Be Overloaded

成员函数可被重载

Another way our classes have been simple is that they have defined only a few member functions. In particular, none of our classes have needed to define over-loaded versions of any of their member functions. However, as with nonmember functions, a member function may be overloaded ([Section 7.8, p. 265](#)).

这些类之所以简单，另一个方面也是因为它们只定义了几个成员函数。特别地，这些类都不需要定义其任意成员函数的重载版本。然而，像非成员函数一样，成员函数也可以被重载（[第 7.8 节](#)）。

With the exception of overloaded operators ([Section 14.9.5, p. 547](#)) which have special rules, a member function overloads only other member functions of its own class. A class member function is unrelated to, and cannot overload, ordinary nonmember functions or functions declared in other classes. The same rules apply to overloaded member functions as apply to plain functions: Two overloaded members cannot have the same number and types of parameters. The function-matching ([Section 7.8.2, p. 269](#)) process used for calls of nonmember overloaded functions also applies to calls of overloaded member functions.

重载操作符（[第 14.9.5 节](#)）有特殊规则，是个例外，成员函数只能重载本类的其他成员函数。类的成员函数与普通的非成员函数以及在其他类中声明的函数不相关，也不能重载它们。重载的成员函数和普通函数应用相同的规则：两个重载成员的形参数量和类型不能完全相同。调用非成员重载函数所用到的函数匹配（[第 7.8.2 节](#)）过程也应用于重载成员函数的调用。

Defining Overloaded Member Functions

定义重载成员函数

To illustrate overloading, we might give our `Screen` class two overloaded members to return a given character from the window. One version will return the character currently denoted by the cursor and the other returns the character at a given row and column:

为了举例说明重载，可以给出 `Screen` 类的两个重载成员，用于从窗口返回一个特定字符。两个重载成员中，一个版本返回由当前光标指示的字符，另一个返回指定行列处的字符：

```
class Screen {
```

Section 12.1. Class Definitions and Declarations

```
public:
    typedef std::string::size_type index;
    // return character at the cursor or at a given position
    char get() const { return contents[cursor]; }
    char get(index ht, index wd) const;
    // remaining members
private:
    std::string contents;
    index cursor;
    index height, width;
};
```

As with any overloaded function, we select which version to run by supplying the appropriate number and/or types of arguments to a given call:

与任意的重载函数一样，给指定的函数调用提供适当数目和／或类型的实参来选择运行哪个版本：

```
Screen myscreen;
char ch = myscreen.get(); // calls Screen::get()
ch = myscreen.get(0,0); // calls Screen::get(index, index)
```

Explicitly Specifying `inline` Member Functions

显式指定 `inline` 成员函数

Member functions that are defined inside the class, such as the `get` member that takes no arguments, are automatically treated as `inline`. That is, when they are called, the compiler will attempt to expand the function inline ([Section 7.6](#), p. 256). We can also explicitly declare a member function as `inline`:

在类内部定义的成员函数，例如不接受实参的 `get` 成员，将自动作为 `inline` 处理。也就是说，当它们被调用时，编译器将试图在同一行内扩展该函数（[第 7.6 节](#)）。也可以显式地将成员函数声明为 `inline`：

```
class Screen {
public:
    typedef std::string::size_type index;
    // implicitly inline when defined inside the class declaration
    char get() const { return contents[cursor]; }
    // explicitly declared as inline; will be defined outside the class declaration
    inline char get(index ht, index wd) const;
    // inline not specified in class declaration, but can be defined inline later
    index get_cursor() const;
    // ...
};

// inline declared in the class declaration; no need to repeat on the definition
char Screen::get(index r, index c) const
{
    index row = r * width; // compute the row location
    return contents[row + c]; // offset by c to fetch specified character
}
// not declared as inline in the class declaration, but ok to make inline in definition
inline Screen::index Screen::get_cursor() const
{
    return cursor;
}
```

We can specify that a member is `inline` as part of its declaration inside the class body. Alternatively, we can specify `inline` on the function definition that appears outside the class body. It is legal to specify `inline` both on the declaration and definition. One advantage of defining `inline` functions outside the class is that it can make the class easier to read.

可以在类定义体内部指定一个成员为 `inline`，作为其声明的一部分。或者，也可以在类定义外部的函数定义上指定 `inline`。在声明和定义处指定 `inline` 都是合法的。在类的外部定义 `inline` 的一个好处是可以使得类比较容易阅读。



As with other `inlines`, the definition of an `inline` member function must be visible in every source file that calls the function. The definition for an `inline` member function that is not defined within the class body ordinarily should be placed in the same header file in which the class definition appears.

像其他 `inline` 一样，`inline` 成员函数的定义必须在调用该函数的每个源文件中是可见的。不在类定义体内定义的 `inline` 成员函数，其定义通常应放在有类定义的同一头文件中。

Exercises Section 12.1.3

Exercise 12.8: Define `Sales_item::avg_price` as an inline function.

将 `Sales_item::avg_price` 定义为内联函数。

Exercise 12.9: Write your own version of the `Screen` class presented in this section, giving it a constructor to create a `Screen` from values for height, width, and the contents of the screen.

修改本节中给出的 `Screen` 类，给出一个构造函数，根据屏幕的高度、宽度和内容的值来创建 `Screen`。

Exercise 12.10: Explain each member in the following class:

解释下述类中的每个成员：

```
class Record {
    typedef std::size_t size;
    Record(): byte_count(0) { }
    Record(size s): byte_count(s) { }
    Record(std::string s): name(s), byte_count(0) { }
    size byte_count;
    std::string name;
public:
    size get_count() const { return byte_count; }
    std::string get_name() const { return name; }
};
```

12.1.4. Class Declarations versus Definitions

12.1.4. 类声明与类定义

A class is completely defined once the closing curly brace appears. Once the class is defined, all the class members are known. The size required to store an object of the class is known as well. A class may be defined only once in a given source file. When a class is defined in multiple files, the definition in each file must be identical.

一旦遇到右花括号，类的定义就结束了。并且一旦定义了类，那以我们就知道了所有的类成员，以及存储该类的对象所需的存储空间。在一个给定的源文件中，一个类只能被定义一次。如果在多个文件中定义一个类，那么每个文件中的定义必须是完全相同的。

By putting class definitions in header files, we can ensure that a class is defined the same way in each file that uses it. By using header guards ([Section 2.9.2, p. 69](#)), we ensure that even if the header is included more than once in the same file, the class definition will be seen only once.

将类定义在头文件中，可以保证在每个使用类的文件中以同样的方式定义类。使用头文件保护符 (header guard) ([第 2.9.2 节](#))，来保证即使头文件在同一文件中被包含多次，类定义也只出现一次。

It is possible to declare a class without defining it:

可以声明一个类而不定义它：

```
class Screen; // declaration of the Screen class
```

This declaration, sometimes referred to as a **forward declaration**, introduces the name `Screen` into the program and indicates that `Screen` refers to a class type. After a declaration and before a definition is seen, the type `Screen` is an **incomplete type**—it's known that `Screen` is a type but not known what members that type contains.

这个声明，有时称为前向声明 (**forward declaration**)，在程序中引入了类类型的 `Screen`。在声明之后、定义之前，类 `Screen` 是一个不完全类型 (**incomplete type**)，即已知 `Screen` 是一个类型，但不知道包含哪些成员。



An **incomplete type** can be used only in limited ways. Objects of the type may not be defined. An incomplete type may be used to define only pointers or references to the type or to declare (but not define) functions that use the type as a parameter or return type.

不完全类型 (incomplete type) 只能以有限方式使用。不能定义该类型的对象。不完全类型只能用于定义指向该类型的指针及引用，或者用于声明 (而不是定义) 使用该类型作为形参类型或返回类型的函数。

A class must be fully defined before objects of that type are created. The class must be defined and not just declared so that the compiler can know how much storage to reserve for an object of that class type. Similarly, the class must be defined before a reference or pointer is used to access a member of the type.

在创建类的对象之前，必须完整地定义该类。必须定义类，而不只是声明类，这样，编译器就会给类的对象预定相应的存储空间。同样地，在使用引用或指针访问类的成员之前，必须已经定义类。

Using Class Declarations for Class Members

为类的成员使用类声明

A data member can be specified to be of a class type only if the definition for the class has already been seen. If the type is incomplete, a data member can be only a pointer or a reference to that class type.

只有当类定义已经在前面出现过，数据成员才能被指定为该类类型。如果该类型是不完全类型，那么数据成员只能是指向该类类型的指针或引用。

Because a class is not defined until its class body is complete, a class cannot have data members of its own type. However, a class is considered declared as soon as its class name has been seen. Therefore, a class can have data members that are pointers or references to its own type:

因为只有当类定义体完成后才能定义类，因此类不能具有自身类型的数据成员。然而，只要类名一出现就可以认为该类已声明。因此，类的数据成员可以是指向自身类型的指针或引用：

```
class LinkScreen {
    Screen window;
    LinkScreen *next;
    LinkScreen *prev;
};
```

A common use of class forward declarations is to write classes that are mutually dependent on one another. We'll see an example of such usage in [Section 13.4](#) (p. 486).

类的前身声明一般用来编写相互依赖的类。在第 13.4 节中，我们将看到用法的一个例子。

Exercises Section 12.1.4

Exercise 12.11: Define a pair of classes `x` and `y`, in which `x` has a pointer to `y`, and `y` has an object of type `x`.

定义两个类 `x` 和 `y`, `x` 中有一个指向 `y` 的指针，`y` 中有一个 `x` 类型的对象。

Exercise 12.12: Explain the difference between a class declaration and definition. When would you use a class declaration? A class definition?

解释类声明与类定义之间的差异。何时使用类声明？何时使用类定义？

12.1.5. Class Objects

12.1.5. 类对象

When we define a class, we are defining a type. Once a class is defined, we can define objects of that type. Storage is allocated when we define objects, but (ordinarily) not when we define types:

Section 12.1. Class Definitions and Declarations

定义一个类时，也就是定义了一个类型。一旦定义了类，就可以定义该类型的对象。定义对象时，将为其分配存储空间，但（一般而言）定义类型时不进行存储分配：

```
class Sales_item {  
public:  
    // operations on Sales_item objects  
private:  
    std::string isbn;  
    unsigned units_sold;  
    double revenue;  
};
```

defines a new type, but does not allocate any storage. When we define an object

定义了一个新的类型，但没有进行存储分配。当我们定义一个对象

```
Sales_item item;
```

the compiler allocates an area of storage sufficient to contain a `Sales_item` object. The name `item` refers to that area of storage. Each object has its own copy of the class data members. Modifying the data members of `item` does not change the data members of any other `Sales_item` object.

时，编译器分配了足以容纳一个 `Sales_item` 对象的存储空间。`item` 指的就是那个存储空间。每个对象具有自己的类数据成员的副本。修改 `item` 的数据成员不会改变任何其他 `Sales_item` 对象的数据成员。

Defining Objects of Class Type

定义类类型的对象

After a class type has been defined, the type can be used in two ways:

定义了一个类类型之后，可以按以下两种方式使用。

- Using the class name directly as a type name
将类的名字直接用作类型名。
- Specifying the keyword `class` or `struct`, followed by the class name:
指定关键字 `class` 或 `struct`，后面跟着类的名字：

```
Sales_item item1;           // default initialized object of type Sales_item  
class Sales_item item1; // equivalent definition of item1
```

Both methods of referring to a class type are equivalent. The second method is inherited from C and is also valid in C++. The first, more concise form was introduced by C++ to make class types easier to use.

两种引用类类型方法是等价的。第二种方法是从 C 继承而来的，在 C++ 中仍然有效。第一种更为简练，由 C++ 语言引入，使得类类型更容易使用。

Why a Class Definition Ends in a Semicolon

为什么类的定义以分号结束

We noted on page 64 that a class definition ends with a semicolon. A semicolon is required because we can follow a class definition by a list of object definitions. As always, a definition must end in a semicolon:

我们在第 2.8 节中指出，类的定义分号结束。分号是必需的，因为在类定义之后可以接一个对象定义列表。定义必须以分号结束：

```
class Sales_item { /* ... */;  
class Sales_item { /* ... */ } accum, trans;
```

Ordinarily, it is a bad idea to define an object as part of a class definition. Doing so obscures what's happening. It is confusing to readers to combine definitions of two different entities—the class and a variable—in a single statement.



通常，将对象定义成类定义的一部分是个坏主意。这样做，会使所发生的操作难以理解。对读者而言，将两个不同的实体（类和变量）组合在一个语句中，也会令人迷惑不解。

12.2. The Implicit `this` Pointer

12.2. 隐含的 `this` 指针

As we saw in [Section 7.7.1](#) (p. 260), member functions have an extra implicit parameter that is a pointer to an object of the class type. This implicit parameter is named `this`, and is bound to the object on which the member function is called. Member functions may not define the `this` parameter; the compiler does so implicitly. The body of a member function may explicitly use the `this` pointer, but is not required to do so. The compiler treats an unqualified reference to a class member as if it had been made through the `this` pointer.

在[第 7.7.1 节](#)中已经提到，成员函数具有一个附加的隐含形参，即指向该类对象的一个指针。这个隐含形参命名为 `this`，与调用成员函数的对象绑定在一起。成员函数不能定义 `this` 形参，而是由编译器隐含地定义。成员函数的函数体可以显式使用 `this` 指针，但不是必须这么做。如果对类成员的引用没有限定，编译器会将这种引用处理成通过 `this` 指针的引用。

When to Use the `this` Pointer

何时使用 `this` 指针

Although it is usually unnecessary to refer explicitly to `this` inside a member function, there is one case in which we must do so: when we need to refer to the object as a whole rather than to a member of the object. The most common case where we must use `this` is in functions that return a reference to the object on which they were invoked.

尽管在成员函数内部显式引用 `this` 通常是不必要的，但有一种情况下必须这样做：当我们需要将一个对象作为整体引用而不是引用对象的一个成员时。最常见的情况是在这样的函数中使用 `this`：该函数返回对调用该函数的对象的引用。

The `Screen` class is a good example of the kind of class that might have operations that should return references. So far our class has only a pair of `get` operations. We might logically add:

某种类可能具有某些操作，这些操作应该返回引用，`Screen` 类就是这样的一个类。迄今为止，我们的类只有一对 `get` 操作。逻辑上，我们可以添加下面的操作。

- A pair of `set` operations to set either a specified character or the character denoted by the cursor to a given value
一对 `set` 操作，将特定字符或光标指向的字符设置为给定值。
- A `move` operation that, given two `index` values, moves the `cursor` to that new position
一个 `move` 操作，给定两个 `index` 值，将光标移至新位置。

Ideally, we'd like users to be able to concatenate a sequence of these actions into a single expression:

理想情况下，希望用户能够将这些操作的序列连接成一个单独的表达式：

```
// move cursor to given position, and set that character
myScreen.move(4,0).set('#');
```

We'd like this statement to be equivalent to

这个语句等价于：

```
myScreen.move(4,0);
myScreen.set('#');
```

Returning `*this`

返回 `*this`

To allow us to call `move` and `set` in a single expression, each of our new operations must return a reference to the object on which it executes:

在单个表达式中调用 `move` 和 `set` 操作时，每个操作必须返回一个引用，该引用指向执行操作的那个对象：

Section 12.2. The Implicit this Pointer

```
class Screen {
public:
    // interface member functions
    Screen& move(index r, index c);
    Screen& set(char);
    Screen& set(index, index, char);
    // other members as before
};
```

Notice that the return type of these functions is `Screen&`, which indicates that the member function returns a reference to an object of its own class type. Each of these functions returns the object on which it was invoked. We'll use the `this` pointer to get access to the object. Here is the implementation for two of our new members:

注意，这些函数的返回类型是 `Screen&`，指明该成员函数返回对其自身类类型的对象的引用。每个函数都返回调用自己的那个对象。使用 `this` 指针来访问该对象。下面是对两个新成员的实现：

```
Screen& Screen::set(char c)
{
    contents[cursor] = c;
    return *this;
}
Screen& Screen::move(index r, index c)
{
    index row = r * width; // row location
    cursor = row + c;
    return *this;
}
```

The only interesting part in this function is the `return` statement. In each case, the function returns `*this`. In these functions, `this` is a pointer to a non`const` `Screen`. As with any pointer, we can access the object to which `this` points by dereferencing the `this` pointer.

函数中唯一需要关注的部分是 `return` 语句。在这两个操作中，每个函数都返回 `*this`。在这些函数中，`this` 是一个指向非常量 `Screen` 的指针。如同任意的指针一样，可以通过对 `this` 指针解引用访问 `this` 指向的对象。

Returning `*this` from a `const` Member Function

从 `const` 成员函数返回 `*this`

In an ordinary non`const` member function, the type of `this` is a `const` pointer ([Section 4.2.5](#), p. 126) to the class type. We may change the value to which `this` points but cannot change the address that `this` holds. In a `const` member function, the type of `this` is a `const` pointer to a `const` class-type object. We may change neither the object to which `this` points nor the address that `this` holds.

在普通的非 `const` 成员函数中，`this` 的类型是一个指向类类型的 `const` 指针 ([第 4.2.5 节](#))。可以改变 `this` 所指向的值，但不能改变 `this` 所保存的地址。在 `const` 成员函数中，`this` 的类型是一个指向 `const` 类型对象的 `const` 指针。既不能改变 `this` 所指向的对象，也不能改变 `this` 所保存的地址。



We cannot return a plain reference to the class object from a `const` member function. A `const` member function may return `*this` only as a `const` reference.

不能从 `const` 成员函数返回指向类对象的普通引用。`const` 成员函数只能返回 `*this` 作为一个 `const` 引用。

As an example, we might add a `display` operation to our `Screen` class. This function should print `contents` on a given `ostream`. Logically, this operation should be a `const` member. Printing the `contents` doesn't change the object. If we make `display` a `const` member of `Screen`, then the `this` pointer inside `display` will be a `const Screen* const`.

例如，我们可以给 `Screen` 类增加一个 `display` 操作。这个函数应该在给定的 `ostream` 上打印 `contents`。逻辑上，这个操作应该是一个 `const` 成员。打印 `contents` 不会改变对象。如果将 `display` 作为 `Screen` 的 `const` 成员，则 `display` 内部的 `this` 指针将是一个 `const Screen* const` 型的 `const`。

However, as we can with the `move` and `set` operations, we'd like to be able to use the `display` in a series of actions:

然而，与 `move` 和 `set` 操作一样，我们希望能够在一个操作序列中使用 `display`:

```
// move cursor to given position, set that character and display the screen
myScreen.move(4,0).set('#').display(cout);
```

Section 12.2. The Implicit this Pointer

This usage implies that `display` should return a `Screen` reference and take a reference to an `ostream`. If `display` is a `const` member, then its return type must be `const Screen&`.

这个用法暗示了 `display` 应该返回一个 `Screen` 引用，并接受一个 `ostream` 引用。如果 `display` 是一个 `const` 成员，则它的返回类型必须是 `const Screen&`。

Unfortunately, there is a problem with this design. If we define `display` as a `const` member, then we could call `display` on a `nonconst` object but would not be able to embed a call to `display` in a larger expression. The following code would be illegal:

不幸的是，这个设计存在一个问题。如果将 `display` 定义为 `const` 成员，就可以在非 `const` 对象上调用 `display`，但不能将对 `display` 的调用嵌入到一个长表达式中。下面的代码将是非法的：

```
Screen myScreen;
// this code fails if display is a const member function
// display return a const reference; we cannot call set on a const
myScreen.display().set('*');
```

The problem is that this expression runs `set` on the object returned from `display`. That object is `const` because `display` returns its object as a `const`. We cannot call `set` on a `const` object.

问题在于这个表达式是在由 `display` 返回的对象上运行 `set`。该对象是 `const`，因为 `display` 将其对象作为 `const` 返回。我们不能在 `const` 对象上调用 `set`。

Overloading Based on `const`

基于 `const` 的重载

To solve this problem we must define two `display` operations: one that is `const` and one that isn't. We can overload a member function based on whether it is `const` for the same reasons that we can overload a function based on whether a pointer parameter points to `const` ([Section 7.8.4](#), p. 275). A `const` object will use only the `const` member. A `nonconst` object could use either member, but the `nonconst` version is a better match.

为了解决这个问题，我们必须定义两个 `display` 操作：一个是 `const`，另一个不是 `const`。基于成员函数是否为 `const`，可以重载一个成员函数；同样地，基于一个指针形参是否指向 `const` ([第 7.8.4 节](#))，可以重载一个函数。`const` 对象只能使用 `const` 成员。非 `const` 对象可以使用任一成员，但非 `const` 版本是一个更好的匹配。

While we're at it, we'll define a `private` member named `do_display` to do the actual work of printing the `Screen`. Each of the `display` operations will call this function and then return the object on which it is executing:

在此，我们将定义一个名为 `do_display` 的 `private` 成员来打印 `Screen`。每个 `display` 操作都将调用此函数，然后返回调用自己的那个对象：

```
class Screen {
public:
    // interface member functions
    // display overloaded on whether the object is const or not
    Screen& display(std::ostream &os)
    {
        do_display(os); return *this; }
    const Screen& display(std::ostream &os) const
    {
        do_display(os); return *this; }

private:
    // single function to do the work of displaying a Screen,
    // will be called by the display operations
    void do_display(std::ostream &os) const
    {
        os << contents; }

    // as before
};
```

Now, when we embed `display` in a larger expression, the `nonconst` version will be called. When we `display` a `const` object, then the `const` version is called:

现在，当我们把 `display` 嵌入到一个长表达式中时，将调用非 `const` 版本。当我们 `display` 一个 `const` 对象时，就调用 `const` 版本：

```
Screen myScreen(5,3);
const Screen blank(5, 3);
myScreen.set('#').display(cout); // calls nonconst version
blank.display(cout);           // calls const version
```

Mutable Data Members

可变数据成员

It sometimes (but not very often) happens that a class has a data member that we want to be able to modify, even inside a `const` member

Section 12.2. The Implicit this Pointer

function. We can indicate such members by declaring them as `mutable`.

有时（但不是很经常），我们希望类的数据成员（甚至在 `const` 成员函数内）可以修改。这可以通过将它们声明为 `mutable` 来实现。

A **mutable data member** is a member that is never `const`, even when it is a member of a `const` object. Accordingly, a `const` member function may change a `mutable` member. To declare a data member as mutable, the keyword `mutable` must precede the declaration of the member:

可变数据成员 (**mutable data member**) 永远都不能为 `const`，甚至当它是 `const` 对象的成员时也如此。因此，`const` 成员函数可以改变 `mutable` 成员。要将数据成员声明为可变的，必须将关键字 `mutable` 放在成员声明之前：

```
class Screen {  
public:  
// interface member functions  
private:  
    mutable size_t access_ctr; // may change in a const members  
    // other data members as before  
};
```

We've given `Screen` a new data member named `access_ctr` that is `mutable`. We'll use `access_ctr` to track how often `Screen` member functions are called:

我们给 `Screen` 添加了一个新的可变数据成员 `access_ctr`。使用 `access_ctr` 来跟踪调用 `Screen` 成员函数的频繁程度：

```
void Screen::do_display(std::ostream& os) const  
{  
    ++access_ctr; // keep count of calls to any member function  
    os << contents;  
}
```

Even though `do_display` is `const`, it can increment `access_ctr`. That member is a `mutable` member, so any member function, including `const` functions, can change the value of `access_ctr`.

尽管 `do_display` 是 `const`，它也可以增加 `access_ctr`。该成员是可变成员，所以，任意成员函数，包括 `const` 函数，都可以改变 `access_ctr` 的值。

Advice: Use Private Utility Functions for Common Code

建议：用于公共代码的私有实用函数

Some readers might be surprised that we bothered to define a separate `do_display` operation. After all, the calls to `do_display` aren't much simpler than the action done inside `do_display`. Why bother? We do so for several reasons:

有些读者可能会奇怪为什么要费力地单独定义一个 `do_display` 内部所做的操作更简单。为什么还要如此麻烦？我们这样做有下面几个原因。

1. A general desire to avoid writing the same code in more than one place.

一般愿望是避免在多个地方编写同样的代码。

2. The `display` operation can be expected to become more complicated as our class evolves. As the actions involved become more complex, it makes more obvious sense to write those actions in one place, not two.

`display` 操作预期会随着类的演变而变得更复杂。当所涉及的动作变得更复杂时，只在一处而不是两处编写这些动作有更显著的意义。

3. It is likely that we might want to add debugging information to `do_display` during development that would be eliminated in the final product version of the code. It will be easier to do so if only one definition of `do_display` needs to be changed to add or remove the debugging code.

很可能我们会希望在开发时给 `do_display` 增加调试信息，这些调试信息将会在代码的最终成品版本中去掉。如果只需要改变一个 `do_display` 的定义来增加或删除调试代码，这样做将更容易。

4. There needn't be any overhead involved in this extra function call. We made `do_display` inline, so the run-time performance between calling `do_display` or putting the code directly into the `display` operations should be identical.

这个额外的函数调用不需要涉及任何开销。我们使 `do_display` 成为内联的，所以调用 `do_display` 与将代码直接放入 `display` 操作的运行时性能应该是相同的。

In practice, well-designed C++ programs tend to have lots of small functions such as `do_display` that are called to do the "real" work of some other set of functions.

实际上，设计良好的 C++ 程序经常具有许多像 `do_display` 这样的小函数，它们被调用来完成一些其他函数的“实际”工作。

Exercises Section 12.2

Exercise 12.13: Extend your version of the `Screen` class to include the `move`, `set`, and `display` operations. Test your class by executing the expression:

扩展 `Screen` 类以包含 `move`、`set` 和 `display` 操作。通过执行如下表达式来测试类：

[View full width]

```
→ // move cursor to given position, set that character and  
→     myScreen.move(4,0).set('#').display(cout);
```

Exercise 12.14: It is legal but redundant to refer to members through the `this` pointer. Discuss the pros and cons of explicitly using the `this` pointer to access members.

通过 `this` 指针引用成员虽然合法，但却是多余的。讨论显式使用 `this` 指针访问成员的优缺点。

12.3. Class Scope

12.3. 类作用域

Every class defines its own new scope and a unique type. The declarations of the class members within the class body introduce the member names into the scope of their class. Two different classes have two different class scopes.

每个类都定义了自己的新作用域和唯一的类型。在类的定义体内声明类成员，将成员名引入类的作用域。两个不同的类具有两个的类作用域。



Even if two classes have exactly the same member list, they are different types. The members of each class are distinct from the members of any other class (or any other scope).

即使两个类具有完全相同的成员列表，它们也是不同的类型。每个类的成员不同于任何其他类（或任何其他作用域）的成员。

For example:

例如

```
class First {
public:
    int memi;
    double memd;
};

class Second {
public:
    int memi;
    double memd;
};

First obj1;
Second obj2 = obj1; // error: obj1 and obj2 have different types
```

Using a Class Member

使用类的成员

Outside the class scope, members may be accessed only through an object or a pointer using member access operators dot or arrow, respectively. The left-hand operand to these operators is a class object or a pointer to a class object, respectively. The member name that follows the operator must be declared in the scope of the associated class:

在类作用域之外，成员只能通过对对象或指针分别使用成员访问操作符`.`或`->`来访问。这些操作符左边的操作数分别是一个类对象或指向类对象的指针。跟在操作符后面的成员名字必须在相关联的类的作用域中声明：

```
Class obj;      // class is some class type
Class *ptr = &obj;
// member is a data member of that class
ptr->member;   // fetches member from the object to which ptr points
obj.member;     // fetches member from the object named obj
// memfcn is a function member of that class
ptr->memfcn(); // runs memfcn on the object to which ptr points
obj.memfcn();   // runs memfcn on the object named obj
```

Some members are accessed using the member access operators; others are accessed directly from the class using the scope operator, `(::)`. Ordinary data or function members must be accessed through an object. Members that define types, such as `Screen::index`, are accessed using the scope operator.

一些成员使用成员访问操作符来访问，另一些直接通过类使用作用域操作符`(::)`来访问。一般的数据或函数成员必须通过对对象来访问。定义类型的成员，如`Screen::index`，使用作用域操作符来访问。

Scope and Member Definitions

作用域与成员定义

Member definitions behave as if they are in the scope of the class, even if the member is defined outside the class body. Recall that member definitions that appear outside the class body must indicate the class in which the member appears:

尽管成员是在类的定义体之外定义的，但成员定义就好像它们是在类的作用域中一样。回忆一下，出现在类的定义体之外的成员定义必须指明成员出现在哪个类中：

```
double Sales_item::avg_price() const
{
    if (units_sold)
        return revenue/units_sold;
    else
        return 0;
}
```

Here we use the fully qualified name `Sales_item::avg_price` to indicate that the definition is for the `avg_price` member in the scope of the `Sales_item` class. Once the fully qualified name of the member is seen, the definition is known to be in class scope. Because the definition is in class scope, we can refer to `revenue` and `units_sold` without having to write `this->revenue` or `this->units_sold`.

在这里，我们用完全限定名 `Sales_item::avg_price` 来指出这是类 `Sales_item` 作用域中的 `avg_price` 成员的定义。一旦看到成员的完全限定名，就知道该定义是在类作用域中。因为该定义是在类作用域中，所以我们可以引用 `revenue` 或 `units_sold`，而不必写 `this->revenue` 或 `this->units_sold`。

Parameter Lists and Function Bodies Are in Class Scope

形参表和函数体处于类作用域中

In a member function defined outside the class, the parameter list and member-function body both appear after the member name. These are defined inside the class scope and so may refer to other class members without qualification for example, the definition of the two-parameter version of `get` in class `Screen`:

在定义于类外部的成员函数中，形参表和成员函数体都出现在成员名之后。这些都是在类作用域中定义，所以可以不用限定而引用其他成员。例如，类 `Screen` 中 `get` 的二形参版本的定义：

```
char Screen::get(index r, index c) const
{
    index row = r * width;      // compute the row location
    return contents[row + c];  // offset by c to fetch specified character
}
```

This function uses the type name `index` defined inside `Screen` to name the types of its parameters. Because the parameter list is inside the scope of class `Screen`, there is no need to specify that we want `Screen::index`. It is implicit that the one we want is the one defined in the current class scope. Similarly, the uses of `index`, `width`, and `contents` all refer to names declared inside class `Screen`.

该函数用 `Screen` 内定义的 `index` 类型来指定其形参类型。因为形参表是在 `Screen` 类的作用域内，所以不必指明我们想要的是 `Screen::index`。我们想要的是定义在当前类作用域中的，这是隐含的。同样，使用 `index`、`width` 和 `contents` 时指的都是 `Screen` 类中声明的名字。

Function Return Types Aren't Always in Class Scope

函数返回类型不一定在类作用域中

In contrast to the parameter types, the return type appears before the member name. If the function is defined outside the class body, then the name used for the return type is outside the class scope. If the return type uses a type defined by the class, it must use the fully qualified name. For example, consider the `get_cursor` function:

与形参类型相比，返回类型出现在成员名字前面。如果函数在类定义体之外定义，则用于返回类型的名字在类作用域之外。如果返回类型使用由类定义的类型，则必须使用完全限定名。例如，考虑 `get_cursor` 函数：

```
class Screen {
public:
    typedef std::string::size_type index;
    index get_cursor() const;
};

inline Screen::index Screen::get_cursor() const
{
```

Section 12.3. Class Scope

```
    return cursor;  
}
```

The return type of this function is `index`, which is a type name defined inside the `Screen` class. If we define `get_cursor` outside the class body, the code is not in the class scope until the function name has been processed. When the return type is seen, its name is used outside of the class scope. We must use the fully qualified type name, `Screen::index` to specify that we want the name `index` that is defined inside class `Screen`.

该函数的返回类型是 `index`，这是在 `Screen` 类内部定义的一个类型名。如果在类定义体之外定义 `get_cursor`，则在函数名被处理之前，代码不在类作用域内。当看到返回类型时，其名字是在类作用域之外使用。必须用完全限定的类型名 `Screen::index` 来指定所需要的 `index` 是在类 `Screen` 中定义的名字。

Exercises Section 12.3

Exercise

- 12.15:** List the portions of program text that are in class scope.

列出在类作用域中的程序文本部分。

Exercise

- 12.16:** What would happen if we defined `get_cursor` as follows:

如果如下定义 `get_cursor`，将会发生什么：

```
index Screen::get_cursor() const  
{  
    return cursor;  
}
```

12.3.1. Name Lookup in Class Scope

12.3.1. 类作用域中的名字查找

In the programs we've written so far, [name lookup](#) (the process of finding which declaration is matched to a given use of a name) has been relatively straightforward:

迄今为止，在我们所编写的程序中，[名字查找](#)（寻找与给定的名字使用相匹配的声明的过程）是相对直接的。

1. First, look for a declaration of the name in the block in which the name was used. Only names declared before the use are considered.

首先，在使用该名字的块中查找名字的声明。只考虑在该项使用之前声明的名字。

2. If the name isn't found, the enclosing scope(s) are searched.

如果找不到该名字，则在包围的作用域中查找。

If no declaration is found, then the program is in error. In C++ programs, all names must be declared before they are used.

如果找不到任何声明，则程序出错。在 C++ 程序中，所有名字必须在使用之前声明。

Class scopes may seem to behave a bit differently, but in reality they obey this same rule. Confusion can arise due to the way names are resolved inside a function defined within the class body itself.

类作用域也许表现得有点不同，但实际上遵循同一规则。可能引起混淆的是函数中名字确定的方式，而该函数是在类定义体内定义的。



Class definitions are actually processed in two phases:

类定义实际上是在两个阶段中处理：

1. First, the member declarations are compiled.

首先，编译成员声明；

2. Only after all the class members have been seen are the definitions themselves compiled.

Section 12.3. Class Scope

只有在所有成员出现之后，才编译它们的定义本身。

Of course, the names used in class scope do not always have to be class member names. Name lookup in class scope finds names declared in other scopes as well. During name lookup, if a name used in class scope does not resolve to a class member name, the scopes surrounding the class or member definition are searched to find a declaration for the name.

当然，类作用域中使用的名字并非必须是类成员名。类作用域中的名字查找也会发生在其他作用域中声明的名字。在名字查找期间，如果类作用域中使用的名字不能确定为类成员名，则在包含该类或成员定义的作用域中查找，以便找到该名字的声明。

Name Lookup for Class Member Declarations

类成员声明的名字查找

Names used in the declarations of a class member are resolved as follows:

按以下方式确定在类成员的声明中用到的名字。

1. The declarations of the class members that appear before the use of the name are considered.

检查出现在名字使用之前的类成员的声明。

2. If the lookup in step 1 is not successful, the declarations that appear in the scope in which the class is defined, and that appear before the class definition itself, are considered.

如果第 1 步查找不到，则检查包含类定义的作用域中出现的声明以及出现在类定义之前声明。

For example:

例如：

```
typedef double Money;
class Account {
public:
    Money balance() { return bal; }
private:
    Money bal;
    // ...
};
```

When processing the declaration of the `balance` function, the compiler first looks for a declaration of `Money` in the scope of the class `Account`. The compiler considers only declarations that appear before the use of `Money`. Because no member declaration is found, the compiler then looks for a declaration of `Money` in global scope. Only the declarations located before the definition of the class `Account` are considered. The declaration for the global typedef `Money` is found and is used for the return type of the function `balance` and the data member `bal`.

在处理 `balance` 函数的声明时，编译器首先在类 `Account` 的作用域中查找 `Money` 的声明。编译器只考虑出现在 `Money` 使用之前的声明。因为找不到任何成员声明，编译器随后在全局作用域中查找 `Money` 的声明。只考虑出现在类 `Account` 的定义之前的声明。找到全局的类型别名 `Money` 的声明，并将它用作函数 `balance` 的返回类型和数据成员 `bal` 的类型。



Names of types defined in a class must be seen before they are used as the type of a data member or as the return type or parameter type(s) of a member function.

必须在类中先定义类型名字，才能将它们用作数据成员的类型，或者成员函数的返回类型或形参类型。

The compiler handles member declarations in the order in which they appear in the class. As usual, a name must be defined before it can be used. Moreover, once a name has been used as the name of a type, that name may not be redefined:

编译器按照成员声明在类中出现的次序来处理它们。通常，名字必须在使用之前进行定义。而且，一旦一个名字被用作类型名，该名字就不能被重复定义：

```
typedef double Money;
class Account {
public:
    Money balance() { return bal; } // uses global definition of Money
private:
    // error: cannot change meaning of Money
    typedef long double Money;
    Money bal;
```

Section 12.3. Class Scope

```
}; // ...
```

Name Lookup in Class Member Definitions

类成员定义中的名字查找

A name used in the body of a member function is resolved as follows:

按以下方式确定在成员函数的函数体中用到的名字。

1. Declarations in the member-function local scopes are considered first.

首先检查成员函数局部作用域中的声明。

2. If the a declaration for the name is not found in the member function, the declarations for all the class members are considered.

如果在成员函数中找不到该名字的声明，则检查对所有类成员的声明。

3. If a declaration for the name is not found in the class, the declarations that appear in scope before the member function definition are considered.

如果在类中找不到该名字的声明，则检查在此成员函数定义之前的作用域中出现的声明。

Class Members Follow Normal Block-Scope Name Lookup

类成员遵循常规的块作用域名字查找



Programs that illustrate how name lookup works often have to rely on bad practices. The next several programs contain bad style deliberately.

示例名字查找的程序经常不得不依赖一些坏习惯。下面的几个程序故意包含了坏的风格。

The following function uses the same name for a parameter and a member, which normally should be avoided. We do so here to show how names are resolved:

下面的函数使用了相同的名字来表示形参和成员，这是通常应该避免的。这样做的目的是展示如何确定名字：

```
// Note: This code is for illustration purposes only and reflects bad practice
// It is a bad idea to use the same name for a parameter and a member
int height;
class Screen {
public:
    void dummy_fcn(index height) {
        cursor = width * height; // which height? The parameter
    }
private:
    index cursor;
    index height, width;
};
```

When looking for a declaration for the name `height` used in the definition of `dummy_fcn`, the compiler first looks in the local scope of that function. A function parameter is declared in the local scope of its function. The name `height` used in the body of `dummy_fcn` refers to this parameter declaration.

查找 `dummy_fcn` 的定义中使用的名字 `height` 的声明时，编译器首先在该函数的局部作用域中查找。函数的局部作用域中声明了一个函数形参。`dummy_fcn` 的函数体中使用的名字 `height` 指的就是这个形参声明。

In this case, the `height` parameter hides the member named `height`.

在本例中，`height` 形参屏蔽名为 `height` 的成员。

Section 12.3. Class Scope



Even though the class member is hidden, it is still possible to use it by qualifying the member's name with the name of its class or by using the `this` pointer explicitly.

尽管类的成员被屏蔽了，但仍然可以通过用类名来限定成员名或显式使用 `this` 指针来使用它。

If we wanted to override the normal lookup rules, we could do so:

如果我们想覆盖常规的查找规则，应该这样做：

```
// bad practice: Names local to member functions shouldn't hide member names
void dummy_fcn(index height) {
    cursor = width * this->height; // member height
    // alternative way to indicate the member
    cursor = width * Screen::height; // member height
}
```

After Function Scope, Look in Class Scope

函数作用域之后，在类作用域中查找

If we wanted to use the member named `height`, a much better way to do so would be to give the parameter a different name:

如果想要使用 `height` 成员，更好的方式也许是为形参取一个不同的名字：

```
// good practice: Don't use member name for a parameter or other local variable
void dummy_fcn(index ht) {
    cursor = width * height; // member height
}
```

Now when the compiler looks for the name `height`, it will not find that name in the function. The compiler next looks in the `Screen` class. Because `height` is used inside a member function, the compiler looks at all the member declarations. Even though the declaration of `height` appears after its use inside `dummy_fcn`, the compiler resolves this use to the data member named `height`.

现在当编译器查找名字 `height` 时，它将不会在函数内查找该名字。编译器接着会在 `Screen` 类中查找。因为 `height` 是在成员函数内部使用，所以编译器在所有成员声明中查找。尽管 `height` 是先在 `dummy_fcn` 中使用，然后再声明，编译器还是确定这里用的是名为 `height` 的数据成员。

After Class Scope, Look in the Surrounding Scope

类作用域之后，在外围作用域中查找

If the compiler doesn't find the name in function or class scope, it looks for the name in the surrounding scope. In our example, declarations in global scope that appear before the definition of the `Screen` include a global object named `height`. However, that object is hidden.

如果编译器不能在函数或类作用域中找到，就在外围作用域中查找。在本例子中，出现在 `Screen` 定义之前的全局作用域中声明了一个名为 `height` 的全局声明。然而，该对象被屏蔽了。



Even though the global object is hidden, it is still possible to use it by qualifying the name with the global scope resolution operator.

尽管全局对象被屏蔽了，但通过用全局作用域确定操作符来限定名字，仍然可以使用它。

```
// bad practice: Don't hide names that are needed from surrounding scopes
void dummy_fcn(index height) {
    cursor = width * ::height;// which height? The global one
}
```

Names Are Resolved Where They Appear within the File

在文件中名字的出现处确定名字

When a member is defined outside the class definition, the third step of name lookup not only considers the declarations in global scope that appear before the definition of class Screen, but also considers the global scope declarations that appear before the member function definition for example:

当成员定义在类定义的外部时，名字查找的第 3 步不仅要考虑在 `Screen` 类定义之前的全局作用域中的声明，而且要考虑在成员函数定义之前出现的全局作用域声明。例如：

```
class Screen {
public:
    // ...
    void setHeight(index);
private:
    index height;
};

Screen::index verify(Screen::index);

void Screen::setHeight(index var) {
    // var: refers to the parameter
    // height: refers to the class member
    // verify: refers to the global function
    height = verify(var);
}
```

Notice that the declaration of the global function `verify` is not visible before the definition of the class `Screen`. However, the third step of name lookup considers the surrounding scope declarations that appear before the member definition, and the declaration for the global function `verify` is found.

注意，全局函数 `verify` 的声明在 `Screen` 类定义之前是不可见的。然而，名字查找的第 3 步要考虑那些出现在成员定义之前的外围作用域声明，并找到全局函数 `verify` 的声明。

Exercises Section 12.3.1

Exercise What would happen if we put the `typedef` in the `Screen` class as the last line in the class?

12.17: 如果将 `Screen` 类中的类型别名放到类中的最后一行，将会发生什么？

Exercise Explain the following code. Indicate which definition of `Type` or `initVal` is used for each use of those names. If there are any errors, say how you would fix the program.

解释下述代码。指出每次使用 `Type` 或 `initVal` 时用到的是哪个名字定义。如果存在错误，说明如何改正。

```
typedef string Type;
Type initVal();

class Exercise {
public:
    // ...
    typedef double Type;
    Type setVal(Type);
    Type initVal();
private:
    int val;
};

Type Exercise::setVal(Type parm) {
    val = parm + initVal();
}
```

The definition of the member function `setVal` is in error. Apply the necessary changes so that the class `Exercise` uses the global `typedef Type` and the global function `initVal`.

成员函数 `setVal` 的定义有错。进行必要的修改以便类 `Exercise` 使用全局的类型别名 `Type` 和全局函数 `initVal`。

12.4. Constructors

12.4. 构造函数

Constructors ([Section 2.3.3](#), p. 49) are special member functions that are executed whenever we create new objects of a class type. The job of a constructor is to ensure that the data members of each object start out with sensible initial values. [Section 7.7.3](#) (p. 262) showed how we define a constructor:

[构造函数](#)是特殊的成员函数，只要创建类类型的新对象，都要执行构造函数。构造函数的工作是保证每个对象的数据成员具有合适的初始值。[第 7.7.3 节](#)展示了如何定义构造函数：

```
class Sales_item {
public:
    // operations on Sales_item objects
    // default constructor needed to initialize members of built-in type
    Sales_item(): units_sold(0), revenue(0.0) { }
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;
};
```

This constructor uses the constructor initializer list to initialize the `units_sold` and `revenue` members. The `isbn` member is implicitly initialized by the `string` [default constructor](#) as an empty string.

这个构造函数使用构造函数初始化列表来初始化 `units_sold` 和 `revenue` 成员。`isbn` 成员由 `string` 的[默认构造函数](#)隐式初始化为空串。

Constructors have the same name as the name of the class and may not specify a return type. Like any other function, they may define zero or more parameters.

构造函数的名字与类的名字相同，并且不能指定返回类型。像其他任何函数一样，它们可以没有形参，也可以定义多个形参。

Constructors May Be Overloaded

构造函数可以被重载

There is no constraint on the number of constructors we may declare for a class, provided that the parameter list of each constructor is unique. How can we know which or how many constructors to define? Ordinarily, constructors differ in ways that allow the user to specify differing ways to initialize the data members.

可以为一个类声明的构造函数的数量没有限制，只要每个构造函数的形参表是唯一的。我们如何才能知道应该定义哪个或多少个构造函数？一般而言，不同的构造函数允许用户指定不同的方式来初始化数据成员。

For example, we might logically extend our `Sales_item` class by providing two additional constructors: one that would let users provide an initial value for the `isbn` and another that would let them initialize the object by reading an `istream` object:

例如，逻辑上可以通过提供两个额外的构造函数来扩展 `Sales_item` 类：一个允许用户提供 `isbn` 的初始值，另一个允许用户通过读取 `istream` 对象来初始化对象：

```
class Sales_item;
// other members as before
public:
    // added constructors to initialize from a string or an istream
    Sales_item(const std::string&); // initializes from string
    Sales_item(std::istream&); // initializes from istream
    Sales_item(); // default constructor
};
```

Arguments Determine Which Constructor to Use

实参决定使用哪个构造函数

Our class now defines three constructors. We could use any of these constructors when defining new objects:

我们的类现在定义了三个构造函数。在定义新对象时，可以使用这些构造函数中的任意一个：

```
// uses the default constructor:  
// isbn is the empty string; units_sold and revenue are 0  
Sales_item empty;  
// specifies an explicit isbn; units_sold and revenue are 0  
Sales_item Primer_3rd_Ed("0-201-82470-1");  
// reads values from the standard input into isbn, units_sold, and revenue  
Sales_item Primer_4th_ed(cin);
```

The argument type(s) used to initialize an object determines which constructor is used. In the definition of `empty`, there is no initializer, so the default constructor is run. The constructor that takes a single `string` argument is used to initialize `Primer_3rd_ed`; the one that takes a reference to an `istream` initializes `Primer_4th_ed`.

用于初始化一个对象的实参类型决定使用哪个构造函数。在 `empty` 的定义中，没有初始化式，所以运行默认构造函数。接受一个 `string` 实参的构造函数用于初始化 `Primer_3rd_ed`；接受一个 `istream` 引用的构造函数初始化 `Primer_4th_ed`。

Constructors Are Executed Automatically

构造函数自动执行

The compiler runs a constructor whenever an object of the type is created:

只要创建该类型的一个对象，编译器就运行一个构造函数：

```
// constructor that takes a string used to create and initialize variable  
Sales_item Primer_2nd_ed("0-201-54848-8");  
// default constructor used to initialize unnamed object on the heap  
Sales_item *p = new Sales_item();
```

In the first case, the constructor that takes a `string` is run to initialize the variable named `Primer_2nd_ed`. In the second case, a new `Sales_item` object is allocated dynamically. Assuming that the allocation succeeds, then the object is initialized by running the default constructor.

第一种情况下，运行接受一个 `string` 实参的构造函数，来初始化变量 `Primer_2nd_ed`。第二种情况下，动态分配一个新的 `Sales_item` 对象。假定分配成功，则通过运行默认构造函数初始化该对象。

Constructors for `const` Objects

用于 `const` 对象的构造函数

A constructor may not be declared as `const` ([Section 7.7.1](#), p. 260):

构造函数不能声明为 `const` [第 7.7.1 节](#):

```
class Sales_item {  
public:  
    Sales_item() const; // error  
};
```

There is no need for a `const` constructor. When we create a `const` object of a class type, an ordinary constructor is run to initialize the `const` object. The job of the constructor is to initialize an object. A constructor is used to initialize an object regardless of whether the object is `const`.

`const` 构造函数是不必要的。创建类类型的 `const` 对象时，运行一个普通构造函数来初始化该 `const` 对象。构造函数的工作是初始化对象。不管对象是否为 `const`，都用一个构造函数来初始化该对象。

Exercises Section 12.4

Exercise 12.19: Provide one or more constructors that allows the user of this class to specify initial values for none or all of the data elements of this class:

提供一个或多个构造函数，允许该类的用户不指定数据成员的初始值或指定所有数据成员的初始值：

Section 12.4. Constructors

```
class NoName {
public:
    // constructor(s) go here ...
private:
    std::string *pstring;
    int      ival;
    double   dval;
};
```

Explain how you decided how many constructors were needed and what parameters they should take.

解释如何确定需要多少个构造函数以及它们应接受什么样的形参。

Exercise 12.20: Choose one of the following abstractions (or an abstraction of your own choosing). Determine what data is needed in the class. Provide an appropriate set of constructors. Explain your decisions.

从下述抽象中选择一个（或一个自己定义的抽象），确定类中需要什么数据，并提供适当的构造函数集。解释你的决定：

- (a) Book (b) Date (c) Employee
- (d) Vehicle (e) Object (f) Tree

12.4.1. The Constructor Initializer

12.4.1. 构造函数初始化式

Like any other function, a constructor has a name, a parameter list, and a function body. Unlike other functions, a constructor may also contain a constructor initializer list:

```
// recommended way to write constructors using a constructor initializer
Sales_item::Sales_item(const string &book):
    isbn(book), units_sold(0), revenue(0.0) { }
```

The constructor initializer starts with a colon, which is followed by a comma-separated list of data members each of which is followed by an initializer inside parentheses. This constructor initializes the `isbn` member to the value of its `book` parameter and initializes `units_sold` and `revenue` to 0. As with any member function, constructors can be defined inside or outside of the class. The constructor initializer is specified only on the constructor definition, not its declaration.

构造函数初始化列表以一个冒号开始，接着是一个以逗号分隔的数据成员列表，每个数据成员后面跟一个放在圆括号中的初始化式。这个构造函数将 `isbn` 成员初始化为 `book` 形参的值，将 `units_sold` 和 `revenue` 初始化为 0。与任意的成员函数一样，构造函数可以定义在类的内部或外部。构造函数初始化只在构造函数的定义中而不是声明中指定。

The constructor initializer is a feature that many reasonably experienced C++ programmers have not mastered.

构造函数初始化列表是许多相当有经验的 C++ 程序员都没有掌握的一个特性。

One reason constructor initializers are hard to understand is that it is usually legal to omit the initializer list and assign values to the data members inside the constructor body. For example, we could write the `Sales_item` constructor that takes a `string` as

构造函数初始化列表难以理解的一个原因在于，省略初始化列表在构造函数的函数体内对数据成员赋值是合法的。例如，可以将接受一个 `string` 的 `Sales_item` 构造函数编写为：

Section 12.4. Constructors

```
// legal but sloppier way to write the constructor:  
// no constructor initializer  
Sales_item::Sales_item(const string &book)  
{  
    isbn = book;  
    units_sold = 0;  
    revenue = 0.0;  
}
```

This constructor assigns, but does not explicitly initialize, the members of class `Sales_item`. Regardless of the lack of an explicit initializer, the `isbn` member is initialized before the constructor is executed. This constructor implicitly uses the default `string` constructor to initialize `isbn`. When the body of the constructor is executed, the `isbn` member already has a value. That value is overwritten by the assignment inside the constructor body.

这个构造函数给类 `Sales_item` 的成员赋值，但没有进行显式初始化。不管是否有显式的初始化式，在执行构造函数之前，要初始化 `isbn` 成员。这个构造函数隐式使用默认的 `string` 构造函数来初始化 `isbn`。执行构造函数的函数体时，`isbn` 成员已经有值了。该值被构造函数函数体中的赋值所覆盖。

Conceptually, we can think of a constructor as executing in two phases: (1) the initialization phase and (2) a general computation phase. The computation phase consists of all the statements within the body of the constructor.

从概念上讲，可以认为构造函数分两个阶段执行：(1) 初始化阶段；(2) 普通的计算阶段。计算阶段由构造函数函数体中的所有语句组成。



Data members of class type are *always* initialized in the initialization phase, regardless of whether the member is initialized explicitly in the constructor initializer list. Initialization happens *before* the computation phase begins.

不管成员是否在构造函数初始化列表中显式初始化，类类型的数据成员总是在初始化阶段初始化。初始化发生在计算阶段开始之前。

Each member that is not explicitly mentioned in the constructor initializer is initialized using the same rules as those used to initialize variables ([Section 2.3.4](#), p. 50). Data members of class type are initialized by running the type's default constructor. The initial value of members of built-in or compound type depend on the scope of the object: At local scope those members are uninitialized, at global scope they are initialized to 0.

在构造函数初始化列表中没有显式提及的每个成员，使用与[初始化变量](#)相同的规则来进行初始化。运行该类型的默认构造函数，来初始化类类型的数据成员。内置或复合类型的成员的初始值依赖于对象的作用域：在局部作用域中这些成员不被初始化，而在全局作用域中它们被初始化为 0。

The two versions of the `Sales_item` constructor that we wrote in this section have the same effect: Whether we initialized the members in the constructor initializer list or assigned to them inside the constructor body, the end result is the same. After the constructor completes, the three data members hold the same values. The difference is that the version that uses the constructor initializer *initializes* its data members. The version that does not define a constructor initializer assigns values to the data members in the body of the constructor. How significant this distinction is depends on the type of the data member.

在本节中编写的两个 `Sales_item` 构造函数版本具有同样的效果：无论是在构造函数初始化列表中初始化成员，还是在构造函数函数体中对它们赋值，最终结果是相同的。构造函数执行结束后，三个数据成员保存同样的值。不同之处在于，使用构造函数初始化列表的版本初始化数据成员，没有定义初始化列表的构造函数版本在构造函数函数体中对数据成员赋值。这个区别的重要性取决于数据成员的类型。

Constructor Initializers Are Sometimes Required

有时需要构造函数初始化列表

If an initializer is not provided for a class member, then the compiler implicitly uses the default constructor for the member's type. If that class does not have a default constructor, then the attempt by the compiler to use it will fail. In such cases, an initializer must be provided in order to initialize the data member.

如果没有为类成员提供初始化式，则编译器会隐式地使用成员类型的默认构造函数。如果那个类没有默认构造函数，则编译器尝试使用默认构造函数将会失败。在这种情况下，为了初始化数据成员，必须提供初始化式。



Some members *must* be initialized in the constructor initializer. For such members, assigning to them in the constructor body doesn't work. Members of a class type that do not have a default constructor and members that are `const` or reference types *must* be initialized in the constructor initializer *regardless of type*.

有些成员必须在构造函数初始化列表中进行初始化。对于这样的成员，在构造函数函数体中对它们赋值不起作用。没有默认构造函数的类类型的成员，以及 `const` 或引用类型的成员，不管是哪种类型，都必须在构造函数初始化列表中进行初始化。

Section 12.4. Constructors

Because members of built-in type are not implicitly initialized, it may seem that it doesn't matter whether these members are initialized or assigned. With two exceptions, using an initializer is equivalent to assigning to a nonclass data member both in result and in performance.

因为内置类型的成员不进行隐式初始化，所以对这些成员是进行初始化还是赋值似乎都无关紧要。除了两个例外，对非类类型的数据成员进行赋值或使用初始化式在结果和性能上都是等价的。

For example, the following constructor is in error:

例如，下面的构造函数是错误的：

```
class ConstRef {
public:
    ConstRef(int ii);
private:
    int i;
    const int ci;
    int &ri;
};

// no explicit constructor initializer: error ri is uninitialized
ConstRef::ConstRef(int ii)
{
    // assignments:
    i = ii;    // ok
    ci = ii;   // error: cannot assign to a const
    ri = i;    // assigns to ri which was not bound to an object
}
```

Remember that we can initialize but not assign to `const` objects or objects of reference type. By the time the body of the constructor begins executing, initialization is complete. Our only chance to initialize `const` or reference data members is in the constructor initializer. The correct way to write the constructor is

记住，可以初始化 `const` 对象或引用类型的对象，但不能对它们赋值。在开始执行构造函数的函数体之前，要完成初始化。初始化 `const` 或引用类型数据成员的唯一机会是构造函数初始化列表中。编写该构造函数的正确方式为

```
// ok: explicitly initialize reference and const members
ConstRef::ConstRef(int ii): i(ii), ci(i), ri(ii) { }
```

Advice: Use Constructor Initializers

建议：使用构造函数初始化列表

In many classes, the distinction between initialization and assignment is strictly a matter of low-level efficiency: A data member is initialized and assigned when it could have been initialized directly. More important than the efficiency issue is the fact that some data members must be initialized.

在许多类中，初始化和赋值严格来讲都是低效率的：数据成员可能已经被直接初始化了，还要对它进行初始化和赋值。比较率问题更重要的是，某些数据成员必须要初始化，这是一个事实。



We must use an initializer for any `const` or reference member or for any member of a class type that does not have a default constructor.

必须对任何 `const` 或引用类型成员以及没有默认构造函数的类类型的任何成员使用初始化式。

By routinely using constructor initializers, we can avoid being surprised by compile-time errors when we have a class with a member that requires a constructor initializer.

当类成员需要使用初始化列表时，通过常规地使用构造函数初始化列表，就可以避免发生编译时错误。

Order of Member Initialization

成员初始化的次序

Section 12.4. Constructors

Not surprisingly, each member may be named only once in the constructor initializer. After all, what might it mean to give a member two initial values? What may be more surprising is that the constructor initializer list specifies only the values used to initialize the members, not the order in which those initializations are performed. The order in which members are initialized is the order in which the members are defined. The first member is initialized first, then the next, and so on.

每个成员在构造函数初始化列表中只能指定一次，这不会令人惊讶。毕竟，给一个成员两个初始值意味着什么？也许更令人惊讶的是，构造函数初始化列表仅指定用于初始化成员的值，并不指定这些初始化执行的次序。成员被初始化的次序就是定义成员的次序。第一个成员首先被初始化，然后是第二个，依次类推。



The order of initialization often doesn't matter. However, if one member is initialized in terms of another, then the order in which members are initialized is crucially important.

初始化的次序常常无关紧要。然而，如果一个成员是根据其他成员而初始化，则成员初始化的次序是至关重要的。

Consider the following class:

考虑下面的类：

```
class X {  
    int i;  
    int j;  
public:  
    // run-time error: i is initialized before j  
    X(int val): j(val), i(j) {}  
};
```

In this case, the constructor initializer is written to make it appear as if `j` is initialized with `val` and then `j` is used to initialize `i`. However, `i` is initialized first. The effect of this initializer is to initialize `i` with the as yet uninitialized value of `j`!

在这种情况下，构造函数初始化列表看起来似乎是用`val` 初始化 `j`，然后再用 `j` 来初始化 `i`。然而，`i` 首先被初始化。这个初始化列表的效果是用尚未初始化的 `j` 值来初始化 `i`！

Some compilers are kind enough to generate a warning if the data members are listed in the constructor initializer in a different order from the order in which the members are declared.

如果数据成员在构造函数初始化列表中的列出次序与成员被声明的次序不同，那么有的编译器非常友好，会给出一个警告。



It is a good idea to write constructor initializers in the same order as the members are declared.
Moreover, when possible, avoid using members to initialize other members.

按照与成员声明一致的次序编写构造函数初始化列表是个好主意。此外，尽可能避免使用成员来初始化其他成员。

It is often the case that we can avoid any problems due to order of execution for initializers by (re)using the constructor's parameters rather than using the object's data members. For example, it would be better to write the constructor for `X` as

一般情况下，通过（重复）使用构造函数的形参而不是使用对象的数据成员，可以避免由初始化式的执行次序而引起的任何问题。例如，下面这样为 `X` 编写构造函数可能更好：

```
X(int val): i(val), j(val) {}
```

In this version, the order in which `i` and `j` are initialized doesn't matter.

在这个版本中，`i` 和 `j` 初始化的次序就是无关紧要的。

Initializers May Be Any Expression

初始化式可以是任意表达式

Section 12.4. Constructors

An initializer may be an arbitrarily complex expression. As an example, we could give our `Sales_item` class a new constructor that takes a `string` representing the `isbn`, an `unsigned` representing the number of books sold, and a `double` representing the price at which each of these books was sold:

一个初始化式可以是任意复杂的表达式。例如，可以给 `Sales_item` 类一个新的构造函数，该构造函数接受一个 `string` 表示 `isbn`，一个 `unsigned` 表示售出书的数目，一个 `double` 表示每本书的售出价格：

```
Sales_item(const std::string &book, int cnt, double price):
    isbn(book), units_sold(cnt), revenue(cnt * price) {}
```

This initializer for `revenue` uses the parameters representing price and number sold to calculate the object's `revenue` member.

`revenue` 的初始化式使用表示价格和售出数目的形参来计算对象的 `revenue` 成员。

Initializers for Data Members of Class Type

类类型的数据成员的初始化式

When we initialize a member of class type, we are specifying arguments to be passed to one of the constructors of that member's type. We can use any of that type's constructors. For example, our `Sales_item` class could initialize `isbn` using any of the `string` constructors ([Section 9.6.1](#), p. 338). Instead of using the empty string, we might decide that the default value for `isbn` should be a value that represents an impossibly high value for an ISBN. We could initialize `isbn` to a string of ten 9s:

初始化类类型的成员时，要指定实参并传递给成员类型的一个构造函数。可以使用该类型的任意构造函数。例如，`Sales_item` 类可以使用任意一个 `string` 构造函数来初始化 `isbn` ([第 9.6.1 节](#))。也可以用 ISBN 取值的极限值来表示 `isbn` 的默认值，而不是用空字符串。可以将 `isbn` 初始化为由 10 个 9 构成的串：

```
// alternative definition for Sales_item default constructor
Sales_item(): isbn(10, '9'), units_sold(0), revenue(0.0) {}
```

This initializer uses the `string` constructor that takes a count and a character and generates a `string` holding that character repeated that number of times.

这个初始化式使用 `string` 构造函数，接受一个计数值和一个字符，并生成一个 `string`，来保存重复指定次数的字符。

Exercises Section 12.4.1

Exercise 12.21: Write the default constructor using a constructor initializer for class that contains the following members: a `const string`, an `int`, a `double*`, and an `ifstream&`. Initialize the `string` to hold the name of the class.

使用构造函数初始化列表编写类的默认构造函数，该类包含如下成员：一个 `const string`，一个 `int`，一个 `double*` 和一个 `ifstream&`。初始化 `string` 来保存类的名字。

Exercise 12.22: The following initializer is in error. Identify and fix the problem.

下面的初始化式有错误。找出并改正错误。

```
struct X {
    X (int i, int j): base(i), rem(base % j) { }
    int rem, base;
};
```

Exercise 12.23: Assume we have a class named `NoDefault` that has a constructor that takes an `int` but no default constructor. Define a class `C` that has a member of type `NoDefault`. Define the default constructor for `C`.

假定有个命名为 `NoDefault` 的类，该类有一个接受一个 `int` 的构造函数，但没有默认构造函数。定义有一个 `NoDefault` 类型成员的类 `C`。为类 `C` 定义默认构造函数。

12.4.2. Default Arguments and Constructors

12.4.2. 默认实参与构造函数

Let's look again at our definitions for the default constructor and the constructor that takes a `string`:

再来看看默认构造函数和接受一个 `string` 的构造函数的定义:

```
Sales_item(const std::string &book):
    isbn(book), units_sold(0), revenue(0.0) { }
Sales_item(): units_sold(0), revenue(0.0) { }
```

These constructors are almost the same: The only difference is that the constructor that takes a `string` parameter uses the parameter to initialize `isbn`. The default constructor (implicitly) uses the `string` default constructor to initialize `isbn`.

这两个构造函数几乎是相同的: 唯一的区别在于, 接受一个 `string` 形参的构造函数使用该形参来初始化 `isbn`。默认构造函数 (隐式地) 使用 `string` 的默认构造函数来初始化 `isbn`。

We can combine these constructors by supplying a default argument for the `string` initializer:

可以通过为 `string` 初始化式提供一个默认实参将这些构造函数组合起来:

```
class Sales_item {
public:
    // default argument for book is the empty string
    Sales_item(const std::string &book = ""):
        isbn(book), units_sold(0), revenue(0.0) { }
    Sales_item(std::istream &is);
    // as before
};
```

Here we define only two constructors, one of which provides a default argument for its parameter. The constructor that takes a default argument for its single `string` parameter will be run for either of these definitions:

在这里, 我们只定义了两个构造函数, 其中一个为其形参提供一个默认实参。对于下面的任一定义, 将执行为其 `string` 形参接受默认实参的那个构造函数:

```
Sales_item empty;
Sales_item Primer_3rd_Ed("0-201-82470-1");
```

In the case of `empty`, the default argument is used, whereas `Primer_3rd_Ed` supplies an explicit argument.

在 `empty` 的情况下, 使用默认实参, 而 `Primer_3rd_Ed` 提供了一个显式实参。

Each version of our class provides the same interface: They both initialize a `Sales_item` to the same values given a `string` or given no initializer.

类的两个版本提供同一接口: 给定一个 `string` 或不给定初始化式, 它们都将在一个 `Sales_item` 初始化为相同的值。

We prefer to use a default argument because it reduces code duplication.

我们更喜欢使用默认实参, 因为它减少代码重复。



Exercises Section 12.4.2

Exercise

12.24:

Using the version of `Sales_item` from page 458 that defined two constructors, one of which has a default argument for its single `string` parameter, determine which constructor is used to initialize each of the following variables and list the values of the data members in each object:

上面的 `Sales_item` 定义了两个构造函数, 其中之一有一个默认实参对应其单个 `string` 形参。使用该 `Sales_item` 版本, 确定用哪个构造函数来初始化下述的每个变量, 并列出每个对象中数据成员的值:

Section 12.4. Constructors

```
Sales_item first_item(cin);
int main() {
    Sales_item next;
    Sales_item last("9-999-99999-9");
}
```

Exercise 12.25: Logically, we might want to supply `cin` as a default argument to the constructor that takes an `istream&`. Write the constructor declaration that uses `cin` as a default argument.

逻辑上讲，我们可能希望将 `cin` 作为默认实参提供给接受一个 `istream&` 形参的构造函数。编写使用 `cin` 作为默认实参的构造函数声明。

Exercise 12.26: Would it be legal for both the constructor that takes a `string` and the one that takes an `istream&` to have default arguments? If not, why not?

对于分别接受一个 `string` 和接受一个 `istream&` 的构造函数，具有默认实参都是合法的吗？如果不是，为什么？

12.4.3. The Default Constructor

12.4.3. 默认构造函数

The default constructor is used whenever we define an object but do not supply an initializer. A constructor that supplies default arguments for all its parameters also defines the default constructor.

只要定义一个对象时没有提供初始化式，就使用默认构造函数。为所有形参提供默认实参的构造函数也定义了默认构造函数。

The Synthesized Default Constructor

合成的默认构造函数

If a class defines even one constructor, then the compiler will not generate the default constructor. The basis for this rule is that if a class requires control to initialize an object in one case, then the class is likely to require control in all cases.

一个类哪怕只定义了一个构造函数，编译器也不会再生成默认构造函数。这条规则的根据是，如果一个类在某种情况下需要控制对象初始化，则该类很可能在所有情况下都需要控制。



The compiler generates a default constructor automatically only if a class defines *no* constructors.

只有当一个类没有定义构造函数时，编译器才会自动生成一个默认构造函数。

The **synthesized default constructor** initializes members using the same rules as those that apply for how variables are initialized. Members that are of class type are initialized by running each member's own default constructor. Members of built-in or compound type, such as pointers and arrays, are initialized only for objects that are defined at global scope. When objects are defined at local scope, then members of built-in or compound type are *uninitialized*.

合成的默认构造函数 (synthesized default constructor) 使用与变量初始化相同的规则来初始化成员。具有类类型的成员通过运行各自的默认构造函数来进行初始化。内置和复合类型的成员，如指针和数组，只对定义在全局作用域中的对象才初始化。当对象定义在局部作用域中时，内置或复合类型的成员不进行初始化。



If a class contains data members of built-in or compound type, then the class should not rely on the synthesized default constructor. It should define its own constructor to initialize these members.

如果类包含内置或复合类型的成员，则该类不应该依赖于合成的默认构造函数。它应该定义自己的构造函数来初始化这些成员。

Moreover, every constructor should provide initializers for members of built-in or compound type. A constructor that does not initialize a member of built-in or compound type leaves that member in an undefined state. Using an undefined member in any way other than as the target of an assignment is an error. If every constructor sets every member to an explicit, known state, then member functions can distinguish between an empty object and one that has actual values.

此外，每个构造函数应该为每个内置或复合类型的成员提供初始化式。没有初始化内置或复合类型成员的构造函数，将使那些成员处于未定义的状态。除了作为赋值的目标之外，以任何方式使用一个未定义的成员都是错误的。如果每个构造函数将每个成员设置为明确的已知状态，则成员函数可以区分空对象和具有实际值的对象。

Classes Should Usually Define a Default Constructor

类通常应定义一个默认构造函数

In certain cases, the default constructor is applied implicitly by the compiler. If the class has no default constructor, then the class may not be used in these contexts. To illustrate the cases where a default constructor is required, assume we have a class named `NoDefault` that does not define its own default constructor but does have a constructor that takes a `string` argument. Because the class defines a constructor, the compiler will not synthesize the default constructor. The fact that `NoDefault` has no default constructor means:

在某些情况下，默认构造函数是由编译器隐式应用的。如果类没有默认构造函数，则该类就不能用在这些环境中。为了例示需要默认构造函数的情况，假定有一个 `NoDefault` 类，它没有定义自己的默认构造函数，却有一个接受一个 `string` 实参的构造函数。因为该类定义了一个构造函数，因此编译器将不合成默认构造函数。`NoDefault` 没有默认构造函数，意味着：

1. Every constructor for every class that has a `NoDefault` member must explicitly initialize the `NoDefault` member by passing an initial `string` value to the `NoDefault` constructor.

具有 `NoDefault` 成员的每个类的每个构造函数，必须通过传递一个初始的 `string` 值给 `NoDefault` 构造函数来显式地初始化 `NoDefault` 成员。

2. The compiler will not synthesize the default constructor for classes that have members of type `NoDefault`. If such classes want to provide a default, they must define one explicitly, and that constructor must explicitly initialize their `NoDefault` member.

编译器将不会为具有 `NoDefault` 类型成员的类合成分默认构造函数。如果这样的类希望提供默认构造函数，就必须显式地定义，并且默认构造函数必须显式地初始化其 `NoDefault` 成员。

3. The `NoDefault` type may not be used as the element type for a dynamically allocated array.

`NoDefault` 类型不能用作动态分配数组的元素类型。

4. Statically allocated arrays of type `NoDefault` must provide an explicit initializer for each element.

`NoDefault` 类型的静态分配数组必须为每个元素提供一个显式的初始化式。

5. If we have a container such as `vector` that holds `NoDefault` objects, we cannot use the constructor that takes a size without also supplying an element initializer.

如果有一个保存 `NoDefault` 对象的容器，例如 `vector`，就不能使用接受容器大小而没有同时提供一个元素初始化式的构造函数。



In practice, it is almost always right to provide a default constructor if other constructors are being defined. Ordinarily the initial values given to the members in the default constructor should indicate that the object is "empty."

实际上，如果定义了其他构造函数，则提供一个默认构造函数几乎总是对的。通常，在默认构造函数中给成员提供的初始值应该指出该对象是“空”的。

Using the Default Constructor

使用默认构造函数

A common mistake among programmers new to C++ is to declare an object initialized with the default constructor as follows:

初级 C++ 程序员常犯的一个错误是，采用以下方式声明一个用默认构造函数初始化的对象：



```
// oops! declares a function, not an object
Sales_item myobj();
```

The declaration of `myobj` compiles without complaint. However, when we try to use `myobj`

编译 `myobj` 的声明没有问题。然而，当我们试图使用 `myobj` 时

```
Sales_item myobj(); // ok: but defines a function, not an object
if (myobj.same_isbn(Primer_3rd_ed)) // error: myobj is a function
```

the compiler complains that we cannot apply member access notation to a function! The problem is that our definition of `myobj` is interpreted by the compiler as a declaration of a function taking no parameters and returning an object of type `Sales_item`—hardly what we intended! The correct way to define an object using the default constructor is to leave off the trailing, empty parentheses:

```
// ok: defines a class object ...
Sales_item myobj;
```

On the other hand, this code is fine:

另一方面，下面这段代码也是正确的：

```
// ok: create an unnamed, empty Sales_item and use to initialize myobj
Sales_item myobj = Sales_item();
```

Here we create and value-initialize a `Sales_item` object and to use it to initialize `myobj`. The compiler value-initializes a `Sales_item` by running its default constructor.

在这里，我们创建并初始化一个 `Sales_item` 对象，然后用它来按值初始化 `myobj`。编译器通过运行 `Sales_item` 的默认构造函数来按值初始化一个 `Sales_item`。

Exercises Section 12.4.3

Exercise Which, if any, of the following statements are untrue? Why?

12.27:

下面的陈述中哪个是不正确的（如果有的话）？为什么？

- a. A class must provide at least one constructor.

类必须提供至少一个构造函数。

- b. A default constructor is a constructor with no parameters for its parameter list.

默认构造函数的形参列表中没有形参。

- c. If there are no meaningful default values for a class, the class should not provide a default constructor.

如果一个类没有有意义的默认值，则该类不应该提供默认构造函数。

- d. If a class does not define a default constructor, the compiler generates one automatically, initializing each data member to the default value of its associated type.

如果一个类没有定义默认构造函数，则编译器会自动生成一个，同时将每个数据成员初始化为相关类型的默认值。

12.4.4. Implicit Class-Type Conversions

12.4.4. 隐式类类型转换

As we saw in [Section 5.12](#) (p. 178), the language defines several automatic conversions among the built-in types. We can also define how to implicitly convert an object from another type to our class type or to convert from our class type to another type. We'll see in [Section 14.9](#) (p. 535) how to define conversions from a class type to another type. To define an implicit conversion to a class type, we need to define an appropriate constructor.

在第 5.12 节介绍过，C++ 语言定义了内置类型之间的几个自动转换。也可以定义如何将其他类型的对象隐式转换为我们的类类型，或将我们的类类型的对象隐式转换为其他类型。在第 14.9 节将会看到如何定义从类类型到其他类型的转换。为了定义到类类型的隐式转换，需要定义合适的构造函数。



A constructor that can be called with a single argument defines an implicit conversion from the parameter type to the class type.

可以用单个实参来调用的构造函数定义了从形参类型到该类类型的一个隐式转换。

Let's look again at the version of `Sales_item` that defined two constructors:

让我们再看看定义了两个构造函数的 `Sales_item` 版本：

```
class Sales_item {
public:
    // default argument for book is the empty string
    Sales_item(const std::string &book = "") :
        isbn(book), units_sold(0), revenue(0.0) { }
    Sales_item(std::istream &is);
    // as before
};
```

Each of these constructors defines an implicit conversion. Accordingly, we can use a `string` or an `istream` where an object of type `Sales_item` is expected:

这里的每个构造函数都定义了一个隐式转换。因此，在期待一个 `Sales_item` 类型对象的地方，可以使用一个 `string` 或一个 `istream`:

```
string null_book = "9-999-99999-9";
// ok: builds a Sales_item with 0 units_sold and revenue from
// and isbn equal to null_book
item.same_isbn(null_book);
```

This program uses an object of type `string` as the argument to the `Sales_item same_isbn` function. That function expects a `Sales_item` object as its argument. The compiler uses the `Sales_item` constructor that takes a `string` to generate a new `Sales_item` object from `null_book`. That newly generated (temporary) `Sales_item` is passed to `same_isbn`.

这段程序使用一个 `string` 类型对象作为实参传给 `Sales_item` 的 `same_isbn` 函数。该函数期待一个 `Sales_item` 对象作为实参。编译器使用接受一个 `string` 的 `Sales_item` 构造函数从 `null_book` 生成一个新的 `Sales_item` 对象。新生成的（临时的）`Sales_item` 被传递给 `same_isbn`。

Whether this behavior is desired depends on how we think our users will use the conversion. In this case, it might be a good idea. The `string` in `book` probably represents a nonexistent `ISBN`, and the call to `same_isbn` can detect whether the `Sales_item` in `item` represents a null `Sales_item`. On the other hand, our user might have mistakenly called `same_isbn` on `null_book`.

这个行为是否我们想要的，依赖于我们认为用户将如何使用这个转换。在这种情况下，它可能是一个好主意。`book` 中的 `string` 可能代表一个不存在的 `ISBN`，对 `same_isbn` 的调用可以检测 `item` 中的 `Sales_item` 是否表示一个空的 `Sales_item`。另一方面，用户也许在 `null_book` 上错误地调用了 `same_isbn`。

More problematic is the conversion from `istream` to `Sales_item`:

更成问题的是从 `istream` 到 `Sales_item` 的转换：

Section 12.4. Constructors

```
// ok: uses the Sales_item istream constructor to build an object
// to pass to same_isbn
item.same_isbn(cin);
```

This code implicitly converts `cin` to a `Sales_item`. This conversion executes the `Sales_item` constructor that takes an `istream`. That constructor creates a (temporary) `Sales_item` object by reading the standard input. That object is then passed to `same_isbn`.

这段代码将 `cin` 隐式转换为 `Sales_item`。这个转换执行接受一个 `istream` 的 `Sales_item` 构造函数。该构造函数通过读标准输入来创建一个（临时的）`Sales_item` 对象。然后该对象被传递给 `same_isbn`。

This `Sales_item` object is a temporary (Section 7.3.2, p. 247). We have no access to it once `same_isbn` finishes. Effectively, we have constructed an object that is discarded after the test is complete. This behavior is almost surely a mistake.

这个 `Sales_item` 对象是一个临时对象（第 7.3.2 节）。一旦 `same_isbn` 结束，就不能再访问它。实际上，我们构造了一个在测试完成后被丢弃的对象。这个行为几乎肯定是一个错误。

Suppressing Implicit Conversions Defined by Constructors

抑制由构造函数定义的隐式转换

We can prevent the use of a constructor in a context that requires an implicit conversion by declaring the constructor `explicit`:

可以通过将构造函数声明为 `explicit`，来防止在需要隐式转换的上下文中使用构造函数：

```
class Sales_item {
public:
    // default argument for book is the empty string
    explicit Sales_item(const std::string &book = ""):
        isbn(book), units_sold(0), revenue(0.0) { }
    explicit Sales_item(std::istream &is);
    // as before
};
```

The `explicit` keyword is used only on the constructor declaration inside the class. It is not repeated on a definition made outside the class body:

`explicit` 关键字只能用于类内部的构造函数声明上。在类的定义体外部所做的定义上不再重复它：

```
// error: explicit allowed only on constructor declaration in class header
explicit Sales_item::Sales_item(istream& is)
{
    is >> *this; // uses Sales_item::operator>> to read the members
}
```

Now, neither constructor can be used to implicitly create a `Sales_item` object. Neither of our previous uses will compile:

现在，两个构造函数都不能用于隐式地创建对象。前两个使用都不能编译：

```
item.same_isbn(null_book); // error: string constructor is explicit
item.same_isbn(cin);       // error: istream constructor is explicit
```



When a constructor is declared `explicit`, the compiler will *not* use it as a conversion operator.

当构造函数被声明 `explicit` 时，编译器将不使用它作为转换操作符。

Explicitly Using Constructors for Conversions

为转换而显式地使用构造函数

Section 12.4. Constructors

An [explicit constructor](#) can be used to generate a conversion as long as we do so explicitly:

```
string null_book = "9-999-99999-9";
// ok: builds a Sales_item with 0 units_sold and revenue from
// and isbn equal to null_book
item.same_isbn(Sales_item(null_book));
```

In this code, we create a `Sales_item` from `null_book`. Even though the constructor is `explicit`, this usage is allowed. Making a constructor `explicit` turns off only the use of the constructor implicitly. Any constructor can be used to explicitly create a temporary object.

在这段代码中，从 `null_book` 创建一个 `Sales_item`。尽管构造函数为显式的，但这个用法是允许的。显式使用构造函数只是中止了隐式地使用构造函数。任何构造函数都可以用来显式地创建临时对象。



Ordinarily, single-parameter constructors should be `explicit` unless there is an obvious reason to want to define an implicit conversion. Making constructors `explicit` may avoid mistakes, and a user can explicitly construct an object when a conversion is useful.

通常，除非有明显的理由想要定义隐式转换，否则，单形参构造函数应该为 `explicit`。将构造函数设置为 `explicit` 可以避免错误，并且当转换有用时，用户可以显式地构造对象。

Exercises Section 12.4.4

Exercise 12.28: Explain whether the `Sales_item` constructor that takes a `string` should be explicit. What would be the benefits of making the constructor `explicit`? What would be the drawbacks?

解释一下接受一个 `string` 的 `Sales_item` 构造函数是否应该为 `explicit`。将构造函数设置为 `explicit` 的好处是什么？缺点是什么？

Exercise 12.29: Explain what operations happen during the following definitions:

解释在下面的定义中所发生的操作。

```
string null_isbn = "9-999-99999-9";
Sales_item null1(null_isbn);
Sales_item null("9-999-99999-9");
```

Exercise 12.30: Compile the following code:

编译如下代码：

```
f(const vector<int>&);
int main() {
    vector<int> v2;
    f(v2); // should be ok
    f(42); // should be an error
    return 0;
}
```

What can we infer about the `vector` constructors based on the error on the second call to `f`? If the call succeeded, then what would you conclude?

基于对 `f` 的第二个调用中出现的错误，我们可以对 `vector` 构造函数作出什么推断？如果该调用成功了，那么你能得出什么结论？

12.4.5. Explicit Initialization of Class Members

12.4.5. 类成员的显式初始化

Although most objects are initialized by running an appropriate constructor, it is possible to initialize the data members of simple nonabstract classes directly. Members of classes that define no constructors and all of whose data members are `public` may be initialized in the same way that we initialize array elements:

尽管大多数对象可以通过运行适当的构造函数进行初始化，但是直接初始化简单的非抽象类的数据成员仍是可能的。对于没有定义构造函数并且其全体数据成员均为 `public` 的类，可以采用与初始化数组元素相同的方式初始化其成员：

```
struct Data {
    int ival;
    char *ptr;
};

// val1.ival = 0; val1.ptr = 0
Data val1 = { 0, 0 };

// val2.ival = 1024;
// val2.ptr = "Anna Livia Plurabelle"
Data val2 = { 1024, "Anna Livia Plurabelle" };
```

The initializers are used in the declaration order of the data members. The following, for example, is an error because `ival` is declared before `ptr`:

根据数据成员的声明次序来使用初始化式。例如，因为 `ival` 在 `ptr` 之前声明，所以下面的用法是错误的：

```
// error: can't use "Anna Livia Plurabelle" to initialize the int ival
Data val2 = { "Anna Livia Plurabelle" , 1024 };
```

This form of initialization is inherited from C and is supported for compatibility with C programs. There are three significant drawbacks to explicitly initializing the members of an object of class type:

这种形式的初始化从 C 继承而来，支持与 C 程序兼容。显式初始化类类型对象的成员有三个重大的缺点。

1. It requires that all the data members of the class be `public`.

要求类的全体数据成员都是 `public`。

2. It puts the burden on the programmer to initialize every member of every object. Such initialization is tedious and error-prone because it is easy to forget an initializer or to supply an inappropriate initializer.

将初始化每个对象的每个成员的负担放在程序员身上。这样的初始化是乏味且易于出错的，因为容易遗忘初始化式或提供不适当的初始化式。

3. If a member is added or removed, all initializations have to be found and updated correctly.

如果增加或删除一个成员，必须找到所有的初始化并正确更新。



It is almost always better to define and use constructors. When we provide a default constructor for the types we define, we allow the compiler to automatically run that constructor, ensuring that every class object is properly initialized prior to the first use of that object.

定义和使用构造函数几乎总是较好的。当我们为自己定义的类型提供一个默认构造函数时，允许编译器自动运行那个构造函数，以保证每个类对象在初次使用之前正确地初始化。

Exercises Section 12.4.5

Exercise 12.31: The data members of `pair` are `public`, yet this code doesn't compile. Why?

`pair` 的数据成员为 `public`，然而下面这段代码却不能编译，为什么？

```
pair<int, int> p2 = { 0, 42 }; // doesn't compile, why?
```


12.5. Friends

12.5. 友元

In some cases, it is convenient to let specific nonmember functions access the [private members](#) of a class while still preventing general access. For example, over-loaded operators, such as the input or output operators, often need access to the private data members of a class. For reasons we'll see in [Chapter 14](#) these operators might not be members of the class. Yet, even if they are not members of the class, they are "part of the interface" to the class.

在某些情况下，允许特定的非成员函数访问一个类的私有成员，同时仍然阻止一般的访问，这是很方便做到的。例如，被重载的操作符，如输入或输出操作符，经常需要访问类的私有数据成员。这些操作符不可能为类的成员，具体原因参见[第十四章](#)。然而，尽管不是类的成员，它们仍是类的“接口的组成部分”。

The [friend](#) mechanism allows a class to grant access to its nonpublic members to specified functions or classes. A friend declaration begins with the keyword [friend](#). It may appear only within a class definition. Friend declarations may appear anywhere in the class: Friends are not members of the class granting friendship, and so they are not affected by the access control of the section in which they are declared.

[友元](#)机制允许一个类将对其非公有成员的访问权授予指定的函数或类。友元的声明以关键字 [friend](#) 开始。它只能出现在类定义的内部。友元声明可以出现在类中的任何地方：友元不是授予友元关系的那个类的成员，所以它们不受声明出现部分的访问控制影响。



Ordinarily it is a good idea to group friend declarations together either at the beginning or end of the class definition.

通常，将友元声明成组地放在类定义的开始或结尾是个好主意。

Friendship: An Example

友元关系：一个例子

Imagine that in addition to the [Screen](#) class we had a window manager that manages a group of [Screens](#) on a given display. That class logically might need access to the internal data of the [Screen](#) objects it manages. Assuming that [Window_Mgr](#) is the name of the window-management class, [Screen](#) could let [Window_Mgr](#) access its members as follows:

想像一下，除了 [Screen](#) 类之外，还有一个窗口管理器，管理给定显示器上的一组 [Screen](#)。窗口管理类在逻辑上可能需要访问由其管理的 [Screen](#) 对象的内部数据。假定 [Window_Mgr](#) 是该窗口管理类的名字，[Screen](#) 应该允许 [Window_Mgr](#) 像下面这样访问其成员：

```
class Screen {
    // Window_Mgr members can access private parts of class Screen
    friend class Window_Mgr;
    // ...rest of the Screen class
};
```

The members of [Window_Mgr](#) can refer directly to the [private](#) members of [Screen](#). For example, [Window_Mgr](#) might have a function to relocate a [Screen](#):

[Window_Mgr](#) 的成员可以直接引用 [Screen](#) 的私有成员。例如，[Window_Mgr](#) 可以有一个函数来重定位一个 [Screen](#):

```
Window_Mgr&
Window_Mgr::relocate(Screen::index r, Screen::index c,
                     Screen& s)
{
    // ok to refer to height and width
    s.height += r;
    s.width += c;

    return *this;
}
```

In absence of the friend declaration, this code would be in error: It would not be allowed to use the [height](#) and [width](#) members of its parameter

Section 12.5. Friends

named `s`. Because `Screen` grants friendship to `Window_Mgr`, all the members of `Screen` are accessible to the functions in `Window_Mgr`. 缺少友元声明时，这段代码将会出错：将不允许使用形参 `s` 的 `height` 和 `width` 成员。因为 `Screen` 将友元关系授予 `Window_Mgr`，所以，`Window_Mgr` 中的函数都可以访问 `Screen` 的所有成员。

A friend may be an ordinary, nonmember function, a member function of another previously defined class, or an entire class. In making a class a friend, all the member functions of the friend class are given access to the nonpublic members of the class granting friendship.

友元可以是普通的非成员函数，或前面定义的其他类的成员函数，或整个类。将一个类设为友元，友元类的所有成员函数都可以访问授予友元关系的那个类的非公有成员。

Making Another Class' Member Function a Friend

使其他类的成员函数成为友元

Instead of making the entire `Window_Mgr` class a friend, `Screen` could have specified that only the `relocate` member was allowed access:

如果不是将整个 `Window_Mgr` 类设为友元，`Screen` 就可以指定只允许 `relocate` 成员访问：

```
class Screen {  
    // Window_Mgr must be defined before class Screen  
    friend Window_Mgr&  
        Window_Mgr::relocate(Window_Mgr::index,  
                            Window_Mgr::index,  
                            Screen&);  
    // ...rest of the Screen class  
};
```

When we declare a member function to be a friend, the name of the function must be qualified by the name of the class of which it is a member.

当我们将成员函数声明为友元时，函数名必须用该函数所属的类名字加以限定。

Friend Declarations and Scope

友元声明与作用域

Interdependencies among friend declarations and the definitions of the friends can require some care in order to structure the classes correctly. In the previous example, class `Window_Mgr` must have been defined. Otherwise, class `Screen` could not name a `Window_Mgr` function as a friend. However, the `relocate` function itself can't be defined until class `Screen` has been defined after all, it was made a friend in order to access the members of class `Screen`.

为了正确地构造类，需要注意友元声明与友元定义之间的互相依赖。在前面的例子中，类 `Window_Mgr` 必须先定义。否则，`Screen` 类就不能将一个 `Window_Mgr` 函数指定为友元。然而，只有在定义类 `Screen` 之后，才能定义 `relocate` 函数——毕竟，它被设为友元是为了访问类 `Screen` 的成员。

More generally, to make a member function a friend, the class containing that member must have been defined. On the other hand, a class or nonmember function need not have been declared to be made a friend.

更一般地讲，必须先定义包含成员函数的类，才能将成员函数设为友元。另一方面，不必预先声明类和非成员函数来将它们设为友元。



A friend declaration introduces the named class or nonmember function into the surrounding scope. Moreover, a friend function may be *defined* inside the class. The scope of the function is exported to the scope enclosing the class definition.

友元声明将已命名的类或非成员函数引入到外围作用域中。此外，友元函数可以在类的内部定义，该函数的作用域扩展到包围该类定义的作用域。

Class names and functions (definitions or declarations) introduced in a friend can be used as if they had been previously declared:

用友元引入的类名和函数（定义或声明），可以像预先声明的一样使用：

```
class X {  
    friend class Y;  
    friend void f() { /* ok to define friend function in the class body */ }  
};  
class Z {
```

Section 12.5. Friends

```
Y *ymem; // ok: declaration for class Y introduced by friend in X  
void g() { return ::f(); } // ok: declaration of f introduced by X  
};
```

Overloaded Functions and Friendship

重载函数与友元关系

A class must declare as a friend each function in a set of overloaded functions that it wishes to make a friend:

类必须将重载函数集中每一个希望设为友元的函数都声明为友元：

```
// overloaded storeOn functions  
extern std::ostream& storeOn(std::ostream &, Screen &);  
extern BitMap& storeOn(BitMap &, Screen &);  
class Screen {  
    // ostream version of storeOn may access private parts of Screen objects  
    friend std::ostream& storeOn(std::ostream &, Screen &);  
    // ...  
};
```

Class `Screen` makes the version of `storeOn` that takes an `ostream&` its friend. The version that takes a `BitMap&` has no special access to `Screen`.

类 `Screen` 将接受一个 `ostream&` 的 `storeOn` 版本设为自己的友元。接受一个 `BitMap&` 的版本对 `Screen` 没有特殊访问权。

Exercises Section 12.5

Exercise What is a friend function? A friend class?

12.32: 什么是友元函数？什么是友元类？

Exercise When are friends useful? Discuss the pros and cons of using friends.

12.33: 什么时候友元是有用的？讨论使用友元的优缺点。

Exercise Define a nonmember function that adds two `Sales_item` objects.

12.34: 定义一个增加两个 `Sales_item` 对象的非成员函数。

Exercise Define a nonmember function that reads an `istream` and stores what it reads into a `Sales_item`.

12.35: 定义一个非成员函数，读取一个 `istream` 并将读入的内容存储到一个 `Sales_item` 中。

12.6. **static** Class Members

12.6. **static** 类成员

It is sometimes necessary for all the objects of a particular class type to access a global object. Perhaps a count is needed of how many objects of a particular class type have been created at any one point in the program, or the global object may be a pointer to an error-handling routine for the class, or it may be a pointer to the free-store memory for objects of this class type.

对于特定类类型的全体对象而言，访问一个全局对象有时是必要的。也许，在程序的任意点需要统计已创建的特定类类型对象的数量；或者，全局对象可能是指向类的错误处理例程的一个指针；或者，它是指向类类型对象的内在自由存储区的一个指针。

However, making the object global violates encapsulation: The object exists to support the implementation of a particular class abstraction. If the object is global, general user code can modify the value. Rather than defining a generally accessible global object, a class can define a [class static member](#).

然而，全局对象会破坏封装：对象需要支持特定类抽象的实现。如果对象是全局的，一般的用户代码就可以修改这个值。类可以定义类 静态成员，而不是定义一个可普遍访问的全局对象。

Ordinary, non**static** data members exist in each object of the class type. Unlike ordinary data members, a **static** data member exists independently of any object of its class; each **static** data member is an object associated with the class, not with the objects of that class.

通常，非 **static** 数据成员存在于类类型的每个对象中。不像普通的数据成员，**static** 数据成员独立于该类的任意对象而存在；每个 **static** 数据成员是与类关联的对象，并不与该类的对象相关联。

Just as a class may define shared **static** data members, it may also define **static** member functions. A **static** member function has no **this** parameter. It may directly access the **static** members of its class but may not directly use the non**static** members.

正如类可以定义共享的 **static** 数据成员一样，类也可以定义 **static** 成员函数。**static** 成员函数没有 **this** 形参，它可以直接访问所属类的 **static** 成员，但不能直接使用非 **static** 成员。

Advantages of Using Class **static** Members

使用类的 **static** 成员的优点

There are three advantages to using **static** members rather than globals:

使用 **static** 成员而不是全局对象有三个优点。

1. The name of a **static** member is in the scope of the class, thereby avoiding name collisions with members of other classes or global objects.
static 成员的名字是在类的作用域中，因此可以避免与其他类的成员或全局对象名字冲突。
2. Encapsulation can be enforced. A **static** member can be a private member; a global object cannot.
可以实施封装。**static** 成员可以是私有成员，而全局对象不可以。
3. It is easy to see by reading the program that a **static** member is associated with a particular class. This visibility clarifies the programmer's intentions.
通过阅读程序容易看出 **static** 成员是与特定类关联的。这种可见性可清晰地显示程序员的意图。

Defining **static** Members

定义 **static** 成员

A member is made **static** by prefixing the member declaration with the keyword **static**. The **static** members obey the normal public/private access rules.

在成员声明前加上关键字 **static** 将成员设为 **static**。**static** 成员遵循正常的公有／私有访问规则。

Section 12.6. static Class Members

As an example, consider a simple class intended to represent a bank account. Each account has a balance and an owner. Each account earns interest monthly, but the interest rate applied to each account is always the same. We could write this class as

例如，考虑一个简单的表示银行账户的类。每个账户具有余额和拥有者，并且按月获得利息，但应用于每个账户的利率总是相同的。可以按下面的这样编写这个类

```
class Account {  
public:  
    // interface functions here  
    void applyint() { amount += amount * interestRate; }  
    static double rate() { return interestRate; }  
    static void rate(double); // sets a new rate  
private:  
    std::string owner;  
    double amount;  
    static double interestRate;  
    static double initRate();  
};
```

Each object of this class has two data members: `owner` and `amount`. Objects do not have data members that correspond to `static` data members. Instead, there is a single `interestRate` object that is shared by all objects of type `Account`.

这个类的每个对象具有两个数据成员：`owner` 和 `amount`。对象没有与 `static` 数据成员对应的数据成员，但是，存在一个单独的 `interestRate` 对象，由 `Account` 类型的全体对象共享。

Using a Class `static` Member

使用类的 `static` 成员

A `static` member can be invoked directly from the class using the scope operator or indirectly through an object, reference, or pointer to an object of its class type.

可以通过作用域操作符从类直接调用 `static` 成员，或者通过对象、引用或指向该类类型对象的指针间接调用。

```
Account ac1;  
Account *ac2 = &ac1;  
// equivalent ways to call the static member rate function  
double rate;  
rate = ac1.rate();      // through an Account object or reference  
rate = ac2->rate();   // through a pointer to an Account object  
rate = Account::rate(); // directly from the class using the scope operator
```

As with other members, a class member function can refer to a class `static` member without the use of the scope operator:

像使用其他成员一样，类成员函数可以不用作用域操作符来引用类的 `static` 成员：

```
class Account {  
public:  
    // interface functions here  
    void applyint() { amount += amount * interestRate; }  
};
```

Exercises Section 12.6

Exercise 12.36: What is a `static` class member? What are the advantages of `static` members? How do they differ from ordinary members?

什么是 `static` 类成员？`static` 成员的优点是什么？它们与普通有什么不同？

Exercise 12.37: Write your own version of the `Account` class.

编写自己的 `Account` 类版本。

12.6.1. `static` Member Functions

12.6.1. **static** 成员函数

Our `Account` class has two `static` member functions named `rate`, one of which was defined inside the class. When we define a `static` member outside the class, we do not respecify the `static` keyword. The keyword appears only with the declaration inside the class body:

`Account` 类有两个名为 `rate` 的 `static` 成员函数，其中一个定义在类的内部。当我们在类的外部定义 `static` 成员时，无须重复指定 `static` 保留字，该保留字只出现在类定义体内部的声明处：

```
void Account::rate(double newRate)
{
    interestRate = newRate;
}
```

static Functions Have No `this` Pointer

static 函数没有 `this` 指针

A `static` member is part of its class but not part of any object. Hence, a `static` member function does not have a `this` pointer. Referring to `this` either explicitly or implicitly by using a non`static` member is a compile-time error.

`static` 成员是类的组成部分但不是任何对象的组成部分，因此，`static` 成员函数没有 `this` 指针。通过使用非 `static` 成员显式或隐式地引用 `this` 是一个编译时错误。

Because a `static` member is not part of any object, `static` member functions may not be declared as `const`. After all, declaring a member function as `const` is a promise not to modify the object of which the function is a member. Finally, `static` member functions may also not be declared as `virtual`. We'll learn about `virtual` functions in [Section 15.2.4](#) (p. 566).

因为 `static` 成员不是任何对象的组成部分，所以 `static` 成员函数不能被声明为 `const`。毕竟，将成员函数声明为 `const` 就是承诺不会修改该函数所属的对象。最后，`static` 成员函数也不能被声明为虚函数。我们将在[第 15.2.4 节](#)学习虚函数。

Exercises Section 12.6.1

Exercise

12.38:

Define a class named `Foo` that has a single data member of type `int`. Give the class a constructor that takes an `int` value and initializes the data member from that value. Give it a function that returns the value of its data member.

定义一个命名为 `Foo` 的类，具有单个 `int` 型数据成员。为该类定义一个构造函数，接受一个 `int` 值并用该值初始化数据成员。为该类定义一个函数，返回其数据成员的值。

Exercise

12.39:

Given the class `Foo` defined in the previous exercise, define another class `Bar` with two `static` data elements: one of type `int` and another of type `Foo`.

给定上题中定义的 `Foo` 类定义另一个 `Bar` 类。`Bar` 类具有两个 `static` 数据成员：一个为 `int` 型，另一个为 `Foo` 类型。

Exercise

12.40:

Using the classes from the previous two exercises, add a pair of `static` member functions to class `Bar`. The first `static`, named `FooVal`, should return the value of class `Bar`'s `static` member of type `Foo`. The second member, named `callsFooVal`, should keep a count of how many times `xval` is called.

使用上面两题中定义的类，给 `Bar` 类增加一对成员函数：第一个成员命名为 `FooVal`，返回 `Bar` 类的 `Foo` 类型 `static` 成员的值；第二个成员命名为 `callsFooVal`，保存 `xval` 被调用的次数。

12.6.2. **static** Data Members

12.6.2. **static** 数据成员

`static` data members can be declared to be of any type. They can be `consts`, references, arrays, class types, and so forth.

Section 12.6. static Class Members

`static` 数据成员可以声明为任意类型，可以是常量、引用、数组、类类型，等等。

`static` data members must be defined (exactly once) outside the class body. Unlike ordinary data members, `static` members are not initialized through the class constructor(s) and instead should be initialized when they are defined.

`static` 数据成员必须在类定义体的外部定义（正好一次）。不像普通数据成员，`static` 成员不是通过类构造函数进行初始化，而是应该在定义时进行初始化。



The best way to ensure that the object is defined exactly once is to put the definition of `static` data members in the same file that contains the definitions of the class noninline member functions.

保证对象正好定义一次的最好办法，就是将 `static` 数据成员的定义放在包含类非内联成员函数定义的文件中。

`static` data members are defined in the same way that other class members and other variables are defined. The member is defined by naming its type followed by the fully qualified name of the member.

定义 `static` 数据成员的方式与定义其他类成员和变量的方式相同：先指定类型名，接着是成员的完全限定名。

We might define `interestRate` as follows:

可以定义如下 `interestRate`:

```
// define and initialize static class member
double Account::interestRate = initRate();
```

This statement defines the `static` object named `interestRate` that is a member of class `Account` and has type `double`. Like other member definitions, the definition of a `static` member is in class scope once the member name is seen. As a result, we can use the `static` member function named `initRate` directly without qualification as the initializer for `rate`. Note that even though `initRate` is `private`, we can use this function to initialize `interestRate`. The definition of `interestRate`, like any other member definition, is in the scope of the class and hence has access to the `private` members of the class.

这个语句定义名为 `interestRate` 的 `static` 对象，它是类 `Account` 的成员，为 `double` 型。像其他成员定义一样，一旦成员名出现，`static` 成员的就是在类作用域中。因此，我们可以没有限制地直接使用名为 `initRate` 的 `static` 成员函数，作为 `interestRate` 初始化式。注意，尽管 `initRate` 是私有的，我们仍然可以使用该函数来初始化 `interestRate`。像任意的其他成员定义一样，`interestRate` 的定义是在类的作用域中，因此可以访问该类的私有成员。



As with any class member, when we refer to a class `static` member outside the class body, we must specify the class in which the member is defined. The `static` keyword, however, is used *only* on the declaration inside the class body. Definitions are not labeled `static`.

像使用任意的类成员一样，在类定义体外部引用类的 `static` 成员时，必须指定成员是在哪个类中定义的。然而，`static` 关键字只能用于类定义体内部的声明中，定义不能标示为 `static`。

Integral `const static` Members Are Special

特殊的整型 `const static` 成员

Ordinarily, class `static` members, like ordinary data members, cannot be initialized in the class body. Instead, `static` data members are normally initialized when they are defined.

一般而言，类的 `static` 成员，像普通数据成员一样，不能在类的定义体中初始化。相反，`static` 数据成员通常在定义时才初始化。

One exception to this rule is that a `const static` data member of integral type can be initialized within the class body as long as the initializer is a constant expression:

这个规则的一个例外是，只要初始化式是一个常量表达式，整型 `const static` 数据成员就可以在类的定义体中进行初始化：

```
class Account {
public:
    static double rate() { return interestRate; }
    static void rate(double); // sets a new rate
private:
    static const int period = 30; // interest posted every 30 days
    double daily_tbl[period]; // ok: period is constant expression
};
```

Section 12.6. static Class Members

A `const static` data member of integral type initialized with a constant value is a constant expression. As such, it can be used where a constant expression is required, such as to specify the dimension for the array member `daily_tbl`.

用常量值初始化的整型 `const static` 数据成员是一个常量表达式。同样地，它可以用在任何需要常量表达式的地方，例如指定数组成员 `daily_tbl` 的维。



When a `const static` data member is initialized in the class body, the data member must still be defined outside the class definition.

`const static` 数据成员在类的定义体中初始化时，该数据成员仍必须在类的定义体之外进行定义。

When an initializer is provided inside the class, the definition of the member must not specify an initial value:

在类内部提供初始化式时，成员的定义不必再指定初始值：

```
// definition of static member with no initializer;  
// the initial value is specified inside the class definition  
const int Account::period;
```

static Members Are Not Part of Class Objects

`static` 成员不是类对象的组成部分

Ordinary members are part of each object of the given class. `static` members exist independently of any object and are not part of objects of the class type. Because `static` data members are not part of any object, they can be used in ways that would be illegal for non`static` data members.

普通成员都是给定类的每个对象的组成部分。`static` 成员独立于任何对象而存在，不是类类型对象的组成部分。因为 `static` 数据成员不是任何对象的组成部分，所以它们的使用方式对于非 `static` 数据成员而言是不合法的。

As an example, the type of a `static` data member can be the class type of which it is a member. A non`static` data member is restricted to being declared as a pointer or a reference to an object of its class:

例如，`static` 数据成员的类型可以是该成员所属的类类型。非 `static` 成员被限定声明为其自身类对象的指针或引用：

```
class Bar {  
public:  
    // ...  
private:  
    static Bar mem1; // ok  
    Bar *mem2;      // ok  
    Bar mem3;       // error  
};
```

Similarly, a `static` data member can be used as a default argument:

类似地，`static` 数据成员可用作默认实参：

```
class Screen {  
public:  
    // bkground refers to the static member  
    // declared later in the class definition  
    Screen& clear(char = bkground);  
private:  
    static const char bkground = '#';  
};
```

A non`static` data member may not be used as a default argument because its value cannot be used independently of the object of which it is a part. Using a non`static` data member as a default argument provides no object from which to obtain the member's value and so is an error.

非 `static` 数据成员不能用作默认实参，因为它的值不能独立于所属的对象而使用。使用非 `static` 数据成员作默认实参，将无法提供对象以获取该成员的值，因而是错误的。

Exercises Section 12.6.2

Exercise 12.41: Given the classes `Foo` and `Bar` that you wrote for the exercises to [Section 12.6.1](#) (p. 470), initialize the `static` members of `Foo`. Initialize the `int` member to 20 and the `Foo` member to 0.

利用第 12.6.1 节的习题中编写的类 `Foo` 和 `Bar`，初始化 `Foo` 的 `static` 成员。将 `int` 成员初始化为 20，并将 `Foo` 成员初始化为 0。

Exercise 12.42: Which, if any, of the following `static` data member declarations and definitions are errors? Explain why.

下面的 `static` 数据成员声明和定义中哪些是错误的（如果有的话）？解释为什么。

```
// example.h
class Example {
public:
    static double rate = 6.5;

    static const int vecSize = 20;
    static vector<double> vec(vecSize);
};

// example.C
#include "example.h"
double Example::rate;
vector<double> Example::vec;
```

Chapter Summary

小结

Classes are the most fundamental feature in C++. Classes let us define new types that are tailored to our own applications, making our programs shorter and easier to modify.

类是 C++ 中最基本的特征，允许定义新的类型以适应应用程序的需要，同时使程序更短且更易于修改。

Data abstraction—the ability to define both data and function members—and encapsulation—the ability to protect class members from general access—are fundamental to classes. Member functions define the interface to the class. We encapsulate the class by making the data and functions used by the implementation of a class `private`.

数据抽象是指定义数据和函数成员的能力，而封装是指从常规访问中保护类成员的能力，它们都是类的基础。成员函数定义类的接口。通过将类的实现所用到的数据和函数设置为 `private` 来封装类。

Classes may define constructors, which are special member functions that control how objects of the class are initialized. Constructors may be overloaded. Every constructor should initialize every data member. Constructors should use a constructor initializer list to initialize the data members. Initializer lists are lists of name-value pairs where the name is a member and the value is an initial value for that member.

类可以定义构造函数，它们是特殊的成员函数，控制如何初始化类的对象。可以重载构造函数。每个构造函数就初始化每个数据成员。初始化列表包含的是名—值对，其中的名是一个成员，而值则是该成员的初始值。

Classes may grant access to their non`public` members to other classes or functions. A class grants access by making the class or function a friend.

类可以将对其非 `public` 成员的访问权授予其他类或函数，并通过将其他的类或函数设为友元来授予其访问权。

Classes may also define `mutable` or `static` members. A `mutable` member is a data member that is never `const`; its value may be changed inside a `const` member function. A `static` member can be either function or data; `static` members exist independently of the objects of the class type.

类也可以定义 `mutable` 或 `static` 成员。`mutable` 成员永远都不能为 `const`；它的值可以在 `const` 成员函数中修改。`static` 成员可以是函数或数据，独立于类类型的对象而存在。

Defined Terms

术语

abstract data type (抽象数据类型)

A data structure that uses encapsulation to hide its implementation, allowing programmers using the type to think *abstractly* about what the type does rather than *concretely* about how the type is represented. Classes in C++ can be used to define abstract data types.

使用封装来隐藏其实现的数据结构，允许使用类型的程序员抽象地考虑该类型做什么，而不是具体地考虑类型如何表示。C++ 中的类可用来定义抽象数据类型。

access label (访问标号)

A `public` or `private` label that defines whether the following members are accessible to users of the class or only to the friends and members of the class. Each label sets the access protection for the members declared up to the next label. Labels may appear multiple times within the class.

`public` 或 `private` 标号，指定后面的成员可以被类的使用者访问或者只能被类的友元和成员访问。每个标号为在该标号到下一个标号之间声明的成员设置访问保护。标号可以在类中出现多次。

class (类)

C++ mechanism for defining our own abstract data types. Classes may have data, function or type members. A class defines a new type and a new scope.

是 C++ 中定义抽象数据类型的一种机制，可以有数据、函数或类型成员。一个类定义了新的类型和新的作用域。

class declaration (类声明)

A class may be declared before it is defined. A class declaration is the keyword `class` (or `struct`) followed by the class name followed by a semicolon. A class that is declared but not defined is an incomplete type.

类可以在定义之前声明。类声明用关键字 `class` (或 `struct`) 表示，后面加类名字和一个分号。已声明但没有定义的类是一个不完全的类型。

class keyword (class 关键字)

In a class defined following the `class` keyword, the initial implicit access label is `private`.

用在 `class` 关键字定义的类中，初始的隐式访问标号是 `private`。

class scope (类作用域)

Each class defines a scope. Class scopes are more complicated than other scopesmember functions defined within the class body may use names that appear after the definition.

每个类定义一个作用域。类作用域比其他作用域复杂得多——在类的定义体内定义的成员函数可以使用出现在该定义之后的名字。

concrete class (具体类)

A class that exposes its implementation.

暴露其实现细节的类。

const member function (常量成员函数)

A member function that may not change an object's ordinary (i.e., neither `static` nor `mutable`) data members. The `this` pointer in a `const` member is a pointer to `const`. A member function may be overloaded based on whether the function is `const`.

一种成员函数，不能改变对象的普通（即，既不是 `static` 也不是 `mutable`）数据成员。`const` 成员中的 `this` 指针指向 `const` 对象。成员函数是否可以被重载取决于该函数是否为 `const`。

constructor initializer list (构造函数初始化列表)

Specifies initial values of the data members of a class. The members are initialized to the values specified in the initializer list before the

Keyterm Defined Terms

body of the constructor executes. Class members that are not initialized in the initializer list are implicitly initialized by using their default constructor.

指定类的数据成员的初始值。在构造函数体现执行前，用初始化列表中指定的值初始化成员。没有在初始化列表中初始化的类成员，使用它们的默认构造函数隐式初始化。

conversion constructor (转换构造函数)

A non`explicit` constructor that can be called with a single argument. A conversion constructor is used implicitly to convert from the argument's type to the class type.

可用单个实参调用的非 `explicit` 构造函数。隐式使用转换构造函数将实参的类型转换为类类型。

data abstraction (数据抽象)

Programming technique that focuses on the interface to a type. Data abstraction allows programmers to ignore the details of how a type is represented and to think instead about the operations that the type can perform. Data abstraction is fundamental to both object-oriented and generic programming.

注重类型接口的编程技术。数据抽象允许程序员忽略类型如何表示的细节，而只考虑该类型可以执行的操作。数据抽象是面向对象编程和泛型编程的基础。

default constructor (默认构造函数)

The constructor that is used when no initializer is specified.

没有指定初始化时使用的构造函数。

encapsulation (封装)

Separation of implementation from interface; encapsulation hides the implementation details of a type. In C++, encapsulation is enforced by preventing general user access to the `private` parts of a class.

实现与接口的分离。封闭隐藏了类型的实现细节。在 C++ 中，实施封装可以阻止普通用户访问类的 `private` 部分。

explicit constructor (显式构造函数)

Constructor that can be called with a single argument but that may not be used to perform an implicit conversion. A constructor is made explicit by prepending the keyword `explicit` to its declaration.

可以用单个实参调用但不能用于执行隐式转换的构造函数。通过将关键字 `explicit` 放在构造函数的声明之前而将其设置为 `explicit`。

forward declaration (前向声明)

Declaration of an as yet undefined name. Most often used to refer to the declaration of a class that appears prior to the definition of that class. See incomplete type.

对尚未定义的名字的声明。大多用于引用出现在类定义之前的类声明。参见不完全类型。

friend (友元)

Mechanism by which a class grants access to its non`public` members. Both classes and functions may be named as `friends`. `friends` have the same access rights as members.

类授权访问其非 `public` 成员的机制。类和函数都可以被指定为友元。友元拥有与成员一样的访问权。

incomplete type (不完全类型)

A type that has been declared but not yet defined. It is not possible use an incomplete type to define a variable or class member. It is legal to define references or pointers to incomplete types.

已声明但未定义的类型。不能使用不完全类型来定义变量或类成员。定义指向不完全类型的引用或指针是合法的。

member function (成员函数)

Class member that is a function. Ordinary member functions are bound to an object of the class type through the implicit `this` pointer. Static member functions are not bound to an object and have no `this` pointer. Member functions may be overloaded, provided that the versions of the function are distinguished by number or type of their parameters.

类的函数成员。普通成员函数通过隐式的 `this` 指针绑定到类类型的对象。`static` 成员函数不与对象绑定且没有 `this` 指针。成员函数可以被重载，只要该函数的版本可由形参的数目或类型来区别。

mutable data member (可变数据成员)

Data member that is never `const`, even when it is a member of a `const` object. A `mutable` member can be changed inside a `const` function.

一种永远也不能为 `const` 对象的数据成员，即使作为 `const` 对象的成员，也不能为 `const` 对象。`mutable` 成员可以在 `const` 函数中改变。

[name lookup \(名字查找\)](#)

The process by which the use of a name is matched to its corresponding declaration.

将名字的使用与其相应的声明相匹配的过程。

[private members \(私有成员\)](#)

Members defined after a `private` access label; accessible only to the `friends` and other class members. Data members and utility functions used by the class that are not part of the type's interface are usually declared `private`.

在 `private` 访问标号之后定义的成员，只能被友元和其他的类成员访问。类所使用的数据成员和实用函数在不作为类型接口的组成部分时，通常声明为 `private`。

[public members \(公用成员\)](#)

Members defined after a `public` access label; `public` members are accessible to any user of the class. Ordinarily, only the functions that define the interface to the class should be defined in the `public` sections.

在 `public` 访问标号之后定义的成员，可被类的任意使用者访问。一般而言，只有定义类接口的函数应定义在 `public` 部分。

[static member \(静态成员\)](#)

Data or function member that is not a part of any object but is shared by all objects of a given class.

不是任意对象的组成部分、但由给定类的全体对象所共享的数据或函数成员。

[struct keyword \(struct 关键字\)](#)

In a class defined following the `struct` keyword, the initial implicit access label is `public`.

用在 `struct` 关键字定义的类中，初始的隐式访问标号为 `public`。

[synthesized default constructor \(合成的默认构造函数\)](#)

The default constructor created (synthesized) by the compiler for classes that do not define any constructors. This constructor initializes members of class type by running that class's default constructor; members of built-in type are uninitialized.

编译器为没有定义任何构造函数的类创建（合成）的构造函数。这个构造函数通过运行该类的默认构造函数来初始化类类型的成员，内置类型的成员不进行初始化。

Chapter 13. Copy Control

CONTENTS

<u>Section 13.1</u> The Copy Constructor	<u>476</u>
<u>Section 13.2</u> The Assignment Operator	<u>482</u>
<u>Section 13.3</u> The Destructor	<u>484</u>
<u>Section 13.4</u> A Message-Handling Example	<u>486</u>
<u>Section 13.5</u> Managing Pointer Members	<u>492</u>
<u>Chapter Summary</u>	<u>502</u>
<u>Defined Terms</u>	<u>502</u>

Each type, whether a built-in or class type, defines the meaning of a (possibly empty) set of operations on objects of that type. We can add two `int` values, run `size` on a `vector`, and so on. These operations define what can be done with objects of the given type.

每种类型，无论是内置类型还是类类型，都对该类型对象的一组（可能为空的）操作的含义进行了定义。比如，我们可以将两个 `int` 值相加，运行 `vector` 对象的 `size` 操作，等等。这些操作定义了用给定类型的对象可以完成什么任务。

Each type also defines what happens when objects of the type are created. Initialization of objects of class type is defined by constructors. Types also control what happens when objects of the type are copied, assigned, or destroyed. Classes control these actions through special member functions: the copy constructor, the assignment operator, and the destructor. This chapter covers these operations.

每种类型还定义了创建该类型的对象时会发生什么——构造函数定义了该类类型对象的初始化。类型还能控制复制、赋值或撤销该类型的对象时会发生什么——类通过特殊的成员函数：复制构造函数、赋值操作符和析构函数来控制这些行为。本章将介绍这些操作。

When we define a new type, we specify explicitly or implicitly what happens when objects of that type are copied, assigned, and destroyed. We do so by defining special members: the copy constructor, the assignment operator, and the destructor. If we do not explicitly define the copy constructor or the assignment operator, the compiler will (usually) define them for us.

当定义一个新类型的时候，需要显式或隐式地指定复制、赋值和撤销该类型的对象时会发生什么——这是通过定义特殊成员：复制构造函数、赋值操作符和析构函数来达到的。如果没有显式定义复制构造函数或赋值操作符，编译器（通常）会为我们定义。

The [copy constructor](#) is a special constructor that has a single parameter that is a (usually `const`) reference to the class type. The copy constructor is used explicitly when we define a new object and initialize it from an object of the same type. It is used implicitly when we pass or return objects of that type to or from functions.

复制构造函数是一种特殊构造函数，具有单个形参，该形参（常用 `const` 修饰）是对该类类型的引用。当定义一个新对象并用一个同类型的对象对它进行初始化时，将显式使用复制构造函数。当将该类型的对象传递给函数或函数返回该类型的对象时，将隐式使用复制构造函数。

The **destructor** is complementary to the constructors: It is applied automatically when an object goes out of scope or when a dynamically allocated object is deleted. The destructor is used to free resources acquired when the object was constructed or during the lifetime of the object. Regardless of whether a class defines its own destructor, the compiler automatically executes the destructors for the **nonstatic** data members of the class.

析构函数是构造函数的互补：当对象超出作用域或动态分配的对象被删除时，将自动应用析构函数。析构函数可用于释放对象时构造或在对象的生命期中所获取的资源。不管类是否定义了自己的析构函数，编译器都自动执行类中非 **static** 数据成员的析构函数。

We'll learn more about operator overloading in the next chapter, but in this chapter we cover the [assignment operator](#). Like constructors, the assignment operator may be overloaded by specifying different types for the right-hand operand. The version whose right-hand operand is of the class type is special: If we do not write one, the compiler will synthesize one for us.

在下一章我们将进一步学习操作符重载，本章中我们先介绍赋值操作符。与构造函数一样，赋值操作符可以通过指定不同类型的右操作数而重载。右操作数为类类型的版本比较特殊：如果我们没有编写这种版本，编译器将为我们合成一个。

Collectively, the copy constructor, assignment operator, and destructor are referred to as [copy control](#). The compiler automatically implements these operations, but the class may define its own versions.

复制构造函数、赋值操作符和析构函数总称为复制控制。编译器自动实现这些操作，但类也可以定义自己的版本。

Copy control is an essential part of defining any C++ class. Programmers new to C++ are often confused by having to define what happens when objects are copied, assigned, or destroyed. This confusion is



compounded because if we do not explicitly define these operations, the compiler defines them for us although they might not behave as we intend.

复制控制是定义任意 C++ 类必不可少的部分。初学 C++ 的程序员常对必须定义在复制、赋值或撤销对象时发生什么感到困惑。因为如果我们没有显式定义这些操作，编译器将为我们定义它们（尽管它们也许不像我们期望的那样工作），这往往使初学者更加困惑。

Often the compiler-synthesized copy-control functions are fine—they do exactly the work that needs to be done. But for some classes, relying on the default definitions leads to disaster. Frequently, the most difficult part of implementing the copy-control operations is recognizing when we need to override the default versions. One especially common case that requires the class to define its own copy-control members is if the class has a pointer member.

通常，编译器合成的复制控制函数是非常精练的——它们只做必需的工作。但对某些类而言，依赖于默认定义会导致灾难。实现复制控制操作最困难的部分，往往在于识别何时需要覆盖默认版本。有一种特别常见的情况需要类定义自己的复制控制成员的：类具有指针成员。

Team LiB

◀ PREVIOUS NEXT ▶

13.1. The Copy Constructor

13.1. 复制构造函数

The constructor that takes a single parameter that is a (usually `const`) reference to an object of the class type itself is called the copy constructor. Like the default constructor, the copy constructor can be implicitly invoked by the compiler. The copy constructor is used to

只有单个形参，而且该形参是对本类类型对象的引用（常用 `const` 修饰），这样的构造函数称为复制构造函数。与默认构造函数一样，复制构造函数可由编译器隐式调用。复制构造函数可用于：

- Explicitly or implicitly initialize one object from another of the same type

根据另一个同类型的对象显式或隐式初始化一个对象。

- Copy an object to pass it as an argument to a function

复制一个对象，将它作为实参传给一个函数。

- Copy an object to return it from a function

从函数返回时复制一个对象。

- Initialize the elements in a sequential container

初始化顺序容器中的元素。

- Initialize elements in an array from a list of element initializers

根据元素初始化式列表初始化数组元素。

Forms of Object Definition

对象的定义形式

Recall that C++ supports two forms of initialization ([Section 2.3.3](#), p. 48): direct and copy. Copy-initialization uses the `=` symbol, and direct-initialization places the initializer in parentheses.

回忆一下，C++ 支持两种初始化形式（[第 2.3.3 节](#)）：直接初始化和复制初始化。复制初始化使用 `=` 符号，而直接初始化将初始化式放在圆括号中。

The copy and direct forms of initialization, when applied to objects of class type, are subtly different. Direct-initialization directly invokes the constructor matched by the arguments. Copy-initialization always involves the copy constructor. Copy-initialization first uses the indicated constructor to create a temporary object ([Section 7.3.2](#), p. 247). It then uses the copy constructor to copy that temporary into the one we are creating:

当用于类类型对象时，初始化的复制形式和直接形式有所不同：直接初始化直接调用与实参匹配的构造函数，复制初始化总是调用复制构造函数。复制初始化首先使用指定构造函数创建一个临时对象（[第 7.3.2 节](#)），然后用复制构造函数将那个临时对象复制到正在创建的对象：

```
string null_book = "9-999-99999-9"; // copy-initialization
string dots(10, '.');                // direct-initialization

string empty_copy = string();          // copy-initialization
string empty_direct;                  // direct-initialization
```

For objects of class type, copy-initialization can be used only when specifying a single argument or when we explicitly build a temporary object to copy.

对于类类型对象，只有指定单个实参或显式创建一个临时对象用于复制时，才使用复制初始化。

When `dots` is created, the `string` constructor that takes a count and a character is called and directly initializes the members in `dots`. To create `null_book`, the compiler first creates a temporary by invoking the `string` constructor that takes a C-style character string parameter. The compiler then uses the `string` copy constructor to initialize `null_book` as a copy of that temporary.

Section 13.1. The Copy Constructor

创建 `dots` 时，调用参数为一个数量和一个字符的 `string` 构造函数并直接初始化 `dots` 的成员。创建 `null_book` 时，编译器首先调用接受一个 C 风格字符串形参的 `string` 构造函数，创建一个临时对象，然后，编译器使用 `string` 复制构造函数将 `null_book` 初始化为那个临时对象的副本。

The initialization of `empty_copy` and `empty_direct` both call the `string` default constructor. In the first case, the default constructor creates a temporary object, which is then used by the copy constructor to initialize `empty_copy`. In the second case, the default constructor is run directly on `empty_direct`.

`empty_copy` 和 `empty_direct` 的初始化都调用默认构造函数。对前者初始化时，默认构造函数函数创建一个临时对象，然后复制构造函数用该对象初始化 `empty_copy`。对后者初始化时，直接运行 `empty_direct` 的默认构造函数。

The copy form of initialization is primarily supported for compatibility with C usage. When it can do so, the compiler is permitted (but not obligated) to skip the copy constructor and create the object directly.

支持初始化的复制形式主要是为了与 C 的用法兼容。当情况许可时，可以允许编译器跳过复制构造函数直接创建对象，但编译器没有义务这样做。

Usually the difference between direct- or copy-initialization is at most a matter of low-level optimization. However, for types that do not support copying, or when using a constructor that is non`explicit` ([Section 12.4.4](#), p. 462) the distinction can be essential:

通常直接初始化和复制初始化仅在低级别上存在差异。然而，对于不支持复制的类型，或者使用非 `explicit` 构造函数 ([第 12.4.4 节](#)) 的进修，它们有本质区别：

```
ifstream file1("filename"); // ok: direct initialization
ifstream file2 = "filename"; // error: copy constructor is private
// This initialization is okay only if
// the Sales_item(const string&) constructor is not explicit
Sales_item item = string("9-999-99999-9");
```

The initialization of `file1` is fine. The `ifstream` class defines a constructor that can be called with a C-style string. That constructor is used to initialize `file1`.

`file1` 的初始化是正确的。`ifstream` 类定义了一个可用 C 风格字符串调用的构造函数，使用该构造函数初始化 `file1`。

The seemingly equivalent initialization of `file2` uses copy-initialization. That definition is not okay. We cannot copy objects of the IO types ([Section 8.1](#), p. 287), so we cannot use copy-initialization on objects of these types.

看上去等效的 `file2` 初始化使用复制初始化，但该定义不正确。由于不能复制 IO 类型的对象 ([第 8.1 节](#))，所以不能对那些类型的对象使用复制初始化。

Whether the initialization of `item` is okay depends on which version of our `Sales_item` class we are using. Some versions define the constructor that takes a `string` as `explicit`. If the constructor is `explicit`, then the initialization fails. If the constructor is not `explicit`, then the initialization is fine.

`item` 的初始化是否正确，取决于正在使用哪个版本的 `Sales_item` 类。某些版本将参数为一个 `string` 的构造函数定义为 `explicit`。如果构造函数是显式的，则初始化失败；如果构造函数不是显式的，则初始化成功。

Parameters and Return Values

形参与返回值

As we know, when a parameter is a nonreference type ([Section 7.2.1](#), p. 230), the argument is copied. Similarly, a nonreference return value ([Section 7.3.2](#), p. 247) is returned by copying the value in the `return` statement.

正如我们所知，当形参为非引用类型 ([第 7.2.1 节](#)) 的时候，将复制实参的值。类似地，以非引用类型作返回值时，将返回 `return` 语句 中的值的副本 ([第 7.3.2 节](#))。

When the parameter or return type is a class type, the copy is done by the copy constructor. For example, consider our `make_plural` function from page 248:

当形参或返回值为类类型时，由复制构造函数进行复制。例如，考虑 [第 7.3.2 节](#) 的 `make_plural` 函数：

```
// copy constructor used to copy the return value;
// parameters are references, so they aren't copied
string make_plural(size_t, const string&, const string&);
```

This function implicitly uses the `string` copy constructor to return the plural version of a given word. The parameters are `const` references; they are not copied.

这个函数隐式使用 `string` 复制构造函数返回给定单词的复数形式。形参是 `const` 引用，不能复制。

Initializing Container Elements

初始化容器元素

The copy constructor is used to initialize the elements in a sequential container. For example, we can initialize a container using a single parameter that represents a size ([Section 3.3.1](#), p. 92). This form of construction uses both the default constructor and the copy constructor for the element container:

复制构造函数可用于初始化顺序容器中的元素。例如，可以用表示容量的单个形参来初始化容器（[第 3.3.1 节](#)）。容器的这种构造方式使用默认构造函数和复制构造函数：

```
// default string constructor and five string copy constructors invoked
vector<string> svec(5);
```

The compiler initializes `svec` by first using the default `string` constructor to create a temporary value. The copy constructor is then used to copy the temporary into each element of `svec`.

编译器首先使用 `string` 默认构造函数创建一个临时值来初始化 `svec`，然后使用复制构造函数将临时值复制到 `svec` 的每个元素。



As a general rule ([Section 9.1.1](#), p. 307), unless you intend to use the default initial value of the container elements, it is more efficient to allocate an empty container and add elements as the values for those elements become known.

作为一般规则（[第 9.1.1 节](#)），除非你想使用容器元素的默认初始值，更有效的方法是，分配一个空容器并将已知元素的值加入容器。

Constructors and Array Elements

构造函数与数组元素

If we provide no element initializers for an array of class type, then the default constructor is used to initialize each element. However, if we provide explicit element initializers using the normal brace-enclosed array initialization list ([Section 4.1.1](#), p. 111), then each element is initialized using copy-initialization. An element of the appropriate type is created from the specified value, and then the copy constructor is used to copy that value to the corresponding element:

如果没有为类类型数组提供元素初始化式，则将用默认构造函数初始化每个元素。然而，如果使用常规的花括号括住的数组初始化列表（[第 4.1.1 节](#)）来提供显式元素初始化式，则使用复制初始化来初始化每个元素。根据指定值创建适当类型的元素，然后用复制构造函数将该值复制到相应元素：

```
Sales_item primer_eds[] = { string("0-201-16487-6"),
    string("0-201-54848-8"),
    string("0-201-82470-1"),
    Sales_item()
};
```

A value that can be used to invoke a single-argument constructor for the element type can be specified directly, as in the initializers for the first three elements. If we wish to specify no arguments or multiple arguments, we need to use the full constructor syntax, as we do in the initializer for the last element.

如前三个元素的初始化式中所示可以直接指定一个值，用于调用元素类型的单实参构造函数。如果希望不指定实参或指定多个实参，就需要使用完整的构造函数语法，正如最后一个元素的初始化那样。

Exercises Section 13.1

Exercise What is a copy constructor? When is it used?

13.1:

什么是复制构造函数？何时使用它？

Exercise The second initialization below fails to compile. What can we infer about the definition of `vector`?

下面第二个初始化不能编译。可以从 `vector` 的定义得出什么推断？

[\[View full width\]](#)

```

vector<int> v1(42); // ok: 42 elements, each 0
vector<int> v2 = 42; // error: what does this error tell us
→ about vector?

```

Exercise 13.3: Assuming `Point` is a class type with a `public` copy constructor, identify each use of the copy constructor in this program fragment:

假定 `Point` 为类类型，该类类型有一个复制构造函数，指出下面程序段中每一个使用了复制构造函数的地方：

```

Point global;
Point foo_bar(Point arg)
{
    Point local = arg;
    Point *heap = new Point(global);
    *heap = local;
    Point pa[ 4 ] = { local, *heap };
    return *heap;
}

```

13.1.1. The Synthesized Copy Constructor

13.1.1. 合成的复制构造函数

If we do not otherwise define the copy constructor, the compiler synthesizes one for us. Unlike the synthesized default constructor ([Section 12.4.3, p. 458](#)), a copy constructor is synthesized even if we define other constructors. The behavior of the **synthesized copy constructor** is to **memberwise initialize** the new object as a copy of the original object.

如果我们没有定义复制构造函数，编译器就会为我们合成一个。与合成的默认构造函数（[第 12.4.3 节](#)）不同，即使我们定义了其他构造函数，也会合成复制构造函数。**合成复制构造函数**的行为是，执行[逐个成员初始化](#)，将新对象初始化为原对象的副本。

By memberwise, we mean that taking each `nonstatic` member in turn, the compiler copies the member from the existing object into the one being created. With one exception, the type of each member determines what it means to copy it. The synthesized copy constructor directly copies the value of members of built-in type. Members of class type are copied by using the copy constructor for that class. The one exception concerns array members. Even though we ordinarily cannot copy an array, if a class has a member that is an array, then the synthesized copy constructor will copy the array. It does so by copying each element.

所谓“逐个成员”，指的是编译器将现在对象的每个非 `static` 成员，依次复制到正创建的对象。只有一个例外，每个成员类型决定了复制该成员的含义。合成复制构造函数直接复制内置类型成员的值，类类型成员使用该类的复制构造函数进行复制。数组成员的复制是个例外。虽然一般不能复制数组，但如果一个类具有数组成员，则合成复制构造函数将复制数组。复制数组时合成复制构造函数将复制数组的每一个元素。

The simplest conceptual model of memberwise initialization is to think of the synthesized copy constructor as one in which each data member is initialized in the constructor initializer list. For example, given our `Sales_item` class, which has three data members

逐个成员初始化最简单的概念模型是，将合成复制构造函数看作这样一个构造函数：其中每个数据成员在构造函数初始化列表中进行初始化。例如，对于我们的 `Sales_item` 类，它有三个数据成员：

```

class Sales_item {
// other members and constructors as before
private:
    std::string isbn;
    int units_sold;
    double revenue;
};

```

the synthesized `Sales_item` copy constructor would look something like:

合成复制构造函数如下所示：

```

Sales_item::Sales_item(const Sales_item &orig):
    isbn(orig.isbn), // uses string copy constructor
    units_sold(orig.units_sold), // copies orig.units_sold
    revenue(orig.revenue) // copy orig.revenue
{ } // empty body

```

13.1.2. Defining Our Own Copy Constructor

13.1.2. 定义自己的复制构造函数

The copy constructor is the constructor that takes a single parameter that is a (usually `const`) reference to the class type:

复制构造函数就是接受单个类类型引用形参（通常用 `const` 修饰）的构造函数：

```
class Foo {
public:
    Foo();           // default constructor
    Foo(const Foo&); // copy constructor
    // ...
};
```

Usually the parameter is a `const` reference, although we can also define the copy constructor to take a non`const` reference. Because the constructor is used (implicitly) to pass and return objects to and from functions, it usually should not be made `explicit` ([Section 12.4.4](#), p. 462). The copy constructor should copy the members from its argument into the object that is being constructed.

虽然也可以定义接受非 `const` 引用的复制构造函数，但形参通常是一个 `const` 引用。因为用于向函数传递对象和从函数返回对象，该构造函数一般不应设置为 `explicit`（[第 12.4.4 节](#)）。复制构造函数应将实参的成员复制到正在构造的对象。

For many classes, the synthesized copy constructor does exactly the work that is needed. Classes that contain only members that are of class type or members that are of built-in (but not pointer type) often can be copied without explicitly defining the copy constructor.

对许多类而言，合成复制构造函数只完成必要的工作。只包含类类型成员或内置类型（但不是指针类型）成员的类，无须显式地定义复制构造函数，也可以复制。

However, some classes *must* take control of what happens when objects are copied. Such classes often have a data member that is a pointer or that represents another resource that is allocated in the constructor. Other classes have bookkeeping that must be done whenever a new object is created. In both these cases, the copy constructor must be defined.

然而，有些类必须对复制对象时发生的事情加以控制。这样的类经常有一个数据成员是指针，或者有成员表示在构造函数中分配的其他资源。而另一些类在创建新对象时必须做一些特定工作。这两种情况下，都必须定义复制构造函数。

Often the hardest part about defining a copy constructor is recognizing that a copy constructor is needed. Defining the constructor is usually pretty easy once the need for the constructor is recognized. The copy constructor itself is defined like any other constructor: It has the same name as the name of the class, it has no return value, it may (should) use a constructor initializer to initialize the members of the newly created object, and it may do any other necessary work inside a function body.

通常，定义复制构造函数最困难的部分在于认识到需要复制构造函数。只要能认识到需要复制构造函数，定义构造函数一般非常简单。复制构造函数的定义与其他构造函数一样：它与类同名，没有返回值，可以（而且应该）使用构造函数初始化列表初始化新创建对象的成员，可以在函数体中做任何其他必要工作。

We'll look at examples of classes that require class-defined copy constructors in later sections. [Section 13.4](#) (p. 486) looks at a pair of classes that require an explicit copy constructor to handle bookkeeping associated with a simple message-handling application; classes with members that are pointers are covered in [Section 13.5](#) (p. 492).

后续章节中将给出一些需要定义复制构造函数的类的例子。[第 13.4 节](#)给出了一对类，它们需要显式复制构造函数，用于处理与简单消息处理应用程序相关的工作。具有指针成员的类在[第 13.5 节](#)给出。

Exercises Section 13.1.2

Exercise Given the following sketch of a class, write a copy constructor that copies all the elements.

13.4: Copy the object to which `pstring` points, not the pointer.

对于如下的类的简单定义，编写一个复制构造函数复制所有成员。复制 `pstring` 指向的对象而不是复制指针。

```
struct NoName {
    NoName(): pstring(new std::string), i(0), d(0) { }
private:
    std::string *pstring;
    int i;
    double d;
};
```

Exercise Which class definition is likely to need a copy constructor?

13.5: 哪个类定义可能需要一个复制构造函数？

Section 13.1. The Copy Constructor

- a. A `Point3w` class containing four float members
包含四个 `float` 成员的 `Point3w` 类。
- b. A `Matrix` class in which the actual matrix is allocated dynamically within the constructor and is deleted within its destructor
`Matrix` 类，其中，实际矩阵在构造函数中动态分配，在析构函数中删除。
- c. A `Payroll` class in which each object is provided with a unique ID
`Payroll` 类，在这个类中为每个对象提供唯一 ID。
- d. A `Word` class containing a `string` and a `vector` of line and column location pairs
`Word` 类，包含一个 `string` 和一个以行列位置对为元素的 `vector`。

Exercise 13.6: The parameter of the copy constructor does not strictly need to be `const`, but it does need to be a reference. Explain the rationale for this restriction. For example, explain why the following definition could not work.

复制构造函数的形参并不限制为 `const`，但必须是一个引用。解释这个限制的基本原理，例如，解释为什么下面的定义不能工作。

```
Sales_item::Sales_item(const Sales_item rhs);
```

13.1.3. Preventing Copies

13.1.3. 禁止复制

Some classes need to prevent copies from being made at all. For example, the `iostream` classes do not permit copying ([Section 8.1](#), p. 287). It might seem that if we want to forbid copies, we could omit the copy constructor. However, if we don't define a copy constructor, the compiler will synthesize one.

有些类需要完全禁止复制。例如，`iostream` 类就不允许复制（[第 8.1 节](#)）。如果想要禁止复制，似乎可以省略复制构造函数，然而，如果不定义复制构造函数，编译器将合成一个。



To prevent copies, a class must explicitly declare its copy constructor as `private`.

为了防止复制，类必须显式声明其复制构造函数为 `private`。

If the copy constructor is private, then user code will not be allowed to copy objects of the class type. The compiler will reject any attempt to make a copy.

如果复制构造函数是私有的，将不允许用户代码复制该类类型的对象，编译器将拒绝任何进行复制的尝试。

However, the friends and members of the class could still make copies. If we want to prevent copies even within the friends and members, we can do so by declaring a (`private`) copy constructor but not defining it.

然而，类的友元和成员仍可以进行复制。如果想要连友元和成员中的复制也禁止，就可以声明一个 (`private`) 复制构造函数但不对其定义。

It is legal to declare but not define a member function. However, any attempt to use an undefined member results in a link-time failure. By declaring (but not defining) a `private` copy constructor, we can forestall any attempt to copy an object of the class type: Attempts to make copies in user code will be flagged as an error at compile time, and attempts to make copies in member functions or friends will result in an error at link time.

声明而不定义成员函数是合法的，但是，使用未定义成员的任何尝试将导致链接失败。通过声明（但不定义）`private` 复制构造函数，可以禁止任何复制类类型对象的尝试：用户代码中复制尝试将在编译时标记为错误，而成员函数和友元中的复制尝试将在链接时导致错误。

Section 13.1. The Copy Constructor

Most Classes Should Define Copy and Default Constructors

大多数类应定义复制构造函数和默认构造函数

Classes that do not define the default constructor and/or the copy constructor impose serious limits on users of the class. Objects of classes that do not allow copies may be passed to (or returned from) a function only as a reference. They also may not be used as elements in a container.

不定义复制构造函数和／或默认构造函数，会严重局限类的使用。不允许复制的类对象只能作为引用传递给函数或从函数返回，它们也不能用作容器的元素。



It is usually best to define either implicitly or explicitly the default and copy constructors. The default constructor is synthesized only if there are no other constructors. If the copy constructor is defined, then the default constructor must be defined as well.

一般来说，最好显式或隐式定义默认构造函数和复制构造函数。只有不存在其他构造函数时才合成默认构造函数。如果定义了复制构造函数，也必须定义默认构造函数。

Team LiB

◀ PREVIOUS NEXT ▶

13.2. The Assignment Operator

13.2. 赋值操作符

Just as classes control how objects are initialized, they also define what happens when objects of their type are assigned:

与类要控制初始化对象的方式一样，类也定义了该类型对象赋值时会发生什么：

```
Sales_item trans, accum;
trans = accum;
```

As with the copy constructor, the compiler synthesizes an assignment operator if the class does not define its own.

与复制构造函数一样，如果类没有定义自己的赋值操作符，则编译器会合成一个。

Introducing Overloaded Assignment

介绍重载赋值

Before we look at the synthesized assignment operator, we need to know a bit about [overloaded operators](#), which we cover in detail in [Chapter 14](#).

在介绍合成赋值操作符之前，需要简单了解一下[重载操作符](#)，我们将在[第十四章](#)详细介绍。

Overloaded operators are functions that have the name `operator` followed by the symbol for the operator being defined. Hence, we define assignment by defining a function named `operator=`. Like any other function, an operator function has a return type and a parameter list. The parameter list must have the same number of parameters (including the implicit `this` parameter if the operator is a member) as the operator has operands. Assignment is binary, so the operator function has two parameters: The first parameter corresponds to the left-hand operand, and the second to the right-hand operand.

重载操作符是一些函数，其名字为 `operator` 后跟着所定义的操作符的符号。因此，通过定义名为 `operator=` 的函数，我们可以对赋值进行定义。像任何其他函数一样，操作符函数有一个返回值和一个形参表。形参表必须具有与该操作符数目相同的形参（如果操作符是一个类成员，则包括隐式 `this` 形参）。赋值是二元运算，所以该操作符函数有两个形参：第一个形参对应着左操作数，第二个形参对应右操作数。

Most operators may be defined as member or nonmember functions. When an operator is a member function, its first operand is implicitly bound to the `this` pointer. Some operators, assignment among them, must be members of the class for which the operator is defined. Because assignment must be a member of its class, `this` is bound to a pointer to the left-hand operand. The assignment operator, therefore, takes a single parameter that is an object of the same class type. Usually, the right-hand operand is passed as a `const` reference.

大多数操作符可以定义为成员函数或非成员函数。当操作符为成员函数时，它的第一个操作数隐式绑定到 `this` 指针。有些操作符（包括赋值操作符）必须是定义自己的类的成员。因为赋值必须是类的成员，所以 `this` 绑定到指向左操作数的指针。因此，赋值操作符接受单个形参，且该形参是同一类型的对象。右操作数一般作为 `const` 引用传递。

The return type from the assignment operator should be the same as the return from assignment for the built-in types ([Section 5.4.1](#), p. 160). Assignment to a built-in type returns a reference to its left-hand operand. Therefore, the assignment operator also returns a reference to the same type as its class.

赋值操作符的返回类型应该与内置类型赋值运算返回的类型相同（[第 5.4.1 节](#)）。内置类型的赋值运算返回对右操作数的引用，因此，赋值操作符也返回对同一类型的引用。

For example, the assignment operator for `Sales_item` might be declared as

例如，`Sales_item` 的赋值操作符可以声明为：

```
class Sales_item {
public:
    // other members as before
    // equivalent to the synthesized assignment operator
    Sales_item& operator=(const Sales_item &);
};
```

The Synthesized Assignment Operator

合成赋值操作符

The **synthesized assignment operator** operates similarly to the synthesized copy constructor. It performs **memberwise assignment**: Each member of the right-hand object is assigned to the corresponding member of the left-hand object. Except for arrays, each member is assigned in the usual way for its type. For arrays, each array element is assigned.

合成赋值操作符与合成复制构造函数的操作类似。它会执行逐个成员赋值：右操作数对象的每个成员赋值给左操作数对象的对应成员。除数组之外，每个成员用所属类型的常规方式进行赋值。对于数组，给每个数组元素赋值。

As an example, the synthesized `Sales_item` assignment operator would look something like:

例如，`Sales_item` 的合成赋值操作符可能如下所示：

```
// equivalent to the synthesized assignment operator
Sales_item&
Sales_item::operator=(const Sales_item &rhs)
{
    isbn = rhs.isbn;           // calls string::operator=
    units_sold = rhs.units_sold; // uses built-in int assignment
    revenue = rhs.revenue;     // uses built-in double assignment
    return *this;
}
```

The synthesized assignment operator assigns each member in turn, using the built-in or class-defined assignment operator as appropriate to the type of the member. The operator returns `*this`, which is a reference to the left-hand object.

合成赋值操作符根据成员类型使用适合的内置或类定义的赋值操作符，依次给每个成员赋值，该操作符返回 `*this`，它是对左操作数对象的引用。

Copy and Assign Usually Go Together

复制和赋值常一起使用

Classes that can use the synthesized copy constructor usually can use the synthesized assignment operator as well. Our `Sales_item` class has no need to define either the copy constructor or the assignment operator: The synthesized versions of these operators work fine.

可以使用合成复制构造函数的类通常也可以使用合成赋值操作符。我们的 `Sales_item` 类无须定义复制构造函数或赋值操作符，这些操作符的合成版本工作得很好。

However, a class may define its own assignment operator. In general, if a class needs a copy constructor, it will also need an assignment operator.

然而，类也可以定义自己的赋值操作符。一般而言，如果类需要复制构造函数，它也会需要赋值操作符。



In fact, these operations should be thought of as a unit. If we require one, we almost surely require the other.

实际上，就将这两个操作符看作一个单元。如果需要其中一个，我们几乎也肯定需要另一个。

We'll see examples of classes that need to define their own assignment operators in [Section 13.4](#) (p. 486) and [Section 13.5](#) (p. 492).

我们将在[第 13.4 节](#)和[第 13.5 节](#)介绍类需要自定义赋值操作符的例子。

Exercises Section 13.2

Exercise When does a class need to define an assignment operator?

13.7:

类何时需要定义赋值操作符？

Exercise For each type listed in the first exercise in [Section 13.1.2](#) (p. 481) indicate whether the class would need an assignment operator.

Section 13.2. The Assignment Operator

对于第 13.1.2 节习题 13.5 中列出的每个类型，指出类是否需要赋值操作符。

Exercise 13.9: The first exercise in [Section 13.1.2](#) (p. 481) included a skeletal definition for class `NoName`. Determine whether that class needs an assignment operator. If so, implement it.

[第 13.1.2 节](#)习题 13.5 中包括 `NoName` 类的简单定义。确定这个类是否需要赋值操作符。如果需要，实现它。

Exercise 13.10: Define an `Employee` class that contains the employee's name and a unique employee identifier. Give the class a default constructor and a constructor that takes a `string` representing the employee's name. If the class needs a copy constructor or assignment operator, implement those functions as well.

定义一个 `Employee` 类，包含雇员名字和一个唯一的雇员标识。为该类定义默认构造函数和参数为表示雇员名字的 `string` 的构造函数。如果该类需要复制构造函数或赋值操作符，实现这些函数。

Team LiB

◀ PREVIOUS NEXT ▶

13.3. The Destructor

13.3. 析构函数

One purpose of a constructor is to provide for the automatic acquisition of a resource. For example, a constructor might allocate a buffer or open a file. Having allocated the resource in the constructor, we need a corresponding operation that automatically deallocates or otherwise releases the resource. The destructor is a special member function that can be used to do whatever resource deallocation is needed. It serves as the complement to the constructors of the class.

构造函数的一个用途是自动获取资源。例如，构造函数可以分配一个缓冲区或打开一个文件，在构造函数中分配了资源之后，需要一个对应操作自动回收或释放资源。析构函数就是这样的一个特殊函数，它可以完成所需的资源回收，作为类构造函数的补充。

When a Destructor Is Called

何时调用析构函数

The destructor is called automatically whenever an object of its class is destroyed:

撤销类对象时会自动调用析构函数：

```
// p points to default constructed object
Sales_item *p = new Sales_item;
{
    // new scope
    Sales_item item(*p); // copy constructor copies *p into item
    delete p;            // destructor called on object pointed to by p
}                         // exit local scope; destructor called on item
```

Variables such as `item` are destroyed automatically when they go out of scope. Hence, the destructor on `item` is run when the close curly is encountered.

变量（如 `item`）在超出作用域时应该自动撤销。因此，当遇到右花括号时，将运行 `item` 的析构函数。

An object that is dynamically allocated is destroyed only when a pointer pointing to the object is `delete` d. If we do not `delete` a pointer to a dynamically allocated object, then the destructor is never run on that object. The object will persist forever, leading to a memory leak. Moreover, any resources used inside the object will also not be released.

动态分配的对象只有在指向该对象的指针被删除时才撤销。如果没有删除指向动态对象的指针，则不会运行该对象的析构函数，对象就一直存在，从而导致内存泄漏，而且，对象内部使用的任何资源也不会释放。



The destructor is *not* run when a reference or a pointer to an object goes out of scope. The destructor is run only when a pointer to a dynamically allocated object is `deleted` or when an actual object (not a reference to the object) goes out of scope.

当对象的引用或指针超出作用域时，不会运行析构函数。只有删除指向动态分配对象的指针或实际对象（而不是对象的引用）超出作用域时，才会运行析构函数。

Destructors are also run on the elements of class type in a container whether a library container or built-in array when the container is destroyed:

撤销一个容器（不管是标准库容器还是内置数组）时，也会运行容器中的类类型元素的析构函数：

```
{
    Sales_item *p = new Sales_item[10]; // dynamically allocated
    vector<Sales_item> vec(p, p + 10); // local object
    // ...
    delete [] p; // array is freed; destructor run on each element
} // vec goes out of scope; destructor run on each element
```

Section 13.3. The Destructor

The elements in the container are always destroyed in reverse order: The element indexed by `size() - 1` is destroyed first, followed by the one indexed by `size() - 2` and so on until element `[0]`, which is destroyed last.

容器中的元素总是按逆序撤销：首先撤销下标为 `size() - 1` 的元素，然后是下标为 `size() - 2` 的元素……直到最后撤销下标为 `[0]` 的元素。

When to Write an Explicit Destructor

何时编写显式析构函数

Many classes do not require an explicit destructor. In particular, a class that has a constructor does not necessarily need to define its own destructor. Destructors are needed only if there is work for them to do. Ordinarily they are used to relinquish resources acquired in the constructor or during the lifetime of the object.

许多类不需要显式析构函数，尤其是具有构造函数的类不一定需要定义自己的析构函数。仅在有些工作需要析构函数完成时，才需要析构函数。析构函数通常用于释放放在构造函数或在对象生命期内获取的资源。



A useful rule of thumb is that if a class needs a destructor, it will also need the assignment operator and a copy constructor. This rule is often referred to as the [Rule of Three](#), indicating that if you need a destructor, then you need all three copy-control members.

如果类需要析构函数，则它也需要赋值操作符和复制构造函数，这是一个有用的经验法则。这个规则常称为[三法则](#)，指的是如果需要析构函数，则需要所有这三个复制控制成员。

A destructor is not limited only to relinquishing resources. A destructor, in general, can perform any operation that the class designer wishes to have executed subsequent to the last use of an object of that class.

析构函数并不仅限于用来释放资源。一般而言，析构函数可以执行任意操作，该操作是类设计者希望在该类对象的使用完毕之后执行的。

The Synthesized Destructor

合成析构函数

Unlike the copy constructor or assignment operator, the compiler always synthesizes a destructor for us. The synthesized destructor destroys each `nonstatic` member in the reverse order from that in which the object was created. In consequence, it destroys the members in reverse order from which they are declared in the class. For each member that is of class type, the synthesized destructor invokes that member's destructor to destroy the object.

与复制构造函数或赋值操作符不同，编译器总是会为我们合成一个析构函数。合成析构函数按对象创建时的逆序撤销每个非 `static` 成员，因此，它按成员在类中声明次序的逆序撤销成员。对于类类型的每个成员，合成析构函数调用该成员的析构函数来撤销对象。



Destroying a member of built-in or compound type has no effect. In particular, the synthesized destructor does not `delete` the object pointed to by a pointer member.

撤销内置类型成员或复合类型的成员没什么影响。尤其是，合成析构函数并不删除指针成员所指向的对象。

How to Write a Destructor

如何编写析构函数

Our `Sales_item` class is an example of a class that allocates no resources and so does not need its own destructor. Classes that do allocate resources usually need to define a destructor to free those resources. The destructor is a member function with the name of the class prefixed by a tilde (~). It has no return value and takes no parameters. Because it cannot specify any parameters, it cannot be overloaded. Although we can define multiple class constructors, we can provide only a single destructor to be applied to all objects of our class.

Section 13.3. The Destructor

`Sales_item` 类是类没有分配资源因此不需要自己的析构函数的一个例子。分配了资源的类一般需要定义析构函数以释放那些资源。析构函数是个成员函数，它的名字是在类名之前加上一个代字号 (~)，它没有返回值，没有形参。因为不能指定任何形参，所以不能重载析构函数。虽然可以为一个类定义多个构造函数，但只能提供一个析构函数，应用于类的所有对象。

An important difference between the destructor and the copy constructor or assignment operator is that even if we write our own destructor, the synthesized destructor is still run. For example, we might write the following empty destructor for class `Sales_item`:

析构函数与复制构造函数或赋值操作符之间的一个重要区别是，即使我们编写了自己的析构函数，合成析构函数仍然运行。例如，可以为 `Sales_item`: 类编写如下的空析构函数：

```
class Sales_item {
public:
    // empty; no work to do other than destroying the members,
    // which happens automatically
    ~Sales_item() { }
    // other members as before
};
```

When objects of type `Sales_item` are destroyed, this destructor, which does nothing, would be run. After it completes, the synthesized destructor would also be run to destroy the members of the class. The synthesized destructor destroys the `string` member by calling the `string` destructor, which frees the memory used to hold the `isbn`. The `units_sold` and `revenue` members are of built-in type, so the synthesized destructor does nothing to destroy them.

撤销 `Sales_item` 类型的对象时，将运行这个什么也不做的析构函数，它执行完毕后，将运行合成析构函数以撤销类的成员。合成析构函数调用 `string` 析构函数来撤销 `string` 成员，`string` 析构函数释放了保存 `isbn` 的内存。`units_sold` 和 `revenue` 成员是内置类型，所以合成析构函数撤销它们不需要做什么。

Exercises Section 13.3

Exercise 13.11: What is a destructor? What does the synthesized destructor do? When is a destructor synthesized? When must a class define its own destructor?

什么是析构函数？合成析构函数有什么用？什么时候会合成析构函数？什么时候一个类必须定义自己的析构函数？

Exercise 13.12: Determine whether the `NoName` class sketched in the exercises on page 481, is likely to need a destructor. If so, implement it.

确定在第 13.1.2 节习题 13.4 中概略定义的 `NoName` 类是否需要析构函数，如果需要，实现它。

Exercise 13.13: Determine whether the `Employee` class, defined in the exercises on page 484, needs a destructor. If so, implement it.

确定在第 13.2 节习题 13.10 中定义的 `Employee` 类是否需要析构函数，如果需要，实现它。

Exercise 13.14: A good way to understand copy-control members and constructors is to define a simple class with these members in which each member prints its name:

理解复制控制成员和构造函数的一个良好方式是定义一个简单类，该类具有这些成员，每个成员打印自己的名字：

```
struct Exmpl {
    Exmpl() { std::cout << "Exmpl()" << std::endl; }
    Exmpl(const Exmpl&)
        { std::cout << "Exmpl(const Exmpl&)" << std::endl; }
    // ...
};
```

Write a class like `Exmpl`, giving it the copy-control members and other constructors. Now write a program using objects of type `Exmpl` in various ways: pass them as non-reference and reference parameters; dynamically allocate them; put them in containers, and so forth. Studying which constructors and copy-control members are executed and when can be helpful in cementing your understanding of these concepts.

编写一个像 `Exmpl` 这样的类，给出复制控制成员和其他构造函数。然后写一个程序，用不同方式使用 `Exmpl` 类型的对象：作为非引用形参和引用形参传递，动态分配，放在容器中，等等。研究何时执行哪个构造函数和复制构造函数，可以帮助你融会贯通地理解这些概念。

Exercise 13.15: How many destructor calls occur in the following code fragment?

下面的代码段中发生了多少次析构函数的调用？

```
void fcn(const Sales_item *trans, Sales_item accum)
{
    Sales_item item1(*trans), item2(accum);
```

Section 13.3. The Destructor

```
if (!item1.same_isbn(item2)) return;
if (item1.avg_price() <= 99) return;
else if (item2.avg_price() <= 99) return;
// ...  
}
```

Team LiB

◀ PREVIOUS NEXT ▶

13.4. A Message-Handling Example

13.4. 消息处理示例

As an example of a class that needs to control copies in order to do some bookkeeping, we'll sketch out two classes that might be used in a mail-handling application. These classes, `Message` and `Folder`, represent, respectively, email (or other) messages and directories in which a message might appear. A given `Message` might appear in more than one `Folder`. We'll have `save` and `remove` operations on `Message` that save or remove that message in the specified `Folder`.

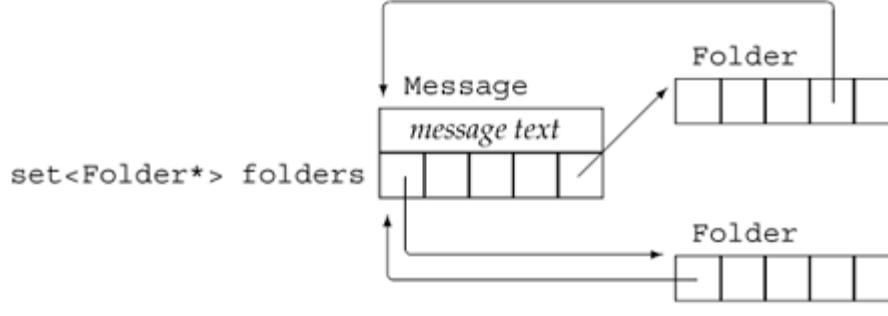
有些类为了做一些工作需要对复制进行控制。为了给出这样的例子，我们将概略定义两个类，这两个类可用于邮件处理应用程序。`Message` 类和 `Folder` 类分别表示电子邮件（或其他）消息和消息所出现的目录，一个给定消息可以出现在多个目录中。`Message` 上有 `save` 和 `remove` 操作，用于在指定 `Folder` 中保存或删除该消息。

Rather than putting a copy of each `Message` into each `Folder`, we'll have each `Message` hold a `set` of pointers to the `Folders` in which this `Message` appears. Each `Folder` will also store pointers to the `Message`s it contains. [Figure 13.1](#) (p. 488) illustrates the data structure we'll implement.

对每个 `Message`，我们并不是在每个 `Folder` 中都存放一个副本，而是使每个 `Message` 保存一个指针集（`set`），`set` 中的指针指向该 `Message` 所在的 `Folder`。每个 `Folder` 也保存着一些指针，指向它所包含的 `Message`。将要实现的数据结构如[图 13.1](#) 所示。

Figure 13.1. `Message` and `Folder` Class Design

图 13.1. `Message` 和 `Folder` 类设计



When we create a new `Message`, we will specify the contents of the message but no `Folder`. Calling `save` will put a `Message` in a `Folder`.

创建新的 `Message` 时，将指定消息的内容但不指定 `Folder`。调用 `save` 将 `Message` 放入一个 `Folder`。

When we copy a `Message`, we'll copy both the contents of the original message and the set of `Folder` pointers. We must also add a pointer to this `Message` to each of the `Folder`s that points to the original `Message`.

复制一个 `Message` 对象时，将复制原始消息的内容和 `Folder` 指针集，还必须给指向源 `Message` 的每个 `Folder` 增加一个指向该 `Message` 的指针。

Assigning one `Message` to another behaves similarly to copying a `Message`: After the assignment, the contents and set of `Folders` will be the same. We'll start by removing the existing left-hand `message` from the `Folders` it was in prior to the assignment. Once the old `Message` is gone, we'll copy the contents and set of `Folders` from the right-hand operand into the left. We'll also have to add a pointer to the left-hand `Message` to each `Folder` in this set.

将一个 `Message` 对象赋值给另一个，类似于复制一个 `Message`：赋值之后，内容和 `Folder` 集将是相同的。首先从左边 `Message` 在赋值之前所处的 `Folder` 中删除该 `Message`。原来的 `Message` 去掉之后，再将右边操作数的内容和 `Folders` 集复制到左边，还必须在这个 `Folder` 集中的每个 `Folders` 中增加一个指向左边 `Message` 的指针。

When we destroy a `Message`, we must update each `Folder` that points to the `Message`. Once the `Message` goes away, those pointers will be no good, so we must remove the pointer to this `Message` from each `Folder` in the `Message`'s own `set` of `Folder` pointers.

撤销一个 `Message` 对象时，必须更新指向该 `Message` 的每个 `Folder`。一旦去掉了 `Message`，指向该 `Message` 的指针将失效，所以必须从该 `Message` 的 `Folder` 指针集的每个 `Folder` 中删除这个指针。

Looking at this list of operations, we can see that the destructor and the assignment operator share the work of removing messages from the list of `Folders` that had held a given `Message`. Similarly, the copy constructor and the assignment operator share the work of adding a `Message` to a given list of `Folders`. We'll define a pair of `private` utility functions to do these tasks.

查看这个操作列表，可以看到，析构函数和赋值操作符分担了从保存给定 `Message` 的 `Folder` 列表中删除消息的工作。类似地，复制构造函数和赋值操作符分担将一个

Section 13.4. A Message-Handling Example

`Message` 加到给定 `Folder` 列表的工作。我们将定义一对 `private` 实用函数完成这些任务。

The `Message` Class

`Message` 类

Given this design, we can write a fair bit of our `Message` class:

对于以上的设计，可以如下编写 `Message` 类的部分代码：

```
class Message {
public:
    // folders is initialized to the empty set automatically
    Message(const std::string &str = "") :
        contents(str) {}

    // copy control: we must manage pointers to this Message
    // from the Folders pointed to by folders
    Message(const Message&);
    Message& operator=(const Message&);
    ~Message();

    // add/remove this Message from specified Folder's set of messages
    void save(Folder&);
    void remove(Folder&);

private:
    std::string contents;      // actual message text
    std::set<Folder*> folders; // Folders that have this Message
    // Utility functions used by copy constructor, assignment, and destructor:
    // Add this Message to the Folders that point to the parameter
    void put_Msg_in_Folders(const std::set<Folder*>&);

    // remove this Message from every Folder in folders
    void remove_Msg_from_Folders();
};
```

The class defines two data members: `contents`, which is a `string` that holds the actual message, and `folders`, which is a `set` of pointers to the `Folders` in which this `Message` appears.

`Message` 类定义了两个数据成员：`contents` 是一个保存实际消息的 `string`，`folders` 是一个 `set`，包含指向该 `Message` 所在的 `Folder` 的指针。

The constructor takes a single `string` parameter representing the contents of the message. The constructor stores a copy of the message in `contents` and (implicitly) initializes the `set` of `Folders` to the empty set. This constructor provides a default argument, which is the empty string, so it also serves as the `Message` default constructor.

构造函数接受单个 `string` 形参，表示消息的内容。构造函数将消息的副本保存在 `contents` 中，并（隐式）将 `Folder` 的 `set` 初始化为空集。这个构造函数提供一个默认实参（空串），所以它也可以作为默认构造函数。

The utility functions provide the actions shared among the copy-control members. The `put_Msg_in_Folders` function adds a copy of its own `Message` to the `Folders` that point to the given `Message`. After this function completes, each `Folder` that points to the parameter will also point to this `Message`. This function will be used by both the copy constructor and the assignment operator.

实用函数提供由复制控制成员共享的行为。`put_Msg_in_Folders` 函数将自身 `Message` 的一个副本添加到指向给定 `Message` 的各 `Folder` 中，这个函数执行完后，形参指向的每个 `Folder` 也将指向这个 `Message`。复制构造函数和赋值操作符都将使用这个函数。

The `remove_Msg_from_Folders` function is used by the assignment operator and destructor. It removes the pointer to this `Message` from each of the `Folders` in the `folders` member.

`remove_Msg_from_Folders` 函数用于赋值操作符和析构函数，它从 `folders` 成员的每个 `Folder` 中删除指向这个 `Message` 的指针。

Copy Control for the `Message` Class

`Message` 类的复制控制

When we copy a `Message`, we have to add the newly created `Message` to each `Folder` that holds the `Message` from which we're copying. This work is beyond what the synthesized constructor would do for us, so we must define our own copy constructor:

复制 `Message` 时，必须将新创建的 `Message` 添加到保存原 `Message` 的每个 `Folder` 中。这个工作超出了合成构造函数的能力范围，所以我们必须定义自己的复制构造函数：

```
Message::Message(const Message &m):
    contents(m.contents), folders(m.folders)
{
    // add this Message to each Folder that points to m
    put_Msg_in_Folders(folders);
}
```

The copy constructor initializes the data members of the new object as copies of the members from the old. In addition to these initializations which the synthesized copy constructor would have done for us we must also iterate through `folders`, adding this `Message` to each `Folder` in that set. The copy constructor uses the `put_Msg_in_Folder` function to do this work.

复制构造函数将用旧对象成员的副本初始化新对象的数据成员。除了这些初始化之外（合成复制构造函数可以完成这些初始化），还必须用 `folders` 进行迭代，将这个新的 `Message` 加到那个集的每个 `Folder` 中。复制构造函数使用 `put_Msg_in_Folder` 函数完成这个工作。



When we write our own copy constructor, we must explicitly copy any members that we want copied. An explicitly defined copy constructor copies nothing automatically.

编写自己的复制构造函数时，必须显式复制需要复制的任意成员。显式定义的复制构造函数不会进行任何自动复制。

As with any other constructor, if we do not initialize a class member, then that member is initialized using the member's default constructor. Default initialization in a copy constructor does *not* use the member's copy constructor.

像其他任何构造函数一样，如果没有初始化某个类成员，则那个成员用该成员的默认构造函数初始化。复制构造函数中的默认初始化不会使用成员的复制构造函数。

The `put_Msg_in_Folders` Member

`put_Msg_in_Folders` 成员

`put_Msg_in_Folders` iterates through the pointers in the `folders` member of the parameter `rhs`. These pointers denote each `Folder` that points to `rhs`. We need to add a pointer to this `Message` to each of those `Folders`.

`put_Msg_in_Folders` 通过形参 `rhs` 的成员 `folders` 中的指针进行迭代。这些指针表示指向 `rhs` 的每个 `Folder`，需要将指向这个 `Message` 的指针加到每个 `Folder`。

The function does this work by looping through `rhs.folders`, calling the `Folder` member named `addMsg`. That function will do whatever it takes to add a pointer to this `Message` to that `Folder`:

函数通过 `rhs.folders` 进行循环，调用命名为 `addMsg` 的 `Folder` 成员来完成这个工作，`addMsg` 函数将指向该 `Message` 的指针加到 `Folder` 中。

```
// add this Message to Folders that point to rhs
void Message::put_Msg_in_Folders(const set<Folder*> &rhs)
{
    for(std::set<Folder*>::const_iterator beg = rhs.begin();
        beg != rhs.end(); ++beg)
        (*beg)->addMsg(this); // *beg points to a Folder
}
```

The only tricky part in this function is the call to `addMsg`:

这个函数中唯一复杂的部分是对 `addMsg` 的调用：

```
(*beg)->addMsg(this); // *beg points to a Folder
```

That call starts with `(*beg)`, which dereferences the iterator. Dereferencing the iterator yields a pointer to a `Folder`. The expression then applies the arrow operator to the `Folder` pointer in order to run the `addMsg` operation. We pass `this`, which points to the `Message` we want to add to the `Folder`.

那个调用以 `(*beg)` 开头，它解除迭代器引用。解除迭代器引用将获得一个指向 `Folder` 的指针。然后表达式对 `Folder` 指针应用箭头操作符以执行 `addMsg` 操作，将 `this` 传给 `addMsg`，该指针指向我们想要添加到 `Folder` 中的 `Message`。

Message Assignment Operator

Message 赋值操作符

Assignment is more complicated than the copy constructor. Like the copy constructor, assignment must assign the `contents` and update `folders` to match that of the right-hand operand. It must also add this `Message` to each of the `Folders` that points to the `rhs`. We can use our `put_Msg_in_Folders` function to do this part of the assignment.

Section 13.4. A Message-Handling Example

赋值比复制构造函数更复杂。像复制构造函数一样，赋值必须对 `contents` 赋值并更新 `folders` 使之与右操作数的 `folders` 相匹配。它还必须将该 `Message` 加到指向 `rhs` 的每个 `Folder` 中，可以使用 `put_Msg_in_Folders` 函数完成赋值的这一部分工作。

Before copying from the `rhs`, we must first remove this `Message` from each of the `Folders` that currently point to it. We'll need to iterate through `folders`, removing the pointer to this `Message` from each `Folder` in `folders`. The function named `remove_Msg_from_Folders` will do this work.

在从 `rhs` 复制之前，必须首先从当前指向该 `Message` 的每个 `Folder` 中删除它。我们需要通过 `folders` 进行迭代，从 `folders` 的每个 `Folder` 中删除指向该 `Message` 的指针。命名为 `remove_Msg_from_Folders` 的函数将完成这项工作。

Given `remove_Msg_from_Folders` and `put_Msg_in_Folders`, which do the real work, the assignment operator itself is fairly simple:

对于完成实际工作的 `remove_Msg_from_Folders` 和 `put_Msg_in_Folders`，赋值操作符本身相当简单：

```
Message& Message::operator=(const Message &rhs)
{
    if (&rhs != this) {
        remove_Msg_from_Folders(); // update existing Folders
        contents = rhs.contents; // copy contents from rhs
        folders = rhs.folders; // copy Folder pointers from rhs
        // add this Message to each Folder in rhs
        put_Msg_in_Folders(rhs.folders);
    }
    return *this;
}
```

The assignment operator starts by checking that the left- and right-hand operands are not the same. We do this check for reasons that will become apparent as we walk through the rest of the function. Assuming that the operands are different objects, we call `remove_Msg_from_Folders` to remove this `Message` from each of the `Folders` in the `folders` member. Once that work is done, we have to assign the `contents` and `folders` members from the right-hand operand to this object. Finally, we call `put_Msg_in_Folders` to add a pointer to this `Message` in each of the `Folders` that also point to `rhs`.

赋值操作符首先检查左右操作数是否相同。查看函数的后续部分可以清楚地看到进行这一检查的原因。假定操作数是不同对象，调用 `remove_Msg_from_Folders` 从 `folders` 成员的每个 `Folder` 中删除该 `Message`。一旦这项工作完成，必须将右操作数的 `contents` 和 `folders` 成员赋值给这个对象。最后，调用 `put_Msg_in_Folders` 将指向这个 `Message` 的指针添加至指向 `rhs` 的每个 `Folder` 中。

Now that we've seen work that `remove_Msg_from_Folders` does, we can see why we start the assignment operator by checking that the objects are different. Assignment involves obliterating the left-hand operand. Once the members of the left-hand operand are destroyed, those in the right-hand operand are assigned to the corresponding left-hand members. If the objects were the same, then destroying the left-hand members would also destroy the right-hand members!

了解了 `remove_Msg_from_Folders` 的工作之后，我们来看看为什么赋值操作符首先要检查对象是否不同。赋值时需删除左操作数，并在撤销左操作数的成员之后，将右操作数的成员赋值给左操作数的相应成员。如果对象是相同的，则撤销左操作数的成员也将撤销右操作数的成员！



It is crucially important for assignment operators to work correctly, even when an object is assigned to itself. A common way to ensure this behavior is by checking explicitly for self-assignment.

即使对象赋值给自己，赋值操作符的正确工作也非常重要。保证这个行为的通用方法是显式检查对自身的赋值。

The `remove_Msg_from_Folders` Member

`remove_Msg_from_Folders` 成员

The implementation of the `remove_Msg_from_Folders` function is similar to that of `put_Msg_in_Folders`, except that this time we'll call `remMsg` to remove this `Message` from each `Folder` pointed to by `folders`:

除了调用 `remMsg` 从 `folders` 指向的每个 `Folder` 中删除这个 `Message` 之外，`remove_Msg_from_Folders` 函数的实现与 `put_Msg_in_Folders` 类似：

```
// remove this Message from corresponding Folders
void Message::remove_Msg_from_Folders()
{
    // remove this message from corresponding folders
    for(std::set<Folder*>::const_iterator beg =
        folders.begin (); beg != folders.end (); ++beg)
        (*beg)->remMsg(this); // *beg points to a Folder
}
```

The `Message` Destructor

`Message` 析构函数

The remaining copy-control function that we must implement is the destructor:

剩下必须实现的复制控制函数是析构函数：

```
Message::~Message()
{
    remove_Msg_from_Folders();
}
```

Given the `remove_Msg_from_Folders` function, writing the destructor is trivial. We call that function to clean up `folders`. The system automatically invokes the `string` destructor to free `contents` and the `set` destructor to clean up the memory used to hold the `folders` member. Thus, the only work for the `Message` destructor is to call `remove_Msg_from_Folders`.

有了 `remove_Msg_from_Folders` 函数，编写析构函数将非常简单。我们调用 `remove_Msg_from_Folders` 函数清除 `folders`，系统自动调用 `string` 析构函数释放 `contents`，自动调用 `set` 析构函数清除用于保存 `folders` 成员的内存，因此，`Message` 析构函数唯一要做的是调用 `remove_Msg_from_Folders`。



The assignment operator often does the same work as is needed in the copy constructor and destructor. In such cases, the common work should be put in `private` utility functions.

赋值操作符通常要做复制构造函数和析构函数也要完成的工作。在这种情况下，通用工作应在 `private` 实用函数中。

Exercises Section 13.4

Exercise

13.16: Write the `Message` class as described in this section.

编写本节中描述的 `Message` 类。

Exercise

13.17: Add functions to the `Message` class that are analogous to the `Folder` operations `addMsg` and `remMsg`. These functions, which could be named `addFldr` and `remFldr`, should take a pointer to a `Folder` and insert that pointer into `folders`. These functions can be `private` because they will be used only in the implementation of the `Folder` class.

为 `Message` 类增加与 `Folder` 的 `addMsg` 和 `remMsg` 操作类似的函数。这些函数可以命名为 `addFldr` 和 `remFldr`，应接受一个指向 `Folder` 的指针并将该指针插入到 `folders`。这些函数可为 `private` 的，因为它们将仅在 `Message` 类的实现中使用。

Exercise

13.18: Write the corresponding `Folder` class. That class should hold a `set<Message*>` that contains elements that point to `Message`s.

编写相应的 `Folder` 类。该类应保存一个 `set<Message*>`，包含指向 `Message` 的元素。

Exercise

13.19: Add a `save` and `remove` operation to the `Message` and `Folder` classes. These operations should take a `Folder` and add (or remove) that `Folder` to (from) the `set` of `Folders` that point to this `Message`. The operation must also update the `Folder` to know that it points to this `Message`, which can be done by calling `addMsg` or `remMsg`.

在 `Message` 类和 `Folder` 类中增加 `save` 和 `remove` 操作。这些操作应接受一个 `Folder`，并将该 `Folder` 加入到指向这个 `Message` 的 `Folder` 集中（或从其中删除 `Folder`）。操作还必须更新 `Folder` 以反映它指向该 `Message`，这可以通过调用 `addMsg` 或 `remMsg`。

13.5. Managing Pointer Members

13.5. 管理指针成员

This book generally advocates the use of the standard library. One reason we do so is that using the standard library greatly reduces the need for pointers in modern C++ programs. However, many applications still require the use of pointers, particularly in the implementation of classes. Classes that contain pointers require careful attention to copy control. The reason they must do so is that copying a pointer copies only the address in the pointer. Copying a pointer does not copy the object to which the pointer points.

本书始终提倡使用标准库。这样做的一个原因是，使用标准库能够大大减少现代 C++ 程序中对指针的需要。然而，许多应用程序仍需要使用指针，特别是在类的实现中。包含指针的类需要特别注意复制控制，原因是复制指针时只复制指针中的地址，而不会复制指针指向的对象。

When designing a class with a pointer member, the first decision a class author must make is what behavior that pointer should provide. When we copy one pointer to another, the two pointers point to the same object. When two pointers point to the same object, it is possible to use either pointer to change the underlying object. Similarly, it is possible for one pointer to `delete` the object even though the user of the other pointer still thinks the underlying object exists.

设计具有指针成员的类时，类设计者必须首先需要决定的是该指针应提供什么行为。将一个指针复制到另一个指针时，两个指针指向同一对象。当两个指针指向同一对象时，可能使用任一指针改变基础对象。类似地，很可能一个指针删除了一对象时，另一指针的用户还认为基础对象仍然存在。

By default, a pointer member has the same behavior as a pointer object. However, through different copy-control strategies we can implement different behavior for pointer members. Most C++ classes take one of three approaches to managing pointer members:

指针成员默认具有与指针对象同样的行为。然而，通过不同的复制控制策略，可以为指针成员实现不同的行为。大多数 C++ 类采用以下三种方法之一管理指针成员：

1. The pointer member can be given normal pointerlike behavior. Such classes will have all the pitfalls of pointers but will require no special copy control.

指针成员采取常规指针型行为。这样的类具有指针的所有缺陷但无需特殊的复制控制。

2. The class can implement so-called "smart pointer" behavior. The object to which the pointer points is shared, but the class prevents dangling pointers.

类可以实现所谓的“智能指针”行为。指针所指向的对象是共享的，但类能够防止悬垂指针。

3. The class can be given valuelike behavior. The object to which the pointer points will be unique to and managed separately by each class object.

类采取值型行为。指针所指向的对象是唯一的，由每个类对象独立管理。

In this section we look at three classes that implement each of these different approaches to managing their pointer members.

本节中介绍三个类，分别实现管理指针成员的三种不同方法。

A Simple Class with a Pointer Member

一个带指针成员的简单类

To illustrate the issues involved, we'll implement a simple class that contains an `int` and a pointer:

为了阐明所涉及的问题，我们将实现一个简单类，该类包含一个 `int` 值和一个指针：

```
// class that has a pointer member that behaves like a plain pointer
class HasPtr {
public:
    // copy of the values we're given
    HasPtr(int *p, int i): ptr(p), val(i) {}

    // const members to return the value of the indicated data member
    int *get_ptr() const { return ptr; }
    int get_int() const { return val; }

    // non const members to change the indicated data member
    void set_ptr(int *p) { ptr = p; }
    void set_int(int i) { val = i; }

    // return or change the value pointed to, so ok for const objects
    int get_ptr_val() const { return *ptr; }
    void set_ptr_val(int val) const { *ptr = val; }
```

```
private:
    int *ptr;
    int val;
};
```

The `HasPtr` constructor takes two parameters, which it copies into `HasPtr`'s data members. The class provides simple accessor functions: The `const` functions `get_int` and `get_ptr` return the value of the `int` and pointer members, respectively; the `set_int` and `set_ptr` members let us change these members, giving a new value to the `int` or making the pointer point to a different object. We also define the `get_ptr_val` and `set_ptr_val` members. These members get and set the underlying value to which the pointer points.

`HasPtr` 构造函数接受两个形参，将它们复制到 `HasPtr` 的数据成员。`HasPtr` 类提供简单的访问函数：函数 `get_int` 和 `get_ptr` 分别返回 `int` 成员和指针成员的值：`set_int` 和 `set_ptr` 成员则使我们能够改变这些成员，给 `int` 成员一个新值或使指针成员指向不同的对象。还定义了 `get_ptr_val` 和 `set_ptr_val` 成员，它们能够获取和设置指针所指向的基础值。

Default Copy/Assignment and Pointer Members

默认复制／赋值与指针成员

Because the class does not define a copy constructor, copying one `HasPtr` object to another copies both members:

因为 `HasPtr` 类没有定义复制构造函数，所以复制一个 `HasPtr` 对象将复制两个成员：

```
int obj = 0;
HasPtr ptr1(&obj, 42); // int* member points to obj, val is 42
HasPtr ptr2(ptr1);     // int* member points to obj, val is 42
```

After the copy, the pointers in `ptr1` and `ptr2` both address the same object and the `int` values in each object are the same. However, the behavior of these two members appears quite different, because the value of a pointer is distinct from the value of the object to which it points. After the copy, the `int` values are distinct and independent, whereas the pointers are intertwined.

复制之后，`ptr1` 和 `ptr2` 中的指针指向同一对象且两个对象中的 `int` 值相同。但是，因为指针的值不同于它所指对象的值，这两个成员的行为看来非常不同。复制之后，`int` 值是清楚和独立的，而指针则纠缠在一起。



Classes that have pointer members and use default synthesized copy control have all the pitfalls of ordinary pointers. In particular, the class itself has no way to avoid dangling pointers.

具有指针成员且使用默认合成复制构造函数的类具有普通指针的所有缺陷。尤其是，类本身无法避免悬垂指针。

Pointers Share the Same Object

指针共享同一对象

When we copy an arithmetic value, the copy is independent from the original. We can change one copy without changing the other:

复制一个算术值时，副本独立于原版，可以改变一个副本而不改变另一个：

```
ptr1.set_int(0); // changes val member only in ptr1
ptr2.get_int();  // returns 42
ptr1.get_int();  // returns 0
```

When we copy a pointer, the address values are distinct, but the pointers point to the same underlying object. If we call `set_ptr_val` on either object, the underlying object is changed for both:

复制指针时，地址值是可区分的，但指针指向同一基础对象。如果在任一对象上调用 `set_ptr_val`，则二者的基础对象都会改变：

Section 13.5. Managing Pointer Members

```
ptr1.set_ptr_val(42); // sets object to which both ptr1 and ptr2 point  
ptr2.get_ptr_val(); // returns 42
```

When two pointers point to the same object, either one can change the value of the shared object.

两个指针指向同一对象时，其中任意一个都可以改变共享对象的值。

Dangling Pointers Are Possible

可能出现悬垂指针

Because our class copies the pointers directly, it presents our users with a potential problem: `HasPtr` stores the pointer it was given. It is up to the user to guarantee that the object to which that pointer points stays around as long as the `HasPtr` object does:

因为类直接复制指针，会使用户面临潜在的问题：`HasPtr` 保存着给定指针。用户必须保证只要 `HasPtr` 对象存在，该指针指向的对象就存在：

```
int *ip = new int(42); // dynamically allocated int initialized to 42  
HasPtr ptr(ip, 10); // Has Ptr points to same object as ip does  
delete ip; // object pointed to by ip is freed  
ptr.set_ptr_val(0); // disaster: The object to which Has Ptr points was freed!
```

The problem here is that `ip` and the pointer inside `ptr` both point to the same object. When that object is deleted, the pointer inside `HasPtr` no longer points to a valid object. However, there is no way to know that the object is gone.

这里的问题是 `ip` 和 `ptr` 中的指针指向同一对象。删除了该对象时，`ptr` 中的指针不再指向有效对象。然而，没有办法得知对象已经不存在了。

Exercises Section 13.5

Exercise 13.20: Given the original version of the `HasPtr` class that relies on the default definitions for copy-control, describe what happens in the following code:

对于 `HasPtr` 类的原始版本（依赖于复制控制的默认定义），描述下面代码中会发生什么：

```
int i = 42;  
HasPtr p1(&i, 42);  
HasPtr p2 = p1;  
cout << p2.get_ptr_val() << endl;  
p1.set_ptr_val(0);  
cout << p2.get_ptr_val() << endl;
```

Exercise 13.21: What would happen if we gave our `HasPtr` class a destructor that `deleted` its pointer member?

如果给 `HasPtr` 类添加一个析构函数，用来删除指针成员，会发生什么？

13.5.1. Defining Smart Pointer Classes

13.5.1. 定义智能指针类

In the previous section we defined a simple class that held a pointer and an `int`. The pointer member behaved in all ways like any other pointer. Any changes made to the object to which the pointer pointed were made to a single, shared object. If the user deleted that object, then our class had a dangling pointer. Its pointer member pointed at an object that no longer existed.

上节中我们定义了一个简单类，保存一个指针和一个 `int` 值。其中指针成员的行为与其他任意指针完全相同。对该指针指向的对象所做的任意改变都将作用于共享对象。如果用户删除该对象，则类就有一个悬垂指针，指向一个不复存在的对象。

An alternative to having a pointer member behave exactly like a pointer is to define what is sometimes referred to as a [smart pointer](#) class. A

Section 13.5. Managing Pointer Members

smart pointer behaves like an ordinary pointer except that it adds functionality. In this case, we'll give our smart pointer the responsibility for deleting the shared object. Users will dynamically allocate an object and pass the address of that object to our new `HasPtr` class. The user may still access the object through a plain pointer but must not `delete` the pointer. The `HasPtr` class will ensure that the object is deleted when the last `HasPtr` that points to it is destroyed.

除了使指针成员与指针完全相同之外，另一种方法是定义所谓的智能指针类。智能指针除了增加功能外，其行为像普通指针一样。本例中让智能指针负责删除共享对象。用户将动态分配一个对象并将该对象的地址传给新的 `HasPtr` 类。用户仍然可以通过普通指针访问对象，但绝不能删除指针。`HasPtr` 类将保证在撤销指向对象的最后一个 `HasPtr` 对象时删除对象。

In other ways, our `HasPtr` will behave like a plain pointer. In particular, when we copy a `HasPtr` object, the copy and the original will point to the same underlying object. If we change that object through one copy, the value will be changed when accessed through the other.

`HasPtr` 在其他方面的行为与普通指针一样。具体而言，复制对象时，副本和原对象将指向同一基础对象，如果通过一个副本改变基础对象，则通过另一对象访问的值也会改变。

Our new `HasPtr` class will need a destructor to delete the pointer. However, the destructor cannot delete the pointer unconditionally. If two `HasPtr` objects point to the same underlying object, we don't want to delete the object until both objects are destroyed. To write the destructor, we need to know whether this `HasPtr` is the last one pointing to a given object.

新的 `HasPtr` 类需要一个析构函数来删除指针，但是，析构函数不能无条件地删除指针。如果两个 `HasPtr` 对象指向同一基础对象，那么，在两个对象都撤销之前，我们并不希望删除基础对象。为了编写析构函数，需要知道这个 `HasPtr` 对象是否为指向给定对象的最后一个。

Introducing Use Counts

引入使用计数

A common technique used in defining smart pointers is to use a [use count](#). The pointerlike class associates a counter with the object to which the class points. The use count keeps track of how many objects of the class share the same pointer. When the use count goes to zero, then the object is deleted. A use count is sometimes also referred to as a [reference count](#).

定义智能指针的通用技术是采用一个[使用计数](#)。智能指针类将一个计数器与类指向的对象相关联。使用计数跟踪该类有多少个对象共享同一指针。使用计数为 0 时，删除对象。使用计数有时也称为[引用计数](#)。

Each time a new object of the class is created, the pointer is initialized and the use count is set to 1. When an object is created as a copy of another, the copy constructor copies the pointer and increments the associated use count. When an object is assigned to, the assignment operator decrements the use count of the object to which the left-hand operand points (and deletes that object if the use count goes to zero) and increments the use count of the object pointed to by the right-hand operand. Finally, when the destructor is called, it decrements the use count and deletes the underlying object if the count goes to zero.

每次创建类的新对象时，初始化指针并将使用计数置为 1。当对象作为另一对象的副本而创建时，复制构造函数复制指针并增加与之相应的使用计数的值。对一个对象进行赋值时，赋值操作符减少左操作数所指对象的使用计数的值（如果使用计数减至 0，则删除对象），并增加右操作数所指对象的使用计数的值。最后，调用析构函数时，析构函数减少使用计数的值，如果计数减至 0，则删除基础对象。

The only wrinkle is deciding where to put the use count. The counter cannot go directly into our `HasPtr` object. To see why, consider what happens in the following case:

唯一的创新在于决定将使用计数放在哪里。计数器不能直接放在 `HasPtr` 对象中，为什么呢？考虑下面的情况：

```
int obj;
HasPtr p1(&obj, 42);
HasPtr p2(p1); // p1 and p2 both point to same int object
HasPtr p3(p1); // p1, p2, and p3 all point to same int object
```

If the use count is stored in a `HasPtr` object, how can we update it correctly when `p3` is created? We could increment the count in `p1` and copy that count into `p3`, but how would we update the counter in `p2`?

如果使用计数保存在 `HasPtr` 对象中，创建 `p3` 时怎样更新它？可以在 `p1` 中将计数增量并复制到 `p3`，但怎样更新 `p2` 中的计数？

The Use-Count Class

使用计数类

There are two classic strategies for implementing a use count, one of which we will use here; the other approach is described in [Section 15.8.1](#) (p. 599). In the approach we use here, we'll define a separate concrete class to encapsulate the use count and the associated pointer:

实现使用计数有两种经典策略，在这里将使用其中一种，另一种方法在[第 15.8.1 节](#)中讲述。这里所用的方法中，需要定义一个单独的具体类以封闭使用计数和相关指针：

```
// private class for use by HasPtr only
```

Section 13.5. Managing Pointer Members

```
class U_Ptr {  
    friend class HasPtr;  
    int *ip;  
    size_t use;  
    U_Ptr(int *p): ip(p), use(1) {}  
    ~U_Ptr() { delete ip; }  
};
```

All the members of this class are `private`. We don't intend ordinary users to use the `U_Ptr` class, so we do not give it any `public` members. The `HasPtr` class is made a friend so that its members will have access to the members of `U_Ptr`.

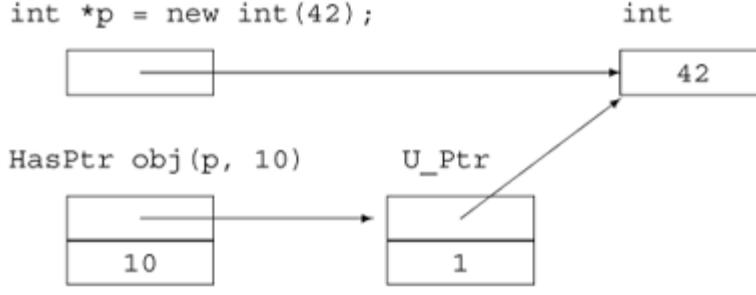
这个类的所有成员均为 `private`。我们不希望用户使用 `U_Ptr` 类，所以它没有任何 `public` 成员。将 `HasPtr` 类设置为友元，使其成员可以访问 `U_Ptr` 的成员。

The class is pretty simple, although the concept of how it works can be slippery. The `U_Ptr` class holds the pointer and the use count. Each `HasPtr` will point to a `U_Ptr`. The use count will keep track of how many `HasPtr` objects point to each `U_Ptr` object. The only functions `U_Ptr` defines are its constructor and destructor. The constructor copies the pointer, which the destructor deletes. The constructor also sets the use count to 1, indicating that a `HasPtr` object points to this `U_Ptr`.

尽管该类的工作原理比较难，但这个类相当简单。`U_Ptr` 类保存指针和使用计数，每个 `HasPtr` 对象将指向一个 `U_Ptr` 对象，使用计数将跟踪指向每个 `U_Ptr` 对象的 `HasPtr` 对象的数目。`U_Ptr` 定义的仅有函数是构造函数和析构函数，构造函数复制指针，而析构函数删除它。构造函数还将使用计数置为 1，表示一个 `HasPtr` 对象指向这个 `U_Ptr` 对象。

Assuming we just created a `HasPtr` object from a pointer that pointed to an `int` value of 42, we might picture the objects as follows:

假定刚从指向 `int` 值 42 的指针创建一个 `HasPtr` 对象，可以画出这些对象，如下图：



If we copy this object, then the objects will be as shown on the next page.

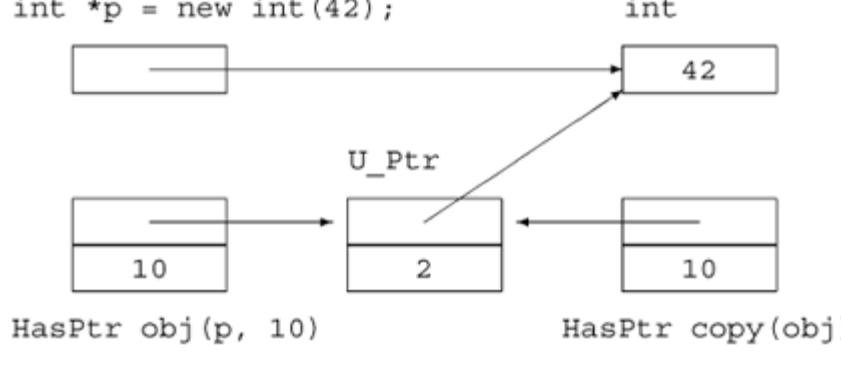
如果复制这个对象，则对象如下图所示。

Using the Use-Counted Class

使用计数类的使用

Our new `HasPtr` class holds a pointer to a `U_Ptr`, which in turn points to the actual underlying `int` object. Each member must be changed to reflect the fact that the class points to a `U_Ptr` rather than an `int*`.

新的 `HasPtr` 类保存一个指向 `U_Ptr` 对象的指针，`U_Ptr` 对象指向实际的 `int` 基础对象。必须改变每个成员以说明的 `HasPtr` 类指向一个 `U_Ptr` 对象而不是一个 `int`*



We'll look first at the constructors and copy-control members:

先看看构造函数和复制控制成员：

```
/* smart pointer class: takes ownership of the dynamically allocated  
*          object to which it is bound  
* User code must dynamically allocate an object to initialize a HasPtr
```

Section 13.5. Managing Pointer Members

```
* and must not delete that object; the HasPtr class will delete it
*/
class HasPtr {
public:
    // HasPtr owns the pointer; pmust have been dynamically allocated
    HasPtr(int *p, int i): ptr(new U_Ptr(p)), val(i) {}

    // copy members and increment the use count
    HasPtr(const HasPtr &orig):
        ptr(orig.ptr), val(orig.val) { ++ptr->use; }
    HasPtr& operator=(const HasPtr&);

    // if use count goes to zero, delete the U_Ptr object
    ~HasPtr() { if (--ptr->use == 0) delete ptr; }

private:
    U_Ptr *ptr;      // points to use-counted U_Ptr class
    int val;
};
```

The `HasPtr` constructor that takes a pointer and an `int` uses its pointer parameter to create a new `U_Ptr` object. After the `HasPtr` constructor completes, the `HasPtr` object points to a newly allocated `U_Ptr` object. That `U_Ptr` object stores the pointer we were given. The use count in that new `U_Ptr` is 1, indicating that only one `HasPtr` object points to it.

接受一个指针和一个 `int` 值的 `HasPtr` 构造函数使用其指针形参创建一个新的 `U_Ptr` 对象。`HasPtr` 构造函数执行完毕后，`HasPtr` 对象指向一个新分配的 `U_Ptr` 对象，该 `U_Ptr` 对象存储给定指针。新 `U_Ptr` 中的使用计数为 1，表示只有一个 `HasPtr` 对象指向它。

The copy constructor copies the members from its parameter and increments the use count. After the constructor completes, the newly created object points to the same `U_Ptr` object as the original and the use count of that `U_Ptr` object is incremented by one.

复制构造函数从形参复制成员并增加使用计数的值。复制构造函数执行完毕后，新创建对象与原有对象指向同一 `U_Ptr` 对象，该 `U_Ptr` 对象的使用计数加 1。

The destructor checks the use count in the underlying `U_Ptr` object. If the use count goes to 0, then this is the last `HasPtr` object that points to this `U_Ptr`. In this case, the `HasPtr` destructor deletes its `U_Ptr` pointer. Deleting that pointer has the effect of calling the `U_Ptr` destructor, which in turn deletes the underlying `int` object.

析构函数将检查 `U_Ptr` 基础对象的使用计数。如果使用计数为 0，则这是最后一个指向该 `U_Ptr` 对象的 `HasPtr` 对象，在这种情况下，`HasPtr` 析构函数删除其 `U_Ptr` 指针。删除该指针将引起对 `U_Ptr` 析构函数的调用，`U_Ptr` 析构函数删除 `int` 基础对象。

Assignment and Use Counts

赋值与使用计数

The assignment operator is a bit more complicated than the copy constructor:

赋值操作符比复制构造函数复杂一点：

```
HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    ++rhs.ptr->use;      // increment use count on rhs first
    if (--ptr->use == 0)
        delete ptr;      // if use count goes to 0 on this object, delete it
    ptr = rhs.ptr;        // copy the U_Ptr object
    val = rhs.val;        // copy the int member
    return *this;
}
```

Here we start by incrementing the use count in the right-hand operand. Then we decrement and check the use count on this object. As with the destructor, if this is the last object pointing to the `U_Ptr`, we delete the object, which in turn destroys the underlying `int`. Having decremented (and possibly destroyed) the existing value in the left-hand operand, we then copy the pointer from `rhs` into this object. As usual, assignment returns a reference to this object.

在这里，首先将右操作数中的使用计数加 1，然后将左操作数对象的使用计数减 1 并检查这个使用计数。像析构函数中那样，如果这是指向 `U_Ptr` 对象的最后一个对象，就删除该对象，这会依次撤销 `int` 基础对象。将左操作数中的当前值减 1（可能撤销该对象）之后，再将指针从 `rhs` 复制到这个对象。赋值照常返回对这个对象的引用。



This assignment operator guards against self-assignment by incrementing the use count of `rhs` before decrementing the use count of the left-hand operand.

这个赋值操作符在减少左操作数的使用计数之前使 `rhs` 的使用计数加 1，从而防止自身赋值。

If the left and right operands are the same, the effect of this assignment operator will be to increment and then immediately decrement the use count in the underlying `U_Ptr` object.

如果左右操作数相同，赋值操作符的效果将是 `U_Ptr` 基础对象的使用计数加 1 之后立即减 1。

Changing Other Members

改变其他成员

The other members that access the `int*` now need to change to get to the `int` indirectly through the `U_Ptr` pointer:

现在需要改变访问 `int*` 的其他成员，以便通过 `U_Ptr` 指针间接获取 `int`:

```
class HasPtr {
public:
    // copy control and constructors as before
    // accessors must change to fetch value from U_Ptr object
    int *get_ptr() const { return ptr->ip; }
    int get_int() const { return val; }

    // change the appropriate data member
    void set_ptr(int *p) { ptr->ip = p; }
    void set_int(int i) { val = i; }

    // return or change the value pointed to, so ok for const objects
    // Note: *ptr->ip is equivalent to *(ptr->ip)
    int get_ptr_val() const { return *ptr->ip; }
    void set_ptr_val(int i) { *ptr->ip = i; }
private:
    U_Ptr *ptr;           // points to use-counted U_Ptr class
    int val;
};
```

The functions that get and set the `int` member are unchanged. Those that operate on the pointer have to dereference the `U_Ptr` to get to the underlying `int*`.

获取和设置 `int` 成员的函数不变。那些使用指针操作的函数必须对 `U_Ptr` 解引用，以便获取 `int*` 基础对象。

When we copy `HasPtr` objects, the `int` member behaves the same as in our first class. Its value is copied; the members are independent. The pointer members in the copy and the original still point to the same underlying object. A change made to that object will affect the value as seen by either `HasPtr` object. However, users of `HasPtr` do not need to worry about dangling pointers. As long as they let the `HasPtr` class take care of freeing the object, the class will ensure that the object stays around as long as there are `HasPtr` objects that point to it.

复制 `HasPtr` 对象时，`int` 成员的行为与第一个类中一样。所复制的是 `int` 成员的值，各成员是独立的，副本和原对象中的指针仍指向同一基础对象，对基础对象的改变将影响通过任一 `HasPtr` 对象所看到的值。然而，`HasPtr` 的用户无须担心悬垂指针。只要他们让 `HasPtr` 类负责释放对象，`HasPtr` 类将保证只要有指向基础对象的 `HasPtr` 对象存在，基础对象就存在。

Advice: Managing Pointer Members

建议：管理指针成员

Objects with pointer members often need to define the copy-control members. If we rely on the synthesized versions, then the class puts a burden on its users. Users must ensure that the object to which the member points stays around for at least as long as the object that points to it does.

具有指针成员的对象一般需要定义复制控制成员。如果依赖合成版本，会给类的用户增加负担。用户必须保证成员所指向的对象存在，只要还有对象指向该对象。

To manage a class with pointer members, we must define all three copy-control members: the copy constructor, assignment operator, and the destructor. These members can define either pointerlike or valuelike behavior for the pointer member.

为了管理具有指针成员的类，必须定义三个复制控制成员：复制构造函数、赋值操作符和析构函数。这些成员可以定义指针成员的指针型行为或值型行为。

Valuelike classes give each object its own copy of the underlying values pointed to by pointer members. The copy constructor allocates a new element and copies the value from the object it is copying. The assignment operator destroys the existing object it holds and copies the value from its right-hand operand into its left-hand operand. The destructor destroys the object.

Section 13.5. Managing Pointer Members

值型类将指针成员所指基础值的副本给每个对象。复制构造函数分配新元素并从被复制对象处复制值，赋值操作符撤销所保存的原对象并从右操作数向左操作数复制值，析构函数撤销对象。

As an alternative to defining either valuelike behavior or pointerlike behavior some classes are so-called "smart pointers." These classes share the same underlying value between objects, thus providing pointerlike behavior. But they use copy-control techniques to avoid some of the pitfalls of regular pointers. To implement smart pointer behavior, a class needs to ensure that the underlying object stays around until the last copy goes away. Use counting ([Section 13.5.1, p. 495](#)), is a common technique for managing smart pointer classes. Each copy of the same underlying value is given a use count. The copy constructor copies the pointer from the old object into the new one and increments the use count. The assignment operator decrements the use count of the left-hand operand and increments the count of the right-hand operand. If the use count of the left-hand operand goes to zero, the assignment operator must delete the object to which it points. Finally, the assignment operator copies the pointer from the right-hand operand into its left-hand operand. The destructor decrements the use count and deletes the underlying object if the count goes to zero.

作为定义值型行为或指针型行为的另一选择，是使用称为“智能指针”的一些类。这些类在对象间共享同一基础值，从而提供了指针型行为。但它们使用复制控制技术以避免常规指针的一些缺陷。为了实现智能指针行为，类需要保证基础对象一直存在，直到最后一个副本消失。使用计数（[第 13.5.1 节](#)）是管理智能指针类的通用技术。同一基础值的每个副本都有一个使用计数。复制构造函数将指针从旧对象复制到新对象时，会将使用计数加 1。赋值操作符将左操作数的使用计数减 1 并将右操作数的使用计数加 1，如果左操作数的使用计数减至 0，赋值操作符必须删除它所指向的对象，最后，赋值操作符将指针从右操作数复制到左操作数。析构函数将使用计数减 1，并且，如果使用计数减至 0，就删除基础对象。



These approaches to managing pointers occur so frequently that programmers who use classes with pointer members must be thoroughly familiar with these programming techniques.

管理指针的这些方法用得非常频繁，因此使用带指针成员类的程序员必须充分熟悉这些编程技术。

Exercises Section 13.5.1

Exercise What is a use count?

13.22:

什么是使用计数？

Exercise What is a smart pointer? How does a smart pointer class differ from one that implements plain pointer behavior?

13.23:

什么是智能指针？智能指针类如何与实现普通指针行为的类相区别？

Exercise Implement your own version of the use-counted `HasPtr` class.

13.24:

实现你自己的使用计数的 `HasPtr` 类的版本。

13.5.2. Defining Valuelike Classes

13.5.2. 定义值型类

A completely different approach to the problem of managing pointer members is to give them [value semantics](#). Simply put, classes with value semantics define objects that behave like the arithmetic types: When we copy a valuelike object, we get a new, distinct copy. Changes made to the copy are not reflected in the original, and vice versa. The `string` class is an example of a valuelike class.

处理指针成员的另一个完全不同的方法，是给指针成员提供[值语义](#)。具有值语义的类所定义的对象，其行为很像算术类型的对象：复制值型对象时，会得到一个不同的新副本。对副本所做的改变不会反映在原有对象上，反之亦然。`string` 类是值型类的一个例子。

To make our pointer member behave like a value, we must copy the object to which the pointer points whenever we copy the `HasPtr` object:

要使指针成员表现得像一个值，复制 `HasPtr` 对象时必须复制指针所指向的对象：

Section 13.5. Managing Pointer Members

```
/*
 * Valuelike behavior even though HasPtr has a pointer member:
 * Each time we copy a HasPtr object, we make a new copy of the
 * underlying int object to which ptr points.
 */
class HasPtr {
public:
    // no point to passing a pointer if we're going to copy it anyway
    // store pointer to a copy of the object we're given
    HasPtr(const int &p, int i): ptr(new int(p)), val(i) {}

    // copy members and increment the use count
    HasPtr(const HasPtr &orig):
        ptr(new int (*orig.ptr)), val(orig.val) {}

    HasPtr& operator=(const HasPtr&);

    ~HasPtr() { delete ptr; }

    // accessors must change to fetch value from Ptr object
    int get_ptr_val() const { return *ptr; }
    int get_int() const { return val; }

    // change the appropriate data member
    void set_ptr(int *p) { ptr = p; }
    void set_int(int i) { val = i; }

    // return or change the value pointed to, so ok for const objects
    int *get_ptr() const { return ptr; }
    void set_ptr_val(int p) const { *ptr = p; }

private:
    int *ptr;           // points to an int
    int val;
};
```

The copy constructor no longer copies the pointer. It now allocates a new `int` object and initializes that object to hold the same value as the object of which it is a copy. Each object always holds its own, distinct copy of its `int` value. Because each object holds its own copy, the destructor unconditionally deletes the pointer.

复制构造函数不再复制指针，它将分配一个新的 `int` 对象，并初始化该对象以保存与被复制对象相同的值。每个对象都保存属于自己的 `int` 值的不同副本。因为每个对象保存自己的副本，所以析构函数将无条件删除指针。

The assignment operator doesn't need to allocate a new object. It just has to remember to assign a new value to the object to which its `int` pointer points rather than assigning to the pointer itself:

赋值操作符不需要分配新对象，它只是必须记得给其指针所指向的对象赋新值，而不是给指针本身赋值：

```
HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    // Note: Every HasPtr is guaranteed to point at an actual int;
    // We know that ptr cannot be a zero pointer
    *ptr = *rhs.ptr;           // copy the value pointed to
    val = rhs.val;             // copy the int
    return *this;
}
```

In other words, we change the value pointed to but not the pointer.

换句话说，改变的是指针所指向的值，而不是指针。



As always, the assignment operator must be correct even if we're assigning an object to itself. In this case, the operations are inherently safe even if the left- and right-hand objects are the same. Thus, there is no need to explicitly check for self-assignment.

即使要将一个对象赋值给它本身，赋值操作符也必须总是保证正确。本例中，即使左右操作数相同，操作本质上也是安全的，因此，不需要显式检查自身赋值。

Exercises Section 13.5.2

Exercise What is a valuelike class?

13.25:

什么是值型类？

Exercise Implement your own version of a valuelike `HasPtr` class.

13.26:

实现你自己的值型 `HasPtr` 类版本?

Exercise The valuelike `HasPtr` class defines each of the copy-control members. Describe what would

13.27:

happen if the class defined

值型 `HasPtr` 类定义了所有复制控制成员。描述将会发生什么，如果该类：

- a. The copy constructor and destructor but no assignment operator.

定义了复制构造函数和析构函数但没有定义赋值操作符。

- b. The copy constructor and assignment operator but no destructor.

定义了复制构造函数和赋值操作符但没有定义析构函数。

- c. The destructor but neither the copy constructor nor assignment operator.

定义了析构函数但没有定义复制构造函数和赋值操作符。

Exercise Given the following classes, implement a default constructor and the necessary copy-control

13.28:

members.

对于如下的类，实现默认构造函数和必要的复制控制成员。

```
(a) class TreeNode {
    public:
        //...
    private:
        std::string value;
        int count;
        TreeNode *left;
        TreeNode *right;
};

(b) class BinStrTree {
    public:
        //...
    private:
        TreeNode *root;
};
```

Chapter Summary

小结

In addition to defining the operations on objects of its type, a class also defines what it means to copy, assign, or destroy objects of the type. Special member functions—the copy constructor, the assignment operator, and the destructor define these operations. Collectively these operations are referred to as the "copy control" functions.

类除了定义该类型对象上的操作，还需要定义复制、赋值或撤销该类型对象的含义。特殊成员函数（复制构造函数、赋值操作符和析构函数）可用于定义这些操作。这些操作统称为“复制控制”函数。

If a class does not define one or more of these operations, the compiler will define them automatically. The synthesized operations perform memberwise initialization, assignment, or destruction: Taking each member in turn, the synthesized operation does whatever is appropriate to the member's type to copy, assign, or destroy that member. If the member is a class type, the synthesized operation calls the corresponding operation for that class (e.g., the copy constructor calls the member's copy constructor, the destructor calls its destructor, etc.). If the member is a built-in type or a pointer, the member is copied or assigned directly; the destructor does nothing to destroy members of built-in or pointer type. If the member is an array, the elements in the array are copied, assigned, or destroyed in a manner appropriate to the element type.

如果类没有定义这些操作中的一个或多个，编译器将自动定义它们。合成操作执行逐个成员初始化、赋值或撤销：合成操作依次取得每个成员，根据成员类型进行成员的复制、赋值或撤销。如果成员为类类型的，合成操作调用该类的相应操作（即，复制构造函数调用成员的复制构造函数，析构函数调用成员的析构函数，等等）。如果成员为内置类型或指针，则直接复制或赋值，析构函数对撤销内置类型或指针类型的成员没有影响。如果成员为数组，则根据元素类型以适当方式复制、赋值或撤销数组中的元素。

Unlike the copy constructor and assignment operator, the synthesized destructor is created and run, regardless of whether the class defines its own destructor. The synthesized destructor is run after the class-defined destructor, if there is one, completes.

与复制构造函数和赋值操作符不同，无论类是否定义了自己的析构函数，都会创建和运行合成析构函数。如果类定义了析构函数，则在类定义的析构函数结束之后运行合成析构函数。



The hardest part of defining the copy-control functions is often simply recognizing that they are necessary.

定义复制控制函数最为困难的部分通常在于认识到它们的必要性。

Classes that allocate memory or other resources almost always require that the class define the copy-control members to manage the allocated resource. If a class needs a destructor, then it almost surely needs to define the copy constructor and assignment operator as well.

分配内存或其他资源的类几乎总是需要定义复制控制成员来管理所分配的资源。如果一个类需要析构函数，则它几乎也总是需要定义复制构造函数和赋值操作符。

Defined Terms

术语

assignment operator (赋值操作符)

The assignment operator can be overloaded to define what it means to assign one object of a class type to another of the same type. The assignment operator must be a member of its class and should return a reference to its object. The compiler synthesizes the assignment operator if the class does not explicitly define one.

赋值操作符可以重载，对将某个类类型对象赋值给另一同类型对象的含义进行定义。赋值操作符必须是类的成员并且必须返回对所属类对象的引用。如果类没有定义赋值操作符，则编译器将会合成一个。

copy constructor (复制构造函数)

Constructor that initializes a new object as a copy of another object of the same type. The copy constructor is applied implicitly to pass objects to or from a function by value. If we do not define the copy constructor, the compiler synthesizes one for us.

将新对象初始化为另一同类型对象的副本的构造函数。显式应用复制构造函数进行对象的按值传递，向函数传递对象或从函数返回对象。如果没有定义复制构造函数，编译器就为我们合成一个。

copy control (复制控制)

Special members that control what happens when object of class type are copied, assigned, and destroyed. The compiler synthesizes appropriate definitions for these operations if the class does not otherwise define them.

控制类类型对象的复制、赋值和撤销的特殊成员。如果类没有另外定义它们，则编译器将为这些操作合成适当定义。

destructor (析构函数)

Special member function that cleans up an object when the object goes out of scope or is deleted. The compiler automatically destroys each member. Members of class type are destroyed by invoking their destructor; no explicit work is done to destroy members of built-in or compound type. In particular, the object pointed to by a pointer member is not deleted by the automatic work done by the destructor.

当对象超出作用域或删除对象时，清除对象的特殊成员函数。编译器将自动撤销每个成员。类类型的成员通过调用它们的析构函数而撤销，撤销内置或复合类型的成员无须做显式工作。特别地，析构函数的自动工作不会删除指针成员所指向的对象。

memberwise assignment (逐个成员复制)

Term used to describe how the synthesized assignment operator works. The assignment operator assigns, member by member, from the old object to the new. Members of built-in or compound type are assigned directly. Those that are of class type are assigned by using the member's assignment operator.

用于描述合成赋值操作符如何工作的术语。赋值操作符将成员依次从旧对象赋值给新对象。内置或复合类型成员直接赋值，类类型成员使用成员的赋值操作符进行赋值。

memberwise initialization (逐个成员初始化)

Term used to described how the synthesized copy constructor works. The constructor copies, member by member, from the old object to the new. Members of built-in or compound type are copied directly. Those that are of class type are copied by using the member's copy constructor.

用于描述合成复制构造函数如何工作的术语。构造函数将成员依次从旧对象复制到新对象。内置或复合类型成员直接复制，类类型成员使用成员的复制构造函数进行复制。

overloaded operator (重载操作符)

Function that redefines one of the C++ operators to operate on object(s) of class type. This chapter showed how to define the assignment operator; [Chapter 14](#) covers overloaded operators in more detail.

将一个 C++ 操作符重定义以操作类类型对象的函数。本章介绍了如何定义赋值操作符，[第十四章](#)将更详细地介绍重载操作符。

reference count (引用计数)

Synonym for use count.

使用计数的同义词。

Rule of Three (三法则)

Shorthand for the rule of thumb that if a class needs a nontrivial destructor then it almost surely also needs to define its own copy constructor and an assignment operator.

一个经验原则的简写形式，即，如果一个类需要析构函数，则该类几乎也必然需要定义自己的复制构造函数和赋值操作符。

smart pointer (智能指针)

A class that behaves like a pointer but provides other functionality as well. One common form of smart pointer takes a pointer to a dynamically allocated object and assumes responsibility for deleting that object. The user allocates the object, but the smart pointer class deletes it. Smart pointer classes require that the class implement the copy-control members to manage a pointer to the shared object. That object is deleted only when the last smart pointer pointing to it is destroyed. Use counting is the most popular way to implement smart pointer classes.

一个行为类似指针但也提供其他功能的类。智能指针的一个通用形式接受指向动态分配对象的指针并负责删除该对象。用户分配对象，但由智能指针类删除它。智能指针类需要实现复制控制成员来管理指向共享对象的指针。只有在撤销了指向共享对象的最后一个智能指针后，才能删除该共享对象。使用计数是实现智能指针类最常用的方式。

synthesized assignment operator (合成赋值操作符)

A version of the assignment operator created (synthesized) by the compiler for classes that do not explicitly define one. The synthesized assignment operator memberwise assigns the right-hand operand to the left.

由编译器为没有显式定义赋值操作符的类创建（合成）的赋值操作符版本。合成赋值操作符将右操作数逐个成员地赋值给左操作数。

synthesized copy constructor (合成复制构造函数)

The copy constructor created (synthesized) by the compiler for classes that do not explicitly define the copy constructor. The synthesized copy constructor memberwise initializes the new object from the existing one.

由编译器为没有显式定义复制构造函数的类创建（合成）的复制构造函数。合成复制构造函数将原对象逐个成员地初始化新对象。

use count (使用计数)

Programming technique used in copy-control members. A use count is stored along with a shared object. A separate class is created that points to the shared object and manages the use count. The constructors, other than the copy constructor, set the state of the shared object and set the use count to one. Each time a new copy is made either in the copy constructor or the assignment operator the use count is incremented. When an object is destroyed either in the destructor or as the left-hand side of the assignment operator the use count is decremented. The assignment operator and the destructor check whether the decremented use count has gone to zero and, if so, they destroy the object.

复制控制成员中使用的编程技术。使用计数与共享对象一起存储。需要创建一个单独类指向共享对象并管理使用计数。由构造函数，而不是复制构造函数，设置共享对象的状态并将使用计数置为 1。每当由复制构造函数或赋值操作符生成一个新副本时，使用计数加 1。由析构函数撤销对象或作为赋值操作符的左操作数撤销对象时，使用计数减 1。赋值操作符和析构函数检查使用计数是否已减至 0。如果是，则撤销对象。

value semantics (值语义)

Description of the copy-control behavior of classes that mimic the way arithmetic types are copied. Copies of valuelike objects are independent: Changes made to a copy have no effect on the original object. A valuelike class that has a pointer member must define its own copy-control members. The copy-control operations copy the object to which the pointer points. Valuelike classes that contain only other valuelike classes or built-in types often can rely on the synthesized copy-control members.

对类模拟算术类型复制方式的复制控制行为的描述。值型副本是独立的：对副本的改变不会影响原有对象。具有指针成员的值型类必须定义自己的复制控制成员。复制控制操作复制指针所指向的对象。只包含其他值型类或内置类型的值型类，通常可以依赖合成的复制控制成员。

Chapter 14. Overloaded Operations and Conversions

第十四章 重载操作符与转换

CONTENTS

目录

<u>Section 14.1</u> Defining an Overloaded Operator	506
<u>Section 14.2</u> Input and Output Operators	513
<u>Section 14.3</u> Arithmetic and Relational Operators	517
<u>Section 14.4</u> Assignment Operators	520
<u>Section 14.5</u> Subscript Operator	522
<u>Section 14.6</u> Member Access Operators	523
<u>Section 14.7</u> Increment and Decrement Operators	526
<u>Section 14.8</u> Call Operator and Function Objects	530
<u>Section 14.9</u> Conversions and Class Types	535
<u>Chapter Summary</u>	552
<u>Defined Terms</u>	552

In [Chapter 5](#) we saw that C++ defines a large number of operators and automatic conversions among the built-in types. These facilities allow programmers to write a rich set of mixed-type expressions.

第五章介绍过，C++ 定义了许多内置类型间的操作符和自动转换。使用这些设施程序员能够编写丰富的混合类型表达式。

C++ lets us redefine the meaning of the operators when applied to objects of class type. It also lets us define conversion operations for class types. [Class-type conversions](#) are used like the built-in conversions to implicitly convert an object of one type to another type when needed.

C++ 允许我们重定义操作符用于尖类型对象的含义。如果需要，可以像内置转换那样使用 [类型转换](#)，将一个类型的对象隐式转换到另一类型。

Operator overloading allows the programmer to define versions of the operators for operands of class type. [Chapter 13](#) covered the importance of the assignment operator and showed how to define the assignment operator. We first used overloaded operators in [Chapter 1](#), when our programs used the shift operators (`>>` and `<<`) for input and output and the addition operator (`+`) to add two `Sales_items`. We'll finally see in this chapter how to define these overloaded operators.

通过操作符重载，程序员能够针对尖空尖的操作定义不同的操作符版本。[第十三章](#)阐述了赋值操作符的重要性并介绍了怎样定义赋值操作符。我们第一次使用重载操作符是在[第一章](#)，那里程序用移位操作符(`>>`和`<<`)进行输入输出，用加号操作符(`+`)将两个 `Sales_items` 相加。本章我们终于可以看到怎样定义这些重载操作符了。

Through operator overloading, we can redefine most of the operators from [Chapter 3](#) to work on objects of class type. Judicious use of operator overloading can make class types as intuitive to use as the built-in types. For example, the standard library defines several overloaded operators for the container classes. These classes define the subscript operator to access data elements and `*` and `->` to dereference container iterators. The fact that these library types have the same operators makes using them similar to using built-in arrays and pointers. Allowing programs to use expressions rather than named functions can make the programs much easier to write and read. As an example, compare

通过操作符重载，可以重定义第五章介绍的大多数操作符，使它们用于尖尖型对象。明智地使用操作符重载可以使尖尖型的使用像内向尖尖一样直观。标准库为容器尖尖定义了几个重载操作符。这些容器类定义了下标操作符以访问数据元素，定义了`*`和`->`对容器迭代器解引用。这些标准库的类型具有相同的操作符，使用它们就像使用内置数组和指针一样。允许程序使用表达式而不是命名函数，可以使编写和阅读程序容易得多。将

```
cout << "The sum of " << v1 << " and " << v2  
     << " is " << v1 + v2 << endl;
```

to the more verbose code that would be necessary if TO used named functions.

和以下更为冗长的代码相比较就能够看到。如果 I/O 使用命名函数，类似下面的代码将无法避免：

```
// hypothetical expression if 10 used named functions
cout.print("The sum of ").print(v1).
    print(" and ").print(v2).print(" is ").
    print(v1 + v2).print("\n").flush();
```

Team LiB

◀ PREVIOUS NEXT ▶

14.1. Defining an Overloaded Operator

14.1. 重载操作符的定义

Overloaded operators are functions with special names: the keyword `operator` followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

重载操作符是具有特殊名称的函数：保留字 `operator` 后接需定义的操作符号。像任意其他函数一样，重载操作符具有返回类型和形参表，如下语句：

```
Sales_item operator+(const Sales_item&, const Sales_item&);
```

declares the addition operator that can be used to "add" two `Sales_item` objects and yields a copy of a `Sales_item` object.

声明了加号操作符，可用于将两个 `Sales_item` 对象“相加”并获得一个 `Sales_item` 对象的副本。

With the exception of the function-call operator, an overloaded operator has the same number of parameters (including the implicit `this` pointer for member functions) as the operator has operands. The function-call operator takes any number of operands.

除了函数调用操作符之外，重载操作符的形参数目（包括成员函数的隐式 `this` 指针）与操作符的操作数数目相同。函数调用操作符可以接受任意数目的操作数。

Overloaded Operator Names

重载的操作名

[Table 14.1](#) on the next page lists the operators that may be overloaded. Those that may not be overloaded are listed in [Table 14.2](#).

[表 14.1](#) 列出了可以重载的操作符，不能重载的在[表 14.2](#) 列出。

Table 14.1. Overloadable Operators

表 14.1. 可重载的操作符

+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Table 14.2. Operators That Cannot Be Overloaded

表 14.2. 不能重载的操作符

::	*	.	?:
----	---	---	----

New operators may not be created by concatenating other legal symbols. For example, it would be illegal to attempt to define an `operator**` to provide exponentiation. Overloading `new` and `delete` is described in [Chapter 18](#) (p. 753).

通过连接其他合法符号可以创建新的操作符。例如，定义一个 `operator**` 以提供求幂运算是合法的。[第十八章](#)将介绍重载 `new` 和 `delete`。

Overloaded Operators Must Have an Operand of Class Type

重载操作符必须具有一个类类型操作数

The meaning of an operator for the built-in types may not be changed. For example, the built-in integer addition operation cannot be redefined:

用于内置类型的操作符，其含义不能改变。例如，内置的整型加号操作符不能重定义：

```
// error: cannot redefine built-in operator for ints
int operator+(int, int);
```

Nor may additional operators be defined for the built-in data types. For example, an `operator+` taking two operands of array types cannot be defined.

也不能为内置数据类型重定义加号操作符。例如，不能定义接受两个数组类型操作数的 `operator+`。



An overloaded operator must have at least one operand of class or enumeration ([Section 2.7](#), p. 62) type. This rule enforces the requirement that an overloaded operator may not redefine the meaning of the operators when applied to objects of built-in type.

重载操作符必须具有至少一个类类型或枚举类型（[第 2.7 节](#)）的操作数。这条规则强制重载操作符不能重新定义用于内置类型对象的操作符的含义。

Precedence and Associativity Are Fixed

优先级和结合性是固定的

The precedence ([Section 5.10.1](#), p. 168), associativity, or number of operands of an operator cannot be changed. Regardless of the type of the operands and regardless of the definition of what the operations do, this expression

操作符的优先级（[第 5.10.1 节](#)）、结合性或操作数目不能改变。不管操作数的类型和操作符的功能定义如何，表达式

```
x == y + z;
```

always binds the arguments `y` and `z` to `operator+` and uses that result as the right-hand operand to `operator==`.

总是将实参 `y` 和 `z` 绑定到 `operator+`，并且将结果用作 `operator==` 右操作数。

Four symbols (+, -, *, and &) serve as both unary and binary operators. Either or both of these operators can be overloaded. Which operator is being defined is controlled by the number of operands. Default arguments for overloaded operators are illegal, except for `operator()`, the function-call operator.

有四个符号（+，-，* 和 &）既可作一元操作符又可作二元操作符，这些操作符有的在其中一种情况下可以重载，有的两种都可以，定义的是哪个操作符由操作数数目控制。除了函数调用操作符 `operator()` 之外，重载操作符时使用默认实参是非法的。

Short-Circuit Evaluation Is Not Preserved

不再具备短路求值特性

Overloaded operators make no guarantees about the order in which operands are evaluated. In particular, the operand-evaluation guarantees of the built-in logical AND, logical OR ([Section 5.2](#), p. 152), and comma ([Section 5.9](#), p. 168) operators are not preserved. Both operands to an overloaded version of `&&` or `||` are always evaluated. The order in which those operands are evaluated is not stipulated. The order in which the operands to the comma are evaluated is also not defined. For this reason, it is usually a bad idea to overload `&&`, `||`, or the comma operator.

重载操作符并不保证操作数的求值顺序，尤其是，不会保证内置逻辑 AND、逻辑 OR（[第 5.2 节](#)）和逗号操作符（[第 5.9 节](#)）的操作数求值。在 `&&` 和 `||` 的重载版本中，两个操作数都要进行求值，而且对操作数的求值顺序不做规定。因此，重载 `&&`、`||` 或逗号操作符不是一种好的做法。

Class Member versus Nonmember

类成员与非成员

Most overloaded operators may be defined as ordinary nonmember functions or as class member functions.

大多数重载操作符可以定义为普通非成员函数或类的成员函数。



Overloaded functions that are members of a class may appear to have one less parameter than the number of operands. Operators that are member functions have an implicit `this` parameter that is bound to the first operand.

作为类成员的重载函数，其形参看起来比操作数数目少 1。作为成员函数的操作符有一个隐含的 `this` 形参，限定为第一个操作数。

An overloaded unary operator has no (explicit) parameter if it is a member function and one parameter if it is a nonmember function. Similarly, an overloaded binary operator would have one parameter when defined as a member and two parameters when defined as a nonmember function.

重载一元操作符如果作为成员函数就没有（显式）形参，如果作为非成员函数就有一个形参。类似地，重载二元操作符定义为成员时有一个形参，定义为非成员函数时有两个形参。

The `Sales_item` class offers a good example of member and nonmember binary operators. We know that the class has an addition operator. Because it has an addition operator, we ought to define a compound-assignment (`+=`) operator as well. This operator will add the value of one `Sales_item` object into another.

类 `Sales_item` 中给出了成员和非成员二元操作符的良好例子。我们知道该类有一个加号操作符。因为它有一个加号操作符，所以也应该定义一个复合赋值 (`+=`) 操作符，该操作符将一个 `Sales_item` 对象的值加至另一个 `Sales_item` 对象。

Ordinarily we define the arithmetic and relational operators as nonmember functions and we define assignment operators as members:

一般将算术和关系操作符定义为非成员函数，而将赋值操作符定义为成员：

```
// member binary operator: left-hand operand bound to implicit this pointer
Sales_item& Sales_item::operator+=(const Sales_item&);
// nonmember binary operator: must declare a parameter for each operand
Sales_item operator+(const Sales_item&, const Sales_item&);
```

Both addition and compound assignment are binary operators, yet these functions define a different number of parameters. The reason for the discrepancy is the `this` pointer.

加和复合赋值都是二元操作符，但这些函数定义了不同数目的形参，差异的原因在于 `this` 指针。

When an operator is a member function, `this` points to the left-hand operand. Thus, the nonmember `operator+` defines two parameters, both references to `const Sales_item` objects. Even though compound assignment is a binary operator, the member compound-assignment operator takes only one (explicit) parameter. When the operator is used, a pointer to the left-hand operand is automatically bound to `this` and the right-hand operand is bound to the function's sole parameter.

当操作符为成员函数，`this` 指向左操作数，因此，非成员 `operator+` 定义两个形参，都引用 `const Sales_item` 对象。即使复合赋值是二元操作符，成员复合赋值操作符也只接受一个（显式的）形参。使用操作符时，一个指向左操作数的指针自动绑定到 `this`，而右操作符限定为函数的唯一形参。

It is also worth noting that compound assignment returns a reference and the addition operator returns a `Sales_item` object. This difference matches the return types of these operators when applied to arithmetic types: Addition yields an rvalue and compound assignment returns a reference to the left-hand operand.

复合赋值返回一个引用而加操作符返回一个 `Sales_item` 对象，这也没什么。当应用于算术类型时，这一区别与操作符的返回类型相匹配：加返回一个右值，而复合赋值返回对左操作数的引用。

Operator Overloading and Friendship

操作符重载和友元关系

When operators are defined as nonmember functions, they often must be made friends (Section 12.5, p. 465) of the class(es) on which they operate. We'll see later in this chapter two reasons why operators might be defined as nonmembers. In such cases, the operator often needs access to the private parts of the class.

Section 14.1. Defining an Overloaded Operator

操作符定义为非成员函数时，通常必须将它们设置为所操作类的友元（12.5 节）。在本章的后面部分，将给出操作符可以定义为非成员的两个原因。在这种情况下，操作符通常需要访问类的私有部分。

Our `Sales_item` class is again a good example of why some operators need to be friends. It defines one member operator and has three nonmember operators. Those nonmember operators, which need access to the private data members, are declared as friends:

`Sales_item` 类也是说明为何有些操作符需要设置为友元的一个好例子。它定义了一个成员操作符，并且有三个非成员操作符。这些非成员操作符需要访问私有数据成员，声明为友元：

```
class Sales_item {
    friend std::istream& operator>>
        (std::istream&, Sales_item&);
    friend std::ostream& operator<<
        (std::ostream&, const Sales_item&);

public:
    Sales_item& operator+=(const Sales_item&);
};

Sales_item operator+(const Sales_item&, const Sales_item&);
```

That the input and output operators need access to the `private` data should not be surprising. After all, they read and write those members. On the other hand, there is no need to make the addition operator a friend. It can be implemented using the `public` member `operator+=`.

输入和输出操作符需要访问 `private` 数据不会令人惊讶，毕竟，它们的作用是读入和写出那些成员。另一方面，不需要将加操作符设置为友元，它可以用 `public` 成员 `operator+=` 实现。

Using Overloaded Operators

使用重载操作符

We can use an overloaded operator in the same way that we'd use the operator on operands of built-in type. Assuming `item1` and `item2` are `Sales_item` objects, we might print their sum in the same way that we'd print the sum of two `int`s:

使用重载操作符的方式，与内置类型操作数上使用操作符的方式一样。假定 `item1` 和 `item2` 是 `Sales_item` 对象，可以打印它们的和，就像打印两个 `int` 的和一样：

```
cout << item1 + item2 << endl;
```

This expression implicitly calls the `operator+` that we defined for `Sales_items`.

这个表达式隐式调用为 `Sales_items` 类而定义的 `operator+.`

We also can call an overloaded operator function in the same way that we call an ordinary function: We name the function and pass an appropriate number of arguments of the appropriate type:

也可以像调用普通函数一样调用重载操作符函数，指定函数并传递适当类型的形参：

```
// equivalent direct call to nonmember operator function
cout << operator+(item1, item2) << endl;
```

This call has the same effect as the expression that added `item1` and `item2`.

这个调用与 `item1` 和 `item2` 相加的表达式等效。

We call a member operator function the same way we call any other member function: We name an object on which to run the function and then use the dot or arrow operator to fetch the function we wish to call passing the required number and type of arguments. In the case of a binary member operator function, we must pass a single operand:

调用成员操作符函数与调用任意其他函数是一样的：指定运行函数的对象，然后使用点或箭头操作符获取希望调用的函数，同时传递所需数目和类型的实参。对于二元成员操作符函数的情况，我们必须传递一个操作数：

```
item1 += item2;           // expression based "call"
item1.operator+=(item2);  // equivalent call to member operator function
```

Each of these statements adds the value of `item2` into `item1`. In the first case, we implicitly call the overloaded operator function using expression syntax. In the second, we call the member operator function on the object `item1`.

两个语句都将 `item2` 的值加至 `item1`。第一种情况下，使用表达式语法隐式调用重载操作符函数；第二种情况下，在 `item1` 对象上调用成员操作符函数。

Exercises Section 14.1

习题 14.1

Exercise 14.1: In what ways does an overloaded operator differ from a built-in operator? In what ways are overloaded operators the same as the built-in operators?

在什么情况下重载操作符与内置操作符不同？在什么情况下重载操作符与内置操作符相同？

Exercise 14.2: Write declarations for the overloaded input, output, addition and compound-assignment operators for `Sales_item`.

为 `Sales_item` 编写输入、输出、加以及复合赋值操作符的重载声明。

Exercise 14.3: Explain the following program, assuming that the `Sales_item` constructor that takes a `string` is not `explicit`. Explain what happens if that constructor is `explicit`.

解释如下程序，假定 `Sales_item` 构造函数的参数是一个 `string`，且不为 `explicit`。解释如果构造函数 `explicit` 怎样。

```
string null_book = "9-999-99999-9";
Sales_item item(cin);
item += null_book;
```

Exercise 14.4: Both the `string` and `vector` types define an overloaded `==` that can be used to compare objects of those types. Identify which version of `==` is applied in each of the following expressions:

`string` 和 `vector` 类都定义了一个重载的 `==`，可用于比较这些类的对象。指出下面表达式中应用了哪个 `==` 版本：

```
string s; vector<string> svec1, svec2;
"cobble" == "stone"
svec1[0] == svec2[0];
svec1 == svec2
```

14.1.1. Overloaded Operator Design

14.1.1. 重载操作符的设计

When designing a class there are some useful rules of thumb to keep in mind when deciding which, if any, overloaded operators to provide.

设计类的时候，需要记住一些有用的经验原则，可以有助于确定应该提供哪些重载操作符（如果需要提供）。

Don't Overload Operators with Built-in Meanings

不要重载具有内置含义的操作符

The assignment, address of, and comma operators have default meanings for operands of class types. If there is no overloaded version specified, the compiler defines its own version of these operators:

赋值操作符、取地址操作符和逗号操作符对类类型操作数有默认含义。如果没有特定重载版本，编译器就自己定义以下这些操作符。

- The synthesized assignment operator ([Section 13.2](#), p. 482) does memberwise assignment: It uses each member's own assignment operator to assign each member in turn.

合成赋值操作符 ([第 13.2 节](#)) 进行逐个成员赋值：使用成员自己的赋值：使用成员自己的赋值操作依次对每个成员进行赋值。

- By default the address of (&) and comma (,) operators execute on class type objects the same way they do on objects of built-in type. The

Section 14.1. Defining an Overloaded Operator

address of operator returns the address in memory of the object to which it is applied. The comma operator evaluates each expression from left to right and returns the value of its rightmost operand.

默认情况下，取地址操作符（`&`）和逗号操作符（`,`）在类类型对象上的执行，与在内置类型对象上的执行一样。取地址操作符返回对象的内存地址，逗号操作符从左至右计算每个表达式的值，并返回最右边操作数的值。

- The built-in logical AND (`&&`) and OR(`||`) operators apply short-circuit evaluation ([Section 5.2](#), p. [152](#)). If the operator is redefined, the short-circuit nature of the operators is lost.

内置逻辑与（`&&`）和逻辑或（`||`）操作符使用短路求值（[第 5.2 节](#)）。如果重新定义该操作符，将失去操作符的短路求值特征。

The meaning of these operators can be changed by redefining them for operands of a given class type.

通过为给定类类型的操作数重定义操作符，可以改变这些操作符的含义。



It is usually *not* a good idea to overload the comma, address-of, logical AND, or logical OR operators. These operators have built-in meanings that are useful and become inaccessible if we define our own versions.

重载逗号、取地址、逻辑与、逻辑或等等操作符通常不是好做法。这些操作符具有有用的内置含义，如果我们定义了自己的版本，就不能再使用这些内置含义。

We sometimes must define our own version of assignment. When we do so, it should behave analogously to the synthesized operators: After an assignment, the values in the left-hand and right-hand operands should be the same and the operator should return a reference to its left-hand operand. Overloaded assignment should customize the built-in meaning of assignment, not circumvent it.

有时我们需要定义自己的赋值运算。这样做时，它应表现得类似于合成操作符：赋值之后，左右操作数的值应是相同的，并且操作符应返回对左操作数的引用。重载的赋值运算应在赋值的内置含义基础上进行定制，而不是完全绕开。

Most Operators Have No Meaning for Class Objects

大多数操作符对类对象没有意义

Operators other than assignment, address-of, and comma have no meaning when applied to an operand of class type unless an overloaded definition is provided. When designing a class, we decide which, if any, operators to support.

除非提供了重载定义，赋值、取地址和逗号操作符对于类类型操作数没有意义。设计类的时候，应该确定要支持哪些操作符。

The best way to design operators for a class is first to design the class' public interface. Once the interface is defined, it is possible to think about which operations should be defined as overloaded operators. Those operations with a logical mapping to an operator are good candidates. For example,

为类设计操作符，最好的方式是首先设计类的公用接口。定义了接口之后，就可以考虑应将哪些操作符定义为重载操作符。那些逻辑上可以映射到某个操作符的操作可以考虑作为候选的重载操作符。例如：

- An operation to test for equality should use `operator==`.
相等测试操作应使用 `operator==`。
- Input and output are normally done by overloading the shift operators.
一般通过重载移位操作符进行输入和输出。
- An operation to test whether the object is empty could be represented by the logical NOT operator, `operator!`.
测试对象是否为空的操作可用逻辑非操作符 `operator!` 表示。

Compound Assignment Operators

复合赋值操作符

Section 14.1. Defining an Overloaded Operator

If a class has an arithmetic ([Section 5.1](#), p. 149) or bitwise ([Section 5.3](#), p. 154) operator, then it is usually a good idea to provide the corresponding compound-assignment operator as well. For example, our `Sales_item` class defined the `+` operator. Logically, it also should define `+=`. Needless to say, the `+=` operator should be defined to behave the same way the built-in operators do: Compound assignment should behave as `+` followed by `=`.

如果一个类有算术操作符（[第 5.1 节](#)）或位操作符（[第 5.3 节](#)），那么，提供相应的复合赋值操作符一般是个好的做法。例如，`Sales_item` 类定义了 `+` 操作符，逻辑上，它也应该定义 `+=`。不用说，操作符的行为应定义为与内置操作符一样：复合赋值的行为应与 `+` 之后接着 `=` 类似。

Equality and Relational Operators

相等和关系操作符

Classes that will be used as the key type of an associative container should define the `<` operator. The associative containers by default use the `<` operator of the key type. Even if the type will be stored only in a sequential container, the class ordinarily should define the equality (`==`) and less-than (`<`) operators. The reason is that many algorithms assume that these operators exist. As an example, the `sort` algorithm uses `<` and `find` uses `==`.

将要用作关联容器键类型的类应定义 `<` 操作符。关联容器默认使用键类型的 `<` 操作符。即使该类型将只存储在顺序容器中，类通常也应该定义相等 (`==`) 和小于 (`<`) 操作符，理由是许多算法假定这个操作符存在。例如 `sort` 算法使用 `<` 操作符，而 `find` 算法使用 `==` 操作符。

Caution: Use Operator Overloading Judiciously

警告：审慎使用操作符重载

Each operator has an associated meaning from its use on the built-in types. Binary `+`, for example, is strongly identified with addition. Mapping binary `+` to an analogous operation for a class type can provide a convenient notational shorthand. For example, the library `string` type, following a convention common to many programming languages, uses `+` to represent concatenation "adding" one string to the other.

每个操作符用于内置类型都有关联的含义。例如，二元 `+` 与加法是完全相同的。将二元 `+` 对应到一个类类型的类似操作可提供方便的简写方法。例如，标准库的类型 `string`，遵循许多程序设计语言的通用规范，使用 `+` 表示连接——将一个串“加”至另一个串。

Operator overloading is most useful when there is a logical mapping of a built-in operator to an operation on our type. Using overloaded operators rather than inventing named operations can make our programs more natural and intuitive. Overuse or outright abuse of operator overloading can make our classes incomprehensible.

当内置操作符和类型上的操作存在逻辑对应关系时，操作符重载最有用。使用重载操作符而不是创造命名操作，可以令程序更自然、更直观，而滥用操作符重载使得我们的类难以理解。

Obvious abuses of operator overloading rarely happen in practice. As an example, no responsible programmer would define `operator+` to perform subtraction. More common, but still inadvisable, are uses that contort an operator's "normal" meaning to force a fit to a given type. Operators should be used only for operations that are likely to be unambiguous to users. An operator with ambiguous meaning, in this sense, is one that supports equally well a number of different interpretations.

在实践中很少发生明显操作符重载滥用。例如，不负责任的程序员可能会定义 `operator+` 来执行减法。更常见但仍不可取的是，改变操作符的“正常”含义以强行适应给定类型。操作符应该只用于对用户而言无二义的操作。在这里所谓有二义的操作符，就是指具有多个不同解释的操作符。



When the meaning of an overloaded operator is not obvious, it is better to give the operation a name. It is also usually better to use a named function rather than an operator for operations that are rarely done. If the operation is unusual, the brevity of using an operator is unnecessary.

当一个重载操作符的含义不明显时，给操作取一个名字更好。对于很少用的操作，使用命名函数通常也比用操作符更好。如果不是普通操作，没有必要为简洁而使用操作符。

If the class defines the equality operator, it should also define `!=`. Users of the class will assume that if they can compare for equality, they can also compare for inequality. The same argument applies to the other relational operators as well. If the class defines `<`, then it probably should define all four relational operators (`>`, `>=`, `<`, and `<=`).

Section 14.1. Defining an Overloaded Operator

如果类定义了相等操作符，它也应该定义不等操作符 `!=`。类用户会假设如果可以进行相等比较，则也可以进行不等比较。同样的规则也应用于其他关系操作符。如果类定义了 `<`，则它可能应该定义全部的四个关系操作符 (`>`, `>=`, `<`, `<=`)。

Choosing Member or Nonmember Implementation

选择成员或非成员实现

When designing the overloaded operators for a class, we must choose whether to make each operator a class member or an ordinary nonmember function. In some cases, the programmer has no choice; the operator must be a member. In other cases, there are some rules of thumb that can help guide the decision. The following guidelines can be of help when deciding whether to make an operator a member or an ordinary nonmember function:

- The assignment (`=`), subscript (`[]`), call (`()`), and member access arrow (`->`) operators must be defined as members. Defining any of these operators as a nonmember function is flagged at compile time as an error.
赋值 (`=`)、下标 (`[]`)、调用 (`()`) 和成员访问箭头 (`->`) 等操作符必须定义为成员，将这些操作符定义为非成员函数将在编译时标记为错误。
- Like assignment, the compound-assignment operators ordinarily ought to be members of the class. Unlike assignment, they are not required to be so and the compiler will not complain if a nonmember compound-assignment operator is defined.
像赋值一样，复合赋值操作符通常应定义为类的成员，与赋值不同的是，不一定非得这样做，如果定义非成员复合赋值操作符，不会出现编译错误。
- Other operators that change the state of their object or that are closely tied to their given types such as increment, decrement, and dereference usually should be members of the class.
改变对象状态或与给定类型紧密联系的其他一些操作符，如自增、自减和解引用，通常就定义为类成员。
- Symmetric operators, such as the arithmetic, equality, relational, and bitwise operators, are best defined as ordinary nonmember functions.
对称的操作符，如算术操作符、相等操作符、关系操作符和位操作符，最好定义为普通非成员函数。

Exercises Section 14.1.1

Exercise 14.5:

List the operators that must be members of a class.

列出必须定义为类成员的操作符。

Exercise 14.6:

Explain why and whether each of the following operators should be class members:

解释下面操作符是否应该为类成员，为什么？

(a) + (b) += (c) ++ (d) -> (e) << (f) && (g) == (h) ()

14.2. Input and Output Operators

14.2. 输入和输出操作符

Classes that support I/O ordinarily should do so by using the same interface as defined by the `iostream` library for the built-in types. Thus, many classes provide overloaded instances of the input and output operators.

支持 I/O 操作的类所提供的 I/O 操作接口，一般应该与标准库 `iostream` 为内置类型定义的接口相同，因此，许多类都需要重载输入和输出操作符。

14.2.1. Overloading the Output Operator `<<`

14.2.1. 输出操作符 `<<` 的重载



To be consistent with the IO library, the operator should take an `ostream&` as its first parameter and a reference to a `const` object of the class type as its second. The operator should return a reference to its `ostream` parameter.

为了与 IO 标准库一致，操作符应接受 `ostream&` 作为第一个形参，对类类型 `const` 对象的引用作为第二个形参，并返回对 `ostream` 形参的引用。

The general skeleton of an overloaded output operator is

重载输出操作符一般的简单定义如下：

```
// general skeleton of the overloaded output operator
ostream&
operator <<(ostream& os, const ClassType &object)
{
    // any special logic to prepare object

    // actual output of members
    os << // ...

    // return ostream object
    return os;
}
```

The first parameter is a reference to an `ostream` object on which the output will be generated. The `ostream` is `nonconst` because writing to the stream changes its state. The parameter is a reference because we cannot copy an `ostream` object.

第一个形参是对 `ostream` 对象的引用，在该对象上将产生输出。`ostream` 为非 `const`，因为写入到流会改变流的状态。该形参是一个引用，因为不能复制 `ostream` 对象。

The second parameter ordinarily should be a `const` reference to the class type we want to print. The parameter is a reference to avoid copying the argument. It can be `const` because (ordinarily) printing an object should not change it. By making the parameter a `const` reference, we can use a single definition to print `const` and `nonconst` objects.

第二个形参一般应是对要输出的类类型的引用。该形参是一个引用以避免复制实参。它可以是 `const`，因为（一般而言）输出一个对象不应该改变对象。使形参成为 `const` 引用，就可以使用同一个定义来输出 `const` 和非 `const` 对象。

The return type is an `ostream` reference. Its value is usually the `ostream` object against which the output operator is applied.

返回类型是一个 `ostream` 引用，它的值通常是输出操作符所操作的 `ostream` 对象。

The `Sales_item` Output Operator

`Sales_item` 输出操作符

Section 14.2. Input and Output Operators

We can now write the `Sales_item` output operator:

现在可以编写 `Sales_item` 的输出操作符了：

```
ostream&
operator<<(ostream& out, const Sales_item& s)
{
    out << s.isbn << "\t" << s.units_sold << "\t"
        << s.revenue << "\t" << s.avg_price();
    return out;
}
```

Printing a `Sales_item` entails printing its three data elements and the computed average sales price. Each element is separated by a tab. After printing the values, the operator returns a reference to the `ostream` it just wrote.

输出 `Sales_item`, 就需要输出它的三个数据成员以及计算得到的平均销售价格，每个成员用制表符间隔。输出值之后，该操作符返回对所写 `ostream` 对象的引用。

Output Operators Usually Do Minimal Formatting

输出操作符通常所做格式化尽量少

Class designers face one significant decision about output: whether and how much formatting to perform.

关于输出，类设计者面临一个重要决定：是否格式化以及进行多少格式化。



Generally, output operators should print the contents of the object, with minimal formatting. They should not print a newline.

一般而言，输出操作符应输出对象的内容，进行最小限度的格式化，它们不应该输出换行符。

The output operators for the built-in types do little if any formatting and do not print newlines. Given this treatment for the built-in types, users expect class output operators to behave similarly. By limiting the output operator to printing just the contents of the object, we let the users determine what if any additional formatting to perform. In particular, an output operator should not print a newline. If the operator does print a newline, then users would be unable to print descriptive text along with the object on the same line. By having the output operator perform minimal formatting, we let users control the details of their output.

用于内置类型的输出操作符所做格式化很少，并且不输出换行符。由于内置类型的这种既定处理，用户预期类输出操作符也有类似行为。通过限制输出操作符只输出对象的内容，如果需要执行任意额外的格式化，我们让用户决定该如何处理。尤其是，输出操作符不应该输出换行符，如果该操作符输出换行符，则用户就不能将说明文字与对象输出在同一行上。尽量减少操作符所做格式化，让用户自己控制输出细节。

IO Operators Must Be Nonmember Functions

IO 操作符必须为非成员函数

When we define an input or output operator that conforms to the conventions of the `iostream` library, we must make it a nonmember operator. Why?

当定义符合标准库 `iostream` 规范的输入或输出操作符的时候，必须使它成为非成员操作符，为什么需要这样做呢？

We cannot make the operator a member of our own class. If we did, then the left-hand operand would have to be an object of our class type:

我们不能将该操作符定义为类的成员，否则，左操作数将只能是该类类型的对象：

```
// if operator<< is a member of Sales_item
Sales_item item;
item << cout;
```

This usage is the opposite of the normal way we use output operators defined for other types.

Section 14.2. Input and Output Operators

这个用法与为其他类型定义的输出操作符的正常使用方式相反。

If we want to support normal usage, then the left-hand operand must be of type `ostream`. That means that if the operator is to be a member of any class, it must be a member of class `ostream`. However, that class is part of the standard library. We and anyone else who wants to define IO operators can't go adding members to a class in the library.

如果想要支持正常用法，则左操作数必须为 `ostream` 类型。这意味着，如果该操作符是类的成员，则它必须是 `ostream` 类的成员，然而，`ostream` 类是标准库的组成部分，我们（以及任何想要定义 IO 操作符的人）是不能为标准库中的类增加成员的。

Instead, if we want to use the overloaded operators to do IO for our types, we must define them as a nonmember functions. IO operators usually read or write the nonpublic data members. As a consequence, classes often make the IO operators friends.

相反，如果想要使用重载操作符为该类型提供 IO 操作，就必须将它们定义为非成员函数。IO 操作符通常对非公用数据成员进行读写，因此，类通常将 IO 操作符设为友元。

Exercises Section 14.2.1

Exercise Define an output operator for the following `ChecoutRecord` class:

14.7: 为下面的 `ChecoutRecord` 类定义一个输出操作符：

```
class ChecoutRecord {  
public:  
    // ...  
private:  
    double book_id;  
    string title;  
    Date date_borrowed;  
    Date date_due;  
    pair<string,string> borrower;  
    vector< pair<string,string>*> wait_list;  
};
```

Exercise In the exercises to [Section 12.4](#) (p. 451) you wrote a sketch of one of the following classes:

14.8: [第 12.4 节](#) 的习题中，你编写了下面某个类的框架：

- (a) Book (b) Date (c) Employee
- (d) Vehicle (e) Object (f) Tree

Write the output operator for the class you chose.

为所选择的类编写输出操作符。

14.2.2. Overloading the Input Operator `>>`

14.2.2. 输入操作符 `>>` 的重载

Similar to the output operator, the input operator takes a first parameter that is a reference to the stream from which it is to read, and returns a reference to that same stream. Its second parameter is a nonconst reference to the object into which to read. The second parameter must be nonconst because the purpose of an input operator is to read data into this object.

与输出操作符类似，输入操作符的第一个形参是一个引用，指向它要读的流，并且返回的也是对同一个流的引用。它的第二个形参是对要读入的对象的非 `const` 引用，该形参必须为非 `const`，因为输入操作符的目的是将数据读到这个对象中。



A more important, and less obvious, difference between input and output operators is that input operators must deal with the possibility of errors and end-of-file.

更重要但通常重视不够的是，输入和输出操作符有如下区别：输入操作符必须处理错误和文件结束的可能性。

The `Sales_item` Input Operator

`Sales_item` 的输入操作符

The `Sales_item` input operator looks like:

`Sales_item` 的输入操作符如下：

```
istream&
operator>>(istream& in, Sales_item& s)
{
    double price;
    in >> s.isbn >> s.units_sold >> price;
    // check that the inputs succeeded
    if (in)
        s.revenue = s.units_sold * price;
    else
        s = Sales_item(); // input failed: reset object to default state
    return in;
}
```

This operator reads three values from its `istream` parameter: a `string` value, which it stores in the `isbn` member of its `Sales_item` parameter; an `unsigned`, which it stores in the `units_sold` member; and a `double`, which it stores in a local named `price`. Assuming the reads succeed, the operator uses `price` and `units_sold` to set the object's `revenue` member.

这个操作符从 `istream` 形参中读取三个值：一个 `string` 值，存储到 `isbn` 成员中；一个 `unsigned` 值，存储到 `Sales_item` 形参的 `units_sold` 成员中；一个 `double` 值，存储到 `Sales_item` 形参的 `price` 成员中。假定读取成功，操作符用 `price` 和 `units_sold` 来设置 `Sales_item` 对象的 `revenue` 成员。

Errors During Input

输入期间的错误

Our `Sales_item` input operator reads the expected values and checks whether an error occurred. The kinds of errors that might happen include:

`Sales_item` 的输入操作符将读入所期望的值并检查是否发生错误。可能发生的错误包括如下种类：

1. Any of the read operations could fail because an incorrect value was provided. For example, after reading `isbn`, the input operator assumes that the next two items will be numeric data. If nonnumeric data is input, that read and any subsequent use of the stream will fail.

任何读操作都可能因为提供的值不正确而失败。例如，读入 `isbn` 之后，输入操作符将期望下两项是数值型数据。如果输入非数值型数据，这次的读入以及流的后续使用都将失败。

2. Any of the reads could hit end-of-file or some other error on the input stream.

任何读入都可能碰到输入流中的文件结束或其他一些错误。

Rather than checking each read, we check once before using the data we read:

我们无需检查每次读入，只在使用读入数据之前检查一次即可：

```
// check that the inputs succeeded
if (in)
    s.revenue = s.units_sold * price;
else
    s = Sales_item(); // input failed: reset object to default state
```

If one of the reads failed, then `price` would be uninitialized. Hence, before using `price`, we check that the input stream is still valid. If it is, we do the calculation and store it in `revenue`. If there was an error, we do not worry about which input failed. Instead, we reset the entire object as if it were an empty `Sales_item`. We do so by creating a new, unnamed `Sales_item` constructed using the default constructor and assigning that value to `s`. After this assignment, `s` will have an empty `string` for its `isbn` member, and its `revenue` and `units_sold` members will be zero.

如果这些读入有一个失败了，则 `price` 可能没有初始化。因此，在使用 `price` 之前，我们需要检查输入流是否仍有效。如果有效，就进行计算并将结果存储到 `revenue` 中；如果出现了错误，我们不用关心是哪个输入失败了，相反，我们将整个对象复位，就好像它是一个空 `Sales_item` 对象，具体做法是创建一个新的、未命名的、用默认构造的 `Sales_item` 对象并将它赋值给 `s`。赋值之后，`s` 的 `isbn` 成员是一个空 `string`，它的 `revenue` 和 `units_sold` 成员为 0。

Handling Input Errors

处理输入错误

Section 14.2. Input and Output Operators

If an input operator detects that the input failed, it is often a good idea to make sure that the object is in a usable and consistent state. Doing so is particularly important if the object might have been partially written before the error occurred.

如果输入操作符检测到输入失败了，则确保对象处于可用和一致的状态是个好做法。如果对象在错误发生之前已经写入了部分信息，这样做就特别重要。

For example, in the `Sales_item` input operator, we might successfully read a new `isbn`, and then encounter an error on the stream. An error after reading `isbn` would mean that the `units_sold` and `revenue` members of the old object were unchanged. The effect would be to associate a different `isbn` with that data.

例如，在 `Sales_item` 的输入操作符中，可能成功地读入了一个新的 `isbn`，然后遇到流错误。在读入 `isbn` 之后发生错误意味着旧对象的 `units_sold` 和 `revenue` 成员没变，结果会将另一个 `isbn` 与那个数据关联。

In this operator, we avoid giving the parameter an invalid state by resetting it to the empty `Sales_item` if an error occurs. A user who needs to know whether the input succeeded can test the stream. If the user ignores the possibility of an input error, the object is in a usable state its members are all defined. Similarly, the object won't generate misleading results its data are internally consistent.

在这个操作符中，如果发生了错误，就将形参恢复为空 `Sales_item` 对象，以避免给它一个无效状态。用户如果需要输入是否成功，可以测试流。即使用户忽略了输入可能错误，对象仍处于可用状态——它的成员都已经定义。类似地，对象将不会产生令人误解的结果——它的数据是内在一致的。



When designing an input operator, it is important to decide what to do about error-recovery, if anything.

设计输入操作符时，如果可能，要确定错误恢复措施，这很重要。

Indicating Errors

指出错误

In addition to handling any errors that might occur, an input operator might need to set the condition state ([Section 8.2](#), p. 287) of its input `istream` parameter. Our input operator is quite simple—the only errors we care about are those that could happen during the reads. If the reads succeed, then our input operator is correct and has no need to do additional checking.

除了处理可能发生的任何错误之外，输入操作符还可能需要设置输入形参的条件状态（[第 8.2 节](#)）。我们的输入操作符相当简单——我们只关心读入期间可能发生的错误。如果读入成功，则输入操作符就是正确的而且不需要进行附加检查。

Some input operators do need to do additional checking. For example, our input operator might check that the `isbn` we read is in an appropriate format. We might have read data successfully, but these data might not be suitable when interpreted as an ISBN. In such cases, the input operator might need to set the condition state to indicate failure, even though technically speaking the actual IO was successful. Usually an input operator needs to set only the `failbit`. Setting `eofbit` would imply that the file was exhausted, and setting `badbit` would indicate that the stream was corrupted. These errors are best left to the IO library itself to indicate.

有些输入操作符的确需要进行附加检查。例如，我们的输入操作符可以检查读到的 `isbn` 格式是否恰当。也许我们已成功读取了数据，但这些数据不能恰当解释为 ISBN，在这种情况下，尽管从技术上说实际的 IO 是成功的，但输入操作符仍可能需要设置条件状态以指出失败。通常输入操作符仅需设置 `failbit`。设置 `eofbit` 意思是文件耗尽，设置 `badbit` 可以指出流被破坏，这些错误最好留给 IO 标准库自己来指出。

Exercises Section 14.2.2

Exercise 14.9: Describe the behavior of the `Sales_item` input operator if given the following input:
给定下述输入，描述 `Sales_item` 输入操作符的行为。

- (a) 0-201-99999-9 10 24.95
- (b) 10 24.95 0-210-99999-9

Exercise 14.10: What is wrong with the following `Sales_item` input operator?
下述 `Sales_item` 输入操作符有什么错误？

```
istream& operator>>(istream& in, Sales_item& s)
{
    double price;
```

Section 14.2. Input and Output Operators

```
in >> s.isbn >> s.units_sold >> price;
s.revenue = s.units_sold * price;
return in;
```

What would happen if we gave this operator the data in the previous exercise?

如果将上题中的数据作为输入，将会发生什么？

Exercise 14.11: Define an input operator for the `CheckoutRecord` class defined in the exercises for [Section 14.2.1](#) (p. [515](#)). Be sure the operator handles input errors.

为[第 14.2.1 节](#)习题中定义的 `CheckoutRecord` 类定义一个输入操作符，确保该操作符处理输入错误。

Team LiB

◀ PREVIOUS NEXT ▶

14.3. Arithmetic and Relational Operators

14.3. 算术操作符和关系操作符

Ordinarily, we define the arithmetic and relational operators as nonmember functions, as we do here with our `Sales_item` addition operator:

一般而言，将算术和关系操作符定义为非成员函数，像下面给出的 `Sales_item` 加法操作符一样：

```
// assumes that both objects refer to the same isbn
Sales_item
operator+(const Sales_item& lhs, const Sales_item& rhs)
{
    Sales_item ret(lhs); // copy lhs into a local object that we'll return
    ret += rhs;          // add in the contents of rhs
    return ret; // return ret by value
}
```

The addition operator doesn't change the state of either operand; the operands are references to `const` objects. Instead, it generates and returns a new `Sales_item` object, which is initialized as a copy of `lhs`. We use the `Sales_item` compound-assignment operator to add in the value of `rhs`.

加法操作符并不改变操作符的状态，操作符是对 `const` 对象的引用；相反，它产生并返回一个新的 `Sales_item` 对象，该对象初始化为 `lhs` 的副本。我们使用 `Sales_item` 的复合赋值操作符来加入 `rhs` 的值。

 Note that to be consistent with the built-in operator, addition returns an rvalue, not a reference.

注意，为了与内置操作符保持一致，加法返回一个右值，而不是一个引用。

An arithmetic operator usually generates a new value that is the result of a computation on its two operands. That value is distinct from either operand and is calculated in a local variable. It would be a run-time error to return a reference to that variable.

算术操作符通常产生一个新值，该值是两个操作数的计算结果，它不同于任一操作数且在一个局部变量中计算，返回对那个变量的引用是一个运行时错误。

 Classes that define both an arithmetic operator and the related compound assignment ordinarily ought to implement the arithmetic operator by using the compound assignment.

既定义了算术操作符又定义了相关复合赋值操作符的类，一般应使用复合赋值实现算术操作符。

It is simpler and more efficient to implement the arithmetic operator (e.g., `+`) in terms of the compound-assignment operator (e.g., `+=`) rather than the other way around. As an example, consider our `Sales_item` operators. If we implemented `+=` by calling `+`, then `+=` would needlessly create and destroy a temporary to hold the result from `+`.

根据复合赋值操作符（如 `+=`）来实现算术操作符（如 `+`），比其他方式更简单且更有效。例如，我们的 `Sales_item` 操作符。如果我们调用 `+=` 来实现 `+`，则可以不必创建和撤销一个临时量来保存 `+` 的结果。

Exercises Section 14.3

Exercise Write the `Sales_item` operators so that `+` does the actual addition and `+=` calls `+`. Discuss the

Section 14.3. Arithmetic and Relational Operators

14.12: disadvantages of this approach compared to the way the operators were implemented in this section.

编写 `Sales_item` 操作符，用 `+` 进行实际加法，而 `+=` 调用 `+`。与本节中操作符的实现方法相比较，讨论这个方法的缺点。

Exercise 14.13: Which other arithmetic operators, if any, do you think `Sales_item` ought to support? Define those that you think the class should include.

如果有，你认为 `Sales_item` 还应该支持哪些其他算术操作符？定义你认为该类应包含的哪些。

14.3.1. Equality Operators

14.3.1. 相等操作符

Ordinarily, classes in C++ use the equality operator to mean that the objects are equivalent. That is, they usually compare every data member and treat two objects as equal if and only if all corresponding members are the same. In line with this design philosophy, our `Sales_item` equality operator should compare the `isbn` as well as the sales figures:

通常，C++ 中的类使用相等操作符表示对象是等价的。即，它们通常比较每个数据成员，如果所有对应成员都相同，则认为两个对象相等。与这一设计原则一致，`Sales_item` 的相等操作符应比较 `isbn` 以及销售数据：

```
inline bool
operator==(const Sales_item &lhs, const Sales_item &rhs)
{
    // must be made a friend of Sales_item
    return lhs.units_sold == rhs.units_sold &&
           lhs.revenue == rhs.revenue &&
           lhs.same_isbn(rhs);
}
inline bool
operator!=(const Sales_item &lhs, const Sales_item &rhs)
{
    return !(lhs == rhs); // != defined in terms of operator==
}
```

The definition of these functions is trivial. More important are the design principles that these functions embody:

这些函数的定义并不重要，重要的是这些函数所包含的设计原则：

- If a class defines the `==` operator, it defines it to mean that two objects contain the same data.

如果类定义了 `==` 操作符，该操作符的含义是两个对象包含同样的数据。

- If a class has an operation to determine whether two objects of the type are equal, it is usually right to define that function as `operator==` rather than inventing a named operation. Users will expect to be able to compare objects using `==`, and doing so is easier than remembering a new name.

如果类具有一个操作，能确定该类型的两个对象是否相等，通常将该函数定义为 `operator==` 而不是创造命名函数。用户将习惯于用 `==` 来比较对象，而且这样做比记住新名字更容易。

- If a class defines `operator==`, it should also define `operator!=`. Users will expect that if they can use one operator, then the other will also exist.

如果类定义了 `operator==`，它也应该定义 `operator!=`。用户会期待如果可以用某个操作符，则另一个也存在。

- The equality and inequality operators should almost always be defined in terms of each other. One operator should do the real work to compare objects. The other should call the one that does the real work.

相等和不操作符一般应该相互联系起来定义，让一个操作符完成比较对象的实际工作，而另一个操作符只是调用前者。



Classes that define `operator==` are easier to use with the standard library. Some algorithms, such as `find`, use the `==` operator by default. If a class defines `==`, then these algorithms can be used on that class type without any specialization.

定义了 `operator==` 的类更容易与标准库一起使用。有些算法，如 `find`，默认使用 `==` 操作符，如果类定义了 `==`，则这些算法可以无须任何特殊处理而用于该类类型。

14.3.2. Relational Operators

14.3.2. 关系操作符

Classes for which the equality operator is defined also often have relational operators. In particular, because the associative containers and some of the algorithms use the less-than operator, it can be quite useful to define an `operator<`.

定义了相等操作符的类一般也具有关系操作符。尤其是，因为关联容器和某些算法使用小于操作符，所以定义 `operator<` 可能相当有用。

Although we might think our `Sales_item` class should support the relational operators, it turns out that it probably should not. The reasons are somewhat subtle and deserve understanding.

我们也许认为 `Sales_item` 类应该支持关系操作符，但恰恰相反，它很可能不应该支持关系操作符，原因有些微妙，值得了解。

As we'll see in [Chapter 15](#), we might want to use an associative container to hold `Sales_item` transactions. When we put objects into the container, we'd want them ordered by ISBN, and wouldn't care whether the sales data in two records were different.

正如[第十五章](#)将要介绍的，我们可能想要使用关联容器来保存 `Sales_item` 事务。将对象放在容器中时，我们希望它们按 ISBN 排序，而不会关心两个记录中的销售数据是否不同。

However, if we were to define `operator<` as comparison on `isbn`, that definition would be incompatible with the obvious definition of `==`. If we had two transactions for the same ISBN, neither record would be less than the other. Yet, if the sales figures in those objects were different, then these objects would be `!=`. Ordinarily, if we have two objects, neither of which is less than the other, then we expect that those objects are equal.

但是，如果将 `operator<` 定义为对 `isbn` 的比较，该定义将与前面 `==` 的定义不相容。如果有两个针对同一 ISBN 的事务，其中任意一个都不会小于另一个，然而，如果这两个对象中的销售数据不同，则它们就不相等。但是，一般说来，如果有两个对象，其中任意一个都不小于另一个，则认为它们相等。

Because the logical definition of `<` is inconsistent with the logical definition of `==`, it is better not to define `<` at all. We'll see in [Chapter 15](#) how to use a separate named function to compare `Sales_items` when we want to store them in an associative container.

因为 `<` 的逻辑定义与 `==` 的逻辑定义不一致，所以根本不定义 `<` 会更好。[第十五章](#)将会介绍想要将 `Sales_items` 对象存储到关联容器中时，怎样使用单独的命名函数来比较 `Sales_items` 对象。



The associative containers, as well as some of the algorithms, use the `<` operator by default. Ordinarily, the relational operators, like the equality operators, should be defined as nonmember functions.

关联容器以及某些算法，使用默认 `<` 操作符。一般而言，关系操作符，诸如相等操作符，应定义为非成员函数。

14.4. Assignment Operators

14.4. 赋值操作符

We covered the assignment of one object of class type to another object of its type in [Section 13.2](#) (p. 482). The class assignment operator takes a parameter that is the class type. Usually the parameter is a `const` reference to the class type. However, the parameter could be the class type or a `nonconst` reference to the class type. This operator will be synthesized by the compiler if we do not define it ourselves. The class assignment operator must be a member of the class so the compiler can know whether it needs to synthesize one.

[第 13.2 节](#) 讨论了类类型对象对同类型其他对象的赋值。类赋值操作符接受类类型形参，通常，该形参是对类类型的 `const` 引用，但也可以是类类型或对类类型的非 `const` 引用。如果没有定义这个操作符，则编译器将合成它。类赋值操作符必须是类的成员，以便编译器可以知道是否需要合成一个。

Additional assignment operators that differ by the type of the right-hand operand can be defined for a class type. For example, the library `string` class defines three assignment operators: In addition to the class assignment operator, which takes a `const string&` as its right-hand operand, the `string` class defines versions of assignment that take a C-style character string or a `char` as the right-hand operand. These might be used as follows:

可以为一个类定义许多附加的赋值操作符，这些赋值操作符会因右操作符类型不同而不同。例如，标准库的类 `string` 定义了 3 个赋值操作符：除了接受 `const string&` 作为右操作数的类赋值操作符之外，类还定义了接受 C 风格字符串或 `char` 作为右操作数的赋值操作符，这些操作符可以这样使用：

```
string car ("Volks");
car = "Studebaker"; // string = const char*
string model;
model = 'T'; // string = char
```

To support these operations, the `string` class contains members that look like

为了支持这些操作符，`string` 类包含如下成员：

```
// illustration of assignment operators for class string
class string {
public:
    string& operator=(const string &);      // s1 = s2;
    string& operator=(const char *);          // s1 = "str";
    string& operator=(char);                 // s1 = 'c';
    // ...
};
```



Assignment operators can be overloaded. Unlike the compound-assignment operators, every assignment operator, regardless of parameter type, must be defined as a member function.

赋值操作符可以重载。无论形参为何种类型，赋值操作符必须定义为成员函数，这一点与复合赋值操作符有所不同。

Assignment Should Return a Reference to `*this`

赋值必须返回对 `*this` 的引用

The `string` assignment operators return a reference to `string`, which is consistent with assignment for the built-in types. Moreover, because assignment returns a reference there is no need to create and destroy a temporary copy of the result. The return value is usually a reference to the left-hand operand. For example, here is the definition of the `Sales_item` compound-assignment operator:

`string` 赋值操作符返回 `string` 引用，这与内置类型的赋值一致。而且，因为赋值返回一个引用，就不需要创建和撤销结果的临时副本。返回值通常是左操作数的引用，例如，这是 `Sales_item` 复合赋值操作符的定义：

```
// assumes that both objects refer to the same isbn
Sales_item& Sales_item::operator+=(const Sales_item& rhs)
{
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}
```



Ordinarily, assignment operators and compound-assignment operators ought to return a reference to the left-hand operand.

一般而言，赋值操作符与复合赋值操作符应返回操作符的引用。

Exercises Section 14.4

Exercise Define a version of the assignment operator that can assign an `isbn` to a `Sales_item`.

14.14:

定义一个赋值操作符，将 `isbn` 赋值给 `Sales_item` 对象。

Exercise Define the class assignment operator for the `CheckoutRecord` introduced in the exercises to

14.15:

[Section 14.2.1](#) (p. 515).

为[第 14.2.1 节](#)习题中介绍的 `CheckoutRecord` 类定义赋值操作符。

Exercise Should `CheckoutRecord` define any other assignment operators? If so, explain which types should

14.16:

be used as operands and why. Implement the assignment operators for those types.

`CheckoutRecord` 类还应该定义其他赋值操作符吗？如果是，解释哪些类型应该用作操作数并解释为什么。为这些类型实现赋值操作符。

Team LiB

◀ PREVIOUS NEXT ▶

14.5. Subscript Operator

14.5. 下标操作符

Classes that represent containers from which individual elements can be retrieved usually define the subscript operator, `operator[]`. The library classes, `string` and `vector`, are examples of classes that define the subscript operator.

可以从容器中检索单个元素的容器类一般会定义下标操作符，即 `operator[]`。标准库的类 `string` 和 `vector` 均是定义了下标操作符的类的例子。



The subscript operator must be defined as a class member function.

下标操作符必须定义为类成员函数。

Providing Read and Write Access

提供读写访问

One complication in defining the subscript operator is that we want it to do the right thing when used as either the left- or right-hand operand of an assignment. To appear on the left-hand side, it must yield an lvalue, which we can achieve by specifying the return type as a reference. As long as subscript returns a reference, it can be used on either side of an assignment.

定义下标操作符比较复杂的地方在于，它在用作赋值的左右操作符数时都应该能表现正常。下标操作符出现在左边，必须生成左值，可以指定引用作为返回类型而得到左值。只要下标操作符返回引用，就可用作赋值的任意一方。

It is also a good idea to be able to subscript `const` and non`const` objects. When applied to a `const` object, the return should be a `const` reference so that it is not usable as the target of an assignment.

可以对 `const` 和非 `const` 对象使用下标也是个好主意。应用于 `const` 对象时，返回值应为 `const` 引用，因此不能用作赋值的目标。



Ordinarily, a class that defines subscript needs to define two versions: one that is a non`const` member and returns a reference and one that is a `const` member and returns a `const` reference.

类定义下标操作符时，一般需要定义两个版本：一个为非 `const` 成员并返回引用，另一个为 `const` 成员并返回 `const` 引用。

Prototypical Subscript Operator

原型下标操作符

The following class defines the subscript operator. For simplicity, we assume the data `foo` holds are stored in a `vector<int>`:

下面的类定义了下标操作符。为简单起见，假定 `foo` 所保存的数据存储在一个 `vector<int>` 中：

```
class Foo {
public:
    int &operator[] (const size_t);
    const int &operator[] (const size_t) const;
    // other interface members
private:
    vector<int> data;
    // other member data and private utility functions
```

};

The subscript operators themselves would look something like:

下标操作符本身可能看起来像这样：

```
int& Foo::operator[] (const size_t index)
{
    return data[index]; // no range checking on index
}
const int& Foo::operator[] (const size_t index) const
{
    return data[index]; // no range checking on index
}
```

Exercises Section 14.5

Exercise 14.17: Define a subscript operator that returns a name from the waiting list for the `CheckoutRecord` class from the exercises to [Section 14.2.1](#) (p. 515).

[第 14.2.1 节](#)习题中定义了一个 `CheckoutRecord` 类，为该类定义一个下标操作符，从等待列表中返回一个名字。

Exercise 14.18: Discuss the pros and cons of implementing this operation using the subscript operator.

讨论用下标操作符实现这个操作的优缺点。

Exercise 14.19: Suggest alternative ways to define this operation.

提出另一种方法定义这个操作。

14.6. Member Access Operators

14.6. 成员访问操作符

To support pointerlike classes, such as iterators, the language allows the dereference (`*`) and arrow (`->`) operators to be overloaded.

为了支持指针型类，例如迭代器，C++ 语言允许重载解引用操作符 (`*`) 和箭头操作符 (`->`)。



Operator arrow must be defined as a class member function. The dereference operator is not required to be a member, but it is usually right to make it a member as well.

箭头操作符必须定义为类成员函数。解引用操作不要求定义为成员，但将它作为成员一般也是正确的。

Building a Safer Pointer

构建更安全的指针

The dereference and arrow operators are often used in classes that implement [smart pointers](#) ([Section 13.5.1](#), p. 495). As an example, let's assume that we want to define a class type to represent a pointer to an object of the `Screen` type that we wrote in [Chapter 12](#). We'll name this class `ScreenPtr`.

解引用操作符和箭头操作符常用在实现[智能指针](#) ([第 13.5.1 节](#)) 的类中。作为例子，假定想要定义一个类类型表示指向[第十二章](#) `Screen` 类型对象的指针，将该类命名为 `ScreenPtr`。

Our `ScreenPtr` class will be similar to our second `HasPtr` class. Users of `ScreenPtr` will be expected to pass a pointer to a dynamically allocated `Screen`. The `ScreenPtr` class will own that pointer and arrange to `delete` the underlying object when the last `ScreenPtr` referring to it goes away. In addition, we will not give our `ScreenPtr` class a default constructor. This way we'll know that a `ScreenPtr` object will always refer to a `Screen`. Unlike a built-in pointer, there will be no unbound `ScreenPtrs`. Applications can use `ScreenPtr` objects without first testing whether they refer to a `Screen` object.

`ScreenPtr` 类将类似于我们的第二个 `HasPtr` 类。`ScreenPtr` 的用户将会传递一个指针，该指针指向动态分配的 `Screen`，`ScreenPtr` 类将拥有该指针，并安排在指向基础对象的最后一个 `ScreenPtr` 消失时删除基础对象。另外，不用为 `ScreenPtr` 类定义默认构造函数。因此，我们知道一个 `ScreenPtr` 对象将总是指向一个 `Screen` 对象，不会有未绑定的 `ScreenPtr`，这一点与内置指针不同。应用程序可以使用 `ScreenPtr` 对象而无须首先测试它是否指向一个 `Screen` 对象。

As does the `HasPtr` class, the `ScreenPtr` class will use-count its pointer. We'll define a companion class to hold the pointer and its associated use count:

像 `HasPtr` 类一样，`ScreenPtr` 类将对其指针进行使用计数。我们将定义一个伙伴类保存指针及其相关使用计数：

```
// private class for use by ScreenPtr only
class ScrPtr {
    friend class ScreenPtr;
    Screen *sp;
    size_t use;
    ScrPtr(Screen *p): sp(p), use(1) { }
    ~ScrPtr() { delete sp; }
};
```

This class looks a lot like the `U_Ptr` class and has the same role. `ScrPtr` holds the pointer and associated use count. We make `ScreenPtr` a friend so that it can access the use count. The `ScreenPtr` class manages the use count:

这个类看来很像 `U_Ptr` 类并且作用同样。`ScrPtr` 保存指针及其相关使用计数。将 `ScreenPtr` 设为友元，以便 `ScreenPtr` 可以访问使用计数。`ScreenPtr` 类将管理使用计数：

```
/*
 * smart pointer: Users pass to a pointer to a dynamically allocated Screen, which
 *                 is automatically destroyed when the last ScreenPtr goes away
 */
class ScreenPtr {
public:
    // no default constructor: ScreenPtrs must be bound to an object
```

Section 14.6. Member Access Operators

```
ScreenPtr(Screen *p): ptr(new ScrPtr(p)) { }
// copy members and increment the use count
ScreenPtr(const ScreenPtr &orig):
    ptr(orig.ptr) { ++ptr->use; }
ScreenPtr& operator=(const ScreenPtr&);
// if use count goes to zero, delete the ScrPtr object
~ScreenPtr() { if (--ptr->use == 0) delete ptr; }
private:
    ScrPtr *ptr; // points to use-counted ScrPtr class
};
```

Because there is no default constructor, every object of type `ScreenPtr` must provide an initializer. The initializer must be another `ScreenPtr` or a pointer to a dynamically allocated `Screen`. The constructor allocates a new `ScrPtr` object to hold that pointer and an associated use count.

因为没有默认构造函数，所以 `ScreenPtr` 类型的每个对象必须提供一个初始化函数，初始化函数必须是另一个 `ScreenPtr` 对象或指向动态分配的 `Screen` 的指针。构造函数分配一个新的 `ScrPtr` 对象以保存那个指针及相关的使用计数。

An attempt to define a `ScreenPtr` with no initializer is in error:

试图定义一个不带初始化式的 `ScreenPtr` 对象是错误的：

```
ScreenPtr pl; // error: ScreenPtr has no default constructor
ScreenPtr ps(new Screen(4,4)); // ok: ps points to a copy of myScreen
```

Supporting Pointer Operations

支持指针操作

Among the fundamental operations a pointer supports are dereference and arrow. We can give our class these operations as follows:

指针支持的基本操作有解引用操作和箭头操作。我们的类可以这样定义这些操作：

```
class ScreenPtr {
public:
    // constructor and copy control members as before
    Screen &operator*() { return *ptr->sp; }
    Screen *operator->() { return ptr->sp; }
    const Screen &operator*() const { return *ptr->sp; }
    const Screen *operator->() const { return ptr->sp; }
private:
    ScrPtr *ptr; // points to use-counted ScrPtr class
};
```

Overloading the Dereference Operator

重载解引用操作符

The dereference operator is a unary operator. In this class, it is defined as a member so it has no explicit parameters. The operator returns a reference to the `Screen` to which this `ScreenPtr` points.

解引用操作符是个一元操作符。在这个类中，解引用操作符定义为成员，因此没有显式形参，该操作符返回对 `ScreenPtr` 所指向的 `Screen` 的引用。

As with the subscript operator, we need both `const` and `nonconst` versions of the dereference operator. These differ in their return types: The `const` member returns a reference to `const` to prevent users from changing the underlying object.

像下标操作符一样，我们需要解引用操作符的 `const` 和非 `const` 版本。它们的区别在于返回类型：`const` 成员返回 `const` 引用以防止用户改变基础对象。

Overloading the Arrow Operator

重载箭头操作符

Operator arrow is unusual. It may appear to be a binary operator that takes an object and a member name, dereferencing the object in order to fetch the member. Despite appearances, the arrow operator takes no explicit parameter.

Section 14.6. Member Access Operators

箭头操作符与众不同。它可能表现得像二元操作符一样：接受一个对象和一个成员名。对对象解引用以获取成员。不管外表如何，箭头操作符不接受显式形参。

There is no second parameter because the right-hand operand of `->` is not an expression. Rather, the right-hand operand is an identifier that corresponds to a member of a class. There is no obvious, useful way to pass an identifier as a parameter to a function. Instead, the compiler handles the work of fetching the member.

这里没有第二个形参，因为 `->` 的右操作数不是表达式，相反，是对应着类成员的一个标识符。没有明显可行的途径将一个标识符作为形参传递给函数，相反，由编译器处理获取成员的工作。

When we write

当这样编写时：

```
point->action();
```

precedence rules make it equivalent to writing

由于优先级规则，它实际等价于编写：

```
(point->action)();
```

In other words, we want to call the result of evaluating `point->action`. The compiler evaluates this code as follows:

换句话说，我们想要调用的是对 `point->action` 求值的结果。编译器这样对该代码进行求值：

1. If `point` is a pointer to a class object that has a member named `action`, then the compiler writes code to call the `action` member of that object.

如果 `point` 是一个指针，指向具有名为 `action` 的成员的类对象，则编译器将代码编译为调用该对象的 `action` 成员。

2. Otherwise, if `point` is an object of a class that defines `operator->`, then `point->action` is the same as `point.operator->()>action`. That is, we execute `operator->()` on `point` and then repeat these three steps, using the result of executing `operator->` on `point`.

否则，如果 `action` 是定义了 `operator->` 操作符的类的一个对象，则 `point->action` 与 `point.operator->()>action` 相同。即，执行 `point` 的 `operator->()`，然后使用该结果重复这三步。

3. Otherwise, the code is in error.

否则，代码出错。

Using Overloaded Arrow

使用重载箭头

We can use a `ScreenPtr` object to access members of a `Screen` as follows:

可以这样使用 `ScreenPtr` 对象访问 `Screen` 对象的成员：

```
ScreenPtr p(&myScreen);      // copies the underlying Screen
p->display(cout);
```

Because `p` is a `ScreenPtr`, the meaning of `p->display` is the same as evaluating `(p.operator->())>display`. Evaluating `p.operator->()` calls the `operator->` from class `ScreenPtr`, which returns a pointer to a `Screen` object. That pointer is used to fetch and run the `display` member of the object to which the `ScreenPtr` points.

因为 `p` 是一个 `ScreenPtr` 对象，`p->display` 的含义与对 `(p.operator->())>display` 求值相同。对 `p.operator->()` 求值将调用 `ScreenPtr` 类的 `operator->`，它返回指向 `Screen` 对象的指针，该指针用于获取并运行 `ScreenPtr` 所指对象的 `display` 成员。

Constraints on the Return from Overloaded Arrow

对重载箭头的返回值的约束

The overloaded arrow operator *must* return either a pointer to a class type or an object of a class type that defines its own operator arrow.

重载箭头操作符必须返回指向类类型的指针，或者返回定义了自己的箭头操作符的类类型对象。



If the return type is a pointer, then the built-in arrow operator is applied to that pointer. The compiler dereferences the pointer and fetches the indicated member from the resulting object. If the type pointed to does not define that member, then the compiler generates an error.

如果返回类型是指针，则内置箭头操作符可用于该指针，编译器对该指针解引用并从结果对象获取指定成员。如果被指向的类型没有定义那个成员，则编译器产生一个错误。

If the return value is another object of class type (or reference to such an object), then the operator is applied recursively. The compiler checks whether the type of the object returned has a member arrow and if so, applies that operator. Otherwise, the compiler generates an error. This process continues until either a pointer to an object with the indicated member is returned or some other value is returned, in which case the code is in error.

如果返回类型是类类型的其他对象（或是这种对象的引用），则将递归应用该操作符。编译器检查返回对象所属类型是否具有成员箭头，如果有，就应用那个操作符；否则，编译器产生一个错误。这个过程继续下去，直到返回一个指向带有指定成员的对象的指针，或者返回某些其他值，在后一种情况下，代码出错。

Exercises Section 14.6

Exercise 14.20: In our sketch for the `ScreenPtr` class, we declared but did not define the assignment operator.
14.20: Implement the `ScreenPtr` assignment operator.

在 `ScreenPtr` 类的概略定义中，声明但没有定义赋值操作符。请实现 `ScreenPtr` 赋值操作符。

Exercise 14.21: Define a class that holds a pointer to a `ScreenPtr`. Define the overloaded arrow operator for that class.

定义一个类，该类保存一个指向 `ScreenPtr` 的指针。为该类定义一个重载的箭头操作符。

Exercise 14.22: A smart pointer probably should define the equality and inequality operators to test whether two pointers are equal or unequal. Add these operations to the `ScreenPtr` class.

智能指针可能应该定义相等操作符和不等操作符，以便测试两个指针是否相等或不等。将这些操作加入到 `ScreenPtr` 类。

14.7. Increment and Decrement Operators

14.7. 自增操作符和自减操作符

The increment (`++`) and decrement (`--`) operators are most often implemented for classes, such as iterators, that provide pointer like behavior on the elements of a sequence. As an example, we might define a class that points to an array and provides checked access to elements in that array. Ideally, our checked-pointer class could be used on arrays of any type, which we'll learn how to do in [Chapter 16](#) when we cover class templates. For now, our class will handle arrays of `int`s:

```
/*
 * smart pointer: Checks access to elements throws an out_of_range
 *                 exception if attempt to access a nonexistent element
 * users allocate and free the array
 */
class CheckedPtr {
public:
    // no default constructor; CheckedPtrs must be bound to an object
    CheckedPtr(int *b, int *e): beg(b), end(e), curr(b) { }
    // dereference and increment operations
private:
    int* beg;    // pointer to beginning of the array
    int* end;    // one past the end of the array
    int* curr;   // current position within the array
};
```

Like `ScreenPtr`, this class has no default constructor. We must supply pointers to an array when we create a `CheckedPtr`. A `CheckedPtr` has three data members: `beg`, which points to the first element in the array; `end`, which points one past the end of the array; and `curr`, which points to the array element to which this `CheckedPtr` object currently refers.

像 `ScreenPtr` 一样，这个类没有默认构造函数。创建一个 `CheckedPtr` 对象时，必须提供指向数组的指针。一个 `CheckedPtr` 对象有三个数据成员：`beg`，指向数组的第一个元素；`end`，指向数组的末端；`curr`，指向 `CheckedPtr` 对象当前引用的数组元素。

The constructor takes two pointers: one pointing to the beginning of the array and the other one past the end of the array. The constructor initializes `beg` and `end` from these pointers and initializes `curr` to point to the first element.

构造函数的参数是两个指针：一个指向数组的开始，另一个指向数组的末端。构造函数用这两个指针初始化 `beg` 和 `end`，并将 `curr` 初始化为指向第一个元素。

Defining the Increment/Decrement Operators

定义自增／自减操作符



There is no language requirement that the increment or decrement operators be made members of the class. However, because these operators change the state of the object on which they operate, our preference is to make them members.

C++ 语言不要求自增操作符或自减操作符一定作为类的成员，但是，因为这些操作符改变操作对象的状态，所以更倾向于将它们作为成员。

Before we can define the overloaded increment and decrement operators for `CheckedPtr`, we must think about one more thing. For the built-in types, there are both prefix and postfix versions of the increment and decrement operators. Not surprisingly, we can define both the prefix and postfix instances of these operators for our own classes as well. We'll look at the prefix versions first and then implement the postfix ones.

在为类定义重载的自增操作符和自减操作符之前，还必须考虑另一件事情。对内置类型而言，自增操作符和自减操作符有前缀和后缀两种形式。毫不奇怪，也可以为我们自己的类定义自增操作符和自减操作符的前缀和后缀实例。我们首先介绍前缀形式，然后实现后缀形式。

Defining Prefix Increment/Decrement Operators

定义前自增／前自减操作符

The declarations for the prefix operators look as one might expect:

前缀式操作符的声明看起来像这样：

```
class CheckedPtr {
public:
    CheckedPtr& operator++();           // prefix operators
    CheckedPtr& operator--();
    // other members as before
};
```



For consistency with the built-in operators, the prefix operations should return a reference to the incremented or decremented object.

为了与内置类型一致，前缀式操作符应返回被增量或减量对象的引用。

This increment operator ensures that the user can't increment past the end of the array by checking `curr` against `end`. We throw an `out_of_range` exception if the increment would move `curr` past `end`; otherwise, we increment `curr` and return a reference to the object:

这个自增操作符根据 `end` 检查 `curr`，从而确保用户不能将 `curr` 增量到超过数组的末端。如果 `curr` 增量到超过 `end`，就抛出一个 `out_of_range` 异常；否则，将 `curr` 加 1 并返回对象引用：

```
// prefix: return reference to incremented/decremented object
CheckedPtr& CheckedPtr::operator++()
{
    if (curr == end)
        throw out_of_range
            ("increment past the end of CheckedPtr");
    ++curr;           // advance current state
    return *this;
}
```

The decrement operator behaves similarly, except that it decrements `curr` and checks whether the decrement would move `curr` past `beg`:

除了将 `curr` 减 1 并检查是否会减到 `beg`，自减操作符的行为与自增操作符类似：

```
CheckedPtr& CheckedPtr::operator--()
{
    if (curr == beg)
        throw out_of_range
            ("decrement past the beginning of CheckedPtr");
    --curr;           // move current state back one element
    return *this;
}
```

Differentiating Prefix and Postfix Operators

区别操作符的前缀和后缀形式

There is one problem with defining both the prefix and postfix operators: They each take the same number and type of parameters. Normal overloading cannot distinguish between whether the operator we're defining is the prefix version or the postfix.

同时定义前缀式操作符和后缀式操作符存在一个问题：它们的形参数目和类型相同，普通重载不能区别所定义的前缀式操作符还是后缀式操作符。

To solve this problem, the postfix operator functions take an extra (unused) parameter of type `int`. When we use the postfix operator, the compiler supplies 0 as the argument for this parameter. Although our postfix function could use this extra parameter, it usually should not. That parameter is not needed for the work normally performed by a postfix operator. Its sole purpose is to distinguish the definition of the postfix function from the prefix version.

为了解决这一问题，后缀式操作符函数接受一个额外的（即，无用的）`int` 型形参。使用后缀式操作符进，编译器提供 0 作为这个形参的实参。尽管我们的前缀式操作符函数

Section 14.7. Increment and Decrement Operators

可以使用这个额外的形参，但通常不应该这样做。那个形参不是后缀式操作符的正常工作所需要的，它的唯一目的是使后缀函数与前缀函数区别开来。

Defining the Postfix Operators

定义后缀式操作符

We can now add the postfix operators to `CheckedPtr`:

现在将后缀式操作符加到 `CheckedPtr`:

```
class CheckedPtr {
public:
    // increment and decrement
    CheckedPtr operator++(int);           // postfix operators
    CheckedPtr operator--(int);
    // other members as before
};
```



For consistency with the built-in operators, the postfix operators should return the old (unincremented or undecremented) value. That value is returned as a value, not a reference.

为了与内置操作符一致，后缀式操作符应返回旧值（即，尚未自增或自减的值），并且，应作为值返回，而不是返回引用。

The postfix operators might be implemented as follows:

后缀式操作符可以这样实现：

```
// postfix: increment/decrement object but return unchanged value
CheckedPtr CheckedPtr::operator++(int)
{
    // no check needed here, the call to prefix increment will do the check
    CheckedPtr ret(*this);           // save current value
    ++*this;                      // advance one element, checking the increment
    return ret;                    // return saved state
}
CheckedPtr CheckedPtr::operator--(int)
{
    // no check needed here, the call to prefix decrement will do the check
    CheckedPtr ret(*this);          // save current value
    --*this;                      // move backward one element and check
    return ret;                    // return saved state
}
```

The postfix versions are a bit more involved than the prefix operators. They have to remember the current state of the object before incrementing the object. These operators define a local `CheckedPtr`, which is initialized as a copy of `*this` that is, `ret` is a copy of the current state of this object.

操作符的后缀式比前缀式复杂一点，必须记住对象在加 1 / 减 1 之前的当前状态。这些操作符定义了一个局部 `CheckedPtr` 对象，将它初始化为 `*this` 的副本，即 `ret` 是这个对象当前状态的副本。

Having kept a copy of the current state, the operator calls its own prefix operator to do the increment or decrement, respectively:

保存了当前状态的副本后，操作符调用自己的前缀式操作符分别进行加 1 或减 1：

```
++*this
```

calls the `CheckedPtr` prefix increment operator on this object. That operator checks that the increment is safe and either increments `curr` or throws an exception. Assuming no exception was thrown, the postfix function completes by returning the stored copy in `ret`. Thus, after the return, the object itself has been advanced, but the value returned reflects the original, unincremented value.

调用这个对象的 `CheckedPtr` 前缀自增操作符，该操作符检查自增是否安全并将 `curr` 加 1 或抛出一个异常。假定不抛出异常，前自增操作符函数以返回存储在 `ret` 的副本而结束。因此，返回之后，对象本身加了 1，但返回的是尚未自增的原值。

Because these operators are implemented by calling the prefix versions, there is no need to check that the `curr` is in range. That check, and the

Section 14.7. Increment and Decrement Operators

`throw` if necessary, is done inside the corresponding prefix operator.

因为通过调用前缀式版本实现这些操作符，不需要检查 `curr` 是否在范围之内，那个检查以及必要的 `throw`，在相应的前缀式操作符中完成。



The `int` parameter is not used, so we do not give it a name.

因为不使用 `int` 形参，所以没有对其命名。

Calling the Postfix Operators Explicitly

显式调用前缀式操作符

As we saw on page 509, we can explicitly call an overloaded operator rather than using it as an operator in an expression. If we want to call the postfix version using a function call, then we must pass a value for the integer argument:

正如在第 14.1 节所见，可以显式调用重载操作符而不是将它作为操作符用在表达式中。如果想要使用函数调用来调用后缀式操作符，必须给出一个整型实参值：

```
CheckedPtr parr(ia, ia + size);           // ia points to an array of ints  
parr.operator++(0);                      // call postfix operator++  
parr.operator++();                      // call prefix operator++
```

The value passed usually is ignored but is necessary to alert the compiler that the postfix version is desired.

所传递的值通常被忽略，但该值是必要的，用于通知编译器需要的是后缀式版本。



Ordinarily it is best to define both the prefix and postfix versions. Classes that define only the prefix version or only the postfix version will surprise users who are accustomed to being able to use either form.

一般而言，最好前缀式和后缀式都定义。只定义前缀式或只定义后缀式的类，将会让习惯于使用两种形式的用户感到奇怪。

Exercises Section 14.7

Exercise

14.23:

The class `CheckedPtr` represents a pointer that points to an array of `ints`. Define an overloaded subscript and dereference for this class. Have the operator ensure that the `CheckedPtr` is valid: It should not be possible to dereference or index one past the end of the array.

`CheckedPtr` 类表示指向数组的指针。为该类重载下标操作符和解引用操作符。使操作符确保 `CheckedPtr` 有效：它应该不可能对超出数组末端的元素进行解引用或索引。

Exercise

14.24:

Should the dereference or subscript operators defined in the previous exercise also check whether an attempt is being made to dereference or index one before the beginning of the array? If not, why not? If so, why?

上题中定义的解引用操作符或下标操作符，是否也应该检查对数组起点之前的元素进行的解引用或索引？解释你的答案。

Exercise

14.25:

To behave like a pointer to an array, our `CheckedPtr` class should implement the equality and relational operators to determine whether two `CheckedPtrs` are equal, or whether one is less-than another, and so on. Add these operations to the `CheckedPtr` class.

为了表现得像数组指针，`CheckedPtr` 类应实现相等和关系操作符，以便确定两个 `CheckedPtr` 对象是否相等，或者一个小于另一个，诸如此类。为 `CheckedPtr` 类增加这些操作。

Section 14.7. Increment and Decrement Operators

Exercise 14.26: Define addition and subtraction for `ScreenPtr` so that these operators implement pointer arithmetic ([Section 4.2.4](#), p. 123).

为 `ScreenPtr` 类定义加法或减法，以便这些操作符实现指针运算 ([第 4.2.4 节](#))。

Exercise 14.27: Discuss the pros and cons of allowing an empty array argument to the `CheckedPtr` constructor.

讨论允许将空数组实参传给 `CheckedPtr` 构造函数的优缺点。

Exercise 14.28: We did not define a `const` version of the increment and decrement operators. Why?

没有定义自增和自减操作符的 `const` 版本，为什么？

Exercise 14.29: We also didn't implement arrow. Why?

我们也没有实现箭头操作符，为什么？

Exercise 14.30: Define a version of `CheckedPtr` that holds an array of `Screens`. Implement the overloaded increment, decrement, dereference, and arrow operators for this class.

定义一个 `CheckedPtr` 版本，保存 `Screen` 数组。为该类实现重载的自增、自减、解引用、箭头等操作符。

14.8. Call Operator and Function Objects

14.8. 调用操作符和函数对象

The function-call operator can be overloaded for objects of class type. Typically, the call operator is overloaded for classes that represent an operation. For example, we could define a struct named `absInt` that encapsulates the operation of converting a value of type `int` to its absolute value:

可以为类类型的对象重载函数调用操作符。一般为表示操作的类重载调用操作符。例如，可以定义名为 `absInt` 的结构，该结构封装将 `int` 类型的值转换为绝对值的操作：

```
struct absInt {
    int operator() (int val) {
        return val < 0 ? -val : val;
    }
};
```

This class is simple. It defines a single operation: the function-call operator. That operator takes a single parameter and returns the absolute value of its parameter.

这个类很简单，它定义了一个操作：函数调用操作符，该操作符有一个形参并返回形参的绝对值。

We use the call operator by applying an argument list to an object of the class type, in a way that looks like a function call:

通过为类类型的对象提供一个实参表而使用调用操作符，所用的方式看起来像一个函数调用：

```
int i = -42;
absInt absObj; // object that defines function call operator
unsigned int ui = absObj(i); // calls absInt::operator(int)
```

Even though `absObj` is an object and not a function, we can make a "call" on that object. The effect is to run the overloaded call operator defined by the object `absObj`. That operator takes an `int` value and returns its absolute value.

尽管 `absObj` 是一个对象而不是函数，我们仍然可以“调用”该对象，效果是运行由 `absObj` 对象定义的重载调用操作符，该操作符接受一个 `int` 并值并返回它的绝对值。



The function-call operator must be declared as a member function. A class may define multiple versions of the call operator, each of which differs as to the number or types of their parameters.

函数调用操作符必须声明为成员函数。一个类可以定义函数调用操作符的多个版本，由形参的数目或类型加以区别。

Objects of class types that define the call operator are often referred to as function objects; that is, they are objects that act like functions.

定义了调用操作符的类，其对象常称为函数对象，即它们是行为类似函数的对象。

Exercises Section 14.8

Exercise 14.31: Define a function object to perform an if-then-else operation: The function object should take three parameters. It should test its first parameter and if that test succeeds, it should return its second parameter, otherwise, it should return its third parameter.

定义一个函数对象执行“如果—则—否则”操作：该函数对象应接受三个形参，它应该测试第一个形参，如果测试成功，就返回第二个形参，否则，就返回第三个形参。

Exercise 14.32: How many operands may an overloaded function-call operator take?

一个重载的函数调用操作符可以接受多少操作数？

14.8.1. Using Function Objects with Library Algorithms

14.8.1. 将函数对象用于标准库算法

Function objects are most often used as arguments to the generic algorithms. As an example, recall the problem we solved in [Section 11.2.3](#) (p. 400). That program analyzed words in a set of stories, counting how many of them were of size six or greater. One part of that solution involved defining a function to determine whether a given `string` was longer than six characters in length:

函数对象经常用作通用算法的实参。在[第 11.2.3 节](#)解决的问题就是这样一个例子。那个程序分析一组故事中的单词，计算有多少个单词长度在 6 字符以上。该解决方案的一个部分包括定义一个函数以确定给定 `string` 的长度是否大于 6 字符：

```
// determine whether a length of a given word is 6 or more
bool GT6(const string &s)
{
    return s.size() >= 6;
}
```

We used `GT6` as an argument to the `count_if` algorithm to count the number of words for which `GT6` returned true:

使用 `GT6` 作为传给 `count_if` 算法的实参，以计算使 `GT6` 返回 true 的单词的数目：

```
vector<string>::size_type wc =
    count_if(words.begin(), words.end(), GT6);
```

Function Objects Can Be More Flexible than Functions

函数对象可以比函数更灵活

There was a serious problem with our implementation: It hardwired the number six into the definition of the `GT6` function. The `count_if` algorithm runs a function that takes a single parameter and returns a `bool`. Ideally, we'd pass both the `string` and the size we wanted to test. In that way, we could use the same code to count strings of differing sizes.

我们的实现有个严重问题：它将 6 这个数字固化在 `GT6` 函数的定义中。`count_if` 算法运行只用一个形参且返回 `bool` 的函数。理想情况下，应传递 `string` 和我们想要的长度进行测试。通过该方式，可以使用同一代码对不同长度的字符串进行计数。

We could gain the flexibility we want by defining `GT6` as a class with a function-call member. We'll name this class `GT_cls` to distinguish it from the function:

通过将 `GT6` 定义为带函数调用成员类，可以获得所需的灵活性。将这个类命名为 `GT_cls` 以区别于函数：

```
// determine whether a length of a given word is longer than a stored bound
class GT_cls {
public:
    GT_cls(size_t val = 0): bound(val) { }
    bool operator()(const string &s)
        { return s.size() >= bound; }
private:
    std::string::size_type bound;
};
```

This class has a constructor that takes an integral value and remembers that value in its member named `bound`. If no value is provided, the constructor sets `bound` to zero. The class also defines the call operator, which takes a `string` and returns a `bool`. That operator compares the length of its `string` argument to the value stored in its data member `bound`.

这个类有一个构造函数，该构造函数接受一个整型值并用名为 `bound` 的成员记住那个值。如果没有提供值，构造函数将 `bound` 置 0。该类也定义了调用操作符，接受一个 `string` 参数并返回一个 `bool`。调用操作符将 `string` 实参的长度与数据成员 `bound` 中存储的值相比较。

Using a `GT_cls` Function Object

使用 `GT_cls` 函数对象

We can do the same count as before but this time we'll use an object of type `GT_cls` rather than the `GT6` function:

Section 14.8. Call Operator and Function Objects

可以像前面一样进行计数，但这一次使用 `GT_cls` 类型的对象而不是 `GT6` 函数：

```
cout << count_if(words.begin(), words.end(), GT_cls(6))
<< " words 6 characters or longer" << endl;
```

This call to `count_if` passes a temporary object of type `GT_cls` rather than the function named `GT6`. We initialize that temporary using the value `6`, which the `GT_cls` constructor stores in its `bound` member. Now, each time `count_if` calls its function parameter, it uses the call operator from `GT_cls`. That call operator tests the size of its `string` argument against the value in `bound`.

这个 `count_if` 调用传递一个 `GT_cls` 类型的临时对象而不再是名为 `GT6` 的函数。用整型值 `6` 来初始化那个临时对象，构造函数将这个值存储在 `bound` 成员中。现在，`count_if` 每次调用它的函数形参时，它都使用 `GT_cls` 的调用操作符，该调用操作符根据 `bound` 的值测试其 `string` 实参的长度。

Using the function object, we can easily revise our program to test against another value. We need to change only the argument to the constructor for the object we pass to `count_if`. For example, we could count the number of words of length five or greater by revising our program as follows:

使用函数对象，容易修改程序以根据其他值进行测试，只需为传给 `count_if` 的对象改变构造函数实参即可。例如，这样修改程序，就可以计算长度在 5 个字符以上的单词数：

```
cout << count_if(words.begin(), words.end(), GT_cls(5))
<< " words 5 characters or longer" << endl;
```

More usefully, we could count the number of words with lengths greater than one through ten:

更为有用的是，还可以计算长度在 1 到 10 个字符的单词数：

```
for (size_t i = 0; i != 11; ++i)
cout << count_if(words.begin(), words.end(), GT(i))
<< " words " << i
<< " characters or longer" << endl;
```

To write this program using a function instead of a function object would require that we write ten different functions, each of which would test against a different value.

如果使用函数代替函数对象来编写这个程序，可能需要编写 10 个不同的函数，每个函数测试一个不同的值。

Exercises Section 14.8.1

Exercise

14.33:

Using the library algorithms and the `GT_cls` class, write a program to find the first element in a sequence that is larger than a specified value.

使用标准库算法和 `GT_cls` 类，编写一个程序查找序列中第一个比指定值大的元素。

Exercise

14.34:

Write a function-object class similar to `GT_cls` but that tests whether two values are equal. Use that object and the library algorithms to write a program to replace all instances of a given value in a sequence.

编写类似于 `GT_cls` 的函数对象类，但测试两个值是否相等。使用该对象和标准库算法编写程序，替换序列中给定值的所有实例。

Exercise

14.35:

Write a class similar to `GT_cls`, but that tests whether the length of a given `string` matches its `bound`. Use that object to rewrite the program in [Section 11.2.3](#) (p. 400) to report how many words in the input are of sizes 1 through 10 inclusive.

编写类似于 `GT_cls` 的类，但测试给定的长度是否与其边界相匹配。使用该对象重写 [第 11.2.3 节](#) 中的程序，以便报告输入中有多少单词的长度在 1 到 10 之间（含 1 和 10）。

Exercise

14.36:

Revise the previous program to report the count of words that are sizes 1 through 9 and 10 or more.

修改前面程序以报告长度在 1 到 9 之间以及 10 以上的单词的数目。

14.8.2. Library-Defined Function Objects

14.8.2. 标准库定义的函数对象

The standard library defines a set of arithmetic, relational, and logical function-object classes, which are listed in [Table 14.3](#) on the following page. The library also defines a set of function adaptors that allow us to specialize or extend the function-object classes defined by the library or those that we define ourselves. The library function-object types are defined in the `functional` header.

标准库定义了一组算术、关系与逻辑函数对象类，[表 14.3](#) 列出了这些类。标准库还定义了一组函数适配器，使我们能够特化或者扩展标准库所定义的以及自定义的函数对象类。这些标准库函数对象类型是在 `functional` 头文件中定义的。

Table 14.3. Library Arithmetic Function Objects

表 14.3. 标准库函数对象

Arithmetic Function Objects Types

算术函数对象类型

<code>plus<Type></code>	<i>applies +</i>
<code>minus<Type></code>	<i>applies -</i>
<code>multiplies<Type></code>	<i>applies *</i>
<code>divides<Type></code>	<i>applies /</i>
<code>modulus<Type></code>	<i>applies %</i>
<code>negate<Type></code>	<i>applies -</i>

Relational Function Objects Types

关系函数对象类型

<code>equal_to<Type></code>	<i>applies ==</i>
<code>not_equal_to<Type></code>	<i>applies !=</i>
<code>greater<Type></code>	<i>applies ></i>
<code>greater_equal<Type></code>	<i>applies >=</i>
<code>less<Type></code>	<i>applies <</i>
<code>less_equal<Type></code>	<i>applies <=</i>

Logical Function Object Types

逻辑函数对象类型

<code>logical_and<Type></code>	<i>applies &&</i>
<code>logical_or<Type></code>	<i>applies </i>
<code>logical_not<Type></code>	<i>applies !</i>

Each Class Represents a Given Operator

每个类表示一个给定操作符

Each of the library function-object classes represents an operator that is, each class defines the call operator that applies the named operation. For example, `plus` is a template type that represents the addition operator. The call operator in the `plus` template applies `+` to a pair of operands.

每个标准库函数对象类表示一个操作符，即，每个类都定义了应用命名操作的调用操作符。例如，`plus` 是表示加法操作符的模板类型。`plus` 模板中的调用操作符对一对操作数应用 `+` 运算。

Different function-object classes define call operators that perform different operations. Just as `plus` defines a call operator that executes the `+`

Section 14.8. Call Operator and Function Objects

operator; the `modulus` class defines a call operator that applies the binary `%` operator; the `equal_to` class applies `==`; and so on.

不同的函数对象定义了执行不同操作的调用操作符。正如 `plus` 定义了执行 `+` 操作符的调用操作符, `modulus` 类定义了应用二元操作符 `%` 的调用操作符, `equal_to` 类应用 `==`, 等等。

There are two **unary function-object** classes: unary minus (`negate<Type>`) and logical NOT (`logical_not<Type>`). The remaining library function objects are **binary function-object** classes representing the binary operators. The call operators defined for the binary operators expect two parameters of the given type; the unary function-object types define a call operator that takes a single argument.

有两个**一元函数对象类**: 一元减 (`negate<Type>`) 和逻辑非 (`logical_not<Type>`)。其余的标准库函数对象都是表示二元操作符的**二元函数对象类**。为二元操作符定义的调用操作符需要两个给定类型的形参, 而一元函数对象类型定义了接受一个实参的调用操作符。

The Template Type Represents the Operand(s) Type

表示操作数类型的模板类型

Each of the function-object classes is a class template to which we supply a single type. As we know from the sequential containers such as `vector`, a class template is a class that can be used on a variety of types. The template type for the function-object classes specifies the parameter type for the call operator.

每个函数对象类都是一个类模板, 我们需要为该模板提供一个类型。正如从诸如 `vector` 的顺序容器所了解的, 类模板是可以用于不同类型的类。函数对象类的模板类型指定调用操作符的形参类型。

For example, `plus<string>` applies the `string` addition operator to `string` objects; for `plus<int>` the operands are `int`s; `plus<Sales_item>` applies `+` to `Sales_items`; and so on:

例如, `plus<string>` 将 `string` 加法操作符应用于 `string` 对象, 对于 `plus<int>`, 操作数是 `int` 值, `plus<Sales_item>` 将 `+` 应用于 `Sales_items`; 对象, 依次类推:

```
plus<int> intAdd;          // function object that can add two int values
negate<int> intNegate;    // function object that can negate an int value
// uses intAdd::operator(int, int) to add 10 and 20
int sum = intAdd(10, 20);      // sum = 30
// uses intNegate::operator(int) to generate -10 as second parameter
// to intAdd::operator(int, int)
sum = intAdd(10, intNegate(10)); // sum = 0
```

Using a Library Function Object with the Algorithms

在算法中使用标准库函数

Function objects are often used to override the default operator used by an algorithm. For example, by default, `sort` uses `operator<` to sort a container in ascending order. To sort the container in descending order, we could pass the function object `greater`. That class generates a call operator that invokes the greater-than operator of the underlying element type. If `svec` is a `vector<string>`

函数对象常用于覆盖算法使用的默认操作符。例如, `sort` 默认使用 `operator<` 按升序对容器进行排序。为了按降序对容器进行排序, 可以传递函数对象 `greater`。该类将产生一个调用操作符, 调用基础对象的大于操作符。如果 `svec` 是一个 `vector<string>` 对象, 以下代码

```
// passes temporary function object that applies > operator to two strings
sort(svec.begin(), svec.end(), greater<string>());
```

sorts the `vector` in descending order. As usual, we pass a pair of iterators to denote the sequence that should be sorted. The third argument is used to pass a predicate ([Section 11.2.3](#), p. 402) function to use to compare elements. That argument is a temporary of type `greater<string>`, which is a function object that applies the `>` operator to two `string` operands.

将按降序对 `vector` 进行排序。像通常那样, 传递一对迭代器以指明被排序序列。第三个实参用于传递比较元素的谓词 ([第 11.2.3 节](#)) 函数。该实参 `greater<string>` 类型的临时对象, 是一个将 `>` 操作符应用于两个 `string` 操作符的函数对象。

14.8.3. Function Adaptors for Function Objects

14.8.3. 函数对象的函数适配器

The standard library provides a set of **function adaptors** with which to specialize and extend both unary and binary function objects. The function

Section 14.8. Call Operator and Function Objects

adaptors are divided into the following two categories.

标准库提供了一组函数适配器，用于特化和扩展一元和二元函数对象。函数适配器分为如下两类：

- Binders: A **binder** is a function adaptor that converts a binary function object into a unary function object by binding one of the operands to a given value.

绑定器，是一种函数适配器，它通过将一个操作数绑定到给定值而将二元函数对象转换为一元函数对象。

- Negators: A **negator** is a function adaptor that reverses the truth value of a predicate function object.

求反器，是一种函数适配器，它将谓词函数对象的真值求反。

The library defines two binder adaptors: `bind1st` and `bind2nd`. Each binder takes a function object and a value. As you might expect, `bind1st` binds the given value to the first argument of the binary function object, and `bind2nd` binds the value to the second. For example, to count all the elements within a container that are less than or equal to 10, we would pass `count_if` the following:

标准库定义了两个绑定器适配器：`bind1st` 和 `bind2nd`。每个绑定器接受一个函数对象和一个值。正如你可能想到的，`bind1st` 将给定值绑定到二元函数对象的第一个实参，`bind2nd` 将给定值绑定到二元函数对象的第二个实参。例如，为了计算一个容器中所有小于或等于 10 的元素的个数，可以这样给 `count_if` 传递值：

```
count_if(vec.begin(), vec.end(),
         bind2nd(less_equal<int>(), 10));
```

The third argument to `count_if` uses the `bind2nd` function adaptor. That adaptor returns a function object that applies the `<=` operator using 10 as the right-hand operand. This call to `count_if` counts the number of elements in the input range that are less than or equal to 10.

传给 `count_if` 的第三个实参使用 `bind2nd` 函数适配器，该适配器返回一个函数对象，该对象用 10 作右操作数应用 `<=` 操作符。这个 `count_if` 调用计算输入范围中小于或等于 10 的元素的个数。

The library also provides two negators: `not1` and `not2`. Again, as you might expect, `not1` reverses the truth value of a unary predicate function object, and `not2` reverses the truth value of a binary predicate function object.

标准库还定义了两个求反器：`not1` 和 `not2`。你可能已经想到的，`not1` 将一元函数对象的真值求反，`not2` 将二元函数对象的真值求反。

To negate our binding of the `less_equal` function object, we would write

为了对 `less_equal` 函数对象的绑定求反，可以编写这样的代码：

```
count_if(vec.begin(), vec.end(),
         not1(bind2nd(less_equal<int>(), 10)));
```

Here we first bind the second operand of the `less_equal` object to 10, effectively transforming that binary operation into a unary operation. We then negate the return from the operation using `not1`. The effect is that each element will be tested to see if it is `<=` to 10. Then, the truth value of that result will be negated. In effect, this call counts those elements that are not `<=` to 10.

这里，首先将 `less_equal` 对象的第二个操作数绑定到 10，实际上是将该二元操作转换为一元操作。再用 `not1` 对操作的返回值求反，效果是测试每个元素是否 `<=`。然后，对结果真值求反。这个 `count_if` 调用的效果是对不 `<=` 10 的那些元素进行计数。

Exercises Section 14.8.3

Exercise Using the library function objects and adaptors, define an object to:
14.37:

使用标准库函数对象和函数适配器，定义一个对象用于：

- Find all values that are greater than 1024.

查找大于 1024 的所有值。

- Find all strings that are not equal to `pooh`.

查找不等于 `pooh` 的所有字符串。

- Multiply all values by 2.

将所有值乘以 2。

Exercise In the last call to `count_if` we used `not1` to negate the result from `bind2nd` of
14.38: (`less_equal<int>()`, 10). Why did we use `not1` rather than `not2`?

最后一个 `count_if` 调用中，用 `not1` 将 `bind2nd` 的结果求反。为什么使用 `not1` 而不用 `not2`？

Exercise Use library function objects in place of `GT_cls` to find the words of a specified length.

14.39:

使用标准库函数对象代替 `GT_cls` 来查找指定长度的单词。

Team LiB

◀ PREVIOUS

NEXT ▶

14.9. Conversions and Class Types

14.9. 转换与类类型

In [Section 12.4.4](#) (p. 461) we saw that a non-explicit constructor that can be called with one argument defines an implicit conversion. The compiler will use that conversion when an object of the argument type is supplied and an object of the class type is needed. Such constructors define conversions to the class type.

在第 [12.4.4 节](#) 介绍过，可用一个实参调用的非 `explicit` 构造函数定义一个隐式转换。当提供了实参类型的对象而需要一个类类型的对象时，编译器将使用该转换。这种构造函数定义了到类类型的转换。

In addition to defining conversions *to* a class type, we can also define conversions *from* the class type. That is, we can define a conversion operator that, given an object of the class type, will generate an object of another type. As with other conversions, the compiler will apply this conversion automatically. Before showing how to define such conversions, we'll look at why they might be useful.

除了定义到类类型的转换之外，我们还可以定义从类类型的转换。即，我们可以定义转换操作符，给定类类型的对象，该操作符将产生其他类型的对象。像其他转换一样，编译器将自动应用这个转换。在介绍如何定义这种转换之前，将说明它们为什么可能有用。

14.9.1. Why Conversions Are Useful

14.9.1. 转换为什么有用

Assume that we want to define a class, which we'll name `SmallInt`, to implement safe small integers. Our class will allow us to define objects that could hold the same range of values as an 8-bit unsigned `char` that is, 0 to 255. This class would catch under- and overflow errors and so would be safer to use than a built-in `unsigned char`.

假定想要定义一个名为 `SmallInt` 的类，该类实现安全小整数，这个类将使我们能够定义对象以保存与 8 位 `unsigned char` 同样范围的值，即，0 到 255。这个类可以捕获下溢和上溢错误，因此使用起来比内置 `unsigned char` 更安全。

We'd want our class to define all the same operations as are supported by an `unsigned char`. In particular, we'd want to define the five arithmetic operators (`+`, `-`, `*`, `/`, and `%`) and the corresponding compound-assignment operators; the four relational operators (`<`, `<=`, `>`, and `>=`); and the equality operators (`==` and `!=`). Evidently, we'd need to define 16 operators.

我们希望这个类定义 `unsigned char` 支持的所有操作。具体而言，我们想定义 5 个算术操作符 (`+`, `-`, `*`, `/`, `%`) 及其对应的复合赋值操作符，4 个关系操作符 (`<`, `<=`, `>`, `>=`)，以及相等操作符 (`==`, `!=`)。显然，需要定义 16 个操作符。

Supporting Mixed-Type Expressions

支持混合类型表达式

Moreover, we'd like to be able to use these operators in mixed-mode expressions. For example, it should be possible to add two `SmallInt` objects and also possible to add any of the arithmetic types to a `SmallInt`. We could come close by defining three instances for each operator:

而且，我们希望可以在混合模式表达式中使用这些操作符。例如，应该可以将两个 `SmallInt` 对象相加，也可以将任意算术类型加到 `SmallInt`。通过为每个操作符定义三个实例来达到目标：

```
int operator+(int, const SmallInt&);
int operator+(const SmallInt&, int);
SmallInt operator+(const SmallInt&, const SmallInt&);
```

Because there is a conversion to `int` from any of the arithmetic types, these three functions would cover our desire to support mixed mode use of `SmallInt` objects. However, this design only approximates the behavior of built-in integer arithmetic. It wouldn't properly handle mixed-mode operations for the floating-point types, nor would it properly support addition of `long`, `unsigned int`, or `unsigned long`. The problem is that this design converts all arithmetic types even those bigger than `int` to `int` and does an `int` addition.

因为存在从任意算术类型到 `int` 的转换，这三个函数可以涵盖支持 `SmallInt` 对象的混合模式使用的要求。但是，这个设计仅仅接近内置整数运算的行为，它不能适当处理浮点类型混合模式操作，也不能适当支持 `long`、`unsigned int` 或 `unsigned long` 的加运算。问题在于这个设计将所有算术类型（甚至包括那些比 `int` 大的）转换为 `int` 并进行 `int` 加运算。

Conversions Reduce the Number of Needed Operators

转换减少所需操作符的数目

Even ignoring the issue of floating-point or large integral operands, if we implemented this design, we'd have to define 48 operators! Fortunately, C++ provides a mechanism by which a class can define its own conversions that can be applied to objects of its class type. For `SmallInt`, we could define a conversion from `SmallInt` to type `int`. If we define the conversion, then we won't need to define any of the arithmetic, relational, or equality operators. Given a conversion to `int`, a `SmallInt` object could be used anywhere an `int` could be used.

即使忽略浮点或大整型操作数的问题，如果要实现这个设计，也必须定义 48 个操作符！幸好，C++ 提供了一种机制，利用这种机制，一个类可以定义自己的转换，应用于其类型对象。对 `SmallInt` 而言，可以定义一个从 `SmallInt` 到 `int` 类型的转换。如果定义了该转换，则无须再定义任何算术、关系或相等操作符。给定到 `int` 的转换，`SmallInt` 对象可以用在任何可用 `int` 值的地方。

If there were a conversion to `int`, then

如果存在一个到 `int` 的转换，则以下代码：

```
SmallInt si(3);
si + 3.14159;           // convert si to int, then convert to double
```

would be resolved by

可这样确定：

1. Converting `si` to an `int`.

将 `si` 转换为 `int` 值。

2. Converting the resulting `int` to `double` and adding it to the double literal constant `3.14159`, yielding a `double` value.

将所得 `int` 结果转换为 `double` 值并与双精度字面值常量 `3.14159` 相加，得到 `double` 值。

14.9.2. Conversion Operators

14.9.2. 转换操作符

A **conversion operator** is a special kind of class member function. It defines a conversion that converts a value of a class type to a value of some other type. A conversion operator is declared in the class body by specifying the keyword `operator` followed by the type that is the target type of the conversion:

转换操作符是一种特殊的类成员函数。它定义将类类型值转变为其他类型值的转换。转换操作符在类定义体内声明，在保留字 `operator` 之后跟着转换的目标类型：

```
class SmallInt {
public:
    SmallInt(int i = 0): val(i)
    { if (i < 0 || i > 255)
        throw std::out_of_range("Bad SmallInt initializer");
    }
    operator int() const { return val; }
private:
    std::size_t val;
};
```

A conversion function takes the general form

转换函数采用如下通用形式：

```
operator type();
```

where `type` represents the name of a built-in type, a class type, or a name defined by a `typedef`. Conversion functions can be defined for any type (other than `void`) that could be a function return type. In particular, conversions to an array or function type are not permitted. Conversions to pointer types both data and function pointers and to reference types are allowed.

这里，`type` 表示内置类型名、类类型名或由类型别名定义的名字。对任何可作为函数返回类型的类型（除了 `void` 之外）都可以定义转换函数。一般而言，不允许转换为数组或函数类型，转换为指针类型（数据和函数指针）以及引用类型是可以的。

A conversion function must be a member function. The function may not specify a return type, and the parameter list must be empty.



转换函数必须是成员函数，不能指定返回类型，并且形参表必须为空。

All of the following declarations are errors:

下述所有声明都是错误的：

```
operator int(SmallInt &);           // error: nonmember
class SmallInt {
public:
    int operator int();             // error: return type
    operator int(int = 0);         // error: parameter list
    // ...
};
```

Although a conversion function does not specify a return type, each conversion function must explicitly return a value of the named type. For example, `operator int` returns an `int`; if we defined an `operator Sales_item`, it would return a `Sales_item`; and so on.

虽然转换函数不能指定返回类型，但是每个转换函数必须显式返回一个指定类型的值。例如，`operator int` 返回一个 `int` 值；如果定义 `operator Sales_item`，它将返回一个 `Sales_item` 对象，诸如此类。



Conversion operations ordinarily should not change the object they are converting. As a result, conversion operators usually should be defined as `const` members.

转换函数一般不应该改变被转换的对象。因此，转换操作符通常应定义为 `const` 成员。

Using a Class-Type Conversion

使用类类型转换

Once a conversion exists, the compiler will call it automatically ([Section 5.12.1](#), p. 179) in the same places that a built-in conversion would be used:

只要存在转换，编译器将在可以使用内置转换的地方自动调用它 ([第 5.12.1 节](#))：

- In expressions:

在表达式中：

```
SmallInt si;
double dval;
si >= dval      // si converted to int and then convert to double
```

- In conditions:

在条件中：

```
if (si)          // si converted to int and then convert to bool
```

- When passing arguments to or returning values from a function:

将实参传给函数或从函数返回值：

Section 14.9. Conversions and Class Types

```
int calc(int);
SmallInt si;
int i = calc(si); // convert si to int and call calc
```

- As operands to overloaded operators:

作为重载操作符的操作数:

```
// convert si to int then call operator<< on the int value
cout << si << endl;
```

- In an explicit cast:

在显式类型转换中:

```
int ival;
SmallInt si = 3.541; //
instruct compiler to cast si to int
ival = static_cast<int>(si) + 3;
```

Class-Type Conversions and Standard Conversions

类类型转换和标准转换

When using a conversion function, the converted type need not exactly match the needed type. A class-type conversion can be followed by a standard conversion ([Section 5.12.3](#), p. 181) if needed to obtain the desired type. For example, in the comparison between a `SmallInt` and a `double`

使用转换函数时，被转换的类型不必与所需要的类型完全匹配。必要时可在类类型转换之后跟上标准转换以获得想要的类型。例如，在一个 `SmallInt` 对象与一个 `double` 值的比较中：

```
SmallInt si;
double dval;
si >= dval // si converted to int and then convert to double
```

`si` is first converted from a `SmallInt` to an `int`, and then the `int` value is converted to `double`.

首先将 `si` 从 `SmallInt` 对象转换为 `int` 值，然后将该 `int` 值转换为 `double` 值。

Only One Class-Type Conversion May Be Applied

只能应用一个类类型转换



A class-type conversion may not be followed by another class-type conversion. If more than one class-type conversion is needed then the code is in error.

类类型转换之后不能再跟另一个类类型转换。如果需要多个类类型转换，则代码将出错。

For example, assume we had another class, `Integral`, that could be converted to `SmallInt` but that had no conversion to `int`:

例如，假定有另一个类 `Integral`，它可以转换为 `SmallInt` 但不能转换为 `int`:

```
// class to hold unsigned integral values
class Integral {
public:
    Integral(int i = 0): val(i) { }
    operator SmallInt() const { return val % 256; }
```

Section 14.9. Conversions and Class Types

```
private:  
    std::size_t val;  
};
```

We could use an `Integral` where a `SmallInt` is needed, but not where an `int` is required:

可以在需要 `SmallInt` 的地方使用 `Integral`, 但不能在需要 `int` 的地方使用 `Integral`:

```
int calc(int);  
Integral intVal;  
SmallInt si(intVal); // ok: convert intVal to SmallInt and copy to si  
int i = calc(si); // ok: convert si to int and call calc  
int j = calc(intVal); // error: no conversion to int from Integral
```

When we create `si`, we use the `SmallInt` copy constructor. First `int_val` is converted to a `SmallInt` by invoking the `Integral` conversion operator to generate a temporary value of type `SmallInt`. The (synthesized) `SmallInt` copy constructor then uses that value to initialize `si`.

创建 `si` 时使用 `SmallInt` 复制构造函数。首先调用 `Integral` 转换操作符产生一个 `SmallInt` 类型的临时值, 将 `int_val` 对象转换为 `SmallInt`。然后 (合成的) 复制构造函数使用该对象值初始化 `si`。

The first call to `calc` is also okay: The argument `si` is automatically converted to `int`, and the `int` value is passed to the function.

第一个 `calc` 调用也是正确的: 将实参 `si` 自动转换为 `int`, 然后将 `int` 值传给函数。

The second call is an error: There is no direct conversion from `Integral` to `int`. To get an `int` from an `Integral` would require two class-type conversions: first from `Integral` to `SmallInt` and then from `SmallInt` to `int`. However, the language allows only one class-type conversion, so the call is in error.

第二个 `calc` 调用是错误的: 没有从 `Integral` 到 `int` 的直接转换。从 `int` 需要两次类类型转换: 首先从 `Integral` 到 `SmallInt`, 然后从 `SmallInt` 到 `int`。但是, 语言只允许一次类类型转换, 所以该调用出错。

Standard Conversions Can Precede a Class-Type Conversion

标准转换可放在类类型转换之前

When using a constructor to perform an implicit conversion (Section 12.4.4, p. 462), the parameter type of the constructor need not exactly match the type supplied. For example, the following code invokes the constructor `SmallInt(int)` defined in class `SmallInt` to convert `sobj` to the type `SmallInt`:

使用构造函数执行隐式转换(第 12.4.4 节)的时候, 构造函数的形参类型不必与所提供的类型完全匹配。例如, 下面的代码调用 `SmallInt(int)` 类中定义的构造函数(`SmallInt(int)`)将 `sobj` 转换为 `SmallInt` 类型:

```
void calc(SmallInt);  
short sobj;  
// sobj promoted from short to int  
// that int converted to SmallInt through the SmallInt(int) constructor  
calc(sobj);
```

If needed, a standard conversion sequence can be applied to an argument before a constructor is called to perform a class-type conversion. To call the function `calc()`, a standard conversion is applied to convert `dobj` from type `double` to type `int`. The `SmallInt(int)` constructor is then invoked to convert the result of the conversion to the type `SmallInt`.

如果需要, 在调用构造函数执行类类型转换之前, 可将一个标准转换序列应用于实参。为了调用函数 `calc()`, 应用标准转换将 `dobj` 从 `double` 类型转换为 `int` 类型, 然后调用构造函数 `SmallInt(int)` 将转换结果转换为 `SmallInt` 类型。

Exercises Section 14.9.2

Exercise 14.40: Write operators that could convert a `Sales_item` to `string` and to `double`. What values do you think these operators should return? Do you think these conversions are a good idea? Explain why or why not.

编写可将 `Sales_item` 对象转换为 `string` 类型和 `double` 类型的操作符。你认为这些操作符应返回什么值? 你认为定义这些操作符是个好办法吗? 解释你的结论。

Section 14.9. Conversions and Class Types

Exercise Explain the difference between these two conversion operators:

14.41:

解释这两个转换操作符之间的不同:

```
class Integral {  
public:  
    const int();  
    int() const;  
};
```

Are either of these conversions too restricted? If so, how might you make the conversion more general?

这两个转换操作符是否太严格了? 如果是, 怎样使得转换更通用一些?

Exercise Define a conversion operator to `bool` for the `CheckoutRecord` class from the exercises in [Section 14.2.1](#) (p. 515).

为第 14.2.1 节习题中的 `CheckoutRecord` 类定义到 `bool` 的转换操作符。

Exercise Explain what the `bool` conversion operator does. Is that the only possible meaning for this conversion for the `CheckoutRecord` type? Explain whether you think this conversion is a good use of a conversion operation.

解释 `bool` 转换操作符做了什么。这是这个 `CheckoutRecord` 类型转换唯一可能的含义吗? 解释你是否认为这个转换是一种转换操作的良好使用。

14.9.3. Argument Matching and Conversions

14.9.3. 实参匹配和转换



The rest of this chapter covers a somewhat advanced topic. It can be safely skipped on first reading.

本章其余部分讨论比较高级的主题。在第一次阅读时可跳过这些内容。

Class-type conversions can be a boon to implementing and using classes. By defining a conversion to `int` for `SmallInts`, we made the class easier to implement and easier to use. The `int` conversion lets users of `SmallInt` use all the arithmetic and relational operators on `SmallInt` objects. Moreover, users can safely write expressions that intermix `SmallInts` and other arithmetic types. The class implementor's job is made much easier by defining a single conversion operator instead of having to define 48 (or more) overloaded operators.

类类型转换可能是实现和使用类的一个好处。通过为 `SmallInt` 定义到 `int` 的转换, 能够更容易实现和使用 `SmallInt` 类。`int` 转换使 `SmallInt` 的用户能够对 `SmallInt` 对象使用所有算术和关系操作符, 而且, 用户可以安全编写将 `SmallInt` 和其他算术类型混合使用的表达式。定义一个转换操作符就能代替定义 48 个(或更多)重载操作符, 类实现者的工作就简单多了。

Class-type conversions can also be a great source of compile-time errors. Problems arise when there are multiple ways to convert from one type to another. If there are several class-type conversions that could be used, the compiler must figure out which one to use for a given expression. In this section, we look at how class-type conversions are used to match an argument to its corresponding parameter. We look first at how parameters are matched for functions that are not overloaded and then look at overloaded functions.

类类型转换也可能是编译时错误的一大来源。当从一个类型转换到另一类型有多种方式时, 问题就出现了。如果有几个类类型转换可以使用, 编译器必须决定对给定表达式使用哪一个。在这一节, 我们介绍怎样用类类型转换将实参和对应形参相匹配。首先介绍非重载函数的形参匹配, 然后介绍重载函数的形参匹配。

Used carefully, class-type conversions can greatly simplify both class and user code. Used too freely, they can lead to mysterious compile-time errors that can be hard to understand and hard to avoid.

如果小心使用, 类类型转换可以大大简化类代码和用户代码。如果使用得太过自由, 类类型转换会产生令人迷惑的编译时错误, 这些错误难以理解而且难以避免。



Argument Matching and Multiple Conversion Operators

实参匹配和多个转换操作符

To illustrate how conversions on values of class type interact with function matching, we'll add two additional conversions to our `SmallInt` class. We'll add a second constructor that takes a `double` and also define a second conversion operator to convert `SmallInt` to `double`:

为了举例说明类类型值的转换怎样与函数匹配相互作用，我们给 `SmallInt` 类加上另外两个转换，包括接受一个 `double` 参数的构造函数和一个将 `SmallInt` 转换为 `double` 的转换操作符：

```
// unwise class definition:  
// multiple constructors and conversion operators to and from the built-in types  
// can lead to ambiguity problems  
class SmallInt {  
public:  
    // conversions to SmallInt from int and double  
    SmallInt(int = 0);  
    SmallInt(double);  
    // Conversions to int or double from SmallInt  
    // Usually it is unwise to define conversions to multiple arithmetic types  
    operator int() const { return val; }  
    operator double() const { return val; }  
    // ...  
private:  
    std::size_t val;  
};
```



Ordinarily it is a bad idea to give a class conversions to or from two built-in types. We do so here to illustrate the pitfalls involved.

一般而言，给出一个类与两个内置类型之间的转换是不好的做法，在这里这样做是为了举例说明所包含的缺陷。

Consider the simple case where we call a function that is not overloaded:

考虑最简单的调用非重载函数的情况：

```
void compute(int);  
void fp_compute(double);  
void extended_compute(long double);  
SmallInt si;  
compute(si);      // SmallInt::operator int() const  
fp_compute(si);   // SmallInt::operator double() const  
extended_compute(si); // error: ambiguous
```

Either conversion operator could be used in the call to `compute`:

任一转换操作符都可用于 `compute` 调用中：

1. `operator int` generates an exact match to the parameter type.

`operator int` 产生对形参类型的完全匹配。

Section 14.9. Conversions and Class Types

2. `operator double` followed by the standard conversion from `double` to `int` matches the parameter type.

首先调用 `operator double` 进行转换，后跟从 `double` 到 `int` 的标准转换与形参类型匹配。

An exact match is a better conversion than one that requires a standard conversion. Hence, the first conversion sequence is better. The conversion function `SmallInt::operator int()` is chosen to convert the argument.

完全匹配转换比需要标准转换的其他转换更好，因此，第一个转换序列更好，选择转换函数 `SmallInt::operator int()` 来转换实参。

Similarly, in the second call, `fp_compute` could be called using either conversion. However, the conversion to `double` is an exact match; it requires no additional standard conversion.

类似地，在第二个调用中，可用任一转换调用 `fp_compute`。但是，到 `double` 的转换是一个完全匹配，不需要额外的标准转换。

The final call to `extended_compute` is ambiguous. Either conversion function could be used, but each would have to be followed by a standard conversion to get to `long double`. Hence, neither conversion is better than the other, so the call is ambiguous.

最后一个对 `extended_compute` 的调用有二义性。可以使用任一转换函数，但每个都必须跟上一个标准转换来获得 `long double`，因此，没有一个转换比其他的更好，调用具有二义性。



If two conversion operators could be used in a call, then the rank of the standard conversion ([Section 7.8.4](#), p. 272), if any, following the conversion function is used to select the best match.

如果两个转换操作符都可用在一个调用中，而且在转换函数之后存在标准转换（[第 7.8.4 节](#)），则根据该标准转换的类别选择最佳匹配。

Argument Matching and Conversions by Constructors

实参匹配和构造函数转换

Just as there might be two conversion operators, there can also be two constructors that might be applied to convert a value to the target type of a conversion.

正如可能存在两个转换操作符，也可能存在两个构造函数可以用来将一个值转换为目标类型。

Consider the `manip` function, which takes an argument of type `SmallInt`:

考虑 `manip` 函数，它接受一个 `SmallInt` 类型的实参：

```
void manip(const SmallInt &);  
double d; int i; long l;  
manip(d); // ok: use SmallInt(double) to convert the argument  
manip(i); // ok: use SmallInt(int) to convert the argument  
manip(l); // error: ambiguous
```

In the first call, we could use either of the `SmallInt` constructors to convert `d` to a value of type `SmallInt`. The `int` constructor requires a standard conversion on `d`, whereas the `double` constructor is an exact match. Because an exact match is better than a standard conversion, the constructor `SmallInt(double)` is used for the conversion.

在第一个调用中，可以用任一构造函数将 `d` 转换为 `SmallInt` 类型的值。`int` 构造函数需要对 `d` 的标准转换，而 `double` 构造函数完全匹配。因为完全匹配比标准转换更好，所以用构造函数 `SmallInt(double)` 进行转换。

In the second call, the reverse is true. The `SmallInt(int)` constructor provides an exact match no additional conversion is needed. To call the `SmallInt` constructor that takes a `double` would require that `i` first be converted to `double`. For this call, the `int` constructor would be used to convert the argument.

在第二个调用中，情况恰恰相反，构造函数 `SmallInt(int)` 提供完全匹配——不需要附加的转换，调用接受一个 `double` 参数的 `SmallInt` 构造函数需要首先将 `i` 转换为 `double` 类型。对于这个调用，用 `int` 构造函数转换实参。

The third call is ambiguous. Neither constructor is an exact match for `long`. Each would require that the argument be converted before using the constructor:

第三个调用具有二义性。没有构造函数完全匹配于 `long`。使用每一个构造函数之前都需要对实参进行转换：

1. standard conversion (`long` to `double`) followed by `SmallInt(double)`

标准转换（从 `long` 到 `double`）后跟 `SmallInt(double)`。

Section 14.9. Conversions and Class Types

2. standard conversion (`long` to `int`) followed by `SmallInt(int)`

标准转换（从 `long` 到 `int`）后跟 `SmallInt(int)`。

These conversion sequences are indistinguishable, so the call is ambiguous.

这些转换序列是不能区别的，所以该调用具有二义性。



When two constructor-defined conversions could be used, the rank of the standard conversion, if any, required on the constructor argument is used to select the best match.

当两个构造函数定义的转换都可以使用时，如果存在构造函数实参所需的标准转换，就用该标准转换的类型选择最佳匹配。

Ambiguities When Two Classes Define Conversions

当两个类定义了转换时的二义性

When two classes define conversions to each other, ambiguities are likely:

当两个类定义了相互转换时，很可能存在二义性：

```
class Integral;
class SmallInt {
public:
    SmallInt(Integral); // convert from Integral to SmallInt
    // ...
};

class Integral {
public:
    operator SmallInt() const; // convert from SmallInt to Integral
    // ...
};
void compute(SmallInt);
Integral int_val;
compute(int_val); // error: ambiguous
```

The argument `int_val` can be converted to a `SmallInt` in two different ways. The compiler could use the `SmallInt` constructor that takes an `Integral` object or it could use the `Integral` conversion operation that converts an `Integral` to a `SmallInt`. Because these two functions are equally good, the call is in error.

实参 `int_val` 可以用两种不同方式转换为 `SmallInt` 对象，编译器可以使用接受 `Integral` 对象的构造函数，也可以使用将 `Integral` 对象转换为 `SmallInt` 对象的 `Integral` 转换操作。因为这两个函数没有高下之分，所以这个调用会出错。

In this case, we cannot use a cast to resolve the ambiguity—the cast itself could use either the conversion operation or the constructor. Instead, we would need to explicitly call the conversion operator or the constructor:

在这种情况下，不能用显式类型转换来解决二义性——显式类型转换本身既可以使用转换操作又可以使用构造函数，相反，需要显式调用转换操作符或构造函数：

```
compute(int_val.operator SmallInt()); // ok: use conversion operator
compute(SmallInt(int_val));           // ok: use SmallInt constructor
```

Moreover, conversions that we might think would be ambiguous can be legal for what seem like trivial reasons. For example, our `SmallInt` class constructor copies its `Integral` argument. If we change the constructor so that it takes a reference to `const Integral`

而且，由于某些似乎微不足道的原因，我们认为可能有二义性的转换是合法的。例如，`SmallInt` 类构造函数复制它的 `Integral` 实参，如果改变构造函数以接受 `const Integral` 引用：

```
class SmallInt {
public:
    SmallInt(const Integral&);
};
```

our call to `compute(int_val)` is no longer ambiguous! The reason is that using the `SmallInt` constructor requires binding a reference to `int_val`, whereas using class `Integral`'s conversion operator avoids this extra step. This small difference is enough to tip the balance in favor of using the conversion operator.

Section 14.9. Conversions and Class Types

则对 `compute(int_val)` 的调用不再有二义性！原因在于使用 `SmallInt` 构造函数需要将一个引用绑定到 `int_val`，而使用 `Integral` 类的转换操作符可以避免这个额外的步骤。这一小区别足以使我们倾向于使用转换操作符。



The best way to avoid ambiguities or surprises is to avoid writing pairs of classes where each offers an implicit conversion to the other.

避免二义性最好的方法是避免编写互相提供隐式转换的成对的类。

Caution: Avoid Overuse of Conversion Functions

警告：避免转换函数的过度使用

As with using overloaded operators, judicious use of conversion operators can greatly simplify the job of a class designer and make using a class easier. However, there are two potential pitfalls: Defining too many conversion operators can lead to ambiguous code, and some conversions can be confusing rather than helpful.

与使用重载操作符一样，转换操作符的适当使用可以大大简化类设计者的工作并使得类的使用更简单。但是，有两个潜在的缺陷：定义太多转换操作符可能导致二义性代码，一些转换可能利大于弊。

The best way to avoid ambiguities is to ensure that there is at most one way to convert one type to another. The best way to do that is to limit the number of conversion operators. In particular there should be only one conversion to a built-in type.

避免二义性最好的方法是，保证最多只有一种途径将一个类型转换为另一类型。做到这点，最好的办法是限制转换操作符的数目，尤其是，到一种内置类型应该只有一个转换。

Conversion operators can be misleading when they are used where there is no obvious single mapping between the class type and the conversion type. In such cases, providing a conversion function may be confusing to the user of the class.

当转换操作符用于没有明显映射关系的类类型和转换类型之间时，容易引起误解，在这种情况下，提供转换函数可能会令类的使用者迷惑不解。

As an example, if we had a class that represented a `Date`, we might think it would be a good idea to provide a conversion from `Date` to `int`. However, what value should the conversion function return? The function might return the Julian date, which is the sequence number of the current date starting from 0 as January 1. But should the year precede the day or follow it? That is, would January 31, 1986 be represented as 1986031 or 311986? Alternatively, the conversion operator might return an `int` representing the day count since some epoch point. The counter might count days since January 1, 1971 or some other starting point.

例如，如果有一个表示 `Date` 的类，我们可能会认为提供从 `Date` 到 `int` 的转换是个好主意，但是，这个转换函数应返回什么值？该函数可以返回公历日期，这是表示当前日期的一个顺序数，以 0 表示 1 月 1 日，但年份是否应放在日期之前或之后？即，1986 年 1 月 31 日是否应表示为 1986031 或 311986？作为一种选择，转换操作符可以返回一个表示从某个新纪元点开始计数的天数，计数器可以从 1971 年 1 月 1 日或其他起始点开始计算天数。

The problem is that whatever choice is made, the use of `Date` objects will be ambiguous because there is no single one-to-one mapping between an object of type `Date` and a value of type `int`. In such cases, it is better not to define the conversion operator. Instead, the class ought to define one or more ordinary members to extract the information in these various forms.

问题在于，无论怎样选择，`Date` 对象的使用将具有二义性，因为没有一个 `Date` 类型对象与 `int` 类型值之间的一对一映射。在这种情况下，不定义转换函数更好。相反，这个类应该定义一个或多个普通成员从这些不同形式中抽取信息。

14.9.4. Overload Resolution and Class Arguments

14.9.4. 重载确定和类的实参

As we have just seen, the compiler automatically applies a class conversion operator or constructor when needed to convert an argument to a function. Class conversion operators, therefore, are considered during function resolution. Function overload resolution ([Section 7.8.2](#), p. 269) consists of three steps:

正如我们看到的，在需要转换函数的实参时，编译器自动应用类的转换操作符或构造函数。因此，应该在函数确定期间考虑类转换操作符。函数重载确定（[第 7.8.2 节](#)）由三

Section 14.9. Conversions and Class Types

步组成：

1. Determine the set of candidate functions: These are the functions with the same name as the function being called.
确定候选函数集合：这些是与被调用函数同名的函数。
2. Select the viable functions: These are the candidate functions for which the number and type of the function's parameters match the arguments in the call. When selecting the viable functions, the compiler also determines which conversion operations, if any, are needed to match each parameter.
选择可行的函数：这些是形参数目和类型与函数调用中的实参相匹配的候选函数。选择可行函数时，如果有转换操作，编译器还要确定需要哪个转换操作来匹配每个形参。
3. The best match function is selected. To determine the best match, the type conversions needed to convert argument(s) to the type of the corresponding parameter(s) are ranked. For arguments and parameters of class type, the set of possible conversions includes class-type conversions.
选择最佳匹配的函数。为了确定最佳匹配，对将实参转换为对应形参所需的类型转换进行分类。对于类类型的实参和形参，可能的转换的集合包括类类型转换。

Standard Conversions Following Conversion Operator

转换操作符之后的标准转换

Which function is the best match can depend on whether one or more class-type conversions are involved in matching different functions.

哪个函数是最佳匹配，可能依赖于在匹配不同函数中是否涉及了一个或多个类类型转换。



If two functions in the overload set can be matched *using the same conversion function*, then the rank of the standard conversion sequence that follows or precedes the conversion is used to determine which function has the best match.

如果重载集中的两个函数可以用同一转换函数匹配，则使用在转换之后或之前的标准转换序列的等级来确定哪个函数具有最佳匹配。

Otherwise, if *different conversion operations* could be used, then the conversions are considered equally good matches, regardless of the rank of any standard conversions that might or might not be required.

否则，如果可以使用不同转换操作，则认为这两个转换是一样的匹配，不管可能需要或不需要的标准转换的等级如何。

On page 541 we looked at the effect of class-type conversions on calls to functions that are not overloaded. Now, we'll look at similar calls but assume that the functions are overloaded:

第 14.9.3 节中介绍了类类型转换在非重载函数调用上的效果，现在，我们将看看类似的调用，但假定函数是重载的：

```
void compute(int);
void compute(double);
void compute(long double);
```

Assuming we use our original `SmallInt` class that only defines one conversion operator—the conversion to `int`—then if we pass a `SmallInt` to `compute`, the call is matched to the version of `compute` that takes an `int`.

假定使用原来的 `SmallInt` 类，该类只定义了一个转换操作符——从 `SmallInt` 到 `int` 的转换，那么，如果将 `SmallInt` 对象传给 `compute`，该调用与接受一个 `int` 的 `compute` 版本相匹配。

All three `compute` functions are viable:

三个函数都是可行的：

- `compute(int)` is viable because `SmallInt` has a conversion to `int`. That conversion is an exact match for the parameter.
`compute(int)` 可行，因为 `SmallInt` 有到 `int` 的转换，该转换是对形参的完全匹配。
- `compute(double)` and `compute(long double)` are also viable, by using the conversion to `int` followed by the appropriate standard conversion to either `double` or `long double`.
`compute(double)` 和 `compute(long double)` 也是可行的，可以使用到 `int` 的转换，后面跟上适当的用于 `double` 或 `long double` 的标准转换。

Section 14.9. Conversions and Class Types

Because all three functions would be matched using the *same class-type conversion*, the rank of the standard conversion, if any, is used to determine the best match. Because an exact match is better than a standard conversion, the function `compute(int)` is chosen as the best viable function.

因为可以用同一类类型转换来匹配这三个函数，如果存在标准转换，就用标准转换的等级确定最佳匹配。因为完全匹配比标准转换更好，所以选择 `compute(int)` 函数作为最佳可行函数。



The standard conversion sequence following a class-type conversion is used as a selection criterion only if the two conversion sequences use the same conversion operation.

只有两个转换序列使用同一转换操作时，才用类类型转换之后的标准转换序列作为选择标准。

Multiple Conversions and Overload Resolution

多个转换和重载确定

We can now see one reason why adding a conversion to `double` is a bad idea. If we use the revised `SmallInt` class that defines conversions to both `int` and `double`, then calling `compute` on a `SmallInt` value is ambiguous:

现在可以看看为什么增加一个到 `double` 的转换是个坏主意。如果使用修改后的定义了到 `int` 和 `double` 的转换的 `SmallInt` 类，则用 `SmallInt` 值调用 `compute` 具有二义性：

```
class SmallInt {
public:
    // Conversions to int or double from SmallInt
    // Usually it is unwise to define conversions to multiple arithmetic types
    operator int() const { return val; }
    operator double() const { return val; }
    // ...
private:
    std::size_t val;
};
void compute(int);
void compute(double);
void compute(long double);
SmallInt si;
compute(si);    // error: ambiguous
```

In this case we could use the `operator int` to convert `si` and call the version of `compute` that takes an `int`. Or we could use `operator double` to convert `si` and call `compute(double)`.

在这个例子中，可以使用 `operator int` 转换 `si` 并调用接受 `int` 参数的 `compute` 版本，或者，可以使用 `operator double` 转换 `si` 并调用 `compute(double)`。

The compiler will not attempt to distinguish between two different class-type conversions. In particular, even if one of the calls required a standard conversion following the class-type conversion and the other were an exact match, the compiler would still flag the call as an error.

编译器将不会试图区别两个不同的类类型转换。具体而言，即使一个调用需要在类类型转换之后跟一个标准转换，而另一个是完全匹配，编译器仍会将该调用标记为错误。

Explicit Constructor Call to Disambiguate

显式强制转换消除二义性

A programmer who is faced with an ambiguous conversion can use a cast to indicate explicitly which conversion operation to apply:

面对二义性转换，程序员可以使用强制转换来显式指定应用哪个转换操作：

```
void compute(int);
void compute(double);
SmallInt si;
compute(static_cast<int>(si)); // ok: convert and call compute(int)
```

This call is now legal because it explicitly says which conversion operation to apply to the argument. The type of the argument is forced to `int` by the cast. That type exactly matches the parameter of the first version of `compute` that takes an `int`.

Section 14.9. Conversions and Class Types

这个调用现在是合法的，因为它显式指出了将哪个转换操作应用到实参。实参类型强制转换为 `int`，该类型与接受 `int` 参数的第一个 `compute` 版本完全匹配。

Standard Conversions and Constructors

标准转换和构造函数

Let's look at overload resolution when multiple conversion constructors exist:

现在来看存在多个转换构造函数的重载确定：

```
class SmallInt {
public:
    SmallInt(int = 0);
};

class Integral {
public:
    Integral(int = 0);
};

void manip(const Integral&);
void manip(const SmallInt&);

manip(10); // error: ambiguous
```

The problem is that both classes, `Integral` and `SmallInt`, provide constructors that take an `int`. Either constructor could be used to match a version of `manip`. Hence, the call is ambiguous: It could mean convert the `int` to `Integral` and call the first version of `manip`, or it could mean convert the `int` to a `SmallInt` and call the second version.

问题在于，`Integral` 和 `SmallInt` 这两个类都提供接受 `int` 参数的构造函数，其中任意一个构造函数都可以与 `manip` 的一个版本相匹配，因此，函数调用有二义性：它既可以表示将 `Integral` 转换为 `int` 并调用 `manip` 的第一个版本，也可以表示 `SmallInt` 转换为 `int` 并调用 `manip` 的第二个版本。

This call would be ambiguous even if one of the classes defined a constructor that required a standard conversion for the argument. For example, if `SmallInt` defined a constructor that took a `short` instead of an `int`, the call `manip(10)` would require a standard conversion from `int` to `short` before using that constructor. The fact that one call requires a standard conversion and the other does not is immaterial when selecting among overloaded versions of a call. The compiler will not prefer the direct constructor; the call would still be ambiguous.

即使其中一个类定义了实参需要标准转换的构造函数，这个函数调用也可能具有二义性。例如，如果 `SmallInt` 定义了一个构造函数，接受 `short` 而不是 `int` 参数，函数调用 `manip(10)` 将在使用构造函数之前需要一个从 `int` 到 `short` 的标准转换。在函数调用的重载版本中进行选择时，一个调用需要标准转换而另一个不需要，这一事实不是实质性，编译器不会更喜欢直接构造函数，调用仍具有二义性。

Explicit Constructor Call to Disambiguate

显式构造函数调用消除二义性

The caller can disambiguate by explicitly constructing a value of the desired type:

调用者可以通过显式构造所需类型的值而消除二义性：

```
manip(SmallInt(10)); // ok: call manip(SmallInt)
manip(Integral(10)); // ok: call manip(Integral)
```



Needing to use a constructor or a cast to convert an argument in a call to an overloaded function is a sign of bad design.

在调用重载函数时，需要使用构造函数或强制类型转换来转换实参，这是设计拙劣的表现。

Exercises Section 14.9.4

Exercise Show the possible class-type conversion sequences for each of the following initializations. What
14.44: is the outcome of each initialization?

为下述每个初始化列出可能的类类型转换序列。每个初始化的结果是什么？

Section 14.9. Conversions and Class Types

```
class LongDouble {  
    operator double();  
    operator float();  
};  
LongDouble ldObj;  
(a) int ex1 = ldObj;    (b) float ex2 = ldObj;
```

Exercise 14.45: Which `calc()` function, if any, is selected as the best viable function for the following call? Show the conversion sequences needed to call each function and explain why the best viable function is selected.

哪个 `calc()` 函数是如下函数调用的最佳可行函数？列出调用每个函数所需的转换序列，并解释为什么所选定的就是最佳可行函数。

```
class LongDouble {  
public  
    LongDouble(double);  
    // ...  
};  
void calc(int);  
void calc(LongDouble);  
double dval;  
  
calc(dval); // which function?
```

14.9.5. Overloading, Conversions, and Operators

14.9.5. 重载、转换和操作符

Overloaded operators are overloaded functions. The same process that is used to resolve a call to an overloaded function is used to determine which operator built-in or class-typeto apply to a given expression. Given code such as

重载操作符就是重载函数。使用与确定重载函数调用一样的过程来确定将哪个操作符（内置的还是类类型的）应用于给定表达式。给定如下代码：

```
ClassX sc;  
int iobj = sc + 3;
```

there are four possibilities:

有四种可能性：

- There is an overloaded addition operator that matches `ClassX` and `int`.
- 有一个重载的加操作符与 `ClassX` 和 `int` 相匹配。
- There are conversions to convert `sc` and/or to convert an `int` to types for which `+` is defined. If so, this expression will use the conversion(s) followed by applying the appropriate addition operator.
- 存在转换，将 `sc` 和／或 `int` 值转换为定义了 `+` 的类型。如果是这样，该表达式将先使用转换，接着应用适当的加操作符。
- The expression is ambiguous because both a conversion operator and an overloaded version of `+` are defined.
- 因为既定义了转换操作符又定义了 `+` 的重载版本，该表达式具有二义性。
- The expression is invalid because there is neither a conversion nor an over-loaded `+` to use.
- 因为既没有转换又没有重载的 `+` 可以使用，该表达式非法。

Overload Resolution and Operators

重载确定和操作符



The fact that member and nonmember functions are possible changes how the set of candidate functions is selected.

成员函数和非成员函数都是可能的，这一事实改变了选择候选函数集的方式。

Overload resolution ([Section 7.8.2](#), p. 269) for operators follows the usual three-step process:

操作符的重载确定（[第 7.8.2 节](#)）遵循常见的三步过程：

1. Select the candidate functions.
选择候选函数。
2. Select the viable functions including identifying potential conversions sequences for each argument.
选择可行函数，包括识别每个实参的潜在转换序列。
3. Select the best match function.
选择最佳匹配的函数。

Candidate Functions for Operators

操作符的候选函数

As usual, the set of candidate functions consists of all functions that have the name of the function being used, and that are visible from the place of the call. In the case of an operator used in an expression, the candidate functions include the built-in versions of the operator along with all the ordinary nonmember versions of that operator. In addition, if the left-hand operand has class type, then the candidate set will contain the overloaded versions of the operator, if any, defined by that class.

一般而言，候选函数集由所有与被使用的函数同名的函数构成，被使用的函数可以从函数调用处看到。对于操作符用在表达式中的情况，候选函数包括操作符的内置版本以及该操作符的普通非成员版本。另外，如果左操作符具有类类型，而且该类定义了该操作符的重载版本，则候选集将包含操作符的重载版本。



Ordinarily, the candidate set for a call includes only member functions or nonmember functions but not both. When resolving the use of an operator, it is possible for both nonmember and member versions of the operator to be candidates.

一般而言，函数调用的候选集只包括成员函数或非成员函数，不会两者都包括。而确定操作符的使用时，操作符的非成员和成员版本可能都是候选者。

When resolving a call to a named function (as opposed to the use of an operator), the call itself determines the scope of names that will be considered. If the call is through an object of a class type (or through a reference or pointer to such an object), then only the member functions of that class are considered. Member and nonmember functions with the same name do *not* overload one another. When we use an overloaded operator, the call does not tell us anything about the scope of the operator function that is being used. Therefore, both member and nonmember versions must be considered.

确定指定函数的调用时，与操作符的使用相反，由调用本身确定所考虑的名字的作用域。如果是通过类类型的对象（或通过这种对象的引用或指针）的调用，则只需考虑该类的成员函数。具有同一名字的成员函数和非成员函数不会相互重载。使用重载操作符时，调用本身不会告诉我们与使用的操作符函数作用域相关的任何事情，因此，成员和非成员版本都必须考虑。

Caution: Conversions and Operators

警告：转换和操作符

Correctly designing the overloaded operators, conversion constructors, and conversion functions for a class requires some care. In particular, ambiguities are easy to generate if a class defines both conversion operators and overloaded operators. A few rules of thumb can be helpful:

正确设计类的重载操作符、转换构造函数和转换函数需要多加小心。尤其是，如果类既定义了转换操作符又定义了重载操作符，容易产生二义性。下面几条经验规则会有所帮助：

1. **Never define mutually converting classes that is, if class `Foo` has a constructor that takes an object of class `Bar`, do not give class `Bar` a conversion operator to type `Foo`.**
不要定义相互转换的类，即如果类 `Foo` 具有接受类 `Bar` 的对象的构造函数，不要再为类 `Bar` 定义到类型 `Foo` 的转换操作符。
2. **Avoid conversions to the built-in arithmetic types. In particular, if you do define a conversion to an arithmetic type, then**

Section 14.9. Conversions and Class Types

避免到内置算术类型的转换。具体而言，如果定义了到算术类型的转换，则

- Do not define overloaded versions of the operators that take arithmetic types. If users need to use these operators, the conversion operation will convert objects of your type, and then the built-in operators can be used.
- 不要定义接受算术类型的操作符的重载版本。如果用户需要使用这些操作符，转换操作符将转换你所定义的类型的对象，然后可以使用内置操作符。
- Do not define a conversion to more than one arithmetic type. Let the standard conversions provide conversions to the other arithmetic types.
- 不要定义转换到一个以上算术类型的转换。让标准转换提供到其他算术类型的转换。

The easiest rule of all: Avoid defining conversion functions and limit nonexplicit constructors to those that are "obviously right."

最简单的规则是：对于那些“明显正确”的，应避免定义转换函数并限制非显式构造函数。

Conversions Can Cause Ambiguity with Built-In Operators

转换可能引起内置操作符的二义性

Let's extend our `SmallInt` class once more. This time, in addition to a conversion operator to `int` and a constructor from `int`, we'll give our class an overloaded addition operator:

我们再次扩展 `SmallInt` 类。这一次，除了到 `int` 的转换操作符和接受 `int` 参数的构造函数之外，将增加一个重载的加操作符：

```
class SmallInt {
public:
    SmallInt(int = 0); // convert from int to SmallInt
    // conversion to int from SmallInt
    operator int() const { return val; }
    // arithmetic operators
    friend SmallInt
        operator+(const SmallInt&, const SmallInt&);
private:
    std::size_t val;
};
```

Now we could use this class to add two `SmallInts`, but we will run into ambiguity problems if we attempt to perform mixed-mode arithmetic:

现在，可以用这个类将两个 `SmallInts` 对象相加，但是，如果试图进行混合模式运算，将会遇到二义性问题：

```
SmallInt s1, s2;
SmallInt s3 = s1 + s2;           // ok: uses overloaded operator+
int i = s3 + 0;                // error: ambiguous
```

The first addition uses the overloaded version of `+` that takes two `SmallInt` values. The second addition is ambiguous. The problem is that we could convert `0` to a `SmallInt` and use the `SmallInt` version of `+`, or we could convert `s3` to `int` and use the built-in addition operator on `ints`.

第一个加使用接受两个 `SmallInt` 值的 `+` 的重载版本。第二个加有二义性，问题在于，可以将 `0` 转换为 `SmallInt` 并使用 `+` 的 `SmallInt` 版本，也可以将 `s3` 转换为 `int` 值并使用 `int` 值上的内置加操作符。



Providing both conversion functions to an arithmetic type and over-loaded operators for the same class type may lead to ambiguities between the overloaded operators and the built-in operators.

既为算术类型提供转换函数，又为同一类类型提供重载操作符，可能会导致重载操作符和内置操作符之间的二义性。

Viable Operator Functions and Conversions

可行的操作符函数和转换

We can understand the behavior of these two calls by listing the viable functions for each call. In the first call, there are two viable addition operators:

通过为每个调用列出可行函数，可以理解这两个调用的行为。在第一个调用中，有两个可行的加操作符：

- `operator+(const SmallInt&, const SmallInt&)`
 - The built-in `operator+(int, int)`
- 内置的 `operator+(int, int)`。

The first addition requires no conversions on either argument `s1` and `s2` match exactly the types of the parameters. Using the built-in addition operator for this addition would require conversions on both arguments. Hence, the overloaded operator is a better match for both arguments and is the one that is called. For the second addition

第一个加不需要实参转换——`s1` 和 `s2` 与形参的类型完全匹配。使用内置加操作符对两个实参都需要转换，因此，重载操作符与两个实参匹配得较好，所以将调用它。对于第二个加运算：

```
int i = s3 + 0;           // error: ambiguous
```

the same two functions are viable. In this case, the overloaded version of `+` matches the first argument exactly, but the built-in version is an exact match for the second argument. The first viable function is better for the left operand, whereas the second viable function is better for the right operand. The call is flagged as ambiguous because no best viable function can be found.

两个函数同样可行。在这种情况下，重载的 `+` 版本与第一个实参完全匹配，而内置版本与第二个实参完全匹配。第一个可行函数对左操作数而言较好，而第二个可行函数对右操作数而言较好。因为找不到最佳可行函数，所以将该调用标记为有二义性的。

Exercises Section 14.9.5

Exercise 14.46: Which `operator+`, if any, is selected as the best viable function for the addition operation in `main`? List the candidate functions, the viable functions, and the type conversions on the arguments for each viable function.

对于 `main` 中的加操作，哪个 `operator+` 是最佳可行函数？列出候选函数、可行函数以及对每个可行函数中实参的类型转换。

```
class Complex {
    Complex(double);
    // ...
};

class LongDouble {
    friend LongDouble operator+(LongDouble&, int);
public:
    LongDouble(int);
    operator double();
    LongDouble operator+(const Complex &);
    // ...
};
LongDouble operator+(const LongDouble &, double);
LongDouble ld(16.08);
double res = ld + 15.05; // which operator+ ?
```

Chapter Summary

小结

[Chapter 5](#) described the rich set of operators that C++ defines for the built-in types. That chapter also covered the standard conversions, which automatically convert operands from one type to another.

[第五章](#)介绍了 C++ 为内置类型所定义的丰富的操作符集合，该章也涵盖了标准转换，标准转换自动将操作数从一个类型转换为另一类型。

We can define a similarly rich set of expressions for objects of our own types (i.e., class or enumeration types) by defining overloaded versions of the built-in operators. An overloaded operator must have at least one operand of class or enumeration type. An overloaded operator has the same number of operands, associativity, and precedence as the corresponding operator when applied to the built-in types.

通过定义内置操作符的重载版本，我们可以为自己的类型（即，类类型或枚举类型）的对象定义同样丰富的表达式集合。重载操作符必须具有至少一个类类型或枚举类型的操作符。应用于内置类型时，重载操作符与对应操作符具有同样数目的操作数、同样的结合性和优先级。

Most overloaded operators can be defined as class members or as ordinary non-member functions. The assignment, subscript, call, and arrow operators must be class members. When an operator is defined as a member, it is a normal member function. In particular, member operators have an implicit `this` pointer, which is bound to the first (only operand for unary operators, left-hand operand for binary operators) operand.

大多数重载操作符可以定义为类成员或普通非成员函数，赋值操作符、下标操作符、调用操作符和箭头操作符必须为类成员。操作符定义为成员时，它是普通成员函数。具体而言，成员操作符有一个隐式 `this` 指针，该指针一定是第一个操作数，即，一元操作符唯一的操作数，二元操作符的左操作数。

Objects of classes that overload `operator()`, the function call operator, are known as "function objects." Such objects are often used to define predicate functions to be used in combination with the standard algorithms.

重载了 `operator()` (即，函数调用操作符) 的类的对象，称为“函数对象”。这种对象通常用于定义与标准算法结合使用的谓词函数。

Classes can define conversions that will be applied automatically when an object of one type is used where an object of a different type is needed. Constructors that take a single parameter and are not designated as `explicit` ([Section 12.4.4](#), p. 462) define conversions from the class type to other types. Overloaded operator conversion functions define conversions from other types to the class type. Conversion operators must be members of the class that they convert. They have no parameters and define no return value. Conversion operators return a value of the type of the operator for example, `operator int` returns an `int`.

类可以定义转换，当一个类型的对象在需要另一不同类型对象的地方时，自动应用这些转换。接受单个形参且未指定为 `explicit` ([第 12.4.4 节](#)) 的构造函数定义了从其他类型到类类型的转换，重载操作符转换函数则定义了从类类型到其他类型的转换。转换操作符必须为所转换类的成员，没有形参并且不定义返回值，转换操作符返回操作符所具有的类型的值，例如，`operator int` 返回 `int`。

Both overloaded operators and class-type conversions can make types easier and more natural to use. However, care should be taken to avoid designing operators or conversions that are not obvious to users of the type and to avoid defining multiple conversions between one type and another.

重载操作符和类型转换都有地更容易、更自然地使用类型，但是，必须注意避免设计对用户而言不明显操作符和转换，而且应避免定义一个类型与另一类型之间的多个转换。

Defined Terms

术语

binary function object (二元函数对象)

A class that has a function-call operator and represents one of the binary operators, such as one of the arithmetic or relational operators.
具有函数调用操作符且表示一个二元操作符（例如一个算术操作符或关系操作符）的类。

binder (绑定器)

An adaptor that binds an operand of a specified function object. For example, `bind2nd(minus<int>(), 2)` generates a unary function object that subtracts two from its operand.

绑定指定函数对象的一个操作数的适配器。例如，`bind2nd(minus<int>(), 2)` 产生一个一元函数对象，从操作数中减去 2。

class-type conversion (类类型转换)

Conversions to or from class types. Non-explicit constructors that take a single parameter define a conversion from the parameter type to the class type. Conversion operators define conversions from the class type to the type specified by the operator.

到类类型或从类类型的转换。接受一个形参的非显式构造函数定义从形参类型到类类型的转换。转换操作符定义从类类型到操作符所指定类型的转换。

conversion operators (转换操作符)

Conversion operators are member functions that define conversions from the class type to another type. Conversion operators must be a member of their class. They do not specify a return type and take no parameters. They return a value of the type of the conversion operator. That is, `operator int` returns an `int`, `operator Sales_item` returns a `Sales_item`, and so on.

转换操作符是定义从类类型到另一类型的转换的成员函数。转换操作符必须是类的成员，而且不能指定返回类型不能接受形参。转换操作符返回转换操作符类型的值，即，`operator int` 返回 `int`，`operator Sales_item` 返回 `Sales_item`，依此类推。

function adaptor (函数适配器)

Library type that provides a new interface for a function object.

为函数对象提供新接口的标准库的类型。

function object (函数对象)

Object of a class that defines an overloaded call operator. Function objects can be used where functions are normally expected.

定义了重载调用操作符的类的对象。函数对象可以用在需要函数的地方。

negator (求反器)

An adaptor that negates the value returned by the specified function object. For example, `not2(equal_to<int>())` generates a function object that is equivalent to `not_equal_to<int>`.

将指定函数对象的返回值求反的适配器。例如，`not2(equal_to<int>())` 产生与 `not_equal_to<int>` 等价的函数对象。

smart pointer (智能指针)

A class that defines pointer-like behavior and other functionality, such as reference counting, memory management, or more thorough checking. Such classes typically define overloaded versions of dereference (`operator*`) and member access (`operator->`).

一个类，定义了指针式行为和其他功能，如，引用计数、内存管理、更全面的检查等。这种类通常定义了解引用操作符 (`operator*`) 和成员访问操作符 (`operator->`) 的重载版本。

unary function object (一元函数对象)

A class that has a function-call operator and represents one of the unary operators, unary minus or logical NOT.

具有函数调用操作符且表示一个一元操作符的类，如一元减或逻辑非。

Team LiB

◀ PREVIOUS NEXT ▶

Chapter 15. Object-Oriented Programming

CONTENTS

Section 15.1 OOP: An Overview	558
Section 15.2 Defining Base and Derived Classes	560
Section 15.3 Conversions and Inheritance	577
Section 15.4 Constructors and Copy Control	580
Section 15.5 Class Scope under Inheritance	590
Section 15.6 Pure Virtual Functions	595
Section 15.7 Containers and Inheritance	597
Section 15.8 Handle Classes and Inheritance	598
Section 15.9 Text Queries Revisited	607
Chapter Summary	621
Defined Terms	621

Object-oriented programming is based on three fundamental concepts: data abstraction, inheritance, and dynamic binding. In C++ we use classes for data abstraction and class derivation to inherit one class from another: A derived class inherits the members of its base class(es). Dynamic binding lets the compiler determine at run time whether to use a function defined in the base or derived class.

面向对象编程基于三个基本概念：数据抽象、继承和动态绑定。在 C++ 中，用类进行数据抽象，用类派生从一个类继承另一个：派生类继承基类的成员。动态绑定使编译器能够在运行时决定是使用基类中定义的函数还是派生类中定义的函数。

Inheritance and dynamic binding streamline our programs in two ways: They make it easier to define new classes that are similar, but not identical, to other classes, and they make it easier for us to write programs that can ignore the details of how those similar types differ.

继承和动态绑定在两个方面简化了我们的程序：能够容易地定义与其他类相似但又不相同的新类，能够更容易地编写忽略这些相似类型之间区别的程序。

Many applications are characterized by concepts that are related but slightly different. For example, our bookstore might offer different pricing strategies for different books. Some books might be sold only at a given price. Others might be sold subject to some kind of discount strategy. We might give a discount to purchasers who buy a specified number of copies of the book. Or we might give a discount for only the first few copies purchased but charge full price for any bought beyond a given limit.

许多应用程序的特性可以用一些相关但略有不同的概念来描述。例如，书店可以为不同的书提供不同的定价策略，有些书可以只按给定价格出售，另一些书可以根据不同的折扣策略出售。可以给购买某书一定数量的顾客打折，或者，购买一定数量以内可以打折而超过给定限制就付全价。

Object-oriented programming (OOP) is a good match to this kind of application. Through inheritance we can define types that model the different kinds of books. Through dynamic binding we can write applications that use these types but that can ignore the type-dependent differences.

面向对象编程 (Object-oriented programming, OOP) 与这种应用非常匹配。通过继承可以定义一些类型，以模拟不同种类的书，通过动态绑定可以编写程序，使用这些类型而又忽略与具体类型相关的差异。

The ideas of inheritance and dynamic binding are conceptually simple but have profound implications for how we build our applications and for the features that programming languages must support. Before covering how C++ supports OOP, we'll look at the concepts that are fundamental to this style of programming.

继承和动态绑定的思想在概念上非常简单，但对于如何创建应用程序以及对于程序设计语言必须支持哪些特性，它们的含义深远。在讨论 C++ 如何支持面向对象编程之前，我们将介绍这种编程风格的一些基本概念。

15.1. OOP: An Overview

15.1. 面向对象编程: 概述

The key idea behind OOP is **polymorphism**. Polymorphism is derived from a Greek word meaning "many forms." We speak of types related by inheritance as polymorphic types, because in many cases we can use the "many forms" of a derived or base type interchangeably. As we'll see, in C++, polymorphism applies only to references or pointers to types related by inheritance.

面向对象编程的关键思想是多态性 (polymorphism)。多态性派生一个希腊单词，意思是“许多形态”。之所以称通过继承而相关联的类型为多态类型，是因为在许多情况下可以互换地使用派生类型或基类型的“许多形态”。正如我们将看到的，在 C++ 中，多态性仅用于通过继承而相关联的类型的引用或指针。

Inheritance

继承

Inheritance lets us define classes that model relationships among types, sharing what is common and specializing only that which is inherently different. Members defined by the **base class** are inherited by its **derived classes**. The derived class can use, without change, those operations that do not depend on the specifics of the derived type. It can redefine those member functions that do depend on its type, specializing the function to take into account the peculiarities of the derived type. Finally, a derived class may define additional members beyond those it inherits from its base class.

通过继承我们能够定义这样的类，它们对类型之间的关系建模，共享公共的东西，仅仅特化本质上不同的东西。派生类 (derived class) 能够继承基类 (base class) 定义的成员，派生类可以无须改变而使用那些与派生类型具体特性不相关的操作，派生类可以重定义那些与派生类型相关的成员函数，将函数特化，考虑派生类型的特性。最后，除了从基类继承的成员之外，派生类还可以定义更多的成员。

Classes related by inheritance are often described as forming an **inheritance hierarchy**. There is one class, referred to as the root, from which all the other classes inherit, directly or indirectly. In our bookstore example, we will define a base class, which we'll name `Item_base`, to represent undiscounted books. From `Item_base` we will inherit a second class, which we'll name `Bulk_item`, to represent books sold with a quantity discount.

我们经常称因继承而相关联的类为构成了一个**继承层次**。其中有一个类称为根，所以其他类直接或间接继承根类。在书店例子中，我们将定义一个基类，命名为 `Item_base`，命名为 `Bulk_item`，表示带数量折扣销售的书。

At a minimum, these classes will define the following operations:

这些类至少定义如下操作：

- an operation named `book` that will return the ISBN
名为 `book` 的操作，返回 ISBN。
- an operation named `net_price` that returns the price for purchasing a specified number of copies of a book
名为 `net_price` 的操作，返回购买指定数量的书的价格。

Classes derived from `Item_base` will inherit the `book` function without change: The derived classes have no need to redefine what it means to fetch the ISBN. On the other hand, each derived class will need to define its own version of the `net_price` function to implement an appropriate discount pricing strategy.

`Item_base` 的派生类将无须改变地继承 `book` 函数：派生类不需要重新定义获取 ISBN 的含义。另一方面，每个派生类需要定义自己的 `net_price` 函数版本，以实现适当的折扣价格策略。

In C++, a base class must indicate which of its functions it intends for its derived classes to redefine. Functions defined as **virtual** are ones that the base expects its derived classes to redefine. Functions that the base class intends its children to inherit are not defined as virtual.

在 C++ 中，基类必须指出希望派生类重写哪些函数，定义为 **virtual** 的函数是基类期待派生类重新定义的，基类希望派生类继承的函数不能定义为虚函数。

Given this discussion, we can see that our classes will define three (`const`) member functions:

讨论过这些之后，可以看到我们的类将定义三个 (`const`) 成员函数：

- A nonvirtual function, `std::string book()`, that returns the ISBN. It will be defined by `Item_base` and inherited by `Bulk_item`.
非虚函数 `std::string book()`, 返回 ISBN。由 `Item_base` 定义, `Bulk_item` 继承。
- Two versions of the virtual function, `double net_price(size_t)`, to return the total price for a given number of copies of a specific book. Both `Item_base` and `Bulk_item` will define their own versions of this function.
虚函数 `double net_price(size_t)` 的两个版本, 返回给定数目的某书的总价。`Item_base` 类和 `Bulk_item` 类将定义该函数自己的版本。

Dynamic Binding

动态绑定

Dynamic binding lets us write programs that use objects of any type in an inheritance hierarchy without caring about the objects' specific types. Programs that use these classes need not distinguish between functions defined in the base or in a derived class.

动态绑定我们能够编写程序使用继承层次中任意类型的对象, 无须关心对象的具体类型。使用这些类的程序无须区分函数是在基类还是在派生类中定义的。

For example, our bookstore application would let a customer select several books in a single sale. When the customer was done shopping, the application would calculate the total due. One part of figuring the final bill would be to print for each book purchased a line reporting the total quantity and sales price for that portion of the purchase.

例如, 书店应用程序可以允许顾客在一次交易中选择几本书, 当顾客购书时, 应用程序可以计算总的应付款, 指出最终账单的一个部分将是为每本书打印一行, 以显示总数和售价。

We might define a function named `print_total` to manage this part of the application. The `print_total` function, given an item and a count, should print the ISBN and the total price for purchasing the given number of copies of that particular book. The output of this function should look like:

可以定义一个名为 `print_total` 的函数管理应用程序的这个部分。给定一个项目和数量, 函数应打印 ISBN 以及购买给定数量的某书的总价。这个函数的输出应该像这样:

```
ISBN: 0-201-54848-8 number sold: 3 total price: 98
ISBN: 0-201-82470-1 number sold: 5 total price: 202.5
```

Our `print_total` function might look something like the following:

可以这样编写 `print_total` 函数:

```
// calculate and print price for given number of copies, applying any discounts
void print_total(ostream &os,
                 const Item_base &item, size_t n)
{
    os << "ISBN: " << item.book() // calls Item_base::book
    << "\tnumber sold: " << n << "\ttotal price: "
    // virtual call: which version of net_price to call is resolved at run time
    << item.net_price(n) << endl;
}
```

The function's work is trivial: It prints the results of calling `book` and `net_price` on its `item` parameter. There are two interesting things about this function.

该函数的工作很普通: 调用其 `item` 形参的 `book` 和 `net_price` 函数, 打印结果。关于这个函数, 有两点值得注意。

First, even though its second parameter is a reference to `Item_base`, we can pass either an `Item_base` object or a `Bulk_item` object to this function.

第一, 虽然这个函数的第二形参是 `Item_base` 的引用但可以将 `Item_base` 对象或 `Bulk_item` 对象传给它。

Second, because the parameter is a reference and the `net_price` function is virtual, the call to `net_price` will be resolved at run time. The version of `net_price` that is called will depend on the type of the argument passed to `print_total`. When the argument to `print_total` is a `Bulk_item`, the version of `net_price` that is run will be the one defined in `Bulk_item` that applies a discount. If the argument is an `Item_base` object, then the call will be to the version defined by `Item_base`.

第二, 因为形参是引用且 `net_price` 是虚函数, 所以对 `net_price` 的调用将在运行时确定。调用哪个版本的 `net_price` 将依赖于传给 `print_total` 的实参。如果传给 `print_total` 的实参是一个 `Bulk_item` 对象, 将运行 `Bulk_item` 中定义的应用折扣的 `net_price`; 如果实参是一个 `Item_base` 对象, 则调用由 `Item_base` 定义的版本。



In C++, dynamic binding happens when a virtual function is called through a reference (or a pointer) to a base class. The fact that a reference (or pointer) might refer to either a base- or a derived-class object is the key to dynamic binding. Calls to virtual functions made through a reference (or pointer) are resolved at run time: The function that is called is the one defined by the actual type of the object to which the reference (or pointer) refers.

在 C++ 中，通过基类的引用（或指针）调用虚函数时，发生动态绑定。引用（或指针）既可以指向基类对象也可以指向派生类对象，这一事实是动态绑定的关键。用引用（或指针）调用的虚函数在运行时确定，被调用的函数是引用（或指针）所指对象的实际类型所定义的。

15.2. Defining Base and Derived Classes

15.2. 定义基类和派生类

In many ways, base and derived classes are defined like other classes we have already seen. However, there are some additional features that are required when defining classes in an inheritance hierarchy. This section will present those features. Subsequent sections will see how use of these features impacts classes and the programs we write using inherited classes.

基类和派生类的定义在许多方面像我们已见过的其他类一样。但是，在继承层次中定义类还需要另外一些特性，本节将介绍这些特性，后续的章节将介绍这些特性的使用对类以及使用继承类编写的程序有何影响。

15.2.1. Defining a Base Class

15.2.1. 定义基类

Like any other class, a base class has data and function members that define its interface and implementation. In the case of our (very simplified) bookstore pricing application, our `Item_base` class defines the `book` and `net_price` functions and needs to store an ISBN and the standard price for the book:

像任意其他类一样，基类也有定义其接口和实现的数据和函数成员。在（非常简化的）书店定价应用程序的例子中，`Item_base` 类定义了 `book` 和 `net_price` 函数并且需要存储每本书的 ISBN 和标准价格：

```
// Item sold at an undiscounted price
// derived classes will define various discount strategies
class Item_base {
public:
    Item_base(const std::string &book = "", double sales_price = 0.0):
        isbn(book), price(sales_price) { }
    std::string book() const { return isbn; }
    // returns total sales price for a specified number of items
    // derived classes will override and apply different discount algorithms
    virtual double net_price(std::size_t n) const
    { return n * price; }
    virtual ~Item_base() { }
private:
    std::string isbn;      // identifier for the item
protected:
    double price;         // normal, undiscounted price
};
```

For the most part, this class looks like others we have seen. It defines a constructor along with the functions we have already described. That constructor uses default arguments ([Section 7.4.1](#), p. [253](#)), which allows it to be called with zero, one, or two arguments. It initializes the data members from these arguments.

这个类的大部分看起来像我们已见过的其他类一样。它定义了一个构造函数以及我们已描述过的函数，该构造函数使用[默认实参（第 7.4.1 节）](#)，允许用 0 个、1 个或两个实参进行调用，它用这些实参初始化数据成员。

The new parts are the `protected` access label and the use of the `virtual` keyword on the destructor and the `net_price` function. We'll explain `virtual` destructors in [Section 15.4.4](#) (p. [587](#)), but for now it is worth noting that classes used as the root class of an inheritance hierarchy generally define a `virtual` destructor.

新的部分是 `protected` 访问标号以及对析构函数和 `net_price` 函数所使用的保留字 `virtual`。我们将[第 15.4.4 节](#)解释虚析构函数，现在只需注意到继承层次的根类一般都要定义虚析构函数即可。

Base-Class Member Functions

基类成员函数

The `Item_base` class defines two functions, one of which is preceded by the keyword `virtual`. The purpose of the `virtual` keyword is to enable dynamic binding. By default, member functions are nonvirtual. Calls to nonvirtual functions are resolved at compile time. To specify that a function is virtual, we precede its return type by the keyword `virtual`. Any `nonstatic` member function, other than a constructor, may be virtual. The

Section 15.2. Defining Base and Derived Classes

`virtual` keyword appears only on the member-function declaration inside the class. The `virtual` keyword may not be used on a function definition that appears outside the class body.

`Item_base` 类定义了两个函数，其中一个前面带有保留字 `virtual`。保留字 `virtual` 的目的是启用动态绑定。成员默认为非虚函数，对非虚函数的调用在编译时确定。为了指明函数为虚函数，在其返回类型前面加上保留字 `virtual`。除了构造函数之外，任意非 `static` 成员函数都可以是虚函数。保留字只在类内部的成员函数声明中出现，不能用在类定义体外部出现的函数定义上。

We'll have more to say about virtual functions in [Section 15.2.4 \(p. 566\)](#).

[第 15.2.4 节](#) 将进一步介绍虚函数。



A base class usually should define as `virtual` any function that a derived class will need to redefine.

基类通常应将派生类需要重定义的任意函数定义为虚函数。

Access Control and Inheritance

访问控制和继承

In a base class, the `public` and `private` labels have their ordinary meanings: User code may access the `public` members and may not access the `private` members of the class. The `private` members are accessible only to the members and friends of the base class. A derived class has the same access as any other part of the program to the `public` and `private` members of its base class: It may access the `public` members and has no access to the `private` members.

在基类中，`public` 和 `private` 标号具有普通含义：用户代码可以访问类的 `public` 成员而不能访问 `private` 成员，`private` 成员只能由基类的成员和友元访问。派生类对基类的 `public` 和 `private` 成员的访问权限与程序中任意其他部分一样：它可以访问 `public` 成员而不能访问 `private` 成员。

Sometimes a class used as a base class has members that it wants to allow its derived classes to access, while still prohibiting access to those same members by other users. The `protected` access label is used for such members. A `protected` member may be accessed by a derived object but may not be accessed by general users of the type.

有时作为基类的类具有一些成员，它希望允许派生类访问但仍禁止其他用户访问这些成员。对于这样的成员应使用受保护的访问标号。`protected` 成员可以被派生类对象访问但不能被该类型的普通用户访问。

Our `Item_base` class expects its derived classes to redefine the `net_price` function. To do so, those classes will need access to the `price` member. Derived classes are expected to access `isbn` in the same way as ordinary users: through the `book` access function. Hence, the `isbn` member is `private` and is inaccessible to classes that inherit from `Item_base`.

我们的 `Item_base` 类希望它的派生类重定义 `net_price` 函数，为了重定义 `net_price` 函数，这些类将需要访问 `price` 成员。希望派生类用与普通用户一样通过 `book` 访问函数访问 `isbn`，因此，`isbn` 成员为 `private`，不能被 `Item_base` 的继承类所访问。

Exercises Section 15.2.1

Exercise

- 15.1: What is a virtual member?

什么是虚成员？

Exercise

- 15.2: Define the `protected` access label. How does it differ from `private`?

给出 `protected` 访问标号的定义。它与 `private` 有何不同？

Exercise

- 15.3: Define your own version of the `Item_base` class.

定义自己的 `Item_base` 类版本。

Exercise

- 15.4: A library has different kinds of materials that it lends outbooks, CDs, DVDs, and so forth. Each of the different kinds of lending material has different check-in, check-out, and overdue rules. The following class defines a base class that we might use for this application. Identify which functions are likely to be defined as virtual and which, if any, are likely to be common among all lending materials. (Note: we assume that `LibMember` is a class representing a customer of the

Section 15.2. Defining Base and Derived Classes

library, and `Date` is a class representing a calendar day of a particular year.)

图书馆可以借阅不同种类的资料——书、CD、DVD 等等。不同种类的借阅资料有不同的登记、检查和过期规则。下面的类定义了这个应用程序可以使用的基类。指出在所有借阅资料中，哪些函数可能定义为虚函数，如果有，哪些函数可能是公共的。（注：假定 `LibMember` 是表示图书馆读者的类，`Date` 是表示特定年份的日历日期的类。）

```
class Library {
public:
    bool check_out(const LibMember&);
    bool check_in (const LibMember&);
    bool is_late(const Date& today);
    double apply_fine();
    ostream& print(ostream& = cout);
    Date due_date() const;
    Date date_borrowed() const;
    string title() const;
    const LibMember& member() const;
};
```

15.2.2. `protected` Members

15.2.2. `protected` 成员

The `protected` access label can be thought of as a blend of `private` and `public`:

可以认为 `protected` 访问标号是 `private` 和 `public` 的混合：

- Like `private` members, `protected` members are inaccessible to users of the class.
像 `private` 成员一样，`protected` 成员不能被类的用户访问。
- Like `public` members, the `protected` members are accessible to classes derived from this class.
像 `public` 成员一样，`protected` 成员可被该类的派生类访问。

In addition, `protected` has another important property:

此外，`protected` 还有另一重要性质：

- A derived object may access the `protected` members of its base class *only* through a derived object. The derived class has no special access to the `protected` members of base type objects.
派生类只能通过派生类对象访问其基类的 `protected` 成员，派生类对其基类型对象的 `protected` 成员没有特殊访问权限。

As an example, let's assume that `Bulk_item` defines a member function that takes a reference to a `Bulk_item` object and a reference to an `Item_base` object. This function may access the `protected` members of its own object as well as those of its `Bulk_item` parameter. However, it has no special access to the `protected` members in its `Item_base` parameter:

例如，假定 `Bulk_item` 定义了一个成员函数，接受一个 `Bulk_item` 对象的引用和一个 `Item_base` 对象的引用，该函数可以访问自己对象的 `protected` 成员以及 `Bulk_item` 形参的 `protected` 成员，但是，它不能访问 `Item_base` 形参的 `protected` 成员。

```
void Bulk_item::memfcn(const Bulk_item &d, const Item_base &b)
{
    // attempt to use protected member
    double ret = price; // ok: uses this->price
    ret = d.price; // ok: uses price from a Bulk_item object
    ret = b.price; // error: no access to price from an Item_base
}
```

The use of `d.price` is okay, because the reference to `price` is through an object of type `Bulk_item`. The use of `b.price` is illegal because `Bulk_item` has no special access to objects of type `Item_base`.

`d.price` 的使用正确，因为是通过 `Bulk_item` 类型对象引用 `price`；`b.price` 的使用非法，因为对 `Base_item` 类型的对象没有特殊访问访问权限。

Key Concept: Class Design and Protected Members

关键概念：类设计与受保护成员

In the absence of inheritance, a class has two kinds of users: members of the class itself and the users of that class. This separation between kinds of users is reflected in the division of the class into **private** and **public** access levels. Users may access only the **public** interface; class members and friends may access both the **public** and **private** members.

如果没有继承，类只有两种用户：类本身的成员和该类的用户。将类划分为 **private** 和 **public** 访问级别反映了用户种类的这一分隔：用户只能访问 **public** 接口，类成员和友元既可以访问 **public** 成员也可以访问 **private** 成员。

Under inheritance, there is now a third kind of user of a class: programmers who will define new classes that are derived from the class. The provider of a derived class often (but not always) needs access to the (ordinarily **private**) base-class implementation. To allow that access while still preventing general access to the implementation, an additional access label, **protected**, is provided. The data and function members in a **protected** section of a class remain inaccessible to the general program, yet are accessible to the derived class. Anything placed within a **private** section of the base class is accessible only to the class itself and its friends. The **private** members are not accessible to the derived classes.

有了继承，就有了类的第三种用户：从类派生定义新类的程序员。派生类的提供者通常（但并不总是）需要访问（一般为 **private** 的）基类实现，为了允许这种访问而仍然禁止对实现的一般访问，提供了附加的 **protected** 访问标号。类的 **protected** 部分仍然不能被一般程序访问，但可以被派生类访问。只有类本身和友元可以访问基类的 **private** 部分，派生类不能访问基类的 **private** 成员。

When designing a class to serve as a base class, the criteria for designating a member as **public** do not change: It is still the case that interface functions should be **public** and data generally should not be **public**. A class designed to be inherited from must decide which parts of the implementation to declare as **protected** and which should be **private**. A member should be made **private** if we wish to prevent subsequently derived classes from having access to that member. A member should be made **protected** if it provides an operation or data that a derived class will need to use in its implementation. In other words, the interface to the derived type is the combination of both the **protected** and **public** members.

定义类充当基类时，将成员设计为 **public** 的标准并没有改变：仍然是接口函数应该为 **public** 而数据一般不应为 **public**。被继承的类必须决定实现的哪些部分声明为 **protected** 而哪些部分声明为 **private**。希望禁止派生类访问的成员应该设为 **private**，提供派生类实现所需操作或数据的成员应设为 **protected**。换句话说，提供给派生类型的接口是 **protected** 成员和 **public** 成员的组合。

15.2.3. Derived Classes

15.2.3. 派生类

To define a derived class, we use a [class derivation list](#) to specify the base class(es). A class derivation list names one or more base classes and has the form

为了定义派生类，使用[类派生列表](#)指定基类。类派生列表指定了一个或多个基类，具有如下形式：

```
class classname: access-label base-class
```

where *access-label* is one of **public**, **protected**, or **private**, and *base-class* is the name of a previously defined class. As we'll see, a derivation list might name more than one base class. Inheritance from a single base class is most common and is the topic of this chapter. [Section 17.3](#) (p. 731) covers use of multiple base classes.

这里 *access-label* 是 **public**、**protected** 或 **private**，*base-class* 是已定义的类的名字。类派生列表可以指定多个基类。继承单个基类是常见，也是本章的主题。[第 17.3 节](#)讨论多个基类的使用。

We'll have more to say about the access label used in a derivation list in [Section 15.2.5](#) (p. 570). For now, what's useful to know is that the access label determines the access to the inherited members. When we want to inherit the interface of a base class, then the derivation should be **public**.

[第 15.2.5 节](#)将进一步介绍派生列表中使用的访问标号，现在，只需要了解访问标号决定了对继承成员的访问权限。如果想要继承基类的接口，则应该进行 **public** 派生。

A derived class inherits the members of its base class and may define additional members of its own. Each derived object contains two parts: those members that it inherits from its base and those it defines itself. Typically, a derived class (re)defines only those aspects that differ from or extend the behavior of the base.

派生类继承基类的成员并且可以定义自己的附加成员。每个派生类对象包含两个部分：从基类继承的成员和自己定义的成员。一般而言，派生类只（重）定义那些与基类不同或扩展基类行为的方面。

Defining a Derived Class

定义派生类

In our bookstore application, we will derive `Bulk_item` from `Item_base`, so `Bulk_item` will inherit the `book`, `isbn`, and `price` members. `Bulk_item` must redefine its `net_price` function and define the data members needed for that operation:

在书店应用程序中，将从 `Item_base` 类派生 `Bulk_item` 类，因此 `Bulk_item` 类将继承 `book`、`isbn` 和 `price` 成员。`Bulk_item` 类必须重定义 `net_price` 函数定义该操作所需要的数据成员：

```
// discount kicks in when a specified number of copies of same book are sold
// the discount is expressed as a fraction used to reduce the normal price
class Bulk_item : public Item_base {
public:
    // redefines base version so as to implement bulk purchase discount policy
    double net_price(std::size_t) const;
private:
    std::size_t min_qty; // minimum purchase for discount to apply
    double discount;    // fractional discount to apply
};
```

Each `Bulk_item` object contains four data elements: It inherits `isbn` and `price` from `Item_base` and defines `min_qty` and `discount`. These latter two members specify the minimum quantity and the discount to apply once that number of copies are purchased. The `Bulk_item` class also needs to define a constructor, which we shall do in [Section 15.4](#) (p. 580).

每个 `Bulk_item` 对象包含四个数据成员：从 `Item_base` 继承的 `isbn` 和 `price`，自己定义的 `min_qty` 和 `discount`，后两个成员指定最小数量以及购买超过该数量时给的折扣。`Bulk_item` 类还需要定义一个构造函数，我们将在[第 15.4 节](#) 定义它。

Derived Classes and `virtual` Functions

派生类和虚函数

Ordinarily, derived classes redefine the virtual functions that they inherit, although they are not required to do so. If a derived class does not redefine a virtual, then the version it uses is the one defined in its base class.

尽管不是必须这样做，派生类一般会重定义所继承的虚函数。派生类没有重定义某个虚函数，则使用基类中定义的版本。

A derived type must include a declaration for each inherited member it intends to redefine. Our `Bulk_item` class says that it will redefine the `net_price` function but will use the inherited version of `book`.

派生类型必须对想要重定义的每个继承成员进行声明。`Bulk_item` 类指出，它将重定义 `net_price` 函数但将使用 `book` 的继承版本。

With one exception, the declaration ([Section 7.4](#), p. 251) of a virtual function in the derived class must exactly match the way the function is defined in the base. That exception applies to virtuals that return a reference (or pointer) to a type that is itself a base class. A virtual function in a derived class can return a reference (or pointer) to a class that is `publicly` derived from the type returned by the base-class function.

派生类中虚函数的声明（[第 7.4 节](#)）必须与基类中的定义方式完全匹配，但有一个例外：返回对基类型的引用（或指针）的虚函数。派生类中的虚函数可以返回基类函数所返回类型的派生类的引用（或指针）。

For example, the `Item_base` class might define a virtual function that returned an `Item_base*`. If it did, then the instance defined in the `Bulk_item` class could be defined to return either an `Item_base*` or a `Bulk_item*`. We'll see an example of this kind of virtual in [Section 15.9](#) (p. 607).

例如，`Item_base` 类可以定义返回 `Item_base*` 的虚函数，如果这样，`Bulk_item` 类中定义的实例可以定义为返回 `Item_base*` 或 `Bulk_item*`。[第 15.9 节](#) 将介绍这种虚函数的一个例子。



Once a function is declared as `virtual` in a base class it remains `virtual`; nothing the derived classes do can change the fact that the function is `virtual`. When a derived class redefines a `virtual`, it may use the `virtual` keyword, but it is not required to do so.

一旦函数在基类中声明为虚函数，它就一直为虚函数，派生类无法改变该函数为虚函数这一事实。派生类重定义虚函数时，可以使用 `virtual` 保留字，但不是必须这样做。

Derived Objects Contain Their Base Classes as Subobjects

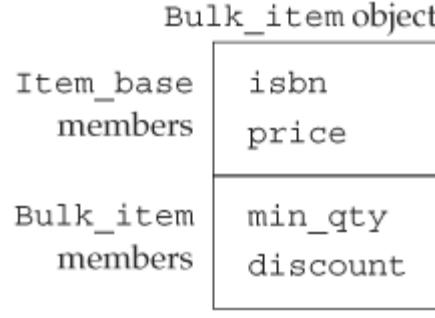
派生类对象包含基类对象作为子对象

A derived object consists of multiple parts: the (non`static`) members defined in the derived class itself plus the subobjects made up of the (non`static`) members of its base class. We can think of our `Bulk_item` class as consisting of two parts as represented in [Figure 15.1](#).

派生类对象由多个部分组成：派生类本身定义的（非 `static`）成员加上由基类（非 `static`）成员组成的子对象。可以认为 `Bulk_item` 对象由图 15.1 表示的两个部分组成。

Figure 15.1. Conceptual Structure of a `Bulk_item` Object

图 15.1. `Bulk_item` 对象的概念结构



 There is no requirement that the compiler lay out the base and derived parts of an object contiguously. Hence, [Figure 15.1](#) is a conceptual, not physical, representation of how classes work.

C++ 语言不要求编译器将对象的基类部分和派生部分和派生部分连续排列，因此，图 15.1 是关于类如何工作的概念表示而不是物理表示。

Functions in the Derived May Use Members from the Base

派生类中的函数可以使用基类的成员

As with any member function, a derived class function can be defined inside the class or outside, as we do here for the `net_price` function:

像任意成员函数一样，派生类函数可以在类的内部或外部定义，正如这里的 `net_price` 函数一样：

```
// if specified number of items are purchased, use discounted price
double Bulk_item::net_price(size_t cnt) const
{
    if (cnt >= min_qty)
        return cnt * (1 - discount) * price;
    else
        return cnt * price;
}
```

This function generates a discounted price: If the given quantity is more than `min_qty`, we apply the `discount` (which was stored as a fraction) to the `price`.

该函数产生折扣价格：如果给定数量多于 `min_qty`，就对 `price` 应用 `discount` (discount 存储为分数)。



Because each derived object has a base-class part, classes may access the `public` and `protected` members of its base class as if those members were members of the derived class itself.

因为每个派生类对象都有基类部分，类可以访问共基类的 `public` 和 `protected` 成员，就好像那些成员是派生类自己的成员一样。

A Class Must Be Defined to Be Used as a Base Class

用作基类的类必须是已定义的

A class must be defined before it can be used as a base class. Had we declared, but not defined, `Item_base`, we could not use it as our base class:

已定义的类才可以用作基类。如果已经声明了 `Item_base` 类, 但没有定义它, 则不能用 `Item_base` 作基类:

```
class Item_base; // declared but not defined
// error: Item_base must be defined
class Bulk_item : public Item_base { ... };
```

The reason for this restriction should already be easy to see: Each derived class contains, and may access, the members of its base class. To use those members, the derived class must know what they are. One implication of this rule is that it is impossible to derive a class from itself.

这一限制的原因应该很容易明白: 每个派生类包含并且可以访问其基类的成员, 为了使用这些成员, 派生类必须知道它们是什么。这一规则暗示着不可能从类自身派生出一个类。

Using a Derived Class as a Base Class

用派生类作基类

A base class can itself be a derived class:

基类本身可以是一个派生类:

```
class Base { /* ... */ };
class D1: public Base { /* ... */ };
class D2: public D1 { /* ... */ };
```

Each class inherits all the members of its base class. The most derived type inherits the members of its base, which in turn inherits the members of its base and so on up the inheritance chain. Effectively, the most derived object contains a subobject for each of its [immediate-base](#) and [indirect-base](#) classes.

每个类继承其基类的所有成员。最底层的派生类继承其基类的成员, 基类又继承自己的基类的成员, 如此沿着继承链依次向上。从效果来说, 最底层的派生类对象包含其每个[直接基类](#)和[间接基类](#)的子对象。

Declarations of Derived Classes

派生类的声明

If we need to declare (but not yet define) a derived class, the declaration contains the class name but does not include its derivation list. For example, the following forward declaration of `Bulk_item` results in a compile-time error:

如果需要声明 (但并不实现) 一个派生类, 则声明包含类名但不包含派生列表。例如, 下面的前向声明会导致编译时错误:

```
// error: a forward declaration must not include the derivation list
class Bulk_item : public Item_base;
```

The correct forward declarations are:

正确的前向声明为:

```
// forward declarations of both derived and nonderived class
class Bulk_item;
class Item_base;
```

Exercises Section 15.2.3

Exercise Which of the following declarations, if any, are incorrect?

15.5:

如果有，下面声明中哪些是错误的？

```
class Base { ... };
(a) class Derived : public Derived { ... };
(b) class Derived : Base { ... };
(c) class Derived : private Base { ... };
(d) class Derived : public Base;
(e) class Derived inherits Base { ... };
```

Exercise Write your own version of the `Bulk_item` class.

15.6:

编写自己的 `Bulk_item` 类版本。

Exercise We might define a type to implement a limited discount strategy. This class would give a

15.7: discount for books purchased up to a limit. If the number of copies purchased exceeds that limit, then the normal price should be applied to any books purchased beyond the limit. Define a class that implements this strategy.

可以定义一个类型实现有限折扣策略。这个类可以给低于某个上限的购书量一个折扣，如果购买的数量超过该上限，则超出部分的书应按正常价格购买。定义一个类实现这种策略。

15.2.4. `virtual` and Other Member Functions

15.2.4. `virtual` 与其他成员函数

By default, function calls in C++ do not use dynamic binding. To trigger dynamic binding, two conditions must be met: First, only member functions that are specified as `virtual` can be dynamically bound. By default, member functions are not `virtual`; nonvirtual functions are not dynamically bound. Second, the call must be made through a reference or a pointer to a base-class type. To understand this requirement, we need to understand what happens when we use a reference or pointer to an object that has a type from an inheritance hierarchy.

C++ 中的函数调用默认不使用动态绑定。要触发动态绑定，满足两个条件：第一，只有指定为虚函数的成员函数才能进行动态绑定，成员函数默认为非虚函数，非虚函数不进行动态绑定；第二，必须通过基类类型的引用或指针进行函数调用。要理解这一要求，需要理解在使用继承层次中某一类型的对象的引用或指针时会发生什么。

Derived to Base Conversions

从派生类型到基类的转换

Because every derived object contains a base part, we can bind a base-type reference to the base-class part of a derived object. We can also use a pointer to base to point to a derived object:

因为每个派生类对象都包含基类部分，所以可将基类类型的引用绑定到派生类对象的基类部分，也可以用指向基类的指针指向派生类对象：

```
// function with an Item_base reference parameter
double print_total(const Item_base&, size_t);
Item_base item;           // object of base type
// ok: use pointer or reference to Item_base to refer to an Item_base object
print_total(item, 10);   // passes reference to an Item_base object
Item_base *p = &item;     // p points to an Item_base object

Bulk_item bulk;          // object of derived type
// ok: can bind a pointer or reference to Item_base to a Bulk_item object
print_total(bulk, 10);   // passes reference to the Item_base part of bulk
p = &bulk;               // p points to the Item_base part of bulk
```

This code uses the same base-type pointer to point to an object of the base type and to an object of the derived type. It also calls a function that expects a reference to the base type, passing an object of the base-class type and also passing an object of the derived type. Both uses are fine, because every derived object has a base part.

这段代码使用同一基类类型指针指向基类类型的对象和派生类型的对象，该代码还传递基类类型和派生类型的对象来调用需要基类类型引用的函数，两种使用都是正确的，因为每个派生类对象都拥有基类部分。

Because we can use a base-type pointer or reference to refer to a derived-type object, when we use a base-type reference or pointer, we don't know the type of the object to which the pointer or reference is bound: A base-type reference or pointer might refer to an object of base type or an object of derived type. Regardless of which actual type the object has, the compiler treats the object as if it is a base type object. Treating a derived object as if it were a base is safe, because every derived object has a base subobject. Also, the derived class inherits the operations of the base class, meaning that any operation that might be performed on a base object is available through the derived object as well.

因为可以使用基类类型的指针或引用来引用派生类型对象，所以，使用基类类型的引用或指针时，不知道指针或引用所绑定的对象的类型：基类类型的引用或指针可以引用基类类型对象，也可以引用派生类型对象。无论实际对象具有哪种类型，编译器都将它当作基类类型对象。将派生类对象当作基类对象是安全的，因为每个派生类对象都拥有基类子对象。而且，派生类继承基类的操作，即，任何可以在基类对象上执行的操作也可以通过派生类对象使用。



The crucial point about references and pointers to base-class types is that the **static type** the type of the reference or pointer, which is knowable at compile time and the **dynamic type** the type of the object to which the pointer or reference is bound, which is knowable only at run time may differ.

基类类型引用和指针的关键点在于静态类型（在编译时可知的引用类型或指针类型）和动态类型（指针或引用所绑定的对象的类型这是仅在运行时可知的）可能不同。

Calls to **virtual** Functions May Be Resolved at Run time

可以在运行时确定 **virtual** 函数的调用

Binding a base-type reference or pointer to a derived object has no effect on the underlying object. The object itself is unchanged and remains a derived object. The fact that the actual type of the object might differ from the static type of the reference or pointer addressing that object is the key to dynamic binding in C++.

将基类类型的引用或指针绑定到派生类对象对基类对象没有影响，对象本身不会改变，仍为派生类对象。对象的实际类型可能不同于该对象引用或指针的静态类型，这是 C++ 中动态绑定的关键。

When a virtual function is called through a reference or pointer, the compiler generates code to *decide at run time* which function to call. The function that is called is the one that corresponds to the dynamic type. As an example, let's look again at the **print_total** function:

通过引用或指针调用虚函数时，编译器将生成代码，在运行时确定调用哪个函数，被调用的是与动态类型相对应的函数。例如，我们再来看 **print_total** 函数：

```
// calculate and print price for given number of copies, applying any discounts
void print_total(ostream &os,
                 const Item_base &item, size_t n)
{
    os << "ISBN: " << item.book() // calls Item_base::book
    << "\tnumber sold: " << n << "\ttotal price: "
    // virtual call: which version of net_price to call is resolved at run time
    << item.net_price(n) << endl;
}
```

Because the **item** parameter is a reference and **net_price** is virtual, the version of **net_price** that is called in **item.net_price(n)** depends at run time on the actual type of the argument bound to the **item** parameter:

因为 **item** 形参是一个引用且 **net_price** 是虚函数，**item.net_price(n)** 所调用的 **net_price** 版本取决于在运行时绑定到 **item** 形参的实参类型：

```
Item_base base;
Bulk_item derived;
// print_total makes a virtual call to net_price
print_total(cout, base, 10);      // calls Item_base::net_price
print_total(cout, derived, 10);   // calls Bulk_item::net_price
```

In the first call, the **item** parameter is bound, at run time, to an object of type **Item_base**. As a result, the call to **net_price** inside **print_total** calls the version defined in **Item_base**. In the second call, **item** is bound to an object of type **Bulk_item**. In this call, the version of **net_price** called from **print_total** will be the one defined by the **Bulk_item** class.

在第一个调用中，**item** 形参在运行时绑定到 **Item_base** 类型的对象，因此，**print_total** 内部调用 **Item_base** 中定义的 **net_price** 版本。在第二个调用中，**item** 形参绑定到 **Bulk_item** 类型的对象，从 **print_total** 调用的是 **Bulk_item** 类定义的 **net_price** 版本。

Key Concept: Polymorphism in C++

关键概念：C++ 中的多态性

The fact that the static and dynamic types of references and pointers can differ is the cornerstone of how C++ supports polymorphism.

引用和指针的静态类型与动态类型可以不同，这是 C++ 用以支持多态性的基石。

When we call a function defined in the base class through a base-class reference or pointer, we do not know the precise type of the object on which the function is executed. The object on which the function executes might be of the base type or it might be an object of a derived type.

通过基类引用或指针调用基类中定义的函数时，我们并不知道执行函数的对象的确切类型，执行函数的对象可能是基类类型的，也可能是派生类型的。

If the function called is nonvirtual, then regardless of the actual object type, the function that is executed is the one defined by the base type. If the function is virtual, then the decision as to which function to run is delayed until run time. The version of the virtual function that is run is the one defined by the type of the object to which the reference is bound or to which the pointer points.

如果调用非虚函数，则无论实际对象是什么类型，都执行基类类型所定义的函数。如果调用虚函数，则直到运行时才能确定调用哪个函数，运行的虚函数是引用所绑定的或指针所指向的对象所属类型定义的版本。

From the perspective of the code that we write, we need not care. As long as the classes are designed and implemented correctly, the operations will do the right thing whether the actual object is of base or derived type.

从编写代码的角度看我们无需担心。只要正确地设计和实现了类，不管实际对象是基类类型或派生类型，操作都将完成正确的工作。

On the other hand, an object is not polymorphic its type is known and unchanging. The dynamic type of an object (as opposed to a reference or pointer) is always the same as the static type of the object. The function that is run, virtual or nonvirtual, is the one defined by the type of the object.

另一方面，对象是非多态的——对象类型已知且不变。对象的动态类型总是与静态类型相同，这一点与引用或指针相反。运行的函数（虚函数或非虚函数）是由对象的类型定义的。



Virtuals are resolved at run time only if the call is made through a reference or pointer. Only in these cases is it possible for an object's dynamic type to be unknown until run time.

只有通过引用或指针调用，虚函数才在运行时确定。只有在这些情况下，直到运行时才知道对象的动态类型。

Nonvirtual Calls Are Resolved at Compile Time

在编译时确定非 `virtual` 调用

Regardless of the actual type of the argument passed to `print_total`, the call of `book` is resolved at compile time to `Item_base::book`.

不管传给 `print_total` 的实参的实际类型是什么，对 `book` 的调用在编译时确定为调用 `Item_base::book`。



Even if `Bulk_item` defined its own version of the `book` function, this call would call the one from the base class.

即使 `Bulk_item` 定义了自己的 `book` 函数版本，这个调用也会调用基类中的版本。

Nonvirtual functions are always resolved at compile time based on the type of the object, reference, or pointer from which the function is called. The type of `item` is reference to `const Item_base`, so a call to a nonvirtual function on that object will call the one from `Item_base` regardless of the type of the actual object to which `item` refers at run time.

非虚函数总是在编译时根据调用该函数的对象、引用或指针的类型而确定。`item` 的类型是 `const Item_base` 的引用，所以，无论在运行时 `item` 引用的实际对象是什么类型，调用该对象的非虚函数都将会调用 `Item_base` 中定义的版本。

Overriding the Virtual Mechanism

覆盖虚函数机制

In some cases, we want to override the virtual mechanism and force a call to use a particular version of a virtual function. We can do so by using the scope operator:

在某些情况下，希望覆盖虚函数机制并强制函数调用使用虚函数的特定版本，这里可以使用作用域操作符：

```
Item_base *baseP = &derived;
// calls version from the base class regardless of the dynamic type of baseP
double d = baseP->Item_base::net_price(42);
```

This code forces the call to `net_price` to be resolved to the version defined in `Item_base`. The call will be resolved at compile time.

这段代码强制将 `net_price` 调用确定为 `Item_base` 中定义的版本，该调用将在编译时确定。



Only code inside member functions should ever need to use the scope operator to override the virtual mechanism.

只有成员函数中的代码才应该使用作用域操作符覆盖虚函数机制。

Why might we wish to override the virtual mechanism? The most common reason is when a derived-class virtual calls the version from the base. In such cases, the base-class version might do work common to all types in the hierarchy. Each derived type adds only whatever is particular to its own type.

为什么会希望覆盖虚函数机制？最常见的理由是为了派生类虚函数调用基类中的版本。在这种情况下，基类版本可以完成继承层次中所有类型的公共任务，而每个派生类型只添加自己的特殊工作。

For example, we might define a `Camera` hierarchy with a virtual `display` operation. The `display` function in the `Camera` class would display information common to all `Cameras`. A derived class, such as `PerspectiveCamera`, would need to display both that common information and the information unique to `PerspectiveCamera`. Rather than duplicate the `Camera` operations within `PerspectiveCamera`'s implementation of `display`, we could explicitly invoke the `Camera` version to display the common information. In a case such as this one, we'd know exactly which instance to invoke, so there would be no need to go through the virtual mechanism.

例如，可以定义一个具有虚操作的 `Camera` 类层次。`Camera` 类中的 `display` 函数可以显示所有的公共信息，派生类（如 `PerspectiveCamera`）可能既需要显示公共信息又需要显示自己的独特信息。可以显式调用 `Camera` 版本以显示公共信息，而不是在 `PerspectiveCamera` 的 `display` 实现中复制 `Camera` 的操作。在这种情况下，已经确切知道调用哪个实例，因此，不需要通过虚函数机制。



When a derived virtual calls the base-class version, it *must* do so explicitly using the scope operator. If the derived function neglected to do so, then the call would be resolved at run time and would be a call to itself, resulting in an infinite recursion.

派生类虚函数调用基类版本时，必须显式使用作用域操作符。如果派生类函数忽略了这样做，则函数调用会在运行时确定并且将是一个自身调用，从而导致无穷递归。

Virtual Functions and Default Arguments

虚函数与默认实参

Like any other function, a virtual function can have default arguments. As usual, the value, if any, of a default argument used in a given call is determined at compile time. If a call omits an argument that has a default value, then the value that is used is the one defined by the type through which the function is called, irrespective of the object's dynamic type. When a virtual is called through a reference or pointer to base, then the default argument is the value specified in the declaration of the virtual in the base class. If a virtual is called through a pointer or reference to derived, the default argument is the one declared in the version in the derived class.

像其他任何函数一样，虚函数也可以有默认实参。通常，如果有用在给定调用中的默认实参值，该值将在编译时确定。如果一个调用省略了具有默认值的实参，则所用的值由调用该函数的类型定义，与对象的动态类型无关。通过基类的引用或指针调用虚函数时，默认实参为在基类虚函数声明中指定的值，如果通过派生类的指针或引用调用虚函数，则默认实参是在派生类的版本中声明的值。

Using different default arguments in the base and derived versions of the same virtual is almost guaranteed to cause trouble. Problems are likely to arise when the virtual is called through a reference or pointer to base, but the version that is executed is the one defined by the derived. In such cases, the default argument defined for the base version of the virtual will be passed to the derived version, which was defined using a different default argument.

在同一虚函数的基类版本和派生类版本中使用不同的默认实参几乎一定会引起麻烦。如果通过基类的引用或指针调用虚函数，但实际执行的是派生类中定义的版本，这时就可能会出现问题。在这种情况下，为虚函数的基类版本定义的默认实参将传给派生类定义的版本，而派生类版本是用不同的默认实参定义的。

Exercises Section 15.2.4

Exercise Given the following classes, explain each `print` function:

15.8:

对于下面的类，解释每个函数：

```
struct base {
    string name() { return basename; }
    virtual void print(ostream &os) { os << basename; }
private:
    string basename;
};

struct derived {
    void print() { print(ostream &os); os << " " << mem; }
private:
    int mem;
};
```

If there is a problem in this code, how would you fix it?

如果该代码有问题，如何修正？

Exercise Given the classes in the previous exercise and the following objects, determine which function is called at run time:

给定上题中的类和如下对象，确定在运行时调用哪个函数：

```
base bobj;    base *bp1 = &base;  base &br1 = bobj;
derived dobj; base *bp2 = &dobj; base &br2 = dobj;

(a) bobj.print(); (b) dobj.print();  (c) bp1->name();
(d) bp2->name(); (e) br1.print();  (f) br2.print();
```

15.2.5. Public, Private, and Protected Inheritance

15.2.5. 公用、私有和受保护的继承

Access to members defined within a derived class is controlled in exactly the same way as access is handled for any other class ([Section 12.1.2](#), p. 432). A derived class may define zero or more access labels that specify the access level of the members following that label. Access to the members the class inherits is controlled by a combination of the access level of the member in the base class and the access label used in the derived class' derivation list.

派生类中定义的成员访问控制的处理与任意其他类中完全一样（[第 12.1.2 节](#)）。派生类可以定义零个或多个访问标号，指定跟随其后的成员的访问级别。对类所继承的成员的访问由基类中的成员访问级别和派生类派生列表中使用的访问标号共同控制。



Each class controls access to the members it defines. A derived class may further restrict but may not loosen the access to the members that it inherits.

每个类控制它所定义的成员的访问。派生类可以进一步限制但不能放松对所继承的成员的访问。

The base class itself specifies the minimal access control for its own members. If a member is `private` in the base class, then only the base class and its friends may access that member. The derived class has no access to the `private` members of its base class, nor can it make those members accessible to its own users. If a base class member is `public` or `protected`, then the access label used in the derivation list determines the access level of that member in the derived class:

基类本身指定对自身成员的最小访问控制。如果成员在基类中为 `private`, 则只有基类和基类的友元可以访问该成员。派生类不能访问基类的 `private` 成员, 也不能使自己的用户能够访问那些成员。如果基类成员为 `public` 或 `protected`, 则派生列表中使用的访问标号决定该成员在派生类中的访问级别:

- In `public inheritance`, the members of the base retain their access levels: The `public` members of the base are `public` members of the derived and the `protected` members of the base are `protected` in the derived.

如果是公用继承, 基类成员保持自己的访问级别: 基类的 `public` 成员为派生类的 `public` 成员, 基类的 `protected` 成员为派生类的 `protected` 成员。

- In `protected inheritance`, the `public` and `protected` members of the base class are `protected` members in the derived class.

如果是受保护继承, 基类的 `public` 和 `protected` 成员在派生类中为 `protected` 成员。

- In `private inheritance`, all the members of the base class are `private` in the derived class.

如果是私有继承, 基类的所有成员在派生类中为 `private` 成员。

As an example, consider the following hierarchy:

例如, 考虑下面的继承层次:

```
class Base {
public:
    void basemem(); // public member
protected:
    int i; // protected member
};
struct Public_derived : public Base {
    int use_base() { return i; } // ok: derived classes can access i
    // ...
};
struct Private_derived : private Base {
    int use_base() { return i; } // ok: derived classes can access i
};
```

All classes that inherit from `Base` have the same access to the members in `Base`, regardless of the access label in their derivation lists. The derivation access label controls the access that *users* of the derived class have to the members inherited from `Base`:

无论派生列表中是什么访问标号, 所有继承 `Base` 的类对 `Base` 中的成员具有相同的访问。派生访问标号将控制派生类的用户对从 `Base` 继承而来的成员的访问:

```
Base b;
Public_derived d1;
Private_derived d2;
b.basemem(); // ok: basemem is public
d1.basemem(); // ok: basemem is public in the derived class
d2.basemem(); // error: basemem is private in the derived class
```

Both `Public_derived` and `Private_derived` inherit the `basemem` function. That member retains its access level when the inheritance is `public`, so `d1` can call `basemem`. In `Private_derived`, the members of `Base` are `private`; users of `Private_derived` may not call `basemem`.

`Public_derived` 和 `Private_derived` 都继承了 `basemem` 函数。当进行 `public` 继承时, 该成员保持其访问标号, 所以, `d1` 可以调用 `basemem`。在 `Private_derived` 中, `Base` 的成员为 `private`, `Private_derived` 的用户不能调用 `basemem`。

The derivation access label also controls access from indirectly derived classes:

派生访问标号还控制来自非直接派生类的访问:

```
struct Derived_from_Private : public Private_derived {
    // error: Base::i is private in Private_derived
    int use_base() { return i; }
};
struct Derived_from_Public : public Public_derived {
    // ok: Base::i remains protected in Public_derived
    int use_base() { return i; }
};
```

Classes derived from `Public_derived` may access `i` from the `Base` class because that member remains a `protected` member in `Public_derived`. Classes derived from `Private_derived` have no such access. To them all the members that `Private_base` inherited from `Base` are `private`.

从 `Public_derived` 派生的类可以访问来自 `Base` 类的 `i`, 是因为该成员在 `Public_derived` 中仍为 `protected` 成员。从 `Private_derived` 派生的类没有这样的访问, 对它们而言, `Private_derived` 从 `Base` 继承的所有成员均为 `private`。

Interface versus Implementation Inheritance

接口继承与实现继承

A `publicly` derived class inherits the interface of its base class; it has the same interface as its base class. In well-designed class hierarchies, objects of a `publicly` derived class can be used wherever an object of the base class is expected.

`public` 派生类继承基类的接口, 它具有与基类相同的接口。设计良好的类层次中, `public` 派生类的对象可以用在任何需要基类对象的地方。

Classes derived using either `private` or `protected` do not inherit the base-class interface. Instead, these derivations are often referred to as implementation inheritance. The derived class uses the inherited class in its implementation but does not expose the fact of the inheritance as part of its interface.

使用 `private` 或 `protected` 派生的类不继承基类的接口, 相反, 这些派生类通常被称为实现继承。派生类在实现中使用被继承但继承基类的部分并未成为其接口的一部分。

As we'll see in [Section 15.3](#) (p. 577), whether a class uses interface or implementation inheritance has important implications for users of the derived class.

如[第 15.3 节](#)所介绍的, 类是使用接口继承还是实现继承对派生类的用户具有重要意义。



By far the most common form of inheritance is `public`.

迄今为止, 最常见的继承形式是 `public`。

Key Concept: Inheritance versus Composition

关键概念：继承与组合

The design of inheritance hierarchies is a complex topic in its own right and well beyond the scope of this language primer. However, there is one important design guide that is so fundamental that every programmer should be familiar with it.

继承层次的设计本身是个复杂的主题, 已超出本书的范围。但是, 有一个重要的设计指南非常基础, 每个程序员都应该熟悉它。

When we define one class as publicly inherited from another, the derived class should reflect a so-called "Is A" relationship to the base class. In our bookstore example, our base class represents the concept of a book sold at a stipulated price. Our `Bulk_item` is a kind of book, but one with a different pricing strategy.

定义一个类作为另一个类的公用派生类时, 派生类应反映与基类的“是一种 (Is A) ”关系。在书店例子中, 基类表示按规定价格销售的书的概念, `Bulk_item` 是一种书, 但具有不同的定价策略。

Another common relationship among types is a so-called "Has A" relationship. Our bookstore classes have a price and they have an ISBN. Types related by a "Has A" relationship imply membership. Thus, our bookstore classes are composed from members representing the price and the ISBN.

类型之间另一种常见的关系是称为“有一个 (Has A) ”的关系。书店例子中的类具有价格和 ISBN。通过“有一个”关系而相关的类型暗含有成员关系, 因此, 书店例子中的类由表示价格和 ISBN 的成员组成。

Exempting Individual Members

去除个别成员

Section 15.2. Defining Base and Derived Classes

When inheritance is `private` or `protected`, the access level of members of the base may be more restrictive in the derived class than it was in the base:

如果进行 `private` 或 `protected` 继承，则基类成员的访问级别在派生类中比在基类中更受限：

```
class Base {  
public:  
    std::size_t size() const { return n; }  
protected:  
    std::size_t n;  
};  
class Derived : private Base { . . . };
```



The derived class can restore the access level of an inherited member. The access level cannot be made more or less restrictive than the level originally specified within the base class.

派生类可以恢复继承成员的访问级别，但不能使访问级别比基类中原来指定的更严格或更宽松。

In this hierarchy, `size` is `public` in `Base` but `private` in `Derived`. To make `size` `public` in `Derived` we can add a `using` declaration for it to a `public` section in `Derived`. By changing the definition of `Derived` as follows, we can make the `size` member accessible to users and `n` accessible to classes subsequently derived from `Derived`:

在这一继承层次中，`size` 在 `Base` 中为 `public`，但在 `Derived` 中为 `private`。为了使 `size` 在 `Derived` 中成为 `public`，可以在 `Derived` 的 `public` 部分增加一个 `using` 声明。如下这样改变 `Derived` 的定义，可以使 `size` 成员能够被用户访问，并使 `n` 能够被从 `Derived` 派生的类访问：

```
class Derived : private Base {  
public:  
    // maintain access levels for members related to the size of the object  
    using Base::size;  
protected:  
    using Base::n;  
    // ...  
};
```

Just as we can use a `using` declaration ([Section 3.1](#), p. 78) to use names from the `std` namespace, we may also use a `using` declaration to access a name from a base class. The form is the same except that the left-hand side of the scope operator is a class name instead of a namespace name.

正如可以使用 `using` 声明 ([第 3.1 节](#)) 从命名空间使用名字，也可以使用 `using` 声明访问基类中的名字，除了在作用域操作符左边用类名字代替命名空间名字之外，使用形式是相同的。

Default Inheritance Protection Levels

默认继承保护级别

In [Section 2.8](#) (p. 65) we learned that classes defined with the `struct` and `class` keywords have different default access levels. Similarly, the default inheritance access level differs depending on which keyword is used to define the derived class. A derived class defined using the `class` keyword has `private` inheritance. A class is defined with the `struct` keyword, has `public` inheritance:

在[第 2.8 节](#)介绍过用 `struct` 和 `class` 保留字定义的类具有不同的默认访问级别，同样，默认继承访问级别根据使用哪个保留字定义派生类也不相同。使用 `class` 保留字定义的派生默认具有 `private` 继承，而用 `struct` 保留字定义的类默认具有 `public` 继承：

```
class Base { /* ... */ };  
struct D1 : Base { /* ... */ }; // public inheritance by default  
class D2 : Base { /* ... */ }; // private inheritance by default
```

It is a common misconception to think that there are deeper differences between classes defined using the `struct` keyword and those defined using `class`. The only differences are the default protection level for members and the default protection level for a derivation. There are no other distinctions:

有一种常见的误解认为用 `struct` 保留字定义的类与用 `class` 定义的类有更大的区别。唯一的不同只是默认的成员保护级别和默认的派生保护级别，没有其他区别：

```
class D3 : public Base {  
public:  
    /* ... */  
};
```

```
// equivalent definition of D3
struct D3 : Base {           // inheritance public by default
    /* ... */                 // initial member access public by default
};

struct D4 : private Base {
private:
    /* ... */
};

// equivalent definition of D4
class D4 : Base {           // inheritance private by default
/* ... */                  // initial member access private by default
};
```



Although private inheritance is the default when using the `class` keyword, it is also relatively rare in practice. Because private inheritance is so rare, it is usually a good idea to explicitly specify `private`, rather than rely on the default. Being explicit makes it clear that private inheritance is intended and not an oversight.

尽管私有继承在使用 `class` 保留字时是默认情况，但这在实践中相对罕见。因为私有继承是如此罕见，通常显式指定 `private` 是比依赖于默认更好的办法。显式指定可清楚指出想要私有继承而不是一时疏忽。

Exercises Section 15.2.5

Exercise 15.10: In the exercises to [Section 15.2.1](#) (p. 562) you wrote a base class to represent the lending policies of a library. Assume the library offers the following kinds of lending materials, each with its own check-out and check-in policy. Organize these items into an inheritance hierarchy:

[第 15.2.1 节](#) 的习题中编写了一个表示图书馆借阅政策的基类。假定图书馆提供下列种类的借阅资料，每一种有自己的检查和登记政策。将这些项目组织成一个继承层次：

book children's puppet cdrom book sony play station	audio book sega video game nintendo video game video game	record video rental book
--	--	--------------------------------

Exercise 15.11: Choose one of the following general abstractions containing a family of types (or choose one of your own). Organize the types into an inheritance hierarchy:

在下列包含一族类型的一般抽象中选择一种（或者自己选择一个），将这些类型组织成一个继承层次。

[\[View full width\]](#)

- (a) Graphical file formats (such as gif, tiff, jpeg, bmp)
 (a) 图像文件格式（如 gif, tiff, jpeg, bmp）
- (b) Geometric primitives (such as box, circle, sphere, cone)
 (b) 几何图元（如矩形, 圆, 球形, 锥形）
- (c) C++ language types (such as class, function, member
 (c) C++ 语言的类型（如类, 函数, 成员函数）
➡ function)

Exercise 15.12: For the class you chose in the previous exercise, identify some of the likely virtual functions as well as `public` and `protected` members.

对上题中选择的类，标出可能的虚函数以及 `public` 和 `protected` 成员。

15.2.6. Friendship and Inheritance

15.2.6. 友元关系与继承

Section 15.2. Defining Base and Derived Classes

As with any other class, a base or derived class can make other class(es) or function(s) friends ([Section 12.5](#), p. 465). Friends may access the class' `private` and `protected` data.

像其他类一样，基类或派生类可以使其他类或函数成为友元（[第 12.5 节](#)）。友元可以访问类的 `private` 和 `protected` 数据。



Friendship is not inherited. Friends of the base have no special access to members of its derived classes. If a base class is granted friendship, only the base has special access. Classes derived from that base have no access to the class granting friendship.

友元关系不能继承。基类的友元对派生类的成员没有特殊访问权限。如果基类被授予友元关系，则只有基类具有特殊访问权限，该基类的派生类不能访问授予友元关系的类。

Each class controls friendship to its own members:

每个类控制对自己的成员的友元关系：

```
class Base {
    friend class Frnd;
protected:
    int i;
};

// Frnd has no access to members in D1
class D1 : public Base {
protected:
    int j;
};

class Frnd {
public:
    int mem(Base b) { return b.i; } // ok: Frnd is friend to Base
    int mem(D1 d) { return d.i; }   // error: friendship doesn't inherit
};

// D2 has no access to members in Base
class D2 : public Frnd {
public:
    int mem(Base b) { return b.i; } // error: friendship doesn't inherit
};
```

If a derived class wants to grant access to its members to the friends of its base class, the derived class must do so explicitly: Friends of the base have no special access to types derived from that base class. Similarly, if a base and its derived types all need access to another class, that class must specifically grant access to the base and each derived class.

如果派生类想要将自己成员的访问权授予其基类的友元，派生类必须显式地这样做：基类的友元对从该基类派生的类型没有特殊访问权限。同样，如果基类和派生类都需要访问另一个类，那个类必须特地将访问权限授予基类的和每一个派生类。

15.2.7. Inheritance and Static Members

15.2.7. 继承与静态成员

If a base class defines a `static` member ([Section 12.6](#), p. 467) there is only one such member defined for the entire hierarchy. Regardless of the number of classes derived from the base class, there exists a single instance of each `static` member. `static` members obey normal access control: If the member is `private` in the base class, then derived classes have no access to it. Assuming the member is accessible, we can access the `static` member either through the base or derived class. As usual, we can use either the scope operator or the dot or arrow member access operators.

如果基类定义 `static` 成员（[第 12.6 节](#)），则整个继承层次中只有一个这样的成员。无论从基类派生出多少个派生类，每个 `static` 成员只有一个实例。`static` 成员遵循常规访问控制：如果成员在基类中为 `private`，则派生类不能访问它。假定可以访问成员，则既可以通过基类访问 `static` 成员，也可以通过派生类访问 `static` 成员。一般而言，既可以使用作用域操作符也可以使用点或箭头成员访问操作符。

```
struct Base {
    static void statmem(); // public by default
};

struct Derived : Base {
    void f(const Derived&);
};

void Derived::f(const Derived &derived_obj)
{
    Base::statmem();      // ok: Base defines statmem
    Derived::statmem();   // ok: Derived in herits statmem
    // ok: derived objects can be used to access static from base
    derived_obj.statmem(); // accessed through Derived object
```

```
statmem();           // accessed through this class
```

Exercises Section 15.2.7

Exercise**15.13:**

Given the following classes, list all the ways a member function in `c1` might access the `static` members of `ConcreteBase`. List all the ways an object of type `c2` might access those members.

对于下面的类，列出 `c1` 中的成员函数访问 `ConcreteBase` 的 `static` 成员的所有方式，列出 `c2` 类型的对象访问这些成员的所有方式。

```
struct ConcreteBase {  
    static std::size_t object_count();  
protected:  
    static std::size_t obj_count;  
};  
struct C1 : public ConcreteBase { /* . . . */ };  
struct C2 : public ConcreteBase { /* . . . */ };
```

Team LiB**◀ PREVIOUS** **NEXT ▶**

15.3. Conversions and Inheritance

15.3. 转换与继承



Understanding conversions between base and derived types is essential to understanding how object-oriented programming works in C++.

理解基类类型和派生类型之间的转换，对于理解面向对象编程在 C++ 中如何工作非常关键。

As we've seen, every derived object contains a base part, which means that we can execute operations on a derived object as if it were a base object. Because a derived object is also a base, there is an automatic conversion from a reference to a derived type to a reference to its base type(s). That is, we can convert a reference to a derived object to a reference to its base subobject and likewise for pointers.

我们已经看到，每个派生类对象包含一个基类部分，这意味着可以像使用基类对象一样在派生类对象上执行操作。因为派生类对象也是基类对象，所以存在从派生类型引用到基类类型引用的自动转换，即，可以将派生类对象的引用转换为基类子对象的引用，对指针也类似。

Base-type objects can exist either as independent objects or as part of a derived object. Therefore, a base object might or might not be part of a derived object. As a result, there is no (automatic) conversion from reference (or pointer) to base to reference (or pointer) to derived.

基类类型对象既可以作为独立对象存在，也可以作为派生类对象的一部分而存在，因此，一个基类对象可能是也可能不是一个派生类对象的部分，结果，没有从基类引用（或基类指针）到派生类引用（或派生类指针）的（自动）转换。

The situation with respect to conversions of objects (as opposed to references or pointers) is more complicated. Although we can usually use an object of a derived type to initialize or assign an object of the base type, there is no direct conversion from an object of a derived type to an object of the base type.

相对于引用或指针而言，对象转换的情况更为复杂。虽然一般可以使用派生类型的对象对基类类型的对象进行初始化或赋值，但，没有从派生类型对象到基类类型对象的直接转换。

15.3.1. Derived-to-Base Conversions

15.3.1. 派生类到基类的转换

If we have an object of a derived type, we can use its address to assign or initialize a pointer to the base type. Similarly, we can use a reference or object of the derived type to initialize a reference to the base type. Pedantically speaking, there is no similar conversion for objects. The compiler will not automatically convert an object of derived type into an object of the base type.

如果有一个派生类型的对象，则可以使用它的地址对基类类型的指针进行赋值或初始化。同样，可以使用派生类型的引用或对象初始化基类类型的引用。严格说来，对对象没有类似转换。编译器不会自动将派生类型对象转换为基类类型对象。

It is, however, usually possible to use a derived-type object to initialize or assign an object of base type. The difference between initializing and/or assigning an object and the automatic conversion that is possible for a reference or pointer is subtle and must be well understood.

但是，一般可以使用派生类型对象对基类对象进行赋值或初始化。对对象进行初始化和／或赋值以及可以自动转换引用或指针，这之间的区别是微妙的，必须好好理解。

Conversion to a Reference is Not the Same as Converting an Object

引用转换不同于转换对象

As we've seen, we can pass an object of derived type to a function expecting a reference to base. We might therefore think that the object is converted. However, that is not what happens. When we pass an object to a function expecting a reference, the reference is bound directly to that object. Although it appears that we are passing an object, the argument is actually a reference to that object. The object itself is not copied and

Section 15.3. Conversions and Inheritance

the conversion doesn't change the derived-type object in any way. It remains a derived-type object.

我们已经看到，可以将派生类型的对象传给希望接受基类引用的函数。也许会因此认为对象进行转换，但是，事实并非如此。将对象传给希望接受引用的函数时，引用直接绑定到该对象，虽然看起来在传递对象，实际上实参是该对象的引用，对象本身未被复制，并且，转换不会在任何方面改变派生类型对象，该对象仍是派生类型对象。

When we pass a derived object to a function expecting a base-type object (as opposed to a reference) the situation is quite different. In that case, the parameter's type is fixed both at compile time and run time it will be a base-type object. If we call such a function with a derived-type object, then the base-class portion of that derived object is copied into the parameter.

将派生类对象传给希望接受基类类型对象（而不是引用）的函数时，情况完全不同。在这种情况下，形参的类型是固定的——在编译时和运行时形参都是基类类型对象。如果用派生类型对象调用这样的函数，则该派生类对象的基类部分被复制到形参。

It is important to understand the difference between converting a derived object to a base-type reference and using a derived object to initialize or assign to a base-type object.

一个是派生类对象转换为基类类型引用，一个是用派生类对象对基类对象进行初始化或赋值，理解它们之间的区别很重要。

Using a Derived Object to Initialize or Assign a Base Object

用派生类对象对基类对象进行初始化或赋值

When we initialize or assign an object of base type, we are actually calling a function: When we initialize, we're calling a constructor; when we assign, we're calling an assignment operator.

对基类对象进行初始化或赋值，实际上是在调用函数：初始化时调用构造函数，赋值时调用赋值操作符。

When we use a derived-type object to initialize or assign a base object, there are two possibilities. The first (albeit unlikely) possibility is that the base class might explicitly define what it means to copy or assign an object of the derived type to an object of the base type. It would do so by defining an appropriate constructor or assignment operator:

用派生类对象对基类对象进行初始化或赋值时，有两种可能性。第一种（虽然不太可能的）可能性是，基类可能显式定义了将派生类型对象复制或赋值给基类对象的含义，这可以通过定义适当的构造函数或赋值操作符实现：

```
class Derived;
class Base {
public:
    Base(const Derived&); // create a new Base from a Derived
    Base &operator=(const Derived&); // assign from a Derived
    // ...
};
```

In this case, the definition of these members would control what happens when a `Derived` object is used to initialize or assign to a `Base` object.

在这种情况下，这些成员的定义将控制用 `Derived` 对象对 `Base` 对象进行初始化或赋值时会发生什么。

However, it is uncommon for classes to define explicitly how to initialize or assign an object of the base type from an object of derived type. Instead, base classes usually define (either explicitly or implicitly) their own copy constructor and assignment operator ([Chapter 13](#)). These members take a parameter that is a (`const`) reference to the base type. Because there is a conversion from reference to derived to reference to base, these copy-control members can be used to initialize or assign a base object from a derived object:

然而，类显式定义怎样用派生类型对象对基类类型进行初始化或赋值并不常见，相反，基类一般（显式或隐式地）定义自己的复制构造函数和赋值操作符（[第十三章](#)），这些成员接受一个形参，该形参是基类类型的（`const`）引用。因为存在从派生类引用到基类引用的转换，这些复制控制成员可用于从派生类对象对基类对象进行初始化或赋值：

```
Item_base item; // object of base type
Bulk_item bulk; // object of derived type
// ok: uses Item_base::Item_base(const Item_base&) constructor
Item_base item(bulk); // bulk is "sliced down" to its Item_base portion
// ok: calls Item_base::operator=(const Item_base&)
item = bulk; // bulk is "sliced down" to its Item_base portion
```

When we call the `Item_base` copy constructor or assignment operator on an object of type `Bulk_item`, the following steps happen:

用 `Bulk_item` 类型的对象调用 `Item_base` 类的复制构造函数或赋值操作符时，将发生下列步骤：

- The `Bulk_item` object is converted to a reference to `Item_base`, which means only that an `Item_base` reference is bound to the `Bulk_item` object.
将 `Bulk_item` 对象转换为 `Item_base` 引用，这仅仅意味着将一个 `Item_base` 引用绑定到 `Bulk_item` 对象。
- That reference is passed as an argument to the copy constructor or assignment operator.

Section 15.3. Conversions and Inheritance

将该引用作为实参传给复制构造函数或赋值操作符。

- Those operators use the `Item_base` part of `Bulk_item` to initialize and assign, respectively, the members of the `Item_base` on which the constructor or assignment was called.

那些操作符使用 `Bulk_item` 的 `Item_base` 部分分别对调用构造函数或赋值的 `Item_base` 对象的成员进行初始化或赋值。
- Once the operator completes, the object is an `Item_base`. It contains a copy of the `Item_base` part of the `Bulk_item` from which it was initialized or assigned, but the `Bulk_item` parts of the argument are ignored.

一旦操作符执行完毕，对象即为 `Item_base`。它包含 `Bulk_item` 的 `Item_base` 部分的副本，但实参的 `Bulk_item` 部分被忽略。

In these cases, we say that the `Bulk_item` portion of `bulk` is "sliced down" as part of the initialization or assignment to `item`. An `Item_base` object contains only the members defined in the base class. It does not contain the members defined by any of its derived types. There is no room in an `Item_base` object for the derived members.

在这种情况下，我们说 `bulk` 的 `Bulk_item` 部分在对 `item` 进行初始化或赋值时被“切掉”了。`Item_base` 对象只包含基类中定义的成员，不包含由任意派生类型定义的成员，`Item_base` 对象中没有派生类成员的存储空间。

Accessibility of Derived-to-Base Conversion

派生类到基类转换的可访问性

Like an inherited member function, the conversion from derived to base may or may not be accessible. Whether the conversion is accessible depends on the access label specified on the derived class' derivation.

像继承的成员函数一样，从派生类到基类的转换可能是也可能不是可访问的。转换是否访问取决于在派生类的派生列表中指定的访问标号。



To determine whether the conversion to base is accessible, consider whether a `public` member of the base class would be accessible. If so, the conversion is accessible; otherwise, it is not.

要确定到基类的转换是否可访问，可以考虑基类的 `public` 成员是否访问，如果可以，转换是可访问的，否则，转换是不可访问的。

If the inheritance is `public`, then both user code and member functions of subsequently derived classes may use the derived-to-base conversion. If a class is derived using `private` or `protected` inheritance, then user code may not convert an object of derived type to a base type object. If the inheritance is `private`, then classes derived from the privately inherited class may not convert to the base class. If the inheritance is `protected`, then the members of subsequently derived classes may convert to the base type.

如果是 `public` 继承，则用户代码和后代类都可以使用派生类到基类的转换。如果类是使用 `private` 或 `protected` 继承派生的，则用户代码不能将派生类型对象转换为基类对象。如果是 `private` 继承，则从 `private` 继承类派生的类不能转换为基类。如果是 `protected` 继承，则后续派生类的成员可以转换为基类类型。

Regardless of the derivation access label, a `public` member of the base class is accessible to the derived class itself. Therefore, the derived-to-base conversion is always accessible to the members and friends of the derived class itself.

无论是什么派生访问标号，派生类本身都可以访问基类的 `public` 成员，因此，派生类本身的成员和友元总是可以访问派生类到基类的转换。

15.3.2. Conversions from Base to Derived

15.3.2. 基类到派生类的转换

There is no automatic conversion from the base class to a derived class. We cannot use a base object when a derived object is required:

从基类到派生类的自动转换是不存在的。需要派生类对象时不能使用基类对象：

```
Item_base base;
Bulk_item* bulkP = &base; // error: can't convert base to derived
Bulk_item& bulkRef = base; // error: can't convert base to derived
Bulk_item bulk = base; // error: can't convert base to derived
```

The reason that there is no (automatic) conversion from base type to derived type is that a base object might be just that a base. It does not contain the members of the derived type. If we were allowed to assign a base object to a derived type, then we might attempt to use that derived

Section 15.3. Conversions and Inheritance

object to access members that do not exist.

没有从基类类型到派生类型的（自动）转换，原因在于基类对象只能是基类对象，它不能包含派生类型成员。如果允许用基类对象给派生类型对象赋值，那么就可以试图使用该派生类对象访问不存在的成员。

What is sometimes a bit more surprising is that the restriction on converting from base to derived exists even when a base pointer or reference is actually bound to a derived object:

有时更令人惊讶的是，甚至当基类指针或引用实际绑定到派生类对象时，从基类到派生类的转换也存在限制：

```
Bulk_item bulk;
Item_base *itemP = &bulk; // ok: dynamic type is Bulk_item
Bulk_item *bulkP = itemP; // error: can't convert base to derived
```

The compiler has no way to know at compile time that a specific conversion will actually be safe at run time. The compiler looks only at the static types of the pointer or reference to determine whether a conversion is legal.

编译器在编译时无法知道特定转换在运行时实际上是安全的。编译器确定转换是否合法，只看指针或引用的静态类型。

In those cases when we *know* that the conversion from base to derived is safe, we can use a `static_cast` ([Section 5.12.4](#), p. 183) to override the compiler. Alternatively, we could request a conversion that is checked at run time by using a `dynamic_cast`, which is covered in [Section 18.2.1](#) (p. 773).

在这些情况下，如果知道从基类到派生类的转换是安全的，就可以使用 `static_cast` ([第 5.12.4 节](#)) 强制编译器进行转换。或者，可以用 `dynamic_cast` 申请在运行时进行检查，[第 18.2.1 节](#)将介绍 `dynamic_cast`。

Team LiB

◀ PREVIOUS NEXT ▶

15.4. Constructors and Copy Control

15.4. 构造函数和复制控制

The fact that each derived object consists of the (non`static`) members defined in the derived class plus one or more base-class subobjects affects how derived-type objects are constructed, copied, assigned, and destroyed. When we construct, copy, assign, or destroy an object of derived type, we also construct, copy, assign, or destroy those base-class subobjects.

每个派生类对象由派生类中定义的（非 `static`）成员加上一个或多个基类子对象构成，这一事实影响着派生类型对象时，也会构造、复制、赋值和撤销这些基类子对象。

Constructors and the copy-control members are not inherited; each class defines its own constructor(s) and copy-control members. As is the case for any class, synthesized versions of the default constructor and the copy-control members will be used if the class does not define its own versions.

构造函数和复制控制成员不能继承，每个类定义自己的构造函数和复制控制成员。像任何类一样，如果类不定义自己的默认构造函数和复制控制成员，就将使用合成版本。

15.4.1. Base-Class Constructors and Copy Control

15.4.1. 基类构造函数和复制控制

Constructors and copy control for base classes that are not themselves a derived class are largely unaffected by inheritance. Our `Item_base` constructor looks like many we've seen before:

本身不是派生类的基类，其构造函数和复制控制基本上不受继承影响。构造函数看起来像已经见过的许多构造函数一样：

```
Item_base(const std::string &book = "",
          double sales_price = 0.0):
    isbn(book), price(sales_price) { }
```

The only impact inheritance has on base-class constructors is that there is a new kind of user that must be considered when deciding which constructors to offer. Like any other member, constructors can be made `protected` or `private`. Some classes need special constructors that are intended only for their derived classes to use. Such constructors should be made `protected`.

继承对基类构造函数的唯一影响是，在确定提供哪些构造函数时，必须考虑一类新用户。像任意其他成员一样，构造函数可以为 `protected` 或 `private`，某些类需要只希望派生类使用的特殊构造函数，这样的构造函数应定义为 `protected`。

15.4.2. Derived-Class Constructors

15.4.2. 派生类构造函数

Derived constructors are affected by the fact that they inherit from another class. Each derived constructor initializes its base class in addition to initializing its own data members.

派生类的构造函数受继承关系的影响，每个派生类构造函数除了初始化自己的数据成员之外，还要初始化基类。

The Synthesized Derived-Class Default Constructor

合成的派生类默认构造函数

A derived-class synthesized default constructor ([Section 12.4.3](#), p. 458) differs from a nonderived constructor in only one way: In addition to initializing the data members of the derived class, it also initializes the base part of its object. The base part is initialized by the default constructor of the base class.

派生类的合成默认构造函数（[第 12.4.3 节](#)）与非派生的构造函数只有一点不同：除了初始化派生类的数据成员之外，它还初始化派生类对象的基类部分。基类部分由基类的默认构造函数初始化。

For our `Bulk_item` class, the synthesized default constructor would execute as follows:

Section 15.4. Constructors and Copy Control

对于 `Bulk_item` 类，合成的默认构造函数会这样执行：

1. Invoke the `Item_base` default constructor, which initializes the `isbn` member to the empty string and the `price` member to zero.
调用 `Item_base` 的默认构造函数，将 `isbn` 成员初始化空串，将 `price` 成员初始化为 0。
2. Initialize the members of `Bulk_item` using the normal variable initialization rules, which means that the `qty` and `discount` members would be uninitialized.
用常规变量初始化规则初始化 `Bulk_item` 的成员，也就是说，`qty` 和 `discount` 成员会是未初始化的。

Defining a Default Constructor

定义默认构造函数

Because `Bulk_item` has members of built-in type, we should define our own default constructor:

因为 `Bulk_item` 具有内置类型成员，所以应定义自己的默认构造函数：

```
class Bulk_item : public Item_base {  
public:  
    Bulk_item(): min_qty(0), discount(0.0) { }  
    // as before  
};
```

This constructor uses the constructor initializer list ([Section 7.7.3](#), p. 263) to initialize its `min_qty` and `discount` members. The constructor initializer also implicitly invokes the `Item_base` default constructor to initialize its base-class part.

这个构造函数使用构造函数初始化列表（[第 7.7.3 节](#)）初始化 `min_qty` 和 `discount` 成员，该构造函数还隐式调用 `Item_base` 的默认构造函数初始化对象的基类部分。

The effect of running this constructor is that first the `Item_base` part is initialized using the `Item_base` default constructor. That constructor sets `isbn` to the empty string and `price` to zero. After the `Item_base` constructor finishes, the members of the `Bulk_item` part are initialized, and the (empty) body of the constructor is executed.

运行这个构造函数的效果是，首先使用 `Item_base` 的默认构造函数初始化 `Item_base` 部分，那个构造函数将 `isbn` 置为空串并将 `price` 置为 0。`Item_base` 的构造函数执行完毕后，再初始化 `Bulk_item` 部分的成员并执行构造函数的函数体（函数体为空）。

Passing Arguments to a Base-Class Constructor

向基类构造函数传递实参

In addition to the default constructor, our `Item_base` class lets users initialize the `isbn` and `price` members. We'd like to support the same initialization for `Bulk_item` objects. In fact, we'd like our users to be able to specify values for the entire `Bulk_item`, including the discount rate and quantity.

除了默认构造函数之外，`Item_base` 类还使用户能够初始化 `isbn` 和 `price` 成员，我们希望支持同样 `Bulk_item` 对象的初始化，事实上，我们希望用户能够指定整个 `Bulk_item` 的值，包括折扣率和数量。

The constructor initializer list for a derived-class constructor may initialize only the members of the derived class; it may not directly initialize its inherited members. Instead, a derived constructor indirectly initializes the members it inherits by including its base class in its constructor initializer list:

派生类构造函数的初始化列表只能初始化派生类的成员，不能直接初始化继承成员。相反派生类构造函数通过将基类包含在构造函数初始化列表中来间接初始化继承成员。

```
class Bulk_item : public Item_base {  
public:  
    Bulk_item(const std::string& book, double sales_price,  
              std::size_t qty = 0, double disc_rate = 0.0):  
        Item_base(book, sales_price),  
        min_qty(qty), discount(disc_rate) { }  
    // as before  
};
```

This constructor uses the two-parameter `Item_base` constructor to initialize its base subobject. It passes its own `book` and `sales_price` arguments to that constructor. We might use this constructor as follows:

这个构造函数使用有两个形参 `Item_base` 构造函数初始化基类子对象，它将自己的 `book` 和 `sales_price` 实参传递给该构造函数。这个构造函数可以这样使用：

```
// arguments are the isbn, price, minimum quantity, and discount  
Bulk_item bulk("0-201-82470-1", 50, 5, .19);
```

`bulk` is built by first running the `Item_base` constructor, which initializes `isbn` and `price` from the arguments passed in the `Bulk_item` constructor initializer. After the `Item_base` constructor finishes, the members of `Bulk_item` are initialized. Finally, the (empty) body of the `Bulk_item` constructor is run.

要建立 `bulk`, 首先运行 `Item_base` 构造函数, 该构造函数使用从 `Bulk_item` 构造函数初始化列表传来的实参初始化 `isbn` 和 `price`。`Item_base` 构造函数执行完毕之后, 再初始化 `Bulk_item` 的成员。最后, 运行 `Bulk_item` 构造函数的(空)函数体。



The constructor initializer list supplies initial values for a class' base class and members. It does not specify the order in which those initializations are done. The base class is initialized first and then the members of the derived class are initialized in the order in which they are declared.

构造函数初始化列表为类的基类和成员提供初始值, 它并不指定初始化的执行次序。首先初始化基类, 然后根据声明次序初始化派生类的成员。

Using Default Arguments in a Derived Constructor

在派生类构造函数中使用默认实参

Of course, we might write these two `Bulk_item` constructors as a single constructor that takes default arguments:

当然, 也可以将这两个 `Bulk_item` 构造函数编写为一个接受默认实参的构造函数:

```
class Bulk_item : public Item_base {
public:
    Bulk_item(const std::string& book, double sales_price,
              std::size_t qty = 0, double disc_rate = 0.0):
        Item_base(book, sales_price),
        min_qty(qty), discount(disc_rate) { }
    // as before
};
```

Here we provide defaults for each parameter so that the constructor might be used with zero to four arguments.

这里为每个形参提供了默认值, 因此, 可以用 0 至 4 个实参使用该构造函数。

Only an Immediate Base Class May Be Initialized

只能初始化直接基类

A class may initialize only its own immediate base class. An immediate base class is the class named in the derivation list. If class `C` is derived from class `B`, which is derived from class `A`, then `B` is the immediate base of `C`. Even though every `C` object contains an `A` part, the constructors for `C` may not initialize the `A` part directly. Instead, class `C` initializes `B`, and the constructor for class `B` in turn initializes `A`. The reason for this restriction is that the author of class `B` has specified how to construct and initialize objects of type `B`. As with any user of class `B`, the author of class `C` has no right to change that specification.

一个类只能初始化自己的直接基类。直接就是在派生列表中指定的类。如果类 `C` 从类 `B` 派生, 类 `B` 从类 `A` 派生, 则 `B` 是 `C` 的直接基类。虽然每个 `C` 类对象包含一个 `A` 类部分, 但 `C` 的构造函数不能直接初始化 `A` 部分。相反, 需要类 `C` 初始化类 `B`, 而类 `B` 的构造函数再初始化类 `A`。这一限制的原因是, 类 `B` 的作者已经指定了怎样构造和初始化 `B` 类型的对象。像类 `B` 的任何用户一样, 类 `C` 的作者无权改变这个规约。

As a more concrete example, our bookstore might have several discount strategies. In addition to a bulk discount, it might offer a discount for purchases up to a certain quantity and then charge the full price thereafter. Or it might offer a discount for purchases above a certain limit but not for purchases up to that limit.

作为更具体的例子, 书店可以有几种折扣策略。除了批量折扣外, 还可以为购买某个数量打折, 此后按全价销售, 或者, 购买量超过一定限度的可以打折, 在该限度之内不打折。

Each of these discount strategies is the same in that it requires a quantity and a discount amount. We might support these differing strategies by defining a new class named `Disc_item` to store the quantity and the discount amount. This class would not define a `net_price` function but would serve as a base class for classes such as `Bulk_item` that define the different discount strategies.

这些折扣策略都需要一个数量和一个折扣量, 可以定义名为 `Disc_item` 的新类存储数量和折扣量, 以支持这些不同的折扣策略。`Disc_item` 类可以不定义 `net_price` 函数, 但可以作为定义不同折扣策略的其他类(如 `Bulk_item` 类)的基类。

Key Concept: Refactoring

关键概念：重构

Adding `Disc_item` to the `Item_base` hierarchy is an example of refactoring. Refactoring involves redesigning a class hierarchy to move operations and/or data from one class to another. Refactoring happens most often when classes are redesigned to add new functionality or handle other changes in that application's requirements.

将 `Disc_item` 加到 `Item_base` 层次是重构（refactoring）的一个例子。重构包括重新定义类层次，将操作和／或数据从一个类移到另一个类。为了适应应用程序的需要而重新设计类以便增加新函数或处理其他改变时，最有可能需要进行重构。

Refactoring is common in OO applications. It is noteworthy that even though we changed the inheritance hierarchy, code that uses the `Bulk_item` or `Item_base` classes would not need to change. However, when classes are refactored, or changed in any other way, any code that uses those classes must be recompiled.

重构常见在面向对象应用程序中非常常见。值得注意的是，虽然改变了继承层次，使用 `Bulk_item` 类或 `Item_base` 类的代码不需要改变。然而，对类进行重构，或以任意其他方式改变类，使用这些类的任意代码都必须重新编译。

To implement this design, we first need to define the `Disc_item` class:

要实现这个设计，首先需要定义 `Disc_item` 类：

```
// class to hold discount rate and quantity
// derived classes will implement pricing strategies using these data
class Disc_item : public Item_base {
public:
    Disc_item(const std::string& book = "",
              double sales_price = 0.0,
              std::size_t qty = 0, double disc_rate = 0.0):
        Item_base(book, sales_price),
        quantity(qty), discount(disc_rate) { }
protected:
    std::size_t quantity; // purchase size for discount to apply
    double discount;      // fractional discount to apply
};
```

This class inherits from `Item_base` and defines its own members, `discount` and `quantity`. Its only member function is the constructor, which initializes its `Item_base` base class and the members defined by `Disc_item`.

这个类继承 `Item_base` 类并定义了自己的 `discount` 和 `quantity` 成员。它唯一的成员函数是构造函数，用以初始化基类和 `Disc_item` 定义的成员。

Next, we can reimplement `Bulk_item` to inherit from `Disc_item`, rather than inheriting directly from `Item_base`:

其次，可以重新实现 `Bulk_item` 以继承 `Disc_item`，而不再直接继承 `Item_base`：

```
// discount kicks in when a specified number of copies of same book are sold
// the discount is expressed as a fraction to use to reduce the normal price
class Bulk_item : public Disc_item {
public:
    Bulk_item(const std::string& book = "",
              double sales_price = 0.0,
              std::size_t qty = 0, double disc_rate = 0.0):
        Disc_item(book, sales_price, qty, disc_rate) { }
    // redefines base version so as to implement bulk purchase discount policy
    double net_price(std::size_t) const;
};
```

The `Bulk_item` class now has a **direct base class**, `Disc_item`, and an indirect base class, `Item_base`. Each `Bulk_item` object has three subobjects: an (empty) `Bulk_item` part and a `Disc_item` subobject, which in turn has an `Item_base` base subobject.

`Bulk_item` 类现在有一个直接基类 `Disc_item`，还有一个间接基类 `Item_base`。每个 `Bulk_item` 对象有三个子对象：一个（空的）`Bulk_item` 部分和一个 `Disc_item` 子对象，`Disc_item` 子对象又有一个 `Item_base` 基类子对象。

Even though `Bulk_item` has no data members of its own, it defines a constructor in order to obtain values to use to initialize its inherited members.

虽然 `Bulk_item` 没有自己的数据成员，但为获取值用来初始化其继承成员，它定义了一个构造函数。

A derived constructor may initialize only its immediate base class. Naming `Item_base` in the `Bulk_item` constructor initializer would be an error.

派生类构造函数只能初始化自己的直接基类，在 `Bulk_item` 类的构造函数初始化列表中指定 `Item_base` 是一个错误。

Key Concept: Respecting the Base-Class Interface

关键概念：尊重基类接口

The reason that a constructor can initialize only its immediate base class is that each class defines its own interface. When we define `Disc_item`, we specify how to initialize a `Disc_item` by defining its constructors. Once a class has defined its interface, all interactions with objects of that class should be through that interface, even when those objects are part of a derived object.

构造函数只能初始化其直接基类的原因是每个类都定义了自己的接口。定义 `Disc_item` 时，通过定义它的构造函数指定了怎样初始化 `Disc_item` 对象。一旦类定义了自己的接口，与该类对象的所有交互都应该通过该接口，即使对象是派生类对象的一部分也不例外。

For similar reasons, derived-class constructors may not initialize and should not assign to the members of its base class. When those members are `public` or `protected`, a derived constructor could assign values to its base class members inside the constructor body. However, doing so would violate the interface of the base. Derived classes should respect the initialization intent of their base classes by using constructors rather than assigning to these members in the body of the constructor.

同样，派生类构造函数不能初始化基类的成员且不应该对基类成员赋值。如果那些成员为 `public` 或 `protected`，派生构造函数可以在构造函数函数体中给基类成员赋值，但是，这样做会违反基类的接口。派生类应通过使用基类构造函数尊重基类的初始化意图，而不是在派生类构造函数函数体中对这些成员赋值。

Exercises Section 15.4.2

Exercise 15.14: Redefine the `Bulk_item` and `Item_base` classes so that they each need to define only a single constructor.

重新定义 `Bulk_item` 和 `Item_base` 类，使每个类只需定义一个构造函数。

Exercise 15.15: Identify the base- and derived-class constructors for the library class hierarchy described in the first exercise on page [575](#).

对于第 [15.2.5 节](#) 习题第一题中描述的图书馆类层次，识别基类和派生类构造函数。

Exercise 15.16: Given the following base class definition,

对于下面的基类定义：

```
struct Base {
    Base(int val): id(val) { }
protected:
    int id;
};
```

explain why each of the following constructors is illegal.

解释为什么下述每个构造函数是非法的。

```
(a) struct C1 : public Base {
    C1(int val): id(val) { }
};

(b) struct C2 : public
    C1 { C2(int val): Base(val), C1(val){ }
};

(c) struct C3 : public
    C1 { C3(int val): Base(val) { }
};

(d) struct C4 : public Base {
    C4(int val) : Base(id + val){ }
};

(e) struct C5 : public Base {
    C5() { }
};
```

15.4.3. Copy Control and Inheritance

15.4.3. 复制控制和继承

Like any other class, a derived class may use the synthesized copy-control members described in [Chapter 13](#). The synthesized operations copy, assign, or destroy the base-class part of the object along with the members of the derived part. The base part is copied, assigned, or destroyed by using the base class' copy constructor, assignment operator, or destructor.

像任意其他类一样，派生类也可以使用[第十三章](#)所介绍的合成复制控制成员。合成操作对对象的基类部分连同派生部分的成员一起进行复制、赋值或撤销，使用基类的复制构造函数、赋值操作符或析构函数对基类部分进行复制、赋值或撤销。

Whether a class needs to define the copy-control members depends entirely on the class' own direct members. A base class might define its own copy control while the derived uses the synthesized versions or vice versa.

类是否需要定义复制控制成员完全取决于类自身的直接成员。基类可以定义自己的复制控制而派生类使用合成版本，反之亦然。

Classes that contain only data members of class type or built-in types other than pointers usually can use the synthesized operations. No special control is required to copy, assign, or destroy such members. Classes with pointer members often need to define their own copy control to manage these members.

只包含类型或内置类型数据成员、不含指针的类一般可以使用合成操作，复制、赋值或撤销这样的成员不需要特殊控制。具有指针成员的类一般需要定义自己的复制控制来管理这些成员。

Our `Item_base` class and its derived classes can use the synthesized versions of the copy-control operations. When a `Bulk_item` is copied, the (synthesized) copy constructor for `Item_base` is invoked to copy the `isbn` and `price` members. The `isbn` member is copied by using the `string` copy constructor; the `price` member is copied directly. Once the base part is copied, then the derived part is copied. Both members of `Bulk_item` are `doubles`, and these members are copied directly. The assignment operator and destructor are handled similarly.

`Item_base` 类及其派生类可以使用复制控制操作的合成版本。复制 `Bulk_item` 对象时，调用（合成的）`Item_base` 复制构造函数复制 `isbn` 和 `price` 成员。使用 `string` 复制构造函数复制 `isbn`，直接复制 `price` 成员。一旦复制了基类部分，就复制派生部分。`Bulk_item` 的两个成员都是 `double` 型，直接复制这些成员。赋值操作符和析构函数类似处理。

Defining a Derived Copy Constructor

定义派生类复制构造函数



If a derived class explicitly defines its own copy constructor or assignment operator, that definition completely overrides the defaults. The copy constructor and assignment operator for inherited classes are responsible for copying or assigning their base-class components as well as the members in the class itself.

如果派生类显式定义自己的复制构造函数或赋值操作符，则该定义将完全覆盖默认定义。被继承类的复制构造函数和赋值操作符负责对基类成分以及类自己的成员进行复制或赋值。

If a derived class defines its own copy constructor, that copy constructor usually should explicitly use the base-class copy constructor to initialize the base part of the object:

如果派生类定义了自己的复制构造函数，该复制构造函数一般应显式使用基类复制构造函数初始化对象的基类部分：

```
class Base { /* ... */;
class Derived: public Base {
public:
    // Base::Base(const Base&) not invoked automatically
    Derived(const Derived& d):
        Base(d) /* other member initialization */ { /*... */ }
};
```

The initializer `Base(d)` converts ([Section 15.3](#), p. 577) the derived object, `d`, to a reference to its base part and invokes the base-class copy constructor. Had the initializer for the base class been omitted,

初始化函数 `Base(d)` 将派生类对象 `d` 转换（[第 15.3 节](#)）为它的基类部分的引用，并调用基类复制构造函数。如果省略基类初始化函数，如下代码：

```
// probably incorrect definition of the Derived copy constructor
Derived(const Derived& d) /* derived member initializations */
    {/* ... */}
```

Section 15.4. Constructors and Copy Control

the effect would be to run the `Base` default constructor to initialize the base part of the object. Assuming that the initialization of the `Derived` members copied the corresponding elements from `a`, then the newly constructed object would be oddly configured: Its `Base` part would hold default values, while its `Derived` members would be copies of another object.

效果是运行 `Base` 的默认构造函数初始化对象的基类部分。假定 `Derived` 成员的初始化从 `a` 复制对应成员，则新构造的对象将具有奇怪的配置：它的 `Base` 部分将保存默认值，而它的 `Derived` 成员是另一对象的副本。

Derived-Class Assignment Operator

派生类赋值操作符

As usual, the assignment operator is similar to the copy constructor: If the derived class defines its own assignment operator, then that operator must assign the base part explicitly:

赋值操作符通常与复制构造函数类似：如果派生类定义了自己的赋值操作符，则该操作符必须对基类部分进行显式赋值。

```
// Base::operator=(const Base&) not invoked automatically
Derived &Derived::operator=(const Derived &rhs)
{
    if (this != &rhs) {
        Base::operator=(rhs); // assigns the base part
        // do whatever needed to clean up the old value in the derived part
        // assign the members from the derived
    }
    return *this;
}
```

The assignment operator must, as always, guard against self-assignment. Assuming the left- and right-hand operands differ, then we call the `Base` class assignment operator to assign the base-class portion. That operator might be defined by the class or it might be the synthesized assignment operator. It doesn't matter we can call it directly. The base-class operator will free the old value in the base part of the left-hand operand and will assign the new values from `rhs`. Once that operator finishes, we continue doing whatever is needed to assign the members in the derived class.

赋值操作符必须防止自身赋值。假定左右操作数不同，则调用 `Base` 类的赋值操作符给基类部分赋值。该操作符可以由类定义，也可以是合成赋值操作符，这没什么关系——我们可以直接调用它。基类操作符将释放左操作数中基类部分的值，并赋以来自 `rhs` 的新值。该操作符执行完毕后，接着要做的是为派生类中的成员赋值。

Derived-Class Destructor

派生类析构函数

The destructor works differently from the copy constructor and assignment operator: The derived destructor is never responsible for destroying the members of its base objects. The compiler always implicitly invokes the destructor for the base part of a derived object. Each destructor does only what is necessary to clean up its own members:

析构函数的工作与复制构造函数和赋值操作符不同：派生类析构函数不负责撤销基类对象的成员。编译器总是显式调用派生类对象基类部分的析构函数。每个析构函数只负责清除自己的成员：

```
class Derived: public Base {
public:
    // Base::~Base invoked automatically
    ~Derived()    { /* do what it takes to clean up derived members */ }
};
```

Objects are destroyed in the opposite order from which they are constructed: The derived destructor is run first, and then the base-class destructors are invoked, walking back up the inheritance hierarchy.

对象的撤销顺序与构造顺序相反：首先运行派生析构函数，然后按继承层次依次向上调用各基类析构函数。

15.4.4. Virtual Destructors

15.4.4. 虚析构函数

The fact that destructors for the base parts are invoked automatically has an important consequence for the design of base classes.

自动调用基类部分的析构函数对基类的设计有重要影响。

Section 15.4. Constructors and Copy Control

When we `delete` a pointer that points to a dynamically allocated object, the destructor is run to clean up the object before the memory for that object is freed. When dealing with objects in an inheritance hierarchy, it is possible that the static type of the pointer might differ from the dynamic type of the object that is being deleted. We might `delete` a pointer to the base type that actually points to a derived object.

删除指向动态分配对象的指针时，需要运行析构函数在释放对象的内存之前清除对象。处理继承层次中的对象时，指针的静态类型可能与被删除对象的动态类型不同，可能会删除实际指向派生类对象的基类类型指针。

If we `delete` a pointer to base, then the base-class destructor is run and the members of the base are cleaned up. If the object really is a derived type, then the behavior is undefined. To ensure that the proper destructor is run, the destructor must be virtual in the base class:

如果删除基类指针，则需要运行基类析构函数并清除基类的成员，如果对象实际是派生类型的，则没有定义该行为。要保证运行适当的析构函数，基类中的析构函数必须为虚函数：

```
class Item_base {
public:
    // no work, but virtual destructor needed
    // if base pointer that points to a derived object is ever deleted
    virtual ~Item_base() { }
};
```

If the destructor is virtual, then when it is invoked through a pointer, which destructor is run will vary depending on the type of the object to which the pointer points:

如果析构函数为虚函数，那么通过指针调用时，运行哪个析构函数将因指针所指对象类型的不同而不同：

```
Item_base *itemP = new Item_base; // same static and dynamic type
delete itemP;                  // ok: destructor for Item_base called
itemP = new Bulk_item;          // ok: static and dynamic types differ
delete itemP;                  // ok: destructor for Bulk_item called
```

Like other virtual functions, the virtual nature of the destructor is inherited. Therefore, if the destructor in the root class of the hierarchy is virtual, then the derived destructors will be virtual as well. A derived destructor will be virtual whether the class explicitly defines its destructor or uses the synthesized destructor.

像其他虚函数一样，析构函数的虚函数性质都将继承。因此，如果层次中根类的析构函数为虚函数，则派生类析构函数也将是虚函数，无论派生类显式定义析构函数还是使用合成析构函数，派生类析构函数都是虚函数。

Destructors for base classes are an important exception to the Rule of Three ([Section 13.3](#), p. 485). That rule says that if a class needs a destructor, then the class almost surely needs the other copy-control members. A base class almost always needs a destructor so that it can make the destructor virtual. If a base class has an empty destructor in order to make it virtual, then the fact that the class has a destructor is not an indication that the assignment operator or copy constructor is also needed.

基类析构函数是三法则（[第 13.3 节](#)）的一个重要例外。三法则指出，如果类需要析构函数，则类几乎也确实需要其他复制控制成员。基类几乎总是需要构造函数，从而可以将析构函数设为虚函数。如果基类为了将析构函数设为虚函数则具有空析构函数，那么，类具有析构函数并不表示也需要赋值操作符或复制构造函数。



The root class of an inheritance hierarchy should define a virtual destructor even if the destructor has no work to do.

即使析构函数没有工作要做，继承层次的根类也应该定义一个虚析构函数。

Constructors and Assignment Are Not Virtual

构造函数和赋值操作符不是虚函数

Of the copy-control members, only the destructor should be defined as virtual. Constructors cannot be defined as virtual. Constructors are run before the object is fully constructed. While the constructor is running, the object's dynamic type is not complete.

在复制控制成员中，只有析构函数应定义为虚函数，构造函数不能定义为虚函数。构造函数是在对象完全构造之前运行的，在构造函数运行的时候，对象的动态类型还不完整。

Although we can define a virtual `operator=` member function in the base class, doing so does not affect the assignment operators used in the derived classes. Each class has its own assignment operator. The assignment operator in a derived class has a parameter that has the same type as the class itself. That type must differ from the parameter type for the assignment operator in any other class in the hierarchy.

Section 15.4. Constructors and Copy Control

虽然可以在基类中将成员函数 `operator=` 定义为虚函数，但这样做并不影响派生类中使用的赋值操作符。每个类有自己的赋值操作符，派生类中的赋值操作符有一个与类本身类型相同的形参，该类型必须不同于继承层次中任意其他类的赋值操作符的形参类型。

Making the assignment operator virtual is likely to be confusing because a virtual function must have the same parameter type in base and derived classes. The base-class assignment operator has a parameter that is a reference to its own class type. If that operator is virtual, then each class gets a virtual member that defines an `operator=` that takes a base object. But this operator is not the same as the assignment operator for the derived class.

将赋值操作符设为虚函数可能会令人混淆，因为虚函数必须在基类和派生类中具有同样的形参。基类赋值操作符有一个形参是自身类类型的引用，如果该操作符为虚函数，则每个类都将得到一个虚函数成员，该成员定义了参数为一个基类对象的 `operator=`。但是，对派生类而言，这个操作符与赋值操作符是不同的。



Making the class assignment operator virtual is likely to be confusing and unlikely to be useful.

将类的赋值操作符设为虚函数很可能会令人混淆，而且不会有用处。

Exercises Section 15.4.4

Exercise

15.17: Describe the conditions under which a class should have a virtual destructor.

说明在什么情况下类应该具有虚析构函数。

Exercise

15.18: What operations must a virtual destructor perform?

虚析构函数必须执行什么操作？

Exercise

15.19: What if anything is likely to be incorrect about this class definition?

如果这个类定义有错，可能是什么错？

[\[View full width\]](#)

```
class AbstractObject {
public:
    virtual void doit();
    // other members not including any of the copy-control
    ↪ functions
};
```

Exercise

15.20: Recalling the exercise from [Section 13.3](#) (p. 487) in which you wrote a class whose copy-control members printed a message, add print statements to the constructors of the `Item_base` and `Bulk_item` classes. Define the copy-control members to do the same job as the synthesized versions but that also print a message. Now write programs using objects and functions that use the `Item_base` types. In each case, predict what objects will be created and destroyed and compare your predictions with what your programs generate. Continue experimenting until you can correctly predict which copy-control members are executed for a given bit of code.

回忆在[第 13.3 节](#)习题中编写的类，该类的复制控制成员打印一条消息，为 `Item_base` 和 `Bulk_item` 类的构造函数增加打印语句。定义复制控制成员，使之完成与合成版本相同的工作外，还打印一条消息。应用使用了 `Item_base` 类型的那些对象和函数编写一些程序，在每种情况下，预测将会创建和撤销什么对象，并将你的预测与程序所产生的结果进行比较。继续实验，直至你能够正确地预测对于给定的代码片段，会执行哪些复制控制成员。

15.4.5. Virtuals in Constructors and Destructors

15.4.5. 构造函数和析构函数中的虚函数

A derived object is constructed by first running a base-class constructor to initialize the base part of the object. While the base-class constructor is executing, the derived part of the object is uninitialized. In effect, the object is not yet a derived object.

构造派生类对象时首先运行基类构造函数初始化对象的基类部分。在执行基类构造函数时，对象的派生类部分是未初始化的。实际上，此时对象还不是一个派生类对象。

When a derived object is destroyed, its derived part is destroyed first, and then its base parts are destroyed in the reverse order of how they were constructed.

撤销派生类对象时，首先撤销它的派生类部分，然后按照与构造顺序的逆序撤销它的基类部分。

In both cases, while a constructor or destructor is running, the object is incomplete. To accommodate this incompleteness, the compiler treats the object as if its type changes during construction or destruction. Inside a base-class constructor or destructor, a derived object is treated as if it were an object of the base type.

在这两种情况下，运行构造函数或析构函数的时候，对象都是不完整的。为了适应这种不完整，编译器将对象的类型视为在构造或析构期间发生了变化。在基类构造函数或析构函数中，将派生类对象当作基类类型对象对待。

The type of an object during construction and destruction affects the binding of virtual functions.

构造或析构期间的对象类型对虚函数的绑定有影响。



If a virtual is called from inside a constructor or destructor, then the version that is run is the one defined for the type of the constructor or destructor itself.

如果在构造函数或析构函数中调用虚函数，则运行的是为构造函数或析构函数自身类型定义的版本。

This binding applies to a virtual whether the virtual is called directly by the constructor (or destructor) or is called indirectly from a function that the constructor (or destructor) called.

无论由构造函数（或析构函数）直接调用虚函数，或者从构造函数（或析构函数）所调用的函数间接调用虚函数，都应用这种绑定。

To understand this behavior, consider what would happen if the derived-class version of a virtual function were called from a base-class constructor (or destructor). The derived version of the virtual probably accesses members of the derived object. After all, if the derived-class version didn't need to use members from the derived object, the derived class could probably use the definition from the base class. However, the members of the derived part of the object aren't initialized while the base constructor (or destructor) is running. In practice, if such access were allowed, the program would probably crash.

要理解这种行为，考虑如果从基类构造函数（或析构函数）调用虚函数的派生类版本会怎么样。虚函数的派生类版本很可能会访问派生类对象的成员，毕竟，如果派生类版本不需要使用派生类对象的成员，派生类多半能够使用基类中的定义。但是，对象的派生部分的成员不会在基类构造函数运行期间初始化，实际上，如果允许这样的访问，程序很可能崩溃。

15.5. Class Scope under Inheritance

15.5. 继承情况下的类作用域

Each class maintains its own scope ([Section 12.3](#), p. 444) within which the names of its members are defined. Under inheritance, the scope of the derived class is nested within the scope of its base classes. If a name is unresolved within the scope of the derived class, the enclosing base-class scope(s) are searched for a definition of that name.

每个类都保持着自己的作用域（[第 12.3 节](#)），在该作用域中定义了成员的名字。在继承情况下，派生类的作用域嵌套在基类作用域中。如果不能在派生类作用域中确定名字，就在外围基类作用域中查找该名字的定义。

It is this hierarchical nesting of class scopes under inheritance that allows the members of the base class to be accessed directly as if they are members of the derived class. When we write

正是这种类作用域的层次嵌套使我们能够直接访问基类的成员，就好象这些成员是派生类成员一样。如果编写如下代码：

```
Bulk_item bulk;
cout << bulk.book();
```

the use of the name `book` is resolved as follows:

名字 `book` 的使用将这样确定：

1. `bulk` is an object of the `Bulk_item` class. The `Bulk_item` class is searched for `book`. That name is not found.

`bulk` 是 `Bulk_item` 类对象，在 `Bulk_item` 类中查找，找不到名字 `book`。

2. Because `Bulk_item` is derived from `Item_Base`, the `Item_Base` class is searched next. The name `book` is found in the `Item_base` class. The reference is resolved successfully.

因为从 `Item_base` 派生 `Bulk_item`，所以接着在 `Item_base` 类中查找，找到名字 `book`，引用成功地确定了。

15.5.1. Name Lookup Happens at Compile Time

15.5.1. 名字查找在编译时发生

The static type of an object, reference, or pointer determines the actions that the object can perform. Even when the static and dynamic types might differ, as can happen when a reference or pointer to a base type is used, the static type determines what members can be used. As an example, we might add a member to the `Disc_item` class that returns a `pair` holding the minimum (or maximum) quantity and the discounted price:

对象、引用或指针的静态类型决定了对象能够完成的行为。甚至当静态类型和动态类型可能不同的时候，就像使用基类类型的引用或指针时可能会发生的，静态类型仍然决定着可以使用什么成员。例如，可以给 `Disc_item` 类增加一个成员，该成员返回一个保存最小（或最大）数量和折扣价格的 `pair` 对象：

```
class Disc_item : public Item_base {
public:
    std::pair<size_t, double> discount_policy() const
        { return std::make_pair(quantity, discount); }
    // other members as before
};
```

We can access `discount_policy` only through an object, pointer, or reference of type `Disc_item` or a class derived from `Disc_item`:

只能通过 `Disc_item` 类型或 `Disc_item` 派生类型的对象、指针或引用访问 `discount_policy`：

```
Bulk_item bulk;
Bulk_item *bulkP = &bulk; // ok: static and dynamic types are the same
Item_base *itemP = &bulk; // ok: static and dynamic types differ
bulkP->discount_policy(); // ok: bulkP has type Bulk_item*
itemP->discount_policy(); // error: itemP has type Item_base*
```

The call through `itemP` is an error because a pointer (reference or object) to a base type can access only the base parts of an object and there is

Section 15.5. Class Scope under Inheritance

no `discount_policy` member defined in the base class.

重新定义 `itemP` 的访问是错误的，因为基类类型的指针（引用或对象）只能访问对象的基类部分，而在基类中没有定义 `discount_policy` 成员。

Exercises Section 15.5.1

Exercise 15.21: Redefine your `Item_base` hierarchy to include a `Disc_item` class.

重新定义 `Item_base` 层次以包含 `Disc_item` 类。

Exercise 15.22: Redefine `Bulk_item` and the class you implemented in the exercises from [Section 15.2.3](#) (p. 567) that represents a limited discount strategy to inherit from `Disc_item`.

重新定义 `Bulk_item` 和你在 [第 15.2.3 节](#) 习题中实现的那个继承 `Disc_item` 的表示有限折扣策略的类。

15.5.2. Name Collisions and Inheritance

15.5.2. 名字冲突与继承

Although a base-class member can be accessed directly as if it were a member of the derived class, the member retains its base-class membership. Normally we do not care which actual class contains the member. We usually need to care only when a base- and derived-class member share the same name.

虽然可以直接访问基类成员，就像它是派生类成员一样，但是成员保留了它的基类成员资格。一般我们并不关心是哪个实际类包含成员，通常只在基类和派生类共享同一名字时才需要注意。



A derived-class member with the same name as a member of the base class hides direct access to the base-class member.

与基类成员同名的派生类成员将屏蔽对基类成员的直接访问。

```
struct Base {
    Base(): mem(0) { }
protected:
    int mem;
};

struct Derived : Base {
    Derived(int i): mem(i) { } // initializes Derived::mem
    int get_mem() { return mem; } // returns Derived::mem
protected:
    int mem; // hides mem in the base
};
```

The reference to `mem` inside `get_mem` is resolved to the name inside `Derived`. Were we to write

`get_mem` 中对 `mem` 的引用被确定为使用 `Derived` 中的名字。如果编写如下代码：

```
Derived d(42);
cout << d.get_mem() << endl; // prints 42
```

then the output would be `42`.

则输出将是 `42`。

Using the Scope Operator to Access Hidden Members

Section 15.5. Class Scope under Inheritance

使用作用域操作符访问被屏蔽成员

We can access a hidden base-class member by using the scope operator:

可以使用作用域操作符访问被屏蔽的基类成员：

```
struct Derived : Base {  
    int get_base_mem() { return Base::mem; }  
};
```

The scope operator directs the compiler to look for `mem` starting in `Base`.

作用域操作符指示编译器在 `Base` 中查找 `mem`。



When designing a derived class, it is best to avoid name collisions with members of the base class whenever possible.

设计派生类时，只要可能，最好避免与基类成员的名字冲突。

Exercises Section 15.5.2

Exercise Given the following base- and derived-class definitions

15.23:

对于下面的基类和派生类定义：

```
struct Base {  
    foo(int);  
protected:  
    int bar;  
    double foo_bar;  
};  
  
struct Derived : public Base {  
    foo(string);  
    bool bar(Base *pb);  
    void foobar();  
protected:  
    string bar;  
};
```

identify the errors in each of the following examples and how each might be fixed:

找出下述每个例子中的错误并说明怎样改正：

```
(a) Derived d; d.foo(1024);  
(b) void Derived::foobar() { bar = 1024; }  
(c) bool Derived::bar(Base *pb)  
     { return foo_bar == pb->foo_bar; }
```

15.5.3. Scope and Member Functions

15.5.3. 作用域与成员函数

A member function with the same name in the base and derived class behaves the same way as a data member: The derived-class member hides the base-class member within the scope of the derived class. The base member is hidden, even if the prototypes of the functions differ:

在基类和派生类中使用同名的成员函数，其行为与数据成员一样：在派生类作用域中派生类成员将屏蔽基类成员。即使函数原型不同，基类成员也会被屏蔽：

```
struct Base {
```

Section 15.5. Class Scope under Inheritance

```
int memfcn();  
};  
struct Derived : Base {  
    int memfcn(int); // hides memfcn in the base  
};  
Derived d; Base b;  
b.memfcn();      // calls Base::memfcn  
d.memfcn(10);    // calls Derived::memfcn  
d.memfcn();      // error: memfcn with no arguments is hidden  
d.Base::memfcn(); // ok: calls Base::memfcn
```

The declaration of `memfcn` in `Derived` hides the declaration in `Base`. Not surprisingly, the first call through `b`, which is a `Base` object, calls the version in the base class. Similarly, the second call through `d` calls the one from `Derived`. What can be surprising is the third call:

`Derived` 中的 `memfcn` 声明隐藏了 `Base` 中的声明。这并不奇怪，第一个调用通过 `Base` 对象 `b` 调用基类中的版本，同样，第二个调用通过 `d` 调用 `Derived` 中的版本。可能比较奇怪的是第三个调用：

```
d.memfcn(); // error: Derived has no memfcn that takes no arguments
```

To resolve this call, the compiler looks for the name `memfcn`, which it finds in the class `Derived`. Once the name is found, the compiler looks no further. This call does not match the definition of `memfcn` in `Derived`, which expects an `int` argument. The call provides no such argument and so is in error.

要确定这个调用，编译器需要查找名字 `memfcn`，并在 `Derived` 类中找到。一旦找到了名字，编译器就不再继续查找了。这个调用与 `Derived` 中的 `memfcn` 定义不匹配，该定义希望接受 `int` 实参，而这个函数调用没有提供那样的实参，因此出错。



Recall that functions declared in a local scope do not overload functions defined at global scope ([Section 7.8.1](#), p. 268). Similarly, functions defined in a derived class do *not* overload members defined in the base. When the function is called through a derived object, the arguments must match a version of the function defined in the derived class. The base class functions are considered only if the derived does not define the function at all.

回忆一下，局部作用域中声明的函数不会重载全局作用域中定义的函数（[第 7.8.1 节](#)），同样，派生类中定义的函数也不重载基类中定义的成员。通过派生类对象调用函数时，实参必须与派生类中定义的版本相匹配，只有在派生类根本没有定义该函数时，才考虑基类函数。

Overloaded Functions

重载函数

As with any other function, a member function (virtual or otherwise) can be over-loaded. A derived class can redefine zero or more of the versions it inherits.

像其他任意函数一样，成员函数（无论虚还是非虚）也可以重载。派生类可以重定义所继承的 0 个或多个版本。



If the derived class redefines any of the overloaded members, then only the one(s) redefined in the derived class are accessible through the derived type.

如果派生类重定义了重载成员，则通过派生类型只能访问派生类中重定义的那些成员。

If a derived class wants to make all the overloaded versions available through its type, then it must either redefine all of them or none of them.

如果派生类想通过自身类型使用的重载版本，则派生类必须要重定义所有重载版本，要么一个也不重定义。

Sometimes a class needs to redefine the behavior of only some of the versions in an overloaded set, and wants to inherit the meaning for others. It would be tedious in such cases to have to redefine every base-class version in order to redefine the ones that the class needs to specialize.

有时类需要仅仅重定义一个重载集中某些版本的行为，并且想要继承其他版本的含义，在这种情况下，为了重定义需要特化的某个版本而不得不重定义每一个基类版本，可能会令人厌烦。

Section 15.5. Class Scope under Inheritance

Instead of redefining every base-class version that it inherits, a derived class can provide a `using` declaration ([Section 15.2.5, p. 574](#)) for the overloaded member. A `using` declaration specifies only a name; it may not specify a parameter list. Thus, a `using` declaration for a base-class member function name adds all the overloaded instances of that function to the scope of the derived-class. Having brought all the names into its scope, the derived class need redefine only those functions that it truly must define for its type. It can use the inherited definitions for the others.

派生类不用重定义所继承的每一个基类版本，它可以为重载成员提供 `using` 声明 ([第 15.2.5 节](#))。一个 `using` 声明只能指定一个名字，不能指定形参表，因此，为基类成员函数名称而作的 `using` 声明将该函数的所有重载实例加到派生类的作用域。将所有名字加入作用域之后，派生类只需要重定义本类型确实必须定义的那些函数，对其他版本可以使用继承的定义。

15.5.4. Virtual Functions and Scope

15.5.4. 虚函数与作用域

Recall that to obtain dynamic binding, we must call a virtual member through a reference or a pointer to a base class. When we do so, the compiler looks for the function in the base class. Assuming the name is found, the compiler checks that the arguments match the parameters.

还记得吗，要获得动态绑定，必须通过基类的引用或指针调用虚成员。当我们这样做时，编译器将在基类中查找函数。假定找到了名字，编译器就检查实参是否与形参匹配。

We can now understand why virtual functions must have the same prototype in the base and derived classes. If the base member took different arguments than the derived-class member, there would be no way to call the derived function from a reference or pointer to the base type. Consider the following (artificial) collection of classes:

现在可以理解虚函数为什么必须在基类和派生类中拥有同一原型了。如果基类成员与派生类成员接受的实参不同，就没有办法通过基类类型的引用或指针调用派生类函数。考虑如下（人为的）为集合：

```
class Base {
public:
    virtual int fcn();
};

class D1 : public Base {
public:
    // hides fcn in the base; this fcn is not virtual
    int fcn(int); // parameter list differs from fcn in Base
    // D1 inherits definition of Base::fcn()
};

class D2 : public D1 {
public:
    int fcn(int); // nonvirtual function hides D1::fcn(int)
    int fcn();    // redefines virtual fcn from Base
};
```

The version of `fcn` in `D1` does not redefine the virtual `fcn` from `Base`. Instead, it hides `fcn` from the base. Effectively, `D1` has two functions named `fcn`: The class inherits a virtual named `fcn` from the `Base` and defines its own, nonvirtual member named `fcn` that takes an `int` parameter. However, the virtual from the `Base` cannot be called from a `D1` object (or reference or pointer to `D1`) because that function is hidden by the definition of `fcn(int)`.

`D1` 中的 `fcn` 版本没有重定义 `Base` 的虚函数 `fcn`，相反，它屏蔽了基类的 `fcn`。结果 `D1` 有两个名为 `fcn` 的函数：类从 `Base` 继承了一个名为 `fcn` 的虚函数，类又定义了自己的名为 `fcn` 的非虚成员函数，该函数接受一个 `int` 形参。但是，从 `Base` 继承的虚函数不能通过 `D1` 对象（或 `D1` 的引用或指针）调用，因为该函数被 `fcn(int)` 的定义屏蔽了。

The class `D2` redefines both functions that it inherits. It redefines the virtual version of `fcn` originally defined in `Base` and the nonvirtual defined in `D1`.

类 `D2` 重定义了它继承的两个函数，它重定义了 `Base` 中定义的 `fcn` 的原始版本并重定义了 `D1` 中定义的非虚版本。

Calling a Hidden Virtual through the Base Class

通过基类调用被屏蔽的虚函数

When we call a function through a base-type reference or pointer, the compiler looks for that function in the base class and ignores the derived classes:

通过基类类型的引用或指针调用函数时，编译器将在基类中查找该函数而忽略派生类：

```
Base bobj; D1 dlobj; D2 d2obj;
Base *bp1 = &bobj, *bp2 = &dlobj, *bp3 = &d2obj;
bp1->fcn(); // ok: virtual call, will call Base::fcnat run time
bp2->fcn(); // ok: virtual call, will call Base::fcnat run time
bp3->fcn(); // ok: virtual call, will call D2::fcnat run time
```

All three pointers are pointers to the base type, so all three calls are resolved by looking in `Base` to see if `fcn` is defined. It is, so the calls are legal. Next, because `fcn` is virtual, the compiler generates code to make the call at run time based on the actual type of the object to which the

Section 15.5. Class Scope under Inheritance

reference or pointer is bound. In the case of `bp2`, the underlying object is a `D1`. That class did not redefine the virtual version of `fcn` that takes no arguments. The call through `bp2` is made (at run time) to the version defined in `Base`.

三个指针都是基类类型的指针，因此通过在 `Base` 中查找 `fcn` 来确定这三个调用，所以这些调用是合法的。另外，因为 `fcn` 是虚函数，所以编译器会生成代码，在运行时基于引用指针所绑定的对象的实际类型进行调用。在 `bp2` 的情况，基本对象是 `D1` 类的，`D1` 类没有重定义不接受实参的虚函数版本，通过 `bp2` 的函数调用（在运行时）调用 `Base` 中定义的版本。

Key Concept: Name Lookup and Inheritance

关键概念：名字查找与继承

Understanding how function calls are resolved is crucial to understanding inheritance hierarchies in C++. The following four steps are followed:

理解 C++ 中继承层次的关键在于理解如何确定函数调用。确定函数调用遵循以下四个步骤：

1. Start by determining the static type of the object, reference, or pointer through which the function is called.
首先确定进行函数调用的对象、引用或指针的静态类型。
2. Look for the function in that class. If it is not found, look in the immediate base class and continue up the chain of classes until either the function is found or the last class is searched. If the name is not found in the class or its enclosing base classes, then the call is in error.
在该类中查找函数，如果找不到，就在直接基类中查找，如此循着类的继承链往上找，直到找到该函数或者查找完最后一个类。如果不能在类或其相关基类中找到该名字，则调用是错误的。
3. Once the name is found, do normal type-checking ([Section 7.1.2, p. 229](#)) to see if this call is legal given the definition that was found.
一旦找到了该名字，就进行常规类型检查（[第 7.1.2 节](#)），查看如果给定找到的定义，该函数调用是否合法。
4. Assuming the call is legal, the compiler generates code. If the function is virtual and the call is through a reference or pointer, then the compiler generates code to determine which version to run based on the dynamic type of the object. Otherwise, the compiler generates code to call the function directly.
假定函数调用合法，编译器就生成代码。如果函数是虚函数且通过引用或指针调用，则编译器生成代码以确定根据对象的动态类型运行哪个函数版本，否则，编译器生成代码直接调用函数。

Exercises Section 15.5.4

Exercise

15.24: Why is it that, given

对于如下代码：

```
Bulk_item bulk;
Item_base item(bulk);
Item_base *p = &bulk;
```

the expression

为什么表达式

```
p->net_price(10);
```

invokes the `Bulk_item` instance of `net_price`, whereas

调用 `net_price` 的 `Bulk_item` 实例，而表达式

```
item.net_price(10);
```

invokes the `Item_base` instance?

调用 `Item_base` 实例？

Exercise

15.25: Assume `Derived` inherits from `Base` and that `Base` defines each of the following functions as virtual. Assuming `Derived` intends to define its own version of the virtual, determine which declarations in `Derived` are in error and specify what's wrong.

Section 15.5. Class Scope under Inheritance

假定 `Derived` 继承 `Base`, 并且 `Base` 将下面的函数定义为虚函数, 假定 `Derived` 打算定义自己的这个虚函数的版本, 确定在 `Derived` 中哪个声明是错误的, 并指出为什么错。

- (a) `Base* Base::copy(Base*);`
`Base* Derived::copy(Derived*);`
- (b) `Base* Base::copy(Base*);`
`Derived* Derived::copy(Base*);`
- (c) `ostream& Base::print(int, ostream&=cout);`
`ostream& Derived::print(int, ostream&);`
- (d) `void Base::eval() const;`
`void Derived::eval();`

Team LiB

◀ PREVIOUS NEXT ▶

15.6. Pure Virtual Functions

15.6. 纯虚函数

The `Disc_item` class that we wrote on page 583 presents an interesting problem: That class inherits the `net_price` function from `Item_base` but does not redefine it. We didn't redefine `net_price` because there is no meaning to ascribe to that function for the `Disc_item` class. A `Disc_item` doesn't correspond to any discount strategy in our application. This class exists solely for other classes to inherit from it.

在第 15.4.2 节所编写的 `Disc_item` 类提出了一个有趣的问题：该类从 `Item_base` 继承了 `net_price` 函数但没有重定义该函数。因为对 `Disc_item` 类而言没有可以给予该函数的意义，所以没有重定义该函数。在我们的应用程序中，`Disc_item` 不对应任何折扣策略，这个类的存在只是为了让其他类继承。

We don't intend for users to define `Disc_item` objects. Instead, `Disc_item` objects should exist only as part of an object of a type derived from `Disc_item`. However, as defined, there is nothing that prevents users from defining a plain `Disc_item` object. That leaves open the question of what would happen if a user did create a `Disc_item` and invoked `net_price` function on it. We now know from the scope discussion in the previous section that the effect would be to call the `net_price` function inherited from `Item_base`, which generates the undiscounted price.

我们不想让用户定义 `Disc_item` 对象，相反，`Disc_item` 对象只应该作为 `Disc_item` 派生类型的对象的一部分而存在。但是，正如已定义的，没有办法防止用户定义一个普通的 `Disc_item` 对象。这带来一个问题：如果用户创建一个 `Disc_item` 对象并调用该对象的 `net_price` 函数，会发生什么呢？从前面章节的讨论中了解到，结果将是调用从 `Item_base` 继承而来的 `net_price` 函数，该函数产生的是不打折的价格。

It's hard to say what behavior users might expect from calling `net_price` on a `Disc_item`. The real problem is that we'd rather they couldn't create such objects at all. We can enforce this design intent and correctly indicate that there is no meaning for the `Disc_item` version of `net_price` by making `net_price` a [pure virtual function](#). A pure virtual function is specified by writing `= 0` after the function parameter list:

很难说用户可能期望调用 `Disc_item` 的 `net_price` 会有什么样的行为。真正的问题在于，我们宁愿用户根本不能创建这样的对象。可以使 `net_price` 成为纯虚函数，强制实现这一设计意图并正确指出 `Disc_item` 的 `net_price` 版本没有意义。在函数形参表后面写上 `= 0` 以指定纯虚函数：

```
class Disc_item : public Item_base {
public:
    double net_price(std::size_t) const = 0;
};
```

Defining a virtual as pure indicates that the function provides an interface for subsequent types to override but that the version in this class will never be called. As importantly, users will not be able to create objects of type `Disc_item`.

将函数定义为纯虚能够说明，该函数为后代类型提供了可以覆盖的接口，但是这个类中的版本决不会调用。重要的是，用户将不能创建 `Disc_item` 类型的对象。

An attempt to create an object of an abstract base class is a compile-time error:

试图创建抽象基类的对象将发生编译时错误：

```
// Disc_item declares pure virtual functions
Disc_item discounted; // error: can't define a Disc_item object
Bulk_item bulk;      // ok: Disc_item subobject within Bulk_item
```



A class containing (or inheriting) one or more pure virtual functions is an [abstract base class](#). We may not create objects of an abstract type except as parts of objects of classes derived from the abstract base.

含有（或继承）一个或多个纯虚函数的类是抽象基类。除了作为抽象基类的派生类的对象的组成部分，不能创建抽象类型的对象。

Exercises Section 15.6

Exercise Make your version of the `Disc_item` class an abstract class.
15.26: 使你的 `Disc_item` 类版本成为抽象类。

Exercise Try to define an object of type `Disc_item` and see what errors you get from the compiler.

15.27:

试试定义 `Disc_item` 类型的一个对象，看看会从编译器得到什么错误。

Team LiB

◀ PREVIOUS NEXT ▶

15.7. Containers and Inheritance

15.7. 容器与继承

We'd like to use containers (or built-in arrays) to hold objects that are related by inheritance. However, the fact that objects are not polymorphic ([Section 15.3.1](#), p. 577) affects how we can use containers with types in an inheritance hierarchy.

我们希望使用容器（或内置数组）保存因继承而相关联的对象。但是，对象不是多态的（[第 15.3.1 节](#)），这一事实对将容器用于继承层次中的类型有影响。

As an example, our bookstore application would probably have the notion of a basket that represents the books a customer is buying. We'd like to be able to store the purchases in a `multiset` ([Section 10.5](#), p. 375). To define the `multiset`, we must specify the type of the objects that the container will hold. When we put an object in a container, the element is copied ([Section 9.3.3](#), p. 318).

例如，书店应用程序中可能有购物篮，购物篮代表顾客正在购买的书。我们希望能够在 `multiset` ([第 10.5 节](#)) 中存储购买物，要定义 `multiset`，必须指定容器将保存的对象的类型。将对象放进容器时，复制元素 ([第 9.3.3 节](#))。

If we define the `multiset` to hold objects of the base type

如果定义 `multiset` 保存基类类型的对象：

```
multiset<Item_base> basket;
Item_base base;
Bulk_item bulk;
basket.insert(base); // ok: add copy of base to basket
basket.insert(bulk); // ok: but bulk sliced down to its base part
```

then when we add objects that are of the derived type, only the base portion of the object is stored in the container. Remember, when we copy a derived object to a base object, the derived object is sliced down ([Section 15.3.1](#), p. 577).

则加入派生类型的对象时，只将对象的基类部分保存在容器中。记住，将派生类对象复制到基类对象时，派生类对象将被切掉（[第 15.3.1 节](#)）。

The elements in the container are `Item_base` objects. Regardless of whether the element was made as a copy of a `Bulk_item` object, when we calculate the `net_price` of an element the element would be priced without a discount. Once the object is put into the `multiset`, it is no longer a derived object.

容器中的元素是 `Item_base` 对象，无论元素是否作为 `Bulk_item` 对象的副本而建立，当计算元素的 `net_price` 时，元素将按不打折定价。一旦对象放入了 `multiset`，它就不再是派生类对象了。



Because derived objects are "sliced down" when assigned to a base object, containers and types related by inheritance do not mix well.

因为派生类对象在赋值给基类对象时会被“切掉”，所以容器与通过继承相关的类型不能很好地融合。

We cannot fix this problem by defining the container to hold derived objects. In this case, we couldn't put objects of `Item_base` into the container there is no standard conversion from base to derived type. We could explicitly cast a base-type object into a derived and add the resulting object to the container. However, if we did so, disaster would strike when we tried to use such an element. In this case, the element would be treated as if it were a derived object, but the members of the derived part would be uninitialized.

不能通过定义容器保存派生类对象来解决这个问题。在这种情况下，不能将 `Item_base` 对象放入容器——没有从基类类型到派生类型的转换。可以显式地将基类对象强制转换为派生类对象并将结果对象加入容器，但是，如果这样做，当试图使用这样的元素时，会产生大问题：在这种情况下，元素可以当作派生类对象对待，但派生类部分的成员将是未初始化的。

The only viable alternative would be to use the container to hold pointers to our objects. This strategy works but at the cost of pushing onto our users the problem of managing the objects and pointers. The user must ensure that the objects pointed to stay around for as long as the container. If the objects are dynamically allocated, then the user must ensure that they are properly freed when the container goes away. The next section presents a better and more common solution to this problem.

唯一可行的选择可能是使用容器保存对象的指针。这个策略可行，但代价是需要用户面对管理对象和指针的问题，用户必须保证只要容器存在，被指向的对象就存在。如果对象是动态分配的，用户必须保证在容器消失时适当地释放对象。下一节将介绍对这个问题更好更通用的解决方案。

Exercises Section 15.7

Exercise 15.28: Define a `vector` to hold objects of type `Item_base` and copy a number of objects of type `Bulk_item` into the `vector`. Iterate over the `vector` and generate the `net_price` for the elements in the container.

定义一个 `vector` 保存 `Item_base` 类型的对象，并将一些 `Bulk_item` 类型对象复制到 `vector` 中。遍历并计算容器中元素的总和。

Exercise 15.29: Repeat your program, but this time store pointers to objects of type `Item_base`. Compare the resulting sum.

重复程序，但这次存储 `Item_base` 类型对象的指针。比较结果总和。

Exercise 15.30: Explain any discrepancy in the amount generated by the previous two programs. If there is no discrepancy, explain why there isn't one.

解释上两题程序所产生总和的差异。如果没有差异，解释为什么没有。

15.8. Handle Classes and Inheritance

15.8. 句柄类与继承

One of the ironies of object-oriented programming in C++ is that we cannot use objects to support it. Instead, we must use pointers and references, not objects. For example, in the following code fragment,

C++ 中面向对象编程的一个颇具讽刺意味的地方是，不能使用对象支持面向对象编程，相反，必须使用指针或引用。例如，下面的代码段中：

```
void get_prices(Item_base object,
                const Item_base *pointer,
                const Item_base &reference)
{
    // which version of net_price is called is determined at run time
    cout << pointer->net_price(1) << endl;
    cout << reference.net_price(1) << endl;

    // always invokes Item_base::net_price
    cout << object.net_price(1) << endl;
}
```

the invocations through `pointer` and `reference` are resolved at run time based on the dynamic types of the object to which they are bound.

通过 `pointer` 和 `reference` 进行的调用在运行时根据它们所绑定对象的动态类型而确定。

Unfortunately, using pointers or references puts a burden on the users of our classes. We saw one such burden in the previous section that discussed the inter-actions between objects of inherited types and containers.

但是，使用指针或引用会加重类用户的负担。在[前一节](#)中讨论继承类型对象与容器的相互作用时，已经碰到了一种这样的负担。

A common technique in C++ is to define a so-called cover or [handle class](#). The handle class stores and manages a pointer to the base class. The type of the object to which that pointer points will vary; it can point at either a base- or a derived-type object. Users access the operations of the inheritance hierarchy through the handle. Because the handle uses its pointer to execute those operations, the behavior of virtual members will vary at run time depending on the kind of object to which the handle is actually bound. Users of the handle thus obtain dynamic behavior but do not themselves have to worry about managing the pointer.

C++ 中一个通用的技术是定义包装（cover）类或[句柄类](#)。句柄类存储和管理基类指针。指针所指对象的类型可以变化，它既可以指向基类类型对象又可以指向派生类型对象。用户通过句柄类访问继承层次的操作。因为句柄类使用指针执行操作，虚成员的行为将在运行时根据句柄实际绑定的对象的类型而变化。因此，句柄的用户可以获得动态行为但无须操心指针的管理。

Handles that cover an inheritance hierarchy have two important design considerations:

包装了继承层次的句柄有两个重要的设计考虑因素：

- As with any class that holds a pointer ([Section 13.5](#), p. 492), we must decide what to do about copy control. Handles that cover an inheritance hierarchy typically behave like either a smart pointer ([Section 13.5.1](#), p. 495) or a value ([Section 13.5.2](#), p. 499). 像对任何保存指针（[第 13.5 节](#)）的类一样，必须确定对复制控制做些什么。包装了继承层次的句柄通常表现得像一个智能指针（[第 13.5.1 节](#)）或者像一个值（[第 13.5.2 节](#)）。
- The handle class determines whether the handle interface will hide the inheritance hierarchy or expose it. If the hierarchy is not hidden, users must know about and use objects in the underlying hierarchy. 句柄类决定句柄接口屏蔽还是不屏蔽继承层次，如果不屏蔽继承层次，用户必须了解和使用基本层次中的对象。

There is no one right choice among these options; the decisions depend on the details of the hierarchy and how the class designer wants programmers to interact with those class(es). In the next two sections, we'll implement two different kinds of handles that address these design questions in different ways.

对于这些选项没有正确的选择，决定取决于继承层次的细节，以及类设计者希望程序员如何与那些类相互作用。下面两节将实现两种不同的句柄，用不同的方式解决这些设计问题。

15.8.1. A Pointerlike Handle

15.8.1. 指针型句柄

As our first example, we'll define a pointerlike handle class, named `Sales_item`, to represent our `Item_base` hierarchy. Users of `Sales_item` will use it as if it were a pointer: Users will bind a `Sales_item` to an object of type `Item_base` and will then use the `*` and `->` operations to execute `Item_base` operations:

像第一个例子一样，我们将定义一个名为 `Sales_item` 的指针型句柄类，表示 `Item_base` 层次。`Sales_item` 的用户将像使用指针一样使用它：用户将 `Sales_item` 绑定到 `Item_base` 类型的对象并使用 `*` 和 `->` 操作符执行 `Item_base` 的操作：

```
// bind a handle to a Bulk_item object
Sales_item item(Bulk_item("0-201-82470-1", 35, 3, .20));

item->net_price(); // virtual call to net_price function
```

However, users won't have to manage the object to which the handle points; the `Sales_item` class will do that part of the job. When users call a function through a `Sales_item`, they'll get polymorphic behavior.

但是，用户不必管理句柄指向的对象，`Sales_item` 类将完成这部分工作。当用户通过 `Sales_item` 类对象调用函数时，将获得多态行为。

Defining the Handle

定义句柄

We'll give our class three constructors: a default constructor, a copy constructor, and a constructor that takes an `Item_base`. This third constructor will copy the `Item_base` and ensure that the copy stays around as long as the `Sales_item` does. When we copy or assign a `Sales_item`, we'll copy the pointer rather than copying the object. As with our other pointerlike handle classes, we'll use a use count to manage the copies.

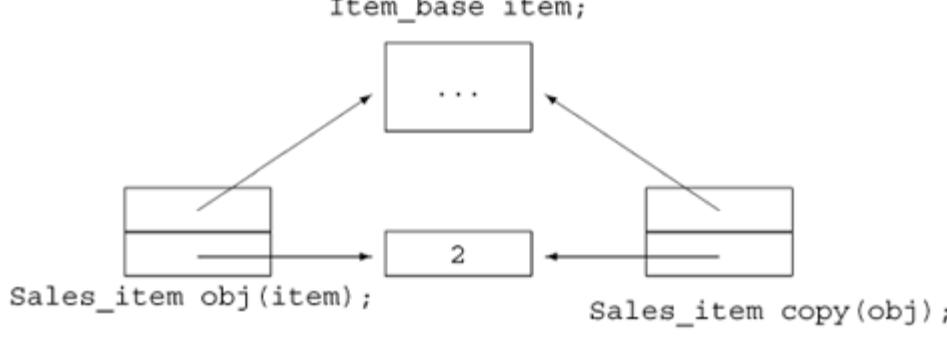
`Sales_item` 类有三个构造函数：默认构造函数、复制构造函数和接受 `Item_base` 对象的构造函数。第三个构造函数将复制 `Item_base` 对象，并保证：只要 `Sales_item` 对象存在副本就存在。当复制 `Sales_item` 对象或给 `Sales_item` 对象赋值时，将复制指针而不是复制对象。像对其他指针型句柄类一样，将用使用计数来管理副本。

The use-counted classes we've used so far have used a companion class to store the pointer and associated use count. In this class, we'll use a different design, as illustrated in [Figure 15.2](#). The `Sales_item` class will have two data members, both of which are pointers: One pointer will point to the `Item_base` object and the other will point to the use count. The `Item_base` pointer might point to an `Item_base` object or an object of a type derived from `Item_base`. By pointing to the use count, multiple `Sales_item` objects can share the same counter.

迄今为止，我们已经使用过的使用计数类，都使用一个伙伴类来存储指针和相关的使用计数。这个例子将使用不同的设计，如[图 15.2](#) 所示。`Sales_item` 类将有两个数据成员，都是指针：一个指针将指向 `Item_base` 对象，而另一个将指向使用计数。`Item_base` 指针可以指向 `Item_base` 对象也可以指向 `Item_base` 派生类型的对象。通过指向使用计数，多个 `Sales_item` 对象可以共享同一计数器。

Figure 15.2. Use-Count Strategy for the `Sales_item` Handle Class

图 15.2. `Sales_item` 句柄类的使用计数策略



In addition to managing the use count, the `Sales_item` class will define the dereference and arrow operators:

除了管理使用计数之外，`Sales_item` 类还将定义解引用操作符和箭头操作符：

```
// use counted handle class for the Item_base hierarchy
class Sales_item {
public:
    // default constructor: unbound handle
    Sales_item(): p(0), use(new std::size_t(1)) { }
```

```

// attaches a handle to a copy of the Item_base object
Sales_item(const Item_base&);

// copy control members to manage the use count and pointers
Sales_item(const Sales_item &i):
    p(i.p), use(i.use) { ++use; }
~Sales_item() { decr_use(); }
Sales_item& operator=(const Sales_item&);

// member access operators
const Item_base *operator->() const { if (p) return p;
    else throw std::logic_error("unbound Sales_item"); }
const Item_base &operator*() const { if (p) return *p;
    else throw std::logic_error("unbound Sales_item"); }

private:
    Item_base *p;           // pointer to shared item
    std::size_t *use;        // pointer to shared use count
    // called by both destructor and assignment operator to free pointers
    void decr_use()
    { if (--*use == 0) { delete p; delete use; } }
};

}

```

Use-Counted Copy Control

使用计数式复制控制

The copy-control members manipulate the use count and the `Item_base` pointer as appropriate. Copying a `Sales_item` involves copying the two pointers and incrementing the use count. The destructor decrements the use count and destroys the pointers if the count goes to zero. Because the assignment operator will need to do the same work, we implement the destructor's actions in a private utility function named `decr_use`.

复制控制成员适当地操纵使用计数和 `Item_base` 指针。复制 `Sales_item` 对象包括复制两个指针和将使用计数加 1。析构函数将使用计数减 1，如果计数减至 0 就撤销指针。因为赋值操作符需要完成同样的工作，所以在一个名为 `decr_use` 的私有实用函数中实现析构函数的行为。

The assignment operator is a bit more complicated than the copy constructor:

赋值操作符比复制构造函数复杂一点：

```

// use-counted assignment operator; use is a pointer to a shared use count
Sales_item&
Sales_item::operator=(const Sales_item &rhs)
{
    ++rhs.use;
    decr_use();
    p = rhs.p;
    use = rhs.use;
    return *this;
}

```

The assignment operator acts like the copy constructor in that it increments the use count of the right-hand operand and copies the pointer. It also acts like the destructor in that we first have to decrement the use count of the left-hand operand and then delete the pointers if the use count goes to zero.

赋值操作符像复制构造函数一样，将右操作数的使用计数加 1 并复制指针；它也像析构函数一样，首先必须将左操作数的使用计数减 1，如果使用计数减至 0 就删除指针。

As usual with an assignment operator, we must protect against self-assignment. This operator handles self-assignment by first incrementing the use count in the right-hand operand. If the left- and right-hand operands are the same, the use count will be at least 2 when `decr_use` is called. That function decrements and checks the use count of the left-hand operand. If the use count goes to zero, then `decr_use` will free the `Item_base` and `use` objects currently in this object. What remains is to copy the pointers from the right-hand to the left-hand operand. As usual, our assignment operator returns a reference to the left-hand operand.

像通常对赋值操作符一样，必须防止自身赋值。这个操作符通过首先将右操作数的使用计数减 1 来处理自身赋值。如果左右操作数相同，则调用 `decr_use` 时使用计数将至少为 2。该函数将左操作数的使用计数减 1 并进行检查，如果使用计数减至 0，则 `decr_use` 将释放该对象中的 `Item_base` 对象和 `use` 对象。剩下的是从右操作数向左操作数复制指针，像平常一样，我们的赋值操作符返回左操作数的引用。

Aside from the copy-control members, the only other functions `Sales_item` defines are the operator functions, `operator*` and `operator->`. Users will access `Item_base` members through these operators. Because these operators return a pointer and reference, respectively, functions called through these operators will be dynamically bound.

除了复制控制成员以外，`Sales_item` 定义的其他函数是操作函数 `operator*` 和 `operator->`，用户将通过这些操作符访问 `Item_base` 成员。因为这两个操作符分别返回指针和引用，所以通过这些操作符调用的函数将进行动态绑定。

We define only the `const` versions of these operators because the `public` members in the underlying `Item_base` hierarchy are all `const`.

我们只定义了这些操作符的 `const` 版本，因为基础 `Item_base` 层次中的成员都是 `const` 成员。

Constructing the Handle

构造句柄

Our handle has two constructors: the default constructor, which creates an un-bound `Sales_item`, and a second constructor, which takes an object to which it attaches the handle.

我们句柄有两个构造函数：默认构造函数创建未绑定的 `Sales_item` 对象，第二个构造函数接受一个对象，将句柄与其关联。

The first constructor is easy: We set the `Item_base` pointer to 0 to indicate that this handle is not attached to any object. The constructor allocates a new use counter and initializes it to 1.

第一个构造函数容易定义：将 `Item_base` 指针置 0 以指出该句柄没有关联任何对象上。构造函数分配一个新的计数器并将它初始化为 1。

The second constructor is more difficult. We'd like users of our handle to create their own objects, to which they could attach a handle. The constructor will allocate a new object of the appropriate type and copy the parameter into that newly allocated object. That way the `Sales_item` class will own the object and can guarantee that the object is not deleted until the last `Sales_item` attached to the object goes away.

第二个构造函数难一点，我们希望句柄的用户创建自己的对象，在这些对象上关联句柄。构造函数将分配适当类型的新对象并将形参复制到新分配的对象中，这样，`Sales_item` 类将拥有对象并能够保证在关联到该对象的最后一个 `Sales_item` 对象消失之前不会删除对象。

15.8.2. Cloning an Unknown Type

15.8.2. 复制未知类型

To implement the constructor that takes an `Item_base`, we must first solve a problem: We do not know the actual type of the object that the constructor is given. We know that it is an `Item_base` or an object of a type derived from `Item_base`. Handle classes often need to allocate a new copy of an existing object *without knowing the precise type of the object*. Our `Sales_item` constructor is a good example.

要实现接受 `Item_base` 对象的构造函数，必须首先解决一个问题：我们不知道给予构造函数的对象的实际类型。我们知道它是一个 `Item_base` 对象或者是一个 `Item_base` 派生类型的对象。句柄类经常需要在不知道对象的确切类型时分配书籍对象的新副本。`Sales_item` 构造函数是个好例子。



The common approach to solving this problem is to define a virtual operation to do the copy, which we'll name `clone`.

解决这个问题的通用方法是定义虚操作进行复制，我们称将该操作命名为 `clone`。

To support our handle class, we'll need to add `clone` to each of the types in the hierarchy, starting with the base class, which must define the function as `virtual`:

为了句柄类，需要从基类开始，在继承层次的每个类型中增加 `clone`，基类必须将该函数定义为虚函数：

```
class Item_base {
public:
    virtual Item_base* clone() const
        { return new Item_base(*this); }
};
```

Each class must now redefine the `virtual`. Because the function exists to generate a new copy of an object of the class, we'll define the return type to reflect the type of the class itself:

每个类必须重定义该虚函数。因为函数的存在是为了生成类对象的新副本，所以定义返回类型为类本身：

```
class Bulk_item : public Item_base {
public:
    Bulk_item* clone() const
        { return new Bulk_item(*this); }
};
```

On page 564 we said there is one exception to the requirement that the return type of the derived class must match exactly that of the base class instance. That exception supports cases such as this one. If the base instance of a `virtual` function returns a reference or pointer to a class type, the derived version of the `virtual` may return a class `publicly` derived from the class returned by the base class instance (or a pointer or a reference to a class type).

[第 15.2.3 节](#)介绍过，对于派生类的返回类型必须与基类实例的返回类型完全匹配的要求，但有一个例外。这个例外支持像这个类这样的情况。如果虚函数的基类实例返回类类

型的引用或指针，则该虚函数的派生类实例可以返回基类实例返回的类型的派生类（或者是类类型的指针或引用）。

Defining the Handle Constructors

定义句柄构造函数

Once the `clone` function exists, we can write the `Sales_item` constructor:

一旦有了 `clone` 函数，就可以这样编写 `Sales_item` 构造函数：

```
Sales_item::Sales_item(const Item_base &item):
    p(item.clone()), use(new std::size_t(1)) { }
```

Like the default constructor, this constructor allocates and initializes its use count. It calls `clone` on its parameter to generate a (virtual) copy of that object. If the argument is an `Item_base`, then the `clone` function for `Item_base` is run; if the argument is a `Bulk_item`, then the `Bulk_item` `clone` is executed.

像默认构造函数一样，这个构造函数分配并初始化使用计数，它调用形参的 `clone` 产生那个对象的（虚）副本。如果实参是 `Item_base` 对象，则运行 `Item_base` 的 `clone` 函数；如果实参是 `Bulk_item` 对象，则执行 `Bulk_item` 的 `clone` 函数。

Exercises Section 15.8.2

Exercise 15.31: Define and implement the `clone` operation for the limited discount class implemented in the exercises for [Section 15.2.3](#) (p. 567).

为第 15.2.3 节的习题中实现的有限折扣类定义的实现 `clone` 操作。

Exercise 15.32: In practice, our programs are unlikely to run correctly the first time we run them or the first time we run them against real data. It is often useful to incorporate a debugging strategy into the design of our classes. Implement a virtual `debug` function for our `Item_base` class hierarchy that displays the data members of the respective classes.

实际上，程序不太可能在第一次运行或第一次用真实数据运行时就能正确运行。在类的设计中包括调试策略经常是有用的。为 `Item_base` 类层次实现一个 `debug` 虚函数，显示各个类的数据成员。

Exercise 15.33: Given the version of the `Item_base` hierarchy that includes the `Disc_item` abstract base class, indicate whether the `Disc_item` class should implement the `clone` function. If not, why not? If so, why?

对于 `Item_base` 层次的包括 `Disc_item` 抽象基类的版本，指出 `Disc_item` 类是否应实现 `clone` 函数，为什么？

Exercise 15.34: Modify your debug function to let users turn debugging on or off. Implement the control two ways:

修改调试函数以允许用户打开或关闭调试。用两种方式实现控制：

- By defining a parameter to the debug function

通过定义 `debug` 函数的形参。

- By defining a class data member that allows individual objects to turn on or turn off the display of debugging information

通过定义类数据成员。该成员允许个体对象打开或关闭调试信息的显示。

15.8.3. Using the Handle

15.8.3. 句柄的使用

Section 15.8. Handle Classes and Inheritance

Using `Sales_item` objects, we could more easily write our bookstore application. Our code wouldn't need to manage pointers to the `Item_base` objects, yet the code would obtain virtual behavior on calls made through a `Sales_item`.

使用 `Sales_item` 对象可以更容易地编写书店应用程序。代码将不必管理 `Item_base` 对象的指针，但仍然可以获得通过 `Sales_item` 对象进行的调用的虚行为。

As an example, we could use `Item_base` objects to solve the problem proposed in [Section 15.7](#) (p. 597). We could use `Sales_items` to keep track of the purchases a customer makes, storing a `Sales_item` representing each purchase in a `multiset`. When the customer was done shopping, we would total the sale.

例如，可以使用 `Item_base` 对象解决[第 15.7 节](#)提出的问题。可以使用 `Sales_item` 对象跟踪顾客所做购买，在 `multiset` 中保存一个对象表示一次购买，当顾客完成购买时，可以计算销售总数。

Comparing Two `Sales_items`

比较两个 `Sales_item` 对象

Before writing the function to total a sale, we need to define a way to compare `Sales_items`. To use `Sales_item` as the key in an associative container, we must be able to compare them ([Section 10.3.1](#), p. 360). By default, the associative containers use the less-than operator on the key type. However, for the same reasons discussed about our original `Sales_item` type in [Section 14.3.2](#) (p. 520), defining `operator<` for the `Sales_item` handle would be a bad idea: We want to take only the ISBN into account when we use `Sales_item` as a key, but want to consider all data members when determining equality.

在编写函数计算销售总数之前，需要定义比较 `Sales_item` 对象的方法。要用 `Sales_item` 作为关联容器的关键字，必须能够比较它们（[第 10.3.1 节](#)）。关联容器默认使用关键字类型的小于操作符，但是，基于[第 14.3.2 节](#)讨论过的有关原始 `Sales_item` 类型的同样理由，为 `Sales_item` 句柄类定义 `operator>` 可能是个坏主意：当使用 `Sales_item` 作关键字时，只想考虑 ISBN，但确定相等时又想要考虑所有数据成员。

Fortunately, the associative containers allow us to specify a function (or function object ([Section 14.8](#), p. 530)) to use as the comparison function. We do so similarly to the way we passed a separate function to the `stable_sort` algorithm in [Section 11.2.3](#) (p. 403). In that case, we needed only to pass an additional argument to `stable_sort` to provide a comparison function to use in place of the `<` operator. Overriding an associative container's comparison function is a bit more complicated because, as we shall see, we must supply the comparison function when we define the container object.

幸好，关联容器使我们能够指定一个函数或函数对象（[第 14.8 节](#)）用作比较函数，这样做类似于[第 11.2.3 节](#)中将单独函数传给 `stable_sort` 算法的方式。在那种情况下，只需要将附加的实参传给 `stable_sort` 以提供比较函数，代替 `<` 操作符的。覆盖关联容器的比较函数有点复杂，因为，正如我们将看到的，在定义容器对象时必须提供比较函数。

Let's start with the easy part, which is to define a function to use to compare `Sales_item` objects:

让我们比较容易的部分开始，定义一个函数用于比较 `Sales_item` 对象：

```
// compare defines item ordering for the multiset in Basket
inline bool
compare(const Sales_item &lhs, const Sales_item &rhs)
{
    return lhs->book() < rhs->book();
}
```

Our `compare` function has the same interface as the less-than operator. It returns a `bool` and takes two `const` references to `Sales_items`. It compares the parameters by comparing their ISBNs. This function uses the `Sales_item ->` operator, which returns a pointer to an `Item_base` object. That pointer is used to fetch and run the `book` member, which returns the ISBN.

我们的 `compare` 函数与小于操作符有同样的接口，它接受两个 `Sales_item` 对象的 `const` 引用，通过比较 ISBN 而比较形参，返回一个 `book` 值。该函数使用 `Sales_item` 的 `->` 操作符，该操作符返回 `Item_base` 对象的指针，那个指针用于获取并运行成员 `book`，该成员返回 ISBN。

Using a Comparator with an Associative Container

使用带关联容器的比较器

If we think a bit about how the comparison function is used, we'll realize that it must be stored as part of the container. The comparison function is used by any operation that adds or finds an element in the container. In principle, each of these operations could take an optional extra argument that represented the comparison function. However, this strategy would be error-prone: If two operations used different comparison functions, then the ordering would be inconsistent. It's impossible to predict what would happen in practice.

如果考虑一下如何使用比较函数，就会认识到，它必须作为容器的部分而存储。任何在容器中增加或查找元素的操作都要使用比较函数。原则上，每个这样的操作可以接受一个可选的附加实参，表示比较函数。但是，这种策略容易导致出错：如果两个操作使用不同的比较函数，顺序可能会不一致。不可能预测实际上会发生什么。

To work effectively, an associative container needs to use the same comparison function for every operation. Yet, it is unreasonable to expect users to remember the comparison function every time, especially when there is no way to check that each call uses the same comparison function.

Section 15.8. Handle Classes and Inheritance

Therefore, it makes sense for the container to remember the comparison function. By storing the comparator in the container object, we are assured that every operation that compares elements will do so consistently.

要有效地工作，关联容器需要对每个操作使用同一比较函数。然而，期望用户每次记住比较函数是不合理的，尤其是，没有办法检查每个调用使用同一比较函数。因此，容器记住比较函数是有意义的。通过将比较器存储在容器对象中，可以保证比较元素的每个操作将一致地进行。

For the same reasons that the container needs to know the element type, it needs to know the comparator type in order to store the comparator. In principle, the container could infer this type by assuming that the comparator is pointer to a function that returns a `bool` and takes references to two objects of the `key_type` of the container. Unfortunately, this inferred type would be overly restrictive. For one thing, we should allow the comparator to be a function object as well as a plain function. Even if we were willing to require that the comparator be a function, the inferred type would still be too restrictive. After all, the comparison function might return an `int` or any other type that can be used in a condition. Similarly, the parameter type need not exactly match the `key_type`. Any parameter type that is convertible to the `key_type` should also be allowed.

基于同样的理由，容器需要知道元素类型，为了存储比较器，它需要知道比较器类型。原则上，通过假定比较器是一个函数指针，该函数接受两个容器的 `key_type` 类型的对象并返回 `bool` 值，容器可以推断出这个类型。不幸的是，这个推断出的类型可能限制太大。首先，应该允许比较器是函数对象或是普通函数。即使我们愿意要求比较器为函数，这个推断出的类型也可能仍然太受限制了，毕竟，比较函数可以返回 `int` 或者其他任意可用在条件中的类型。同样，形参类型也不需要与 `key_type` 完全匹配，应该允许可以转换为 `key_type` 的任意形参类型。

So, to use our `Sales_item` comparison function, we must specify the comparator type when we define the `multiset`. In our case, that type is a function that returns a `bool` and takes two `const Sales_item` references.

所以，要使用 `Sales_item` 的比较函数，在定义 `multiset` 时必须指定比较器类型。在我们的例子中，比较器类型是接受两个 `const Sales_item` 引用并返回 `bool` 值的函数。

We'll start by defining a `typedef` that is a synonym for this type ([Section 7.9](#), p. 276):

首先定义一个类型别名，作为该类型的同义词 ([第 7.9 节](#)) :

```
// type of the comparison function used to order the multiset
typedef bool (*Comp)(const Sales_item&, const Sales_item&);
```

This statement defines `Comp` as a synonym for the pointer to function type that matches the comparison function we wish to use to compare `Sales_item` objects.

这个语句将 `Comp` 定义为函数类型指针的同义词，该函数类型与我们希望用来比较 `Sales_item` 对象的比较函数相匹配。

Next we'll need to define a `multiset` that holds objects of type `Sales_item` and that uses this `Comp` type for its comparison function. Each constructor for the associative containers allows us to supply the name of the comparison function. We can define an empty `multiset` that uses our `compare` function as follows:

接着需要定义 `multiset`，保存 `Sales_item` 类型的对象并在它的比较函数中使用这个 `Comp` 类型。关联容器的每个构造函数使我们能够提供比较函数的名字。可以这样定义使用 `compare` 函数的空 `multiset`:

```
std::multiset<Sales_item, Comp> items(compare);
```

This definition says that `items` is a `multiset` that holds `Sales_item` objects and uses an object of type `Comp` to compare them. The `multiset` is empty—we supplied no elements—but we did supply a comparison function named `compare`. When we add or look for elements in `items` our `compare` function will be used to order the `multiset`.

这个定义是说，`items` 是一个 `multiset`，它保存 `Sales_item` 对象并使用 `Comp` 类型的对象比较它们。`multiset` 是空的——我们没有提供任何元素，但我们确实提供了一个名为 `compare` 的比较函数。当在 `items` 中增加或查找元素时，将用 `compare` 函数对 `multiset` 进行排序。

Containers and Handle Classes

容器与句柄类

Now that we know how to supply a comparison function, we'll define a class, named `Basket`, to keep track of a sale and calculate the purchase price:

既然知道了怎样提供比较函数，我们将定义名为 `Basket` 的类，以跟踪销售并计算购买价格：

```
class Basket {
    // type of the comparison function used to order the multiset
    typedef bool (*Comp)(const Sales_item&, const Sales_item&);
public:
    // make it easier to type the type of our set
    typedef std::multiset<Sales_item, Comp> set_type;
    // typedefs modeled after corresponding container types
    typedef set_type::size_type size_type;
    typedef set_type::const_iterator const_iter;
    Basket(): items(compare) { } // initialize the comparator
```

Section 15.8. Handle Classes and Inheritance

```
void add_item(const Sales_item &item)
    { items.insert(item); }
size_type size(const Sales_item &i) const
    { return items.count(i); }
double total() const; // sum of net prices for all items in the basket
private:
    std::multiset<Sales_item, Comp> items;
};
```

This class holds the customer's purchases in a `multiset` of `Sales_item` objects. We use a `multiset` to allow the customer to buy multiple copies of the same book.

这个类在 `Sales_item` 对象的 `multiple` 中保存顾客购买的商品，用 `multiple` 使顾客能够购买同一本书的多个副本。

The class defines a single constructor, the `Basket` default constructor. The class needs its own default constructor to pass `compare` to the `multiset` constructor that builds the `items` member.

该类定义了一个构造函数，即 `Basket` 默认构造函数。该类需要自己的默认构造函数，以便将 `compare` 传给建立 `items` 成员的 `multiset` 构造函数。

The operations that the `Basket` class defines are fairly simple: `add_item` takes a reference to a `Sales_item` and puts a copy of that item into the `multiset`; `item_count` returns the number of records for this ISBN in the basket for a given ISBN. In addition to the operations, `Basket` defines three typedefs to make it easier to use its `multiset` member.

`Basket` 类定义的操作非常简单：`add_item` 操作接受 `Sales_item` 对象引用并将该项目的副本放入 `multiset`；对于给定 ISBN，`size` 操作返回购物篮中该 ISBN 的记录数。除了操作，`Basket` 还定义了三个类型别名，这样使用它的 `multiset` 成员就比较容易了。

Using the Handle to Execute a Virtual Function

使用句柄执行虚函数

The only complicated member of class `Basket` is the `total` function, which returns the price for all the items in the basket:

`Basket` 类唯一的复杂成员是 `total` 函数，该函数返回购物篮中所有物品的价格：

```
double Basket::total() const
{
    double sum = 0.0; // holds the running total

    /* find each set of items with the same isbn and calculate
     * the net price for that quantity of items
     * iter refers to first copy of each book in the set
     * upper_bound refers to next element with a different isbn
     */
    for (const_iter iter = items.begin();
         iter != items.end(); iter =
            items.upper_bound(*iter))
    {
        // we know there's at least one element with this key in the Basket
        // virtual call to net_price applies appropriate discounts, if any
        sum += (*iter)->net_price(items.count(*iter));
    }
    return sum;
}
```

The `total` function has two interesting parts: the call to the `net_price` function, and the structure of the `for` loop. We'll look at each in turn.

`total` 函数有两个有趣的部分：对 `net_price` 函数的调用，以及 `for` 循环结构。我们逐一进行分析。

When we call `net_price`, we need to tell it how many copies of a given book are being purchased. The `net_price` function uses this argument to determine whether the purchase qualifies for a discount. This requirement implies that we'd like to process the `multiset` in chunksprocessing all the records for a given title in one chunk and then the set of those for the next title and so on. Fortunately, `multiset` is well suited to this problem.

调用 `net_price` 函数时，需要告诉它某本书已经购买了多少本，`net_price` 函数使用这个实参确定是否打折。这个要求暗示着我们希望成批处理 `multiset`——处理给定标题的所有记录，然后处理下一个标题的所有记录，以此类推。幸好，`multiset` 非常适合处理这个问题。

Our `for` loop starts by defining and initializing `iter` to refer to the first element in the `multiset`. We use the `multiset` `count` member ([Section 10.3.6, p. 367](#)) to determine how many elements in the `multiset` have the same key (e.g., same `isbn`) and use that number as the argument to the call to `net_price`.

`for` 循环开始于定义 `iter` 并将 `iter` 初始化为指向 `multiset` 中的第一个元素。我们使用 `multiset` 的 `count` 成员 ([第 10.3.6 节](#)) 确定 `multiset` 中的多少成员具有相同的键（即，相同的 `isbn`），并且使用该数目作为实参调用 `net_price` 函数。

The interesting bit is the "increment" expression in the `for`. Rather than the usual loop that reads each element, we advance `iter` to refer to the next key. We skip over all the elements that match the current key by calling `upper_bound` ([Section 10.5.2, p. 377](#)). The call to `upper_bound` returns

the iterator that refers to the element just past the last one with the same key as in `iter`. That iterator we get back denotes either the end of the set or the next unique book. We test the new value of `iter`. If `iter` is equal to `items.end()`, we drop out of the `for`. Otherwise, we process the next book.

`for` 循环中的“增量”表达式很有意思。与读每个元素的一般循环不同，我们推进 `iter` 指向下一个键。调用 `upper_bound` 函数以跳过与当前键匹配的所有元素，`upper_bound` 函数的调用返回一个迭代器，该迭代器指向与 `iter` 键相同的最后一个元素的下一元素，即，该迭代器指向集合的末尾或下一本书。测试 `iter` 的新值，如果与 `items.end()` 相等，则跳出 `for` 循环，否则，就处理下一本。

The body of the `for` calls the `net_price` function. That call can be a bit tricky to read:

`for` 循环的循环体调用 `net_price` 函数，阅读这个调用需要一点技巧：

```
sum += (*iter)->net_price(items.count(*iter));
```

We dereference `iter` to get the underlying `Sales_item` to which we apply the overloaded arrow operator from the `Sales_item` class. That operator returns the underlying `Item_base` object to which the handle is attached. From that object we call `net_price`, passing the `count` of items with the same `isbn`. The `net_price` function is virtual, so the version of the pricing function that is called depends on the type of the underlying `Item_base` object.

对 `iter` 解引用获得基础 `Sales_item` 对象，对该对象应用 `Sales_item` 类重载的箭头操作符，该操作符返回句柄所关联的基础 `Item_base` 对象，用该 `Item_base` 对象调用 `net_price` 函数，传递具有相同 `isbn` 的图书的 `count` 作为实参。`net_price` 是虚函数，所以调用的定价函数的版本取决于基础 `Item_base` 对象的类型。

Exercises Section 15.8.3

Exercise 15.35: Write your own version of the `compare` function and `Basket` class and use them to manage a sale.

编写自己的 `compare` 函数和 `Basket` 类的版本并使用它们管理销售。

Exercise 15.36: What is the underlying type of `Basket::const_iter`?

`Basket::const_iter` 的基础类型是什么？

Exercise 15.37: Why did we define the `Comp` typedef in the `private` part of `Basket`?

为什么在 `Basket` 的 `private` 部分定义 `Comp` 类型别名？

Exercise 15.38: Why did we define two `private` sections in `Basket`?

为什么在 `Basket` 中定义两个 `private` 部分？

15.9. Text Queries Revisited

15.9. 再谈文本查询示例

As a final example of inheritance, we'll extend our text query application from [Section 10.6](#) (p. 379). The class we developed there let us look for occurrences of a given word in a text file. We'd like to extend the system to support more complex queries.

作为继承的最后一个例子，我们来扩展[第 10.6 节](#)的文本查询应用程序。使用在[第 10.6 节](#)开发的类，已经能够在文本文件中查找给定单词的出现，但我们想扩展系统以支持更复杂的查询。

For illustration purposes, we'll run queries against the following simple story:

为了说明问题，将用下面的简单小说来运行查询：

```
Alice Emma has long flowing red hair.  
Her Daddy says when the wind blows  
through her hair, it looks almost alive,  
like a fiery bird in flight.  
A beautiful fiery bird, he tells her,  
magical but untamed.  
"Daddy, shush, there is no such thing,"  
she tells him, at the same time wanting  
him to tell her more.  
Shyly, she asks, "I mean, Daddy, is there?"
```

Our system should support:

系统应该支持：

1. Word queries that find a single word. All lines in which the word appears should be displayed in ascending order:

查找单个单词的查询。按升序显示所有包含该单词的行：

```
Executed Query for:  
Daddy match occurs 3 times:  
(line 2) Her Daddy says when the wind blows  
(line 7) "Daddy, shush, there is no such thing,"  
(line 10) Shyly, she asks, "I mean, Daddy, is there?"
```

2. Not queries, using the ~ operator. All lines that do not match the query are displayed:

“非”查询，使用 ~ 操作符。显示所有不匹配的行：

```
Executed Query for: ~(Alice)  
match occurs 9 times:  
(line 2) Her Daddy says when the wind blows  
(line 3) through her hair, it looks almost alive,  
(line 4) like a fiery bird in flight. ...
```

3. Or queries, using the | operator. All lines in which either of two queries match are displayed:

“或”查询，使用 | 操作符。显示与两个查询条件中任意一个匹配的所有行：

```
Executing Query for: (hair | Alice)  
match occurs 2 times:  
(line 1) Alice Emma has long flowing red hair.  
(line 3) through her hair, it looks almost alive,
```

4. And queries, using the & operator. All lines in which both queries match are displayed.

“与”查询，使用 & 操作符。显示与两个查询条件都匹配的所有行：

```
Executed query: (hair & Alice)  
match occurs 1 time:  
(line 1) Alice Emma has long flowing red hair.
```

Moreover, these elements can be combined, as in

而且，可以组合这些元素，如

```
fiery & bird | wind
```

Our system will not be sophisticated enough to read these expressions. Instead, we'll build them up inside a C++ program. Hence, we'll evaluate compound expressions such as this example using normal C++ precedence rules. The evaluation of this query will match a line in which `fiery` and `bird` appear or one in which `wind` appears. It will not match a line on which `fiery` or `bird` appears alone:

我们的系统没有复杂到能够读这些表达式。我们将在 C++ 程序中创建它们，因此，将用常规 C++ 优先级规则对诸如此类的复合表达式求值。这个查询的求值结果将与出现的 `fiery` 和 `bird` 的行或者出现 `wind` 的行相匹配，而不会与 `fiery` 或 `bird` 单独出现的行相匹配：

```
Executing Query for: ((fiery & bird) | wind)
match occurs 3 times:
(line 2) Her Daddy says when the wind blows
(line 4) like a fiery bird in flight.
(line 5) A beautiful fiery bird, he tells her,
```

Our output will print the query, using parentheses to indicate the way in which the query was interpreted. As with our original implementation, our system must be smart enough not to display the same line more than once.

输出将打印查询，并使用圆括号指出解释该查询的方法。像原来的实现一样，系统必须足够聪明，不会重复显示相同行。

15.9.1. An Object-Oriented Solution

15.9.1. 面向对象的解决方案

We might think that we could use the `TextQuery` class from page 382 to represent our word queries. We might then derive our other queries from that class.

可以考虑使用第 10.6.2 节的 `TextQuery` 表示单词查询，然后从 `TextQuery` 类派生其他类。

However, this design would be flawed. Conceptually, a "not" query is not a kind of word query. Instead, a not query "has a" query (word query or any other kind of query) whose value it negates.

但是，这个设计可能缺陷。概念上，“非”查询不是一种单词查询，相反，非查询“有一个”查询（单词或其他任意种类的查询），非查询对该查询的值求反。

This observation suggests that we model our different kinds of queries as independent classes that share a common base class:

注意到这一点，我们将不同种类的查询建模为独立的类，它们共享一个公共基类：

```
WordQuery // Shakespeare
NotQuery // ~Shakespeare
OrQuery // Shakespeare / Marlowe
AndQuery // William & Shakespeare
```

Instead of inheriting from `TextQuery`, we will use that class to hold the file and build the associated `word_map`. We'll use the query classes to build up expressions that will ultimately run queries against the file in a `TextQuery` object.

我们不继承 `TextQuery`，而是使用 `TextQuery` 类保存文件并建立相关的 `word_map`，使用查询类建立表达式，这些表达式最终对 `TextQuery` 对象中的文件运行查询。

Abstract Interface Class

抽象接口类

We have identified four kinds of query classes. These classes are conceptually siblings. Each class shares the same abstract interface, which suggests that we'll need to define an abstract base class (Section 15.6, p. 595) to represent the operations performed by a query. We'll name our abstract class `Query_base`, indicating that its role is to serve as the root of our query hierarchy.

已经识别出四各查询类，这些类在概念上是兄弟类。它们共享相同的抽象接口，这暗示我们定义一个抽象基类（第 15.6 节）以表示由查询执行的操作。将该抽象基类命名为 `Query_base`，以指出它的作用是作为查询继承层次的根。

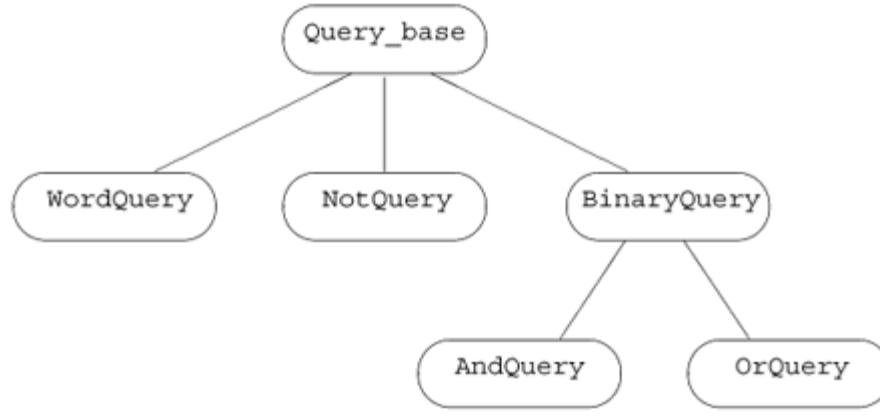
We'll derive `WordQuery` and `NotQuery` directly from our abstract base. The `AndQuery` and `OrQuery` classes share one property that the other classes in our system do not: They each have two operands. To model this fact, we'll add another abstract class, named `BinaryQuery`, to our hierarchy to represent queries with two operands. The `AndQuery` and `OrQuery` classes will inherit from the `BinaryQuery` class, which in turn will inherit from `Query_base`. These decisions give us the class design represented in Figure 15.3 on the next page.

直接从抽象基类派生 `WordQuery` 和 `NotQuery` 类，`WordQuery` 和 `NotQuery` 类具有系统中其他类所没有的一个性质：它们都有两个操作数。要为此建立模型，将在继承层次中增加另一个名为 `BinaryQuery` 的抽象类，表示带两个操作数的查询。`WordQuery` 和 `NotQuery` 类将继承 `BinaryQuery` 类，`BinaryQuery` 类继承 `Query_base` 类。这些决

定得出了图 15.3 所示的类设计。

Figure 15.3. `Query_base` Inheritance Hierarchy

图 15.3. `Query_base` 继承层次



Operations

操作

Our `Query_base` classes exist mostly to represent kinds of queries; they do little actual work. We'll reuse our `TextQuery` class to store the file, build the query `map`, and search for each word. Our query types need only two operations:

`Query_base` 类的存在主要是为了表示查询类型，不做实际工作。我们将重用 `TextQuery` 类以存储文件、建立查询以及查找每个单词。查询类型只需要两个操作：

1. An `eval` operation to return the `set` of matching line numbers. This operation takes a `TextQuery` object on which to execute the query.

`eval` 操作，返回匹配行号编号的集合。该操作接受 `TextQuery` 对象，在 `TextQuery` 对象上执行查询。

2. A `display` operation that takes a reference to an `ostream` and prints the query that a given object performs on that stream.

`display` 操作，接受 `ostream` 引用并打印给定对象在该 `ostream` 上执行的查询。

We'll define each of these operations as pure `virtual` functions ([Section 15.6](#), p. 595) in the `Query_base` class. Each of our derived classes will have to define its own version of these functions.

我们将这些操作定义为 `Query_base` 中的纯虚函数 ([第 15.6 节](#))，每个派生类都必须对这些函数定义自己的版本。

15.9.2. A Valuelike Handle

15.9.2. 值型句柄

Our program will deal with evaluating queries, not with building them. However, we need to be able to create queries in order to run our program. The simplest way to do so is to write C++ expressions to create queries directly. For example, we'd like to be able to write code such as

程序将处理计算查询，而不建立查询，但是，需要能够创建查询以便运行程序。最简单的办法是编写 C++ 表达式直接创建查询，例如，可以编写这样的代码：

```
Query q = Query("fiery") & Query("bird") | Query("wind");
```

to generate the compound query previously described.

以产生前面描述的复合查询。

This problem description implicitly suggests that user-level code won't use our inherited classes directly. Instead, we'll define a handle class named `Query`, which will hide the hierarchy. User code will execute in terms of the handle; user code will only indirectly manipulate `Query_base` objects.

Section 15.9. Text Queries Revisited

这个问题描述暗示我们，用户级代码将不能直接使用我们的继承层次，相反，我们将定义一个名为 `Query` 的句柄类，用它隐藏继承层次。用户代码将根据句柄执行，用户代码只能间接操纵 `Query_base` 对象。

As with our `Sales_item` handle, our `Query` handle will hold a pointer to an object of a type in an inheritance hierarchy. The `Query` class will also point to a use count, which we'll use to manage the object to which the handle points.

像 `Sales_item` 句柄一样，`Query` 句柄将保存指向继承层次中一个类型的对象的指针，`Query` 类还指向一个使用计数，我们用这个使用计数管理句柄指向的对象。

In this case, our handle will completely hide the underlying inheritance hierarchy. Users will create and manipulate `Query_base` objects only indirectly through operations on `Query` objects. We'll define three overloaded operators on `Query` objects and a `Query` constructor that will dynamically allocate a new `Query_base` object. Each operator will bind the generated `Query_base` object to a `Query` handle: The `&` operator will generate a `Query` bound to a new `AndQuery`; the `|` operator will generate a `Query` bound to a new `OrQuery`; and the `~` operator will generate a `Query` bound to a new `NotQuery`. We'll give `Query` a constructor that takes a `string`. This constructor will generate a new `WordQuery`.

在这种情况下，句柄将完全屏蔽基础继承层次，用户将只能间接地通过 `Query` 对象的操作创建和操纵 `Query_base` 对象。我们将定义 `Query` 对象的三个重载操作符以及 `Query` 构造函数，`Query` 构造函数将动态分配新的 `Query_base` 对象。每个操作符将生成的对象绑定到 `Query` 句柄：`&` 操作符将生成绑定到新的 `AndQuery` 对象的 `Query` 对象；`|` 操作符将生成绑定到新的 `OrQuery` 对象的 `Query` 对象；`~` 操作符将生成绑定到新的 `NotQuery` 对象的 `Query` 对象。给 `Query` 定义一个参数为 `string` 对象的构造函数，该构造函数将生成新的 `WordQuery`。

The `Query` class will provide the same operations as the `Query_base` classes: `eval` to evaluate the associated query, and `display` to print the query. It will define an overloaded output operator to display the associated query.

`Query` 类将提供与 `Query_base` 类同样的操作：`eval` 对相关查询进行计算，`display` 打印查询。它将定义重载输出操作符显示相关查询。

Table 15.1. Query Program Design: A Recap

表 15.1. 查询程序设计：扼要重述

<code>TextQuery</code>	Class that reads a specified file and builds an associated lookup map. That class provides a <code>query_text</code> operation that takes a <code>string</code> argument and returns a <code>set</code> of line numbers on which the argument appears. 读指定文件并建立数得上映射的类，该类提供 <code>query_text</code> 操作，该操作接受 <code>string</code> 实参并返回一个 <code>set</code> ，保存出现实参的行的编号。
<code>Query_base</code>	Abstract base class for the query classes. 查询类的抽象基类。
<code>Query</code>	Use-counted handle class, which points to an object of a type derived from <code>Query_base</code> . 用户计数的句柄类，它指向 <code>Query_base</code> 派生类型的对象。
<code>WordQuery</code>	Class derived from <code>Query_base</code> that looks for a given word. 从 <code>Query_base</code> 派生的类，查找给定单词。
<code>NotQuery</code>	Class derived from <code>Query_base</code> that returns the set of lines in which its <code>Query</code> operand does not appear. 从 <code>Query_base</code> 派生的类，返回操作数不出现的行的编号集合。
<code>BinaryQuery</code>	Abstract base type derived from <code>Query_base</code> that represents queries with two <code>Query</code> operands. 从 <code>Query_base</code> 派生的抽象基类类型，表示带两个 <code>Query</code> 操作数的查询。
<code>OrQuery</code>	Class derived from <code>BinaryQuery</code> that returns the union of the line numbers in which its two operands appear. 从 <code>BinaryQuery</code> 派生的类，返回两个操作数出现的行编号集的并集。
<code>AndQuery</code>	Class derived from <code>BinaryQuery</code> that returns the intersection of the line numbers in which its two operands appear. 从 <code>BinaryQuery</code> 派生的类，返回两个操作数出现的行编号集的交集。
<code>q1 & q2</code>	Returns a <code>Query</code> bound to a new <code>AndQuery</code> object that holds <code>q1</code> and <code>q2</code> . 返回 <code>Query</code> 对象，该 <code>Query</code> 对象绑定到保存 <code>q1</code> 和 <code>q2</code> 的新 <code>AndQuery</code> 对象。
<code>q1 q2</code>	Returns a <code>Query</code> bound to a new <code>OrQuery</code> object that holds <code>q1</code> and <code>q2</code> . 返回 <code>Query</code> 对象，该 <code>Query</code> 对象绑定到保存 <code>q1</code> 和 <code>q2</code> 的新 <code>OrQuery</code> 对象。

Section 15.9. Text Queries Revisited

<code>~q</code>	Returns a <code>Query</code> bound to a new <code>NotQuery</code> object that holds <code>q</code> . 返回 <code>Query</code> 对象，该 <code>Query</code> 对象绑定到保存 <code>q</code> 的新 <code>NotQuery</code> 对象。
<code>Query q(s)</code>	Binds the <code>Query q</code> to a new <code>WordQuery</code> that holds the <code>string s</code> . 将 <code>Query q</code> 绑定到保存 <code>string s</code> 的新 <code>WordQuery</code> 对象。

Our Design: A Recap

我们的设计：扼要重述



It is often the case, especially when new to designing object-oriented systems, that understanding the design is the hardest part. Once we're comfortable with the design, the implementation flows naturally.

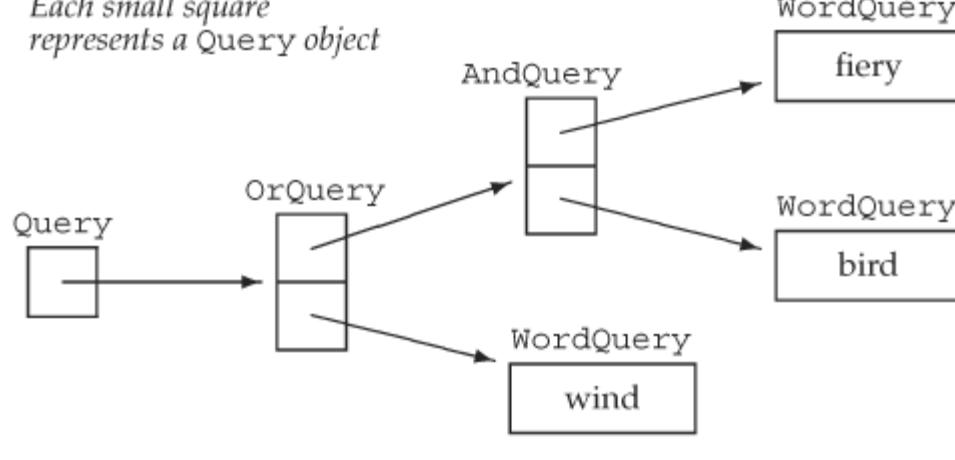
理解设计经常是最困难的部分，尤其是刚开始设计面向对象系统时，一旦熟悉了设计，实现就是顺理成章的了。

It is important to realize that much of the work in this application consists of building objects to represent the user's query. As illustrated in [Figure 15.4](#) on the following page, an expression such as

这个应用程序的主要工作由建立对象表示用户的查询构成，认识到这一点很重要。正如[图 15.4](#) 所示，表达式

Figure 15.4. Objects Created by `Query` Expressions

图 15.4. `Query` 表达式创建的对象



```
Query q = Query("fiery") & Query("bird") | Query("wind");
```

generates ten objects: five `Query_base` objects and their associated handles. The five `Query_base` objects are three `WordQuery`s, an `OrQuery`, and an `AndQuery`.

生成 10 个对象：5 个 `Query_base` 对象及其相关联的句柄。5 个 `Query_base` 对象分别是 3 个 `WordQuery` 对象，一个 `OrQuery` 对象和一个 `AndQuery` 对象。

Once the tree of objects is built up, evaluating (or displaying) a given query is basically a process (managed for us by the compiler) of following these links, asking each object in the tree to evaluate (or display) itself. For example, if we call `eval` on `q` (i.e., on the root of this tree), then `eval` will ask the `OrQuery` to which it points to `eval` itself. Evaluating this `OrQuery` calls `eval` on its two operands, which in turn calls `eval` on the `AndQuery` and `WordQuery` that looks for the word `wind`, and so on.

Section 15.9. Text Queries Revisited

一旦建立了对象树，计算（或显示）给定查询基本上是沿着这些链接，要求树中每个对象计算（或显示）自己的过程，该过程由编译器管理。例如，如果调用 `q`（即，在这棵树的树根）的 `eval`，则 `eval` 将要求 `q` 指向的 `OrQuery` 对象调用 `eval` 来计算自己，计算这个 `OrQuery` 对象用两个操作数调用 `eval`，这个依次调用 `AndQuery` 对象和 `WordQuery` 对象的 `eval`，查找单词 `wind`，依此类推。

```
Objects Created by the Expression
Query("fiery") & Query("bird") | Query("wind");
```

Exercises Section 15.9.2

Exercise

15.39: Given that `s1`, `s2`, `s3` and `s4` are all `strings`, determine what objects are created in the following uses of the `Query` class:

给定 `s1`、`s2`、`s3` 和 `s4` 均为 `string` 对象，确定下述 `Query` 类的使用创建什么对象：

- (a) `Query(s1) | Query(s2) & ~Query(s3);`
- (b) `Query(s1) | (Query(s2) & ~Query(s3));`
- (c) `(Query(s1) & (Query(s2) | (Query(s3) & Query(s4))));`

15.9.3. The `Query_base` Class

15.9.3. `Query_base` 类

Now that we've explained our design, we'll start our implementation by defining the `Query_base` class:

现在我们的设计已经解释清楚了，该开始实现了。首先来定义 `Query_base` 类：

```
// private, abstract class acts as a base class for concrete query types
class Query_base {
    friend class Query;
protected:
    typedef TextQuery::line_no line_no;
    virtual ~Query_base() { }
private:
    // eval returns the /set/ of lines that this Query matches
    virtual std::set<line_no>
        eval(const TextQuery&) const = 0;
    // display prints the query
    virtual std::ostream&
        display(std::ostream& = std::cout) const = 0;
};
```

The class defines two interface members: `eval` and `display`. Both are pure `virtual` functions ([Section 15.6](#), p. 595), which makes this class abstract. There will be no objects of type `Query_base` in our applications.

这个类定义了两个接口成员：`eval` 和 `display`。两个成员都是纯虚函数（[第 15.6 节](#)），因此该类为抽象类，应用程序中将没有 `Query_base` 类型的对象。

Users and the derived classes will use the `Query_base` class only through the `Query` handle. Therefore, we made our `Query_base` interface `private`. The (`virtual`) destructor ([Section 15.4.4](#), p. 587) and the `typedef` are `protected` so that the derived types can access these members. The destructor is used (implicitly) by the derived-class destructors and so must be accessible to them.

用户和派生类将只通过 `Query` 句柄使用 `Query_base` 类，因此，将 `Query_base` 接口设为 `private`。（虚）析构函数（[第 15.4.4 节](#)）和类型别名为 `protected`，这些派生类型就能够访问这些成员，构造函数由派生类构造函数（隐式）使用，因此派生类必须能够访问构造函数。

We grant friendship to the `Query` handle class. Members of that class will call the `virtuals` in `Query_base` and so must have access to them.

给 `Query` 句柄类授予友元关系，该类的成员将调用 `Query_base` 中的虚函数因此必须能够访问它们。

15.9.4. The `Query` Handle Class

15.9.4. `Query` 句柄类

Section 15.9. Text Queries Revisited

Our `Query` handle will be similar to the `Sales_item` class in that it will hold a pointer to the `Query_base` and a pointer to a use count. As in the `Sales_item` class, the copy-control members of `Query` will manage the use count and the `Query_base` pointer.

`Query` 句柄将类似于 `Sales_item` 类，因为它将保存 `Query_base` 指针和使用计数指针。像 `Sales_item` 类一样，`Query` 的复制控制成员将管理使用计数和 `Query_base` 指针。

Unlike the `Sales_item` class, `Query` will provide the only interface to the `Query_base` hierarchy. Users will not directly access any of the members of `Query_base` or its derived classes. This design decision leads to two differences between `Query` and `Sales_item`. The first is that the `Query` class won't define overloaded versions of dereference and arrow operators. The `Query_base` class has no `public` members. If the `Query` handle defined the dereference or arrow operators, they would be of no use! Any attempt to use those operators to access a `Query_base` member would fail. Instead, `Query` must define its own versions of the `Query_base` interface functions `eval` and `display`.

与 `Sales_item` 类不同的是，`Query` 类将只为 `Query_base` 继承层次提供接口。用户将不能直接访问 `Query` 或其派生类的任意成员，这一设计决定导致 `Query` 和 `Sales_item` 之间存在两个区别。第一个区别是，`Query` 类将不定义解引用操作符和箭头操作符的重载版本。`Query_base` 类没有 `public` 成员，如果 `Query` 句柄定义了解引用操作符和箭头操作符，它们将没有用处！使用那些操作符访问成员的任何尝试都将失败，相反，`Query` 类必须定义接口函数 `eval` 和 `display` 的自身版本。

The other difference results from how we intend objects of the hierarchy to be created. Our design says that objects derived from `Query_base` will be created only through operations on the `Query` handle. This difference results in different constructors being required for the `Query` class than were used in the `Sales_item` handle.

另一个区别来自于我们打算怎样创建继承层次的对象。我们的设计指出将只通过 `Query` 句柄的操作创建 `Query_base` 的派生类对象，这个区别导致 `Query` 类需要与 `Sales_item` 句柄中所用的构造函数不同的构造函数。

The `Query` Class

`Query` 类

Given the preceding design, the `Query` class itself is quite simple:

按照前面的设计，`Query` 类本身相当简单：

```
// handle class to manage the Query_base inheritance hierarchy
class Query {
    // these operators need access to the Query_base* constructor
    friend Query operator-(const Query &);
    friend Query operator|(const Query&, const Query&);
    friend Query operator&(const Query&, const Query&);

public:
    Query(const std::string&); // builds a new WordQuery
    // copy control to manage pointers and use counting
    Query(const Query &c): q(c.q), use(c.use) { ++*use; }
    ~Query() { decr_use(); }
    Query& operator=(const Query&);

    // interface functions: will call corresponding Query_base operations
    std::set<TextQuery::line_no>
    eval(const TextQuery &t) const { return q->eval(t); }
    std::ostream &display(std::ostream &os) const
        { return q->display(os); }

private:
    Query(Query_base *query):
        q(query),
        use(new std::size_t(1)) { }
    Query_base *q;
    std::size_t *use;
    void decr_use()
        { if (--*use == 0) { delete q; delete use; } }
};


```

We start by naming as friends the operators that create `Query` objects. We'll see shortly why these operators need to be friends.

首先指定创建 `Query` 对象的操作符为友元，我们将很快看快看到为什么需要将这些操作符设为友元。

In the `public` interface for `Query`, we declare, but cannot yet define, the constructor that takes a `string`. That constructor creates a `WordQuery` object, so we cannot define the constructor until we have defined the `WordQuery` class.

在 `Query` 类的 `public` 接口中，声明了但没有定义接受 `string` 对象的构造函数，该构造函数创建 `WordQuery` 对象，因此在定义 `WordQuery` 类之前不能定义它。

The next three members handle copy control and are the same as the corresponding members of the `Sales_item` class.

后面三个成员处理复制控制，与 `Sales_item` 类中的对应成员相同。

The last two `public` members represent the interface for `Query_base`. In each case, the `Query` operation uses its `Query_base` pointer to call the respective `Query_base` operation. These operations are virtual. The actual version that is called is determined at run time and will depend on the type of the object to which `q` points.

最后两个 `public` 成员表示对 `Query_base` 类的接口。每种情况下，`Query` 操作都使用它的 `Query_base` 指针调用相应 `Query_base` 操作。这些操作是虚函数，在运行时根据 `q` 指向的对象的类型确定调用的实际版本。

Section 15.9. Text Queries Revisited

The `private` implementation of `Query` includes a constructor that takes a pointer to a `Query_base` object. This constructor stores in `q` the pointer it is given and allocates a new use counter, which it initializes to one. This constructor is `private` because we don't intend general user code to define `Query_base` objects. Instead, the constructor is needed for the operators that create `Query` objects. Because the constructor is `private`, the operators had to be made friends.

`Query` 类实现的 `private` 部分包括一个接受 `Query_base` 对象指针的构造函数，该构造函数将获得的指针存储在 `q` 中并分配新的使用计数，将使用计数初始化为 1。该构造函数为 `private`，是因为我们不希望普通用户代码定义 `Query_base` 对象，相反，创建 `Query` 对象的操作符需要这个构造函数。构造函数为 `private`，所以必须将操作符设为友元。

The `Query` Overloaded Operators

`Query` 重载操作符

The `|`, `&` and `~` operators create `OrQuery`, `AndQuery`, and `NotQuery` objects, respectively:

`|`、`&` 和 `~` 操作符分别创建 `OrQuery`、`AndQuery` 和 `NotQuery` 对象：

```
inline Query operator&(const Query &lhs, const Query &rhs)
{
    return new AndQuery(lhs, rhs);
}
inline Query operator|(const Query &lhs, const Query &rhs)
{
    return new OrQuery(lhs, rhs);
}
inline Query operator~(const Query &oper)
{
    return new NotQuery(oper);
}
```

Each of these operations dynamically allocates a new object of a type derived from `Query_base`. The `return` (implicitly) uses the `Query` constructor that takes a pointer to a `Query_base` to create the `Query` object from the `Query_base` pointer that the operation allocates. For example the `return` statement in the `~` operator is equivalent to

每个操作符动态分配 `Query_base` 派生类型的新对象，`return` 语句（隐式）使用接受 `Query_base` 指针的 `Query` 构造函数，用操作分配的 `Query_base` 指针创建 `Query` 对象。例如，`~` 操作符中的 `return` 语句等价于：

```
// allocate a new Not Query object
// convert the resulting pointer to NotQuery to a pointer to Query_base
Query_base *tmp = new NotQuery(expr);

return Query(tmp); // use Query constructor that takes a pointer to Query_base
```

There is no operator to create a `WordQuery`. Instead, we gave our `Query` class a constructor that takes a `string`. That constructor generates a `WordQuery` to look for the given `string`.

没有操作符创建 `WordQuery` 对象，相反，为 `Query` 类定义一个接受 `string` 对象的构造函数，该构造函数生成 `WordQuery` 对象查找给定 `string`。

The `Query` Output Operator

`Query` 输出操作符

We'd like users to be able to print `Query`s using the normal (overloaded) output operator. However, we also need the print operation to be virtual printing a `Query` should print the `Query_base` object to which the `Query` points. There's only one problem: only member functions can be virtual, but the output operator cannot be a member of the `Query_base` classes ([Section 14.2.1](#), p. 514).

我们希望用户可以用标准（重载的）输出操作符打印 `Query` 对象，但是，也需要打印操作是虚函数——打印 `Query` 对象应打印 `Query` 对象指向的 `Query_base` 对象。这里存在一个问题：只有成员函数可以为虚函数，但输出操作符不能是 `Query_base` 类的成员（[第 14.2.1 节](#)）。

To obtain the necessary virtual behavior, our `Query_base` classes defined a virtual `display` member, which the `Query` output operator will use:

要获得必要的虚函数行为，`Query_base` 类定义了一个虚函数成员 `display`，`Query` 输出操作符将使用它：

```
inline std::ostream&
operator<<(std::ostream &os, const Query &q)
{
    return q.display(os);
}
```

When we write

如果编写

```
Query andq = Query(sought1) & Query(sought2);
cout << "\nExecuted query: " << andq << endl;
```

the `Query` output operator is invoked. That operator calls

将调用 `Query` 输出操作符，该操作符调用

```
q.display(os)
```

with `q` referring to the `Query` object that points to this `AndQuery`, an `dos` bound to `cout`. When we write

其中，`q` 引用指向该 `AndQuery` 对象的 `Query` 对象，`os` 绑定到 `cout`。如果编写

```
Query name(sought);
cout << "\nExecuted Query for: " << name << endl;
```

the `WordQuery` instance of `display` is called. More generally, a call

将调用 `display` 的 `WordQuery` 实例。更一般的，以下代码

```
Query query = some_query;
cout << query << endl;
```

invokes the instance of `display` associated with the object that `query` addresses at that point in the execution of our program.

将调用程序运行到此时与 `query` 所指对象相关联的 `display` 实例。

15.9.5. The Derived Classes

15.9.5. 派生类

We next need to implement our concrete query classes. The one interesting part about these classes is how they are represented. The `WordQuery` class is most straightforward. Its job is to hold the search word.

下面要实现具体的查询类。关于这些类，一个有趣的部分是如何表示它们。`WordQuery` 类最直接，它的工作是保存要查找的单词。

The other classes operate on one or two `Query` operands. A `NotQuery` negates the result of another `Query`. Both `AndQuery` and `OrQuery` have two operands, which are actually stored in their common base class, `BinaryQuery`.

其他类操作一个或两个 `Query` 操作数。`NotQuery` 对象对别的 `Query` 对象的结果求反，`AndQuery` 类和 `OrQuery` 类都有两个操作数，操作数实际存储在它们的公共基类 `BinaryQuery` 中。

In each of these classes, the operand(s) could be an object of any of the concrete `Query_base` classes: A `NotQuery` could be applied to a `WordQuery`, an `AndQuery`, an `OrQuery`, or another `NotQuery`. To allow this flexibility, the operands must be stored as pointers to `Query_base` that might point to any one of the concrete `Query_base` classes.

在这些类当中，操作数都可以是任意具体 `Query_base` 类的对象：`NotQuery` 对象可以应用于 `WordQuery` 对象、`AndQuery` 对象、`OrQuery` 对象或其他 `NotQuery` 对象。要允许这种灵活性，操作数必须存储为 `Query_base` 指针，它可以指向任意具体的 `Query_base` 类。

However, rather than storing a `Query_base` pointer, our classes will themselves use the `Query` handle. Just as user code is simplified by using a handle, we can simplify our own class code by using the same handle class. We'll make the `Query` operand `const` because once a given `Query_base` object is built, there are no operations that can change the operand(s).

但是，我们的类不存储 `Query_base` 指针，而是自己使用 `Query` 句柄。正如使用句柄可以简化用户代码，也可以使用同样的句柄类简化类代码。将 `Query` 操作数设为 `const`，因为一旦创立了 `Query_base` 对象，就没有操作可以改变操作数了。

Now that we know the design for these classes, we can implement them.

了解了这些类的设计之后，就可以实现它们了。

The `WordQuery` Class

WordQuery 类

A **WordQuery** is a kind of **Query_base** that looks for a specified word in a given query map:

WordQuery 是一种 **Query_base**, 它在给定的查询映射中查找指定单词:

```
class WordQuery: public Query_base {
    friend class Query; // Query uses the WordQuery constructor
    WordQuery(const std::string &s): query_word(s) { }
    // concrete class: WordQuery defines all inherited pure virtual functions
    std::set<line_no> eval(const TextQuery &t) const
        { return t.run_query(query_word); }
    std::ostream& display(std::ostream &os) const
        { return os << query_word; }
    std::string query_word; // word for which to search
};
```

Like **Query_base**, **WordQuery** has no **public** members; **WordQuery** must make **Query** a friend to allow **Query** to access the **WordQuery** constructor.

像 **Query_base** 类一样, **WordQuery** 类没有 **public** 成员, **WordQuery** 为必须将 **Query** 类设为友元以允许 **Query** 访问 **WordQuery** 构造函数。

Each of the concrete query classes must define the inherited pure virtual functions. The **WordQuery** operations are simple enough to define in the class body. The **eval** member calls the **query_text** member of its **TextQuery** parameter passing it the **string** that was used to create this **WordQuery**. To **display** a **WordQuery**, we print the **query_word**.

每个具体的查询类必须定义继承的纯虚函数。**WordQuery** 类的操作足够简单, 可以定义在类定义中。**eval** 成员调用其 **TextQuery** 形参的 **query_word** 成员, 将用于创建该 **WordQuery** 对象的 **string** 对象传给它。要 **display** 一个 **WordQuery** 对象, 就打印 **query_word** 对象。

The **NotQuery** Class**NotQuery** 类

A **NotQuery** holds a **const Query**, which it negates:

NotQuery 对象保存一个 **const Query** 对象, 对它求反:

```
class NotQuery: public Query_base {
    friend Query operator~(const Query &);
    NotQuery(Query q): query(q) { }
    // concrete class: NotQuery defines all inherited pure virtual functions
    std::set<line_no> eval(const TextQuery&) const;
    std::ostream& display(std::ostream &os) const
        { return os << "~(" << query << ")"; }
    const Query query;
};
```

The **Query** overloaded **~** operator is made a friend to allow that operator to create a new **NotQuery** object. To **display** a **NotQuery**, we print the **~** symbol followed by the underlying **Query**. We parenthesize the output to ensure that precedence is clear to the reader.

将 **Query** 的重载 **~** 操作符设为友元, 从而允许该操作符创建新的 **NotQuery** 对象。为了 **display** 一个 **NotQuery** 对象, 打印 **~** 对象, 将输出用圆括号括住以保证读者清楚优先级。



The use of the output operator in the **display** operation is ultimately a virtual call to a **Query_base** object:

display 操作中输出操作符的使用最终是对 **Query_base** 对象的虚函数调用:

```
// uses the Query output operator, which calls Query::display
// that function makes a virtual call to Query_base::display
{ return os << "~(" << query << ")" }
```

The **eval** member is complicated enough that we will implement it outside the class body. The **eval** function appears in [Section 15.9.6](#) (p. 620).

eval 成员比较复杂, 我们将在类定义体之外实现它, **eval** 函数见第 [15.9.6](#) 节。

The `BinaryQuery` Class

`BinaryQuery` 类

The `BinaryQuery` class is an abstract class that holds the data needed by the two query types, `AndQuery` and `OrQuery`, that operate on two operands:

`BinaryQuery` 类是一个抽象类，保存 `AndQuery` 和 `OrQuery` 两个查询类型所需的数据，`AndQuery` 和 `OrQuery` 有两个操作数：

```
class BinaryQuery: public Query_base {
protected:
    BinaryQuery(Query left, Query right, std::string op):
        lhs(left), rhs(right), oper(op) { }
    // abstract class: BinaryQuery doesn't define eval
    std::ostream& display(std::ostream &os) const
    { return os << "(" << lhs << " " << oper << " "
      << rhs << ")"; }
    const Query lhs, rhs; // right- and left-hand operands
    const std::string oper; // name of the operator
};
```

The data in a `BinaryQuery` are the two `Query` operands and the operator symbol to use when displaying the query. These data are all declared `const`, because the contents of a query should not change once it has been constructed. The constructor takes the two operands and the operator symbol, which it stores in the appropriate data members.

`BinaryQuery` 中的数据是两个 `Query` 操作数，以及显示查询时使用的操作符符号。这些数据均声明为 `const`，因为一旦建立了查询的内容就不应该再改变。构造函数接受两个操作数以及操作符符号，将它们存储在适当的数据成员中。

To `display` a `BinaryOperator`, we print the parenthesized expression consisting of the left-hand operand, followed by the operator, followed by the right-hand operand. As when we displayed a `NotQuery`, the overloaded `<<` operator that is used to print `left` and `right` ultimately makes a virtual call to the underlying `Query_base display`.

要显示一个 `BinaryOperator` 对象，打印由圆括号括住的表达式、该表达式由左操作数后接操作、再接右操作数构成。像显示 `NotQuery` 对象一样，用于打印 `left` 和 `right` 的重载 `<<` 操作符最终对基础 `Query_base` 对象的 `display` 进行虚函数调用。



The `BinaryQuery` class does not define the `eval` function and so inherits a pure virtual. As such, `BinaryQuery` is also an abstract class, and we cannot create objects of `BinaryQuery` type.

`BinaryQuery` 类没有定义 `eval` 函数，因此继承了一个纯虚函数。这样，`BinaryQuery` 也是一个抽象类，不能创建 `BinaryQuery` 类型的对象。

The `AndQuery` and `OrQuery` Classes

`AndQuery` 和 `OrQuery` 类

The `AndQuery` and `OrQuery` classes are nearly identical:

`AndQuery` 类和 `OrQuery` 类几乎完全相同：

```
class AndQuery: public BinaryQuery {
    friend Query operator&(const Query&, const Query&);
    AndQuery (Query left, Query right):
        BinaryQuery(left, right, "&") { }
    // concrete class: And Query inherits display and defines remaining pure virtual
    std::set<line_no> eval(const TextQuery&) const;
};

class OrQuery: public BinaryQuery {
    friend Query operator|(const Query&, const Query&);
    OrQuery(Query left, Query right):
        BinaryQuery(left, right, "|") { }
    // concrete class: OrQuery inherits display and defines remaining pure virtual
    std::set<line_no> eval(const TextQuery&) const;
};
```

These classes make the respective operator a friend and define a constructor to create their `BinaryQuery` base part with the appropriate operator. They inherit the `BinaryQuery` definition of `display`, but each defines its own version of the `eval` function.

这两个类将各自的操作符设为友元，并定义了构造函数用适当的操作符创建它们的 `BinaryQuery` 基类部分。它们继承 `BinaryQuery` 类的 `display` 函数定义，但各自定义了自

己的 `eval` 函数版本。

Exercises Section 15.9.5

Exercise For the expression built in [Figure 15.4](#) (p. 612)

15.40:

对图 15.4 中建立的表达式：

- List the constructors executed in processing this expression.
列出处理这个表达式所执行的构造函数。
- List the calls to `display` and to the overloaded `<<` operator that are made in executing `cout << q`.
列出执行 `cout << q` 所调用的 `display` 函数和重载的 `<<` 操作符。
- List the calls to `eval` made when evaluating `q.eval`.
列出计算 `q.eval` 时所调用的 `eval` 函数。

15.9.6. The `eval` Functions

15.9.6. `eval` 函数

The heart of the query class hierarchy are the `eval` virtual functions. Each of these functions calls `eval` on its operand(s) and then applies its own logic: The `AndQuery eval` operation returns the union of the results of its two operands; `OrQuery` returns the intersection. The `NotQuery` is more complicated: It must return the line numbers not in its operand's set.

查询类层次的中心是虚函数 `eval`。每个 `eval` 函数调用其操作数的 `eval` 函数，然后应用自己的逻辑：`AndQuery` 的 `eval` 操作返回两个操作数的结果的并集，`OrQuery` 的 `eval` 操作返回交集，`NotQuery` 的 `eval` 操作比较复杂：它必须返回不在其操作数的集合中的行编号。

`OrQuery::eval`

An `OrQuery` merges the set of line numbers returned by its two operands its result is the union of the results for its two operands:

`OrQuery` 对象合并由它的两个操作数返回的行号编号集合——其结果是它的两个操作数的结果的并集：

```
// returns union of its operands' result sets
set<TextQuery::line_no>
OrQuery::eval(const TextQuery& file) const
{
    // virtual calls through the query handle to get result sets for the operands
    set<line_no> right = rhs.eval(file);
    ret_lines = lhs.eval(file); // destination to hold results
    // inserts the lines from right that aren't already in ret_lines
    ret_lines.insert(right.begin(), right.end());
    return ret_lines;
}
```

The `eval` function starts by calling `eval` on each of its `Query` operands. Those calls call `Query::eval`, which in turn makes a virtual call to `eval` on the underlying `Query_base` object. Each of these calls yields a `set` of line numbers in which its operand appears. We then call `insert` on `ret_lines`, passing a pair of iterators denoting the `set` returned from evaluating the right-hand operand. Because `ret_lines` is a `set`, this call adds the elements from `right` that are not also in `left` into `ret_lines`. After the call to `insert`, `ret_lines` contains each line number that was in either of the `left` or `right` sets. We complete the function by returning `ret_lines`.

`eval` 函数首先调用每个操作数的 `eval` 函数，操作数的 `eval` 函数调用 `Query::eval`，`Query::eval` 再调用基础 `Query_base` 对象的虚函数 `eval`，每个调用获得其操作数出现在其中表示对右操作数求值所返回的 `set`。因为 `ret_lines` 是一个 `set` 对象，这个调用将 `right` 中未在 `left` 中出现的元素加到 `ret_lines` 中。调用 `insert` 函数之后，`ret_lines` 包含在 `left` 集或在 `right` 集的每个行编号。返回 `ret_lines` 而结束 `OrQuery::eval` 函数。

AndQuery::eval

The `AndQuery` version of `eval` uses one of the library algorithms that performs setlike operations. These algorithms are described in the Library Appendix, in [Section A.2.8](#) (p. 821):

`AndQuery` 的 `eval` 版本使用了完成集合式操作的一个标准库算法。标准库附录中说明了这些算法, 见 [A.2.8 节](#)。

```
// returns intersection of its operands' result sets
set<TextQuery::line_no>
AndQuery::eval(const TextQuery& file) const
{
    // virtual calls through the Query handle to get result sets for the operands
    set<line_no> left = lhs.eval(file),
                  right = rhs.eval(file);
    set<line_no> ret_lines; // destination to hold results
    // writes intersection of two ranges to a destination iterator
    // destination iterator in this call adds elements to ret
    set_intersection(left.begin(), left.end(),
                     right.begin(), right.end(),
                     inserter(ret_lines, ret_lines.begin())));
    return ret_lines;
}
```

This version of `eval` uses the `set_intersection` algorithm to find the lines in common to both queries: That algorithm takes five iterators: The first four denote two input ranges, and the last denotes a destination. The algorithm writes each element that is in both of the two input ranges into the destination. The destination in this call is an insert iterator ([Section 11.3.1](#), p. 406) which inserts new elements into `ret_lines`.

`eval` 函数这个版本使用 `set_intersection` 算法查找两个查询中的公共行: 该算法接受 5 个迭代器, 前 4 个表示两个输入范围, 最后一个表示目的地。算法将同时在两个输入范围内存在在每个元素写到目的地。该调用的目的地是一个迭代器 ([第 11.3.1 节](#)), 它将新元素插入到 `ret_lines` 中。

NotQuery::eval

`NotQuery` 找出文本行中未出现操作数的每行。为了支持这个函数, 我们需要 `TextQuery` 类增加一个成员返回文件的大小, 以便了解存在什么样的行编号。

`NotQuery` 查找未出现操作数的每个文本行。要支持这个函数, 需要 `TextQuery` 类增加一个成员返回文件的大小, 以便了解存在什么样的行编号。

```
// returns lines not in its operand's result set
set<TextQuery::line_no>
NotQuery::eval(const TextQuery& file) const
{
    // virtual call through the Query handle to eval
    set<TextQuery::line_no> has_val = query.eval(file);
    set<line_no> ret_lines;
    // for each line in the input file, check whether that line is in has_val
    // if not, add that line number to ret_lines
    for (TextQuery::line_no n = 0; n != file.size(); ++n)
        if (has_val.find(n) == has_val.end())
            ret_lines.insert(n);
    return ret_lines;
}
```

As in the other `eval` functions, we start by calling `eval` on this object's operand. That call returns the `set` of line numbers on which the operand appears. What we want is the `set` of line numbers on which the operand does not appear. We obtain that `set` by looking at each line number in the input file. We use the `size` member that must be added to `TextQuery` to control the `for` loop. That loop adds each line number to `ret_lines` that does not appear in `has_val`. Once we've processed all the line numbers, we return `ret_lines`.

像其他 `eval` 函数一样, 首先调用该对象的操作数的 `eval` 函数。该调用返回操作数所出现的行编号的 `set`, 而我们想要的是不出现操作数的行编号的 `set`, 通过查找输入文件的每个行编号获得该 `set`。使用必须加到 `TextQuery` 的 `size` 成员控制 `for` 循环, 该循环将没有在 `has_val` 中出现的每个行编号加到 `ret_lines` 中, 一旦处理完所有的行编号, 就返回 `ret_lines`。

Exercises Section 15.9.6

- Exercise 15.41:** Implement the `Query` and `Query_base` classes, and add the needed `size` operation to the `TextQuery` class from [Chapter 10](#). Test your application by evaluating and printing a query such as the one in [Figure 15.4](#) (p. 612).

实现 `Query` 类和 `Query_base` 类, 并为[第十章](#)的 `TextQuery` 类增加需要的 `size` 操作。通过计算和打印如图 [15.4](#) 所示的查询, 测试你的应用程序。

- Exercise** Design and implement one of the following enhancements:

15.42:

设计并实现下述增强中的一个：

- a. Introduce support for evaluating words based on their presence within the same sentence rather than the same line.
引入基于同一句子而不是同一行计算单词的支持。
- b. Introduce a history system in which the user can refer to a previous query by number, possibly adding to it or combining it with another.
引入历史系统，用户可以用编号查阅前面的查询，并可以在其中增加内容或与其他查询组合。
- c. Rather than displaying the count of matches and all the matching lines, allow the user to indicate a range of lines to display, both for intermediate query evaluation and the final query.
除了显示匹配数目和所有匹配行之外，允许用户对中间查询计算和最终查询指出要显示的行的范围。

Chapter Summary

小结

The ideas of inheritance and dynamic binding are simple but powerful. Inheritance lets us write new classes that share behavior with their base class(es) but redefine that behavior as needed. Dynamic binding lets the compiler decide at run time which version of a function to run based on an object's dynamic type. The combination of inheritance and dynamic binding lets us write type-independent programs that have type-specific behavior.

继承和动态绑定的思想，简单但功能强大。继承使我们能够编写新类，新类与基类共享行为但重定义必要的行为。动态绑定使编译器能够在运行时根据对象的动态类型确定运行函数的那个版本。继承和动态绑定的结合使我们能够编写具有特定类型行为而又独立于类型的程序。

In C++, dynamic binding applies *only* to functions declared as *virtual* when called through a reference or pointer. It is common for C++ programs to define handle classes to interface to an inheritance hierarchy. These classes allocate and manage pointers to objects in the inheritance hierarchy, thus obtaining dynamic behavior while shielding user code from having to deal with pointers.

在 C++ 中，动态绑定仅在通过引用或指针调用时才能应用于声明为虚的函数。C++ 程序定义继承层次接口的句柄类很常见，这些类分配并管理指向继承层次中对象的指针，因此能够使用户代码在无须处理指针的情况下获得动态行为。

Inherited objects are composed of base-class part(s) and a derived-class part. Inherited objects are constructed, copied, and assigned by constructing, copying, and assigning the base part(s) of the object before handling the derived part. Because a derived object contains a base part, it is possible to convert a reference or pointer to a derived type to a reference or pointer to its base type.

继承对象由基类部分和派生类部分组成。继承对象是通过在处理派生部分之前对对象的基类部分进行构造、复制和赋值，而进行构造、复制和赋值的。因为派生类对象包含基类部分，所以可以将派生类型的引用或指针转换为基类类型的引用或指针。

Base classes usually should define a virtual destructor even if the class otherwise has no need for a destructor. The destructor must be virtual if a pointer to a base is ever deleted when it actually addresses a derived-type object.

即使另外不需要析构函数，基类通常也应定义一个虚析构函数。如果经常会在指向基类的指针实际指向派生类对象时删除它，析构函数就必须为虚函数。

Defined Terms

术语

abstract base class (抽象基类)

Class that has or inherits one or more pure virtual functions. It is not possible to create objects of an abstract base-class type. Abstract base classes exist to define an interface. Derived classes will complete the type by defining type-specific implementations for the pure virtuals defined in the base.

具有或继承了一个或多个纯虚函数的类。不能创建抽象基类类型的对象。抽象基类的存在定义了一个接口，派生类将为基类中定义的纯虚函数定义特定类型的实现，以完成类型。

base class (基类)

Class from which another class inherits. The members of the base class become members of the derived class.

从中派生其他类的类。基类的成员成为派生类的成员。

class derivation list (类派生列表)

Used by a class definition to indicate that the class is a derived class. A derivation list includes an optional access level and names the base class. If no access label is specified, the type of inheritance depends on the keyword used to define the derived class. By default, if the derived class is defined with the `struct` keyword, then the base class is inherited `publicly`. If the class is defined using the `class` keyword, then the base class is inherited `privately`.

类定义用它指出类是派生类。派生列表包含可选的访问级别和基类的名字，如果不指定访问标号，继承的类型取决于用来定义派生类的保留字，默认情况下，如果派生类用保留字 `struct` 定义，则仅有继承基类，如果派生类用保留字 `class` 定义，则私有继承基类。

derived class (派生类)

A class that inherits from another class. The members of the base class are also members of the derived class. A derived class can redefine the members of its base and can define new members. A derived-class scope is nested in the scope of its base class(es), so the derived class can access members of the base class directly. Members defined in the derived with the same name as members in the base hide those base members; in particular, member functions in the derived do not overload members from the base. A hidden member in the base can be accessed using the scope operator.

继承了其他类的类。基类的成员也是派生类的成员，派生类可以重定义其基类的成员还可以定义新成员。派生类作用域嵌套的基类作用域中，因此派生类可以直接访问基类的成员。派生类中定义的与基类成员同名的成员屏蔽基类成员，具体而言，派生类中的成员函数不重载基类成员。基类中的被屏蔽成员可以用作用域操作符访问。

direct base class (直接基类)

Synonym for immediate base class.

immediate base class 的同义词。

dynamic binding (动态绑定)

Delaying until run time the selection of which function to run. In C++, dynamic binding refers to the run-time choice of which `virtual` function to run based on the underlying type of the object to which a reference or pointer is bound.

延迟到运行时才选择运行哪个函数。在 C++ 中，动态绑定指的是在运行时基于引用或指针绑定的对象的基础类型而选择运行哪个 `virtual` 函数。

dynamic type (动态类型)

T

ype at run time. Pointers and references to base-class types can be bound to objects of derived type. In such cases the static type is reference (or pointer) to base, but the dynamic type is reference (or pointer) to derived.

运行时类型。基类类型的指针和引用可以绑定到派生类型的对象，在这种情况下，静态类型是基类引用（或指针），但动态类型是派生类引用（或指针）。

handle class (句柄类)

Class that provides an interface to another class. Commonly used to allocate and manage a pointer to an object of an inheritance hierarchy.

Keyterm Defined Terms

提供到其他类的接口的类。一般用于分配和管理继承层次对象的指针。

immediate base class (直接基类)

A base class from which a derived class inherits directly. The immediate base is the class named in the derivation list. The immediate base may itself be a derived class.

被派生类直接继承的基类。直接基类是在派生列表中指定的基类，直接基类本身可以是派生类。

indirect base class (间接基类)

A base class that is not immediate. A class from which the immediate base class inherits, directly or indirectly, is an indirect base class to the derived class.

非直接基类。直接基类直接或间接继承的类是派生类的间接基类。

inheritance hierarchy (继承层次)

Term used to describe the relationships among classes related by inheritance that share a common base class.

描述因共享公共基类的继承而相关联的类之间关系的术语。

object-oriented programming (面向对象编程)

Term used to describe programs that use data abstraction, inheritance, and dynamic binding.

描述使用数据抽象、继承和动态绑定的程序的术语。

polymorphism (多态性)

A term derived from a Greek word that means "many forms." In object-oriented programming, polymorphism refers to the ability to obtain type-specific behavior based on the dynamic type of a reference or pointer.

从意为“多种形态”的希腊单词派生的术语。在面向对象编程中，多态性指的是基于引用或指针的动态类型获得类型明确的行为的能力。

private inheritance (私有继承)

A form of implementation inheritance in which the `public` and `protected` members of a `private` base class are `private` in the derived.

实现继承的一种形式，在这种形式中，`private` 基类的 `public` 和 `protected` 成员在派生类中均为 `private`。

protected access label (受保护访问标号)

Members defined after a `protected` label may be accessed by class members and friends and by the members (but not friends) of a derived class. `protected` members are not accessible to ordinary users of the class.

定义在 `protected` 标号之后的成员可以被类成员、友元和派生类成员（非友元）访问。类的普通用户不能访问 `protected` 成员。

protected inheritance (受保护继承)

In `protected` inheritance the `protected` and `public` members of the base class are `protected` in the derived class.

在受保护继承中，基类的 `protected` 和 `public` 成员在派生类中均为 `protected`。

public inheritance (公有继承)

The `public` interface of the base class is part of the `public` interface of the derived class.

基类的 `public` 接口是派生类的 `public` 接口的一部分。

pure virtual (纯虚函数)

A virtual function declared in the class header using `=0` at the end of the function's parameter list. A pure virtual is one that need not be (but may be) defined by the class. A class with a pure virtual is an abstract class. If a derived class does not define its own version of an inherited pure virtual, then the derived class is abstract as well.

类的头文件中在函数形参表末尾 `=0` 声明的虚函数。类不必（但可以）定义纯虚函数。带纯虚函数的类为抽象类。如果派生类没有定义所继承的纯虚函数的自身版本，则派生类也是抽象类。

refactoring (重构)

Redesigning programs to collect related parts into a single abstraction, replacing the original code by uses of the new abstraction. In OO

programs, refactoring frequently happens when redesigning the classes in an inheritance hierarchy. Refactoring often occurs in response to a change in requirements. In general, classes are refactored to move data or function members to the highest common point in the hierarchy to avoid code duplication.

重新设计程序将相关部分集合到一个抽象中，用新抽象的使用代替原来的代码。在面向对象编程中，重新设计继承层次中的类时经常发生重构。响应需求的改变通常需要进行重构。一般而言，对类进行重构时，需要将数据或函数成员移到继承层次的最高公共点以避免代码重复。

sliced (切掉的)

Term used to describe what happens when an object of derived type is used to initialize or assign an object of the base type. The derived portion of the object is "sliced down," leaving only the base portion, which is assigned to the base.

描述用派生类型对象对基类类型对象进行初始化或赋值时情况的术语。对象的派生部分被“切掉”，只留下基类部分，赋给基类对象。

static type (静态类型)

Compile-time type. Static type of an object is the same as its dynamic type. The dynamic type of an object to which a reference or pointer refers may differ from the static type of the reference or pointer.

编译时类型。对象的静态类型与动态类型相同。引用或指针所引用的对象的动态类型可以不同于引用或指针的静态类型。

virtual function (虚函数)

A member function that defines type-specific behavior. Calls to a virtual made through a reference or pointer are resolved at run time, based on the type of the object to which the reference or pointer is bound.

定义类型明确的行为的成员函数。通过引用或指针进行的虚函数调用，在运行时基于引用或指针定制对象而确定。

Chapter 16. Templates and Generic Programming

第十六章 模板和泛型编程

CONTENTS

Section 16.1 Template Definitions	624
Section 16.2 Instantiation	636
Section 16.3 Template Compilation Models	643
Section 16.4 Class Template Members	647
Section 16.5 A Generic Handle Class	666
Section 16.6 Template Specializations	671
Section 16.7 Overloading and Function Templates	679
Chapter Summary	683
Defined Terms	683

Generic programming involves writing code in a way that is independent of any particular type. When we use a generic program we supply the type(s) or value(s) on which that instance of the program will operate. The library containers, iterators, and algorithms described in [Part II](#) are examples of generic programming. There is a single definition of each container, such as `vector`, but we can define many different kinds of `vectors` that differ by the element type that the `vector` contains.

所谓泛型编程就是以独立于任何特定类型的方式编写代码。使用泛型程序时，我们需要提供具体程序实例所操作的类型或值。[第二部分](#)中描述的标准库的容器、迭代器和算法都是泛型编程的例子。每种容器(如 `vector`)都有单一的定义，但可以定义许多不同种类的 `vector`，它们的区别在于所包含的元素类型。

Templates are the foundation of generic programming. We can, and have, used templates without understanding how they are defined. In this chapter we'll see how we can define our own template classes and functions.

模板是泛型编程的基础。使用模板时可以无须了解模板的定义。本章将介绍怎样定义自己的模板类和模板函数。

Generic programming, like object-oriented programming, relies on a form of polymorphism. The polymorphism in OOP applies at run time to classes related by inheritance. We can write code that uses such classes in ways that ignore the type differences among the base and derived classes. As long as we use references or pointers to the base type, we can use the same code on objects of the base type or a type derived from that type.

泛型编程与面向对象编程一样，都依赖于某种形式的多态性。面向对象编程中的多态性在运行时应用于存在继承关系的类。我们能够编写使用这些类的代码，忽略基类与派生类之间类型上的差异。只要使用基类的引用或指针，基类类型或派生类类型的对象就可以使用相同的代码。

Generic programming lets us write classes and functions that are polymorphic across unrelated types at compile time. A single class or function can be used to manipulate objects of a variety of types. The standard library containers, iterators, and algorithms are good examples of generic programming. The library defines each of the containers, iterators, and algorithms in a type-independent manner. We can use library classes and functions on most any kind of type. For example, we can define a `vector` of `Sales_item` objects even though the designers of `vector` could have had no knowledge of our application-specific class.

在泛型编程中，我们所编写的类和函数能够多态地用于跨越编译时不相关的类型。一个类或一个函数可以用来操纵多种类型的对象。标准库中的容器、迭代器和算法是很好的泛型编程的例子。标准库用独立于类型的方式定义每个容器、迭代器和算法，因此几乎可以在任意类型上使用标准库的类和函数。例如，虽然 `vector` 的设计者不可能了解应用程序特定的类，但我们能够定义 `Sales_item` 对象组成的 `vector`。

In C++, templates are the foundation for generic programming. A template is a blueprint or formula for creating a class or a function. For example, the standard library defines a single class template that defines what it means to be a `vector`. That template is used to generate any number of type-specific `vector` classes for example, `vector<int>` or `vector<string>`. Part II showed how to use generic types and functions; this chapter shows how we can define our own templates.

在 C++ 中，模板是泛型编程的基础。模板是创建类或函数的蓝图或公式。例如，标准库定义了一个类模板，该模板定义了 `vector` 的含义，它可用于产生任意数量的特定类型的 `vector` 类，例如，`vector<int>` 或 `vector<string>`。本书[第二部分](#)介绍了怎样使用泛型类型和泛型函数，本章将介绍怎样自定义模板。

16.1. Template Definitions

16.1. 模板定义

Let's imagine that we want to write a function to compare two values and indicate whether the first is less than, equal to, or greater than the second. In practice, we'd want to define several such functions, each of which could compare values of a given type. Our first attempt might be to define several overloaded functions:

假设想要编写一个函数比较两个值并指出第一个值是小于、等于还是大于第二个值。实践中，我们可能希望定义几个这样的函数，每一个可以比较一种给定类型的值，第一次尝试可能是定义几个重载函数：

```
// returns 0 if the values are equal, -1 if v1 is smaller, 1 if v2 is smaller
int compare(const string &v1, const string &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
int compare(const double &v1, const double &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

These functions are nearly identical: The only difference between them is the type of their parameters. The function body is the same in each function.

这些函数几乎相同，它们之间唯一的区别是形参的类型，每个函数的函数体是相同的。

Having to repeat the body of the function for each type that we compare is tedious and error-prone. More importantly, we need to know *in advance* all the types that we might ever want to `compare`. This strategy cannot work if we want to be able to use the function on types that we don't know about.

每个要比较的类型都需要重复函数的函数体，不仅麻烦而且容易出错。更重要的是，需要事先知道空间可能会比较哪些类型。如果希望将函数用于未知类型，这种策略就不起作用了。

16.1.1. Defining a Function Template

16.1.1. 定义函数模板

Rather than defining a new function for each type, we can define a single [function template](#). A function template is a type-independent function that is used as a formula for generating a type-specific version of the function. For example, we might write a function template named `compare`, which would tell the compiler how to generate specific versions of `compare` for the types that we want to compare.

我们可以不用为每个类型定义一个新函数，而是只定义一个函数模板（function template）。函数模板是一个独立于类型的函数，可作为一种方式，产生函数的特定类型版本。例如，可以编写名为 `compare` 的函数模板，它告诉编译器如何为我们想要比较的类型产生特定的 `compare` 版本。

The following is a template version of `compare`:

下面是 `compare` 的模板版本：

```
// implement strcmp-like generic compare function
// returns 0 if the values are equal, 1 if v1 is larger, -1 if v1 is smaller
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

A template definition starts with the keyword `template` followed by a [template parameter list](#), which is a comma-separated list of one or more [template parameters](#) bracketed by the less-than (<) and greater-than (>) tokens.

模板定义以关键字 `template` 开始，后接[模板形参表](#)，模板形参表是用尖括号括住的一个或多个[模板形参](#)的列表，形参之间以逗号分隔。

Section 16.1. Template Definitions



The template parameter list cannot be empty.

模板形参表不能为空。

Template Parameter List

模板形参表

The template parameter list acts much like a function parameter list. A function parameter list defines local variable(s) of a specified type but leaves those variables uninitialized. At run time, arguments are supplied that initialize the parameters.

模板形参表很像函数形参表，函数形参表定义了特定类型的局部变量但并不初始化那些变量，在运行时再提供实参来初始化形参。

Analogously, template parameters represent types or values we can use in the definition of a class or function. For example, our `compare` function declares one type parameter named `T`. Inside `compare`, we can use the name `T` to refer to a type. Which *actual type T* represents is determined by the compiler based on how the function is used.

同样，模板形参表示可以在类或函数的定义中使用的类型或值。例如，`compare` 函数声明一个名为 `T` 的类型形参。在 `compare` 内部，可以使用名字 `T` 引用一个类型，`T` 表示哪个实际类型由编译器根据所用的函数而确定。

A template parameter can be a **type parameter**, which represents a type, or a **nontype parameter**, which represents a constant expression. A nontype parameter is declared following a type specifier. We'll see more about nontype parameters in [Section 16.1.5](#) (p. 632). A type parameter is defined following the keyword `class` or `typename`. For example, `class T` is a type parameter named `T`. There is no difference between `class` and `typename` in this context.

模板形参可以是表示类型的类型形参，也可以是表示常量表达式的非类型形参。非类型形参跟在类型说明符之后声明，[第 16.1.5 节](#)将进一步介绍非类型形参。类型形参跟在关键字 `class` 或 `typename` 之后定义，例如，`class T` 是名为 `T` 的类型形参，在这里 `class` 和 `typename` 没有区别。

Using a Function Template

使用函数模板

When we use a function template, the compiler infers what **template argument(s)** to bind to the template parameter(s). Once the compiler determines the actual template argument(s), it **instantiates** an instance of the function template for us. Essentially, the compiler figures out what type to use in place of each type parameter and what value to use in place of each nontype parameter. Having deduced the actual template arguments, it generates and compiles a version of the function using those arguments in place of the corresponding template parameters. The compiler takes on the tedium of (re)writing the function for each type we use.

使用函数模板时，编译器会推断哪个（或哪些）模板实参绑定到模板形参。一旦编译器确定了实际的模板实参，就称它实例化了函数模板的一个实例。实质上，编译器将确定用什么类型代替每个类型形参，以及用什么值代替每个非类型形参。推导出实际模板实参后，编译器使用实参代替相应的模板形参产生编译该版本的函数。编译器承担了为我们使用的每种类型而编写函数的单调工作。

Given the calls

对于以下调用

```
int main ()  
{  
    // T is int;  
    // compiler instantiates int compare(const int&, const int&)  
    cout << compare(1, 0) << endl;  
    // T is string;  
    // compiler instantiates int compare(const string&, const string&)  
    string s1 = "hi", s2 = "world";  
    cout << compare(s1, s2) << endl;  
    return 0;  
}
```

the compiler will instantiate two different versions of `compare`. The compiler will create one version that replaces `T` by `int` and a second version that uses `string` in place of `T`.

编译器将实例化 `compare` 的两个不同版本，编译器将用 `int` 代替 `T` 创建第一个版本，并用 `string` 代替 `T` 创建第二个版本。

Section 16.1. Template Definitions

inline Function Templates

inline 函数模板

A function template can be declared `inline` in the same way as a nontemplate function. The specifier is placed following the template parameter list and before the return type. It is not placed in front of the `template` keyword.

函数模板可以用与非模板函数一样的方式声明为 `inline`。说明符放在模板形参表之后、返回类型之前，不能放在关键字 `template` 之前。

```
// ok: inline specifier follows template parameter list
template <typename T> inline T min(const T&, const T&);
// error: incorrect placement of inline specifier
inline template <typename T> T min(const T&, const T&);
```

Exercises Section 16.1.1

Exercise 16.1: Write a template that returns the absolute value of its parameter. Call the template on values of at least three different types. Note: until we discuss how the compiler handles template instantiation in [Section 16.3](#) (p. 643), you should put each template definition and all uses of that template in the same file.

编写一个模板返回形参的绝对值。至少用三种不同类型的值调用模板。注意：在[第 16.3 节](#)讨论编译器怎样处理模板实例化之前，你应该将每个模板定义和该模板的所有使用放在同一文件中。

Exercise 16.2: Write a function template that takes a reference to an `ostream` and a value, and writes the value to the stream. Call the function on at least four different types. Test your program by writing to `cout`, to a file, and to a `stringstream`.

编写一个函数模板，接受一个 `ostream` 引用和一个值，将该值写入流。用至少四种不同类型调用函数。通过写至 `cout`、写至文件和写至 `stringstream` 来测试你的程序。

Exercise 16.3: When we called `compare` on two `strings`, we passed two `string` objects, which we initialized from string literals. What would happen if we wrote:

当调用两个 `string` 对象的 `compare` 时，传递用字符串字面值初始化的两个 `string` 对象。如果编写以下代码会发生什么？

```
compare ("hi", "world");
```

16.1.2. Defining a Class Template

16.1.2. 定义类模板

Just as we can define function templates, we can also define class templates.

就像可以定义函数模板一样，也可以定义类模板。



To illustrate class templates, we'll implement our own version of the standard library `queue` ([Section 9.7](#), p. 348) class. User programs ought to use the standard `queue` class, not the one we define here.

为了举例说明类模板，我们将为标准库 `queue` 类（[第 9.7 节](#)）实现一个自己的版本。用户程序应使用标准的 `queue` 类，而不是我们这里定义的这个 `Queue` 类。

Section 16.1. Template Definitions

Our `Queue` must be able to hold objects of different types, so we'll define it as a [class template](#). The operations our `Queue` will support are a subset of the interface of the standard `queue`:

我们自定义的 `Queue` 类必须能够支持不同类型的对象，所以将它定义为类模板。`Queue` 类将支持的操作是标准 `queue` 类接口的子集：

- `push` to add an item to the back of the queue
`push` 操作，在队尾增加一项
- `pop` to remove the item at the head of the queue
`pop` 操作，从队头删除一项
- `front` to return a reference to the element at the head of the queue
`front` 操作，返回队头元素的引用
- `empty` to indicate whether there are any elements in the queue
`empty` 操作，指出队列中是否有元素

We'll look at how we might implement our `Queue` in [Section 16.4](#) (p. 647), but we can start by defining its interface:

[第 16.4 节](#) 将介绍怎样实现 `Queue` 类，这里先定义它的接口：

```
template <class Type> class Queue {  
public:  
    Queue ();           // default constructor  
    Type &front ();    // return element from head of Queue  
    const Type &front () const;  
    void push (const Type &); // add element to back of Queue  
    void pop();         // remove element from head of Queue  
    bool empty() const; // true if no elements in the Queue  
private:  
    // ...  
};
```

A class template is a template, so it must begin with the keyword `template` followed by a template parameter list. Our `Queue` template takes a single template type parameter named `Type`.

类模板也是模板，因此必须以关键字 `template` 开头，后接模板形参表。`Queue` 模板接受一个名为 `Type` 的模板类型形参。

With the exception of the template parameter list, the definition of a class template looks like any other class. A class template may define data, function, and type members; it may use access labels to control access to those members; it defines constructors and destructors; and so on. In the definition of the class and its members, we can use the template parameters as stand-ins for types or values that will be supplied when the class is used.

除了模板形参表外，类模板的定义看起来与任意其他类同相似。类模板可以定义数据成员、函数成员和类型成员，也可以使用访问标号控制对成员的访问，还可以定义构造函数和析构函数等等。在类和类成员的定义中，可以使用模板形参作为类型或值的占位符，在使用类时再提供那些类型或值。

For example, our `Queue` template has one template type parameter. We can use that parameter anywhere a type name can be used. In this template definition, we use `Type` to name the return type from the overloaded `front` operations and as the parameter type for the `push` operation.

例如，`Queue` 模板有一个模板类型形参，可以在任何可以使用类型名字的地方使用该形参。在这个模板定义中，用 `Type` 指定重载 `front` 操作的返回类型以及作为 `push` 操作的形参类型。

Using a Class Template

使用类模板

In contrast to calling a function template, when we use a class template, we must explicitly specify arguments for the template parameters:

与调用函数模板形成对比，使用类模板时，必须为模板形参显式指定实参：

```
Queue<int> qi;           // Queue that holds ints  
Queue<vector<double>> qc; // Queue that holds vectors of doubles  
Queue<string> qs;        // Queue that holds strings
```

Section 16.1. Template Definitions

The compiler uses the arguments to instantiate a type-specific version of the class. Essentially, the compiler rewrites our `Queue` class replacing `Type` by the specified actual type provided by the user. In this case, the compiler will instantiate three classes: a version of `Queue` with `Type` replaced by `int`, a second `Queue` class that uses `vector<double>` in place of `Type`, and a third that replaces `Type` by `string`.

编译器使用实参来实例化这个类的特定类型版本。实质上，编译器用用户提供的实际特定类型代替 `Type`，重新编写 `Queue` 类。在这个例子中，编译器将实例化三个 `Queue` 类：第一个用 `int` 代替 `Type`，第二个用 `vector<double>` 代替 `Type`，第三个用 `string` 代替 `Type`。

Exercises Section 16.1.2

Exercise 16.4: What is a function template? What is a class template?
什么是函数模板？什么是类模板？

Exercise 16.5: Define a function template to return the larger of two values.
定义一个函数模板，返回两个值中较大的一个。

Exercise 16.6: Similar to our a simplified version of `queue`, write a class template named `List` that is a simplified version of the standard `list` class.
类似于我们的 `queue` 简化版本，编写一个名为 `List` 的类模板，作为标准 `list` 类的简化版本。

16.1.3. Template Parameters

16.1.3. 模板形参

As with a function parameter, the name chosen by the programmer for a template parameter has no intrinsic meaning. In our example, we named `compare`'s template type parameter `T`, but we could have named it anything:

像函数形参一样，程序员为模板形参选择的名字没有本质含义。在我们的例子中，将 `compare` 的模板类型形参命名为 `T`，但也可以将它命名为任意名字：

```
// equivalent template definition
template <class Glorp>
int compare(const Glorp &v1, const Glorp &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

This code defines the same `compare` template as before.

该代码定义的 `compare` 模板与前面一样。

The only meaning we can ascribe to a template parameter is to distinguish whether the parameter is a type parameter or a nontype parameter. If it is a type parameter, then we know that the parameter represents an as yet unknown type. If it is a nontype parameter, we know it is an as yet unknown value.

可以给模板形参赋予的唯一含义是区别形参是类型形参还是非类型形参。如果是类型形参，我们就知道该形参表示未知类型，如果是非类型形参，我们就知道它是一个未知值。

When we wish to use the type or value that a template parameter represents, we use the same name as the corresponding template parameter. For example, all references to `Glorp` in the `compare` function template will be resolved to the same type when the function is instantiated.

如果希望使用模板形参所表示的类型或值，可以使用与对应模板形参相同的名字。例如，`compare` 函数中所有的 `Glorp` 引用将在该函数被实例化时确定为同一类型。

Template Parameter Scope

模板形参作用域

The name of a template parameter can be used after it has been declared as a template parameter and until the end of the template declaration or definition.

Section 16.1. Template Definitions

模板形参的名字可以在声明为模板形参之后直到模板声明或定义的末尾处使用。

Template parameters follow normal name-hiding rules. A template parameter with the same name as an object, function, or type declared in global scope hides the global name:

模板形参遵循常规名字屏蔽规则。与全局作用域中声明的对象、函数或类型同名的模板形参会屏蔽全局名字：

```
typedef double T;
template <class T> T calc(const T &a, const T &b)
{
    // tmp has the type of the template parameter T
    // not that of the global typedef
    T tmp = a;
    // ...
    return tmp;
}
```

The global typedef that defines `T` as `double` is hidden by the type parameter named `T`. Thus, `tmp` is not a `double`. Instead, the type of `tmp` is whatever type gets bound to the template parameter `T`.

将 `T` 定义为 `double` 的全局类型型别名将被名为 `T` 的类型形参所屏蔽，因此，`tmp` 不是 `double` 型，相反，`tmp` 的类型是绑定到模板形参的任意类型。

Restrictions on the Use of a Template Parameter Name

使用模板形参名字的限制

A name used as a template parameter may not be reused within the template:

用作模板形参的名字不能在模板内部重用。

```
template <class T> T calc(const T &a, const T &b)
{
    typedef double T; // error: redeclares template parameter T
    T tmp = a;
    // ...
    return tmp;
}
```

This restriction also means that the name of a template parameter can be used only once within the same template parameter list:

这一限制还意味着模板形参的名字只能在同一模板形参表中使用一次：

```
// error: illegal reuse of template parameter name V
template <class V, class V> V calc(const V&, const V&) ;
```

Of course, just as we can reuse function parameter names, the name of a template parameter can be reused across different templates:

当然，正如可以重用函数形参名字一样，模板形参的名字也能在不同模板中重用：

```
// ok: reuses parameter type name across different templates
template <class T> T calc (const T&, const T&);
template <class T> int compare(const T&, const T&);
```

Template Declarations

模板声明

As with any other function or class, we can declare a template without defining it. A declaration must indicate that the function or class is a template:

像其他任意函数或类一样，对于模板可以只声明而不定义。声明必须指出函数或类是一个模板：

```
// declares compare but does not define it
template <class T> int compare(const T&, const T&);
```

The names of the template parameters need not be the same across declarations and the definition of the same template:

Section 16.1. Template Definitions

同一模板的声明和定义中，模板形参的名字不必相同。

```
// all three uses of calc refer to the same function template
// forward declarations of the template
template <class T> T calc(const T&, const T&) ;
template <class U> U calc(const U&, const U&) ;
// actual definition of the template
template <class Type>
Type calc(const Type& a, const Type& b) { /* ... */ }
```

Each template type parameter must be preceded either by the keyword `class` or `typename`; each nontype parameter must be preceded by a type name. It is an error to omit the keyword or a type specifier:

每个模板类型形参前面必须带上关键字 `class` 或 `typename`，每个非类型形参前面必须带上类型名字，省略关键字或类型说明符是错误的：

```
// error: must precede U by either typename or class
template <typename T, U> T calc (const T&, const U&) ;
```

Exercises Section 16.1.3

Exercise Explain each of the following function template definitions and identify whether any are illegal.
16.7: Correct each error that you find.

解释下面每个函数模板的定义并指出是否有非法的。改正所发现的错误。

```
(a) template <class T, U, typename V> void f1(T, U, V) ;
(b) template <class T> T f2(int &T) ;
(c) inline template <class T> T foo(T, unsigned int*) ;
(d) template <class T> f4 (T, T) ;
(e) typedef char Ctype ;
template <typename Ctype> Ctype f5(Ctype a) ;
```

Exercise Explain which, if any, of the following declarations are errors and why.
16.8: 如果有，解释下面哪些声明是错误的说明为什么。

```
(a) template <class Type> Type bar(Type, Type) ;
template <class Type> Type bar(Type, Type) ;
(b) template <class T1, class T2> void bar(T1, T2) ;
template <class C1, typename C2> void bar(C1, C2) ;
```

Exercise Write a template that acts like the library `find` algorithm. Your template should take a single type parameter that will name the type for a pair of iterators that should be parameters to the function. Use your function to find a given value in a `vector<int>` and in a `list<string>`.

编写行为类似于标准库中 `find` 算法的模板。你的模板应接受一个类型形参，该形参指定函数形参（一对迭代器）的类型。使用你的函数在 `vector<int>` 和 `list<string>` 中查找给定值。

16.1.4. Template Type Parameters

16.1.4. 模板类型形参

Type parameters consist of the keyword `class` or the keyword `typename` followed by an identifier. In a template parameter list, these keywords have the same meaning: They indicate that the name that follows represents a type.

类型形参由关键字 `class` 或 `typename` 后接说明符构成。在模板形参表中，这两个关键字具有相同的含义，都指出后面所接的名字表示一个类型。

A template type parameter can be used as a type specifier anywhere in the template, in exactly the same way as a built-in or class type specifier. In particular, it can be used to name the return type or a function parameter type, and for variable declarations or casts inside the function body:

模板类型形参可作为类型说明符在模板中的任何地方，与内置类型说明符或类类型说明符的使用方式完全相同。具体而言，它可用于指定返回类型或函数形参类型，以及在函

Section 16.1. Template Definitions

数体中用于变量声明或强制类型转换。

```
// ok: same type used for the return type and both parameters
template <class T> T calc (const T& a, const T& b)
{
    // ok: tmp will have same type as the parameters & return type
    T tmp = a;
    // ...
    return tmp;
}
```

Distinction Between `typename` and `class`

`typename` 与 `class` 的区别

In a function template parameter list, the keywords `typename` and `class` have the same meaning and can be used interchangeably. Both keywords can be used in the same template parameter list:

在函数模板形参表中，关键字 `typename` 和 `class` 具有相同含义，可以互换使用，两个关键字都可以在同一模板形参表中使用：

```
// ok: no distinction between typename and class in template parameter list
template <typename T, class U> calc (const T&, const U&);
```

It may seem more intuitive to use the keyword `typename` instead of the keyword `class` to designate a template type parameter; after all, we can use built-in (nonclass types) types as the actual type parameter. Moreover, `typename` more clearly indicates that the name that follows is a type name. However, the keyword `typename` was added to C++ as part of Standard C++, so older programs are more likely to use the keyword `class` exclusively.

使用关键字 `typename` 代替关键字 `class` 指定模板类型形参也许更为直观，毕竟，可以使用内置类型（非类类型）作为实际的类型形参，而且，`typename` 更清楚地指明后面的名字是一个类型名。但是，关键字 `typename` 是作为标准 C++ 的组成部分加入到 C++ 中的，因此旧的程序更有可能只用关键字 `class`。

Designating Types inside the Template Definition

在模板定义内部指定类型

In addition to defining data or function members, a class may define type members. For example, the library container classes define various types, such as `size_type`, that allow us to use the containers in a machine-independent way. When we want to use such types inside a function template, we must tell the compiler that the name we are using refers to a type. We must be explicit because the compiler (and a reader of our program) cannot tell by inspection when a name defined by a type parameter is a type or a value. As an example, consider the following function:

除了定义数据成员或函数成员之外，类还可以定义类型成员。例如，标准库的容器类定义了不同的类型，如 `size_type`，使我们能够以独立于机器的方式使用容器。如果要在函数模板内部使用这样的类型，必须告诉编译器我们正在使用的名字指的是一个类型。必须显式地这样做，因为编译器（以及程序的读者）不能通过检查得知，由类型形参定义的名字何时是一个类型何时是一个值。例如，考虑下面的函数：

```
template <class Parm, class U>
Parm fcn(Parm* array, U value)
{
    Parm::size_type * p; // If Parm::size_type is a type, then a declaration
                         // If Parm::size_type is an object, then multiplication
}
```

We know that `size_type` must be a member of the type bound to `Parm`, but we do not know whether `size_type` is the name of a type or a data member. By default, the compiler assumes that such names name data members, not types.

我们知道 `size_type` 必定是绑定到 `Parm` 的那个类型的成员，但我们不知道 `size_type` 是一个类型成员的名字还是一个数据成员的名字，默认情况下，编译器假定这样的名字指定数据成员，而不是类型。

If we want the compiler to treat `size_type` as a type, then we must explicitly tell the compiler to do so:

如果希望编译器将 `size_type` 当作类型，则必须显式告诉编译器这样做：

```
template <class Parm, class U>
Parm fcn(Parm* array, U value)
{
    typename Parm::size_type * p; // ok: declares p to be a pointer
}
```

We tell the compiler to treat a member as a type by prefixing uses of the member name with the keyword `typename`. By writing `typename Parm::size_type` we say that member `size_type` of the type bound to `Parm` is the name of a type. Of course, this declaration puts an obligation on the types used to instantiate `fcn`: Those types must have a member named `size_type` that is a type.

通过在成员名前加上关键字 `typename` 作为前缀，可以告诉编译器将成员当作类型。通过编写 `typename parm::size_type`，指出绑定到 `Parm` 的类型的 `size_type` 成员是类型的名字。当然，这一声明给用实例化 `fcn` 的类型增加了一个职责：那些类型必须具有名为 `size_type` 的成员，而且该成员是一个类型。



If there is any doubt as to whether `typename` is necessary to indicate that a name is a type, it is a good idea to specify it. There is no harm in specifying `typename` before a type, so if the `typename` was unnecessary, it won't matter.

如果拿不准是否需要以 `typename` 指明一个名字是一个类型，那么指定它是个好主意。在类型之前指定 `typename` 没有害处，因此，即使 `typename` 是不必要的，也没有关系。

Exercises Section 16.1.4

Exercise 16.10: What, if any, are the differences between a type parameter that is declared as a `typename` and one that is declared as a `class`?

声明为 `typename` 的类型形参与声明为 `class` 的类型形参有区别吗？区别在哪里？

Exercise 16.11: When must `typename` be used?

何时必须使用 `typename`？

Exercise 16.12: Write a function template that takes a pair of values that represent iterators of unknown type. Find the value that occurs most frequently in the sequence.

编写一个函数模板，接受表示未知类型迭代器的一对值，找出在序列中出现得最频繁的值。

Exercise 16.13: Write a function that takes a reference to a container and prints the elements in that container. Use the container's `size_type` and `size` members to control the loop that prints the elements.

编写一个函数，接受一个容器的引用并打印该容器的元素。使用容器的 `size_type` 和 `size` 成员控制打印元素的循环。

Exercise 16.14: Rewrite the function from the previous exercise to use iterators returned from `begin` and `end` to control the loop.

重新编写上题的函数，使用从 `begin` 和 `end` 返回的迭代器来控制循环。

16.1.5. Nontype Template Parameters

16.1.5. 非类型模板形参

A template parameter need not be a type. In this section we'll look at nontype parameters as used by function templates. We'll look at nontype parameters for class templates in [Section 16.4.2](#) (p. 655) after we've seen more about how class templates are implemented.

模板形参不必都是类型。本节将介绍函数模板使用的非类型形参。在介绍了类模板实现的更多内容之后，[第 16.4.2 节](#)将介绍类模板的非类型形参。

Nontype parameters are replaced by values when the function is called. The type of that value is specified in the template parameter list. For example, the following function template declares `array_init` as a function template with one type and one nontype template parameter. The function itself takes a single parameter, which is a reference to an array ([Section 7.2.4](#), p. 240):

在调用函数时非类型形参将用值代替，值的类型在模板形参数表中指定。例如，下面的函数模板声明了 `array_init` 是一个含有一个类型模板形参和一个非类型模板形参的函数模板。函数本身接受一个形参，该形参是数组的引用（[第 7.2.4 节](#)）：

Section 16.1. Template Definitions

```
// initialize elements of an array to zero
template <class T, size_t N> void array_init(T (&parm)[N])
{
    for (size_t i = 0; i != N; ++i) {
        parm[i] = 0;
    }
}
```

A template nontype parameter is a constant value inside the template definition. A nontype parameter can be used when constant expressions are required for example, as we do here to specify the size of an array.

模板非类型形参是模板定义内部的常量值，在需要常量表达式的时候，可使用非类型形参（例如，像这里所做的一样）指定数组的长度。

When `array_init` is called, the compiler figures out the value of the nontype parameter from the array argument:

当调用 `array_init` 时，编译器从数组实参计算非类型形参的值：

```
int x[42];
double y[10];
array_init(x); // instantiates array_init(int(&)[42])
array_init(y); // instantiates array_init(double(&)[10])
```

The compiler will instantiate a separate version of `array_init` for each kind of array used in a call to `array_init`. For the program above, the compiler instantiates two versions of `array_init`: The first instance has its parameter bound to `int[42]`, and in the other, that parameter is bound to `double[10]`.

编译器将为 `array_init` 调用中用到的每种数组实例化一个 `array_init` 版本。对于上面的程序，编译器将实例化 `array_init` 的两个版本：第一个实例的形参绑定到 `int[42]`，另一个实例中的形参绑定到 `double[10]`。

Type Equivalence and Nontype Parameters

类型等价性与非类型形参

Expressions that evaluate to the same value are considered equivalent template arguments for a template nontype parameter. The following calls to `array_init` both refer to the same instantiation, `array_init<int, 42>`:

对模板的非类型形参而言，求值结果相同的表达式将认为是等价的。下面的两个 `array_init` 调用引用的是相同的实例——`array_init<int, 42>`：

```
int x[42];
const int sz = 40;
int y[sz + 2];
array_init(x); // instantiates array_init(int(&)[42])
array_init(y); // equivalent instantiation
```

Exercises Section 16.1.5

Exercise Write a function template that can determine the size of an array.

16.15:

编写可以确定数组长度的函数模板。

Exercise Rewrite the `printValues` function from page 240 as a function template that could be used to print the contents of arrays of varying sizes.

将第 7.2.4 节的 `printValues` 函数重新编写为可用于打印不同长度数组内容的函数模板。

16.1.6. Writing Generic Programs

16.1.6. 编写泛型程序

When we write a template, the code may not be overtly type-specific, but template code always makes some assumptions about the types that will

Section 16.1. Template Definitions

be used. For example, although our `compare` function is technically valid for any type, in practice the instantiated version might be illegal.

编写模板时，代码不可能针对特定类型，但模板代码总是要对将使用的类型做一些假设。例如，虽然 `compare` 函数从技术上说任意类型都是有效的，但实际上，实例化的版本可能是非法的。

Whether the generated program is legal depends on the operations used in the function and the operations supported by the type or types used. Our `compare` function has three statements:

产生的程序是否合法，取决于函数中使用的操作以及所用类型支持的操作。`compare` 函数有三条语句：

```
if (v1 < v2) return -1; // < on two objects of type T
if (v2 < v1) return 1; // < on two objects of type T
return 0;           // return int; not dependent on T
```

The first two statements contain code that implicitly depends on the parameter type. The `if` tests use the `<` operator on the parameters. The type of those parameters isn't known until the compiler sees a call to `compare` and `T` is bound to an actual type. Which `<` operator is used depends entirely on the argument type.

前两条语句包含隐式依赖于形参类型的代码，`if` 测试对形参使用 `<` 操作符，直到编译器看见 `compare` 调用并且 `T` 绑定到一个实际类型时，才知道形参的类型，使用哪个 `<` 操作符完全取决于实参类型。

If we call `compare` on an object that does not support the `<` operator, then the call will be invalid:

如果用不支持 `<` 操作符的对象调用 `compare`，则该调用将是无效的：

```
Sales_item item1, item2;
// error: no < on Sales_item
cout << compare(item1, item2) << endl;
```

The program is in error. The `Sales_item` type does not define the `<` operator, so the program won't compile.

程序会出错。`Sales_item` 类型没有定义 `<` 操作符，所以该程序不能编译。



The operations performed inside a function template constrains the types that can be used to instantiate the function. It is up to the programmer to guarantee that the types used as the function arguments actually support any operations that are used, and that those operations behave correctly in the context in which the template uses them.

在函数模板内部完成的操作限制了可用于实例化该函数的类型。程序员的责任是，保证用作函数实参的类型实际上支持所用的任意操作，以及保证在模板使用哪些操作的环境中那些操作运行正常。

Writing Type-Independent Code

编写独立于类型的代码

The art of writing good generic code is beyond the scope of this language primer. However, there is one overall guideline that is worth noting.

编写良好泛型代码的技巧超出了本书的范围，但是，有个一般原则值得注意。



When writing template code, it is useful to keep the number of requirements placed on the argument types as small as possible.

编写模板代码时，对实参类型的要求尽可能少是很有益的。

Simple though it is, our `compare` function illustrates two important principles for writing generic code:

虽然简单，但它说明了编写泛型代码的两个重要原则：

Section 16.1. Template Definitions

- The parameters to the template are `const` references.

模板的形参是 `const` 引用。

- The tests in the body use only `<` comparisons.

函数体中的测试只用 `<` 比较。

By making the parameters `const` references, we allow types that do not allow copying. Most types including the built-in types and, except for the IO types, all the library types we've used do allow copying. However, there can be class types that do not allow copying. By making our parameters `const` references, we ensure that such types can be used with our `compare` function. Moreover, if `compare` is called with large objects, then this design will also make the function run faster.

通过将形参设为 `const` 引用，就可以允许使用不允许复制的类型。大多数类型（包括内置类型和我们已使用过的除 IO 类型之外的所有标准库的类型）都允许复制。但是，也有不允许复制的类类型。将形参设为 `const` 引用，保证这种类型可以用于 `compare` 函数，而且，如果有比较大的对象调用 `compare`，则这个设计还可以使函数运行得更快。

Some readers might think it would be more natural for the comparisons to be done using both the `<` and `>` operators:

一些读者可能认为使用 `<` 和 `>` 操作符两者进行比较会更加自然：

```
// expected comparison
if (v1 < v2) return -1;
if (v1 > v2) return 1;
return 0;
```

However, by writing the code as

但是，将代码编写为

```
// expected comparison
if (v1 < v2) return -1;
if (v2 < v1) return 1; // equivalent to v1 > v2
return 0;
```

we reduce the requirements on types that can be used with our `compare` function. Those types must support `<`, but they need not also support `>`.

可以减少对可用于 `compare` 函数的类型的要求，这些类型必须支持 `<`，但不必支持 `>`。

Exercises Section 16.1.6

Exercise 16.17: In the "Key Concept" box on page 95, we noted that as a matter of habit C++ programmers prefer using `!=` to using `<`. Explain the rationale for this habit.

在第 3.3.2 节的“关键概念”中，我们注意到，C++ 程序员习惯于使用 `!=` 而不用 `<`，解释这一习惯的基本原理。

Exercise 16.18: In this section we noted that we deliberately wrote the test in `compare` to avoid requiring a type to have both the `<` and `>` operators. On the other hand, we tend to assume that types will have both `==` and `!=`. Explain why this seeming discrepancy in treatment actually reflects good programming style.

本节中我们提到应该慎重地编写 `compare` 中的信息论以避免要求类型同时具有 `<` 和 `>` 操作符，另一方面，往往假定类型既有 `==` 又有 `!=`。解释为什么这一看似不一致的处理实际上反映了良好的编程风格。

Caution: Compile-Time Errors at Link-Time

警告：链接时的编译时错误

In general, when compiling a template, there are three stages during which the compiler might flag an error: The first is when we compile the template definition itself. The compiler generally can't find many errors at this stage. Syntax errors, such as forgetting a semicolon or misspelling a variable name, can be detected.

一般而言，编译模板时，编译器可能会在三个阶段中标识错误：第一阶段是编译模板定义本身时。在这个阶段中编译器一般不能发现许多错误，可以检测到

Section 16.1. Template Definitions

诸如漏掉分号或变量名拼写错误一类的语法错误。

The second error-detection time is when the compiler sees a use of the template. At this stage, there is still not much the compiler can check. For a call to a function template, many compilers check only that the number and types of the arguments are appropriate. The compiler can detect that there are too many or too few arguments. It can also detect whether two arguments that are supposed to have the same type do so. For a class template, the compiler can check that the right number of template arguments are provided but not much else.

第二个错误检测时间是在编译器见到模板的使用时。在这个阶段，编译器仍没有很多检查可做。对于函数模板的调用，许多编译器只检查实参的数目和类型是否恰当，编译器可以检测到实参太多或太少，也可以检测到假定类型相同的两个实参是否真地类型相同。对于类模板，编译器可以检测提供的模板实参的正确数目。

The third time when errors are generated is during instantiation. It is only then that type-related errors can be found. Depending on how the compiler manages instantiation, which we'll cover on page [643](#), these errors may be reported at link time.

产生错误的第三个时间是在实例化的时候，只有在这个时候可以发现类型相关的错误。根据编译器管理实例化的方式（将在第 16.3 节讨论），有可能在链接时报告这些错误。

It is important to realize that when we compile a template definition, we do not know much about how valid the program is. Similarly, we may obtain compiler errors even after we have successfully compiled each file that uses the template. It is not uncommon to detect errors only during instantiation, which may happen at link-time.

重要的是，要认识到编译模板定义的时候，对程序是否有效所知不多。类似地，甚至可能会在已经成功编译了使用模板的每个文件之后出现编译错误。只在实例化期间检测错误的情况很少，错误检测可能发生在链接时。

16.2. Instantiation

16.2. 实例化

A template is a blueprint; it is not itself a class or a function. The compiler uses the template to generate type-specific versions of the specified class or function. The process of generating a type-specific instance of a template is known as instantiation. The term reflects the notion that a new "instance" of the template type or function is created.

模板是一个蓝图，它本身不是类或函数。编译器用模板产生指定的类或函数的特定类型版本。产生模板的特定类型实例的过程称为实例化，这个术语反映了创建模板类型或模板函数的新“实例”的概念。

A template is instantiated when we use it. A class template is instantiated when we refer to the an actual template class type, and a function template is instantiated when we call it or use it to initialize or assign to a pointer to function.

模板在使用时将进行实例化，类模板在引用实际模板类类型时实例化，函数模板在调用它或用它对函数指针进行初始化或赋值时实例化。

Instantiating a Class

类的实例化

When we write

当编写

```
Queue<int> qi;
```

the compiler automatically creates a class named `Queue<int>`. In effect, the compiler creates the `Queue<int>` class by rewriting the `Queue` template, replacing every occurrence of the template parameter `Type` by the type `int`. The instantiated class is as if we had written:

编译器自动创建名为 `Queue` 的类。实际上，编译器通过重新编写 `Queue` 模板，用类型 `int` 代替模板形参的每次出现而创建 `Queue` 类。实例化的类就像已经编写的一样：

```
// simulated version of Queue instantiated for type int
template <class Type> class Queue<int> {
public:
    Queue();           // this bound to Queue<int>*
    int &front();      // return type bound to int
    const int &front() const; // return type bound to int
    void push(const int &); // parameter type bound to int
    void pop();         // type invariant code
    bool empty() const; // type invariant code
private:
    // ...
};
```

To create a `Queue` class for objects of type `string`, we'd write:

要为 `string` 类型的对象创建 `Queue` 类，可以编写

```
Queue<string> qs;
```

In this case, each occurrence of `Type` would be replaced by `string`.

在这个例子中，用 `string` 代替 `Type` 的每次出现。



Each instantiation of a class template constitutes an independent class type. The `Queue` instantiation for the type `int` has no relationship to nor any special access to the members of any other `Queue` type.

类模板的每次实例化都会产生一个独立的类类型。为 `int` 类型实例化的 `Queue` 与任意其他 `Queue` 类型没有关系，对其他 `Queue` 类型成员也没有特殊的访问权。

Class Template Arguments Are Required

类模板形参是必需的

When we want to use a class template, we must always specify the template arguments explicitly.

想要使用类模板，就必须显式指定模板实参：

```
Queue qs; // error: which template instantiation?
```

A class template does not define a type; only a specific instantiation defines a type. We define a specific instantiation by providing a template argument to match each template parameter. Template arguments are specified in a comma-separated list and bracketed by the (<) and (>) tokens:

类模板不定义类型，只有特定的实例才定义了类型。特定的实例化是通过提供模板实参与每个模板形参匹配而定义的。模板实参在用逗号分隔并用尖括号括住的列表中指定：

```
Queue<int> qi;           // ok: defines Queue that holds ints
Queue<string> qs;        // ok: defines Queue that holds strings
```

The type defined by a template class always includes the template argument(s). For example, `Queue` is not a type; `Queue<int>` or `Queue<string>` are.

用模板类定义的类型总是模板实参。例如，`Queue` 不是类型，而 `Queue<int>` 或 `Queue<string>` 是类型。

Function-Template Instantiation

函数模板实例化

When we use a function template, the compiler will usually infer the template arguments for us:

使用函数模板时，编译器通常会为我们推断模板实参：

```
int main()
{
    compare(1, 0);          // ok: binds template parameter to int
    compare(3.14, 2.7);     // ok: binds template parameter to double
    return 0;
}
```

This program instantiates two versions of `compare`: one where `T` is replaced by `int` and the other where it is replaced by `double`. The compiler essentially writes for us these two instances of `compare`:

这个程序实例化了 `compare` 的两个版本：一个用 `int` 代替 `T`，另一个用 `double` 代替 `T`，实质上是编译器为我们编写了 `compare` 的这两个实例：

```
int compare(const int &v1, const int &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
int compare(const double &v1, const double &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

16.2.1. Template Argument Deduction

16.2.1. 模板实参推断

To determine which functions to instantiate, the compiler looks at each argument. If the corresponding parameter was declared with a type that is a type parameter, then the compiler infers the type of the parameter from the type of the argument. In the case of `compare`, both arguments have the same template type: they were each declared using the type parameter `T`.

要确定应该实例化哪个函数，编译器会查看每个实参。如果相应形参声明为类型形参的类型，则编译器从实参的类型推断形参的类型。在 `compare` 的例子中，两个实参有同样的模板类型，都是用类型形参 `T` 声明的。

In the first call, `compare(1, 0)`, those arguments are type `int`; in the second, `compare(3.14, 2.7)`, they have type `double`. The process of determining the types and values of the template arguments from the type of the function arguments is called [template argument deduction](#).

第一个调用 `compare(1, 0)` 中，实参为 `int` 类型；第二个调用 `compare(3.14, 2.7)` 中，实参为 `double` 类型。从函数实参确定模板实参的类型和值的过程叫做模板实参推断。

Multiple Type Parameter Arguments Must Match Exactly

多个类型形参的实参必须完全匹配

A template type parameter may be used as the type of more than one function parameter. In such cases, template type deduction must generate the same template argument type for each corresponding function argument. If the deduced types do not match, then the call is an error:

模板类型形参可以用作一个以上函数形参的类型。在这种情况下，模板类型推断必须为每个对应的函数实参产生相同的模板实参类型。如果推断的类型不匹配，则调用将会出错：

```
template <typename T>
int compare(const T& v1, const T& v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
int main()
{
    short si;
    // error: cannot instantiate compare(short, int)
    // must be: compare(short, short) or
    // compare(int, int)
    compare(si, 1024);
    return 0;
}
```

This call is in error because the arguments to `compare` don't have the same type. The template argument deduced from the first argument is `short`; the one for the second is `int`. These types don't match, so template argument deduction fails.

这个调用是错误的，因为调用 `compare` 时的实参类型不相同，从第一个实参推断出的模板类型是 `short`，从第二个实参推断出 `int` 类型，两个类型不匹配，所以模板实参推断失败。

If the designer of `compare` wants to allow normal conversions on the arguments, then the function must be defined with two type parameters:

如果 `compare` 的设计者想要允许实参的常规转换，则函数必须用两个类型形参来定义：

```
// argument types can differ, but must be compatible
template <typename A, typename B>
int compare(const A& v1, const B& v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

Now the user may supply arguments of different types:

现在用户可以提供不同类型的实参了：

```
short si;
compare(si, 1024); // ok: instantiates compare(short, int)
```

However, a `<` operator must exist that can compare values of those types.

但是，比较那些类型的值的 `<` 操作符必须存在。

Limited Conversions on Type Parameter Arguments

类型形参的实参的受限转换

Consider the following calls to `compare`:

考虑下面的 `compare` 调用:

```
short s1, s2;
int i1, i2;
compare(i1, i2);           // ok: instantiate compare(int, int)
compare(s1, s2);           // ok: instantiate compare(short, short)
```

The first call generates an instance of `compare` with `T` bound to `int`. A new instance is created for the second call, binding `T` to `short`.

第一个调用产生将 `T` 绑定到 `int` 的实例，为第二个调用创建新实例，将 `T` 绑定到 `short`。

Had `compare(int, int)` been an ordinary nontemplate function, then the second call would match that function. The `short` arguments would be promoted ([Section 5.12.2](#), p. 180) to `int`. Because `compare` is a template, a new function is instantiated with the type parameter bound to `short`.

如果 `compare(int, int)` 是普通的非模板函数，则第二个调用会匹配那个函数，`short` 实参将提升（[第 5.12.2 节](#)）为 `int`。因为 `compare` 是一个模板，所以将实例化一个新函数，将类型形参绑定到 `short`。

In general, arguments are not converted to match an existing instantiation; instead, a new instance is generated. There are only two kinds of conversions that the compiler will perform rather than generating a new instantiation:

一般而论，不会转换实参以匹配已有的实例化，相反，会产生新的实例。除了产生新的实例化之外，编译器只会执行两种转换：

- `const` conversions: A function that takes a reference or pointer to a `const` can be called with a reference or pointer to a `nonconst` object, respectively, without generating a new instantiation. If the function takes a nonreference type, then `const` is ignored on either the parameter type or the argument. That is, the same instantiation will be used whether we pass a `const` or `nonconst` object to a function defined to take a nonreference type.
`const` 转换：接受 `const` 引用或 `const` 指针的函数可以分别用非 `const` 对象的引用或指针来调用，无须产生新的实例化。如果函数接受非引用类型，形参类型实参都忽略 `const`，即，无论传递 `const` 或非 `const` 对象给接受非引用类型的函数，都使用相同的实例化。
- array or function to pointer conversions: If the template parameter is not a reference type, then the normal pointer conversion will be applied to arguments of array or function type. An array argument will be treated as a pointer to its first element, and a function argument will be treated as a pointer to the function's type.
数组或函数到指针的转换：如果模板形参不是引用类型，则对数组或函数类型的实参应用常规指针转换。数组实参将当作指向其第一个元素的指针，函数实参当作指向函数类型的指针。

As examples, consider calls to the functions `fobj` and `fref`. The `fobj` function copies its parameters, whereas `fref`'s parameters are references:

例如，考虑对函数 `fobj` 和 `fref` 的调用。`fobj` 函数复制它的形参，而 `fref` 的形参是引用：

```
template <typename T> T fobj(T, T); // arguments are copied
template <typename T>
T fref(const T&, const T&);        // reference arguments
string s1("a value");
const string s2("another value");
fobj(s1, s2);          // ok: calls f(string, string), const is ignored
fref(s1, s2);          // ok: non const object s1 converted to const reference
int a[10], b[42];
fobj(a, b); // ok: calls f(int*, int*)
fref(a, b); // error: array types don't match; arguments aren't converted to pointers
```

In the first case, we pass a `string` and a `const string` as arguments. Even though these types do not match exactly, both calls are legal. In the call to `fobj`, the arguments are copied, so whether the original object is `const` doesn't matter. In the call to `fref`, the parameter type is a reference to `const`. Conversion to `const` for a reference parameter is one of the acceptable conversions, so this call is also okay.

第一种情况下，传递 `string` 对象和 `const string` 对象作为实参，即使这些类型不完全匹配，两个调用也都是合法的。在 `fobj` 的调用中，实参被复制，因此原来的对象是否为 `const` 无关紧要。在 `fref` 的调用中，形参类型是 `const` 引用，对引用形参而言，转换为 `const` 是可以接受的转换，所以这个调用也正确。

In the next case, we pass array arguments in which the arrays are different sizes. In the call to `fobj`, the fact that the arrays are different doesn't matter. Both arrays are converted to pointers. The template parameter type in `fobj` is `int*`. The call to `fref`, however, is illegal. When the parameter is a reference ([Section 7.2.4](#), p. 240), the arrays are not converted to pointers. The types of `a` and `b` don't match, so the call is in error.

在第二种情况中，将传递不同长度的数组实参。`fobj` 的调用中，数组不同无关紧要，两个数组都转换为指针，`fobj` 的模板形参类型是 `int*`。但是，`fref` 的调用是非法的，

Section 16.2. Instantiation

当形参为引用时 ([第 7.2.4 节](#))，数组不能转换为指针，`a` 和 `b` 的类型不匹配，所以调用将出错。

Normal Conversions Apply for Nontemplate Arguments

应用于非模板实参的常规转换



The restriction on type conversions applies only to those arguments whose types are template parameters.

类型转换的限制只适用于类型为模板形参的那些实参。

Normal conversions ([Section 7.1.2](#), p. 229) are allowed for parameters defined using ordinary types. The following function template `sum` has two parameters:

用普通类型定义的形参可以使用常规转换 ([第 7.1.2 节](#))，下面的函数模板 `sum` 有两个形参：

```
template <class Type> Type sum(const Type &op1, int op2)
{
    return op1 + op2;
}
```

The first parameter, `op1`, has a template parameter type. Its actual type cannot be known until the function is used. The type of the second parameter, `op2`, is known: It's `int`.

第一个形参 `op1` 具有模板形参类型，它的实际类型到函数使用时才知道。第二个形参 `op2` 的类型已知，为 `int`。

Because the type of `op2` is fixed, normal conversions can be applied to arguments passed to `op2` when `sum` is called:

因为 `op2` 的类型是固定的，在调用 `sum` 的时候，可以对传递给 `op2` 的实参应用常规转换：

```
double d = 3.14;
string s1("hiya"), s2(" world");
sum(1024, d); // ok: instantiates sum(int, int), converts d to int
sum(1.4, d); // ok: instantiates sum(double, int), converts d to int
sum(s1, s2); // error: s2 cannot be converted to int
```

In the first two calls, the type of the second argument `dd` is not the same as the type of the corresponding function parameter. However, these calls are okay: There is a conversion from `double` to `int`. Because the type of the second parameter does not depend on a template parameter, the compiler will implicitly convert `dd`. The first call causes the function `sum(int, int)` to be instantiated; `sum(double, int)` is instantiated by the second call.

在前两个调用中，第二个实参 `dd` 的类型与对应函数形参的类型不同，但是，这些调用是正确的：有从 `double` 到 `int` 的转换。因为第二个形参的类型独立于模板形参，编译器将隐式转换 `dd`。第一个调用导致实例化函数 `sum(int, int)`，第二个调用实例化 `sum(double, int)`。

The third call is an error. There is no conversion from `string` to `int`. Using a `string` argument to match an `int` parameter is, as usual, illegal.

第三个调用是错误的。没有从 `string` 到 `int` 的转换，使用 `string` 实参来匹配 `int` 形参与一般情况一样，是非法的。

Template Argument Deduction and Function Pointers

模板实参推断与函数指针

We can use a function template to initialize or assign to a function pointer ([Section 7.9](#), p. 276). When we do so, the compiler uses the type of the pointer to instantiate a version of the template with the appropriate template argument(s).

可以使用函数模板对函数指针进行初始化或赋值 ([第 7.9 节](#))，这样做的时候，编译器使用指针的类型实例化具有适当模板实参的模板版本。

As an example, assume we have a function pointer that points to a function returning an `int` that takes two parameters, each of which is a reference to a `const int`. We could use that pointer to point to an instantiation of `compare`:

例如，假定有一个函数指针指向返回 `int` 值的函数，该函数接受两个形参，都是 `const int` 引用，可以用该指针指向 `compare` 的实例化

```
template <typename T> int compare(const T&, const T&);
```

Section 16.2. Instantiation

```
// pf1 points to the instantiation int compare (const int&, const int&)
int (*pf1) (const int&, const int&) = compare;
```

The type of `pf1` is "pointer to function returning an `int` taking two parameters of type `const int&`." The type of the parameters in `pf1` determines the type of the template argument for `T`. The template argument for `T` is `int`. The pointer `pf1` refers to the instantiation with `T` bound to `int`.

`pf1` 的类型是一个指针，指向“接受两个 `const int&` 类型形参并返回 `int` 值的函数”，形参的类型决定了 `T` 的模板实参的类型，`T` 的模板实参为 `int` 型，指针 `pf1` 引用的是将 `T` 绑定到 `int` 的实例化。



When the address of a function-template instantiation is taken, the context must be such that it allows a unique type or value to be determined for each template parameter.

获取函数模板实例化的地址的时候，上下文必须是这样的：它允许为每个模板形参确定唯一的类型或值。

It is an error if the template arguments cannot be determined from the function pointer type. For example, assume we have two functions named `func`. Each function takes a pointer to function argument. The first version of `func` takes a pointer to a function that has two `const string` reference parameters and returns a `string`. The second version of `func` takes a pointer to a function taking two `const int` reference parameters and returning an `int`. We cannot use `compare` as an argument to `func`:

如果不能从函数指针类型确定模板实参，就会出错。例如，假定有两个名为 `func` 的函数，每个函数接受一个指向函数实参的指针。`func` 的第一个版本接受有两个 `const string` 引用形参并返回 `string` 对象的函数的指针，`func` 的第二个版本接受带两个 `const int` 引用形参并返回 `int` 值的函数的指针，不能使用 `compare` 作为传给 `func` 的实参：

```
// overloaded versions of func; each take a different function pointer type
void func(int(*) (const string&, const string&));
void func(int(*) (const int&, const int&));
func(compare); // error: which instantiation of compare?
```

The problem is that by looking at the type of `func`'s parameter, it is not possible to determine a unique type for the template argument. The call to `func` could instantiate either of the following functions:

问题在于，通过查看 `func` 的形参类型不可能确定模板实参的唯一类型，对 `func` 的调用可以实例化下列函数中的任意一个：

```
compare(const string&, const string&)
compare(const int&, const int&)
```

Because it is not possible to identify a unique instantiation for the argument to `func`, this call is a compile-time (or link-time) error.

因为不能为传给 `func` 的实参确定唯一的实例化，该调用会产生一个编译时（或链接时）错误。

Exercises Section 16.2.1

Exercise What is instantiation?

16.19: 什么是实例化？

Exercise What happens during template argument deduction?

16.20: 在模板实参推断期间发生什么？

Exercise Name two type conversions allowed on function arguments involved in template argument deduction.

指出对模板实参推断中涉及的函数实参允许的类型转换。

Exercise Given the following templates

16.22: 对于下面的模板

```
template <class Type>
Type calc (const Type* array, int size);
template <class Type>
Type fcn(Type p1, Type p2);
```

which ones of the following calls, if any, are errors? Why?

下面这些调用有错吗? 如果有, 哪些是错误的? 为什么?

```
double dobj; float fobj; char cobj;
int ai[5] = { 511, 16, 8, 63, 34 };

(a) calc(cobj, 'c');
(b) calc(dobj, fobj);
(c) fcn(ai, cobj);
```

16.2.2. Function-Template Explicit Arguments

16.2.2. 函数模板的显式实参

In some situations, it is not possible to deduce the types of the template arguments. This problem arises most often when a function return type must be a type that differs from any used in the parameter list. In such situations, it is necessary to override the template argument deduction mechanism and explicitly specify the types or values to be used for the template parameters.

在某些情况下, 不可能推断模板实参的类型。当函数的返回类型必须与形参表中所用的所有类型都不同, 最常出现这一问题。在这种情况下, 有必要覆盖模板实参推断机制, 并显式指定为模板形参所用的类型或值。

Specifying an Explicit Template Argument

指定显式模板实参

Consider the following problem. We wish to define a function template called `sum` that takes arguments of two different types. We'd like the return type to be large enough to contain the sum of two values of any two types passed in any order. How can we do that? How should we specify `sum`'s return type?

考虑下面的问题。我们希望定义名为 `sum`、接受两个不同类型实参的函数模板, 希望返回类型足够大, 可以包含按任意次序传递的任意两个类型的两个值的和, 怎样才能做到? 应如何指定 `sum` 的返回类型?

```
// T or U as the return type?
template <class T, class U> ??? sum(T, U);
```

In this case, the answer is that neither parameter works all the time. Using either parameter is bound to fail at some point:

在这个例子中, 答案是没有一个形参在任何时候都可行, 使用任一形参都一定会在某些时候失败:

```
// neither T nor U works as return type
sum(3, 4L); // second type is larger; want U sum(T, U)
sum(3L, 4); // first type is larger; want T sum(T, U)
```

One approach to solving this problem would be to force callers of `sum` to cast ([Section 5.12.4](#), p. 183) the smaller type to the type we wish to use as the result:

解决这一问题的一个办法, 可能是强制 `sum` 的调用者将较小的类型强制转换 ([第 5.12.4 节](#)) 为希望作为结果使用的类型:

```
// ok: now either T or U works as return type
int i; short s;
sum(static_cast<int>(s), i); // ok: instantiates int sum(int, int)
```

Using a Type Parameter for the Return Type

Section 16.2. Instantiation

在返回类型中使用类型形参

An alternative way to specify the return type is to introduce a third template parameter that must be explicitly specified by our caller:

指定返回类型的一种方式是引入第三个模板形参，它必须由调用者显式指定：

```
// T1 cannot be deduced: it doesn't appear in the function parameter list
template <class T1, class T2, class T3>
T1 sum(T2, T3);
```

This version adds a template parameter to specify the return type. There is only one catch: There is no argument whose type can be used to infer the type of `T1`. Instead, the caller must explicitly provide an argument for this parameter on each call to `sum`.

这个版本增加了一个模板形参以指定返回类型。只有一个问题：没有实参的类型可用于推断 `T1` 的类型，相反，调用者必须在每次调用 `sum` 时为该形参显式提供实参。

We supply an explicit template argument to a call in much the same way that we define an instance of a class template. Explicit template arguments are specified in a comma-separated list, bracketed by the less-than (`<`) and greater-than (`>`) tokens. The list of explicit template types appears after the function name and before the argument list:

为调用提供显式模板实参与定义类模板的实例很类似，在以逗号分隔、用尖括号括住的列表中指定显式模板实参。显式模板类型的列表出现在函数名之后、实参表之前：

```
// ok: T1 explicitly specified; T2 and T3 inferred from argument types
long val3 = sum<long>(i, lng); // ok: calls long sum(int, long)
```

This call explicitly specifies the type for `T1`. The compiler deduces the types for `T2` and `T3` from the arguments passed in the call.

这一调用显式指定 `T1` 的类型，编译器从调用中传递的实参推断 `T2` 和 `T3` 的类型。

Explicit template argument(s) are matched to corresponding template parameter(s) from left to right; the first template argument is matched to the first template parameter, the second argument to the second parameter, and so on. An explicit template argument may be omitted only for the trailing (rightmost) parameters, assuming these can be deduced from the function parameters. If our `sum` function had been written as

显式模板实参从左至右对应模板形参相匹配，第一个模板实参与第一个模板形参匹配，第二个实参与第二个形参匹配，以此类推。假如可以从函数形参推断，则结尾（最右边）形参的显式模板实参可以省略。如果这样编写 `sum` 函数：

```
// poor design: Users must explicitly specify all three template parameters
template <class T1, class T2, class T3>
T3 alternative_sum(T2, T1);
```

then we would always have to specify arguments for all three parameters:

则总是必须为所有三个形参指定实参：

```
// error: can't infer initial template parameters
long val3 = alternative_sum<long>(i, lng);
// ok: All three parameters explicitly specified
long val2 = alternative_sum<long, int, long>(i, lng);
```

Explicit Arguments and Pointers to Function Templates

显式实参与函数模板的指针

Another example where explicit template arguments would be useful is the ambiguous program from page [641](#). We could disambiguate that case by using explicit template argument:

可以使用显式模板实参的另一个例子是[第 16.2.1 节](#)中有二义性程序，通过使用显式模板实参能够消除二义性：

```
template <typename T> int compare(const T&, const T&);
// overloaded versions of func; each take a different function pointer type
void func(int(*) (const string&, const string&));
void func(int(*) (const int&, const int&));
func(compare<int>); // ok: explicitly specify which version of compare
```

As before, we want to pass an instantiation of `compare` in the call to the overloaded function named `func`. It is not possible to select which instantiation of `compare` to pass by looking at the parameter lists for the different versions of `func`. Two different instantiations of `compare` could satisfy the call. The explicit template argument indicates which instantiation of `compare` should be used and which `func` function is called.

像前面一样，需要在调用中传递 `compare` 实例给名为 `func` 的重载函数。只查看不同版本 `func` 的形参表来选择传递 `compare` 的哪个实例是不可能的，两个不同的实例都可能满足该调用。显式模板形参需要指出应使用哪个 `compare` 实例以及调用哪个 `func` 函数。

Exercises Section 16.2.2

Exercise
16.23:

The library `max` function takes a single type parameter. Could you call `max` passing it an `int` and a `double`? If so, how? If not, why not?

标准库函数 `max` 接受单个类型形参，可以传递 `int` 和 `double` 对象调用 `max` 吗？如果可以，怎样做？如果不呢，为什么？

Exercise
16.24:

In [Section 16.2.1](#) (p. 638) we saw that the arguments to the version of `compare` that has a single template type parameter must match exactly. If we wanted to call the function with compatible types, such as `int` and `short`, we could use an explicit template argument to specify either `int` or `short` as the parameter type. Write a program that uses the version of `compare` that has one template parameter. Call `compare` using an explicit template argument that will let you pass arguments of type `int` and `short`.

在[第 16.2.1 节](#)我们看到，对于具有单个模板类型形参的 `compare` 版本，传给它的实参必须完全匹配，如果想要用兼容类型如 `int` 和 `short` 调用该函数，可以使用显式模板实参指定 `int` 或 `short` 作为形参类型。编写程序使具有一个模板形参的 `compare` 版本，使用允许你传递 `int` 和 `short` 类型实参的显式模板实参调用 `compare`。

Exercise
16.25:

Use an explicit template argument to make it sensible to call `compare` passing two string literals.

使用显式模板实参，传递两个字符串字面值调用 `compare` 是切合实际的。

Exercise
16.26:

Given the following template definition for `sum`

对于下面的 `sum` 模板定义：

```
template <class T1, class T2, class T3> T1 sum(T2, T3);
```

explain each of the following calls. Indicate which, if any, are errors. For each error, explain what is wrong.

解释下面的每个调用，是否有错？如果有，指出哪些是错误的，对每个错误，解释错在哪里。

```
double dobj1, dobj2; float fobj1, fobj2; char cobj1, cobj2;
(a) sum(dobj1, dobj2);
(b) sum<double, double, double>(fobj1, fobj2);
(c) sum<int>(cobj1, cobj2);
(d) sum<double, ,double>(fobj2, dobj2);
```

16.3. Template Compilation Models

16.3. 模板编译模型

When the compiler sees a template definition, it does not generate code immediately. The compiler produces type-specific instances of the template only when it sees a use of the template, such as when a function template is called or an object of a class template is defined.

当编译器看到模板定义的时候，它不立即产生代码。只有在看到用到模板时，如调用了函数模板或调用了类模板的对象的时候，编译器才产生特定类型的模板实例。

Ordinarily, when we call a function, the compiler needs to see only a declaration for the function. Similarly, when we define an object of class type, the class definition must be available, but the definitions of the member functions need not be present. As a result, we put class definitions and function declarations in header files and definitions of ordinary and class-member functions in source files.

一般而言，当调用函数的时候，编译器只需要看到函数的声明。类似地，定义类类型的对象时，类定义必须可用，但成员函数的定义不是必须存在的。因此，应该将类定义和函数声明放在头文件中，而普通函数和类成员函数的定义放在源文件中。

Templates are different: To generate an instantiation, the compiler must have access to the source code that defines the template. When we call a function template or a member function of a class template, the compiler needs the function definition. It needs the code we normally put in the source files.

模板则不同：要进行实例化，编译器必须能够访问定义模板的源代码。当调用函数模板或类模板的成员函数的时候，编译器需要函数定义，需要那些通常放在源文件中的代码。

Standard C++ defines two models for compiling template code. In each of these models, we structure our programs in largely the same way: Class definitions and function declarations go in header files, and function and member definitions go in source files. The two models differ in how the definitions from the source files are made available to the compiler. As of this writing, all compilers support the first model, known as the "inclusion" model; only some compilers support the second, "separate compilation" model.

标准 C++ 为编译模板代码定义了两种模型。在两种模型中，构造程序的方式很大程度上是相同的：类定义和函数声明放在头文件中，而函数定义和成员定义放在源文件中。两种模型的不同在于，编译器怎样使用来自源文件的定义。如本书所述，所有编译器都支持第一种模型，称为“包含”模型，只有一些编译器支持第二种模型，“分别编译”模型。



To compile code that uses your own class and function templates, you must consult your compiler's user's guide to see how your compiler handles instantiation.

要编译使用自己的类模板和函数模板的代码，必须查阅编译器的用户指南，看看编译器怎样处理实例化。

Inclusion Compilation Model

包含编译模型

In the **inclusion compilation model**, the compiler must see the definition for any template that is used. Typically, we make the definitions available by adding a `#include` directive to the headers that declare function or class templates. That `#include` brings in the source file(s) that contain the associated definitions:

在**包含编译模型**中，编译器必须看到用到的所有模板的定义。一般而言，可以通过在声明函数模板或类模板的头文件中添加一条 `#include` 指示使定义可用，该 `#include` 引入了包含相关定义的源文件：

```
// header file utilities.h
#ifndef UTILITIES_H // header guard (Section 2.9.2, p. 69)
#define UTILITIES_H
template <class T> int compare(const T&, const T&);
// other declarations

#include "utilities.cc" // get the definitions for compare etc.
#endif

// implementation file utilities.cc
template <class T> int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
// other definitions
```

This strategy lets us maintain the separation of header files and implementation files but ensures that the compiler will see both files when compiling code that uses the templates.

这一策略使我们能够保持头文件和实现文件的分享，但是需要保证编译器在编译使用模板的代码时能看到两种文件。

Some, especially older, compilers that use the inclusion model may generate multiple instantiations. If two or more separately compiled source files use the same template, these compilers will generate an instantiation for the template in each file. Ordinarily, this approach implies that a given template will be instantiated more than once. At link time, or during a prelink phase, the compiler selects one instantiation, discarding the others. In such cases, compile-time performance can be significantly degraded if there are a lot of files that instantiate the same template. This compile-time degradation is unlikely to be a problem on modern computers for many applications. However, in the context of large systems, the compile-time hit may become important.

某些使用包含模型的编译器，特别是较老的编译器，可以产生多个实例。如果两个或多个单独编译的源文件使用同一模板，这些编译器将为每个文件中的模板产生一个实例。通常，这种方法意味着给定模板将实例化超过一次。在链接的时候，或者在预链接阶段，编译器会选择一个实例化而丢弃其他的。在这种情况下，如果有许多实例化同一模板的文件，编译时性能会显著降低。对许多应用程序而言，这种编译时性能降低不大可能在现代计算机上成为问题，但是，在大系统环境中，编译时选择问题可能变得非常重要。

Such compilers often support mechanisms that avoid the compile-time overhead implicit in multiple instantiations of the same template. The way compilers optimize compile-time performance varies from one compiler to the next. If compile time for programs using templates is too burdensome, consult your compiler's user's guide to see what support your compiler offers to avoid redundant instantiations.

这种编译器通常支持某些机制，避免同一模板的多个实例化中隐含的编译开销。编译器优化编译时性能的方法各不相同。如果使用模板的程序的编译时间难于承担，请查阅编译器的用户指南，看看你的编译器能提供什么支持以避免多余的实例化。

Separate Compilation Model

分别编译模型

In the **separate compilation model**, the compiler keeps track of the associated template definitions for us. However, we must tell the compiler to remember a given template definition. We use the **export keyword** to do so.

在**分别编译模型**中，编译器会为我们跟踪相关的模板定义。但是，我们必须让编译器知道要记住给定的模板定义，可以使用 **export 关键字** 来做这件事。

The **export** keyword indicates that a given definition might be needed to generate instantiations in other files. A template may be defined as exported only once in a program. The compiler figures out how to locate the template definition when it needs to generate these instantiations. The **export** keyword need not appear on the template declaration.

export 关键字能够指明给定的定义可能会需要在其他文件中产生实例化。在一个程序中，一个模板只能定义为导出一次。编译器在需要产生这些实例化时计算出怎样定位模板定义。**export** 关键字不必在模板声明中出现。

Ordinarily, we indicate that a function template is **exported** as part of its definition. We do so by including the keyword **export** before the **template** keyword:

一般我们在函数模板的定义中指明函数模板为导出的，这是通过在关键字 **template** 之前包含 **export** 关键字而实现的：

```
// the template definition goes in a separately-compiled source file
export template <typename Type>
Type sum(Type t1, Type t2) /* ... */
```

The declaration for this function template, should, as usual, be put in a header. The declaration must not specify **export**.

这个函数模板的声明像通常一样应放在头文件中，声明不必指定 **export**。

Using **export** on a class template is a bit more complicated. As usual, the class declaration must go in a header file. The class body in the header should not use the **export** keyword. If we used **export** in the header, then that header could be used by only one source file in the program.

对类模板使用 **export** 更复杂一些。通常，类声明必须放在头文件中，头文件中的类定义体不应该使用关键字 **export**，如果在头文件中使用了 **export**，则该头文件只能被程序中的一个源文件使用。

Instead, we **export** the class in the class implementation file:

相反，应该在类的实现文件中使用 **export**：

```
// class template header goes in shared header file
template <class Type> class Queue { ... };
// Queue.cc implementation file declares Queue as exported
export template <class Type> class Queue;
#include "Queue.h"
// Queue member definitions
```

The members of an exported class are automatically declared as exported. It is also possible to declare individual members of a class template as exported. In this case, the keyword `export` is not specified on the class template itself. It is specified only on the specific member definitions to be exported. The definition of exported member functions need not be visible when the member is used. The definitions of any nonexported member must be treated as in the inclusion model: The definition should be placed inside the header that defines the class template.

导出类的成员将自动声明为导出的。也可以将类模板的个别成员声明为导出的，在这种情况下，关键字 `export` 不在类模板本身指定，而是只在被导出的特定成员定义上指定。导出成员函数的定义不必在使用成员时可见。任意非导出成员的定义必须像在包含模型中一样对待：定义应放在定义类模板的头文件中。

Exercises Section 16.3

Exercise 16.27: Determine which compilation model your compiler uses. Write and call a function template to find the median value in a `vector` that holds objects of unknown type. (Note: The median is a value such that half the elements are larger than the median, and half are smaller.) Structure your program in the normal way: The function definition should go in one file, a declaration for it in a header, which the code that defines and uses the function template should include.

确定你的编译器使用的是哪种编译模型。编写并调用函数模板，在保存未知类型对象的 `vector` 中查找中间值。
(注：中间值是这样一个值，一半元素比它大，一半元素比它小。) 用常规方式构造你的程序：函数定义应放在一个文件中，它的声明放在一个头文件中，定义和使用函数模板的代码应包含该头文件。

Exercise 16.28: Where would you place the definitions for the member functions and `static` data members of your class templates if the compiler you use supports the separation compilation model? Explain why.

如果所用的编译器支持分别编译模型，将类模板的成员函数和 `static` 数据成员的定义放在哪里？为什么？

Exercise 16.29: Where would you put those template member definitions if your compiler uses the inclusion model? Explain why.

如果你的编译器使用包含模型，将那些模板成员定义放在哪里？为什么？

Caution: Name Lookup in Class Templates

警告：类模板中的名字查找

Compiling templates is a surprisingly difficult task. Fortunately, it is a task handled by compiler writers. Unfortunately, some of that complexity is pushed onto users of templates: Templates contain two kinds of names:

编译模板是异常困难的工作。幸好，它是由编译器作者处理的任务。不幸的是，某些复杂性被推到模板用户的身上：模板包含两种名字：

1. Those that do not depend on a template parameter

独立于模板形参的那些名字

2. Those that do depend on a template parameter

依赖于模板形参的那些名字

It is up to the template designer to ensure that all names that do not depend on a template parameter are defined in the same scope as the template itself.

设计者的责任是，保证所有不依赖于模板形参的名字在模板本身的作用域中定义。

It is up to users of a template to ensure that declarations for all functions, types, and operators associated with the types used to instantiate the template are visible. This responsibility means that the user must ensure that these declarations are visible when a member of a class template or a function template is instantiated.

模板用户的责任是，保证与用来实例化模板的类型相关的所有函数、类型和操作符的声明可见。这个责任意味着，在实例化类模板的成员或函数模板的时候，用户必须保证这些声明是可见的。

Both of these requirements are easily satisfied by well-structured programs that make appropriate use of headers. Authors of templates should provide a header that contains declarations for all the names used in the class template or in the definitions of its members. Before defining a template on a particular type or using a member of that template,

the user must ensure that the header for the template type and the header that defines the type used as the element type are included.

适当使用头文件的结构良好的程序都容易满足这两个要求。模板的作者应提供头文件，该头文件包含在类模板或在其成员定义中使用的所有名字的声明。在用特定类型定义模板或者使用该模板的成员之前，用户必须保证包含了模板类型的头文件，以及定义用作成员类型的类型的头文件。

Team LiB

◀ PREVIOUS NEXT ▶

16.4. Class Template Members

16.4. 类模板成员

So far we have seen only how to declare the interface members of our `Queue` class template. In this section, we'll look at how we might implement the class.

到目前为止，我们只介绍了怎样声明 `Queue` 类模板的接口成员，本节将介绍怎样实现该类。



The standard library implements `queue` as an adaptor ([Section 9.7](#), p. 348) on top of another container. To emphasize the programming points involved in using a lower-level data structure, we'll implement our `Queue` class as a linked list. In practice, using a library container in our implementation would probably be a better decision.

标准库将 `queue` 实现为其他容器之上的适配器 ([第 9.7 节](#))。为了强调在使用低级数据结构中涉及的编程要点，我们将 `Queue` 实现为链表。实际上，在我们的实现中使用标准库容器可能是一个更好的决定。

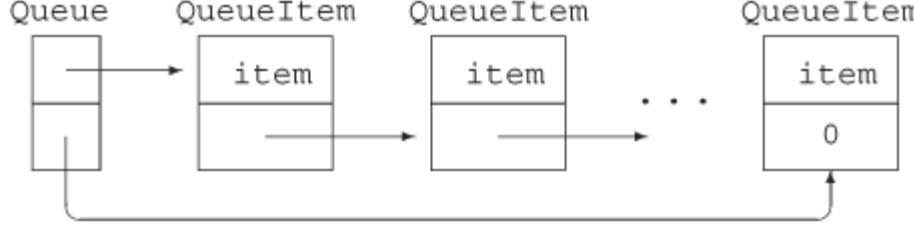
`Queue` Implementation Strategy

`Queue` 的实现策略

Our implementation, shown in [Figure 16.1](#) on the next page, uses two classes:

如图 [16.1](#) 所示，我们的实现使用两个类：

Figure 16.1. `Queue` Implementation



1. Class `QueueItem` will represent a node in `Queue`'s linked list. This class has two data members: `item` and `next`:

`QueueItem` 类表示 `Queue` 的链表中的节点，该类有两个数据成员 `item` 和 `next`：

- `item` holds the value of the element in the `Queue`; its type varies with each instance of `Queue`.
`item` 保存 `Queue` 中元素的值，它的类型随 `Queue` 的每个实例而变化。
- `next` is a pointer to the next `QueueItem` object in the queue.
`next` 是队列中指向下一 `QueueItem` 对象的指针。

Each element in the `Queue` is stored in a `QueueItem` object.

`Queue` 中的每个元素保存在一个 `QueueItem` 对象中。

2. Class `Queue` will provide the interface functions described in [Section 16.1.2](#) (p. 627). The `Queue` class will also have two data members: `head` and `tail`. These members are pointers to `QueueItem`.

`Queue` 类将提供[第 16.1.2 节](#)描述的接口函数，`Queue` 类也有两个数据成员： `head` 和 `tail`，这些成员是 `QueueItem` 指针。

Section 16.4. Class Template Members

As do the standard containers, our `Queue` class will copy the values it's given.

像标准容器一样，`Queue` 类将复制指定给它的值。

The `QueueItem` Class

`QueueItem` 类

We'll start our implementation by writing the `QueueItem` class:

首先编写 `QueueItem` 类：

```
template <class Type> class QueueItem {
    // private class: no public section
    QueueItem(const Type &t): item(t), next(0) { }
    Type item;           // value stored in this element
    QueueItem *next;     // pointer to next element in the Queue
};
```

As it stands, this class is already complete: It holds two data elements, which its constructor initializes. Like `Queue`, `QueueItem` is a class template. The class uses its template parameter to name the type of its `item` member. The value of each element in the `Queue` will be stored in `item`.

这个类似乎已经差不多完整了：它保存由其构造函数初始化的两个数据成员。像 `Queue` 类一样，`QueueItem` 是一个类模板，该类使用模板形参指定 `item` 成员的类型，`Queue` 中每个元素的值将保存在 `item` 中。

Each time we instantiate a `Queue` class, the same version of `QueueItem` will be instantiated as well. For example, if we create `Queue<int>`, then a companion class, `QueueItem<int>`, will be instantiated.

每当实例化一个 `Queue` 类的时候，也将实例化 `QueueItem` 的相同版本。例如，如果创建 `Queue<int>`，则将实例化一个伙伴类 `QueueItem<int>`。

Class `QueueItem` is a private class—it has no public interface. We intend this class to be used to implement `Queue` and have not built it for general use. Hence, it has no public members. We'll need to make class `Queue` a friend of `QueueItem` so that its members can access the members of `QueueItem`. We'll see how to do so in [Section 16.4.4](#) (p. 658).

`QueueItem` 类为私有类——它没有公用接口。我们这个类只是为实现 `Queue`，并不想用于一般目的，因此，它没有公用成员。需要将 `Queue` 类设为 `QueueItem` 类的友元，以便 `Queue` 类成员能够访问 `QueueItem` 的成员。[第 16.4.4 节](#)将介绍怎样做。



Inside the scope of a class template, we may refer to the class using its unqualified name.

在类模板的作用域内部，可以用它的非限定名字引用该类。

The `Queue` Class

`Queue` 类

We can now flesh out our `Queue` class:

现在充实 `Queue` 类：

```
template <class Type> class Queue {
public:
    // empty Queue
    Queue(): head(0), tail(0) { }
    // copy control to manage pointers to QueueItems in the Queue
    Queue(const Queue &Q): head(Q.head), tail(Q.tail)
        { copy_elems(Q); }
    Queue& operator=(const Queue&);
    ~Queue() { destroy(); }
    // return element from head of Queue
    // unchecked operation: front on an empty Queue is undefined
    Type& front() { return head->item; }
    const Type &front() const { return head->item; }
    void push(const Type &);      // add element to back of Queue
```

Section 16.4. Class Template Members

```
void pop (); // remove element from head of Queue
bool empty () const { // true if no elements in the Queue
    return head == 0;
}
private:
    QueueItem<Type> *head; // pointer to first element in Queue
    QueueItem<Type> *tail; // pointer to last element in Queue
    // utility functions used by copy constructor, assignment, and destructor
    void destroy(); // delete all the elements
    void copy_elems(const Queue&); // copy elements from parameter
};
```

In addition to the interface members, we have added the three copy-control members ([Chapter 13](#)) and associated utility functions used by those members. The `private` utility functions `destroy` and `copy_elems` will do the work of freeing the elements in the `Queue` and copying elements from another `Queue` into this one. The copy-control members are needed to manage the data members, `head` and `tail`, which are pointers to the first and last elements in the `Queue`. These elements are values of type `QueueItem<Type>`.

除了接口成员之外，还增加了三个复制控制成员（[第十三章](#)）以及那些成员所用的相关实用函数。`private` 实用函数 `destroy` 和 `copy_elems` 将完成释放 `Queue` 中的元素以及从另一 `Queue` 复制元素到这个 `Queue` 的任务。复制控制成员用于管理数据成员 `head` 和 `tail`，`head` 和 `tail` 是指向 `Queue` 中首尾元素的指针，这些成员是 `QueueItem<Type>` 类型的值。

The class implements several of its member functions:

`Queue` 类实现了几个成员函数：

- The default constructor sets both `head` and `tail` pointers to zero, indicating that the `Queue` is currently empty.
默认构造函数，将 `head` 和 `tail` 指针置 0，指明当前 `Queue` 为空。
- The copy constructor initializes `head` and `tail`, and calls `copy_elems` to copy the elements from its initializer.
复制构造函数，初始化 `head` 和 `tail`，并调用 `copy_elems` 从它的初始器复制元素。
- The `front` functions return the value at the head of the `Queue`. These functions do no checking: As with the analogous operations in the standard `queue`, users may not run `front` on an empty `Queue`.
几个 `front` 函数，返回头元素的值。这些函数不进行检查：像标准 `queue` 中的类似操作一样，用户不能在空 `Queue` 上运行 `front` 函数。
- The `empty` function returns the result of comparing `head` with zero. If `head` is zero, the `Queue` is empty; otherwise, it is not.
`empty` 函数，返回 `head` 与 0 的比较结果。如果 `head` 为 0，`Queue` 为空；否则，`Queue` 是非空的。

References to a Template Type in the Scope of the Template

模板作用域中模板类型的引用

For the most part, this class definition should be familiar. It differs little from other classes that we have defined. What is new is the use (or lack thereof) of the template type parameter in references to the `Queue` and `QueueItem` types.

这个类的主要部分应该是我们熟悉的。它只与我们已经定义过的类有少许区别。新的内容是 `Queue` 类型和 `QueueItem` 类型的引用中对模板类型形参的使用（或缺少）。

Ordinarily, when we use the name of a class template, we must specify the template parameters. There is one exception to this rule: Inside the scope of the class itself, we may use the unqualified name of the class template. For example, in the declarations of the default and copy constructor the name `Queue` is a shorthand notation that stands for `Queue<Type>`. Essentially the compiler infers that when we refer to the name of the class, we are referring to the same version. Hence, the copy constructor definition is really equivalent to writing:

通常，当使用类模板的名字的时候，必须指定模板形参。这一规则有个例外：在类本身的作用域内部，可以使用类模板的非限定名。例如，在默认构造函数和复制构造函数的声明中，名字 `Queue` 是 `Queue<Type>` 缩写表示。实质上，编译器推断，当我们引用类的名字时，引用的是同一版本。因此，复制构造函数定义其实等价于：

```
Queue<Type>(const Queue<Type> &Q): head(0), tail(0)
    { copy_elems(Q); }
```

The compiler performs no such inference for the template parameter(s) for other templates used within the class. Hence, we must specify the type parameter when declaring pointers to the companion `QueueItem` class:

编译器不会为类中使用的其他模板的模板形参进行这样的推断，因此，在声明伙伴类 `QueueItem` 的指针时，必须指定类型形参：

```
QueueItem<Type> *head; // pointer to first element in Queue
QueueItem<Type> *tail; // pointer to last element in Queue
```

These declarations say that for a given instantiation of class `Queue`, `head` and `tail` point to an object of type `QueueItem` instantiated for the same template parameter. That is, the type of `head` and `tail` inside the `Queue<int>` instantiation is `QueueItem<int>*`. It would be an error to omit the template parameter in the definition of the `head` and `tail` members:

这些声明指出，对于 `Queue` 类的给定实例化，`head` 和 `tail` 指向为同一模板形参实例化的 `QueueItem` 类型的对象，即，在 `Queue<int>` 实例化的内部，`head` 和 `tail` 的类型是 `QueueItem<int>*`。在 `head` 和 `tail` 成员的定义中省略模板形参将是错误的：

```
QueueItem *head;      // error: which version of QueueItem?
QueueItem *tail;      // error: which version of QueueItem?
```

Exercises Section 16.4

Exercise 16.30: Identify which, if any, of the following class template declarations (or declaration pairs) are illegal.

如果有，指出下面类模板声明（或声明对）中哪些是非法的。

- (a) template <class Type> class C1;
template <class Type, int size> class C1;
- (b) template <class T, U, class V> class C2;
- (c) template <class C1, typename C2> class C3 { };
- (d) template <typename myT, class myT> class C4 { };
- (e) template <class Type, int *ptr> class C5;
template <class T, int *pi> class C5;

Exercise 16.31: The following definition of `List` is incorrect. How would you fix it?

下面 `List` 的定义不正确，怎样改正？

```
template <class elemType> class ListItem;
template <class elemType> class List {
public:
    List<elemType>();
    List<elemType>(const List<elemType> &);
    List<elemType>& operator=(const List<elemType> &);
    ~List();
    void insert(ListItem *ptr, elemType value);
    ListItem *find(elemType value);
private:
    ListItem *front;
    ListItem *end;
};
```

16.4.1. Class-Template Member Functions

16.4.1. 类模板成员函数

The definition of a member function of a class template has the following form:

类模板成员函数的定义具有如下形式：

- It must start with the keyword `template` followed by the template parameter list for the class.

必须以关键字 `template` 开关，后接类的模板形参表。

- It must indicate the class of which it is a member.

必须指出它是哪个类的成员。

- The class name must include its template parameters.

Section 16.4. Class Template Members

类名必须包含其模板形参。

From these rules, we can see that a member function of class `Queue` defined outside the class will start as

从这些规则可以看到，在类外定义的 `Queue` 类的成员函数的开关应该是：

```
template <class T> ret-type Queue<T>::member-name
```

The `destroy` Function

`destroy` 函数

To illustrate a class template member function defined outside its class, let's look at the `destroy` function:

为了举例说明在类外定义的类模板成员函数，我们来看 `destroy` 函数：

```
template <class Type> void Queue<Type>::destroy()
{
    while (!empty())
        pop();
}
```

This definition can be read from left to right as:

这个定义可以从左至右读作：

- Defining a function template with a single type parameter named `Type`
用名为 `Type` 的类型形参定义一个函数模板；
- that returns `void`,
它返回 `void`；
- which is in the scope of the `Queue<Type>` class template.
它是在类模板 `Queue<Type>` 的作用域中。

The use of `Queue<Type>` preceding the scope operator (`::`) names the class to which the member function belongs.

在作用域操作符 (`::`) 之前使用的 `Queue<Type>` 指定成员函数所属的类。

Following the member-function name is the function definition. In the case of `destroy`, the function body looks very much like an ordinary nontemplate function definition. Its job is to walk the list of entries in this `Queue`, calling `pop` to remove each item.

跟在成员函数名之后的是函数定义。在 `destroy` 的例子中，函数体看来很普通的非模板函数定义，它的工作是遍历这个 `Queue` 的每个分支，调用 `pop` 除去每一项。

The `pop` Function

`pop` 函数

The `pop` member removes the value at the front of the `Queue`:

`pop` 成员的作用是除去 `Queue` 的队头值：

```
template <class Type> void Queue<Type>::pop()
{
    // pop is unchecked: Popping off an empty Queue is undefined
    QueueItem<Type>* p = head; // keep pointer to head so we can delete it
    head = head->next;        // head now points to next element
    delete p;                 // delete old head element
}
```

The `pop` function assumes that users do not call `pop` on an empty `Queue`. The job of `pop` is to remove the element at the start of the `Queue`. We must reset the `head` pointer to point to the next element in the `Queue`, and then delete the element that had been at the `head`. The only tricky part is remembering to keep a separate pointer to that element so we can delete it after resetting the `head` pointer.

Section 16.4. Class Template Members

`pop` 函数假设用户不会在空 `Queue` 上调用 `pop`。`pop` 的工作是除去 `Queue` 的头元素。必须重置 `head` 指针以指向 `Queue` 中的下一元素，然后删除 `head` 位置的元素。唯一有技巧的部分是记得保持指向该元素的一个单独指针，以便在重置 `head` 指针之后可以删除元素。

The `push` Function

`push` 函数

The `push` member places a new item at the back of the queue:

`push` 成员将新项放在队列末尾：

```
template <class Type> void Queue<Type>::push(const Type &val)
{
    // allocate a new QueueItem object
    QueueItem<Type> *pt = new QueueItem<Type>(val);
    // put item onto existing queue
    if (empty())
        head = tail = pt; // the queue now has only one element
    else {
        tail->next = pt; // add new element to end of the queue
        tail = pt;
    }
}
```

This function starts by allocating a new `QueueItem`, which is initialized from the value we were passed. There's actually a surprising bit of work going on in this statement:

这个函数首先分配新的 `QueueItem` 对象，用传递的值初始化它。这里实际上有些令人惊讶的工作，陈述如下：

1. The `QueueItem` constructor copies its argument into the `QueueItem`'s `item` member. As do the standard containers, our `Queue` class stores copies of the elements it is given.

`QueueItem` 构造函数将实参复制到 `QueueItem` 对象的 `item` 成员。像标准容器所做的一样，`Queue` 类存储所给元素的副本。

2. If `item` is a class type, the initialization of `item` uses the copy constructor of whatever type `item` has.

如果 `item` 为类类型，`item` 的初始化使用 `item` 所具有任意类型的复制构造函数。

3. The `QueueItem` constructor also initializes the `next` pointer to 0 to indicate that this element points to no other `QueueItem`.

`QueueItem` 构造函数还将 `next` 指针初始化为 0，以指出该元素没有指向其他 `QueueItem` 对象。

Because we're adding the element at the end of the `Queue`, setting `next` to 0 is exactly what we want.

因为将在 `Queue` 的末尾增加元素，将 `next` 置 0 正是我们所希望的。

Having created and initialized a new element, we must next hook it into the `Queue`. If the `Queue` is empty, then both `head` and `tail` should point to this new element. If there are already other elements in the `Queue`, then we make the current `tail` element point to this new element. The old `tail` is no longer the last element, which we indicate by making `tail` point to the newly constructed element as well.

创建和初始化新元素之后，必须将它链入 `Queue`。如果 `Queue` 为空，则 `head` 和 `tail` 都应该指向这个新元素。如果 `Queue` 中已经有元素了，则使当前 `tail` 元素指向这个新元素。旧的 `tail` 不再是最后一个元素了，这也是通过使 `tail` 指向新构造的元素指明的。

The `copy` Function

`copy_elems` 函数

Aside from the assignment operator, which we leave as an exercise, the only remaining function to write is `copy_elems`. This function is designed to be used by the assignment operator and copy constructor. Its job is to copy the elements from its parameter into this `Queue`:

我们将赋值操作符的实现留作习题，剩下要编写的函数只有 `copy_elems` 了。设计该函数的目的是供赋值操作符和复制构造函数使用，它的工作是从形参中复制元素到这个 `Queue`：

```
template <class Type>
void Queue<Type>::copy_elems(const Queue &orig)
{
    // copy elements from orig into this Queue
    // loop stops when pt == 0, which happens when we reach orig.tail
    for (QueueItem<Type> *pt = orig.head; pt; pt = pt->next)
        push(pt->item); // copy the element
}
```

Section 16.4. Class Template Members

We copy the elements in a `for` loop that starts by setting `pt` equal to the parameter's `head` pointer. The `for` continues until `pt` is 0, which happens after we get to the element that is the last one in `orig`. For each element in `orig`, we `push` a copy of value in that element onto this `Queue` and advance `pt` to point to the next element in `orig`.

在 `for` 循环中复制元素，`for` 循环始于将 `pt` 设为等于形参的 `head` 指针。循环进行直至获得 `orig` 中最后一个元素之后，`pt` 为 0。对于 `orig` 中的每个元素，将该元素值的副本 `push` 到这个 `Queue`，并推进 `pt` 以指向 `orig` 中的下一元素。

Instantiation of Class-Template Member Functions

类模板成员函数的实例化

Member functions of class templates are themselves function templates. Like any other function template, a member function of a class template is used to generate instantiations of that member. Unlike other function templates, the compiler does not perform template-argument deduction when instantiating class template member functions. Instead, the template parameters of a class template member function are determined by the type of the object on which the call is made. For example, when we call the `push` member of an object of type `Queue<int>`, the `push` function that is instantiated is

类模板的成员函数本身也是函数模板。像任何其他函数模板一样，需要使用类模板的成员函数产生该成员的实例化。与其他函数模板不同的是，在实例化类模板成员函数的进修，编译器不执行模板实参推断，相反，类模板成员函数的模板形参由调用该函数的对象的类型确定。例如，当调用 `Queue<int>` 类型对象的 `push` 成员时，实例化的 `push` 函数为

```
void Queue<int>::push(const int &val)
```

The fact that member-function template parameters are fixed by the template arguments of the object means that calling a class template member function is more flexible than comparable calls to function templates. Normal conversions are allowed on arguments to function parameters that were defined using the template parameter:

对象的模板实参能够确定成员函数模板形参，这一事实意味着，调用类模板成员函数比调用类似函数模板更灵活。用模板形参定义的函数形参的实参允许进行常规转换：

```
Queue<int> qi; // instantiates class Queue<int>
short s = 42;
int i = 42;
// ok: s converted to int and passed to push
qi.push(s); // instantiates Queue<int>::push(const int&)
qi.push(i); // uses Queue<int>::push(const int&)
f(s);        // instantiates f(const short&)
f(i);        // instantiates f(const int&)
```

When Classes and Members Are Instantiated

何时实例化类和成员

Member functions of a class template are instantiated only for functions that are used by the program. If a function is never used, then that member function is never instantiated. This behavior implies that types used to instantiate a template need to meet only the requirements of the operations that are actually used. As an example, recall the sequential container constructor ([Section 9.1.1](#), p. 309) that takes only a size parameter. That constructor uses the default constructor for the element type. If we have a type that does not define the default constructor, we may still define a container to hold this type. However, we may not use the constructor that takes only a size.

类模板的成员函数只有为程序所用才进行实例化。如果某函数从未使用，则不会实例化该成员函数。这一行为意味着，用于实例化模板的类型只需满足实际使用的操作的要求。[第 9.1.1 节](#)中只接受一个容量形参的顺序容器构造函数就是这样的例子，该构造函数使用元素类型的默认构造函数。如果有一个没有定义默认构造函数的类型，仍然可以定义容器来保存该类型，但是，不能使用只接受一个容量的构造函数。

When we define an object of a template type, that definition causes the class template to be instantiated. Defining an object also instantiates whichever constructor was used to initialize the object, along with any members called by that constructor:

定义模板类型的对象时，该定义导致实例化类模板。定义对象也会实例化用于初始化该对象的任一构造函数，以及该构造函数调用的任意成员：

```
// instantiates Queue<int> class and Queue<int>::Queue()
Queue<string> qs;
qs.push("hello"); // instantiates Queue<int>::push
```

The first statement instantiates the `Queue<string>` class and its default constructor. The next statement instantiates the `push` member function.

Section 16.4. Class Template Members

第一个语句实例化 Queue 类及其默认构造函数，第二个语句实例化 push 成员函数。

The instantiation of the `push` member:

`push` 成员的实例化：

```
template <class Type> void Queue<Type>::push(const Type &val)
{
    // allocate a new QueueItem object
    QueueItem<Type> *pt = new QueueItem<Type>(val);
    // put item onto existing queue
    if (empty())
        head = tail = pt;      // the queue now has only one element
    else {
        tail->next = pt;      // add new element to end of the queue
        tail = pt;
    }
}
```

in turn instantiates the companion `QueueItem<string>` class and its constructor.

将依次实例化伙伴类 `QueueItem<string>` 及其构造函数。

The `QueueItem` members in `queue` are pointers. Defining a pointer to a class template doesn't instantiate the class; the class is instantiated only when we use such a pointer. Thus, `QueueItem` is not instantiated when we create a `Queue` object. Instead, the `QueueItem` class is instantiated when a `Queue` member such as `front`, `push`, or `pop` is used.

`Queue` 类中的 `QueueItem` 成员是指针。类模板的指针定义不会对类进行实例化，只有用到这样的指针时才会对类进行实例化。因此，在创建 `Queue` 对象进不会实例化 `QueueItem` 类，相反，在使用诸如 `front`、`push` 或 `pop` 这样的 `Queue` 成员时才实例化 `QueueItem` 类。

Exercises Section 16.4.1

Exercise Implement the assignment operator for class `Queue`.

16.32: 为 `Queue` 类实现赋值操作符。

Exercise Explain how the `next` pointers in the newly created `Queue` get set during the `copy_elems`

16.33: 函数。

解释在 `copy_elems` 函数中新创建的 `Queue` 对象中的 `next` 指针怎样设置。

Exercise Write the member function definitions of the `List` class that you defined for the exercises in **16.34:** [Section 16.1.2](#) (p. 628).

编写第 16.1.2 节习题中定义的 `List` 类的成员函数定义。

Exercise Write a generic version of the `CheckedPtr` class described in [Section 14.7](#) (p. 526).

16.35: 编写第 14.7 节中描述的 `CheckedPtr` 类的泛型版本。

16.4.2. Template Arguments for Nontype Parameters

16.4.2. 非类型形参的模板实参

Now that we've seen more about how class templates are implemented, we can look at nontype parameters for class templates. We'll do so by defining a new version of the `Screen` class first introduced in [Chapter 12](#). In this case, we'll redefine `Screen` to be a template, parameterized by its height and width:

我们已经了解了如何实现类模板，现在来看看类模板的非类型形参。我们将为[第十二章](#)引入的 `Screen` 类定义一个新版本，借以介绍类模板的非类型形参。在这个例子中，将 `Screen` 类重新定义为模板，以高度和宽度为形参。

```
template <int hi, int wid>
class Screen {
public:
    // template nontype parameters used to initialize data members
    Screen(): screen(hi * wid, '#'), cursor(0),
```

Section 16.4. Class Template Members

```
height(hi), width(wid) { }  
// ...  
private:  
    std::string          screen;  
    std::string::size_type cursor;  
    std::string::size_type height, width;  
};
```

This template has two parameters, both of which are nontype parameters. When users define `Screen` objects, they must provide a constant expression to use for each of these parameters. The class uses these parameters in the default constructor to set the size of the default `Screen`.

这个模板有两个形参，均为非类型形参。当用户定义 `Screen` 对象时，必须为每个形参提供常量表达式以供使用。类在默认构造函数中使用这些形参设置默认 `Screen` 的尺寸。

As with any class template, the parameter values must be explicitly stated whenever we use the `Screen` type:

像任意类模板一样，使用 `Screen` 类型时必须显式声明形参值：

```
Screen<24,80> hp2621; // screen 24 lines by 80 characters
```

The object `hp2621` uses the template instantiation `Screen<24, 80>`. The template argument for `hi` is 24, and the argument for `wid` is 80. In both cases, the template argument is a constant expression.

对象 `hp2621` 使用模板实例化 `Screen<24, 80>`。`hi` 的模板实参是 24，而 `wid` 的模板实参是 80，两种情况下，模板实参都是常量表达式。



Nontype template arguments must be compile-time constant expressions.

非类型模板实参必须是编译时常量表达式。

Exercises Section 16.4.2

Exercise

16.36: Explain what instantiations, if any, are caused by each labeled statement.

每个带标号的语句，会导致实例化吗？如果会，解释为什么。

```
template <class T> class Stack { };  
void f1(Stack<char>);           // (a)  
class Exercise {  
    Stack<double> &rsd;           // (b)  
    Stack<int> si;              // (c)  
};  
int main() {  
    Stack<char> *sc;            // (d)  
    f1(*sc);                  // (e)  
    int iObj = sizeof(Stack< string >); // (f)  
}
```

Exercise

16.37: Identify which, if any, of the following template instantiations are valid. Explain why the instantiation isn't valid.

下面哪些模板实例化是有效的？解释为什么实例化无效。

```
template <class T, int size> class Array { /* . . . */ };  
template <int hi, int wid> class Screen { /* . . . */ };  
(a) const int hi = 40, wi = 80; Screen<hi, wi+32> sObj;  
(b) const int arr_size = 1024; Array<string, arr_size> a1;  
(c) unsigned int asize = 255; Array<int, asize> a2;  
(e) const double db = 3.1415; Array<double, db> a3;
```

16.4.3. Friend Declarations in Class Templates

16.4.3. 类模板中的友元声明

There are three kinds of friend declarations that may appear in a class template. Each kind of declaration declares friendship to one or more entities:

在类模板中可以出现三种友元声明，每一种都声明了与一个或多个实体友元关系：

1. A friend declaration for an ordinary nontemplate class or function, which grants friendship to the specific named class or function.

普通非模板类或函数的友元声明，将友元关系授予明确指定的类或函数。

2. A friend declaration for a class template or function template, which grants access to all instances of the friend.

类模板或函数模板的友元声明，授予对友元所有实例的访问权。

3. A friend declaration that grants access only to a specific instance of a class or function template.

只授予对类模板或函数模板的特定实例的访问权的友元声明。

Ordinary Friends

普通友元

A nontemplate class or function can be a friend to a class template:

非模板类或非模板函数可以是类模板的友元：

```
template <class Type> class Bar {
    // grants access to ordinary, nontemplate class and function
    friend class FooBar;
    friend void fcn();
    // ...
};
```

This declaration says that the members of `FooBar` and the function `fcn` may access the `private` and `protected` members of any instantiation of class `Bar`.

这个声明是说，`FooBar` 的成员和 `fcn` 函数可以访问 `Bar` 类的任意实例的 `private` 成员和 `protected` 成员。

General Template Friendship

一般模板友元关系

A friend can be a class or function template:

友元可以是类模板或函数模板：

```
template <class Type> class Bar {
    // grants access to Foo1 or templ_fcn1 parameterized by any type
    template <class T> friend class Foo1;
    template <class T> friend void templ_fcn1(const T&);
    // ...
};
```

These friend declarations use a different type parameter than does the class itself. That type parameter refers to the type parameter of `Foo1` and `templ_fcn1`. In both these cases, an unlimited number of classes and functions are made friends to `Bar`. The friend declaration for `Foo1` says that any instance of `Foo1` may access the private elements of any instance of `Bar`. Similarly, any instance of `templ_fcn1` may access any instance of `Bar`.

这些友元声明使用与类本身不同的类型形参，该类型形参指的是 `Foo1` 和 `templ_fcn1` 的类型形参。在这两种情况下，都将没有数目限制的类和函数设为 `Bar` 的友元。`Foo1` 的友元声明是说，`Foo1` 的任意实例都可以访问 `Bar` 的任意实例的私有元素，类似地，`temp_fcn1` 的任意实例可以访问 `Bar` 的任意实例。

This friend declaration establishes a one-to-many mapping between each instantiation of `Bar` and its friends, `Foo1` and `templ_fcn1`. For each instantiation of `Bar`, all instantiations of `Foo1` or `templ_fcn1` are friends.

这个友元声明在 `Bar` 与其友元 `Foo1` 和 `temp_fcn1` 的每个实例之间建立了一对多的映射。对 `Bar` 的每个实例而言，`Foo1` 或 `temp_fcn1` 的所有实例都是友元。

Specific Template Friendship

特定的模板友元关系

Rather than making all instances of a template a friend, a class can grant access to only a specific instance:

除了将一个模板的所有实例设为友元，类也可以只授予对特定实例的访问权：

```
template <class T> class Foo2;
template <class T> void templ_fcn2(const T&);
template <class Type> class Bar {
    // grants access to a single specific instance parameterized by char*
    friend class Foo2<char*>;
    friend void templ_fcn2<char*>(char* const &);
    // ...
};
```

Even though `Foo2` itself is a class template, friendship is extended only to the specific instance of `Foo2` that is parameterized by `char*`. Similarly, the friend declaration for `templ_fcn2` says that only the instance of that function parameterized by `char*` is a friend to class `Bar`. The specific instantiations of `Foo2` and `templ_fcn2` parameterized by `char*` can access every instantiation of `Bar`.

即使 `Foo2` 本身是类模板，友元关系也只扩展到 `Foo2` 的形参类型为 `char*` 的特定实例。类似地，`templ_fcn2` 的友元声明是说，只有形参类型为 `char*` 的函数实例是 `Bar` 类的友元。形参类型为 `char*` 的 `Foo2` 和 `templ_fcn2` 的特定实例可以访问 `Bar` 的每个实例。

More common are friend declarations of the following form:

下面形式的友元声明更为常见：

```
template <class T> class Foo3;
template <class T> void templ_fcn3(const T&);
template <class Type> class Bar {
    // each instantiation of Bar grants access to the
    // version of Foo3 or templ_fcn3 instantiated with the same type
    friend class Foo3<Type>;
    friend void templ_fcn3<Type>(const Type&);
    // ...
};
```

These friends define friendship between a particular instantiation of `Bar` and the instantiation of `Foo3` or `templ_fcn3` that uses the same template argument. Each instantiation of `Bar` has a single associated `Foo3` and `templ_fcn3` friend:

这些友元定义了 `Bar` 的特定实例与使用同一模板实参的 `Foo3` 或 `templ_fcn3` 的实例之间的友元关系。每个 `Bar` 实例有一个相关的 `Foo3` 和 `templ_fcn3` 友元：

```
Bar<int> bi; // Foo3<int> and templ_fcn3<int> are friends
Bar<string> bs; // Foo3<string>, templ_fcn3<string> are friends
```

Only those versions of `Foo3` or `templ_fcn3` that have the same template argument as a given instantiation of `Bar` are friends. Thus, `Foo3<int>` may access the private parts of `Bar<int>` but not of `Bar<string>` or any other instantiation of `Bar`.

只有与给定 `Bar` 实例有相同模板实参的那些 `Foo3` 或 `templ_fcn3` 版本是友元。因此，`Foo3<int>` 可以访问 `Bar<int>` 的私有部分，但不能访问 `Bar<string>` 或者任意其他 `Bar` 实例的私有部分。

Declaration Dependencies

声明依赖性

When we grant access to all instances of a given template, there need not be a declaration for that class or function template in scope. Essentially, the compiler treats the friend declaration as a declaration of the class or function as well.

当授予对给定模板的实例的访问权时候，在作用域中不需要存在该类模板或函数模板的声明。实质上，编译器将友元声明也当作类或函数的声明对待。

When we want to restrict friendship to a specific instantiation, then the class or function must have been declared before it can be used in a friend declaration:

想要限制对特定实例化的友元关系时，必须在可以用于友元声明之前声明类或函数：

```
template <class T> class A;
template <class T> class B {
```

Section 16.4. Class Template Members

```
public:  
    friend class A<T>;      // ok: A is known to be a template  
    friend class C;          // ok: C must be an ordinary, nontemplate class  
    template <class S> friend class D; // ok: D is a template  
    friend class E<T>;      // error: E wasn't declared as a template  
    friend class F<int>;    // error: F wasn't declared as a template  
};
```

If we have not previously told the compiler that the friend is a template, then the compiler will infer that the friend is an ordinary nontemplate class or function.

如果没有事先告诉编译器该友元是一个模板，则编译器将认为该友元是一个普通非模板类或非模板函数。

16.4.4. `Queue` and `QueueItem` Friend Declarations

16.4.4. `Queue` 和 `QueueItem` 的友元声明

Our `QueueItem` class is not intended to be used by the general program: All its members are private. In order for `Queue` to use `QueueItem`, `QueueItem` must make `Queue` a friend.

`QueueItem` 类不打算为一般程序所用：它的所有成员都是私有的。为了让 `Queue` 类使用 `QueueItem` 灰，`QueueItem` 类必须将 `Queue` 类设为友元。

Making a Class Template a Friend

将类模板设为友元

As we have just seen, when making a class template a friend, the class designer must decide how wide to make that friendship. In the case of `QueueItem`, we need to decide whether `QueueItem` should grant friendship to all `Queue` instances or only to a specific instance.

像我们已经看到的，将类模板设为友元的进修，类设计者必须决定友元关系应设置多广。在 `QueueItem` 类的例子中，需要决定 `QueueItem` 类应该将友元关系授予所有的 `Queue` 类实例，还是只授予特定实例。

Making every `Queue` a friend of each `QueueItem` is too broad. It makes no sense to allow a `Queue` instantiated with the type `string` to access members of a `QueueItem` instantiated with type `double`. The `Queue<string>` instantiation should be a friend only to the instantiation of the `QueueItem` for `strings`. That is, we want a one-to-one mapping between a `Queue` and `QueueItem` for each type of `Queue` that is instantiated:

将每个 `Queue` 类设为每个 `QueueItem` 类的友元太宽泛了，允许用 `string` 类型实例化的 `Queue` 类去访问用 `double` 类型实例化的 `QueueItem` 类的成员是没有意义的。`Queue<string>` 实例只应该是用 `string` 实例化的 `QueueItem` 类的友元，即，对于实例化的 `Queue` 类的每种类型，我们想要 `Queue` 类和 `QueueItem` 类之间的一对一映射：

```
// declaration that Queue is a template needed for friend declaration in QueueItem  
template <class Type> class Queue;  
template <class Type> class QueueItem {  
    friend class Queue<Type>;  
    // ...  
};
```

This declaration establishes the desired one-to-one mapping: only the `Queue` class that is instantiated with the same type as `QueueItem` is made a friend.

这个声明建立了想要一对一映射，只将与 `QueueItem` 类用同样类型实例化的 `Queue` 类设为友元。

The `Queue` Output Operator

`Queue` 输出操作符

One operation that might be useful to add to our `Queue` interface is the ability to print the contents of a `Queue` object. We'll do so by providing an overloaded instance of the output operator. This operator will walk the list of elements in the `Queue` and print the value in each element. We'll print the elements inside a pair of brackets.

`Queue` 类接口中可能增加的一个有用操作，是输出 `Queue` 对象的内容的能力。提供输出操作符的重载实例，可以做到这一点。这个操作符将遍历 `Queue` 中的元素链表并输出每个元素的值，将在一对尖括号内输出元素。

Section 16.4. Class Template Members

Because we want to be able to print the contents of `Queues` of any type, we need to make the output operator a template as well:

因为希望能够输出任意类型 `Queue` 的内容，所以需要将输出操作符也设为模板：

```
template <class Type>
ostream& operator<<(ostream &os, const Queue<Type> &q)
{
    os << "< ";
    QueueItem<Type> *p;
    for (p = q.head; p; p = p->next)
        os << p->item << " ";
    os <<">";
    return os;
}
```

If a `Queue` of type `int` contains the values 3, 5, 8, and 13, the output of this `Queue` displays as follows:

如果 `int` 类型的 `Queue` 包含值 3、5、8 和 13，这个 `Queue` 的输出显示如下：

```
<3 5 8 13 >
```

If the `Queue` is empty, the `for` loop body is never executed. The effect will be to print an empty pair of brackets if the `Queue` is empty.

如果 `Queue` 为空，`for` 循环不执行。结果是输出一对空的尖括号。

Making a Function Template a Friend

将函数模板设为友元

The output operator needs to be a friend of both the `Queue` and `QueueItem` classes. It uses the `head` member of class `Queue` and the `next` and `item` members of class `QueueItem`. Our classes grant friendship to the specific instance of the output operator instantiated with the same type:

输出操作符需要成为 `Queue` 类和 `QueueItem` 类的友元。它使用 `Queue` 类的 `head` 成员和 `QueueItem` 类的 `next` 和 `item` 成员。我们的类将友元关系授予用同样类型实例化的输出操作符的特定实例：

```
// function template declaration must precede friend declaration in QueueItem
template <class T>
std::ostream& operator<<(std::ostream&, const Queue<T>&);

template <class Type> class QueueItem {
    friend class Queue<Type>;
    // needs access to item and next
    friend std::ostream&
    operator<< <Type> (std::ostream&, const Queue<Type>&);
    // ...
};

template <class Type> class Queue {
    // needs access to head
    friend std::ostream&
    operator<< <Type> (std::ostream&, const Queue<Type>&);
};
```

Each friend declaration grants access to the corresponding instantiation of the `operator<<`. That is, the output operator that prints a `Queue<int>` is a friend to class `Queue<int>` (and `QueueItem<int>`). It is not a friend to any other `Queue` type.

每个友元声明授予对对应 `operator<<` 实例的访问权，即输出 `Queue<int>` 的输出操作符是 `Queue<int>` 类（以及 `QueueItem<int>`）类的友元，它不是任意其他 `Queue` 类型的友元。

Type Dependencies and the Output Operator

类型依赖性与输出操作符

The `Queue` output `operator<<` relies on the `operator<<` of `item` to actually print each element:

`Queue` 类的输出 `operator<<` 依赖于 `item` 对象的 `operator<<` 实际输出每个元素：

```
os << p->item << " ";
```

When we use `p->item` as an operand of the `<<` operator, we are using the `<<` defined for whatever type `item` has.

Section 16.4. Class Template Members

当使用 `p->item` 作为 `<<` 操作符的操作数的时候，使用的是为 `item` 所属的任意类型而定义的 `<<`。

This code is an example of a type dependency between `Queue` and the element type that `Queue` holds. In effect, each type bound to `Queue` that uses the `Queue` output operator must itself have an output operator. There is no language mechanism to specify or enforce that dependency in the definition of `Queue` itself. It is legal to create a `Queue` for a class that does not define the output operator but it is a compile-time (or link-time) error to print a `Queue` holding such a type.

此代码是 `Queue` 和 `Queue` 保存的元素之间的类型依赖性的例子。实际上，绑定到 `Queue` 且使用 `Queue` 输出操作符的每种类型本身必须有输出操作符。没有语言机制指定或强制 `Queue` 自身定义中的依赖性。为没有定义输出操作符的类创建 `Queue` 对象是合法的，但输出保存这种类型的 `Queue` 对象会发生编译时（或链接时）错误。

Exercises Section 16.4.4

Exercise 16.38: Write a `Screen` class template that uses non-type parameters to define the height and width of the `Screen`.

编写 `Screen` 类模板，使用非类型形参定义 `Screen` 的高度和宽度。

Exercise 16.39: Implement input and output operators for the template `Screen` class.

为 `Screen` 模板类实现输入和输出操作符。

Exercise 16.40: Which, if any, friends are necessary in class `Screen` to make the input and output operators work? Explain why each friend declaration, if any, was needed.

要使输入和输出操作符能够工作，`Screen` 类需要友元吗？如果需要，要哪些友元解释为什么需要每个友元声明。

Exercise 16.41: The friend declaration for `operator<<` in class `Queue` was

`Queue` 类中的 `operator<<` 的友元声明是：

```
friend std::ostream&
operator<< <Type> (std::ostream&, const Queue<Type>&);
```

What would be the effect of writing the `Queue` parameter as `const Queue&` rather than `const Queue<Type>&`?

将 `Queue` 形参写为 `const Queue&` 而不是 `const Queue<Type>&`，会有什么结果？

Exercise 16.42: Write an input operator that reads an `istream` and puts the values it reads into a `Queue`.

编写一个输入操作符，读一个 `istream` 对象并将读到的值放入一个 `Queue` 对象中。

16.4.5. Member Templates

16.4.5. 成员模板

Any class (template or otherwise) may have a member that is itself a class or function template. Such members are referred to as [member templates](#). Member templates may not be virtual.

任意类（模板或非模板）可以拥有本身为类模板或函数模板的成员，这种成员称为[成员模板](#)，成员模板不能为虚。

One example of a member template is the `assign` ([Section 9.3.8](#), p. 328) member of the standard containers. The version `assign` that takes two iterators uses a template parameter to represent the type of its iterator parameters. Another member template example is the container constructor that takes two iterators ([Section 9.1.1](#), p. 307). This constructor and the `assign` member allow containers to be built from sequences of different but compatible element types and/or different container types. Having implemented our own `Queue` class, we now can understand the design of these standard container members a bit better.

成员模板的一个例子是标准容器的 `assign` 成员（[第 9.3.8 节](#)），接受两个迭代器的 `assign` 版本使用模板形参表示其迭代器形参的类型。另一个成员模板例子是接受两个迭代器的容器构造函数（[第 9.1.1 节](#)）。该构造函数和 `assign` 成员使我们能够从不同但兼容的元素类型序列和／或不同容器类型建立容器。实现了自己的 `Queue` 类之后，我们现在能够更好地理解这些标准容器成员的设计了。

Consider the `Queue` copy constructor: It takes a single parameter that is a reference to a `Queue<Type>`. If we wanted to create a `Queue` by copying

Section 16.4. Class Template Members

elements from a `vector`, we could not do so; there is no conversion from `vector` to `Queue`. Similarly, if we wanted to copy elements from a `Queue<short>` into a `Queue<int>`, we could not do so. Again, even though we can convert a `short` to an `int`, there is no conversion from `Queue<short>` to `Queue<int>`. The same logic applies to the `Queue` assignment operator, which also takes a parameter of type `Queue<Type>&`.

考虑 `Queue` 类的复制构造函数：它接受一个形参，是 `Queue<Type>` 的引用。想要通过从 `vector` 对象中复制元素而创建 `Queue` 对象，是办不到的，因为没有从 `vector` 到 `Queue` 的转换。类似地，想要从 `Queue<short>` 复制元素到 `Queue<int>`，也办不到。同样的逻辑应用于赋值操作符，它也接受一个 `Queue<Type>&` 类型的形参。

The problem is that the copy constructor and assignment operator fix both the container and element type. We'd like to define a constructor and an `assign` member that allow both the container and element type to vary. When we need a parameter type to vary, we need to define a function template. In this case, we'll define the constructor and `assign` member to take a pair of iterators that denote a range in some other sequence. These functions will have a single template type parameter that represents an iterator type.

问题在于，复制构造函数和赋值操作符固定了容器和元素的类型。我们希望定义一个构造函数和一个 `assign` 成员，使容器类型和元素类型都能变化。需要形参类型变化的时候，就需要定义函数模板。在这个例子中，我们将定义构造函数和 `assign` 成员接受一对在其他序列指明范围的迭代器，这些函数将有一个表示迭代器类型的模板类型形参。

 The standard `queue` class does not define these members: `queue` doesn't support building or assigning a `queue` from another container. We define these members here for illustration purposes only.

标准 `queue` 类没有定义这些成员：不支持从其他容器建立 `queue` 对象或给 `queue` 对象赋值。我们在这里定义这些成员只是为了举例说明。

Defining a Member Template

定义成员模板

A template member declaration looks like the declaration of any template:

模板成员声明看起来像任意模板的声明一样：

```
template <class Type> class Queue {
public:
    // construct a Queue from a pair of iterators on some sequence
    template <class It>
    Queue(It beg, It end) {
        head(0), tail(0) { copy_elems(beg, end); }
    }
    // replace current Queue by contents delimited by a pair of iterators
    template <class Iter> void assign(Iter, Iter);
    // rest of Queue class as before
private:
    // version of copy to be used by assign to copy elements from iterator range
    template <class Iter> void copy_elems(Iter, Iter);
};
```

The member declaration starts with its own template parameter list. The constructor and `assign` member each have a single template type parameter. These functions use that type parameter as the type for their function parameters, which are iterators denoting a range of elements to copy.

成员声明的开关是自己的模板形参数。构造函数和 `assign` 成员各有一个模板类型形参，这些函数使用该类型形参作为其函数形参的类型，它们的函数形参是指明要复制元素范围的迭代器。

Defining a Member Template Outside the Class

在类外部定义成员模板

Like nontemplate members, a member template can be defined inside or outside of its enclosing class or class template definition. We have defined the constructor inside the class body. Its job is to copy the elements from the iterator range formed by its iterator arguments. It does so by calling the iterator version of `copy_elems` to do the actual copy.

像非模板成员一样，成员模板可以定义在包含它的类或类模板定义的内部或外部。我们已经在类定义体内部定义了构造函数，它的工作是从迭代器实参形成的迭代器范围复制元素，实际复制工作是通过调用 `copy_elems` 的迭代器版本完成的。

When we define a member template outside the scope of a class template, we must include both template parameter lists:

Section 16.4. Class Template Members

```
template <class T> template <class Iter>
void Queue<T>::assign(Iter beg, Iter end)
{
    destroy();           // remove existing elements in this Queue
    copy_elems(beg, end); // copy elements from the input range
}
```

When a member template is a member of a class template, then its definition must include the class-template parameters as well as its own template parameters. The class-template parameter list comes first, followed by the member's own template parameter list. The definition of `assign` starts with

当成员模板是类模板的成员时，它的定义必须包含类模板形参以及自己的模板形参。首先是类模板形参表，后面接着成员自己的模板形参表。`assign` 函数定义的开头为

```
template <class T> template <class Iter>
```

The first template parameter list `template<class T>` is that of the class template. The second template parameter list `template<class Iter>` is that of the member template.

第一个模板形参表 `template<class T>` 是类模板的，第二个模板形参表 `template<class Iter>` 是成员模板的。

The actions of our `assign` function are quite simple: It first calls `destroy`, which, as we've seen, frees the existing members of this `Queue`. The `assign` member then calls a new utility function named `copy_elems` to do the work of copying elements from the input range. That function is also a member template:

`assign` 函数的行为非常简单：它首先调用 `destroy` 函数，`destroy` 函数释放这个 `Queue` 的现在成员，然后 `assign` 成员调用名为 `copy_elems` 的新实用函数，完成从输入范围复制元素的工作。`copy_elems` 函数也是一个成员模板：

```
template <class Type> template <class It>
void Queue<Type>::copy_elems(It beg, It end)
{
    while (beg != end) {
        push(*beg);
        ++beg;
    }
}
```

The iterator version of `copy_elems` walks through an input range denoted by a pair of iterators. It calls `push` on each element in that range, which actually adds the element to the `Queue`.

`copy_elems` 的迭代器版本遍历由一对迭代器指定的输入范围，它对范围内的每个元素调用 `push` 函数，实际上由 `push` 函数将元素加入 `Queue`。



Because `assign` erases elements in the existing container, it is essential that the iterators passed to `assign` refer to elements in a different container. The standard container `assign` members and iterator constructors have the same restrictions.

因为 `assign` 函数删除现在容器中的成员，所以传给 `assign` 函数的迭代器有必要引用不同容器中的元素。标准容器的 `assign` 成员和迭代器构造函数有相同的限制。

Member Templates Obey Normal Access Control

成员模板遵循常规访问控制

A member template follows the same access rules as any other class members. If the member template is private, then only member functions and friends of the class can use that member template. Because the function member template `assign` is a public member, it can be used by the entire program; `copy_elems` is private, so it can be accessed only by the friends and members of `Queue`.

成员模板遵循与任意其他类成员一样的访问规则。如果成员模板为私有的，则只有该类的成员函数和友元可以使用该成员模板。因为函数成员模板 `assign` 是公有的，所以整个程序都可以使用它：`copy_elems` 是私有的，所以只有 `Queue` 的友元和成员可以访问它。

Member Templates and Instantiation

成员模板和实例化

Like any other member, a member template is instantiated only when it is used in a program. The instantiation of member templates of class templates is a bit more complicated than the instantiation of plain member functions of class templates. Member templates have two kinds of template parameters: Those that are defined by the class and those defined by the member template itself. The class template parameters are fixed by the type of the object through which the function is called. The template parameters defined by the member act like parameters of ordinary function templates. These parameters are resolved through normal template argument deduction ([Section 16.2.1](#), p. 637).

与其他成员一样，成员模板只有在程序中使用时才实例化。类模板的成员模板的实例化比类模板的普通成员函数的实例化要复杂一点。成员模板有两种模板形参：由类定义的和由成员模板本身定义的。类模板形参由调用函数的对象的类型确定，成员定义的模板形参的行为与普通函数模板一样。这些形参都通过常规模板实参推断（[第 16.2.1 节](#)）而确定。

To understand how instantiation works, let's look at uses of these members to copy and assign elements from an array of `shorts` or a `vector<int>`:

要理解实例化的原理，我们来看看使用这些成员从 `short` 数组或 `vector<int>`：复制和赋值元素：

```
short a[4] = { 0, 3, 6, 9 };
// instantiates Queue<int>::Queue(short *, short *)
Queue<int> qi(a, a + 4); // copies elements from a into qi
vector<int> vi(a, a + 4);
// instantiates Queue<int>::assign(vector<int>::iterator,
//                                 vector<int>::iterator)
qi.assign(vi.begin(), vi.end());
```

Because we are constructing an object of type `Queue<int>`, we know that the compiler will instantiate the iterator-based constructor for `Queue<int>`. The type of the constructor's own template parameter is deduced by the compiler from the type of `a` and `a + 4`. That type is pointer to `short`. Thus, the definition of `qi` instantiates

因为所构造的是 `Queue<int>` 类型的对象，我们知道编译器将为 `Queue<int>` 实例化基于迭代器的构造函数。该构造函数本身模板形参的类型由编译器根据 `a` 和 `a+4` 的类型推断，而该类型为 `short` 指针。因此，`qi` 的定义将实例化

```
void Queue<int>::Queue(short *, short *);
```

The effect of this constructor is to copy the elements of type `short` from the array named `a` into `qi`.

这个构造函数的效果是，从名为 `a` 的数组中复制 `short` 类型的元素到 `qi`。

The call to `assign` instantiates a member of `qi`, which has type `Queue<int>`. Thus, this call instantiates the `Queue<int>` member named `assign`. That function is itself a function template. As with any other function template, the compiler deduces the template argument for `assign` from the arguments to the call. The type deduced is `vector<int>::iterator`, meaning that this call instantiates

对 `assign` 的调用将实例化 `qi` 的成员。`qi` 具有 `Queue<int>` 类型，因此，这个调用将实例化名为 `assign` 的 `Queue<int>` 成员。该函数本身是函数模板，像对任意其他函数模板一样，编译器从传给调用的实参推断 `assign` 的模板实参，推断得到的类型是 `vector<int>::iterator`，即，这个调用将实例化

```
void Queue<int>::assign(vector<int>::iterator,
                        vector<int>::iterator);
```

16.4.6. The Complete `Queue` Class

16.4.6. 完整的 `Queue` 类

For completeness, here is the final definition of our `Queue` class:

为了完整起见，在这里给出 `Queue` 类的最终定义：

```
// declaration that Queue is a template needed for friend declaration in QueueItem
template <class Type> class Queue;
// function template declaration must precede friend declaration in QueueItem
template <class T>
std::ostream& operator<<(std::ostream&, const Queue<T>&);

template <class Type> class QueueItem {
    friend class Queue<Type>;
    // needs access to item and next
    friend std::ostream&      // defined on page 659
        operator<< (std::ostream&, const Queue<Type>&);

// private class: no public section
    QueueItem(const Type &t): item(t), next(0) { }
    Type item;                // value stored in this element
    QueueItem *next;          // pointer to next element in the queue
};

template <class Type> class Queue {
```

Section 16.4. Class Template Members

```
// needs access to head
friend std::ostream& // defined on page 659
operator<< <Type> (std::ostream&, const Queue<Type>&);
public:
    // empty Queue
    Queue(): head(0), tail(0) { }
    // construct a Queue from a pair of iterators on some sequence
    template <class It>
    Queue(It beg, It end):
        head(0), tail(0) { copy_elems(beg, end); }
    // copy control to manage pointers to QueueItems in the Queue
    Queue(const Queue &Q): head(0), tail(0)
    { copy_elems(Q); }
    Queue& operator=(const Queue&); // left as exercise for the reader
    ~Queue() { destroy(); }
    // replace current Queue by contents delimited by a pair of iterators
    template <class Iter> void assign(Iter, Iter);
    // return element from head of Queue
    // unchecked operation: front on an empty Queue is undefined
    Type& front() { return head->item; }
    const Type &front() const { return head->item; }
    void push(const Type &); // defined on page 652
    void pop(); // defined on page 651
    bool empty() const { // true if no elements in the Queue
        return head == 0;
    }
private:
    QueueItem<Type> *head; // pointer to first element in Queue
    QueueItem<Type> *tail; // pointer to last element in Queue
    // utility functions used by copy constructor, assignment, and destructor
    void destroy(); // defined on page 651
    void copy_elems(const Queue&); // defined on page 652
    // version of copy to be used by assign to copy elements from iterator range
    // defined on page 662
    template <class Iter> void copy_elems(Iter, Iter);
};

// Inclusion Compilation Model: include member function definitions as well
#include "Queue.cc"
```

Members that are not defined in the class itself can be found in earlier sections of this chapter; the comment following such members indicates the page on which the definition can be found.

未在类本身中定义的成员可在本章前面几节中找到，跟在这些成员后面的注释指出了可以在哪一节找到它们。

Exercises Section 16.4.6

Exercise Add the `assign` member and a constructor that takes a pair of iterators to your `List` class.

16.43:

为你的 `List` 类增加 `assign` 成员和一个参数为一对迭代器的构造函数。

Exercise We implemented our own `Queue` class in order to illustrate how class templates are implemented. One way in which our implementation could be simplified would be to define `Queue` on top of one of the existing library container types. That way, we could avoid having to manage the allocation and deallocation of the `Queue` elements. Reimplement `Queue` using `std::list` to hold the actual `Queue` elements.

为了举例说明怎样实现类模板，我们实现了自己的 `Queue` 类。可以简化实现的一种方式可能是将 `Queue` 定义在一个现存的标准库容器类型之上，用这种方法，可以避免必须管理 `Queue` 元素的分配和回收。用 `std::list` 保存实际 `Queue` 元素，重新实现 `Queue` 类。

16.4.7. `static` Members of Class Templates

16.4.7. 类模板的 `static` 成员

A class template can declare `static` members ([Section 12.6](#), p. 467) in the same way as any other class:

Section 16.4. Class Template Members

类模板可以像任意其他类一样声明 `static` 成员（第 12.6 节）。以下代码：

```
template <class T> class Foo {
public:
    static std::size_t count() { return ctr; }
    // other interface members
private:
    static std::size_t ctr;
    // other implementation members
};
```

defines a class template named `Foo` that among other members has a `public static` member function named `count` and a `private static` data member named `ctr`.

定义了名为 `Foo` 的类模板，它有一个名为 `count` 的 `public static` 成员函数和一个名为 `ctr` 的 `private static` 数据成员。

Each instantiation of class `Foo` has its own `static` member:

`Foo` 类的每个实例化有自己的 `static` 成员：

```
// Each object shares the same Foo<int>::ctr and Foo<int>::count members
Foo<int> fi, fi2, fi3;
// has static members Foo<string>::ctr and Foo<string>::count
Foo<string> fs;
```

Each instantiation represents a distinct type, so there is one `static` shared among the objects of any given instantiation. Hence, any objects of type `Foo<int>` share the same `static` member `ctr`. Objects of type `Foo<string>` share a different `ctr` member.

每个实例化表示截然不同的类型，所以给定实例外星人所有对象都共享一个 `static` 成员。因此，`Foo<int>` 类型的任意对象共享同一 `static` 成员 `ctr`，`Foo<string>` 类型的对象共享另一个不同的 `ctr` 成员。

Using a `static` Member of a Class Template

使用类模板的 `static` 成员

As usual, we can access a `static` member of a class template through an object of the class type or by using the scope operator to access the member directly. Of course, when we attempt to use the `static` member through the class, we must refer to an actual instantiation:

通常，可以通过类类型的对象访问类模板的 `static` 成员，或者通过使用作用域操作符直接访问成员。当然，当试图通过类使用 `static` 成员的时候，必须引用实际的实例化：

```
Foo<int> fi, fi2;           // instantiates Foo<int> class
size_t ct = Foo<int>::count(); // instantiates Foo<int>::count
ct = fi.count();             // ok: uses Foo<int>::count
ct = fi2.count();            // ok: uses Foo<int>::count
ct = Foo::count();           // error: which template instantiation?
```

Like any other member function, a `static` member function is instantiated only if it is used in a program.

与任意其他成员函数一样，`static` 成员函数只有在程序中使用时才进行实例化。

Defining a `static` Member

定义 `static` 成员

As with any other `static` data member, there must be a definition for the data member that appears outside the class. In the case of a class template `static`, the member definition must indicate that it is for a class template:

像使用任意其他 `static` 数据成员一样，必须在类外部出现数据成员的定义。在类模板含有 `static` 成员的情况下，成员定义必须指出它是类模板的成员：

```
template <class T>
size_t Foo<T>::ctr = 0; // define and initialize ctr
```

A `static` data member is defined like any other member of a class template that is defined outside the class. It begins with the keyword `template` followed by the class template parameter list and the class name. In this case, the name of the `static` data member is prefixed by `Foo<T>::`, which

Section 16.4. Class Template Members

indicates that the member belongs to the class template `Foo`.

`static` 数据成员像定义在类外部的任意其他类成员一样定义，它用关键字 `template` 开头，后面接着类模板形参表和类名。在这个例子中，`static` 数据成员的名字以 `Foo<T>::` 为前缀，表示成员属于类模板 `Foo`。

Team LiB

◀ PREVIOUS NEXT ▶

16.5. A Generic Handle Class

16.5. 一个泛型句柄类



This example represents a fairly sophisticated use of C++. Understanding it requires understanding both inheritance and templates fairly well. It may be useful to delay studying this example until you are comfortable with these features. On the other hand, this example provides a good test of your understanding of these features.

这个例子体现了 C++ 相当复杂的语言应用，理解它需要很好地理解继承和模板。在熟悉了这些特性之后再研究这个例子也许会帮助。另一方面，这个例子还能很好地测试你对这些我的理解程序。

In [Chapter 15](#) we defined two handle classes: the `Sales_item` ([Section 15.8](#), p. 598) class and the `Query` ([Section 15.9](#), p. 607) class. These classes managed pointers to objects in an inheritance hierarchy. Users of the handle did not have to manage the pointers to those objects. User code was written in terms of the handle class. The handle dynamically allocated and freed objects of the related inheritance classes and forwarded all "real" work back to the classes in the underlying inheritance hierarchy.

在[第十五章](#)定义了两个句柄类：`Sales_item` 类 ([第 15.8 节](#)) 和 `Query` 类 ([第 15.9 节](#))。这两个类管理继承层次中对象的指针，句柄的用户不必管理指向这些对象的指针，用户代码可以使用句柄类来编写。句柄能够动态分配和释放相关继承类的对象，并且将所有“实际”工作转发给继承层次中的底层类。

These handles were similar to but different from each other: They were similar in that each defined use-counted copy control to manage a pointer to an object of a type in an inheritance hierarchy. They differed with respect to the interface they provided to users of the inheritance hierarchy.

这两个句柄类似但并不相同：类似之处在于都定义了使用计数式的复制控制，管理指向继承层次中某类型对象的指针；不同之处在于它们提供给继承层次用户的接口。

The use-counting implementation was the same in both classes. This kind of problem is well suited to generic programming: We could define a class template to manage a pointer and do the use-counting. Our otherwise unrelated `Sales_item` and `Query` types could be simplified by using that template to do the common use-counting work. The handles would remain different as to whether they reveal or hide the underlying inheritance hierarchy.

两个类的使用计数的实现是相同的。这类问题非常适合于泛型编程：可以定义类模板管理指针和进行使用计数。原本不相关的 `Sales_item` 类型和 `Query` 类型，可通过使用该模板进行公共的使用计数工作面得以简化。至于是公开还是隐藏下层的继承层次，句柄可以保持不同。

In this section, we'll implement a [generic handle class](#) to provide the operations that manage the use count and the underlying objects. Then we'll rewrite the `Sales_item` class, showing how it could use the generic handle rather than defining its own use-counting operations.

本节将实现一个泛型句柄类 (generic handle class)，提供管理使用计数和基础对象的操作。然后，我们重新编写 `Sales_item` 类，展示它怎样使用泛型句柄而不是定义自己的使用计数操作。

16.5.1. Defining the Handle Class

16.5.1. 定义句柄类

Our `Handle` class will behave like a pointer: Copying a `Handle` will not copy the underlying object. After the copy, both `Handles` will refer to the same underlying object. To create a `Handle`, a user will be expected to pass the address of a dynamically allocated object of the type (or a type derived from that type) managed by the `Handle`. From that point on, the `Handle` will "own" the given object. In particular, the `Handle` class will assume responsibility for deleting that object once there are no longer any `Handles` attached to it.

`Handle` 类行为类似于指针：复制 `Handle` 对象将不会复制基础对象，复制之后，两个 `Handle` 对象将引用同一基础对象。要创建 `Handle` 对象，用户需要传递属于由 `Handle` 管理的类型（或从该类型派生的类型）的动态分配对象的地址，从此刻起，`Handle` 将“拥有”这个对象。而且，一旦不再有任意 `Handle` 对象与该对象关联，`Handle` 类将负责删除该对象。

Given this design, here is an implementation of our generic `Handle`:

对于这一设计，我们的泛型 `Handle` 类的实现如下：

```
/* generic handle class: Provides pointerlike behavior. Although access through
 * an unbound Handle is checked and throws a runtime_error exception.
 * The object to which the Handle points is deleted when the last Handle goes away.
```

Section 16.5. A Generic Handle Class

```
* Users should allocate new objects of type T and bind them to a Handle.
* Once an object is bound to a Handle, the user must not delete that object.
*/
template <class T> class Handle {
public:
    // unbound handle
    Handle(T *p = 0): ptr(p), use(new size_t(1)) { }
    // overloaded operators to support pointer behavior
    T& operator*();
    T* operator->();
    const T& operator*() const;
    const T* operator->() const;
    // copy control: normal pointer behavior, but last Handle deletes the object
    Handle(const Handle& h): ptr(h.ptr), use(h.use)
    {
        ++*use;
    }
    Handle& operator=(const Handle&);
    ~Handle() { rem_ref(); }
private:
    T* ptr;           // shared object
    size_t *use;      // count of how many Handle s point to *ptr
    void rem_ref()
    {
        if (--*use == 0) { delete ptr; delete use; }
    }
};
```

This class looks like our other handles, as does the assignment operator.

这个类看来与其他句柄类似，赋值操作符也类似。

```
template <class T>
inline Handle<T>& Handle<T>::operator=(const Handle &rhs)
{
    ++rhs.use;          // protect against self-assignment
    rem_ref();          // decrement use count and delete pointers if needed
    ptr = rhs.ptr;
    use = rhs.use;
    return *this;
}
```

The only other members our class will define are the dereference and member access operators. These operators will be used to access the underlying object. We'll provide a measure of safety by having these operations check that the `Handle` is actually bound to an object. If not, an attempt to access the object will throw an exception.

`Handle` 类将定义的其他成员是解引用操作符和成员访问操作符，这些操作符将用于访问基础对象。让这些操作检查 `Handle` 是否确实绑定到对象，可以提供一种安全措施。如果 `Handle` 没有绑定到对象，则试图访问对象将抛出一个异常。

The `nonconst` versions of these operators look like:

这些操作的非 `const` 版本看来如下所示：

```
template <class T> inline T & Handle<T>::operator*()
{
    if (ptr) return *ptr;
    throw std::runtime_error
        ("dereference of unbound Handle");
}
template <class T> inline T* Handle<T>::operator->()
{
    if (ptr) return ptr;
    throw std::runtime_error
        ("access through unbound Handle");
}
```

The `const` versions would be similar and are left as an exercise.

实现一个 `Handle` 类的自己的版本。

Exercises Section 16.5.1

Exercise

16.45: Implement your own version of the `Handle` class.

实现一个 `Handle` 类的自己的版本。

Exercise Explain what happens when an object of type `Handle` is copied.

16.46: 解释复制 `Handle` 类型的对象时会发生什么。

Section 16.5. A Generic Handle Class

Exercise 16.47: What, if any, restrictions does `Handle` place on the types used to instantiate an actual `Handle` class.

`Handle` 类对用来实例化实际 `Handle` 类的类型有限制吗？如果有，限制有哪些？

Exercise 16.48: Explain what happens if the user attaches a `Handle` to a local object. Explain what happens if the user deletes the object to which a `Handle` is attached.

解释如果用户将 `Handle` 对象与局部对象关联会发生什么。解释如果用户删除 `Handle` 对象所关联的对象会发生什么。

16.5.2. Using the Handle

16.5.2. 使用句柄

We intend this class to be used by other classes in their internal implementations. However, as an aid to understanding how the `Handle` class works, we'll look at a simpler example first. This example illustrates the behavior of the `Handle` by allocating an `int` and binding a `Handle` to that newly allocated object:

我们希望 `Handle` 类能够用于其他类的内部实现中。但是，为了帮助理解 `Handle` 类怎样工作，我们首先介绍一个较简单的例子。这个例子通过分配一个 `int` 对象，并将一个 `Handle` 对象绑定到新分配的 `int` 对象而说明 `Handle` 的行为：

```
{ // new scope
// user allocates but must not delete the object to which the Handle is attached
Handle<int> hp(new int(42));
{ // new scope
    Handle<int> hp2 = hp; // copies pointer; use count incremented
    cout << *hp << " " << *hp2 << endl; // prints 42 42
    *hp2 = 10; // changes value of shared underlying int
} // hp2 goes out of scope; use count is decremented
cout << *hp << endl; // prints 10
} // hp goes out of scope; its destructor deletes the int
```

Even though the user of `Handle` allocates the `int`, the `Handle` destructor will delete it. In this code, the `int` is deleted at the end of the outer block when the last `Handle` goes out of scope. To access the underlying object, we apply the `Handle *` operator. That operator returns a reference to the underlying `int` object.

即使是 `Handle` 的用户分配了 `int` 对象，`Handle` 析构函数也将删除它。在外层代码块末尾最后一个 `Handle` 对象超出作用域时，删除该 `int` 对象。为了访问基础对象，应用了 `Handle` 的 `*` 操作符，该操作符返回对基础 `int` 对象的引用。

Using a `Handle` to Use-Count a Pointer

使用 `Handle` 对象对指针进行使用计数

As an example of using `Handle` in a class implementation, we might reimplement our `Sales_item` class (Section 15.8.1, p. 599). This version of the class defines the same interface, but we can eliminate the copy-control members by replacing the pointer to `Item_base` by a `Handle<Item_base>`:

作为在类实现中使用 `Handle` 的例子，可以重新实现 `Sales_item` 类（第 15.8.1 节），该类的这个版本定义相同的接口，但可以通过用 `Handle<Item_base>`：对象代替 `Item_base` 指针而删去复制控制成员：

```
class Sales_item {
public:
    // default constructor: unbound handle
    Sales_item(): h() { }
    // copy item and attach handle to the copy
    Sales_item(const Item_base &item): h(item.clone()) { }
    // no copy control members: synthesized versions work
    // member access operators: forward their work to the Handle class
    const Item_base& operator*() const { return *h; }
    const Item_base* operator->() const
        { return h.operator->(); }
private:
    Handle<Item_base> h; // use-counted handle
};
```

Although the interface to the class is unchanged, its implementation differs considerably from the original:

虽然 `Sales_item` 类的接口没变，它的实现与原来的相当不同：

- Both classes define a default constructor and a constructor that takes a `const` reference to an `Item_base` object.

两个类都定义了默认构造函数和以 `Item_base` 对象为参数和 `const` 引用的构造函数。

- Both define overloaded `*` and `->` operators as `const` members.

两个类都将重载的 `*` 和 `->` 操作符定义为 `const` 成员。

The `Handle`-based version of `Sales_item` has a single data member. That data member is a `Handle` attached to a copy of the `Item_base` object given to the constructor. Because this version of `Sales_item` has no pointer members, there is no need for copy-control members. This version of `Sales_item` can safely use the synthesized copy-control members. The work of managing the use-count and associated `Item_base` object is done inside `Handle`.

基于 `Handle` 的 `Sales_item` 版本有一个数据成员，该数据成员是关联传给构造函数的 `Item_base` 对象的副本上的 `Handle` 对象。因为 `Sales_item` 的这个版本没有指针成员，所以不需要复制控制成员，`Sales_item` 的这个版本可以安全地使用合成的复制控制成员。管理使用计数和相关 `Item_base` 对象的工作在 `Handle` 内部完成。

Because the interface is unchanged, there is no need to change code that uses `Sales_item`. For example, the program we wrote in [Section 15.8.3](#) (p. 603) can be used without change:

因为接口没变，所以不需要改变使用 `Sales_item` 类的代码。例如，[第 15.8.3 节](#) 中编写的程序可以无须改变而使用：

```
double Basket::total() const
{
    double sum = 0.0; // holds the running total
    /* find each set of items with the same isbn and calculate
     * the net price for that quantity of items
     * iter refers to first copy of each book in the set
     * upper_boundrefers to next element with a different isbn
     */
    for (const_iter iter = items.begin();
         iter != items.end();
         iter = items.upper_bound(*iter))
    {
        // we know there's at least one element with this key in the Basket
        // virtual call to net_price applies appropriate discounts, if any
        sum += (*iter)->net_price(items.count(*iter));
    }
    return sum;
}
```

It's worthwhile to look in detail at the statement that calls `net_price`:

调用 `net_price` 函数的语句值得仔细分析一下：

```
sum += (*iter)->net_price(items.count(*iter));
```

This statement uses operator `->` to fetch and run the `net_price` function. What's important to understand is how this operator works:

这个语句使用 `->` 操作符获取并运行 `net_price` 函数，重要的是理解这个操作符怎样工作：

- `(*iter)` returns `h`, our use-counted handle member.

`(*iter)` 返回 `h`，`h` 是使用计数式句柄的成员。

- `(*iter)->` therefore uses the overloaded arrow operator of the handle class

因此，`(*iter)->` 使用句柄类的重载箭头操作符。

- The compiler evaluates `h.operator->()`, which in turn yields the pointer to `Item_base` that the `Handle` holds.

编译器计算 `h.operator->()`，获得该 `Handle` 对象保存的 `Item_base` 指针。

- The compiler dereferences that `Item_base` pointer and calls the `net_price` member for the object to which the pointer points.

编译器对该 `Item_base` 指针解引用，并调用指针所指对象的 `net_price` 成员。

Exercises Section 16.5.2

Exercise

16.49: Implement the version of the `Sales_item` handle presented here that uses the generic `Handle` class to manage the pointer to `Item_base`.

实现本节提出的 `Sales_item` 句柄的版本，该版本使用泛型 `Handle` 类管理 `Item_base` 指针。

Exercise

16.50: Rerun the function to total a sale. List all changes you had to make to get your code to work.

重新运行函数计算销售总额。列出让你的代码工作必须进行的所有修改。

Exercise

16.51: Rewrite the `Query` class from [Section 15.9.4](#) (p. 613) to use the generic `Handle` class. Note that you will need to make the `Handle` a friend of the `Query_base` class to let it access the `Query_base` destructor. List and explain all other changes you made to get the programs to work.

重新编写[Section 15.9.4第 15.9.4 节](#)的 `Query` 类以使用泛型 `Handle` 类。注意你需要将 `Handle` 类设为 `Query_base` 类的友元，以便它能够访问 `Query_base` 构造函数。列出并解释让程序工作要做的其他所有修改。

16.6. Template Specializations

16.6. 模板特化

The rest of this chapter covers a somewhat advanced topic. It can be safely skipped on first reading.

本章其余部分将介绍一个比较高级的主题，在第一次阅读时可以跳过它。



It is not always possible to write a single template that is best suited for every possible template argument with which the template might be instantiated. In some cases, the general template definition is simply wrong for a type. The general definition might not compile or might do the wrong thing. At other times, we may be able to take advantage of some specific knowledge about a type to write a more efficient function than the one that is instantiated from the template.

我们并不总是能够写出对所有可能被实例化的类型都最合适的模板。某些情况下，通用模板定义对于某个类型可能是完全错误的，通用模板定义也许不能编译或者做错误的事情；另外一些情况下，可以利用关于类型的一些特殊知识，编写比从模板实例化来的函数更有效率的函数。

Our `compare` function and our `Queue` class are both good examples of the problem: Neither works correctly when used with C-style character strings. Let's look again at our `compare` function template:

`compare` 函数和 `Queue` 类都是这一问题的好例子：与 C 风格字符串一起使用进，它们都不能正确工作。让我们再来看看 `compare` 函数模板：

```
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

If we call this template definition on two `const char*` arguments, the function compares the pointer values. It will tell us the relative positions in memory of these two pointers but says nothing about the contents of the arrays to which the pointers point.

如果用两个 `const char*` 实参调用这个模板定义，函数将比较指针值。它将告诉我们这两个指针在内存中的相对位置，但没有说明归纳法指针所指数组的内容有关的任何事情。

To get be able to use `compare` with character strings, we would have to provide a specialized definition that knows how to compare C-style strings. The fact that these versions are specialized is transparent to users of these templates. Calls to a specialized function or use of a specialized class are indistinguishable from uses of a version instantiated from the general template.

为了能够将 `compare` 函数用于字符串，必须提供一个知道怎样比较 C 风格字符串的特殊定义。这些版本是特化的，这一事实对模板的用户透明。对用户而言，调用特化函数或使用特化类，与使用从通用模板实例化的版本无法区别。

16.6.1. Specializing a Function Template

16.6.1. 函数模板的特化

A **template specialization** is a separate definition in which the actual type(s) or value(s) of one or more template parameter(s) is (are) specified. The form of a specialization is:

模板特化（**template specialization**）是这样的一个定义，该定义中一个或多个模板形参的实际类型或实际值是指定的。特化的形式如下：

- The keyword `template` followed by an empty bracket pair (`<>`),

关键字 `template` 后面接一对空的尖括号 (`<>`)；

Section 16.6. Template Specializations

- followed by the template name and a bracket pair specifying the template parameters(s) that this specialization defines,
再接模板名和一对尖括号，尖括号中指定这个特化定义的模板形参；
- the function parameter list,
函数形参表；
- and the function body.
函数体。

The following program defines a specialization of `compare` when the template parameter type is bound to `const char*`:

下面的程序定义了当模板形参类型绑定到 `const char*` 时，`compare` 函数的特化：

```
// special version of compare to handle C-style character strings
template <>
int compare<const char*>(const char* const &v1,
                           const char* const &v2)
{
    return strcmp(v1, v2);
}
```

The declaration for the specialization must match that of the corresponding template. In this case, the template has one type parameter and two function parameters. The function parameters are `const` references to the type parameter. Here we are fixing the type parameter to `const char*`; our function parameters, therefore, are `const` references to a `const char*`.

特化的声明必须与对应的模板相匹配。在这个例子中，模板有一个类型形参和两个函数形参，函数形参是类型形参的 `const` 引用，在这里，将类型形参固定为 `const char*`，因此，函数形参是 `const char*` 的 `const` 引用。

Now when we call `compare`, passing it two character pointers, the compiler will call our specialized version. It will call the generic version for any other argument types (including plain `char*`):

现在，当调用 `compare` 函数的时候，传给它两个字符指针，编译器将调用特化版本。编译器将为任意其他实参类型（包括普通 `char*`）调用泛型版本：

```
const char *cp1 = "world", *cp2 = "hi";
int i1, i2;
compare(cp1, cp2); // calls the specialization
compare(i1, i2); // calls the generic version instantiated with int
```

Declaring a Template Specialization

声明模板特化

As with any function, we can declare a function template specialization without defining it. A template specialization declaration looks like the definition but omits the function body:

与任意函数一样，函数模板特化可以声明而无须定义。模板特化声明看起来与定义很像，但省略了函数体：

```
// declaration of function template explicit specialization
template<>
int compare<const char*>(const char* const&,
                           const char* const&);
```

This declaration consists of an empty template parameter list (`template<>`) followed by the return type, the function name (optionally) followed by explicit template argument(s) specified inside a pair of angle brackets, and the function parameter list. A template specialization must always include the empty template parameter specifier, `template<>`, and it must include the function parameter list. If the template arguments can be inferred from the function parameter list, there is no need to explicitly specify the template arguments:

这个声明由一个后接返回类型的空模板形参表 (`template<>`)，后接一对尖括号中指定的显式模板实参的函数名（可选），以及函数形参表构成。模板特化必须总是包含空模板形参说明符，即 `template<>`，而且，还必须包含函数形参表。如果可以从函数形参表推断模板实参，则不必显式指定模板实参：

```
// error: invalid specialization declarations
// missing template<>
int compare<const char*>(const char* const&,
                           const char* const&);

// error: function parameter list missing
template<> int compare<const char*>;
```

```
// ok: explicit template argument const char* deduced from parameter types
template<> int compare(const char* const&,
                      const char* const&);
```

Function Overloading versus Template Specializations

函数重载与模板特化

Omitting the empty template parameter list, `template<>`, on a specialization may have surprising effects. If the specialization syntax is missing, then the effect is to declare an overloaded nontemplate version of the function:

在特化中省略空的模板形参表 `template<>` 会有令人惊讶的结果。如果缺少该特化语法，则结果是声明该函数的重载非模板版本：

```
// generic template definition
template <class T>
int compare(const T& t1, const T& t2) { /* ... */ }

// OK: ordinary function declaration
int compare(const char* const&, const char* const&);
```

The definition of `compare` does not define a template specialization. Instead, it declares an ordinary function with a return type and a parameter list that could match those of a template instantiation.

`compare` 的定义没有定义模板特化，相反，它声明了一个普通函数，该函数含有返回类型和可与模板实例化相匹配的形参表。

We'll look at the interaction of overloading and templates in more detail in the next section. For now, what's important to know is that when we define a nontemplate function, normal conversions are applied to the arguments. When we specialize a template, conversions are not applied to the argument types. In a call to a specialized version of a template, the argument type(s) in the call must match the specialized version function parameter type(s) exactly. If they don't, then the compiler will instantiate an instantiation for the argument(s) from the template definition.

下一节将更详细地介绍重载和模板的交互作用。现在，重要的是知道，当定义非模板函数的时候，对实参应用常规转换；当特化模板的时候，对实参类型不应用转换。在模板特化版本的调用中，实参类型必须与特化版本函数的形参类型完全匹配，如果不完全匹配，编译器将为实参从模板定义实例化一个实例。

Duplicate Definitions Cannot Always Be Detected

不是总能检测到重复定义

If a program consists of more than one file, the declaration for a template specialization must be visible in every file in which the specialization is used. A function template cannot be instantiated from the generic template definition in some files and be specialized for the same set of template arguments in other files.

如果程序由多个文件构成，模板特化的声明必须在使用该特化的每个文件中出现。不能在一些文件中从泛型模板定义实例化一个函数模板，而在其他文件中为同一模板实参集合特化该函数模板。



As with other function declarations, declarations for template specializations should be included in a header file. That header should then be included in every source file that uses the specialization.

与其他函数声明一样，应在一个头文件中包含模板特化的声明，然后使用该特化的每个源文件包含该头文件。

Ordinary Scope Rules Apply to Specializations

普通作用域规则适用于特化

Before we can declare or define a specialization, a declaration for the template that it specializes must be in scope. Similarly, a declaration for the specialization must be in scope before that version of the template is called:

在能够声明或定义特化之前，它所特化的模板的声明必须在作用域中。类似地，在调用模板的这个版本之前，特化的声明必须在作用域中：

Section 16.6. Template Specializations

```
// define the general compare template
template <class T>
int compare(const T& t1, const T& t2) { /* ... */ }

int main() {
    // uses the generic template definition
    int i = compare("hello", "world");
    // ...
}

// invalid program: explicit specialization after call
template<>
int compare<const char*>(const char* const& s1,
                           const char* const& s2)
{ /* ... */ }
```

This program is in error because a call that would match the specialization is made before the specialization is declared. When the compiler sees a call, it must know to expect a specialization for this version. Otherwise, the compiler is allowed to instantiate the function from the template definition.

这个程序有错误，因为在声明特化之前，进行了可以与特化相匹配的一个调用。当编译器看到一个函数调用时，它必须知道这个版本需要特化，否则，编译器将可能从模板定义实例化该函数。



A program cannot have both an explicit specialization and an instantiation for the same template with the same set of template arguments.

对具有同一模板实参集的同一模板，程序不能既有显式特化又有实例化。

It is an error for a specialization to appear after a call to that instance of the template has been seen.

特化出现在对该模板实例的调用之后是错误的。

Exercises Section 16.6.1

Exercise 16.52: Define a function template `count` to count the number of occurrences of some value in a `vector`.

定义函数模板 `count` 计算一个 `vector` 中某些值的出现次数。

Exercise 16.53: Write a program to call the `count` function defined in the previous exercise passing it first a `vector` of `doubles`, then a `vector` of `ints`, and finally a `vector` of `chars`.

编写一个程序调用上题中定义的 `count` 函数，首先传给该函数一个 `double` 型 `vector`，然后传递一个 `int` 型 `vector`，最后传递一个 `char` 型 `vector`。

Exercise 16.54: Introduce a specialized template instance of the `count` function to handle `strings`. Rerun the program you wrote to call the function template instantiations.

引入 `count` 函数的一个特化模板实例以处理 `string` 对象。重新运行你所编写的调用函数模板实例化的程序。

16.6.2. Specializing a Class Template

16.6.2. 类模板的特化

Our `Queue` class has a problem similar to the one in `compare` when used with C-style strings. In this case, the problem is in the `push` function. That function copies the value it's given to create a new element in the `Queue`. By default, copying a C-style character string copies only the pointer, not the characters. Copying a pointer in this case has all the problems that shared pointers have in other contexts. The most serious is that if the pointer points to dynamic memory, it's possible for the user to delete the array to which the pointer points.

当用于 C 风格字符串时，`Queue` 类具有与 `compare` 函数相似的问题。在这种情况下，问题出在 `push` 函数中，该函数复制给定值以创建 `Queue` 中的新元素。默认情况下，复制 C 风格字符串只会复制指针，不会复制字符。这种情况下复制指针将出现共享指针在其他环境中会出现的所有问题，最严重的是，如果指针指向动态内存，用户就有可能删

Section 16.6. Template Specializations

除指针所指的数组。

Defining a Class Specialization

定义类特化

One way to provide the right behavior for `Queue`'s of C-style strings is to define a specialized version of the entire class for `const char*`:

为 C 风格字符串的 `Queue` 提供正确行为的一种途径，是为 `const char*` 定义整个类的特化版本：

```
/* definition of specialization for const char*
 * this class forwards its work to Queue<string>;
 * the push function translates the const char* parameter to a string
 * the front functions return a string rather than a const char*
 */
template<> class Queue<const char*> {
public:
    // no copy control: Synthesized versions work for this class
    // similarly, no need for explicit default constructor either
    void push(const char*);           {real_queue.pop();}
    void pop() const                {return real_queue.empty();}
    // Note: return type does not match template parameter type
    std::string front() const        {return real_queue.front();}
    const std::string &front() const   {return real_queue.front();}

private:
    Queue<std::string> real_queue; // forward calls to real_queue
};
```

This implementation gives `Queue` a single data element: a `Queue` of `strings`. The various members delegate their work to this member for example, `pop` is implemented by calling `pop` on `real_queue`.

这个实现给了 `Queue` 一个数据元素：`string` 对象的 `Queue`。各个成员将它们的工作委派给这个成员。例如，通过调用 `real_queue` 的 `pop` 实现 `pop` 成员。

This version of the class does not define the copy-control members. Its only data element has a class type that does the right thing when copied, assigned, or destroyed; we can use the synthesized copy-control members.

`Queue` 类的这个版本没有定义复制控制成员，它唯一的数据成员为类类型，该类类型在被复制、被赋值或被撤销时完成正确的工作。可以使用合成的复制控制成员。

Our `Queue` class implements mostly, but not entirely, the same interface as the template version of `Queue`. The difference is that we return a `string` rather than a `char*` from the `front` members. We do so to avoid having to manage the character array that would be required if we wanted to return a pointer.

这个 `Queue` 类实现了与 `Queue` 的模板版本大部分相同但不完全相同的接口，区别在于 `front` 成员返回的是 `string` 而不是 `char*`，这样做是为了避免必须管理字符数组——如果想要返回指针，就需要字符数组。

It is worth noting that a specialization may define completely different members than the template itself. If a specialization fails to define a member from the template, that member may not be used on objects of the specialization type. The member definitions of the class template are not used to create the definitions for the members of an explicit specialization.

值得注意的是，特化可以定义与模板本身完全不同的成员。如果一个特化无法从模板定义某个成员，该特化类型的对象就不能使用该成员。类模板成员的定义不会用于创建显式特化成员的定义。



A class template specialization ought to define the same interface as the template it specializes. Doing otherwise will surprise users when they attempt to use a member that is not defined.

类模板特化应该与它所特化的模板定义相同的接口，否则当用户试图使用未定义的成员时会感到奇怪。

Class Specialization Definition

类特化定义

When a member is defined outside the class specialization, it is not preceded by the tokens `template<>`.



在类特化外部定义成员时，成员之前不能加 `template<>` 标记。

Our class defines only one member outside the class:

我们的类只在类的外部定义了一个成员：

```
void Queue<const char*>::push(const char* val)
{
    return real_queue.push(val);
}
```

Although it does little obvious work, this function implicitly copies the character array to which `val` points. The copy is made in the call to `real_queue.push`, which creates a new `string` from the `const char*` argument. That argument uses the `string` constructor that takes a `const char*`. The `string` constructor copies the characters from the array pointed to by `val` into an unnamed `string` that will be stored in the element we push onto `real_queue`.

虽然这个函数几乎没有做什么工作，但它隐式复制了 `val` 指向的字符数组。复制是在对 `real_queue.push` 的调用中进行的，该调用从 `const char*` 实参创建了一个新的 `string` 对象。`const char*` 实参使用了以 `const char*` 为参数的 `string` 构造函数，`string` 构造函数将 `val` 所指的数组中的字符复制到未命名的 `string` 对象，该对象将被存储在 `push` 到 `real_queue` 的元素中。

Exercises Section 16.6.2

Exercise 16.55: The comments on the specialized version of `Queue` for `const char*` note that there is no need to define the default constructor or copy-control members. Explain why the synthesized members suffice for this version of `Queue`.

`Queue` 针对 `const char*` 的特化版本中的注释指出，不必定义默认构造函数或复制控制成员，解释为什么对于 `Queue` 的这个版本合成成员就足够了。

Exercise 16.56: We explained the generic behavior of `Queue` if it is not specialized for `const char*`. Using the generic `Queue` template, explain what happens in the following code:

我们已经解释过未针对 `const char*` 特化的 `Queue` 的泛型行为，使用泛型 `Queue` 模板解释下面代码中会发生什么：

```
Queue<const char*> q1;
q1.push("hi"); q1.push("bye"); q1.push("world");
Queue<const char*> q2(q1); // q2 is a copy of q1

Queue<const char*> q3;      // empty Queue
q1 = q3;
```

In particular, say what the values of `q1` and `q2` are after the initialization of `q2` and after the assignment to `q3`.

具体而言，就是说明在 `q2` 的初始化和 `q3` 的赋值之后，`q1` 和 `q2` 是什么值。

Exercise 16.57: Our specialized `Queue` returns `strings` from the `front` function rather than `const char*`. Why do you suppose we did so? How might you implement the `Queue` to return a `const char*`? Discuss the pros and cons of each approach.

我们的 `Queue` 特化版本从 `front` 函数返回 `string` 对象而不是 `const char*`，你认为为什么这样做？你能够怎样实现 `Queue` 以返回 `const char*`？讨论每种方法的优缺点。

16.6.3. Specializing Members but Not the Class

16.6.3. 特化成员而不特化类

If we look a bit more deeply at our class, we can see that we can simplify our code: Rather than specializing the whole template, we can specialize just the `push` and `pop` members. We'll specialize `push` to copy the character array and `pop` to free the memory we used for that copy:

如果更深入一点分析我们的类，就能够看到代码可以简化：除了特化整个模板之外，还可以只特化 `push` 和 `pop` 成员。我们将特化 `push` 成员以复制字符数组，并且特化 `pop` 成员以释放该副本使用的内存：

```
template <>
void Queue<const char*>::push(const char *const &val)
{
    // allocate a new character array and copy characters from val
    char* new_item = new char[strlen(val) + 1];
    strcpy(new_item, val, strlen(val) + 1);
    // store pointer to newly allocated and initialized element
    QueueItem<const char*> *pt =
        new QueueItem<const char*>(new_item);
    // put item onto existing queue
    if (empty())
        head = tail = pt; // queue has only one element
    else {
        tail->next = pt; // add new element to end of queue
        tail = pt;
    }
}
template <>
void Queue<const char*>::pop()
{
    // remember head so we can delete it
    QueueItem<const char*> *p = head;
    delete head->item; // delete the array allocated in push
    head = head->next; // head now points to next element
    delete p;           // delete old head element
}
```

Now, the class type `Queue<const char*>` will be instantiated from the generic class template definition, with the exception of the `push` and `pop` functions. When we call `push` or `pop` on a `Queue<const char*>`, then the specialized version will be called. When we use any other member, the generic one will be instantiated for `const char*` from the class template.

现在，类类型 `Queue<const char*>` 将从通用类模板定义实例化而来，而 `push` 和 `pop` 函数例外。调用 `Queue<const char*>` 对象的 `push` 或 `pop` 函数时，将调用特化版本；调用任意其他成员时，将从类模板为 `const char*` 实例化一个通用版本。

Specialization Declarations

特化声明

Member specializations are declared just as any other function template specialization. They must start with an empty template parameter list:

成员特化的声明与任何其他函数模板特化一样，必须以空的模板形参表开头：

```
// push and pop specialized for const char*
template <>
void Queue<const char*>::push(const char* const &);
template <> void Queue<const char*>::pop();
```

These declarations should be placed in the `Queue` header file.

这些声明应放在 `Queue` 类的头文件中。

Exercises Section 16.6.3

Exercise

16.58:

The specialization of `Queue` presented in the previous subsection and the specialization in this subsection of `push` and `pop` apply only to `Queues` of `const char*`. Implement these two different ways of specializing `Queue` that could be used with plain `char*`.

前一小节中给出的 `Queue` 类的特化，以及本小节中 `push` 和 `pop` 函数的特化，只适用于 `const char*` 类型的 `Queue`。为普通 `char*` 实现特化 `Queue` 的这两种不同方式。

Exercise 16.59: If we go the route of specializing only the `push` function, what value is returned by `front` for a `Queue` of C-style character strings?

如果走只特化 `push` 函数的路线，对于 C 风格字符串的 `Queue`, `front` 返回什么值？

Exercise 16.60: Discuss the pros and cons of the two designs: defining a specialized version of the class for `const char*` versus specializing only the `push` and `pop` functions. In particular, compare and contrast the behavior of `front` and the possibility of errors in user code corrupting the elements in the `Queue`.

讨论这两个设计的优缺点：为 `const char*` 定义该类的特化版本和只特化 `push` 和 `pop` 函数。具体而言，比较 `front` 的行为的异同以及用户代码中的错误破坏 `Queue` 元素的可能性。

16.6.4. Class-Template Partial Specializations

16.6.4. 类模板的部分特化

If a class template has more than one template parameter, we might want to specialize some but not all of the template parameters. We can do so using a class template partial specialization:

如果类模板有一个以上的模板形参，我们也许想要特化某些模板形参而非全部。使用类模板的部分特化可以做到这一点：

```
template <class T1, class T2>
class some_template {
    // ...
};

// partial specialization: fixes T2 as int and allows T1 to vary
template <class T1>
class some_template<T1, int> {
    // ...
};
```

A class template **partial specialization** is itself a template. The definition of a partial specialization looks like a template definition. Such a definition begins with the keyword `template` followed by a template parameter list enclosed by angle brackets (`<>`). The parameter list of a partial specialization is a subset of the parameter list of the corresponding class template definition. The partial specialization for `some_template` has only one template type parameter named `T1`. The second template argument for `T2` is known to be `int`. The template parameter list for the partial specialization only lists the parameters for which the template arguments are still unknown.

类模板的**部分特化**本身也是模板。部分特化的定义看来像模板定义，这种定义以关键字 `template` 开头，接着是由尖括号 (`<>`) 括住的模板形参表。部分特化的模板形参表是对应的类模板定义形参表的子集。`some_template` 的部分特化只有一个名为 `T1` 的模板类型形参，第二个模板形参 `T2` 的实参已知为 `int`。部分特化的模板形参表只列出未知模板实参的那些形参。

Using a Class-Template Partial Specialization

使用类模板的部分特化

The partial specialization has the same name as the class template to which it correspondsnamely, `some_template`. The name of the class template must be followed by a template argument list. In the previous example, the template argument list is `<T1, int>`. Because the argument value for the first template parameter is unknown, the argument list uses the name of the template parameter `T1` as a placeholder. The other argument is the type `int`, for which the template is partially specialized.

部分特化与对应类模板有相同名字，即这里的 `some_template`。类模板的名字后面必须接着模板实参列表，前面例子中，模板实参列表是 `<T1, int>`。因为第一个模板形参的实参值未知，实参列表使用模板形参名 `T1` 作为占位符，另一个实参是类型 `int`，为 `int` 而部分特化模板。

As with any other class template, a partial specialization is instantiated implicitly when used in a program:

像任何其他类模板一样，部分特化是在程序中使用时隐式实例化：

```
some_template<int, string> foo; // uses template
some_template<string, int> bar; // uses partial specialization
```

Notice that the type of the second variable, `some_template` parameterized by `string` and `int`, could be instantiated from the generic class template

Section 16.6. Template Specializations

definition as well as from the partial specialization. Why is it that the partial specialization is chosen to instantiate the template? When a partial specialization is declared, the compiler chooses the template definition that is the most specialized for the instantiation. When no partial specialization can be used, the generic template definition is used. The instantiated type of `foo` does not match the partial specialization provided. Thus, the type of `foo` must be instantiated from the general class template, binding `int` to `T1` and `string` to `T2`. The partial specialization is only used to instantiate `some_template` types with a second type of `int`.

注意第二个变量的类型，形参为 `string` 和 `int` 的 `some_template`，既可以从普通类模板定义实例化，也可以从部分特化实例化。为什么选择部分特化来实例化该模板呢？当声明了部分特化的时候，编译器将为实例化选择最特化的模板定义，当没有部分特化可以使用的时候，就使用通用模板定义。`foo` 的实例化类型与提供的部分特化不匹配，因此，`foo` 的类型必然从通用类模板实例化，将 `int` 绑定到 `T1` 并将 `string` 绑定到 `T2`。部分特化只用于实例化第二个类型为 `int` 的 `some_template` 类型。

The definition of a partial specialization is completely disjointed from the definition of the generic template. The partial specialization may have a completely different set of members from the generic class template. The generic definitions for the members of a class template are never used to instantiate the members of the class template partial specialization.

部分特化的定义与通用模板的定义完全不会冲突。部分特化可以具有与通用类模板完全不同的成员集合。类模板成员的通用定义永远不会用来实例化类模板部分特化的成员。

16.7. Overloading and Function Templates

16.7. 重载与函数模板

A function template can be overloaded: We can define multiple function templates with the same name but differing numbers or types of parameters. We also can define ordinary nontemplate functions with the same name as a function template.

函数模板可以重载：可以定义有相同名字但形参数目或类型不同的多个函数模板，也可以定义与函数模板有相同名字的普通非模板函数。

Of course, declaring a set of overloaded function templates does not guarantee that they can be called successfully. Overloaded function templates may lead to ambiguities.

当然，声明一组重载函数模板不保证可以成功调用它们，重载的函数模板可能会导致二义性。

Function Matching and Function Templates

函数匹配与函数模板

The steps used to resolve a call to an overloaded function in which there are both ordinary functions and function templates are as follows:

如果重载函数中既有普通函数又有函数模板，确定函数调用的步骤如下：

1. Build the set of candidate functions for this function name, including:

为这个函数名建立候选函数集合，包括：

- a. Any ordinary function with the same name as the called function.

与被调用函数名字相同的任意普通函数。

- b. Any function-template instantiation for which template argument deduction finds template arguments that match the function arguments used in the call.

任意函数模板实例化，在其中，模板实参推断发现了与调用中所用函数实参相匹配的模板实参。

2. Determine which, if any, of the ordinary functions are viable ([Section 7.8.2, p. 269](#)). Each template instance in the candidate set is viable, because template argument deduction ensures that the function could be called.

确定哪些普通函数是可行的（[第 7.8.2 节](#)）（如果有可行函数的话）。候选集合中的每个模板实例都可行的，因为模板实参推断保证函数可以被调用。

3. Rank the viable functions by the kinds of conversions, if any, required to make the call, remembering that the conversions allowed to call an instance of a template function are limited.

如果需要转换来进行调用，根据转换的种类排列可靠函数，记住，调用模板函数实例所允许的转换是有限的。

- a. If only one function is selected, call this function.

如果只有一个函数可选，就调用这个函数。

- b. If the call is ambiguous, remove any function template instances from the set of viable functions.

如果调用有二义性，从可行函数集合中去掉所有函数模板实例。

4. Rerank the viable functions excluding the function template instantiations.

重新排列去掉函数模板实例的可行函数。

- If only one function is selected, call this function.

如果只有一个函数可选，就调用这个函数。

- Otherwise, the call is ambiguous.

否则，调用有二义性。

An Example of Function-Template Matching

函数模板匹配的例子

Consider the following set of overloaded ordinary and function templates:

考虑下面普通函数和函数模板的重载集合：

```
// compares two objects
template <typename T> int compare(const T&, const T&);

// compares elements in two sequences
template <class U, class V> int compare(U, U, V);

// plain functions to handle C-style character strings
int compare(const char*, const char*);
```

The overload set contains three functions: The first template handles simple values, the second template compares elements from two sequences, and the third is an ordinary function to handle C-style character strings.

重载集合包含三个函数：第一个模板处理简单值，第二个模板比较两个序列的元素，第三个是处理 C 风格字符串的普通函数。

Resolving Calls to Overloaded Function Templates

确定重载函数模板的调用

We could call these functions on a variety of types:

可以在不同类型上调用这些函数：

```
// calls compare(const T&, const T&) with T bound to int
compare(1, 0);

// calls compare(U, U, V), with U and V bound to vector<int>::iterator
vector<int> ivec1(10), ivec2(20);
compare(ivec1.begin(), ivec1.end(), ivec2.begin());
int ial[] = {0,1,2,3,4,5,6,7,8,9};

// calls compare(U, U, V) with U bound to int*
// and v bound to vector<int>::iterator
compare(ial, ial + 10, ivec1.begin());

// calls the ordinary function taking const char* parameters
const char const_arr1[] = "world", const_arr2[] = "hi";
compare(const_arr1, const_arr2);

// calls the ordinary function taking const char* parameters
char ch_arr1[] = "world", ch_arr2[] = "hi";
compare(ch_arr1, ch_arr2);
```

We'll look at each call in turn:

下面依次介绍每个调用。

compare(1, 0): Both arguments have type `int`. The candidate functions are the first template instantiated with `T` bound to `int` and the ordinary function named `compare`. The ordinary function, however, isn't viable we cannot pass an `int` to a parameter expecting a `char*`. The instantiated function using `int` is an exact match for the call, so it is selected.

两个形参都是 `int` 类型。候选函数是第一个模板将 `T` 绑定到 `int` 的实例化，以及名为 `compare` 的普通函数。但该普通函数不可行——不能将 `int` 对象传给期待 `char*` 对象的形参。用 `int` 实例化的函数与该调用完全匹配，所以选择它。

```
compare(ivec1.begin(), ivec1.end(), ivec2.begin())
```

compare(ial, ial + 10, ivec1.begin()): In these calls, the only viable function is the instantiation of the template that has three parameters. Neither the template with two arguments nor the ordinary nonoverloaded function can match these calls.

这两个调用中，唯一可行的函数是有三个形参的模板的实例化。带两个参数的模板和普通非模板函数都不能匹配这两个调用。

compare(const_arr1, const_arr2): This call, as expected, calls the ordinary function. Both that function and the first template with `T` bound to `const char*` are viable. Both are also exact matches. By rule 3b, we know that the ordinary function is preferred. We eliminate the instance of the template from consideration, leaving just the ordinary function as viable.

这个调用正如我们所期待的，调用普通函数。该函数和将 `T` 绑定到 `const char*` 的第一个模板都是可行的，也都完全匹配。根据规则 3b，会选择普通函数。从候选集合中去掉模板实例，只剩下普通函数可行。

compare(ch_arr1, ch_arr2): This call also is bound to the ordinary function. The candidates are the version of the function template with `T` bound to `char*` and the ordinary function that takes `const char*` arguments. Both functions require a trivial conversion to convert the arrays, `ch_arr1` and

Section 16.7. Overloading and Function Templates

`ch_arr2`, to pointers. Because both functions are equal matches, the plain function is preferred to the template version.

这个调用也绑定到普通函数。候选者是将 `T` 绑定到 `char*` 的函数模板的版本, 以及接受 `const char*` 实参的普通函数, 两个函数都需要稍加转换将数组 `ch_arr1` 和 `ch_arr2` 转换为指针。因为两个函数一样匹配, 所以普通函数优先于模板版本。

Conversions and Overloaded Function Templates

转换与重载的函数模板

It can be difficult to design a set of overloaded functions in which some are templates and others are ordinary functions. Doing so requires deep understanding of the relationships among types and in particular of the implicit conversions that may or may not take place when templates are involved.

设计一组重载函数, 其中一些是模板而另一些是普通函数, 这可能是困难的。这样做需要深入理解类型之间的关系, 具体而言, 就是当涉及模板时可以发生和不能发生的隐式转换。

Let's look at two examples of why it is hard to design overloaded functions that work properly when there are both template and nontemplate versions in the overload set. First, consider a call to `compare` using pointers instead of the arrays themselves:

来看两个例子, 看看为什么当重载集合中既有模板又有非模板版本的时候, 设计正确工作的重载函数是困难的。首先, 考虑用指针代替数组本身的 `compare` 调用:

```
char *p1 = ch_arr1, *p2 = ch_arr2;
compare(p1, p2);
```

This call matches the template version! Ordinarily, we expect to get the same function whether we pass an array or a pointer to an element to that array. In this case, however, the function template is an exact match for the call, binding `char*` to `T`. The plain version still requires a conversion from `char*` to `const char*`, so the function template is preferred.

这个调用与模板版本匹配! 通常, 我们希望无论是传递数组, 还是传递指向该数组元素的指针, 都获得同一函数。但是, 在这个例子中, 将 `char*` 绑定到 `T` 的函数模板与该调用完全匹配。普通版本仍然需要从 `char*` 到 `const char*` 的转换, 所以优先选择函数模板。

Another change that has surprising results is what happens if the template version of `compare` has a parameter of type `T` instead of a `const` reference to `T`:

另一个具有惊人结果的改变是, 如果 `compare` 的模板版本有一个 `T` 类型的形参代替 `T` 的 `const` 引用, 会发生的情况:

```
template <typename T> int compare2(T, T);
```

In this case, if we have an array of plain `char`; then, whether we pass the array itself or a pointer, the template version is called. The only way to call the nontemplate version is when the arguments are arrays of `const char` or pointers to `const char`:

这个例子中, 如果有一个普通类型的数组, 则无论传递数组本身, 还是传递指针, 都将调用模板版本。调用非模板版本的唯一途径是在实参是 `const char` 或者 `const char*` 指针数组的时候:

```
// calls compare(T, T) with T bound to char*
compare(ch_arr1, ch_arr2);
// calls compare(T, T) with T bound to char*
compare(p1, p2);
// calls the ordinary function taking const char*
parameters compare(const_arr1, const_arr2);
const char *cp1 = const_arr1, *cp2 = const_arr2;
// calls the ordinary function taking const char* parameters
compare(cp1, cp2);
```

In these cases, the plain function and the function template are exact matches. As always, when the match is equally good, the nonoverloaded version is preferred.

在这些情况下, 普通函数和函数模板都完全匹配。像通常一样, 当匹配同样好时, 非模板版本优先。



It is hard to design overloaded function sets involving both function templates and nontemplate functions. Because of the likelihood of surprise to users of the functions, it is almost always better to define a function-template specialization ([Section 16.6](#), p. 671) than to use a nontemplate version.

设计既包含函数模板又包含非模板函数的重载函数集合是困难的, 因为可能会使函数的用户感到奇怪, 定义函数模板特化 ([第 16.6 节](#)) 几乎总是比使用非模板版本更好。

Exercises Section 16.7

Exercise 16.61: Implement the three versions of `compare`. Include a print statement in each function that indicates which function is being called. Use these functions to check your answers to the remaining questions.

实现 `compare` 函数的三个版本。在每个函数中包含一个输出语句，指出正在调用哪个函数。使用这些函数检查对其余问题的回答。

Exercise 16.62: Given the `compare` functions and variables defined in this section, explain which function is called, and why, for each of these calls:

对于本节定义的 `compare` 函数和变量，解释下面每个函数调用中，哪个函数被调用以及为什么。

```
compare(ch_arr1, const_arr1);
compare(ch_arr2, const_arr2);
compare(0, 0);
```

Exercise 16.63: For each of the following calls, list the candidate and viable functions. Indicate whether the call is valid and if so which function is called.

对于下面的每个调用，列出候选函数和可行函数，指出调用是否有效，以及如果有效，调用哪个函数。

```
template <class T> T calc(T, T);
double calc(double, double);
template <> char calc<char>(char, char);
int ival; double dval; float fd;
calc(0, ival);           calc(0.25, dval);
calc(0, fd);            calc(0, 'J');
}
```

Chapter Summary

小结

Templates are a distinctive feature of C++ and are fundamental to the library. A template is a type-independent blueprint that the compiler uses to generate a variety of type-specific instances. We write the template once, and the compiler instantiates the template for the type or types with which we use the template. We can write both function templates and class templates.

模板是 C++ 语言与众不同的特性，是标准库的基础。模板是独立于类型的蓝图，编译器可以用它产生多种特定类型实例。我们只需编写一次模板，编译器将为使用模板的不同类型实例化模板。既可以编写函数模板又可以编写类模板。

Function templates are the base on which the algorithms library is built. Class templates are the base on which the library container and iterator types are built.

函数模板是建立算法库的基础，类模板是建立标准库容器和迭代器类型基础。

Compiling templates requires assistance from the programming environment. The language defines two broad strategies for instantiating templates: the inclusion model and the separate compilation model. These models have impacts on how we build our systems in so far as they dictate whether template definitions go in header files or source files. At this time, all compilers implement the inclusion model, while only some implement the separate compilation model. Your compiler's user's guide should specify how your system manages templates.

编译模板需要编程环境的支持。语言为实例化模板定义了两个主要策略：包含模型和分别编译模型。这些模型规定了模板定义应该放在头文件还是源文件中，就此而言，它们影响着构建系统的方式。现在，所有编译器实现了包含模型，只有一些编译器实现了分别编译模型。编译器的用户指南应该会说明系统怎样管理模板。

An explicit template argument lets us fix the type or value of one or more template parameters. Explicit arguments are useful in letting us design functions in which a template type need not be inferred from a corresponding argument and lets us allow conversions on the arguments.

显式模板实参使我们能够固定一个或多个模板形参的类型或值。显式实参使我们能够设计无需从对应实参推断模板类型的函数，也使我们能够对实参进行转换。

A template specialization is a specialized definition that defines a distinct version of the template that binds one or more parameters to specified types or values. Specializations are useful when there are types for which the default template definition does not apply.

模板特化是一种特化的定义，它定义了模板的不同版本，将一个或多个形参绑定到特定类型或特定值。对于默认模板定义不适用的类型，特化非常有用。

Defined Terms

class template (类模板)

A class definition that can be used to define a set of type-specific classes. Class templates are defined using the `template` keyword followed by a comma-separated list of one or more parameters enclosed in `<` and `>` brackets.

可以用来定义一组特定类型的类的类定义。类模板用 `template` 关键字后接用尖括号 (`<>`) 括住、以逗号分隔的一个或多个形参的列表来定义。

export keyword (导出关键字)

Keyword used to indicate that the compiler must remember the location of the associated template definition. Used by compilers that support the separate-compilation model of template instantiation. The `export` keyword ordinarily goes with a function definition; a class is normally declared as `exported` in the associated class implementation file. A template may be defined with the `export` keyword only once in a program.

用来指出编译器必须记住相关模板定义位置的关键字，支持模板实例化的分别编译模型的编译器使用它。`export` 关键字一般与函数定义一起出现，类通常在相关类实现文件中声明为 `export`。在一个程序中，一个模板只能用 `export` 关键字定义一次。

function template (函数模板)

A definition for a function that can be used for a variety of types. A function template is defined using the `template` keyword followed by a comma-separated list of template one or more parameters enclosed in `<` and `>` brackets.

可用于不同类型的函数的定义。函数模板用 `template` 关键字后接用尖括号 (`<>`) 括住、以逗号分隔的一个或多个形参的列表来定义。

generic handle class (泛型句柄类)

A class that holds and manages a pointer to another class. A generic handle takes a single type parameter and allocates and manages a pointer to an object of that type. The handle class defines the necessary copy control members. It also defines the dereference (`*`) and arrow (`->`) member access operators to provide access to the underlying object. A generic handle requires no knowledge of the type it manages.

保存和管理指向其他类的指针的类。泛型句柄接受单个类型形参，并且分配和管理指向该类型对象的指针。句柄类定义了必要的复制控制成员，它还定义了解引用操作符 (`*`) 和箭头成员访问操作符 (`->`)。泛型句柄不需要了解它管理的类型。

inclusion compilation model (包含编译模型)

Mechanism used by compilers to find template definitions that relies on template definitions being *included* in each file that uses the template. Typically, template definitions are stored in a header, and that header must be included in any file that uses the template.

编译器用来寻找模板定义的机制，它依赖于模板定义被包含在每个使用模板的文件中。一般而言，模板定义存储在一个头文件中，使用模板的任意文件必须包含该文件。

instantiation (实例化)

Compiler process whereby the actual template argument(s) are used to generate a specific instance of the template in which the parameter(s) are replaced by the corresponding argument(s). Functions are instantiated automatically based on the arguments used in a call. Template arguments must be provided explicitly whenever a class template is used.

用实际模板实参产生模板特定实例的编译器过程，在该实例中，用对应实参代替形参。函数基于调用中使用的实参自动实例化，使用类模板时必须显式提供模板实参。

member template (成员模板)

A member of a class or class template that is a function template. A member template may not be virtual.

类或类模板的是函数模板的成员。成员模板不能为虚。

nontype parameter (非类型形参)

A template parameter that represents a value. When a function template is instantiated, each nontype parameter is bound to a constant expression as indicated by the arguments used in the call. When a class template is instantiated, each nontype parameter is bound to a constant expression as indicated by the arguments used in the class instantiation.

表示值的模板形参。在实例化函数模板的时候，将每个非类型形参绑定到一个常量表达式，该常量表达式作为调用中使用的实参给出。在实例化类模板的时候，将每个非类型形参绑定到一个常量表达式，该常量表达式作为类实例化中使用的实参给出。

[partial specialization \(部分特化\)](#)

A version of a class template in which some but not all of the template parameters are specified.

类模板的一个版本，其中指定了某些但非全部的模板形参。

[separate compilation model \(分别编译模型\)](#)

Mechanism used by compilers to find template definitions that allows template definitions and declarations to be stored in independent files. Template declarations are placed in a header, and the definition appears only once in the program, typically in a source file. The compiler implements whatever programming environment support is necessary to find that source file and instantiate the versions of the template used by the program.

编译器用来查找模板定义的机制，它允许将模板定义和声明存储在独立的文件中。模板声明放在一个头文件中，而定义只在程序中出现一次，一般在源文件中，无论什么编程环境所支持的编译器都必须找到该源文件，并实例化程序所使用的模板版本。

[template argument \(模板实参\)](#)

Type or value specified when using a template type, such as when defining an object or naming a type in a cast.

在使用模板类型（如定义对象或强制类型转换中指定类型）的时候，指定的类型或值。

[template argument deduction \(模板实参推断\)](#)

Process by which the compiler determines which function template to instantiate. The compiler examines the types of the arguments that were specified using a template parameter. It automatically instantiates a version of the function with those types or values bound to the template parameters.

编译器用来确定实例化哪个函数模板的过程。编译器检查用模板形参指定的实参的类型，它用绑定到模板形参的类型或值自动实例化函数的一个版本。

[template parameter \(模板形参\)](#)

A name specified in the template parameter list that may be used inside the definition of a template. Template parameters can be type or non-type parameters. To use a class template, actual arguments must be specified for each template parameter. The compiler uses those types or values to instantiate a version of the class in which uses of the parameter(s) are replaced by the actual argument(s). When a function template is used, the compiler deduces the template arguments from the arguments in the call and instantiates a specific function using the deduced template arguments.

在模板形参表中指定的、可以在模板定义内部使用的名字。模板形参可以是类型形参或非类型形参。要使用模板，必须为每个模板形参指定实参，编译器使用这些类型或值来实例化类的一个版本，其中形参的使用以实参代替。当使用函数模板的时候，编译器从函数调用中的实参推断模板实参，并使用推断得到的模板实参实例化特定函数。

[template parameter list \(模板形参表\)](#)

List of type or nontype parameters (separated by commas) to be used in the definition or declaration of a template.

在模板定义或声明中使用的类型形参或非类型形参（以逗号分隔）的列表。

[template specialization \(模板特化\)](#)

Redefinition of a class template or a member of a class template in which the template parameters are specified. A template specialization may not appear until after the class that it specializes has been defined. A template specialization must appear before any use of the template for the specialized arguments is used.

类模板或类模板的成员的重定义，其中指定了模板形参。在定义了被特化的模板之前，不能出现该模板的特化。在使用任意实参特化的模板之前，必须先出现模板特化。

[type parameter \(类型形参\)](#)

Name used in a template parameter list to represent a type. When the template is instantiated, each type parameter is bound to an actual type. In a function template, the types are inferred from the argument types or are explicitly specified in the call. Type arguments must be specified for a class template when the class is used.

模板形参表中使用的表示类型的名字。当实例化模板的时候，将每个类型形参绑定到实际类型。在函数模板中，从实参类型中推断类型或者在调用中显式指定类型。类模板的类型实参必须在使用类的时候指定。

Chapter 17. Tools for Large Programs

第十七章 用于大型程序的工具

CONTENTS

Section 17.1 Exception Handling	688
Section 17.2 Namespaces	712
Section 17.3 Multiple and Virtual Inheritance	731
Chapter Summary	748
Defined Terms	748

C++ is used on problems that have a wide range in complexity. It is used on problems small enough to be solved by a single programmer after a few hours' work to problems requiring enormous systems consisting of tens of millions of lines of code developed and modified over many years. The facilities we covered in the earlier parts of this book are equally useful across this range of programming problems.

用 C++ 解决的问题，其复杂性千变万化，问题一个程序员几小时便可解决，有的则需要用许多年开发和修改的数千万行代码构成的庞大系统。本书前面所介绍的内容对这些编程问题都同样适用。

The language includes some features that are most useful on systems once problems get to be more complex than those that an individual can manage. These features exception handling, namespaces, and multiple inheritance are the topic of this chapter.

C++ 语言包含的一些特征在问题比较复杂，非个人所能管理时最为有用。本章的主题就是这些特征，即异常处理、命名空间和多重继承。

Large-scale programming places greater demands on programming languages than do the needs of systems that can be developed by small teams of programmers. Among the needs that distinguish large-scale applications are:

相对于小的程序员团队所能开发的系统需求而言，大规模编程对程序设计语言的要求更高。大规模应用程序往往具有下列特殊要求：

1. Stricter up-time requirements and the need for more robust error detection and error handling. Error handling often must span independently developed subsystems.

更严格的正常运转时间以及更健壮的错误检测和错误处理。错误处理经常必须跨越独立开发的多个子系统进行。

2. The ability to structure programs that are composed of libraries developed more or less independently.

能够用各种库（可能包含独立开发的库）构造程序。

3. The need to deal with more complicated application concepts.

能够处理更复杂的应用概念。

Three features in C++ are aimed at these needs: exception handling, namespaces, and multiple inheritance. This chapter looks at these three facilities.

C++ 中有下列三个特征分别针对这些要求：异常处理、命名空间和多重继承。本章将这三个特征。

17.1. Exception Handling

17.1. 异常处理

Exception handling allows independently developed parts of a program to communicate about and handle problems that arise during execution of the program. One part of the program can detect a problem that that part of the program cannot resolve. The problem-detecting part can pass the problem along to another part that is prepared to handle what went wrong.

使用异常处理，程序中独立开发的各部分能够就程序执行期间出现的问题相互通信，并处理这些问题。程序的一个部分能够检测出本部分无法解决的问题，这个问题检测部分可以将问题传递给准备处理问题的其他部分。



Exceptions let us separate problem detection from problem resolution. The part of the program that detects a problem need not know how to deal with it.

通过异常我们能够将问题的检测和问题的解决分离，这样程序的问题检测部分可以不必了解如何处理问题。

In C++, exception handling relies on the problem-detecting part throwing an object to a handler. The type and contents of that object allow the two parts to communicate about what went wrong.

C++ 的异常处理中，需要由问题检测部分抛出一个对象给处理代码，通过这个对象的类型和内容，两个部分能够就出现了什么错误进行通信。

[Section 6.13](#) (p. 215) introduced the basic concepts and mechanics of using exceptions in C++. In that section, we hypothesized that a more complex bookstore application might use exceptions to communicate about problems. For example, the `Sales_item` addition operator might throw an exception if the `isbn` members of its operands didn't match:

[第 6.13 节](#)介绍了在 C++ 中使用异常的基本概念和基本机制，在该本节中，假定更复杂的书店应用程序可以通过异常就出现的问题进行通信。例如，如果操作数的 `isbn` 成员不匹配，`Sales_item` 的加操作符可以抛出一个异常：

[View full width]

```
// throws exception if both objects do not refer to the same isbn
Sales_item
operator+(const Sales_item& lhs, const Sales_item& rhs)
{
    if (!lhs.same_isbn(rhs))
        throw runtime_error("Data must refer to same ISBN");
    // ok, if we're still here the ISBNs are the same so it's okay to do the addition
    Sales_item ret(lhs);           // copy lhs into a local object that we'll
→   return
        ret += rhs;                // add in the contents of rhs
        return ret;                // return a copy of ret
}
```

Those parts of the program that added `Sales_item` objects would use a `try` block in order to catch an exception if one occurred:

程序中将 `Sales_item` 对象相加的那些部分可以使用一个 `try` 块，以便在异常发生时捕获异常：

```
// part of the application that interacts with the user
Sales_item item1, item2, sum;
while (cin >> item1 >> item2) {          // read two transactions
    try {
        sum = item1 + item2;           // calculate their sum
        // use sum
    } catch (const runtime_error &e) {
        cerr << e.what() << " Try again.\n"
            << endl;
    }
}
```

Section 17.1. Exception Handling

In this section we'll expand our coverage of these basics and cover some additional exception-handling facilities. Effective use of exception handling requires understanding what happens when an exception is thrown, what happens when it is caught, and the meanings of the objects used to communicate what went wrong.

本节将扩展对这些基础知识的讨论并涵盖另外一些异常处理设施。有效使用异常处理需要理解：在抛出异常时会发生什么，在捕获异常时又会发生什么，还有用来传递错误的对象的含义。

17.1.1. Throwing an Exception of Class Type

17.1.1.1. 抛出类类型的异常

An exception is **raised** by **throwing** an object. The type of that object determines which handler will be invoked. The selected handler is the one nearest in the call chain that matches the type of the object.

异常是通过**抛出**对象而**引发**的。该对象的类型决定应该激活哪个处理代码。被选中的处理代码是调用链中与该对象类型匹配且离抛出异常位置最近的那个。

Exceptions are thrown and caught in ways that are similar to how arguments are passed to functions. An exception can be an object of any type that can be passed to a nonreference parameter, meaning that it must be possible to copy objects of that type.

异常以类似于将实参传递给函数的方式抛出和捕获。异常可以是可传给非引用形参的任意类型的对象，这意味着必须能够复制该类型的对象。

Recall that when we pass an argument of array or function type, that argument is automatically converted to a pointer. The same automatic conversion happens for objects that are thrown. As a consequence, there are no exceptions of array or function types. Instead, if we **throw** an array, the thrown object is converted to a pointer to the first element in the array. Similarly, if we throw a function, the function is converted to a pointer to the function ([Section 7.9](#), p. 276).

回忆一下，传递数组或函数类型实参的时候，该实参自动转换为一个指针。被抛出的对象将发生同样的自动转换，因此，不存在数组或函数类型的异常。相反，如果抛出一个数组，被抛出的对象转换为指向数组首元素的指针，类似地，如果抛出一个函数，函数被转换为指向该函数的指针[第 7.9 节](#)。

When a **throw** is executed, the statement(s) following the **throw** are not executed. Instead, control is transferred from the **throw** to the matching **catch**. That **catch** might be local to the same function or might be in a function that directly or indirectly called the function in which the exception occurred. The fact that control passes from one location to another has two important implications:

执行 **throw** 的时候，不会执行跟在 **throw** 后面的语句，而是将控制从 **throw** 转移到匹配的 **catch**，该 **catch** 可以是同一函数中局部的 **catch**，也可以在直接或间接调用发生异常的函数的另一个函数中。控制从一个地方传到另一地方，这有两个重要含义：

1. Functions along the call chain are prematurely exited. [Section 17.1.2](#) (p. 691) discusses what happens when functions are exited due to an exception.

沿着调用链的函数提早退出。[第 17.1.2 节](#)将讨论函数因异常而退出时会发生什么。

2. In general, the storage that is local to a block that throws an exception is not around when the exception is handled.

一般而言，在处理异常的时候，抛出异常的块中的局部存储不存在了。

Because local storage is freed while handling an exception, the object that is thrown is not stored locally. Instead, the **throw** expression is used to initialize a special object referred to as the **exception object**. The exception object is managed by the compiler and is guaranteed to reside in space that will be accessible to whatever **catch** is invoked. This object is created by a **throw**, and is initialized as a copy of the expression that is thrown. The exception object is passed to the corresponding **catch** and is destroyed after the exception is completely handled.

因为在处理异常的时候会释放局部存储，所以被抛出的对象就不能再局部存储，而是用 **throw** 表达式初始化一个称为**异常对象**的特殊对象。异常对象由编译器管理，而且保证驻留在可能被激活的任意 **catch** 都可以访问的空间。这个对象由 **throw** 创建，并被初始化为被抛出的表达式的副本。异常对象将传给对应的 **catch**，并且在完全处理了异常之后撤销。



The exception object is created by copying the result of the thrown expression; that result must be of a type that can be copied.

异常对象通过复制被抛出表达式的结果创建，该结果必须是可以复制的类型。

异常对象与继承

In practice, many applications throw expressions whose type comes from an inheritance hierarchy. As we'll see in [Section 17.1.7](#) (p. 697), the standard exceptions ([Section 6.13](#), p. 215) are defined in an inheritance hierarchy. What's important to know at this point is how the form of the `throw` expression interacts with types related by inheritance.

在实践中，许多应用程序所抛出的表达式，基类型都来自某个继承层次。正如[第 17.1.7 节](#)将介绍的，标准异常（[第 6.13 节](#)）定义在一个继承层次中。目前重要的是，知道 `throw` 表达式的形式如何与因继承而相关的类型相互影响。



When an exception is thrown, the static, compile-time type of the thrown object determines the type of the exception object.

当抛出一个表达式的时候，被抛出对象的静态编译时类型将决定异常对象的类型。

Ordinarily, the fact that the object is thrown using its static type is not an issue. When we throw an exception, we usually construct the object we are going to throw at the throw point. That object represents what went wrong, so we know the precise exception type.

通常，使用静态类型抛出对象不成问题。当抛出一个异常的时候，通常在抛出点构造将抛出的对象，该对象表示出了什么问题，所以我们知道确切的异常类型。

Exceptions and Pointers

异常与指针

The one case where it matters that a `throw` expression throws the static type is if we dereference a pointer in a `throw`. The result of dereferencing a pointer is an object whose type matches the type of the pointer. If the pointer points to a type from an inheritance hierarchy, it is possible that the type of the object to which the pointer points is different from the type of the pointer. Regardless of the object's actual type, the type of the exception object matches the static type of the pointer. If that pointer is a base-class type pointer that points to a derived-type object, then that object is sliced down ([Section 15.3.1](#), p. 577); only the base-class part is thrown.

用抛出表达式抛出静态类型时，比较麻烦的一种情况是，在抛出中对指针解引用。对指针解引用的结果是一个对象，其类型与指针的类型匹配。如果指针指向继承层次中的一种类型，指针所指对象的类型就有可能与指针的类型不同。无论对象的实际类型是什么，异常对象的类型都与指针的静态类型相匹配。如果该指针是一个指向派生类对象的基类类型指针，则那个对象将被分割（[第 15.3.1 节](#)），只抛出基类部分。

A problem more serious than slicing the object may arise if we `throw` the pointer itself. In particular, it is always an error to `throw` a pointer to a local object for the same reasons as it is an error to return a pointer to a local object ([Section 7.3.2](#), p. 249) from a function. When we `throw` a pointer, we must be certain that the object to which the pointer points will exist when the handler is entered.

如果抛出指针本身，可能会引发比分割对象更严重的问题。具体而言，抛出指向局部对象的指针总是错误的，其理由与从函数返回指向局部对象的指针是错误的一样（[第 7.3.2 节](#)）抛出指针的时候，必须确定进入处理代码时指针所指向的对象存在。

If we `throw` a pointer to a local object and the handler is in another function, then the object to which the pointer points will no longer exist when the handler is executed. Even if the handler is in the same function, we must be sure that the object to which the pointer points exists at the site of the `catch`. If the pointer points to an object in a block that is exited before the `catch`, then that local object will have been destroyed before the `catch`.

如果抛出指向局部对象的指针，而且处理代码在另一函数中，则执行处理代码时指针所指向的对象将不再存在。即使处理代码在同一函数中，也必须确信指针所指向的对象在 `catch` 处存在。如果指针指向某个在 `catch` 之前退出的块中的对象，那么，将在 `catch` 之前撤销该局部对象。



It is usually a bad idea to `throw` a pointer: Throwing a pointer requires that the object to which the pointer points exist wherever the corresponding handler resides.

抛出指针通常是个坏主意：抛出指针要求在对应处理代码存在的任意地方存在指针所指向的对象。

Exercises Section 17.1.1

Exercise What is the type of the exception object in the following `throws`:

17.1:

下面的 `throw` 语句中，异常对象的类型是什么？

```
(a) range_error r("error"); (b) exception *p = &r;
    throw r;           throw *p;
```

Exercise What would happen if the second `throw` were written as `throw p`?

17.2:

如果第二个 `throw` 语句写成 `throw p`，会发生什么情况？

17.1.2. Stack Unwinding

17.1.2. 栈展开

When an exception is thrown, execution of the current function is suspended and the search begins for a matching `catch` clause. The search starts by checking whether the `throw` itself is located inside a `try` block. If so, the `catch` clauses associated with that `try` are examined to see if one of them matches the thrown object. If a matching `catch` is found, the exception is handled. If no `catch` is found, the current function is exited its memory is freed and local objects are destroyed and the search continues in the calling function.

抛出异常的时候，将暂停当前函数的执行，开始查找匹配的 `catch` 子句。首先检查 `throw` 本身是否在 `try` 块内部，如果是，检查与该 `catch` 相关的 `catch` 子句，看是否其中之一与抛出对象相匹配。如果找到匹配的 `catch`，就处理异常；如果找不到，就退出当前函数（释放当前函数的内存并撤销局部对象），并且继续在调用函数中查找。

If the call to the function that threw is in a `try` block, then the `catch` clauses associated with that `try` are examined. If a matching `catch` is found, the exception is handled. If no matching `catch` is found, the calling function is also exited, and the search continues in the function that called this one.

如果对抛出异常的函数的调用是在 `try` 块中，则检查与该 `try` 相关的 `catch` 子句。如果找到匹配的 `catch`，就处理异常；如果找不到匹配的 `catch`，调用函数也退出，并且继续在调用这个函数的函数中查找。

This process, known as **stack unwinding**, continues up the chain of nested function calls until a `catch` clause for the exception is found. As soon as a `catch` clause that can handle the exception is found, that `catch` is entered, and execution continues within this handler. When the `catch` completes, execution continues at the point immediately after the last `catch` clause associated with that `try` block.

这个过程，称之为**栈展开 (stack unwinding)**，沿嵌套函数调用链继续向上，直到为异常找到一个 `catch` 子句。只要找到能够处理异常的 `catch` 子句，就进入该 `catch` 子句，并在该处理代码中继续执行。当 `catch` 结束的时候，在紧接着与该 `try` 块相关的最后一个 `catch` 子句之后的点继续执行。

Destructors Are Called for Local Objects

为局部对象调用析构函数

During stack unwinding, the function containing the `throw`, and possibly other functions in the call chain, are exited prematurely. In general, these functions will have created local objects that ordinarily would be destroyed when the function exited. When a function is exited due to an exception, the compiler guarantees that the local objects are properly destroyed. As each function exits, its local storage is freed. Before releasing the memory, any local object that was created before the exception occurred is destroyed. If the local object is of class type, the destructor for this object is called automatically. As usual, the compiler does no work to destroy an object of built-in type.

栈展开期间，提早退出包含 `throw` 的函数和调用链中可能的其他函数。一般而言，这些函数已经创建了可以在退出函数时撤销的局部对象。因异常而退出函数时，编译器保证适当地撤销局部对象。每个函数退出的时候，它的局部存储都被释放，在释放内存之前，撤销在异常发生之前创建的所有对象。如果局部对象是类类型的，就自动调用该对象的析构函数。通常，编译器不撤销内置类型的对象。

During stack unwinding, the memory used by local objects is freed and destructors for local objects of class type are run.

栈展开期间，释放局部对象所用的内存并运行类类型局部对象的析构函数。



If a block directly allocates a resource, and the exception occurs before that resource is freed, that resource will not be freed during stack unwinding. For example, a block might dynamically allocate memory through a call to `new`. If the block exits due to an exception, the compiler does not `delete` the pointer. The allocated memory will not be freed.

如果一个块直接分配资源，而且在释放资源之前发生异常，在栈展开期间将不会释放该资源。例如，一个块可以通过调用 `new` 动态分配内存，如果该块因异常而退出，编译器不会删除该指针，已分配的内存将不会释放。

Resources allocated by an object of class type generally will be properly freed. Destructors for local objects are run; resources allocated by class-type objects ordinarily are freed by their destructor. [Section 17.1.8 \(p. 700\)](#) describes a programming technique that uses classes to manage resource allocation in the face of exceptions.

由类类型对象分配的资源一般会被适当地释放。运行局部对象的析构函数，由类类型对象分配的资源通常由它们的析构函数释放。[第 17.1.8 节](#)说明面对异常使用类管理资源分配的编程技术。

Destructors Should Never `throw` Exceptions

析构函数应该从不抛出异常

Destructors are often executed during stack unwinding. When destructors are executing, the exception has been raised but not yet handled. It is unclear what should happen if a destructor itself throws a new exception during this process. Should the new exception supersede the earlier exception that has not yet been handled? Should the exception in the destructor be ignored?

栈展开期间会经常执行析构函数。在执行析构函数的时候，已经引发了异常但还没有处理它。如果在这个过程中析构函数本身抛出新的异常，又会发生什么呢？新的异常应该取代仍未处理的早先的异常吗？应该忽略析构函数中的异常吗？

The answer is that while stack unwinding is in progress for an exception, a destructor that throws another exception of its own that it does not also handle, causes the library `terminate` function is called. Ordinarily, `terminate` calls `abort`, forcing an abnormal exit from the entire program.

答案是：在为某个异常进行栈展开的时候，析构函数如果又抛出自己的未经处理的另一个异常，将会导致调用标准库 `terminate` 函数。一般而言，`terminate` 函数将调用 `abort` 函数，强制从整个程序非正常退出。

Because `terminate` ends the program, it is usually a very bad idea for a destructor to do anything that might cause an exception. In practice, because destructors free resources, it is unlikely that they throw exceptions. The standard library types all guarantee that their destructors will not raise an exception.

因为 `terminate` 函数结束程序，所以析构函数做任何可能导致异常的事情通常都是非常糟糕的主意。在实践中，因为析构函数释放资源，所以它不太可能抛出异常。标准库类型都保证它们的析构函数不会引发异常。

Exceptions and Constructors

异常与构造函数

Unlike destructors, it is often the case that something done inside a constructor might throw an exception. If an exception occurs while constructing an object, then the object might be only partially constructed. Some of its members might have been initialized, and others might not have been initialized before the exception occurs. Even if the object is only partially constructed, we are guaranteed that the constructed members will be properly destroyed.

与析构函数不同，构造函数内部做的事情经常会抛出异常。如果在构造函数对象的时候发生异常，则该对象可能只是部分被构造，它的一些成员可能已经初始化，而另一些成员在异常发生之前还没有初始化。即使对象只是部分被构造了，也要保证将会适当地撤销已构造的成员。

Similarly, an exception might occur when initializing the elements of an array or other container type. Again, we are guaranteed that the constructed elements will be destroyed.

类似地，在初始化数组或其他容器类型的元素的时候，也可能发生异常，同样，也要保证将会适当地撤销已构造的元素。

Uncaught Exceptions Terminate the Program

未捕获的异常终止程序

An exception cannot remain unhandled. An exception is an important enough event that the program cannot continue executing normally. If no matching `catch` is found, then the program calls the library `terminate` function.

不能不处理异常。异常是足够重要的、使程序不能继续正常执行的事件。如果找不到匹配的 `catch`，程序就调用库函数 `terminate`。

17.1.3. Catching an Exception

17.1.3. 捕获异常

The exception specifier in a catch clause looks like a parameter list that contains exactly one parameter. The exception specifier is a type name followed by an optional parameter name.

`catch` 子句中的异常说明符看起来像只包含一个形参的形参表，异常说明符是在其后跟一个（可选）形参名的类型名。

The type of the specifier determines what kinds of exceptions the handler can `catch`. The type must be a complete type: It must either be a built-in type or a programmer-defined type that has already been defined. A forward declaration for the type is not sufficient.

说明符的类型决定了处理代码能够捕获的异常种类。类型必须是完全类型，即必须是内置类型或者是已经定义的程序员自定义类型。类型的前向声明不行。

An exception specifier can omit the parameter name when a `catch` needs to know only the type of the exception in order to handle it. If the handler needs information beyond what type of exception occurred, then its exception specifier will include a parameter name. The `catch` uses the name to get access to the exception object.

当 `catch` 为了处理异常只需要了解异常的类型的时候，异常说明符可以省略形参名；如果处理代码需要已发生异常的类型之外的信息，则异常说明符就包含形参名，`catch` 使用这个名字访问异常对象。

Finding a Matching Handler

查找匹配的处理代码

During the search for a matching `catch`, the `catch` that is found is not necessarily the one that matches the exception best. Instead, the `catch` that is selected is the first `catch` found that can handle the exception. As a consequence, in a list of `catch` clauses, the most specialized `catch` must appear first.

在查找匹配的 `catch` 期间，找到的 `catch` 不必是与异常最匹配的那个 `catch`，相反，将选中第一个找到的可以处理该异常的 `catch`。因此，在 `catch` 子句列表中，最特殊的 `catch` 必须最先出现。

The rules for when an exception matches a `catch` exception specifier are much more restrictive than the rules used for matching arguments with parameter types. Most conversions are not allowed—the types of the exception and the `catch` specifier must match exactly with only a few possible differences:

异常与 `catch` 异常说明符匹配的规则比匹配实参和形参类型的规则更严格，大多数转换都不允许——除下面几种可能的区别之外，异常的类型与 `catch` 说明符的类型必须完全匹配：

- Conversions from `nonconst` to `const` are allowed. That is, a `throw` of a `nonconst` object can match a `catch` specified to take a `const` reference.

允许从非 `const` 到 `const` 的转换。也就是说，非 `const` 对象的 `throw` 可以与指定接受 `const` 引用的 `catch` 匹配。

- Conversions from derived type to base type are allowed.

允许从派生类型型到基类类型的转换。

- An array is converted to a pointer to the type of the array; a function is converted to the appropriate pointer to function type.

将数组转换为指向数组类型的指针，将函数转换为指向函数类型的适当指针。

No other conversions are allowed when looking for a matching `catch`. In particular, neither the standard arithmetic conversions nor conversions defined for class types are permitted.

在查找匹配 `catch` 的时候，不允许其他转换。具体而言，既不允许标准算术转换，也不允许为类类型定义的转换。

Exception Specifiers

异常说明符

When a `catch` is entered, the `catch` parameter is initialized from the exception object. As with a function parameter, the exception-specifier type might be a reference. The exception object itself is a copy of the object that was thrown. Whether the exception object is copied again into the `catch` site depends on the exception-specifier type.

进入 `catch` 的时候, 用异常对象初始化 `catch` 的形参。像函数形参一样, 异常说明符类型可以是引用。异常对象本身是被抛出对象的副本。是否再次将异常对象复制到 `catch` 位置取决于异常说明符类型。

If the specifier is not a reference, then the exception object is copied into the `catch` parameter. The `catch` operates on a local copy of the exception object. Any changes made to the `catch` parameter are made to the copy, not to the exception object itself. If the specifier is a reference, then like a reference parameter, there is no separate `catch` object; the `catch` parameter is just another name for the exception object. Changes made to the `catch` parameter are made to the exception object.

如果说明符不是引用, 就将异常对象复制到 `catch` 形参中, `catch` 操作异常对象的不可一世, 对形参所做的任何改变都只作用于副本, 不会作用于异常对象本身。如果说明符是引用, 则像引用形参一样, 不存在单独的 `catch` 对象, `catch` 形参只是异常对象的另一名字。对 `catch` 形参所做的改变作用于异常对象。

Exception Specifiers and Inheritance

异常说明符与继承

Like a parameter declaration, an exception specifier for a base class can be used to `catch` an exception object of a derived type. Again, like a parameter declaration, the static type of the exception specifier determines the actions that the `catch` clause may perform. If the exception object thrown is of derived-class type but is handled by a `catch` that takes a base-class type, then the `catch` cannot use any members that are unique to the derived type.

像形参声明一样, 基类的异常说明符可以用于捕获派生类型的异常对象, 而且, 异常说明符的静态类型决定 `catch` 子句可以执行的动作。如果被抛出的异常对象是派生类类型的, 但由接受基类类型的 `catch` 处理, 那么, `catch` 不能使用派生类特有的任何成员。



Usually, a `catch` clause that handles an exception of a type related by inheritance ought to define its parameter as a reference.

通常, 如果 `catch` 子句处理因继承而相关的类型的异常, 它就应该将自己的形参定义为引用。

If the `catch` parameter is a reference type, then the `catch` object accesses the exception object directly. The static type of the `catch` object and the dynamic type of the exception object to which it refers might differ. If the specifier is not a reference, then the `catch` object is a copy of the exception object. If the `catch` object is an object of the base type and the exception object has derived type, then the exception object is sliced down ([Section 15.3.1](#), p. 577) to its base-class subobject.

如果 `catch` 形参是引用类型, `catch` 对象就直接访问异常对象, `catch` 对象的静态类型可以与 `catch` 对象所引用的异常对象的动态类型不同。如果异常说明符不是引用, 则 `catch` 对象是异常对象的副本, 如果 `catch` 对象是基类类型对象而异常对象是派生类型的, 就将异常对象分割 ([第 15.3.1 节](#)) 为它的基类子对象。

Moreover, as we saw in [Section 15.2.4](#) (p. 566), objects (as opposed to references) are not polymorphic. When we use a virtual function on an object rather than through a reference, the object's static and dynamic type are the same; the fact that the function is virtual makes no difference. Dynamic binding happens only for calls through a reference or pointer, not calls on an object.

而且, 正如[第 15.2.4 节](#)所介绍的, 对象 (相对于引用) 不是多态的。当通过对象而不是引用使用虚函数的时候, 对象的静态类型和动态类型相同, 函数是虚函数也一样。只有通过引用或指针调用时才发生动态绑定, 通过对象调用不进行动态绑定。

Ordering of Catch Clauses Must Reflect Type Hierarchy

`catch` 子句的次序必须反映类型层次

Section 17.1. Exception Handling

When exception types are organized in class hierarchies, users may choose the level of granularity with which their applications will deal with an exception. For example, an application that merely wants to do cleanup and exit might define a single `try block` that encloses the code in `main` with a `catch` such as the following:

```
catch(exception &e) {
    // do cleanup
    // print a message
    cerr << "Exiting: " << e.what() << endl;
    size_t status_indicator = 42; // set and return an
    return(status_indicator); // error indicator
}
```

Other programs with more rigorous uptime requirements might need finer control over exceptions. Such applications will clear whatever caused the exception and continue processing.

有更严格实时需求的程序可能需要更好的异常控制，这样的应用程序将清除导致异常的一切并继续执行。

Because `catch` clauses are matched in the order in which they appear, programs that use exceptions from an inheritance hierarchy must order their `catch` clauses so that handlers for a derived type occurs before a `catch` for its base type.

因为 `catch` 子句按出现次序匹配，所以使用来自继承层次的异常的程序必须将它们的 `catch` 子句排序，以便 派生类型的处理代码出现在其基类类型的 `catch` 之前。



Multiple `catch` clauses with types related by inheritance must be ordered from most derived type to least derived.

带有因继承而相关的类型的多个 `catch` 子句，必须从最低派生类型到最高派生类型排序。

Exercises Section 17.1.3

Exercise Explain why this `try` block is incorrect. Correct it.

17.3: 解释下面这个 `try` 块为什么不正确，并改正它。

```
try {
    // use of the C++ standard library
} catch(exception) {
    // ...
} catch(const runtime_error &re) {
    // ...
} catch(overflow_error eobj) { /* ... */ }
```

17.1.4. Rethrow

17.1.4. 重新抛出

It is possible that a single `catch` cannot completely handle an exception. After some corrective actions, a `catch` may decide that the exception must be handled by a function further up the chain of function calls. A `catch` can pass the exception out to another `catch` further up the list of function calls by [rethrowing](#) the exception. A `rethrow` is a `throw` that is not followed by a type or an expression:

有可能单个 `catch` 不能完全处理一个异常。在进行了一些校正行动之后，`catch` 可能确定该异常必须由函数调用链中更上层的函数来处理，`catch` 可以通过[重新抛出](#)将异常传递函数调用链中更上层的函数。重新抛出是后面不跟类型或表达式的一个 `throw`:

Section 17.1. Exception Handling

```
throw;
```

An empty `throw` rethrows the exception object. An empty `throw` can appear only in a `catch` or in a function called (directly or indirectly) from a `catch`. If an empty `throw` is encountered when a handler is not active, `terminate` is called.

空 `throw` 语句将重新抛出异常对象，它只能出现在 `catch` 或者从 `catch` 调用的函数中。如果在处理代码不活动时碰到空 `throw`，就调用 `terminate` 函数。

Although the rethrow does not specify its own exception, an exception object is still passed up the chain. The exception that is thrown is the original exception object, not the `catch` parameter. When a `catch` parameter is a base type, then we cannot know the actual type thrown by a rethrow expression. That type depends on the dynamic type of the exception object, not the static type of the `catch` parameter. For example, a rethrow from a `catch` with a parameter of base type might actually `throw` an object of the derived type.

虽然重新抛出不指定自己的异常，但仍然将一个异常对象沿链向上传递，被抛出的异常是原来的异常对象，而不是 `catch` 形参。当 `catch` 形参是基类类型的时候，我们不知道由重新抛出表达式抛出的实际类型，该类型取决于异常对象的动态类型，而不是 `catch` 形参的静态类型。例如，来自带基类类型形参 `catch` 的重新抛出，可能实际抛出一个派生类型的对象。

In general, a `catch` might change its parameter. If, after changing its parameter, the `catch` rethrows the exception, then those changes will be propagated only if the exception specifier is a reference:

一般而言，`catch` 可以改变它的形参。在改变它的形参之后，如果 `catch` 重新抛出异常，那么，只有当异常说明符是引用的时候，才会传播那些改变。

```
catch (my_error &eObj) {           // specifier is a reference type
    eObj.status = severeErr;        // modifies the exception object
    throw; // the status member of the exception object is severeErr
} catch (other_error eObj) {         // specifier is a nonreference type
    eObj.status = badErr;           // modifies local copy only
    throw; // the status member of the exception rethrown is unchanged
}
```

17.1.5. The Catch-All Handler

17.1.5. 捕获所有异常的处理代码

A function may want to perform some action before it exits with a thrown exception, even though it cannot handle the exception that is thrown. Rather than provide a specific `catch` clause for every possible exception, and because we can't know all the exceptions that might be thrown, we can use a `catch-all` `catch` clause. A catch-all clause has the form `(...)`. For example:

即使函数不能处理被抛出的异常，它也可能想要在随抛出异常退出之前执行一些动作。除了为每个可能的异常提供特定 `catch` 子句之外，因为不可能知道可能被抛出的所有异常，所以可以使用捕获所有异常 `catch` 子句的。捕获所有异常的 `catch` 子句形式为 `(...)`。例如：

```
// matches any exception that might be thrown
catch (...) {
    // place our code here
}
```

A catch-all clause matches any type of exception.

捕获所有异常的 `catch` 子句与任意类型的异常都匹配。

A `catch(...)` is often used in combination with a rethrow expression. The `catch` does whatever local work can be done and then rethrows the exception:

`catch(...)` 经常与重新抛出表达式结合使用，`catch` 完成可做的所有局部工作，然后重新抛出异常：

```
void manip() {
    try {
        // actions that cause an exception to be thrown
    } catch (...) {
        // work to partially handle the exception
        throw;
    }
}
```

A `catch(...)` clause can be used by itself or as one of several `catch` clauses.

`catch(...)` 子句可以单独使用，也可以用在几个 `catch` 子句中间。



If a `catch(...)` is used in combination with other `catch` clauses, it must be last; otherwise, any `catch` clause that followed it could never be matched.

如果 `catch(...)` 与其他 `catch` 子句结合使用，它必须是最后一个，否则，任何跟在它后面的 `catch` 子句都将不能被匹配。

17.1.6. Function Try Blocks and Constructors

17.1.6. 函数测试块与构造函数

In general, exceptions can occur at any point in the program's execution. In particular, an exception might occur in a constructor, or while processing a constructor initializer. Constructor initializers are processed before the constructor body is entered. A `catch` clause inside the constructor body cannot handle an exception that might occur while processing a constructor initializer.

一般而言，异常可能发生在程序执行的任何一点。具体而言，异常可能发生在构造函数中，或者发生在处理构造函数初始化式的时候。在进入构造函数函数体之前处理构造函数初始化式，构造函数函数体内部的 `catch` 子句不能处理在处理构造函数初始化时可能发生的异常。

To handle an exception from a constructor initializer, we must write the constructor as a [function try block](#). A function try block lets us associate a group of `catch` clauses with the function as a whole. As an example, we might wrap our `Handle` constructor from [Chapter 16](#) in a try block to detect a failure in `new`:

为了处理来自构造函数初始化式的异常，必须将构造函数编写为[函数 try 块](#)。可以使用函数测试块将一组 `catch` 子句与函数联成一个整体。作为例子，可以将[第十六章](#)的 `Handle` 构造函数包装在一个用来检测 `new` 中失败的测试块当中：

```
template <class T> Handle<T>::Handle(T *p)
try : ptr(p), use(new size_t(1))
{
    // empty function body
} catch(const std::bad_alloc &e)
{ handle_out_of_memory(e); }
```

Exercises Section 17.1.5

Exercise

17.4: Given a basic C++ program,

对于如下的基本 C++ 程序

```
int main() {
    // use of the C++ standard library
}
```

modify `main` to catch any exception thrown by functions in the C++ standard library. The handlers should print the error message associated with the exception before calling `abort` (defined in the header `cstdlib`) to terminate `main`.

修改 `main` 函数以捕获由 C++ 标准库中函数抛出的任何异常。处理代码应该在调用 `abort` 函数（在头文件 `cstdlib` 中定义）终止 `main` 函数之前显示与异常相关的错误信息。

Exercise

17.5: Given the following exception types and `catch` clauses, write a `throw` expression that creates an exception object that could be caught by each `catch` clause.

对于下面的异常类型以及 `catch` 子句，编写一个 `throw` 表达式，该表达式创建一个可被每个 `catch` 子句捕获的异常对象。

- (a) `class exceptionType { };`
`catch(exceptionType *pet) { }`
- (b) `catch(...) { }`
- (c) `enum mathErr { overflow, underflow, zeroDivide };`
`catch(mathErr &ref) { }`
- (d) `typedef int EXCPTYPE;`
`catch(EXCPTYPE) { }`

Notice that the keyword `try` precedes the member initialization list, and the compound statement of the `try` block encompasses the constructor function body. The `catch` clause can handle exceptions thrown either from within the member initialization list or from within the constructor body.

注意，关键字 `try` 出现在成员初始化列表之前，并且测试块的复合语句包围了构造函数的函数体。`catch` 子句既可以处理从成员初始化列表中抛出的异常，也可以处理从构造函数函数体中抛出的异常。



The only way for a constructor to handle an exception from a constructor initializer is to write the constructor as a function `try` block.

构造函数要处理来自构造函数初始化式的异常，唯一的方法是将构造函数编写为函数测试块。

17.1.7. Exception Class Hierarchies

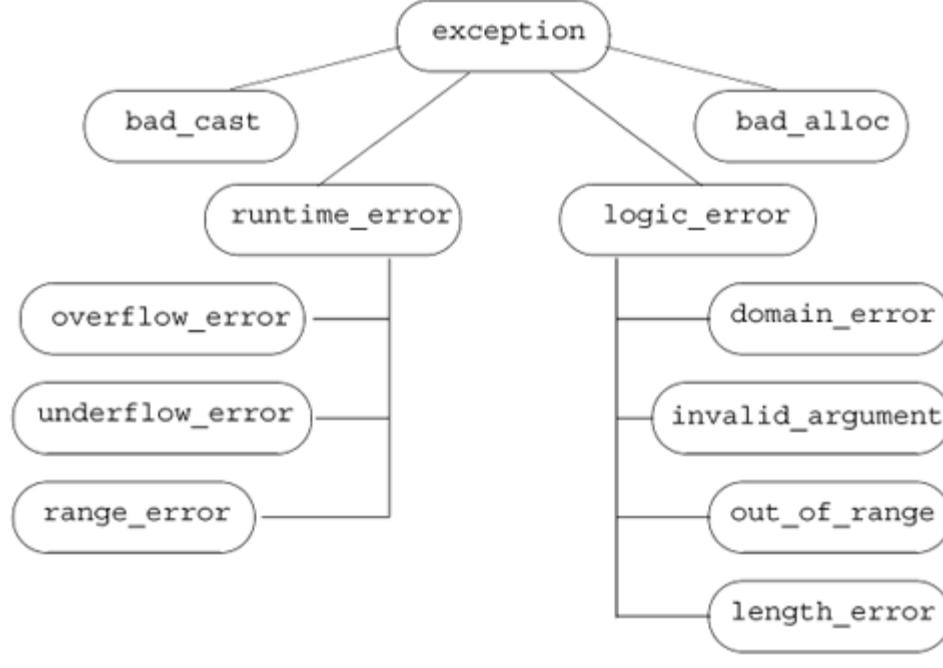
17.1.7. 异常类层次

[Section 6.13](#) (p. 215) introduced the standard-library exception classes. What that section did not cover is that these classes are related by inheritance. The inheritance hierarchy is portrayed in [Figure 17.1](#) on the following page.

[第 6.13 节](#) 介绍过标准库异常类，但该节没有涵盖的是这些类是因继承而相关的，[图 17.1](#) 描绘了这个继承层次。

Figure 17.1. Standard `exception` Class Hierarchy

图 17.1. 标准 `exception` 类层次



The only operation the `exception` types define is a virtual member named `what`. That function returns a `const char*`. It typically returns the

Section 17.1. Exception Handling

message used when constructing the exception object at the throw site. Because `what` is virtual if we catch a base-type reference, a call to the `what` function will execute the version appropriate to the dynamic type of the exception object.

`exception` 类型所定义的唯一操作是一个名为 `what` 的虚成员，该函数返回 `const char*` 对象，它一般返回用来在抛出位置构造异常对象的信息。因为 `what` 是虚函数，如果捕获了基类类型引用，对 `what` 函数的调用将执行适合异常对象的动态类型的版本。

Exception Classes for a Bookstore Application

用于书店应用程序的异常类

The standard exception classes can be used for quite a number of applications. In addition, applications often extend the `exception` hierarchy by deriving additional types from `exception` or one of the intermediate base classes. These newly derived classes can represent exception types specific to the application domain.

标准异常类可以用于许多应用程序，此外，应用程序还经常通过从 `exception` 类或者中间基类派生附加类型来扩充 `exception` 层次。这些新派生的类可以表示特定于应用程序领域的异常类型。

If we were building a real bookstore application, our classes would have been much more complex than the ones presented in this primer. One way in which they might be more elaborate would be in their handling of exceptions. In fact, we probably would have defined our own hierarchy of exceptions to represent application-specific problems that might arise. Our design might include classes such as

如果正在建立一个实际的书店应用程序，我们的类也许比本书给出的复杂得多，使它们更为精细的一个方法可能在于它们的异常处理。事实上，我们可能已经定义了自己的异常层次来表示可能出现的特定于应用程序的问题。我们的设计可能包括下面这样的类：

```
// hypothetical exception classes for a bookstore application
class out_of_stock: public std::runtime_error {
public:
    explicit out_of_stock(const std::string &s):
        std::runtime_error(s) { }
};

class isbn_mismatch: public std::logic_error {
public:
    explicit isbn_mismatch(const std::string &s):
        std::logic_error(s) { }

    isbn_mismatch(const std::string &s,
                  const std::string &lhs, const std::string &rhs):
        std::logic_error(s), left(lhs), right(rhs) { }

    const std::string left, right;
};

// Section 17.1.10 \(p. 706\) explains the destructor and why we need one
virtual ~isbn_mismatch() throw() { }
};
```

Here we defined our application-specific exception types by deriving them from the standard exception classes. As with any hierarchy, we can think of the exception classes as being organized into layers. As the hierarchy becomes deeper, each layer becomes a more specific exception. For example, the first and most general layer of the hierarchy is represented by class `exception`. All we know when we catch an object of this type is that something has gone wrong.

在这里，通过从标准异常类派生，定义了特定于应用程序的异常类型。像任何层次一样，可以认为异常类按层组织。随着层次的加深，每一层变得更特殊的异常。例如，层次中第一层即最一般的层由 `exception` 类代表，当捕获这一类型的对象时，我们所知道的只是有些地方出错了。

The second layer specializes `exception` into two broad categories: run-time or logic errors. Our bookstore exception classes represent an even more specialized layer. The class `out_of_stock` represents something that can go wrong at run time that is particular to our application. It would be used to signal that an order cannot be fulfilled. The `isbn_mismatch` exception is a more particular form of `logic_error`. In principle, a program could detect that the ISBNs don't match by calling `same_isbn`.

第二层将 `exception` 特化为两个大类：运行时错误和逻辑错误。我们的书店异常类表示更特化的层中的事件。`out_of_stock` 类表示可能在运行时出现问题的特定于应用程序的事情，可以用它发出不能履行某个订单的信号。`isbn_mismatch` 异常是从 `logic_error` 派生的更特殊的异常，原则上，程序可以通过调用 `same_isbn` 检测到不匹配的 ISBN。

Using Programmer-Defined Exception Types

使用程序员定义的异常类型

We use our own exception classes in the same way that we use one of the standard library classes. One part of the program throws an object of one of these types, and another part catches and handles the indicated problem. As an example, we might define the overloaded addition operator for our `Sales_item` class to `throw` an error of type `isbn_mismatch` if it detected that the ISBNs didn't match:

用和使用标准库类相同的方法使用自己的异常类。程序的一个部分抛出某个这些类型的对象，程序的另一部分捕获并处理指出的问题。例如，可以为 `Sales_item` 类定义重载加操作符，以便如果它检测到 ISBN 不匹配就抛出一个 `isbn_mismatch` 类型的错误：

```
// throws exception if both objects do not refer to the same isbn
```

Section 17.1. Exception Handling

```
Sales_item  
operator+(const Sales_item& lhs, const Sales_item& rhs)  
{  
    if (!lhs.same_isbn(rhs))  
        throw isbn_mismatch("isbn mismatch",  
                             lhs.book(), rhs.book());  
    Sales_item ret(lhs); // copy lhs into a local object that we'll return  
    ret += rhs; // add in the contents of rhs  
    return ret; // return ret by value  
}
```

Code that uses the addition operator could then detect this error, write an appropriate error message, and continue:

然后，使用加操作符的代码可以检测这个错误，写出适当的错误消息，并继续：

```
// use hypothetical bookstore exceptions  
Sales_item item1, item2, sum;  
while (cin >> item1 >> item2) { // read two transactions  
    try {  
        sum = item1 + item2; // calculate their sum  
        // use sum  
    } catch (const isbn_mismatch &e) {  
        cerr << e.what() << ": left isbn(" << e.left  
        << ") right isbn(" << e.right << ")"  
        << endl;  
    }  
}
```

17.1.8. Automatic Resource Deallocation

17.1.8. 自动资源释放

In [Section 17.1.2](#) (p. 691) we saw that local objects are automatically destroyed when an exception occurs. The fact that destructors are run has important implication for the design of applications. It also is one (among many) reasons why we encourage the use of the standard library classes. Consider the following function:

[第 17.1.2 节](#)介绍过，在发生异常时自动撤销局部对象。运行析构函数这一事实对应用程序的设计具有重要意义，这也是为什么我们鼓励使用标准库类的原因。考虑下面的函数：

```
void f()  
{  
    vector<string> v; // local vector  
    string s;  
    while (cin >> s)  
        v.push_back(s); // populate the vector  
    string *p = new string[v.size()]; // dynamic array  
    // remaining processing  
    // it is possible that an exception occurs in this code  
    // function cleanup is bypassed if an exception occurs  
    delete [] p;  
} // v destroyed automatically when the function exits
```

This function defines a local `vector` and dynamically allocates an array. Under normal execution, both the array and the `vector` are destroyed before the function exits. The array is freed by the last statement in the function, and the `vector` is automatically destroyed when the function ends.

这个函数定义了一个局部 `vector` 并动态分配了一个数组。在正常执行的情况下，数组和 `vector` 都在退出函数之前被撤销，函数中最后一个语句释放数组，在函数结束时自动撤销 `vector`。

However, if an exception occurs inside the function, then the `vector` will be destroyed but the array will not be freed. The problem is that the array is not freed automatically. An exception that occurs after the `new` but before the corresponding `delete` leaves the array undestroyed. No matter when an exception occurs, we are guaranteed that the `vector` destructor is run.

但是，如果在函数内部发生异常，则将撤销 `vector` 而不会释放数组。问题在于数组不是自动释放的。在 `new` 之后但在 `delete` 之前发生的异常使得数组没有被撤销。不管何时发生异常，都保证运行 `vector` 析构函数。

Using Classes to Manage Resource Allocation

用类管理资源分配

Section 17.1. Exception Handling

The fact that destructors are run leads to an important programming technique that makes programs more **exception safe**. By exception safe, we mean that the programs operate correctly even if an exception occurs. In this case, the "safety" comes from ensuring that any resource that is allocated is properly freed if an exception occurs.

对析构函数的运行导致一个重要的编程技术的出现，它使程序更为**异常安全的**。异常安全的意味着，即使发生异常，程序也能正确操作。在这种情况下，“安全”来自于保证“如果发生异常，被分配的任何资源都适当地释放”。

We can guarantee that resources are properly freed by defining a class to encapsulate the acquisition and release of a resource. This technique is often referred to as "resource allocation is initialization," often abbreviated as RAI^I.

通过定义一个类来封闭资源的分配和释放，可以保证正确释放资源。这一技术常称为“资源分配即初始化”，简称 RAI^I。

The resource-managing class should be designed so that the constructor acquires the resource and the destructor frees it. When we want to allocate the resource, we define an object of that class type. If no exception occurs, then the resource will be freed when the object that acquired the resource goes out of scope. More importantly, if an exception occurs after the object is created but before it goes out of scope, then the compiler ensures that the object is destroyed as part of unwinding the scope in which the object was defined.

应该设计资源管理类，以便 构造函数分配资源而析构函数释放资源。想要分配资源的时候，就定义该类类型的对象。如果不发生异常，就在获得资源的对象超出作用域的进修释放资源。更为重要的是，如果在创建了对象之后但在它超出作用域之前发生异常，那么，编译器保证撤销该对象，作为展开定义对象的作用域的一部分。

The following class is a prototypical example in which the constructor acquires a resource and the destructor releases it:

下面的类是一个原型例子，其中构造函数分配资源而析构函数释放资源：

```
class Resource {
public:
    Resource(parms p): r(allocate(p)) { }
    ~Resource() { release(r); }
    // also need to define copy and assignment
private:
    resource_type *r;           // resource managed by this type
    resource_type *allocate(parms p); // allocate this resource
    void release(resource_type*);   // free this resource
};
```

The `Resource` class is a type that allocates and deallocates a resource. It holds data member(s) that represent that resource. The constructor for `Resource` allocates the resource, and the destructor frees it. When we use this class

`Resource` 类是分配资源和回收资源的类型，它保存表示该资源的数据成员。`Resource` 的构造函数分配资源，而析构函数释放它。当使用这个类的时候

```
void fcn()
{
    Resource res(args); // allocates resource_type
    // code that might throw an exception
    // if exception occurs, destructor for res is run automatically
    // ...
} // res goes out of scope and is destroyed automatically
```

the resource is automatically freed. If the function terminates normally, then the resource is freed when the `Resource` object goes out of scope. If the function is exited prematurely by an exception, the destructor for `Resource` is run by the compiler as part of the exception handling process.

自动释放资源。如果函数正常终止，就在 `Resource` 对象超出作用域时释放资源；如果函数因异常而提早退出，编译器就运行 `Resource` 的析构函数作为异常处理过程的一部分。



Programs in which exceptions are possible and that allocate resources should use classes to manage those resources. As described in this section, using classes to manage acquisition and deallocation ensures that resources are freed if an exception occurs.

可能存在异常的程序以及分配资源的程序应该使用类来管理那些资源。如本节所述，使用类管理分配和回收可以保证如果发生异常就释放资源。

Exercises Section 17.1.8

Exercise Given the following function, explain what happens when the exception occurs.

17.6:

给定下面的函数，解释当发生异常时会发生什么。

```
void exercise(int *b, int *e)
{
    vector<int> v(b, e);
    int *p = new int[v.size()];
    ifstream in("ints");
    // exception occurs here
    // ...
}
```

Exercise 17.7: There are two ways to make the previous code exception-safe. Describe them and implement them.

有两种方法可以使上面的代码是异常安全的，描述并实现它们。

17.1.9. The `auto_ptr` Class

17.1.9. `auto_ptr` 类

The standard-library `auto_ptr` class is an example of the exception-safe "resource allocation is initialization" technique described in the previous subsection. The `auto_ptr` class is a template that takes a single type parameter. It provides exception safety for dynamically allocated objects. The `auto_ptr` class is defined in the `memory` header.

标准库的 `auto_ptr` 类是上一节中介绍的异常安全的“资源分配即初始化”技术的例子。`auto_ptr` 类是接受一个类型形参的模板，它为动态分配的对象提供异常安全。`auto_ptr` 类在头文件 `memory` 中定义。

Table 17.1. Class `auto_ptr`

表 17.1. <code>auto_ptr</code> 类	
<code>auto_ptr<T> ap;</code>	Create an unbound <code>auto_ptr</code> named <code>ap</code> .
	创建名为 <code>ap</code> 的未绑定的 <code>auto_ptr</code> 对象
<code>auto_ptr<T> ap(p);</code>	Create an <code>auto_ptr</code> named <code>ap</code> that owns the object pointed to by the pointer <code>p</code> . This constructor is <code>explicit</code> .
	创建名为 <code>ap</code> 的 <code>auto_ptr</code> 对象， <code>ap</code> 拥有指针 <code>p</code> 指向的对象。该构造函数为 <code>explicit</code>
<code>auto_ptr<T> ap1(ap2);</code>	Create an <code>auto_ptr</code> named <code>ap1</code> that holds the pointer originally stored in <code>ap2</code> . Transfers ownership to <code>ap1</code> ; <code>ap2</code> becomes an unbound <code>auto_ptr</code> .
	创建名为 <code>ap1</code> 的 <code>auto_ptr</code> 对象， <code>ap1</code> 保存原来存储在 <code>ap2</code> 中的指针。将所有权转给 <code>ap1</code> ， <code>ap2</code> 成为未绑定的 <code>auto_ptr</code> 对象
<code>ap1 = ap2</code>	Transfers ownership from <code>ap2</code> to <code>ap1</code> . Deletes the object to which <code>ap1</code> points and makes <code>ap1</code> point to the object to which <code>ap2</code> points, making <code>ap2</code> unbound.
	将所有权 <code>ap2</code> 转给 <code>ap1</code> 。删除 <code>ap1</code> 指向的对象并且使 <code>ap1</code> 指向 <code>ap2</code> 指向的对象，使 <code>ap2</code> 成为未绑定的
<code>~ap</code>	Destructor. Deletes the object to which <code>ap</code> points.
	析构函数。删除 <code>ap</code> 指向的对象
<code>*ap</code>	Returns a reference to the object to which <code>ap</code> is bound.
	返回对 <code>ap</code> 所绑定的对象的引用
<code>ap-></code>	Returns the pointer that <code>ap</code> holds.
	返回 <code>ap</code> 保存的指针

Section 17.1. Exception Handling

<code>ap.reset(p)</code>	If the pointer <code>p</code> is not the same value as <code>ap</code> holds, then it deletes the object to which <code>ap</code> points and binds <code>ap</code> to <code>p</code> . 如果 <code>p</code> 与 <code>ap</code> 的值不同，则删除 <code>ap</code> 指向的对象并且将 <code>ap</code> 绑定到 <code>p</code>
<code>ap.release()</code>	Returns the pointer that <code>ap</code> had held and makes <code>ap</code> unbound. 返回 <code>ap</code> 所保存的指针并且使 <code>ap</code> 成为未绑定的
<code>ap.get()</code>	Returns the pointer that <code>ap</code> holds. 返回 <code>ap</code> 保存的指针



`auto_ptr` can be used only to manage single objects returned from `new`. It does not manage dynamically allocated arrays.

`auto_ptr` 只能用于管理从 `new` 返回的一个对象，它不能管理动态分配的数组。

As we'll see, `auto_ptr` has unusual behavior when copied or assigned. As a result, `auto_ptrs` may not be stored in the library container types.

正如我们所见，当 `auto_ptr` 被复制或赋值的时候，有不寻常的行为，因此，不能将 `auto_ptrs` 存储在标准库容器类型中。

An `auto_ptr` may hold only a pointer to an object and may not be used to point to a dynamically allocated array. Using an `auto_ptr` to point to a dynamically allocated array results in undefined run-time behavior.

`auto_ptr` 对象只能保存一个指向对象的指针，并且不能用于指向动态分配的数组，使用 `auto_ptr` 对象指向动态分配的数组会导致未定义的运行时行为。

Each `auto_ptr` is either unbound or it points to an object. When an `auto_ptr` points to an object, it can be said to "own" that object. When the `auto_ptr` goes out of scope or is otherwise destroyed, then the dynamically allocated object to which the `auto_ptr` points is automatically deallocated.

每个 `auto_ptr` 对象绑定到一个对象或者指向一个对象。当 `auto_ptr` 对象指向一个对象的时候，可以说它“拥有”该对象。当 `auto_ptr` 对象超出作用域或者另外撤销的时候，就自动回收 `auto_ptr` 所指向的动态分配对象。

Using `auto_ptr` for Exception-Safe Memory Allocation

为异常安全的内存分配使用 `auto_ptr`

If memory is acquired through a normal pointer and an exception occurs before a `delete` is executed, then that memory won't be freed automatically:

如果通过常规指针分配内存，而且在执行 `delete` 之前发生异常，就不会自动释放该内存：

```
void f()
{
    int *ip = new int(42);      // dynamically allocate a new object
    // code that throws an exception that is not caught inside f
    delete ip;                // return the memory before exiting
}
```

If an exception happens between the `new` and the `delete`, and if that exception is not caught locally, then the `delete` will not be executed. The memory will never be returned.

如果在 `new` 和 `delete` 之间发生异常，并且该异常不被局部捕获，就不会执行 `delete`，则永不回收该内存。

If we use an `auto_ptr` instead, the memory will be freed automatically, even if the block is exited prematurely:

如果使用一个 `auto_ptr` 对象来代替，将会自动释放内存，即使提早退出这个块也是这样：

```
void f()
{
```

Section 17.1. Exception Handling

```
auto_ptr<int> ap(new int(42)); // allocate a new object
// code that throws an exception that is not caught inside f
}
// auto_ptr freed automatically when function ends
```

In this case, the compiler ensures that the destructor for `ap` is run before the stack is unwound past `f`.

在这个例子中，编译器保证在展开栈越过 `f` 之前运行 `ap` 的析构函数。

`auto_ptr` Is a Template and Can Hold Pointers of Any Type

`auto_ptr` 是可以保存任何类型指针的模板

The `auto_ptr` class is a template taking a single type parameter. That type names the type of the object to which the `auto_ptr` may be bound. Thus, we can create `auto_ptr`s of any type:

`auto_ptr` 类是接受单个类型形参的模板，该类型指定 `auto_ptr` 可以绑定的对象的类型，因此，可以创建任何类型的 `auto_ptr`s:

```
auto_ptr<string> ap1(new string("Brontosaurus"));
```

Binding an `auto_ptr` to a Pointer

将 `auto_ptr` 绑定到指针

In the most common case, we initialize an `auto_ptr` to the address of an object returned by a `new` expression:

在最常见的情况下，将 `auto_ptr` 对象初始化为由 `new` 表达式返回的对象的地址:

```
auto_ptr<int> pi(new int(1024));
```

This statement initializes `pi` to the address of the object created by the `new` expression. This `new` expression initializes that `int` to the value 1,024.

这个语句将 `pi` 初始化为由 `new` 表达式创建的对象的地址，这个 `new` 表达式将对象初始化为 1024。

The constructor that takes a pointer is an `explicit` ([Section 12.4.4](#), p. 462) constructor, so we must use the direct form of initialization to create an `auto_ptr`:

接受指针的构造函数为 `explicit` ([第 12.4.4 节](#)) 构造函数，所以必须使用初始化的直接形式来创建 `auto_ptr` 对象:

```
// error: constructor that takes a pointer is explicit and can't be used implicitly
auto_ptr<int> pi = new int(1024);
auto_ptr<int> pi(new int(1024)); // ok: uses direct initialization
```

The object created by the `new` expression to which `pi` refers is deleted automatically when `pi` goes out of scope. If `pi` is a local object, the object to which `pi` refers is deleted at the end of the block in which `pi` is defined. If an exception occurs, then `pi` also goes out of scope. The destructor for `pi` will be run automatically as part of handling the exception. If `pi` is a global object, the object to which `pi` refers is deleted at the end of the program.

`pi` 所指的由 `new` 表达式创建的对象在超出作用域时自动删除。如果 `pi` 是局部对象，`pi` 所指对象在定义 `pi` 的块的末尾删除；如果发生异常，则 `pi` 也超出作用域，析构函数将自动运行 `pi` 的析构函数作为异常处理的一部分；如果 `pi` 是全局对象，就在程序末尾删除 `pi` 引用的对象。

Using an `auto_ptr`

使用 `auto_ptr` 对象

Suppose we wish to access a `string` operation. With an ordinary `string` pointer, we'd do the following:

假设希望访问 `string` 操作。用普通 `string` 指针，像下面这样做:

```
string *pstr_type = new string("Brontosaurus");
```

Section 17.1. Exception Handling

```
if (pstr_type->empty())
    // oops, something wrong
```

The `auto_ptr` class defines overloaded versions of the dereference (*) and arrow (->) operators ([Section 14.6](#), p. 523). Because `auto_ptr` defines these operators, we can use an `auto_ptr` in some ways that are similar to using a built-in pointer:

`auto_ptr` 类定义了解引用操作符 (*) 和箭头操作符 (->) 的重载版本 ([第 14.6 节](#))，因为 `auto_ptr` 定义了这些操作符，所以可以用类似于使用内置指针的方式使用 `auto_ptr` 对象：

```
// normal pointer operations for dereference and arrow
*ap1 = "TRex";           // assigns a new value to the object to which ap1 points
string s = *ap1;          // initializes s as a copy of the object to which ap1 points
if (ap1->empty())         // runs empty on the string to which ap1 points
```

The primary purpose of `auto_ptr` is to support ordinary pointerlike behavior while ensuring that the object to which an `auto_ptr` object refers is automatically deleted. As we'll see, the fact that objects are automatically deleted leads to significant differences between `auto_ptrs` and ordinary pointers with respect to how we copy and access their address value.

`auto_ptr` 的主要目的，在保证自动删除 `auto_ptr` 对象引用的对象的同时，支持普通指针式行为。正如我们所见，自动删除该对象这一事实导致在怎样复制和访问它们的地址值方面，`auto_ptrs` 与普通指针明显不同。

Copy and Assignment on `auto_ptr` Are Destructive Operations

`auto_ptr` 对象的复制和赋值是破坏性操作



There is a crucially important difference between how `auto_ptr` and built-in pointers treat copy and assignment. When we copy an `auto_ptr` or assign its value to another `auto_ptr`, ownership of the underlying object is transferred from the original to the copy. The original `auto_ptr` is reset to an unbound state.

`auto_ptr` 和内置指针对待复制和赋值有非常关键的重要区别。当复制 `auto_ptr` 对象或者将它的值赋给其他 `auto_ptr` 对象的时候，将基础对象的所有权从原来的 `auto_ptr` 对象转给副本，原来的 `auto_ptr` 对象重置为未绑定状态。

Copying (or assigning) ordinary pointers copies (assigns) the address. After the copy (assignment), both pointers point to the same object. After copying (or assigning) `auto_ptrs`, the original points to no object and the new `auto_ptr` (left-hand `auto_ptr`) owns the underlying object:

```
auto_ptr<string> ap1(new string("Stegosaurus"));
// after the copy ap1 is unbound
auto_ptr<string> ap2(ap1); // ownership transferred from ap1 to ap2
```

When we copy or assign an `auto_ptr`, the right-hand `auto_ptr` relinquishes all responsibility for the underlying object and is reset to be an unbound `auto_ptr`. In our example, it is `ap2` that deletes the `string` object, and not `ap1`. After the copy, `ap1` no longer refers to any object.

当复制 `auto_ptr` 对象或者对 `auto_ptr` 对象赋值的时候，右边的 `auto_ptr` 对象让出对基础对象的所有职责并重置为未绑定的 `auto_ptr` 对象之后，在上例中，删除 `string` 对象的是 `ap2` 而不是 `ap1`，在复制之后，`ap1` 不再指向任何对象。

Unlike other copy or assignment operations, `auto_ptr` copy and assignment change the right-hand operand. As a result, both the left- and right-hand operands to assignment must be modifiable lvalues.

与其他复制或赋值操作不同，`auto_ptr` 的复制和赋值改变右操作数，因此，赋值的左右操作数必须都是可修改的左值。

Assignment Deletes the Object Pointed To by the Left Operand

赋值删除左操作数指向的对象

Section 17.1. Exception Handling

In addition to transferring ownership from the right-hand to the left-hand operand, assignment also deletes the object to which the left-hand operand originally referred provided that the two objects are different. As usual, self-assignment has no effect.

除了将所有权从右操作数转给左操作数之外，赋值还删除左操作数原来指向的对象——假如两个对象不同。通常自身赋值没有效果。

```
auto_ptr<string> ap3(new string("Pterodactyl"));
// object pointed to by ap3 is deleted and ownership transferred from ap2 to ap3;
ap3 = ap2; // after the assignment, ap2 is unbound
```

After the assignment of `ap2` to `ap3`,

将 `ap2` 赋给 `ap3` 之后：

- the object to which `ap3` had pointed is deleted;
删除了 `ap3` 指向的对象。
- `ap3` is set to point to the object to which `ap2` pointed; and
将 `ap3` 置为指向 `ap2` 所指向的对象。
- `ap2` is an unbound `auto_ptr`.
`ap2` 是未绑定的 `auto_ptr` 对象。



Because copy and assignment are destructive operations, `auto_ptrs` cannot be stored in the standard containers. The library container classes require that two objects be equal after a copy or assignment. This requirement is not met by `auto_ptr`. If we assign `ap2` to `ap1`, then after the assignment `ap1 != ap2`. Similarly for copy.

因为复制和赋值是破坏性操作，所以 `auto_ptr`s 不能将 `auto_ptr` 对象存储在标准容器中。标准库的容器类要求在复制或赋值之后两个对象相等，`auto_ptr` 不满足这一要求，如果将 `ap2` 赋给 `ap1`，则在赋值之后 `ap1 != ap2`，复制也类似。

The Default `auto_ptr` Constructor

`auto_ptr` 的默认构造函数

If no initializer is given, the `auto_ptr` is *unbound*; it doesn't refer to an object:

如果不给定初始式，`auto_ptr` 对象是未绑定的，它不指向对象：

```
auto_ptr<int> p_auto; // p_auto doesn't refer to any object
```

By default, the internal pointer value of an `auto_ptr` is set to 0. Dereferencing an unbound `auto_ptr` has the same effect as dereferencing an unbound pointer—the program is in error and what happens is undefined:

默认情况下，`auto_ptr` 的内部指针值置为 0。对未绑定的 `auto_ptr` 对象解引用，其效果与对未绑定的指针解引用相同——程序出错并且没有定义会发生什么：

```
*p_auto = 1024; // error: dereference auto_ptr that doesn't point to an object
```

Testing an `auto_ptr`

测试 `auto_ptr` 对象

To check whether a pointer is unbound, we can test the pointer directly in a condition, which has the effect of determining whether the pointer is 0. In contrast, we cannot test an `auto_ptr` directly.

Section 17.1. Exception Handling

为了检查指针是否未绑定，可以在条件中直接测试指针，效果是确定指针是否为 0。相反，不能直接测试 `auto_ptr` 对象：

```
// error: cannot use an auto_ptr as a condition
if (p_auto)
    *p_auto = 1024;
```

The `auto_ptr` type does not define a conversion to a type that can be used as a condition. Instead, to test the `auto_ptr`, we must use its `get` member, which returns the underlying pointer contained in the `auto_ptr`:

```
// revised test to guarantee p_auto refers to an object
if (p_auto.get())
    *p_auto = 1024;
```

To determine whether the `auto_ptr` object refers to an object, we can compare the return from `get` with 0.

为了确定 `auto_ptr` 是否指向一个对象，可以将 `get` 的返回值与 0 比较。



`get` should be used only to interrogate an `auto_ptr` or to use the returned pointer value. `get` should not be used as an argument to create another `auto_ptr`.

应该只用 `get` 询问 `auto_ptr` 对象或者使用返回的指针值，不能用 `get` 作为创建其他 `auto_ptr` 对象的实参。

Using `get` member to initialize another `auto_ptr` violates the class design principle that only one `auto_ptr` holds a given pointer at any one time. If two `auto_ptrs` hold the same pointer, then the pointer will be `deleted` twice.

使用 `get` 成员初始化其他 `auto_ptr` 对象违反 `auto_ptr` 类设计原则：在任意时刻只有一个 `auto_ptr` 对象保存给定指针，如果两个 `auto_ptr` 对象保存相同的指针，该指针就会被 `delete` 两次。

The `reset` Operation

`reset` 操作

Another difference between `auto_ptr` and a built-in pointer is that we cannot assign an address (or other pointer) directly to an `auto_ptr`:

`auto_ptr` 对象与内置指针的另一个区别是，不能直接将一个地址（或者其他指针）赋给 `auto_ptr` 对象：

```
p_auto = new int(1024); // error: cannot assign a pointer to an auto_ptr
```

Instead, we must call `reset` to change the pointer:

相反，必须调用 `reset` 函数来改变指针：

```
// revised test to guarantee p_auto refers to an object
if (p_auto.get())
    *p_auto = 1024;
else
    // reset p_auto to a new object
    p_auto.reset(new int(1024));
```

To unset the `auto_ptr` object, we could pass 0 to `reset`.

要复位 `auto_ptr` 对象，可以将 0 传给 `reset` 函数。

Calling `reset` on an `auto_ptr` deletes the object (if any) to which the `auto_ptr` refers before binding the

Section 17.1. Exception Handling



`auto_ptr` to another object. However, just as self-assignment has no effect, if we call `reset` on the same pointer that the `auto_ptr` already holds, then there is no effect; the object is not deleted.

调用 `auto_ptr` 对象的 `reset` 函数时, 在将 `auto_ptr` 对象绑定到其他对象之前, 会删除 `auto_ptr` 对象所指向的对象(如果存在)。但是, 正如自身赋值是没效果的一样, 如果调用该 `auto_ptr` 对象已经保存的同一指针的 `reset` 函数, 也没有效果, 不会删除对象。

Exercises Section 17.1.9

Exercise 17.8: Which of the following `auto_ptr` declarations are illegal or likely to result in subsequent program error? Explain what the problem is with each one.

下面 `auto_ptr` 声明中, 哪些是不合法的或者可能导致随后的程序错误? 解释每个声明的问题。

```
int ix = 1024, *pi = &ix, *pi2 = new int(2048);
typedef auto_ptr<int> IntP;
(a) IntP p0(ix);          (b) IntP p1(pi);
(c) IntP p2(pi2);         (d) IntP p3(&ix);
(e) IntP p4(new int(2048)); (f) IntP p5(p2.get());
```

Exercise 17.9: Assuming `ps` is a pointer to `string`, what is the difference, if any, between the following two invocations of `assign` ([Section 9.6.2](#), p. 339)? Which do you think is preferable? Why?

假定 `ps` 是一个指向 `string` 的指针, 如果有的话, 下面两个 `assign` ([第 9.6.2 节](#)) 调用有什么不同? 你认为哪个更好, 为什么?

(a) `ps.get() ->assign("Danny");` (b) `ps->assign("Danny");`

17.1.10. Exception Specifications

17.1.10. 异常说明

When looking at an ordinary function declaration, it is not possible to determine what exceptions the function might throw. However, it can be useful to know whether and which exceptions a function might throw in order to write appropriate `catch` clauses. An [exception specification](#) specifies that if the function throws an exception, the exception it throws will be one of the exceptions included in the specification, or it will be a type derived from one of the listed exceptions.

查看普通函数声明的时候, 不可能确定该函数会抛出什么异常, 但是, 为了编写适当的 `catch` 子句, 了解函数是否抛出异常以及会抛出哪种异常是很有用的。[异常说明](#) 指定, 如果函数抛出异常, 被抛出的异常将是包含在该说明中的一种, 或者是从列出的异常中派生的类型。

Caution: `Auto_ptr` Pitfalls

警告: `Auto_ptr` 缺陷

The `auto_ptr` class template provides a measure of safety and convenience for handling dynamically allocated memory. To use `auto_ptr` correctly, we must adhere to the restrictions that the class imposes:

Section 17.1. Exception Handling

`auto_ptr` 焰模板为处理动态分配的内存提供了安全性和便利性的尺度。要正确地使用 `auto_ptr` 类，必须坚持该类强加的下列限制：

1. **Do not use an `auto_ptr` to hold a pointer to a statically allocated object. Otherwise, when the `auto_ptr` itself is destroyed, it will attempt to delete a pointer to a nondynamically allocated object, resulting in undefined behavior.**
不要使用 `auto_ptr` 对象保存指向静态分配对象的指针，否则，当 `auto_ptr` 对象本身被撤销的时候，它将试图删除指向非动态分配对象的指针，导致未定义的行为。
2. **Never use two `auto_ptrs` to refer to the same object. One obvious way to make this mistake is to use the same pointer to initialize or to `reset` two different `auto_ptr` objects. A more subtle way to make this mistake would be to use the result from `get` on one `auto_ptr` to initialize or `reset` another.**
永远不要使用两个 `auto_ptrs` 对象指向同一对象，导致这个错误的一种明显方式是，使用同一指针来初始化或者 `reset` 两个不同的 `auto_ptr` 对象。另一种导致这个错误的微妙方式可能是，使用一个 `auto_ptr` 对象的 `get` 函数的结果来初始化或者 `reset` 另一个 `auto_ptr` 对象。

3. **Do not use an `auto_ptr` to hold a pointer to a dynamically allocated array. When the `auto_ptr` is destroyed, it frees only a single object it uses the plain `delete` operator, not the array `delete []` operator.**
不要使用 `auto_ptr` 对象保存指向动态分配数组的指针。当 `auto_ptr` 对象被删除的时候，它只释放一个对象——它使用普通 `delete` 操作符，而不用数组的 `delete []` 操作符。

4. **Do not store an `auto_ptr` in a container. Containers require that the types they hold define copy and assignment to behave similarly to how those operations behave on the built-in types: After the copy (or assignment), the two objects must have the same value. `auto_ptr` does not meet this requirement.**
不要将 `auto_ptr` 对象存储在容器中。容器要求所保存的类型定义复制和赋值操作符，使它们表现得类似于内置类型的操作符：在复制（或者赋值）之后，两个对象必须具有相同值，`auto_ptr` 类不满足这个要求。

Defining an Exception Specification

定义异常说明

An exception specification follows the function parameter list. An exception specification is the keyword `throw` followed by a (possibly empty) list of exception types enclosed in parentheses:

异常说明跟在函数形参表之后。一个异常说明在关键字 `throw` 之后跟着一个（可能为空的）由圆括号括住的异常类型列表：

```
void recoup(int) throw(runtime_error);
```

This declaration says that `recoup` is a function taking an `int`, and returning `void`. If `recoup` throws an exception, that exception will be a `runtime_error` or an exception of a type derived from `runtime_error`.

这个声明指出，`recoup` 是接受 `int` 值的函数，并返回 `void`。如果 `recoup` 抛出一个异常，该异常将是 `runtime_error` 对象，或者是由 `runtime_error` 派生的类型的异常。

An empty specification list says that the function does not throw any exception:

空说明列表指出函数不抛出任何异常：

```
void no_problem() throw();
```

An exception specification is part of the function's interface. The function definition and any declarations of the function must have the same specification.

异常说明是函数接口的一部分，函数定义以及该函数的任意声明必须具有相同的异常说明。



If a function declaration does not specify an exception specification, the function can throw exceptions of any type.

如果一个函数声明没有指定异常说明，则该函数可以抛出任意类型的异常。

Violating the Exception Specification

违反异常说明

Unfortunately, it is not possible to know at compile time whether or which exceptions a program will throw. Violations of a function's exception specification can be detected only at run time.

但是，不可能在编译时知道程序是否抛出异常以及会抛出哪些异常，只有在运行时才能检测是否违反函数异常说明。

If a function throws an exception not listed in its specification, the library function `unexpected` is invoked. By default, `unexpected` calls `terminate`, which ordinarily aborts the program.

如果函数抛出了没有在其异常说明中列出的异常，就调用标准库函数 `unexpected`。默认情况下，`unexpected` 函数调用 `terminate` 函数，`terminate` 函数一般会终止程序。



The compiler cannot and does not attempt to verify exception specifications at compile time.

在编译的时候，编译器不能也不会试图验证异常说明。

Even if a casual reading of a function's code indicates that it *might* throw an exception missing from the specification, the compiler will not complain:

即使对函数代码的偶然阅读指明，它可能抛出异常说明中没有的异常，编译器也不会给出提示：

```
void f() throw()          // promise not to throw any exception
{
    throw exception();    // violates exception specification
}
```

Instead, the compiler generates code to ensure that `unexpected` is called if an exception violating the exception specification is thrown.

相反，编译器会产生代码以便保证：如果抛出了一个违反异常说明的异常，就调用 `unexpected` 函数。

Specifying that the Function Does Not Throw

确定函数不抛出异常

Because an exception specification cannot be checked at compile time, the practical utility of exception specifications is often limited.

因为不能在编译时检查异常说明，异常说明的应用通常是有限的。



One important case when an exception specification is useful is if a function can guarantee that it will not throw any exceptions.

异常说明有用的一种重要情况是，如果函数可以保证不会抛出任何异常。

Specifying that a function will not `throw` any exceptions can be helpful both to users of the function and to the compiler: Knowing that a function

Section 17.1. Exception Handling

will not `throw` simplifies the task of writing exception-safe code that calls that function. We can know that we need not worry about exceptions when calling it. Moreover, if the compiler knows that no exceptions will be thrown, it can perform optimizations that are suppressed for code that might throw.

确定函数将不抛出任何异常，对函数的用户和编译器都有所帮助：知道函数不抛出异常会简化编写调用该函数的异常安全的代码的工作，我们可以知道在调用函数时不必担心异常，而且，如果编译器知道不会抛出异常，它就可以执行被可能抛出异常的代码所抑制的优化。

Exception Specifications and Member Functions

异常说明与成员函数

As with nonmember functions, an exception specification on a member function declaration follows the function parameter list. For example, the class `bad_alloc` from the C++ standard library is defined so that all its member functions have an empty exception specification. These members promise not to throw an exception:

像非成员函数一样，成员函数声明的异常说明跟在函数形参表之后。例如，C++ 标准库中的 `bad_alloc` 类定义为所有成员都有空异常说明，这些成员承诺不抛出异常：

```
// illustrative definition of library bad_alloc class
class bad_alloc : public exception {
public:
    bad_alloc() throw();
    bad_alloc(const bad_alloc &) throw();
    bad_alloc & operator=(const
        bad_alloc &) throw();
    virtual ~bad_alloc() throw();
    virtual const char* what() const throw();
};
```

Notice that the exception specification follows the `const` qualifier in `const` member function declarations.

注意，在 `const` 成员函数声明中，异常说明跟在 `const` 限定符之后。

Exception Specifications and Destructors

异常说明与析构函数

In [Section 17.1.7](#) (p. 697) we showed two hypothetical bookstore application exception classes. The `isbn_mismatch` class defines its destructor as [第 17.1.7 节](#)介绍了两个假设的书店应用程序异常类，`isbn_mismatch` 类将析构函数定义为：

```
class isbn_mismatch: public std::logic_error {
public:
    virtual ~isbn_mismatch() throw() { }
```

and said that we would explain this usage here.

并说明我们会在这里解释这种用法。

The `isbn_mismatch` class inherits from `logic_error`, which is one of the standard exception classes. The destructors for the standard exception classes include an empty `throw()` specifier; they promise that they will not throw any exceptions. When we inherit from one of these classes, then our destructor must also promise not to throw any exceptions.

`isbn_mismatch` 类从 `logic_error` 类继承而来，`logic_error` 是一个标准异常类，该标准异常类的析构函数包含空 `throw()` 说明符，它们承诺不抛出任何异常。当继承这两个类中的一个时，我们的析构函数也必须承诺不抛出任何异常。

Our `out_of_stock` class had no members, and so its synthesized destructor does nothing that might throw an exception. Hence, the compiler can know that the synthesized destructor will abide by the promise not to throw.

`out_of_stock` 类没有成员，所以它的合成析构函数不做任何可能抛出异常的事情，因此，编译器可以知道合成析构函数将遵守不抛出异常的承诺。

The `isbn_mismatch` class has two members of class `string`, which means that the synthesized destructor for `isbn_mismatch` calls the `string` destructor. The C++ standard stipulates that `string` destructor, like any other library class destructor, will not throw an exception. However, the library destructors do not define exception specifications. In this case, we know, but the compiler doesn't, that the `string` destructor won't throw. We must define our own destructor to reinstate the promise that the destructor will not throw.

`isbn_mismatch` 类有两个 `string` 类成员，这意味着 `isbn_mismatch` 的合成析构函数调用 `string` 析构函数。C++ 标准保证，`string` 析构函数像任意其他标准库类析构函数一样，不抛出异常。但是，标准库的析构函数没有定义异常说明，在这种情况下，我们知道，但编译器不知道，`string` 析构函数将不抛出异常。我们必须定义自己的析构函数来恢复析构函数不抛出异常的承诺。

Exception Specifications and Virtual Functions

异常说明与虚函数

A virtual function in a base class may have an exception specification that differs from the exception specification of the corresponding virtual in a derived class. However, the exception specification of a derived-class virtual function must be either equally or more restrictive than the exception specification of the corresponding base-class virtual function.

基类中虚函数的异常说明，可以与派生类中对应虚函数的异常说明不同。但是，派生类虚函数的异常说明必须与对应基类虚函数的异常说明同样严格，或者比后者更受限。

This restriction ensures that when a pointer to a base-class type is used to call a derived virtual function, the exception specification of the derived class adds no new exceptions to those that the base said could be thrown. For example,

这个限制保证，当使用指向基类类型的指针调用派生类虚函数的时候，派生类的异常说明不会增加新的可抛出异常。例如：

```
class Base {
public:
    virtual double f1(double) throw ();
    virtual int f2(int) throw (std::logic_error);
    virtual std::string f3() throw
        (std::logic_error, std::runtime_error);
};

class Derived : public Base {
public:
    // error: exception specification is less restrictive than Base::f1's
    double f1(double) throw (std::underflow_error);

    // ok: same exception specification as Base::f2
    int f2(int) throw (std::logic_error);
    // ok: Derived f3 is more restrictive
    std::string f3() throw ();
};
```

The declaration of `f1` in the derived class is an error because its exception specification adds an exception to those listed in the version of `f1` in the base class. The reason that the derived class may not add to the specification list is users of the hierarchy should be able to write code that depends on the specification list. If a call is made through a base pointer or reference, then only the exceptions specified in the base should be of concern to a user of these classes.

派生类中 `f1` 的声明是错误的，因为它的异常说明在基类 `f1` 版本列出的异常中增加了一个异常。派生类不能在异常说明列表中增加异常，原因在于，继承层次的用户应该能够编写依赖于该说明列表的代码。如果通过基类指针或引用进行函数调用，那么，这些类的用户所涉及的应该只是在基类中指定的异常。

By restricting which exceptions the derived classes will throw to those listed by the base class, we can write our code knowing what exceptions we must handle. Our code can rely on the fact that the list of exceptions in the base class is a superset of the list of exceptions that a derived-class version of the virtual might throw. As an example, when calling `f3`, we know we need to handle only `logic_error` or `runtime_error`:

通过派生类抛出的异常限制为由基类所列出的那些，在编写代码时就可以知道必须处理哪些异常。代码可以依赖于这样一个事实：基类中的异常列表是虚函数的派生类版本可以抛出的异常列表的超集。例如，当调用 `f3` 的时候，我们知道只需要处理 `logic_error` 或 `runtime_error`：

```
// guarantees not to throw exceptions
void compute(Base *pb) throw()
{
    try {
        // may throw exception of type std::logic_error
        // or std::runtime_error
        pb->f3();
    } catch (const logic_error &le) { /* ... */ }
    catch (const runtime_error &re) { /* ... */ }
}
```

The function `compute` uses the specification in the base class in deciding what exceptions it might need to catch.

在确定可能需要捕获什么异常的时候，`compute` 函数使用基类中的异常说明。

17.1.11. Function Pointer Exception Specifications

17.1.11. 函数指针的异常说明

An exception specification is part of a function type. As such, exception specifications can be provided in the definition of a pointer to function:

异常说明是函数类型的一部分。这样，也可以在函数指针的定义中提供异常说明：

Section 17.1. Exception Handling

```
void (*pf)(int) throw(runtime_error);
```

This declaration says that `pf` points to a function that takes an `int`, returns `void`, and that can throw exceptions only of type `runtime_error`. If no specification is provided, then the pointer may point at a function with matching type that could throw any kind of exception.

这个声明是说，`pf` 指向接受 `int` 值的函数，该函数返回 `void` 对象，该函数只能抛出 `runtime_error` 类型的异常。如果不提供异常说明，该指针就可以指向能够抛出任意类型异常的具有匹配类型的函数。

When a pointer to function with an exception specification is initialized from (or assigned to) another pointer (or to the address of a function), the exception specifications of both pointers do not have to be identical. However, the specification of the source pointer must be at least as restrictive as the specification of the destination pointer

在用另一指针初始化带异常说明的函数的指针，或者将后者赋值给函数地址的时候，两个指针的异常说明不必相同，但是，源指针的异常说明必须至少与目标指针的一样严格。

```
void recoup(int) throw(runtime_error);
// ok: recoup is as restrictive as pf1
void (*pf1)(int) throw(runtime_error) = recoup;
// ok: recoup is more restrictive than pf2
void (*pf2)(int) throw(runtime_error, logic_error) = recoup;
// error: recoup is less restrictive than pf3
void (*pf3)(int) throw() = recoup;
// ok: recoup is more restrictive than pf4
void (*pf4)(int) = recoup;
```

The third initialization is an error. The pointer declaration says that `pf3` points to a function that will not throw any exceptions. However, `recoup` says it can throw exceptions of type `runtime_error`. The `recoup` function throws exception types beyond those specified by `pf3`. The `recoup` function is not a valid initializer for `pf3`, and a compile-time error is issued.

第三个初始化是错误的。指针声明指出，`pf3` 指向不抛出任何异常的函数，但是，`recoup` 函数指出它能抛出 `runtime_error` 类型的异常，`recoup` 函数抛出的异常类型超出了 `pf3` 所指定的，对 `pf3` 而言，`recoup` 函数不是有效的初始化式，并且会引发一个编译时错误。

Exercises Section 17.1.11

Exercise 17.10: What exceptions can a function throw if it has an exception specification of the form `throw()`? If it has no exception specification?

如果函数有形如 `throw()` 的异常说明，它能抛出什么异常？如果没有异常说明呢？

Exercise 17.11: Which, if either, of the following initializations is in error? Why?

如果有，下面哪个初始化是错误的？为什么？

```
void example() throw(string);
(a) void (*pf1)() = example;
(b) void (*pf2)() throw() = example;
```

Exercise 17.12: Which exceptions might the following functions throw?

下面函数可以抛出哪些异常？

```
(a) void operate() throw(logic_error);
(b) int op(int) throw(underflow_error, overflow_error);
(c) char manip(string) throw();
(d) void process();
```

17.2. Namespaces

17.2. 命名空间

Every name defined in a given scope must be unique within that scope. This requirement can be difficult to satisfy for large, complex applications. Such applications tend to have many names defined in the global scope. Complex programs composed of independently developed libraries are even more likely to encounter name collisions—the same name is used in our own code or (more often) in the code supplied to us by independent producers.

在一个给定作用域中定义的每个名字在该作用域中必须是唯一的，对庞大、复杂的应用程序而言，这个要求可能难以满足。这样的应用程序的全局作用域中一般有许多名字定义。由独立开发的库构成的复杂程序更有可能遇到名字冲突——同样的名字既可能在我们自己的代码中使用，也可能（更常见地）在独立供应商提供的代码中使用。

Libraries tend to define a large number of global names—primarily names of templates, types and functions. When writing an application using libraries from many different vendors, it is almost inevitable that some of these names will clash. This name-clashing problem is known as the [namespace pollution](#) problem.

库倾向于定义许多全局名字——主要是模板名、类型名或函数名。在使用来自多个供应商的库编写应用程序的时候，这些名字中有一些几乎不可避免地会发生冲突，这种名字冲突问题称为[命名空间污染](#)问题。

Traditionally, programmers avoided namespace pollution by making names of global entities very long, often prefixing the names in their program with specific character sequences:

传统上，程序员通过将全局实体的名字设得很长来避免命名空间污染，经常用特定字符序列作为程序中名字的前缀：

```
class cplusplus_primer_Query { ... };
ifstream&
cplusplus_primer_open_file(ifstream&, const string&);
```

This solution is far from ideal: It can be cumbersome for programmers to write and read programs that use such long names. [Namespaces](#) provide a much more controlled mechanism for preventing name collisions. Namespaces partition the global namespace, making it easier to use independently produced libraries. A namespace is a scope. By defining a library's names inside a namespace, library authors (and users) can avoid the limitations inherent in global names.

这个解决方案很不理想：程序员编写和阅读使用这种长名字的程序非常麻烦。[命名空间](#)为防止名字冲突提供了更加可控的机制，命名空间能够划分全局命名空间，这样使用独立开发的库就更加容易了。一个命名空间是一个作用域，通过在命名空间内部定义库中的名字，库的作者（以及用户）可以避免全局名字固有的限制。

17.2.1. Namespace Definitions

17.2.1. 命名空间的定义

A namespace definition begins with the keyword `namespace` followed by the namespace name.

命名空间定义以关键字 `namespace` 开始，后接命名空间的名字。

```
namespace cplusplus_primer {
    class Sales_item { /* ... */;
        Sales_item operator+(const Sales_item&,
                             const Sales_item&);

    class Query {
        public:
            Query(const std::string&,
                  std::ostream &display(std::ostream&) const;
            // ...
    };
    class Query_base { /* ... */;
}
```

This code defines a namespace named `cplusplus_primer` with four members: two classes, an overloaded `+` operator, and a function.

这段代码定义了名为 `cplusplus_primer` 的命名空间，它有四个成员：两个类，一个重载的 `+` 操作符，一个函数。

As with other names, the name of a namespace must be unique within the scope in which the namespace is defined. Namespaces may be defined at global scope or inside another namespace. They may not be defined inside a function or a class.

Section 17.2. Namespaces

像其他名字一样，命名空间的名字在定义该命名空间的作用域中必须是唯一的。命名空间可以在全局作用域或其他作用域内部定义，但不能在函数或类内部定义。

Following the namespace name is a block of declarations and definitions delimited by curly braces. Any declaration that can appear at global scope can be put into a namespace: classes, variables (with their initializations), functions (with their definitions), templates, and other namespaces.

命名空间名字后面接着由花括号括住的一块声明和定义，可以在命名空间中放入可以出现在全局作用域的任意声明：类、变量（以及它们的初始化）、函数（以及它们的定义）、模板以及其他命名空间。



A namespace scope does not end with a semicolon.

命名空间作用域不能以分号结束。

Each Namespace Is a Scope

每个命名空间是一个作用域

The entities defined in a namespace are called namespace members. Just as is the case for any scope, each name in a namespace must refer to a unique entity within that namespace. Because different namespaces introduce different scopes, different namespaces may have members with the same name.

定义在命名空间中的实体称为命名空间成员。像任意作用域的情况一样，命名空间中的每个名字必须引用该命名空间中的唯一实体。因为不同命名空间引入不同作用域，所以不同命名空间可以具有同名成员。

Names defined in a namespace may be accessed directly by other members of the namespace. Code outside the namespace must indicate the namespace in which the name is defined:

在命名空间中定义的名字可以被命名空间中的其他成员直接成员，命名空间外部的代码必须指出名字定义在哪个命名空间中：

```
cplusplus_primer::Query q =
    cplusplus_primer::Query("hello");
q.display(cout);
// ...
```

If another namespace (say, `AddisonWesley`) also provides a `TextQuery` class and we want to use that class instead of the one defined in `cplusplus_primer`, we can do so by modifying our code as follows:

如果另一命名空间（如 `AddisonWesley`）也提供 `TextQuery` 类，而且我们想要使用那个类代替 `cplusplus_primer` 中定义的 `TextQuery`，可以通过这样修改代码而实现：

```
AddisonWesley::Query q = AddisonWesley::Query("hello");
q.display(cout);
// ...
```

Using Namespace Members from Outside the Namespace

从命名空间外部使用命名空间成员

Of course, always referring to a namespace member using the qualified name

当然，总是使用限定名

```
namespace_name::member_name
```

can be cumbersome. Just as we've been doing for names defined in the `std` namespace, we can write a `using` declaration ([Section 3.1](#), p. 78) to obtain direct access to names we know we'll use frequently:

引用命名空间成员可能非常麻烦。像对 `std` 中定义的命名空间所做的那样，可以编写 `using` 声明（[第 3.1 节](#)）来获得对我们知道将经常使用的名称的直接访问：

```
using cplusplus_primer::Query;
```

After this `using` declaration, our program can use the name `Query` directly without the `cplusplus_primer` qualifier. We'll see other ways to simplify

Section 17.2. Namespaces

access in [Section 17.2.4](#) (p. 720).

在这个 `using` 声明之后，程序可以无须 `cplusplus_primer` 限定符而直接使用名字 `Query`，在[第 17.2.4 节](#)将介绍简化访问的其他方法。

Namespaces Can Be Discontiguous

命名空间可以是不连续的

Unlike other scopes, a namespace can be defined in several parts. A namespace is made up of the sum of its separately defined parts; a namespace is cumulative. The separate parts of a namespace can be spread over multiple files. Namespace definitions in different text files are also cumulative. Of course, the usual restriction continues to apply that names are visible only in the files in which they are declared. So, if one part of the namespace requires a name defined in another file, that name must still be declared.

与其他作用域不同，命名空间可以在几个部分中定义。命名空间由它的分离定义部分的总和构成，命名空间是累积的。一个命名空间的分离部分可以分散在多个文件中，在不同文本文件中的命名空间定义也是累积的。当然，名字只在声明名字的文件中可见，这一常规限制继续应用，所以，如果命名空间的一个部分需要定义在另一文件中的名字，仍然必须声明该名字。

Writing a namespace definition

编写命名空间定义：

```
namespace namespace_name {  
    // declarations  
}
```

either defines a new namespace or adds to an existing one.

既可以定义新的命名空间，也可以添加到现在命名空间中。

If the name `namespace_name` does not refer to a previously defined namespace, then a new namespace with that name is created. Otherwise, this definition opens an existing namespace and adds these new declarations to that namespace.

如果名字 `namespace_name` 不是引用前面定义的命名空间，则用该名字创建新的命名空间，否则，这个定义打开一个已存在的命名空间，并将这些新声明加到那个命名空间。

Separation of Interface and Implementation

接口和实现的分离

The fact that namespace definitions can be discontiguous means that we can compose a namespace from separate interface and implementation files. Thus, a namespace can be organized in the same way that we manage our own class and function definitions:

命名空间定义可以不连续意味着，可以用分离的接口文件和实现文件构成命名空间，因此，可以用与管理自己的类和函数定义相同的方法来组织命名空间：

1. Namespace members that define classes and declarations for the functions and objects that are part of the class interface can be put into header files. These headers can be included by files that use namespace members.
定义类的命名空间成员，以及作为类接口的一部分的函数声明与对象声明，可以放在头文件中，使用命名空间成员的文件可以包含这些头文件。
2. The definitions of namespace members can be put in separate source files.
命名空间成员的定义可以放在单独的源文件中。

Organizing our namespaces this way also satisfies the requirement that various entities—non-inline functions, static data members, variables, and so forth—may be defined only once in a program. This requirement applies equally to names defined in a namespace. By separating the interface and implementation, we can ensure that the functions and other names we need are defined only once, but the same declaration will be seen whenever the entity is used.

按这种方式组织命名空间，也满足了不同实体（非内联函数、静态数据成员、变量等）只能在一个程序中定义一次的要求，这个要求同样适用于命名空间中定义的名字。通过将接口和实现分离，可以保证函数和其他我们需要的名字只定义一次，但相同的声明可以在任何使用该实体的地方见到。



Namespaces that define multiple, unrelated types should use separate files to represent each type that the namespace defines.

定义多个不相关类型的命名空间应该使用分离的文件，表示该命名空间定义的每个类型。

Defining the Primer Namespace

定义本书的命名空间

Using this strategy for separating interface and implementation, we might define the `cplusplus_primer` library in several separate files. The declarations for `Sales_item` and its related functions that we built in [Part I](#) (p. 31) would be placed in `Sales_item.h`, those for the `Query` classes of [Chapter 15](#) (p. 557) in `Query.h`, and so on. The corresponding implementation files would be in files such as `Sales_item.cc` and `Query.cc`:

使用将接口和实现分离的策略，可以将 `cplusplus_primer` 库定义在几个分离的文件中。本书第一部分建立的 `Sales_item` 的声明及其相关函数可以放在 `Sales_item.h` 中，[第十五章](#) 的 `Query` 类的定义以及相关函数放在 `Query.h` 中，以此类推。对应的实现文件可以是 `Sales_item.cc` 和 `Query.cc`:

```
// ---- Sales_item.h ----
namespace cplusplus_primer {
    class Sales_item { /* ... */;
        Sales_item operator+(const Sales_item&,
                             const Sales_item&);

    // declarations for remaining functions in the Sales_item interface
}

// ---- Query.h----
namespace cplusplus_primer {
    class Query {
    public:
        Query(const std::string&);
        std::ostream &display(std::ostream&) const;
        // ...
    };
    class Query_base { /* ... */;
}

// ---- Sales_item.cc ----
#include "Sales_item.h"
namespace cplusplus_primer {
    // definitions for Sales_item members and overloaded operators
}

// ---- Query.cc ----
#include "Query.h"
namespace cplusplus_primer {
    // definitions for Query members and related functions
}
```

This program organization gives both the developers and users of our library the needed modularity. Each class is still organized into its own interface and implementation files. A user of one class need not compile names related to the others. We can hide the implementations from our users, while allowing the files `Sales_item.cc` and `user.cc` to be compiled and linked into one program without causing any compile-time or link-time error. Developers of the library can work independently on the implementation of each type.

这种程序组织给予开发者和库用户必要的模块性。每个类仍组织在自己的接口和实现文件中，一个类的用户不必编译与其他类相关的名字。如果允许 `Sales_item.cc` 和 `user.cc` 文件编译和链接到一个程序而不会导致编译时错误和运行时错误，就可以对用户隐藏实现。库的开发者可以独立工作于每个类型的实现。

A program using our library would include whichever headers it needed. The names in those headers are defined inside the `cplusplus_primer` namespace:

使用我们的库的程序可以包含需要的头文件，那些头文件中的名字定义在命名空间 `cplusplus_primer` 内部：

```
// ---- user.cc----
// defines the cplusplus_primer::Sales_item class
#include "Sales_item.h"
int main()
{
    // ...
    cplusplus_primer::Sales_item trans1, trans2;
    // ...
    return 0;
}
```

Defining Namespace Members

定义命名空间成员

Functions defined inside a namespace may use the short form for names defined in the same namespace:

Section 17.2. Namespaces

在命名空间内部定义的函数可以使用同一命名空间中定义的名字的简写形式：

```
namespace cplusplus_primer {  
    // members defined inside the namespace may use unqualified names  
    std::istream&  
    operator>>(std::istream& in, Sales_item& s)  
    {  
        // ...  
    }  
}
```

It is also possible to define a namespace member outside its namespace definition. We do so in ways that are similar to defining class members outside a class: The namespace declaration of the name must be in scope, and the definition must specify the namespace to which the name belongs:

也可以在命名空间定义的外部定义命名空间成员，用类似于在类外部定义类成员的方式：名字的命名空间声明必须在作用域中，并且定义必须指定该名字所属的命名空间：

```
// namespace members defined outside the namespace must use qualified names  
cplusplus_primer::Sales_item  
cplusplus_primer::operator+(const Sales_item& lhs,  
                           const Sales_item& rhs)  
{  
    Sales_item ret(lhs);  
    // ...  
}
```

This definition should look similar to class member functions defined outside a class. The return type and function name are qualified by the namespace name. Once the fully qualified function name is seen, we are in the scope of the namespace. Thus, references to namespace members in the parameter list and the function body can use unqualified names to reference `Sales_item`.

这个定义看起来类似于定义在类外部的类成员函数，返回类型和函数名由命名空间名字限定。一旦看到完全限定的函数名，就处于命名空间的作用域中。因此，形参表和函数体中的命名空间成员引用可以使用非限定名引用 `Sales_item`。

Members May Not Be Defined in Unrelated Namespaces

不能在不相关的命名空间中定义成员

Although a namespace member can be defined outside its namespace definition, there are restrictions on where this definition can appear. Only namespaces enclosing the member declaration can contain its definition. For example, `operator+` could be defined in either the `cplusplus_primer` namespace or at global scope. It may not be defined in an unrelated namespace.

虽然可以在命名空间定义的外部定义命名空间成员，对这个定义可以出现的地方仍有些限制，只有包围成员声明的命名空间可以包含成员的定义。例如，`operator+`既可以定义在命名空间 `cplusplus_primer` 中，也可以定义在全局作用域中，但它不能定义在不相关的命名空间中。

The Global Namespace

全局命名空间

Names defined at global scope are declared outside any class, function, or namespace and are defined inside the **global namespace**. The global namespace is implicitly declared and exists in every program. Each file that defines entities at global scope adds those names to the global namespace.

定义在全局作用域的名字（在任意类、函数或命名空间外部声明的名字）是定义在[全局命名空间](#)中的。全局命名空间是隐式声明的，存在于每个程序中。在全局作用域定义实体的每个文件将那些名字加到全局命名空间。

The [scope operator](#) can be used to refer to members of the global namespace. Because the global namespace is implicit, it does not have a name; the notation

可以用[作用域操作符](#)引用全局命名空间的成员。因为全局命名空间是隐含的，它没有名字，所以记号

```
::member_name
```

refers to a member of the global namespace.

引用全局命名空间的成员。

Exercises Section 17.2.1

Exercise

- 17.13:** Define the bookstore exception classes described in [Section 17.1.7](#) (p. 697) as members of namespace named `Bookstore`.

定义第 17.7 节描述的书店异常类，作为名为 `Bookstore` 的命名空间的成员。

Exercise

- 17.14:** Define `Sales_item` and its operators inside the `Bookstore` namespace. Define the addition operator to throw an exception.

在命名空间 `Bookstore` 内部定义 `Sales_item` 及其操作符。定义加操作符抛出一个异常。

Exercise

- 17.15:** Write a program that uses the `Sales_item` addition operator and handles any exceptions. Make this program a member of another namespace named `MyApp`. This program should use the exception classes defined in the `Bookstore` namespace by the previous exercise.

编写一个程序，使用 `Sales_item` 加操作符并处理任何异常。使这个程序成为名为 `MyApp` 的另一命名空间的成员。这个程序应使用上题中在命名空间 `Bookstore` 中定义的异常类。

17.2.2. Nested Namespaces

17.2.2. 嵌套命名空间

A nested namespace is a nested scope; its scope is nested within the namespace that contains it. Names in nested namespaces follow the normal rules: Names declared in an enclosing namespace are hidden by declarations of the same name in a nested namespace. Names defined inside a nested namespace are local to that namespace. Code in the outer parts of the enclosing namespace may refer to a name in a nested namespace only through its qualified name.

一个嵌套命名空间即是一个嵌套作用域——其作用域嵌套在包含它的命名空间内部。嵌套命名空间中的名字遵循常规规则：外围命名空间中声明的名字被嵌套命名空间中同一名字的声明所屏蔽。嵌套命名空间内部定义的名字局部于该命名空间。外围命名空间之外的代码只能通过限定名引用嵌套命名空间中的名字。

Nested namespaces can improve the organization of code in a library:

嵌套命名空间可以改进库中代码的组织：

```
namespace cplusplus_primer {
    // first nested namespace:
    // defines the Query portion of the library
    namespace QueryLib {
        class Query { /* ... */ };
        Query operator&(const Query&, const Query&);
        // ...
    }
    // second nested namespace:
    // defines the Sales_item portion of the library
    namespace Bookstore {
        class Item_base { /* ... */ };
        class Bulk_item : public Item_base { /* ... */ };
        // ...
    }
}
```

The `cplusplus_primer` namespace now contains two nested namespaces: the namespaces named `QueryLib` and `Bookstore`.

命名空间 `cplusplus_primer` 现在包含两个嵌套命名空间：名为 `QueryLib` 的命名空间和名为 `Bookstore` 的命名空间。

Nested namespaces are useful when a library provider needs to prevent names in each part of a library from colliding with names in other parts of the library.

当库提供者需要防止库中每个部分的名字与库中其他部分的名字冲突的时候，嵌套命名空间是很有用的。

The name of a member in a nested namespace is formed from the names of the enclosing namespace(s) and the name of the nested namespace. For example, the name of the class declared in the nested namespace `QueryLib` is

嵌套命名空间中成员的名字由外围命名空间的名字和嵌套命名空间的名字构成。例如，嵌套命名空间 `QueryLib` 中声明的类的名字是

```
cplusplus_primer::QueryLib::Query
```

Exercises Section 17.2.2

Exercise
17.16:

Organize the programs you have written to answer the questions in each chapter into its own namespace. That is, namespace `chapterrefinheriance` would contain code for the `Query` programs and `chapterrefalgs` would contain the `TextQuery` code. Using this structure, compile the `Query` code examples.

将为回答每章中的问题而编写的程序组织到每一章自己的命名空间中，也就是说，命名空间 `chapterrefinheriance` 将包含 `Query` 程序的代码，而 `chapterrefalgs` 将包含 `TextQuery` 代码。使用这个结构，编译 `Query` 代码示例。

Exercise
17.17:

Over the course of this primer, we defined two different classes named `Sales_item`: the initial simple class defined and used in [Part I](#), and the handle class defined in [Section 15.8.1](#) that interfaced to the `Item_base` inheritance hierarchy. Define two namespaces nested inside the `cplusplus_primer` namespace that could be used to distinguish these two class definitions.

在本书中，我们定义了两个名为 `Sales_item` 的不同类：在[第一部分](#)定义和使用的初始简单类，以及在[第 15.8.1 节](#)定义的与 `Item_base` 继承层次接口的句柄类。在命名空间 `cplusplus_primer` 内部定义两个嵌套命名空间，用于区别这两个类定义。

17.2.3. Unnamed Namespaces

17.2.3. 未命名的命名空间

A namespace may be unnamed. An [**unnamed namespace**](#) is a namespace that is defined without a name. An unnamed namespace begins with the keyword `namespace`. Following the `namespace` keyword is a block of declarations delimited by curly braces.

命名空间可以是未命名的，[未命名的命名空间](#)在定义时没有给定名字。未命名的命名空间以关键字 `namespace` 开头，接在关键字 `namespace` 后面的是由花括号定界的声明块。



Unnamed namespaces are not like other namespaces; the definition of an unnamed namespace is local to a particular file and never spans multiple text files.

未命名的命名空间与其他命名空间不同，未命名的命名空间的定义局部于特定文件，从不跨越多个文本文件。

An unnamed namespace may be discontiguous within a given file but does not span files. Each file has its own unnamed namespace.

未命名的命名空间可以在给定文件中不连续，但不能跨越文件，每个文件有自己的未命名的命名空间。

Unnamed namespaces are used to declare entities that are local to a file. Variables defined in an unnamed namespace are created when the program is started and exist until the program ends.

未命名的命名空间用于声明局部于文件的实体。在未命名的命名空间中定义的变量在程序开始时创建，在程序结束之前一直存在。

Names defined in an unnamed namespace are used directly; after all, there is no namespace name with which to qualify them. It is not possible to use the scope operator to refer to members of unnamed namespaces.

未命名的命名空间中定义的名字可直接使用，毕竟，没有命名空间名字来限定它们。不能使用作用域操作符来引用未命名的命名空间的成员。

Names defined in an unnamed namespace are visible only to the file containing the namespace. If another file contains an unnamed namespace, the namespaces are unrelated. Both unnamed namespaces could define the same name, and the definitions would refer to different entities.

未命名的命名空间中定义的名字只在包含该命名空间的文件中可见。如果另一文件包含一个未命名的命名空间，两个命名空间不相关。两个命名空间可以定义相同的名字，而这些定义将引用不同的实体。

Names defined in an unnamed namespace are found in the same scope as the scope at which the namespace is defined. If an unnamed namespace is defined at the outermost scope in the file, then names in the unnamed namespace must differ from names defined at global scope:

Section 17.2. Namespaces

未命名空间中定义的名字可以在定义该命名空间所在的作用域中找到。如果在文件的最外层作用域中定义未命名的命名空间，那么，未命名的空间中的名字必须与全局作用域中定义的名字不同：

```
int i; // global declaration for i
namespace {
    int i;
}
// error: ambiguous defined globally and in an unnested, unnamed namespace
i = 10;
```

An unnamed namespace, like any other namespace, may be nested inside another namespace. If the unnamed namespace is nested, then names in it are accessed in the normal way, using the enclosing namespace name(s):

像任意其他命名空间一样，未命名的命名空间也可以嵌套在另一命名空间内部。如果未命名的命名空间是嵌套的，其中的名字按常规方法使用外围命名空间名字访问：

```
namespace local {
    namespace {
        int i;
    }
}
// ok: i defined in a nested unnamed namespace is distinct from global i
local::i = 42;
```



If a header defines an unnamed namespace then the names in that namespace will define different local entities in each file that includes the header.

如果头文件定义了未命名的命名空间，那么，在每个包含该头文件的文件中，该命名空间中的名字将定义不同的局部实体。

In all other ways, the members of an unnamed namespace are normal program entities.

在所有其他方式中，未命名的命名空间的成员都是普通程序实体。

Unnamed Namespaces Replace File Statics

未命名的命名空间取代文件中的静态声明

Prior to the introduction of namespaces in standard C++, programs had to declare names as **static** to make them local to a file. The use of **file statics** is inherited from C. In C, a global entity declared **static** is invisible outside the file in which it is declared.

在标准 C++ 中引入命名空间之前，程序必须将名字声明为 **static**，使它们局部于一个文件。[文件中静态声明](#)的使用从 C 语言继承而来，在 C 语言中，声明为 **static** 的局部实体在声明它的文件之外不可见。



The use of file static declarations is deprecated by the C++ standard. A deprecated feature is one that may not be supported in future releases. File statics should be avoided and unnamed namespaces used instead.

C++ 不赞成文件静态声明。不造成的特征是在未来版本中可能不支持的特征。应该避免文件静态而使用未命名空间代替。

Exercises Section 17.2.3

Exercise 17.18: Why would you define your own namespace in your programs? When might you use an unnamed namespace?

为什么在程序中可以定义自己的命名空间？何时可以使用未命名空间？

Exercise 17.19: Suppose we have the following declaration of the `operator*` that is a member of the nested namespace `cplusplus_primer::MatrixLib`:

假定有下面的 `operator*` 的声明，`operator*` 是嵌套命名空间 `cplusplus_primer::MatrixLib` 的成员：

```
namespace cplusplus_primer {
    namespace MatrixLib {
        class matrix { /* ... */;
        matrix operator*
            (const matrix &, const matrix &);
        // ...
    }
}
```

How would you define this operator in global scope? Provide only the prototype for the operator's definition.

怎样在全局作用域中定义这个操作符？只需给出操作符定义的原型。

17.2.4. Using Namespace Members

17.2.4. 命名空间成员的使用

Referring to namespace members as `namespace_name::member_name` is admittedly cumbersome, especially if the namespace name is long. Fortunately, there are ways to make it easier to use namespace members. Our programs have used one of these ways, `using` declarations ([Section 3.1](#), p. 78). The others, namespace aliases and `using` directives, will be described in this section.

像命名空间名 `namespace_name::member_name` 成员名这样引用命名空间的成员无可否认是很麻烦，特别是，命名空间名字很长的时候。幸好，有办法让使用命名空间成员比较简单。我们的程序已经使用了其中的一种方法，就是 `using` ([第 3.1 节](#))，本节将介绍其他方法：命名空间别名和 `using` 指示。



Header files should not contain `using` directives or `using` declarations except inside functions or other scopes. A header that includes a `using` directive or declaration at its top level scope has the effect of injecting that name into the file that includes the header. Headers should define only the names that are part of its interface, not names used in its own implementation.

除了在函数或其他作用域内部，头文件不应该包含 `using` 指示或 `using` 声明。在其顶级作用域包含 `using` 指示或 `using` 声明的头文件，具有将该名字注入包含该头文件的文件中的效果。头文件应该只定义作为其接口的一部分的名字，不要定义在其实现中使用的名字。

`using` Declarations, a Recap

`using` 声明，扼要重述

The programs in this book that use names from the standard library generally assume that an appropriate [`using declaration`](#) has been made:

本书中使用标准库中名字的程序一般假设进行了适当的 [`using` 声明](#)：

```
map<string, vector< pair<size_t, size_t> > > word_map;
```

assumes that the following `using` declarations have been made:

Section 17.2. Namespaces

假定进行了下面的声明:

```
using std::map;
using std::pair;
using std::size_t;
using std::string;
using std::vector;
```

A `using` declaration introduces only one namespace member at a time. It allows us to be very specific regarding which names are used in our programs.

一个 `using` 声明一次只引入一个命名空间成员，它使得无论程序中使用哪些名字，都能够非常明确。

Scope of a `using` Declaration

`using` 声明的作用域

Names introduced in a `using` declaration obey normal scope rules. The name is visible from the point of the `using` declaration to the end of the scope in which the declaration is found. Entities with the same name defined in an outer scope are hidden.

`using` 声明中引入的名字遵循常规作用域规则。从 `using` 声明点开始，直到包含 `using` 声明的作用域的末尾，名字都是可见的。外部作用域中定义的同名实体被屏蔽。

The shorthand name may be used only within the scope in which it is declared and in scopes nested within that scope. Once the scope ends, the fully qualified name must be used.

简写名字只能在声明它的作用域及其嵌套作用域中使用，一旦该作用域结束了，就必须使用完全限定名。

A `using` declaration can appear in global, local, or namespace scope. A `using` declaration in class scope is limited to names defined in a base class of the class being defined.

`using` 声明可以出现在全局作用域、局部作用域或者命名空间作用域中。类作用域中的 `using` 声明局限于被定义类的基类中定义的名字。

Namespace Aliases

命名空间别名

A namespace alias can be used to associate a shorter synonym with a namespace name. For example, a long namespace name such as 可用命名空间别名将较短的同义词与命名空间名字相关联。例如，像

```
namespace cplusplus_primer { /* ... */ };
```

can be associated with a shorter synonym as follows:

这样的长命名空间名字，可以像下面这样与较短的同义词相关联:

```
namespace primer = cplusplus_primer;
```

A namespace alias declaration begins with the keyword `namespace`, followed by the (shorter) name of the namespace alias, followed by the `=` sign, followed by the original namespace name and a semicolon. It is an error if the original namespace name has not already been defined as a namespace.

命名空间别名声明以关键字 `namespace` 开头，接(较短的)命名空间别名名字，再接 `=`，再接原来的命名空间名字和分号。如果原来的命名空间名字是未定义的，就会出错。

A namespace alias can also refer to a nested namespace. Rather than writing

命名空间别名也可以引用嵌套的命名空间。除了编写

```
cplusplus_primer::QueryLib::Query tq;
```

we could define and use an alias for `cplusplus_primer::QueryLib`:

之外，我们可以定义和使用 `cplusplus_primer::QueryLib`: 的别名:

```
namespace Qlib = cplusplus_primer::QueryLib;
Qlib::Query tq;
```

Section 17.2. Namespaces



A namespace can have many synonyms, or aliases. All the aliases and the original namespace name can be used interchangeably.

一个命名空间可以有许多别名，所有别名以及原来的命名空间名字都可以互换使用。

using Directives

using 指示

Like a `using` declaration, a `using directive` allows us to use the shorthand form of a namespace name. Unlike a `using` declaration, we retain no control over which names are made visible they all are.

像 `using` 声明一样，`using` 指示使我们能够使用命名空间名字的简写形式。与 `using` 声明不同，`using` 指示无法控制使得哪些名字可见——它们都是可见的。

The Form of a `using` Directive

using 指示的形式

A `using` directive begins with the keyword `using`, followed by the keyword `namespace`, followed by a namespace name. It is an error if the name is not a previously defined namespace name.

`using` 指示以关键字 `using` 开头，后接关键字 `namespace`，再接命名空间名字。如果该名字不是已经定义的命名空间名字，就会出错。

A `using` directive makes all the names from a specific namespace visible without qualification. The short form names can be used from the point of the `using` directive to the end of the scope in which the `using` directive appears.

`using` 指示使得特定命名空间所有名字可见，没有限制。短格式名字可从 `using` 指示点开始使用，直到出现 `using` 指示的作用域的末尾。

A `using` directive may appear in namespace, function, or block scope. It may not appear in a class scope.

`using` 指示使得特定命名空间的所有名字可见，没有限制。短格式名字可从 `using` 指示点开始使用，直到出现 `using` 指示的作用域的末尾。



It can be tempting to write programs with `using` directives, but doing so reintroduces all the problems inherent in name collisions when using multiple libraries.

可以尝试用 `using` 指示编写程序，但在使用多个库的时候，这样做会重新引入名字冲突的所有问题。

using Directives and Scope

using 指示与作用域

The scope of names introduced by a `using` directive is more complicated than those for `using` declarations. A `using` declaration puts the name directly in the same scope in which the `using` declaration itself appears. It is as if the `using` declaration is a local alias for the namespace member. Because the declaration is localized, the chance of collisions is minimized.

用 `using` 指示引入的名字的作用域比 `using` 声明的更复杂。`using` 声明将名字直接放入出现 `using` 声明的作用域，好像 `using` 声明是命名空间成员的局部别名一样。因为这种声明是局部化的，冲突的机会最小。



A `using` directive does not declare local aliases for the namespace member names. Rather, it has the effect of lifting the namespace members into the nearest scope that contains both the namespace itself and the `using` directive.

Section 17.2. Namespaces

`using` 指示不声明命名空间成员名字的别名，相反，它具有将命名空间成员提升到包含命名空间本身和 `using` 指示的最近作用域的效果。

In the simplest case, assume we have a namespace `A` and a function `f`, both defined at global scope. If `f` has a `using` directive for `A`, then in `f` it will be as if the names in `A` appeared in the global scope prior to the definition of `f`:

在最简单的情况下，假定有命名空间 `A` 和函数 `f`，二者都在全局作用域中定义。如果 `f` 有关于 `A` 的 `using` 指示，那么，在 `f` 中，将好像 `A` 中的名字出现在全局作用域中 `f` 的定义之前一样：

```
// namespace A and function f are defined at global scope
namespace A {
    int i, j;
}
void f()
{
    using namespace A;      // injects names from A into the global scope
    cout << i * j << endl; // uses i and j from namespace A
    //...
}
```



One place where `using` directives are useful is in the implementation files for the namespace itself.

`using` 指示有用的一种情况是，用在命名空间本身的实现文件中。

`using` Directives Example

`using` 指示例子

Let's look at an example:

看一个例子：

```
namespace blip {
    int bi = 16, bj = 15, bk = 23;
    // other declarations
}
int bj = 0; // ok: bj inside blip is hidden inside a namespace
void manip()
{
    // using directive - names in blip "added" to global scope
    using namespace blip;
        // clash between :bj and blip::bj
        // detected only if bj is used
    ++bi;           // sets blip::bi to 17
    ++bj;           // error: ambiguous
                    // global bj or blip::bj?
    ++::bj;         // ok: sets global bj to 1
    ++blip::bj;     // ok: sets blip::bj to 16
    int bk = 97;   // local bk hides blip::bk
    ++bk;           // sets local bk to 98
}
```

The `using` directive in `manip` makes all the names in `blip` directly accessible to `manip`: The function can refer to the names of these members, using their short form.

`manip` 中的 `using` 提示使 `manip` 能够直接访问 `blip` 中的所有名字：使用它们的简化形式，该函数可以引用这些成员的名字。

The members of `blip` appear as if they were defined in the scope in which both `blip` and `manip` are defined. Given that `blip` is defined at global scope, then the members of `blip` appear as if they were declared in global scope. Because the names are in different scopes, local declarations within `manip` may hide some of the namespace member names. The local variable `bk` hides the namespace member `blip::bk`. Referring to `bk` within

Section 17.2. Namespaces

`manip` is not ambiguous; it refers to the local variable `bk`.

`blip` 的成员看来好像是在定义 `blip` 和 `manip` 的作用域中定义的一样。如果在全局作用域中定义 `blip`, 则 `blip` 的成员看来好像是声明在全局作用域的一样。因为名字在不同的作用域中, `manip` 内部的局部声明可以屏蔽命名空间的某些成员名字, 局部变量 `bk` 屏蔽命名空间名字 `blip::bk`, 在 `manip` 内部对 `bk` 的引用没有二义性, 它引用局部变量 `bk`。

It is possible for names in the namespace to conflict with other names defined in the enclosing scope. For example, the `blip` member `bj` appears to `manip` as if it were declared at global scope. However, there is another object named `bj` in global scope. Such conflicts are permitted; but to use the name, we must explicitly indicate which version is wanted. Therefore, the use of `bj` within `manip` is ambiguous: The name refers both to the global variable and to the member of namespace `blip`.

命名空间中的名字可能会与外围作用域中定义的其他名字冲突。例如, 对 `manip` 而言, `blip` 成员 `bj` 看来好像声明在全局作用域中, 但是, 全局作用域存在另一名为 `bj` 的对象。这种冲突是允许的, 但为了使用该名字, 必须显式指出想要的是哪个版本, 因此, 在 `manip` 内部的 `bj` 使用是有二义性的: 该名字既可引用全局变量又可引用命名空间 `blip` 的成员。

To use a name such as `bj`, we must use the scope operator to indicate which name is wanted. We would write `::bj` to obtain the variable defined in global scope. To use the `bj` defined in `blip`, we must use its qualified name, `blip::bj`.

为了使用像 `bj` 这样的名字, 必须使用作用域操作符指出想要的是哪个名字。可以编写 `::bj` 来获得在全局作用域中定义的变量, 要使用 `blip` 中定义的 `bj`, 必须使用它的限定名字 `blip::bj`。

Exercises Section 17.2.4

Exercise 17.20: Explain the differences between `using` declarations and `using` directives.

解释 `using` 声明和 `using` 指示之间的区别。

Exercise 17.21: Consider the following code sample:

考虑下面代码样本:

```
namespace Exercise {
    int ivar = 0;
    double dvar = 0;
    const int limit = 1000;
}
int ivar = 0;
// position 1
void manip() {
    // position 2
    double dvar = 3.1416;
    int iobj = limit + 1;
    ++ivar;
    ++::ivar;
}
```

What are the effects of the declarations and expressions in this code sample if `using` declarations for all the members of namespace `Exercise` are located at the location labeled *position 1*? At *position 2* instead? Now answer the same question but replace the `using` declarations with a `using` directive for namespace `Exercise`.

如果命名空间 `Exercise` 的所有成员的 `using` 声明放在标为 *position 1* 的地方, 这个代码样本中的声明和表达式的效果是什么? 如果放在 *position 2* 位置呢? 用命名空间 `Exercise` 的 `using` 指示代替 `using` 声明, 回答同一问题。

Caution: Avoid `using` Directives

警告: 避免 `using` 指示

`using directives`, which inject all the names from a namespace, are deceptively simple to use: With only a single statement, all the member names of a namespace are suddenly visible. Although this approach may seem simple, it can introduce its own problems. If an application uses many libraries, and if the names within these libraries are made

Section 17.2. Namespaces

visible with `using` directives, then we are back to square one, and the global namespace pollution problem reappears.

`using` 指示注入来自一个命名空间的所有名字，它的使用是靠不住的：只用一个语句，命名空间的所有成员名就突然可见了。虽然这个方法看似简单，但也有它自身的问题。如果应用程序使用许多库，并且用 `using` 指示使得这些库中的名字可见，那么，全局命名空间污染问题就重新出现。

Moreover, it is possible that a working program will fail to compile when a new version of the library is introduced. This problem can arise if a new version introduces a name that conflicts with a name that the application is using.

而且，当引入库的新版本的时候，正在工作的程序可能会编译失败。如果新版本引入一个与应用程序正在使用的名字冲突的名字，就会引发这个问题。

Another problem is that ambiguity errors caused by `using` directives are detected only at the point of use. This late detection means that conflicts can arise long after introducing a particular library. If the program begins using a new part of the library, previously undetected collisions may arise.

另一个问题是，由 `using` 指示引起的二义性错误只能在使用处检测，这个后来的检测意味着，可能在特定库引入很久之后才引发冲突，如果程序开始使用该库的新部分，就可能引发先前未检测到的冲突。

Rather than relying on a `using` directive, it is better to use a `using` declaration for each namespace name used in the program. Doing so reduces the number of names injected into the namespace. Ambiguity errors caused by `using` declarations are detected at the point of declaration, not use, and so are easier to find and fix.

相对于依赖于 `using` 指示，对程序中使用的每个命名空间名字使用 `using` 声明更好，这样做减少注入到命名空间中的名字数目，由 `using` 声明引起的二义性错误在声明点而不是使用点检测，因此更容易发现和修正。

17.2.5. Classes, Namespaces, and Scope

17.2.5. 类、命名空间和作用域

As we've noted, namespaces are scopes. As in any other scope, names are visible from the point of their declaration. Names remain visible through any nested scopes until the end of the block in which they were introduced.

正如我们已经注意到的，命名空间是作用域。像在任意其他作用域中一样，名字从声明点开始可见。名字的可见性穿过任意嵌套作用域，直到引入名字的块的末尾。

Name lookup for names used inside a namespace follows the normal C++ lookup rules: When looking for a name, we look outward through the enclosing scopes. An enclosing scope for a name used inside a namespace might be one or more nested namespaces ending finally with the all-encompassing global namespace. Only names that have been declared before the point of use that are in blocks that are still open are considered:

对命名空间内部使用的名字的查找遵循常规 C++ 查找规则：当查找名字的时候，通过外围作用域外查找。对命名空间内部使用的名字而言，外围作用域可能是一个或多个嵌套的命名空间，最终以全包围的全局命名空间结束。只考虑已经在使用点之前声明的名字，而该使用仍在开放的块中：

```
namespace A {
    int i;
    namespace B {
        int i;      // hides A::i within B
        int j;
        int f1()
        {
            int j;  // j is local to f1 and hides A::B::j
            return i; // returns B::i
        }
    } // namespace B is closed and names in it are no longer visible
    int f2()
    {
        return j;  // error: j is not defined
    }
    int j = i;    // initialized from A::i
}
```

Names used in a class member definition are resolved in much the same way, with one important difference: If the name is not local to the member function, we first try to resolve the name to a class member before looking in the outer scopes.

用非常相似的方式确定类成员定义中使用的名字，只有一个重要区别：如果名字不是局部于成员函数的，就试着在查找更外层作用域之前在类成员中确定名字。

As we saw in [Section 12.3](#) (p. 444), members defined inside a class may use names that appear textually after the definition. For example, a constructor defined inside the class body may initialize the data members even if the declaration of those members appears after the constructor definition. When a name is used in a class scope, we look first in the member itself, then in the class, including any base classes. Only after exhausting the class(es) do we examine the enclosing scopes. When a class is wrapped in a namespace, the same lookup happens: Look first in the member, then the class (including base classes), then look in the enclosing scopes, one or more of which might be a namespace:

正如[第 12.3 节](#)所介绍的，类内部所定义的成员可以使用出现在定义文本之后的名字。例如，即使数据成员的定义出现在构造函数定义之后，类定义体内部定义的构造函数也可以初始化那些数据成员。当在类作用域中使用名字的时候，首先在成员本身中查找，然后在类中查找，包括任意基类，只有在查找完类之后，才检查外围作用域。当类包在命名空间中的时候，发生相同的查找：首先在成员中找，然后在类（包括基类）中找，再在包围作用域中找，包围作用域中的一个或多个可以是命名空间：

```
namespace A {
```

Section 17.2. Namespaces

```
int i;
int k;
class C1 {
public:
    C1(): i(0), j(0) { } // ok: initializes C1::i and C1::j
    int f1()
    {
        return k; // returns A::k
    }
    int f2()
    {
        return h; // error: h is not defined
    }
    int f3();
private:
    int i; // hides A::i within C1
    int j;
};
int h = i; // initialized from A::i
}
// member f3 is defined outside class C1 and outside namespace A
int A::C1::f3()
{
    return h; // ok: returns A::h
}
```

With the exception of member definitions, scopes are always searched upward: A name must be declared before it can be used. Hence, the `return` in `f2` will not compile. It attempts to reference the name `h` from namespace `A`, but `h` has not yet been defined. Had that name been defined in `A` before the definition of `C1`, the use of `h` would be legal. Similarly, the use of `h` inside `f3` is okay, because `f3` is defined after `A::h` has been defined.

除了成员定义例外，总是向上查找作用域：名字在使用之前必须声明。因此，`f2` 中的 `return` 语句将不能编译，它试图引用命名空间 `A` 中的名字 `h`，但 `h` 还没有定义。如果使 `A` 中的名字在 `C1` 的定义之前定义，`h` 的使用就是合法的。类似地，`f3` 内部对 `h` 的使用是正确的，因为 `f3` 定义在已经定义了 `A::h` 之后。



The order in which scopes are examined to find a name can be inferred from the qualified name of a function. The qualified name indicates, in reverse order, the scopes that are searched.

可以从函数的限定名推断出查找名字时所检查作用域的次序，限定名以相反次序指出被查找的作用域。

The qualifiers `A::C1::f3` indicate the reverse order in which the class scopes and namespace scopes are to be searched. The first scope searched is that of the function `f3`. Then the class scope of its enclosing class `C1` is searched. The scope of the namespace `A` is searched last before the scope containing the definition of `f3` is examined.

限定符 `A::C1::f3` 指出了查找类作用域和命名空间作用域的相反次序，首先查找函数 `f3` 的作用域，然后查找外围类 `C1` 的作用域。在查找包含 `f3` 定义的作用之前，最后查找命名空间 `A` 的作用域。

Argument-Dependent Lookup and Class Type Parameters

实参相关的查找与类类型形参

Consider the following simple program:

考虑下面的简单程序：

```
std::string s;
// ok: calls std::getline(std::istream&, const std::string&)
getline(std::cin, s);
```

The program uses the `std::string` type, yet it refers without qualification to the `getline` function. Why can we use this function without a specific `std::` qualifier or a `using` declaration?

这段程序使用了 `std::string` 类型，但它不加限制地引用了 `getline` 函数。为什么可以无须特定 `std::` 限定符或 `using` 声明而使用该函数？

It turns out that there is an important exception to the rule that namespace names are hidden.

它给出了屏蔽命名空间名字规则的一个重要例外。

Functions, including overloaded operators, that take parameters of a class type (or pointer or reference to a class type), and that are defined in the same namespace as the class itself, are visible when an object of (or reference or pointer to) the class type is used as an argument.



接受类类型形参（或类类型指针及引用形参）的函数（包括重载操作符），以及与类本身定义在同一命名空间中的函数（包括重载操作符），在用类类型对象（或类类型的引用及指针）作为实参的时候是可见的。

When the compiler sees the use of the `getline` function

当编译器看到 `getline` 函数的使用

```
getline(std::cin, s);
```

it looks for a matching function in the current scope, the scopes enclosing the call to `getline`, and in the namespace(s) in which the type of `cin` and the `string` type are defined. Hence, it looks in the namespace `std` and finds the `getline` function defined by the `string` type.

的时候，它在当前作用域，包含调用的作用域以及定义 `cin` 的类型和 `string` 类型的命名空间中查找匹配的函数。因此，它在命名空间 `std` 中查找并找到由 `string` 类型定义的 `getline` 函数。

The reason that functions are made visible if they have a parameter of the class type is to allow nonmember functions that are conceptually part of a class' interface to be used without requiring a separate `using` declaration. Being able to use nonmember operations is particularly useful for operator functions.

如果函数具有类类型形参就使得函数可见，其原因在于，允许无须单独的 `using` 声明就可以使用概念上作为类接口组成部分的非成员函数。能够使用非成员操作对操作符函数特别有用。

For example, consider the following simple program:

例如，考虑下面的简单程序：

```
std::string s;
cin >> s;
```

In absence of this exception to the lookup rules, we would have to write either:

如果没有查找规则的这个例外，我们将必须编写下面二者之一：

```
using std::operator>>; // need to allow cin >> s
std::operator>>(std::cin, s); // ok: explicitly use std::>>
```

Either of these declarations is awkward and would make simple uses of `strings` and the IO library more complicated.

这两个声明都不方便使用，而且可能使 `string` 和 IO 库的使用变得更复杂。

Implicit Friend Declarations and Namespaces

隐式友元声明与命名空间

Recall that when a class declares a friend function ([Section 12.5](#), p. 465), a declaration for the function need not be visible. If there isn't a declaration already visible, then the friend declaration has the effect of putting a declaration for that function or class into the surrounding scope. If a class is defined inside a namespace, then an otherwise undeclared friend function is declared in the same namespace:

回忆一下，当一个类声明友元函数（[第 12.5 节](#)）的时候，函数的声明不必是可见的。如果不存在可见的声明，那么，友元声明具有将该函数或类的声明放入外围作用域的效果。如果类在命名空间内部定义，则没有另外声明的友元函数在同一命名空间中声明。

```
namespace A {
    class C {
        friend void f(const C&); // makes f a member of namespace A
    };
}
```

Section 17.2. Namespaces

Because the friend takes an argument of a class type and is implicitly declared in the same namespace as the class, it can be used without using an explicit name-space qualifier:

```
// f2 defined at global scope
void f2()
{
    A::C cobj;
    f(cobj); // calls A::f
}
```

17.2.6. Overloading and Namespaces

17.2.6. 重载与命名空间

As we've seen, each namespace maintains its own scope. As a consequence, functions that are members of two distinct namespaces do not overload one another. However, a given namespace can contain a set of overloaded function members.

正如我们所见，每个命名空间维持自己的作用域，因此，作为两个不同命名空间的成员的函数不能互相重载。但是，给定命名空间可以包含一组重载函数成员。

In general, function matching ([Section 7.8.2](#), p. 269) within a namespace happens in the same manner as we've already seen:

一般而言，命名空间内部的函数匹配 ([第 7.8.2 节](#)) 以与我们已经见过的方式相同的方式进行：

1. Find the set of candidate functions. A function is a candidate if a declaration for it is visible at the time of the call and if it has the same name as the called function.

找到候选函数集。如果一个函数在调用时其声明可见并且与被调用函数同名，这个函数就是候选者。

2. Select the viable functions from the set of candidates. A function is viable if it has the same number of parameters as the call has arguments and if each parameter could be matched by the corresponding argument.

从候选集中选择可行函数。如果函数的形参数目与函数调用的实参数目相同，并且每个形参都可用对应实参匹配，这个函数就是可行的。

3. Select the single best match from the viable set and generate code to call that function. If the viable set is empty, then the call is in error, having no match. If the viable set is nonempty and there is no best match, then the call is ambiguous.

从可行集合中选择一个最佳匹配，并产生代码调用该函数。如果可行集合为空，则调用出错，没有匹配；如果可行集合非空且没有最佳匹配，则调用有二义性。

Candidate Functions and Namespaces

候选函数与命名空间

Namespaces can have two impacts on function matching. One of these should be obvious: A `using` declaration or directive can add functions to the candidate set. The other is much more subtle.

命名空间对函数匹配有两个影响。一个影响是明显的：`using` 声明或 `using` 指示可以将函数加到候选集合。另一个影响则微妙得多。

As we saw in the previous section, name lookup for functions that have one or more class-type parameters includes the namespace in which each parameter's class is defined. This rule also impacts how we determine the candidate set. Each namespace that defines a class used as a parameter (and those that define its base class(es)) is searched for candidate functions. Any functions in those namespaces that have the same name as the called function are added to the candidate set. These functions are added even though they otherwise are not visible at the point of the call. Functions with the matching name in those namespaces are added to the candidate set:

正如前节所见，有一个或多个类类型形参的函数的名字查找包括定义每个形参类型的命名空间。这个规则还影响怎样确定候选集合，为找候选函数而查找定义形参类（以及定义其基类）的每个命名空间，将那些命名空间中任意与被调用函数名字相同的函数加入候选集合。即使这些函数在调用点不可见，也将之加入候选集合。将那些命名空间中带有匹配名字的函数加入候选集合：

```
namespace NS {
    class Item_base { /* ... */ };
    void display(const Item_base&);
}

// Bulk_item's base class is declared in namespace NS
class Bulk_item : public NS::Item_base {};
int main() {
    Bulk_item book1;
    display(book1);
    return 0;
}
```

Section 17.2. Namespaces

}

The argument, `book1`, to the `display` function has class type `Bulk_item`. The candidate functions for the call to `display` are not only the functions with declarations that are visible where the function `display` is called, but also the functions in the namespace where the class `Bulk_item` and its base class `Item_base` are declared. The function `display(const Item_base&)` declared in namespace `NS` is added to the set of candidate functions.

`display` 函数的实参 `book1` 具有类类型 `Bulk_item`。`display` 调用的候选函数不仅是在调用 `display` 函数的地方其声明可见的函数，还包括声明 `Bulk_item` 类及其基类 `Item_base` 的命名空间中的函数。命名空间 `NS` 中声明的函数 `display(const Item_base&)` 被加到候选函数集合中。

Overloading and `using` Declarations

重载与 `using` 声明

A `using` declaration declares a name. As we saw in [Section 15.5.3](#) (p. 592), there is no way to write a `using` declaration to refer to a specific function declaration:

`using` 声明声明一个名字。正如[第 15.5.3 节](#)所见，没有办法编写 `using` 声明来引用特定函数声明：

```
using NS::print(int); // error: cannot specify parameter list
using NS::print;      // ok: using declarations specify names only
```

If a function is overloaded within a namespace, then a `using` declaration for the name of that function declares *all* the functions with that name. If there are `print` functions for `int` and `double` in the namespace `NS`, then a `using` declaration for `NS::print` makes both functions visible in the current scope.

如果命名空间内部的函数是重载的，那么，该函数名字的 `using` 声明声明了所有具有该名字的函数。如果命名空间 `NS` 中有用于 `int` 和 `double` 的函数，则 `NS::print` 的 `using` 声明使得两个函数都在当前作用域中可见。

A `using` declaration incorporates all versions of an overloaded function to ensure that the interface of the namespace is not violated. The author of a library provided different functions for a reason. Allowing users to selectively ignore some but not all of the functions from a set of overloaded functions could lead to surprising program behavior.

一个 `using` 声明包括重载函数的所有版本以保证不违反命名空间的接口。库作者为一个理由提供不同函数，允许用户选择性地忽略重载函数集合中的某些但不是全部函数，可能会导致奇怪的程序行为。

The functions introduced by a `using` declaration overload any other declarations of the functions with the same name already present in the scope where the `using` declaration appears.

由 `using` 声明引入的函数，重载出现 `using` 声明的作用域中的任意其他同名函数的声明。

If the `using` declaration introduces a function in a scope that already has a function of the same name with the same parameter list, then the `using` declaration is in error. Otherwise, the `using` declaration defines additional overloaded instances of the given name. The effect is to increase the set of candidate functions.

如果 `using` 声明在已经有同名且带相同形参表的函数的作用域中引入函数，则 `using` 声明出错，否则，`using` 定义给定名字的另一重载实例，效果是增大候选函数集合。

Overloading and `using` Directives

重载与 `using` 指示

A `using` directive lifts the namespace members into the enclosing scope. If a namespace function has the same name as a function declared in the scope at which the namespace is placed, then the namespace member is added to the overload set:

`using` 指示将命名空间成员提升到外围作用域。如果命名空间函数与命名空间所在的作用域中声明的函数同名，就将命名空间成员加到重载集合中：

```
namespace libs_R_us {
    extern void print(int);
    extern void print(double);
}
void print(const std::string &);
// using directive:
using namespace libs_R_us;
// using directive added names to the candidate set for calls to print:
// print(int) from libs_R_us
// print(double) from libs_R_us
```

Section 17.2. Namespaces

```
// print(const std::string &) declared explicitly
void fooBar(int ival)
{
    print("Value: "); // calls global print(const string &)
    print(ival);      // calls libs_R_us::print(int)
}
```

Overloading across Multiple `using` Directives

跨越多个 `using` 指示的重载

If many `using` directives are present, then the names from each namespace become part of the candidate set:

如果存在许多 `using` 指示，则来自每个命名空间的名字成为候选集合的组成部分：

```
namespace AW {
    int print(int);
}
namespace Primer {
    double print(double);
}
// using directives:
// form an overload set of functions from different namespaces
using namespace AW;
using namespace Primer;
long double print(long double);
int main() {
    print(1);    // calls AW::print(int)
    print(3.1); // calls Primer::print(double)
    return 0;
}
```

The overload set for the function `print` in global scope contains the functions `print(int)`, `print(double)`, and `print(long double)`. These functions are all part of the overload set considered for the function calls in `main`, even though these functions were originally declared in different namespace scopes.

全局作用域中 `print` 函数的重载集合包含函数 `print(int)`、`print(double)` 和 `print(long double)`，即使这些函数原来在不同的命名空间中声明，它们都是为 `main` 中函数调用考虑的重载集合的组成部分。

Exercises Section 17.2.6

Exercise 17.22: Given the following code, determine which function, if any, matches the call to `compute`. List the candidate and viable functions. What type conversion sequence, if any, is applied to the argument to match the parameter in each viable function?

给定下面的代码，如果有，确定哪个函数与 `compute` 函数的调用匹配，列出候选函数与可行函数。如果有，对实参应用什么类型转换序列，以匹配每个可行函数的形参？

```
namespace primerLib {
    void compute();
    void compute(const void *);
}
using primerLib::compute;
void compute(int);
void compute(double, double = 3.4);
void compute(char*, char* = 0);

int main()
{
    compute(0);
    return 0;
}
```

What would happen if the `using` declaration were located in `main` before the call to `compute`? Answer the same questions as before.

如果 `main` 中的 `using` 声明放在 `compute` 调用之前，会发生什么情况？回答与前面相同的问题。

17.2.7. Namespaces and Templates

17.2.7. 命名空间与模板

Declaring a template within a namespace impacts how template specializations ([Section 16.6](#), p. 671) are declared: An explicit specialization of a template must be declared in the namespace in which the generic template is defined. Otherwise, the specialization would have a different name than the template it specialized.

在命名空间内部声明模板影响着怎样声明模板特化（[第 16.6 节](#)）：模板的显式特化必须在定义通用模板的命名空间中声明，否则，该特化将与它所特化的模板不同名。

There are two ways to define a specialization: One is to reopen the namespace and add the definition of the specialization, which we can do because namespace definitions are discontiguous. Alternatively, we could define the specialization in the same way that we can define any namespace member outside its namespace definition: by defining the specialization using the template name qualified by the name of the namespace.

有两种定义特化的方式：一种是重新打开命名空间并加入特化的定义，可以这样做是因为命名空间定义是不连续的；或者，可以用与在命名空间外部定义命名空间成员相同的方式来定义特化：使用由命名空间名字限定的模板名定义特化。



To provide our own specializations of templates defined in a namespace, we must ensure that the specialization definition is defined as being in the namespace containing the original template definition.

为了提供命名空间中所定义模板的自己的特化，必须保证在包含原始模板定义的命名空间中定义特化。

17.3. Multiple and Virtual Inheritance

17.3. 多重继承与虚继承

Most C++ applications use `public` inheritance from a single base class. In some cases, however, single inheritance is inadequate, either because it fails to model the problem domain or the model it imposes is unnecessarily complex.

大多数应用程序使用单个基类的公用继承，但是，在某些情况下，单继承是不够用的，因为可能无法为问题域建模，或者会对模型带来不必要的复杂性。

In these cases, **multiple inheritance** may model the application more directly. Multiple inheritance is the ability to derive a class from more than one immediate base class. A multiply derived class inherits the properties of all its parents. Although simple in concept, the details of intertwining multiple base classes can present tricky design-level and implementation-level problems.

在这些情况下，**多重继承**可以更直接地为应用程序建模。多重继承是从多于一个直接基类派生类的能力，多重继承的派生类继承其所有父类的属性。尽管概念简单，缠绕多个基类的细节可能会带来错综复杂的设计问题或实现问题。

17.3.1. Multiple Inheritance

17.3.1. 多重继承

This section uses a pedagogical example of a zoo animal hierarchy. Our zoo animals exist at different levels of abstraction. There are the individual animals, distinguished by their names, such as `Ling-ling`, `Mowgli`, and `Balou`. Each animal belongs to a species; `Ling-Ling`, for example, is a giant panda. Species, in turn, are members of families. A giant panda is a member of the bear family. Each family, in turn, is a member of the animal kingdom in this case, the more limited kingdom of a particular zoo.

本节使用动物园动物层次的一个教学例子。动物园动物在不同抽象级别存在，有个体的动物，由名字区分，如 `Ling-line`、`Mowgli` 和 `Balou`；每个动物属于一个特种，例如，`Ling-Ling` 是一个大熊猫；物种又是科的成员，大熊猫是熊科的成员；每个科又是动物界的成员——在这个例子中，比较受限的是一个特定的动物园。

Each level of abstraction contains data and operations that support a wider category of users. We'll define an abstract `ZooAnimal` class to hold information that is common to all the zoo animals and provides the public interface. The `Bear` class will contain information that is unique to the `Bear` family, and so on.

每个抽象级别包含支持广泛用户的数据和操作。我们将定义一个抽象 `ZooAnimal` 类保存所有动物园动物公共信息并提供公用接口，`Bear` 类将包含 `Bear` 科的独特信息，以此类推。

In addition to the actual zoo-animal classes, there are auxiliary classes that encapsulate various abstractions such as endangered animals. In our implementation of a `Panda` class, for example, a `Panda` is multiply derived from `Bear` and `Endangered`.

除了实际的动物园动物类之外，还有一些辅助类封装不同的抽象，如濒临灭绝的动物。昭，在 `Panda` 类的实现中，`Panda` 同时从 `Bear` 和 `Endangered` 派生。

Defining Multiple Classes

定义多个类

To support multiple inheritance, the derivation list

为了支持多重继承，扩充派生列表

```
class Bear : public ZooAnimal {  
};
```

is extended to support a comma-separated list of base classes:

以支持由逗号分隔的基类列表：

```
class Panda : public Bear, public Endangered {  
};
```

Section 17.3. Multiple and Virtual Inheritance

The derived class specifies (either explicitly or implicitly) the access level `public`, `protected`, or `private` for each of its base classes. As with single inheritance, a class may be used as a base class under multiple inheritance only after it has been defined. There is no language-imposed limit on the number of base classes from which a class can be derived. A base class may appear only once in a given derivation list.

派生类为每个基类（显式或隐式地）指定了访问级别——`public`、`protected` 或 `private`。像单继承一样，只有在定义之后，类才可以用作多重继承的基类。对于类可以继承的基类的数目，没有语言强加强加的限制，但在一个给定派生列表中，一个基类只能出现一次。

Multiply Derived Classes Inherit State from Each Base Class

多重继承的派生类从每个基类中继承状态

Under multiple inheritance, objects of a derived class contain a base-class subobject ([Section 15.2.3](#), p. 565) for each of its base classes. When we write

在多重继承下，派生类的对象包含每个基类的基类子对象（[第 15.2.3 节](#)）。当我们编写

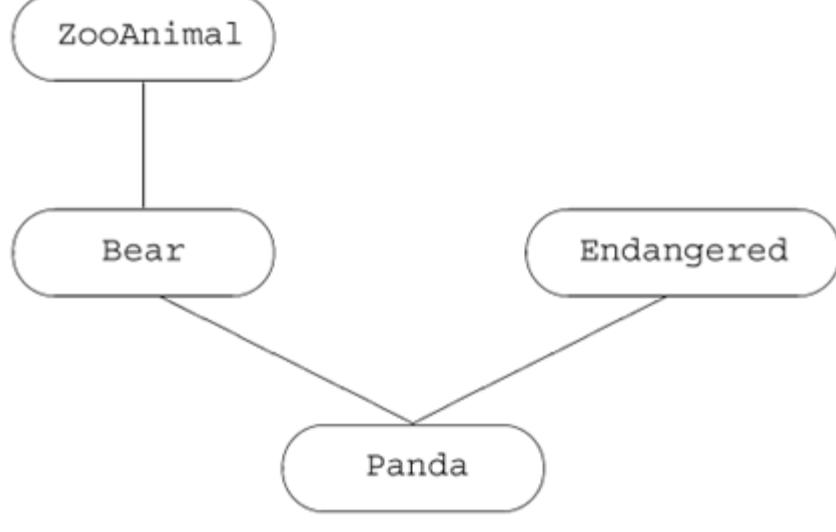
```
Panda ying_yang("ying_yang");
```

the object `ying_yang` is composed of a `Bear` class subobject (which itself contains a `ZooAnimal` base-class subobject), an `Endangered` class subobject, and the non`static` data members, if any, declared within the `Panda` class (see [Figure 17.2](#)).

的时候，对象 `ying_yang` 包含一个 `Bear` 类子对象（`Bear` 类子对象本身包含一个 `ZooAnimal` 基类子对象）、一个 `Endangered` 类子对象以及 `Panda` 类中声明的非 `static` 数据成员（如果有的话），见图 17.2。

Figure 17.2. Multiple Inheritance `Panda` Hierarchy

17.2. 多重继承的 `Panda` 层次



Derived Constructors Initialize All Base Classes

派生类构造函数初始化所有基类

Constructing an object of derived type involves constructing and initializing all its base subobjects. As is the case for inheriting from a single base class ([Section 15.4.1](#), p. 580), derived constructors may pass values to zero or more of their base classes in the constructor initializer:

构造派生类型的对象包括构造和初始化所有基类子对象。像继承单个基类（[第 15.4.1 节](#)）的情况一样，派生类的构造函数可以在构造函数初始化式中给零个或多个基类传递值：

```
// explicitly initialize both base classes
Panda::Panda(std::string name, bool onExhibit)
    : Bear(name, onExhibit, "Panda"),
```

Section 17.3. Multiple and Virtual Inheritance

```
Endangered(Endangered::critical) { }
// implicitly use Bear default constructor to initialize the Bear subobject
Panda::Panda()
: Endangered(Endangered::critical) { }
```

Order of Construction

构造的次序

The constructor initializer controls only the values that are used to initialize the base classes, not the order in which the base classes are constructed. The base-class constructors are invoked in the order in which they appear in the class derivation list. For `Panda`, the order of base-class initialization is:

构造函数初始化式只能控制用于初始化基类的值，不能控制基类的构造次序。基类构造函数按照基类构造函数在类派生列表中的出现次序调用。对 `Panda` 而言，基类初始化的次序是：

1. `ZooAnimal`, the ultimate base class up the hierarchy from `Panda`'s immediate base class `Bear`

`ZooAnimal`, 从 `Panda` 的直接基类 `Bear` 沿层次向上的最终基类。

2. `Bear`, the first immediate base class

`Bear`, 第一个直接基类。

3. `Endangered`, the second immediate base, which itself has no base class

`Endangered`, 第二个直接基类，它本身没有基类。

4. `Panda`; the members of `Panda` itself are initialized, and then the body of its constructor is run.

`Panda`, 初始化 `Panda` 本身的成员，然后运行它的构造函数的函数体。



The order of constructor invocation is not affected by whether the base class appears in the constructor initializer list or the order in which base classes appear within that list.

构造函数调用次序既不受构造函数初始化列表中出现的基类的影响，也不受基类在构造函数初始化列表中的出现次序的影响。

For example, in `Panda`'s default constructor, the `Bear` default constructor is invoked implicitly; it does not appear in the constructor initializer list. Even so, `Bear`'s default constructor is invoked prior to the explicitly listed constructor of `Endangered`.

例如，在 `Panda` 类的默认构造函数中，隐式调用 `Bear` 类的默认构造函数，它不出现在构造函数初始化列表中，虽然如此，仍在显式列出的 `Endangered` 类构造函数之前调用 `Bear` 类的默认构造函数。

Order of Destruction

析构的次序

Destructors are always invoked in the reverse order from which the constructors are run. In our example, the order in which the destructors are called is `~Panda`, `~Endangered`, `~Bear`, `~ZooAnimal`.

总是按构造函数运行的逆序调用析构函数。在我们的例子中，调用析构函数的次序是 `~Panda`, `~Endangered`, `~Bear`, `~ZooAnimal`。

Exercises Section 17.3.1

Exercise 17.23: Which, if any, of the following declarations are in error. Explain why.

如果有，下面哪些声明是错误的。解释为什么。

```
(a) class CADVehicle : public CAD, Vehicle { ... };
(b) class DoublyLinkedList:
        public List, public List { ... };
```

Section 17.3. Multiple and Virtual Inheritance

```
(c) class iostream: public istream, public ostream { ... };
```

Exercise Given the following class hierarchy, in which each class defines a default constructor,

17.24: 给定下面的类层次，其中，每个类定义了一个默认构造函数。

```
class A { ... };
class B : public A { ... };
class C : public B { ... };
class X { ... };
class Y { ... };
class Z : public X, public Y { ... };
class MI : public C, public Z { ... };
```

what is the order of constructor execution for the following definition?

对于下面的定义，构造函数的执行次序是什么？

```
MI mi;
```

17.3.2. Conversions and Multiple Base Classes

17.3.2. 转换与多个基类

Under single inheritance, a pointer or a reference to a derived class can be converted automatically to a pointer or a reference to a base class. The same holds true with multiple inheritance. A pointer or reference to a derived class can be converted to a pointer or reference to any of its base classes. For example, a `Panda` pointer or reference could be converted to a pointer or a reference to `ZooAnimal`, `Bear`, or `Endangered`:

在单个基类情况下，派生类的指针或引用可以自动转换为基类的指针或引用，对于多重继承也是如此，派生类的指针或引用可以转换为其任意其类的指针或引用。例如，`Panda`指针或引用可以转换为 `ZooAnimal`、`Bear` 或 `Endangered` 的指针或引用。

```
// operations that take references to base classes of type Panda
void print(const Bear&);
void highlight(const Endangered&);
ostream& operator<<(ostream&, const ZooAnimal&);
Panda ying_yang("ying_yang"); // create a Panda object
print(ying_yang); // passes Panda as reference to Bear
highlight(ying_yang); // passes Panda as reference to Endangered
cout << ying_yang << endl; // passes Panda as reference to ZooAnimal
```

Under multiple inheritance, there is a greater possibility of encountering an ambiguous conversion. The compiler makes no attempt to distinguish between base classes in terms of a derived-class conversion. Converting to each base class is equally good. For example, if there was an overloaded version of `print`

在多重继承情况下，遇到二义性转换的可能性更大。编译器不会试图根据派生类转换来区别基类间的转换，转换到每个基类都一样好。例如，如果有 `print` 函数的重载版本：

```
void print(const Bear&);
void print(const Endangered&);
```

an unqualified invocation of `print` with a `Panda` object

通过 `Panda` 对象对 `print` 函数的未限定调用

```
Panda ying_yang("ying_yang");
print(ying_yang); // error: ambiguous
```

results in a compile-time error that the call is ambiguous.

导致一个编译时错误，指出该调用是二义性的。

Exercises Section 17.3.2

Exercise 17.25: Given the following class hierarchy, in which each class defines a default constructor,
给定下面的类层次，其中每个类定义了一个默认构造函数，

```
class X { ... };
class A { ... };
class B : public A { ... };
class C : private B { ... };
class D : public X, public C { ... };
```

which, if any, of the following conversions are not permitted?

如果有，下面转换中哪些是不允许的？

```
D *pd = new D;
(a) X *px = pd; (c) B *pb = pd;
(b) A *pa = pd; (d) C *pc = pd;
```

Virtual Functions under Multiple Inheritance

多重继承下的虚函数

To see how the virtual function mechanism is affected by multiple inheritance, let's assume that our classes define the virtual members listed in [Table 17.2](#).

为了看看多重继承怎样影响虚函数机制，假定我们的类定义表 [17.2](#) 中列出的虚成员。

Table 17.2. Virtual Function in the ZooAnimal/Endangered Classes

17.2. ZooAnimal/Endangered 类中的虚函数

Function	Class Defining Own Version
函数	定义自己版本的类
print	ZooAnimal::ZooAnimal Bear::Bear Endangered::Endangered Panda::Panda
highlight	Endangered::Endangered Panda::Panda
toes	Bear::Bear Panda::Panda
cuddle	Panda::Panda
destructor	ZooAnimal::ZooAnimal Endangered::Endangered

Lookup Based on Type of Pointer or Reference

基于指针类型或引用类型的查找

As with single inheritance, a pointer or reference to a base class can be used to access only members defined (or inherited) in the base. It cannot access members introduced in the derived class.

像单继承一样，用基类的指针或引用只能访问基类中定义（或继承）的成员，不能访问派生类中引入的成员。

When a class inherits from multiple base classes, there is no implied relationship between those base classes. Using a pointer to one base does not allow access to members of another base.

当一个类继承于多个基类的时候，那些基类之间没有隐含的关系，不允许使用一个基类的指针访问其他基类的成员。

As an example, we could use a pointer or reference to a `Bear`, `ZooAnimal`, `Endangered`, or `Panda` to access a `Panda` object. The type of the pointer we use determines which operations are accessible. If we use a `ZooAnimal` pointer, only the operations defined in that class are usable. The `Bear`-specific, `Panda`-specific, and `Endangered` portions of the `Panda` interface are inaccessible. Similarly, a `Bear` pointer or reference knows only about the `Bear` and `ZooAnimal` members; an `Endangered` pointer or reference is limited to the `Endangered` members:

作为例子，我们可以使用 `Bear`、`ZooAnimal`、`Endangered` 或 `Panda` 的指针或引用来访问 `Panda` 对象。所用指针的类型决定了可以访问哪些操作。如果使用 `ZooAnimal` 指针，只能使用 `ZooAnimal` 类中定义的操作，不能访问 `Panda` 接口的 `Bear` 特定、`Panda` 特定和 `Endangered` 部分。类似地，`Bear` 指针或引用只知道 `Bear` 和 `ZooAnimal` 成员，`Endangered` 指针或引用局限于 `Endangered` 成员：

```
Bear *pb = new Panda("ying_yang");
pb->print(cout); // ok: Panda::print(ostream&)
pb->cuddle();    // error: not part of Bear interface
pb->highlight(); // error: not part of Bear interface
delete pb;        // ok: Panda::~Panda()
```

If the `Panda` object had been assigned to a `ZooAnimal` pointer, this set of calls would resolve exactly the same way.

如果将 `Panda` 对象赋给 `ZooAnimal` 指针，这个调用集合将完全相同的方式确定。

When a `Panda` is used via an `Endangered` pointer or reference, the `Panda`-specific and `Bear` portions of the `Panda` interface are inaccessible:

在通过 `Endangered` 指针或引用使用 `Panda` 对象的时候，不能访问 `Panda` 接口的 `Panda` 特定的部分和 `Bear` 部分：

```
Endangered *pe = new Panda("ying_yang");
pe->print(cout); // ok: Panda::print(ostream&)
pe->toes();      // error: not part of Endangered interface
pe->cuddle();    // error: not part of Endangered interface
pe->highlight(); // ok: Endangered::highlight()
delete pe;        // ok: Panda::~Panda()
```

Determining Which Virtual Destructor to Use

确定使用哪个虚析构函数

Assuming all the root base classes properly define their destructors as virtual, then the handling of the virtual destructor is consistent regardless of the pointer type through which we delete the object:

假定所有根基类都将它们的析构函数适当定义为虚函数，那么，无论通过哪种指针类型删除对象，虚析构函数的处理都是一致的：

```
// each pointer points to a Panda
delete pz; // pz is a ZooAnimal*
delete pb; // pb is a Bear*
delete pp; // pp is a Panda*
delete pe; // pe is a Endangered*
```

Assuming each of these pointers points to a `Panda` object, the exact same order of destructor invocations occurs in each case. The order of destructor invocations is the reverse of the constructor order: The `Panda` destructor is invoked through the virtual mechanism. Following execution of the `Panda` destructor, the `Endangered`, `Bear`, then `ZooAnimal` destructors are invoked in turn.

假定这些指针每个都向 `Panda` 对象，则每种情况下发生完全相同的析构函数调用次序。析构函数调用的次序是构造函数次序的逆序：通过虚机制调用 `Panda` 析构函数。随着 `Panda` 析构函数的执行，依次调用 `Endangered`、`Bear` 和 `ZooAnimal` 的析构函数。

Exercises Section 17.3.2

Exercise 17.26: On page 735 we presented a series of calls made through a `Bear` pointer that pointed to a `Panda` object. We noted that if the pointer had been a `ZooAnimal` pointer the calls would resolve the same way. Explain why.

本节给出了一系列通过指向 `Panda` 对象的 `Bear` 指针所进行的调用。我们指出，如果指针是 `ZooAnimal` 指针，将以同样方式确定函数调用。解释为什么。

Exercise 17.27: Assume we have two base classes, `Base1` and `Base2`, each of which defines a virtual member named `print` and a virtual destructor. From these base classes we derive the following classes each of which redefines the `print` function:

假定有两个基类 `Base1` 和 `Base2`，其中每一个定义了一个名为 `print` 的虚成员和一个虚析构函数。从这些基类派生下面的类，其中每一个类都重定义了 `print` 函数：

```
class D1 : public Base1 { /* ... */ };
class D2 : public Base2 { /* ... */ };
class MI : public D1, public D2 { /* ... */ };
```

Using the following pointers determine which function is used in each call:

使用下面的指针确定在每个调用中使用哪个函数：

```
Base1 *pb1 = new MI; Base2 *pb2 = new MI;
D1 *pd1 = new MI; D2 *pd2 = new MI;

(a) pb1->print(); (b) pd1->print(); (c) pd2->print();
(d) delete pb2;   (e) delete pd1;   (f) delete pd2;
```

Exercise 17.28: Write the class definitions that correspond to Table 17.2 (p. 735).

编写与图 17.2 对应的类定义。

17.3.3. Copy Control for Multiply Derived Classes

17.3.3. 多重继承派生类的复制控制

The memberwise initialization, assignment and destruction (Chapter 13) of a multiply derived class behaves in the same way as under single inheritance. Each base class is implicitly constructed, assigned or destroyed, using that base class' own copy constructor, assignment operator or destructor.

多重继承的派生类的逐个成员初始化、赋值和析构（第十三章），表现得与单继承下的一样，使用基类自己的复制构造函数、赋值操作符或析构函数隐式构造、赋值或撤销每个基类。

Let's assume that `Panda` uses the default copy control members. Using the default copy constructor, the initialization of `ling_ling`

假定 `Panda` 类使用默认复制控制成员。`ling_ling` 的初始化

```
Panda ying_yang("ying_yang"); // create a Panda object
Panda ling_ling = ying_yang; // uses copy constructor
```

invokes the `Bear` copy constructor, which in turn runs the `ZooAnimal` copy constructor prior to executing the `Bear` copy constructor. Once the `Bear` portion of `ling_ling` is constructed, the `Endangered` copy constructor is run to create that part of the object. Finally, the `Panda` copy constructor is run.

使用默认复制构造函数调用 `Bear` 复制构造函数，`Bear` 复制构造函数依次在执行 `Bear` 复制构造函数之前运行 `ZooAnimal` 复制构造函数。一旦构造了 `ling_ling` 的 `Bear` 部分，就运行 `Endangered` 复制构造函数来创建对象的那个部分。最后，运行 `Panda` 复制构造函数。

The synthesized assignment operator behaves similarly to the copy constructor. It assigns the `Bear` (and through `Bear`, the `ZooAnimal`) parts of the object first. Next, it assigns the `Endangered` part, and finally the `Panda` part.

Section 17.3. Multiple and Virtual Inheritance

合成的赋值操作符的行为类似于复制构造函数，它首先对对象的 `Bear` 部分进行赋值，并通过 `Bear` 对对象的 `ZooAnimal` 部分进行赋值，然后，对 `Endangered` 部分进行赋值，最后对 `Panda` 部分进行赋值。

The synthesized destructor destroys each member of the `Panda` object and calls the destructors for the base class parts, in reverse order from construction.

合成的析构函数撤销 `Panda` 对象的每个成员，并且按构造次序的逆序为基类部分调用析构函数。



As is the case for single inheritance ([Section 15.4.3](#), p. 584), if a class with multiple bases defines its own destructor, that destructor is responsible only for cleaning up the derived class. If the derived class defines its own copy constructor or assignment operator, then the class is responsible for copying (assigning) all the base class subparts. The base parts are automatically copied or assigned only if the derived class uses the synthesized versions of these members.

像单继承 ([第 15.4.3 节](#)) 的情况一样，如果具有多个基类的类定义了自己的析构函数，该析构函数只负责清除派生类。如果派生类定义了自己的复制构造函数或赋值操作符，则类负责复制（赋值）所有的基类子部分。只有派生类使用复制构造函数或赋值操作符的合成版本，才自动复制或赋值基类部分。

17.3.4. Class Scope under Multiple Inheritance

17.3.4. 多重继承下的类作用域

Class scope ([Section 15.5](#), p. 590) is more complicated in multiple inheritance because a derived scope may be enclosed by multiple base class scopes. As usual, name lookup for a name used in a member function starts in the function itself. If the name is not found locally, then lookup continues in the member's class and then searches each base class in turn. Under multiple inheritance, the search simultaneously examines all the base-class inheritance subtrees in our example, both the `Endangered` and the `Bear/ZooAnimal` subtrees are examined in parallel. If the name is found in more than one subtree, then the use of that name must explicitly specify which base class to use. Otherwise, the use of the name is ambiguous.

在多重继承下，类作用域 ([第 15.5 节](#)) 更加复杂，因为多个基类作用域可以包围派生类作用域。通常，成员函数中使用的名字和查找首先在函数本身进行，如果不能在本地找到名字，就继续在成员的类中查找，然后依次查找每个基类。在多重继承下，查找同时检测所有的基类继承子树——在我们的例子中，并行查找 `Endangered` 子树和 `Bear/ZooAnimal` 子树。如果在多个子树中找到该名字，则那个名字的使用必须显式指定使用哪个基类；否则，该名字的使用是二义性的。



When a class has multiple base classes, name lookup happens simultaneously through all the immediate base classes. It is possible for a multiply derived class to inherit a member with the same name from two or more base classes. Unqualified uses of that name are ambiguous.

当一个类有多个基类的时候，通过所有直接基类同时进行名字查找。多重继承的派生类有可能从两个或多个基类继承同名成员，对该成员不加限定的使用是二义性的。

Multiple Base Classes Can Lead to Ambiguities

多个基类可能导致二义性

Assume both `Bear` and `Endangered` define a member named `print`. If `Panda` does not define that member, then a statement such as the following

假定 `Bear` 类和 `Endangered` 类都定义了名为 `print` 的成员，如果 `Panda` 类没有定义该成员，则

```
ying_yang.print(cout);
```

results in a compile-time error.

这样的语句将导致编译时错误。

The derivation of `Panda`, which results in `Panda` having two members named `print`, is perfectly legal. The derivation results in only a *potential* ambiguity. That ambiguity is avoided if no `Panda` object ever calls `print`. The error would also be avoided if each call to `print` specifically indicated which version of `print` was wanted `Bear::print` or `Endangered::print`. An error is issued only if there is an ambiguous attempt to use the member.

Section 17.3. Multiple and Virtual Inheritance

`Panda` 类的派生（它导致有两个名为 `print` 的成员）是完全合法的。派生只是导致潜在的二义性，如果没有 `Panda` 对象调用 `print`，就可以避免这个二义性。如果每个 `print` 调用明确指出想要哪个版本——`Bear::print` 还是 `Endangered::print`，也可以避免错误。只有在存在使用该成员的二义性尝试的时候，才会出错。

If a declaration is found only in one base-class subtree, then the identifier is resolved and the lookup algorithm concludes. For example, class `Endangered` might have an operation to return the given estimated population of its object. If so, the following call

如果只在一个基类子树中找到声明，则标识符得以确定而查找算法结束。例如，`Endangered` 类可能有一个操作返回给定其对象的估计数目，如果是这样，下面的调用

```
ying_yang.population();
```

would compile without complaint. The name `population` would be found in the `Endangered` base class and does not appear in `Bear` or any of its base classes.

可以顺利编译，名字 `population` 将在基类 `Endangered` 中找到，并且在 `Bear` 类或其任意基类中都不会出现。

Name Lookup Happens First

首先发生名字查找

Although the ambiguity of the two inherited `print` members is reasonably obvious, it might be more surprising to learn that an error would be generated even if the two inherited functions had different parameter lists. Similarly, it would be an error even if the `print` function were private in one class and public or protected in the other. Finally, if `print` were defined in `ZooAnimal` and not `Bear`, the call would still be in error.

虽然两个继承的 `print` 成员的二义性相当明显，但是也许更令人惊讶的是，即使两个继承的函数有不同的形参表，也会产生错误。类似地，即使函数在一个类中是私有的而在另一个类中是公用或受保护的，也是错误的。最后，如果在 `ZooAnimal` 类中定义了 `print` 而 `Bear` 类中没有定义，调用仍是错误的。

As always, name lookup happens in two steps ([Section 7.8.1](#), p. 268): First the compiler finds a matching declaration (or, in this case, two matching declarations, which causes the ambiguity). Only then does the compiler decide whether the declaration it found is legal.

名字查找总是以两个步骤发生 ([第 7.8.1 节](#))：首先编译器找到一个匹配的声明（或者，在这个例子中，找到两个匹配的声明，这导致二义性），然后，编译器才确定所找到的声明是否合法。

Avoiding User-Level Ambiguities

避免用户二义性

We could resolve the `print` ambiguity by specifying which class to use:

可以通过指定使用哪个类解决二义性：

```
ying_yang.Endangered::print(cout);
```

The best way to avoid potential ambiguities is to define a version of the function in the derived class that resolves the ambiguity. For example, we should give our `Panda` class a `print` function that chooses which version of `print` to use:

避免潜在二义性最好的方法是，在解决二义性的派生类中定义函数的一个版本。例如，应该给选择使用哪个 `print` 版本的 `Panda` 类一个 `print` 函数：

```
std::ostream& Panda::print(std::ostream &os) const
{
    Bear::print(os);           // print the Bear part
    Endangered::print(os);   // print the Endangered part
    return os;
}
```

Code for Exercises to Section 17.3.4

本节习题的代码

```
struct Basel {
    void print(int) const;
protected:
    int    ival;
    double dval;
```

```

    char  cval;
private:
    int   *id;
};

struct Base2 {
    void print(double) const;
protected:
    double fval;
private:
    double dval;
};

struct Derived : public Base1 {
    void print(std::string) const;
protected:
    std::string sval;
    double      dval;
};

struct MI : public Derived, public Base2 {
    void print(std::vector<double>);
protected:
    int          *ival;
    std::vector<double>  dvec;
};

```

Exercises Section 17.3.4

Exercise 17.29: Given the class hierarchy in the box on this page and the following `MI::foo` member function skeleton,

给定前面给出的类层次以及下面的 `MI::foo` 成员函数框架,

```

int ival;
double dval;

void MI::foo(double dval) { int id; /* ... */ }

```

- identify the member names visible from within `MI`. Are there any names visible from more than one base class?

识别从 `MI` 中可见的成员名字。有从多个基类中都可见的名字吗？

- identify the set of members visible from within `MI::foo`.

识别从 `MI::foo` 中可见的成员的集合。

Exercise 17.30: Given the hierarchy in the box on page 739, why is this call to `print` an error?

给定前面给出的类层次，为什么下面的这个 `print` 调用是错误的？

```

MI mi;
mi.print(42);

```

Revise `MI` to allow this call to `print` to compile and execute correctly.

修改 `MI`，使得这个 `print` 调用可以正确编译和执行。

Exercise 17.31: Using the class hierarchy in the box on page 739, identify which of the following assignments, if any, are in error:

使用前面给出的类层次，如果下面赋值中有错误的，识别哪些是错误的：

```

void MI::bar() {
    int sval;
    // exercise questions occur here ...
}
(a) dval = 3.14159;  (d) fval = 0;
(b) cval = 'a';      (e) sval = *ival; (c) id = 1;

```

Exercise 17.32: Using the class hierarchy defined in the box on page 739 and the following skeleton of the `MI::foobar` member function

给定前面给出的类层次以及下面的 `MI::foobar` 成员函数框架

Section 17.3. Multiple and Virtual Inheritance

```
void MI::foobar(double cval)
{
    int dval;
    // exercise questions occur here ...
}
```

- a. assign to the local instance of `dval` the sum of the `dval` member of `Base1` and the `dval` member of `Derived`.

将 `Base1` 的 `dval` 成员和 `Derived` 的 `dval` 成员的和赋给 `dval` 的局部实例。

- b. assign the value of the last element in `MI::dvec` to `Base2::fval`.

将 `MI::dvec` 中最后一个元素赋给 `Base2::fval`。

- c. assign `cval` from `Base1` to the first character in `sval` from `Derived`.

将 `Base1` 的 `cval` 赋给 `Derived` 中 `sval` 的第一个字符。

17.3.5. Virtual Inheritance

17.3.5. 虚继承

Under multiple inheritance, a base class can occur multiple times in the derivation hierarchy. In fact, our programs have already used a class that inherits from the same base class more than once through its inheritance hierarchy.

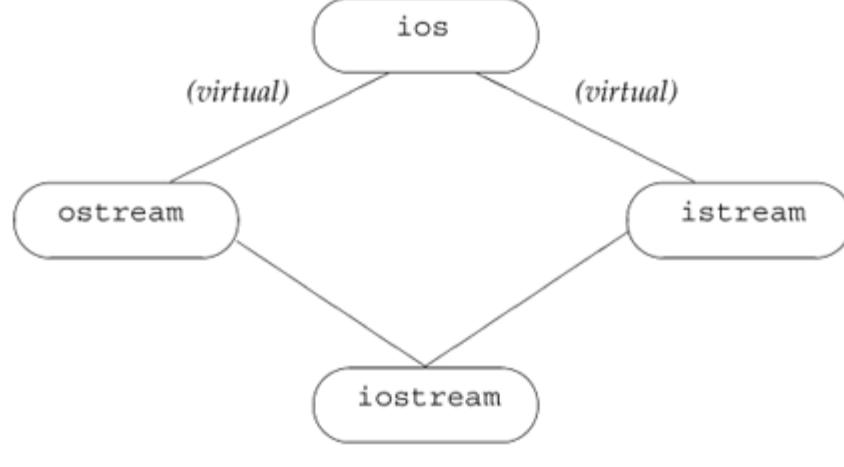
在多重继承下，一个基类可以在派生层次中出现多次。实际上，我们的程序已经使用过通过继承层次多次继承同一基类的类。

Each of the IO library classes inherits from a common abstract base class. That abstract class manages the condition state of the stream and holds the buffer that the stream reads or writes. The `istream` and `ostream` classes inherit directly from this common base class. The library defines another class, named `iostream`, that inherits from both `istream` and `ostream`. The `iostream` class can both read and write a stream. A simplified version of the IO inheritance hierarchy is illustrated in [Figure 17.3](#) on the facing page.

每个 IO 库类都继承了一个共同的抽象基类，那个抽象基类管理流的条件状态并保存流所读写的缓冲区。`istream` 和 `ostream` 类直接继承这个公共基类，库定义了另一个名为 `iostream` 的类，它同时继承 `istream` 和 `ostream`，`iostream` 类既可以对流进行读又可以对流进行写。IO 继承层次的简化版本如图 17.3 所示。

Figure 17.3. Virtual Inheritance `iostream` Hierarchy (Simplified)

17.3. 虚继承 `iostream` 层次 (简化的)



As we know, a multiply inherited class inherits state and action from each of its parents. If the IO types used normal inheritance, then each `iostream` object would contain two `ios` subobjects: one instance contained within its `istream` subobject and the other within its `ostream` subobject. From a design perspective, this implementation is just wrong: The `iostream` class wants to read to and write from a single buffer; it wants the

Section 17.3. Multiple and Virtual Inheritance

condition state to be shared across input and output operations. If there are two separate `ios` objects, this sharing is not possible.

像我们知道的那个，多重继承的类从它的每个父类继承状态和动作，如果 `IO` 类型使用常规继承，则每个 `iostream` 对象可能包含两个 `ios` 子对象：一个包含在它的 `istream` 子对象中，另一个包含在它的 `ostream` 子对象中，从设计角度讲，这个实现正是错误的：`iostream` 类想要对单个缓冲区进行读和写，它希望跨越输入和输出操作符共享条件状态。如果有两个单独的 `ios` 对象，这种共享是不可能的。

In C++ we solve this kind of problem by using [virtual inheritance](#). Virtual inheritance is a mechanism whereby a class specifies that it is willing to share the state of its virtual base class. Under virtual inheritance, only one, shared base-class subobject is inherited for a given virtual base regardless of how many times the class occurs as a virtual base within the derivation hierarchy. The shared base-class subobject is called a [virtual base class](#).

在 C++ 中，通过使用[虚继承](#)解决这类问题。虚继承是一种机制，类通过虚继承指出它希望共享其虚基类的状态。在虚继承下，对给定虚基类，无论该类在派生层次中作为虚基类出现多少次，只继承一个共享的基类子对象。共享的基类子对象称为[虚基类](#)。

The `istream` and `ostream` classes inherit virtually from their base class. By making their base class `virtual`, `istream` and `ostream` specify that if some other class, such as `iostream`, inherits from both of them, then only one copy of their common base class will be present in the derived class. We make a base class `virtual` by including the keyword `virtual` in the derivation list:

`istream` 和 `ostream` 类对它们的基类进行虚继承。通过使基类成为虚基类，`istream` 和 `ostream` 指定，如果其他类（如 `iostream` 同时继承它们两个，则派生类中只出现它们的公共基类的一个副本。通过在派生列表中包含关键字 `virtual` 设置虚基类：

```
class istream : public virtual ios { ... };
class ostream : virtual public ios { ... };

// iostream inherits only one copy of its ios base class
class iostream: public istream, public ostream { ... };
```

A Different Panda Class

一个不同的 Panda 类

For the purposes of illustrating virtual inheritance, we'll continue to use the `Panda` class as a pedagogical example. Within zoological circles, for more than 100 years there has been an occasionally fierce debate as to whether the `Panda` belongs to the `Raccoon` or the `Bear` family. Because software design is primarily a service industry, our most practical solution is to derive `Panda` from both:

为了举例说明虚继承，我们将继续使用 `Panda` 类作为教学例子。在动物学圈子中，对于 `Panda` 是属于 `Raccoon` 科还是 `Bear` 科已经争论了 100 年以上。因为软件设计主要是一种服务行业，所以最现实的解决方案是从二者派生 `Panda`：

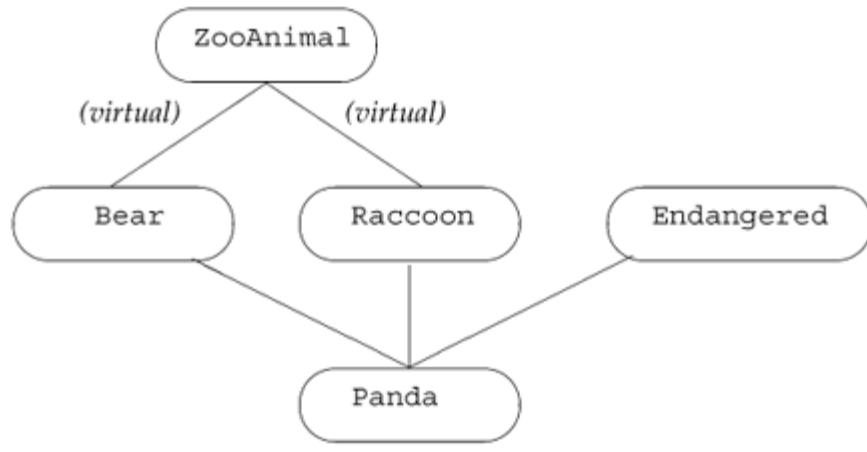
```
class Panda : public Bear,
              public Raccoon, public Endangered {
};
```

Our virtual inheritance `Panda` hierarchy is pictured in [Figure 17.4](#). If we examine that hierarchy, we notice a nonintuitive aspect of virtual inheritance: The virtual derivation (in our case, of `Bear` and `Raccoon`) has to be made prior to any actual need for it to be present. Virtual inheritance becomes necessary only with the declaration of `Panda`, but if `Bear` and `Raccoon` are not already virtually derived, the designer of the `Panda` class is out of luck.

虚继承 `Panda` 层次如图 17.4 所示。如果检查该层次，我们注意到虚继承一个不直观的特征：必须在提出虚派生的任意实际需要之前进行虚派生（在例中，`Bear` 类和 `Raccoon` 类的虚派生）。只有在使用 `Panda` 的声明时，虚继承才是必要的，但如果 `Bear` 类和 `Raccoon` 类不是虚派生的，`Panda` 类的设计者就没有好运气了。

Figure 17.4. Virtual Inheritance `Panda` Hierarchy

17.4. 虚继承 `Panda` 层次



In practice, the requirement that an intermediate base class specify its inheritance as `virtual` rarely causes any problems. Ordinarily, a class hierarchy that uses `virtual` inheritance is designed at one time by either one individual or a project design group. It is exceedingly rare for a class to be developed independently that needs a `virtual` base in one of its base classes and in which the developer of the new base class cannot change the existing hierarchy.

实际上，中间基类指定其继承为虚继承的要求很少引起任何问题。通常，使用虚继承的类层次是一次性由一个人或一个项目设计组设计的，独立开发的类很少需要其基类中的一个基类是虚基类，而且新基类的开发者不能改变已经存在的层次。

17.3.6. Virtual Base Class Declaration

17.3.6. 虚基类的声明

A base class is specified as being derived through `virtual` inheritance by modifying its declaration with the keyword `virtual`. For example, the following declarations make `ZooAnimal` a `virtual` base class of both `Bear` and `Raccoon`:

通过用关键字 `virtual` 修改声明，将基类指定为通过虚继承派生。例如，下面的声明使 `ZooAnimal` 类成为 `Bear` 类和 `Raccoon` 类的虚基类：

```
// the order of the keywords public and virtual is not significant
class Raccoon : public virtual ZooAnimal { /* ... */ };
class Bear : virtual public ZooAnimal { /* ... */ };
```



Specifying `virtual` derivation has an impact only in classes derived from the class that specifies a `virtual` base. Rather than affecting objects of the derived class' own type, it is a statement about the derived class' relationship to its own, future derived class.

指定虚派生只影响从指定了虚基类的类派生的类。除了影响派生类自己的对象之外，它也是关于派生类与自己的未来派生类的关系的一个陈述。

The `virtual` specifier states a willingness to share a single instance of the named base class within a subsequently derived class.

`virtual` 说明符陈述了在后代派生类中共享指定基类的单个实例的愿望。

Any class that can be specified as a base class also could be specified as a `virtual` base class. A `virtual` base may contain any class element normally supported by a `nonvirtual` base class.

任何可被指定为基类的类也可以被指定为虚基类，虚基类可以包含通常由非虚基类支持的任意类元素。

Normal Conversions to Base Are Supported

支持到基类的常规转换

An object of the derived class can be manipulated as usual through a pointer or a reference to a base-class type even though the base class is `virtual`. For example, all of the following `Panda` base class conversions execute correctly even though `Panda` inherits its `ZooAnimal` part as a `virtual`

Section 17.3. Multiple and Virtual Inheritance

base:

即使基类是虚基类，也照常可以通过基类类型的指针或引用操纵派生类的对象。例如，即使 `Panda` 类将它的 `ZooAnimal` 部分作为虚基类继承，下面所有 `Panda` 的基类转换也能正确执行：

```
void dance(const Bear*);  
void rummage(const Raccoon*);  
ostream& operator<<(ostream&, const ZooAnimal&);  
Panda ying_yang;  
dance(&ying_yang); // ok: converts address to pointer to Bear  
rummage(&ying_yang); // ok: converts address to pointer to Raccoon  
cout << ying_yang; // ok: passes ying_yang as a ZooAnimal
```

Visibility of Virtual Base-Class Members

虚基类成员的可见性

Multiple-inheritance hierarchies using virtual bases pose fewer ambiguity problems than do those without virtual inheritance.

使用虚基类的多重继承层次比没有虚继承的引起更少的二义性问题。



Members in the shared virtual base can be accessed unambiguously and directly. Similarly, if a member from the virtual base is redefined along only one derivation path, then that redefined member can be accessed directly. Under a nonvirtual derivation, both kinds of access would be ambiguous.

可以无二义性地直接访问共享虚基类中的成员。同样，如果只沿一个派生路径重定义来自虚基类的成员，则可以直接访问该重定义成员。在非虚派生情况下，两种访问都可能是二义性的。

Assume a member named `x` is inherited through more than one derivation path. There are three possibilities:

假定通过多个派生路径继承名为 `x` 的成员，有下面三种可能性：

1. If in each path `x` represents the same virtual base class member, then there is no ambiguity because a single instance of the member is shared.

如果在每个路径中 `x` 表示同一虚基类成员，则没有二义性，因为共享该成员的单个实例。

2. If in one path `x` is a member of the virtual base class member and in another path `x` is a member of a subsequently derived class, there is also no ambiguity—the specialized derived class instance is given precedence over the shared virtual base class instance.

如果在某个路径中 `x` 是虚基类的成员，而在另一路径中 `x` 是后代派生类的成员，也没有二义性——特定派生类实例的优先级高于共享虚基类实例。

3. If along each inheritance path `x` represents a different member of a subsequently derived class, then the direct access of the member is ambiguous.

如果沿每个继承路径 `x` 表示后代派生类的不同成员，则该成员的直接访问是二义性的。

As in a nonvirtual multiple inheritance hierarchy, ambiguities of this sort are best resolved by the class providing an overriding instance in the derived class.

像非虚多重继承层次一样，这种二义性最好用在派生类中提供覆盖实例的类来解决。

Exercises Section 17.3.6

Exercise

17.33:

Given the following class hierarchy, which inherited members can be accessed without qualification from within the `VMI` class? Which require qualification? Explain your reasoning.

给定下面的类层次，从 `VMI` 类内部可以限定地访问哪些继承成员？哪些继承成员需要限定？解释你的推理。

```
class Base {  
public:  
    bar(int);  
protected:  
    int ival;  
};  
class Derived1 : virtual public Base {  
public:  
}
```

```

        bar(char);
        foo(char);
protected:
    char cval;
};

class Derived2 : virtual public Base {
public:
    foo(int);
protected:
    int ival;
    char cval;
};

class VMI : public Derived1, public Derived2 { };

```

17.3.7. Special Initialization Semantics

17.3.7. 特殊的初始化语义

Ordinarily each class initializes only its own direct base class(es). This initialization strategy fails when applied to a virtual base class. If the normal rules were used, then the virtual base might be initialized multiple times. The class would be initialized along each inheritance path that contains the virtual base. In our `ZooAnimal` example, using normal initialization rules would result in both `Bear` and `Raccoon` attempting to initialize the `ZooAnimal` part of a `Panda` object.

通常，每个类只初始化自己的直接基类。在应用于虚基类的进修，这个初始化策略会失败。如果使用常规规则，就可能会多次初始化虚基类。类将沿着包含该虚基类的每个继承路径初始化。在 `ZooAnimal` 示例中，使用常规规则将导致 `Bear` 类和 `Raccoon` 类都试图初始化 `Panda` 对象的 `ZooAnimal` 类部分。

To solve this duplicate-initialization problem, classes that inherit from a class that has a virtual base have special handling for initialization. In a virtual derivation, the virtual base is initialized by the *most derived constructor*. In our example, when we create a `Panda` object, the `Panda` constructor alone controls how the `ZooAnimal` base class is initialized.

为了解决这个重复初始化问题，从具有虚基类的类继承的类对初始化进行特殊处理。在虚派生中，由最低层派生类的构造函数初始化虚基类。在我们的例子中，当创建 `Panda` 对象的时候，只有 `Panda` 构造函数控制怎样初始化 `ZooAnimal` 基类。

Although the virtual base is initialized by the most derived class, any classes that inherit immediately or indirectly from the virtual base usually also have to provide their own initializers for that base. As long as we can create independent objects of a type derived from a virtual base, that class must initialize its virtual base. These initializers are used only when we create objects of the intermediate type.

虽然由最低层派生类初始化虚基类，但是任何直接或间接继承虚基类的类一般也必须为该基类提供自己的初始化式。只要可以创建虚基类派生类类型的独立对象，该类就必须初始化自己的虚基类，这些初始化式只有创建中间类型的对象时使用。

In our hierarchy, we could have objects of type `Bear`, `Raccoon`, or `Panda`. When a `Panda` is created, it is the most derived type and controls initialization of the shared `ZooAnimal` base. When a `Bear` (or a `Raccoon`) is created, there is no further derived type involved. In this case, the `Bear` (or `Raccoon`) constructors directly initialize their `ZooAnimal` base as usual:

在我们的层次中，可以有 `Bear`、`Raccoon` 或 `Panda` 类型的对象。创建 `Panda` 对象的时候，它是最低层派生类型并控制共享的 `ZooAnimal` 基类的初始化：创建 `Bear` 对象（或 `Raccoon` 对象）的进修，不涉及更低层的派生类型。在这种情况下，`Bear`（或 `Raccoon`）构造函数像平常一样直接初始化它们的 `ZooAnimal` 基类：

```

Bear::Bear(std::string name, bool onExhibit):
    ZooAnimal(name, onExhibit, "Bear") {}
Raccoon::Raccoon(std::string name, bool onExhibit)
    : ZooAnimal(name, onExhibit, "Raccoon") {}

```

The `Panda` constructor also initializes the `ZooAnimal` base, even though it is not an immediate base class:

虽然 `ZooAnimal` 不是 `Panda` 的直接基类，但是 `Panda` 构造函数也初始化 `ZooAnimal` 基类：

```

Panda::Panda(std::string name, bool onExhibit)
    : ZooAnimal(name, onExhibit, "Panda"),
    Bear(name, onExhibit),
    Raccoon(name, onExhibit),
    Endangered(Endangered::critical),
    sleeping_flag(false) {}

```

When a `Panda` is created, it is this constructor that initializes the `ZooAnimal` part of the `Panda` object.

创建 `Panda` 对象的时候，这个构造函数初始化 `Panda` 对象的 `ZooAnimal` 部分。

How a Virtually Inherited Object Is Constructed

怎样构造虚继承的对象

Let's look at how objects under virtual inheritance are constructed.

让我们看看虚继承情况下怎样构造对象。

```
Bear winnie("pooh"); // Bear constructor initializes ZooAnimal
Raccoon meeko("meeko"); // Raccoon constructor initializes ZooAnimal
Panda yolo("yolo"); // Panda constructor initializes ZooAnimal
```

When a `Panda` object is created,

当创建 `Panda` 对象的时候,

1. The `ZooAnimal` part is constructed first, using the initializers specified in the `Panda` constructor initializer list.

首先使用构造函数初始化列表中指定的初始化式构造 `ZooAnimal` 部分。

2. Next, the `Bear` part is constructed. The initializers for `ZooAnimal Bear`'s constructor initializer list are ignored.

接下来, 构造 `Bear` 部分。忽略 `Bear` 的用于 `ZooAnimal` 构造函数初始化列表的初始化式。

3. Then the `Raccoon` part is constructed, again ignoring the `ZooAnimal`.

然后, 构造 `Raccoon` 部分, 再次忽略 `ZooAnimal` 初始化式。

4. Finally, the `Panda` part is constructed.

最后, 构造 `Panda` 部分。

If the `Panda` constructor does not explicitly initialize the `ZooAnimal` base class, then the `ZooAnimal` default constructor is used. If `ZooAnimal` doesn't have a default constructor, then the code is in error. The compiler will issue an error message when the definition of `Panda`'s constructor is compiled.

如果 `Panda` 构造函数不显式初始化 `ZooAnimal` 基类, 就使用 `ZooAnimal` 默认构造函数; 如果 `ZooAnimal` 没有默认构造函数, 则代码出错。当编译 `Panda` 构造函数的定义时, 编译器将给出一个错误信息。

Constructor and Destructor Order

构造函数与析构函数次序



Virtual base classes are always constructed prior to nonvirtual base classes regardless of where they appear in the inheritance hierarchy.

无论虚基类出现在继承层次中任何地方, 总是在构造非虚基类之前构造虚基类。

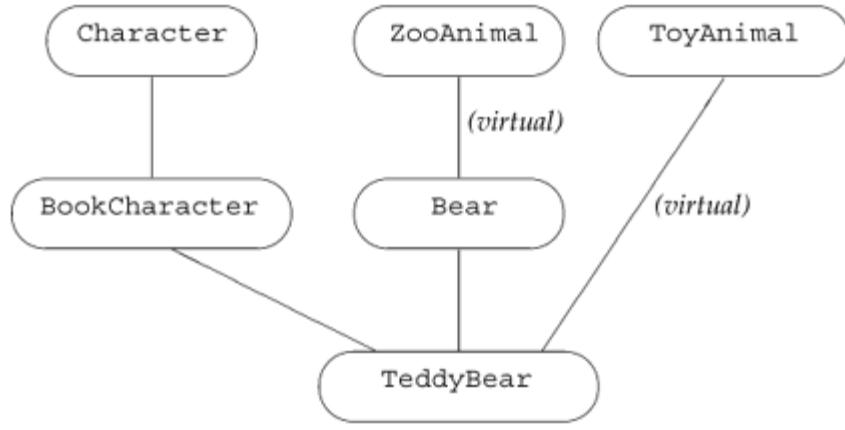
For example, in the following whimsical `TeddyBear` derivation, there are two virtual base classes: the `ToyAnimal` base class and the indirect `ZooAnimal` base class from which `Bear` is derived:

例如, 下面毫无规律的 `TeddyBear` (图 17.5) 派生中, 有两个虚基类: `ToyAnimal` 基类和派生 `Bear` 的间接基类 `ZooAnimal`:

```
class Character { /* ... */;
class BookCharacter : public Character { /* ... */ };
class ToyAnimal { /* ... */ };
class TeddyBear : public BookCharacter,
    public Bear, public virtual ToyAnimal
{ /* ... */};
```

Figure 17.5. Virtual Inheritance `TeddyBear` Hierarchy

图 17.5. 虚继承 `TeddyBear` 层次



The immediate base classes are examined in declaration order to determine whether there are any virtual base classes. In our example, the inheritance subtree of `BookCharacter` is examined first, then that of `Bear`, and finally that of `ToyAnimal`. Each subtree is examined starting at the root class down to the most derived class.

按声明次序检查直接基类，确定是否存在虚基类。例中，首先检查 `BookCharacter` 的继承子树，然后检查 `Bear` 的继承子树，最后检查 `ToyAnimal` 的继承子树。按从根类开始向下到最低层派生类的次序检查每个子树。

The order in which the virtual base classes are constructed for `TeddyBear` is `ZooAnimal` followed by `ToyAnimal`. Once the virtual base classes are constructed, the nonvirtual base-class constructors are invoked in declaration order: `BookCharacter`, which causes the `Character` constructor to be invoked, and then `Bear`. Thus, to create a `TeddyBear`, the constructors are invoked in the following order:

`TeddyBear` 的虚基类的构造次序是先 `ZooAnimal` 再 `ToyAnimal`。一旦构造了虚基类，就按声明次序调用非虚基类的构造函数：首先是 `BookCharacter`，它导致调用 `Character` 构造函数，然后是 `Bear`。因此，为了创建 `TeddyBear` 对象，按下面次序调用构造函数：

```

ZooAnimal();           // Bear's virtual base class
ToyAnimal();           // immediate virtual base class
Character();          // BookCharacter's nonvirtual base class
BookCharacter();       // immediate nonvirtual base class
Bear();                // immediate nonvirtual base class
TeddyBear();           // most derived class
  
```

where the initializers used for `ZooAnimal` and `ToyAnimal` are specified by the most derived class `TeddyBear`.

在这里，由最低层派生类 `TeddyBear` 指定用于 `ZooAnimal` 和 `ToyAnimal` 的初始化式。

The same construction order is used in the synthesized copy constructor; the base classes also are assigned in this order in the synthesized assignment operator. The order of base-class destructor calls is guaranteed to be the reverse order of constructor invocation.

在合成复制构造函数中使用同样的构造次序，在合成赋值操作符中也是按这个次序给基类赋值。保证调用基类析构函数的次序与构造函数的调用次序相反。

Exercises Section 17.3.7

Exercise 17.34: There is one case in which a derived class need not supply initializers for its virtual base class(es). What is this case?

有一种情况下派生类不必为虚基类提供初始化式，这种情况是什么？

Exercise 17.35: Given the following class hierarchy,

给定下面类层次，

```

class Class { ... };
class Base : public Class { ... };
class Derived1 : virtual public Base { ... };
class Derived2 : virtual public Base { ... };
class MI : public Derived1,
            public Derived2 { ... };
class Final : public MI, public Class { ... };
  
```

- a. What is the order of constructor and destructor for the definition of a `Final` object?

一个 `Final` 对象中有几个 `Base` 子对象?

b. How many `Base` subobjects are in a `Final` object? How many `Class` subobjects?

有几个 `class` 子对象?

c. Which of the following assignments is a compile-time error?

下面哪个赋值在编译时有错?

```
Base      *pb;      Class      *pc;
MI       *pmi;     Derived2   *pd2;

(a) pb = new Class;          (c) pmi = pb;
(b) pc = new Final;         (d) pd2 = pmi;
```

Exercise 17.36: Given the previous hierarchy, and assuming that `Base` defines the following three constructors, define the classes that inherit from `Base`, giving each class the same three constructors. Each constructor should use its argument to initialize its `Base` part.

给定前面的层次，并假定 `Base` 定义了下面三个构造函数，定义从 `Base` 派生的类，给每个类同样的三个构造函数，每个构造函数使用实参初始化其 `Base` 部分。

```
struct Base {
    Base();
    Base(std::string);
    Base(const Base&);

protected:
    std::string name;
};
```

Chapter Summary

小结

Most of C++ is applicable to a wide range of problems from those solvable in a few hour's time to those that take years of development by large teams. Some features in C++ are most applicable in the context of large-scale problems: exception handling, namespaces, and multiple or virtual inheritance.

C++ 的大部分特征都可以应用于范围很广的问题——从几小时便可解决的问题到大团队花几年时间才能解决的问题。其中有一些特征则最适用于大规模问题的情况，这些特征包括异常处理、命名空间、多重继承和虚继承。

Exception handling lets us separate the error-detection part of the program from the error-handling part. [Section 6.13](#) (p. 215) introduced exception handling and this chapter completes our coverage of exceptions. When an exception is thrown, the current executing function is suspended and a search is started to find the nearest `catch` clause. Local variables defined inside functions that are exited while searching for a `catch` clause are destroyed as part of handling the exception. The fact that objects are destroyed gives rise to an important programming technique known as "resource allocation is initialization" (RAII).

通过异常处理我们能够将程序的错误检测部分与错误处理部分分开。异常处理是在[第 6.13 节](#)中引入的，本章最终完成了对异常处理的讨论。在抛出异常的时候，会终止当前正在执行的函数并开始查找最近的 `catch` 子句，在查找 `catch` 子句的时候，作为异常处理的一部分，将撤销退出函数内部定义的局部变量。这种撤销对象提供了一个重要的编程技术，称为“资源分配即初始化”（RAII）。

Namespaces are a mechanism for managing large complex applications built from code produced by independent suppliers. A namespace is a scope in which objects, types, functions, template, and other namespaces may be defined. The standard library is defined inside the namespace named `std`.

命名空间是一种机制，用于管理用独立供应商开发的代码建立的大型复杂应用程序。一个命名空间就是一个作用域，其中可以定义对象、类型、函数、模板和其他命名空间。标准库就定义在名为 `std` 的命名空间中。

Names in a namespace may be made available to the current scope one at a time via a `using` declaration. Alternatively, but much less safely, all the names in a namespace may be brought into the current scope via a `using` directive.

通过 `using` 声明，当前作用域中就都可以访问某个命名空间中的名字了。当然，也可以通过 `using` 指示将一个命名空间中的所有名字带入当前作用域，但这种做法很不安全。

Conceptually, multiple inheritance is a simple notion: A derived class may inherit from more than one direct base class. The derived object consists of the derived part and a base part contributed by each of its base classes. Although conceptually simple, the details can be more complicated. In particular, inheriting from multiple base classes introduces new possibilities for name collisions and resulting ambiguous references to names from the base part of an object.

从概念上来看，多重继承很简单：派生类可以继承多个直接基类，派生类对象由派生部分和每个基类所贡献的基类部分构成。虽然多重继承概念简单，但细节可能非常复杂，尤其是，继承多个基类引入了新的名字冲突可能性，并且会导致对对象基类中的名字的引用出现二义性。

When a class inherits from more than one immediate base class, it is possible that those classes may themselves share another base class. In cases such as this, the intermediate classes can opt to make their inheritance virtual, which states a willingness to share its virtual base class with other classes in the hierarchy that inherit virtually from that same base class. In this way there is only one copy of the shared virtual base in a subsequently derived class.

一个类继承多个直接基类的时候，那些类有可能本身还共享另一个基类。在这种情况下，中间类可以选择使用虚继承，声明愿意与层次中虚继承同一基类的其他类共享虚基类。用这种方法，后代派生类中将只有一个共享虚基类的副本。

Defined Terms

术语

abort

Library function that abnormally terminates a program's execution. Ordinarily, `abort` is called by `terminate`. Programs may also call `abort` directly. It is defined in the `cstdlib` header.

异常终止程序执行的库函数。通常，由 `terminate` 调用 `abort`，程序也可以直接调用 `abort`。`abort` 定义在头文件 `cstdlib` 中。

auto_ptr

Library class template that provides exception-safe access to dynamically allocated objects. An `auto_ptr` cannot be bound to an array or a pointer to a variable. Copying and assigning an `auto_ptr` is a destructive operation: Ownership of the object is transferred from the right-hand operand to the left. Assigning to an `auto_ptr` deletes the object in the left-hand operand. As a result, `auto_ptrs` may not be stored in containers.

一个库类模板，提供对动态分配对象的异常安全的访问。不能将 `auto_ptr` 对象绑定到数组或者变量指针，`auto_ptr` 对象的复制和赋值是破坏性操作：将对象的所有权从右操作数转到左操作数。对 `auto_ptr` 对象进行赋值删除左操作数中的对象，因此，不能将 `auto_ptrs` 对象存储在容器中。

catch-all (捕获所有异常子句)

A catch clause in which the exception specifier is `(...)`. A catch-all clause catches an exception of any type. It is typically used to catch an exception that is detected locally in order to do local cleanup. The exception is then rethrown to another part of the program to deal with the underlying cause of the problem.

异常说明符为 `(...)` 的 `catch` 子句。这种子句能够捕获任意类型的异常，它通常用于捕获为进行局部清除而局部检测的异常。异常被重新抛出给程序的其他部分，以处理问题的基本原因。

catch clause (catch 子句)

The part of the program that handles an exception. A catch clause consists of the keyword `catch` followed by an exception specifier and a block of statements. The code inside a `catch` does whatever is necessary to handle an exception of the type defined in its exception specifier.

程序中处理异常的部分，也称异常处理代码。`catch` 子句由关键字 `catch` 后接异常说明符和语句块构成。`catch` 内部的代码完成的必要工作来处理由异常说明符定义的类型的异常。

constructor order (构造函数次序)

Ordinarily, base classes are constructed in the order in which they are named in the class derivation list. A derived constructor should explicitly initialize each base class through the constructor initializer list. The order in which base classes are named in the constructor initializer list does not affect the order in which the base classes are constructed. In a virtual inheritance, the virtual base class(es) are constructed before any other bases. They are constructed in the order in which they appear (directly or indirectly) in the derivation list of the derived type. Only the most derived type may initialize a virtual base; constructor initializers for that base that appear in the intermediate base classes are ignored.

一般而言，应该按照类派生列表中指定的次序构造基类，派生类构造函数应该通过构造函数初始化列表显式初始化每个基类。构造函数初始化列表中指定基类的次序不影响构造基类的次序。在虚继承中，虚基类在任何其他基类之前构造，它们按照在派生类型的派生列表中（直接或间接地）出现在次序进行构造，只有最低层派生类型可以初始化虚基类，中间基类中出现的基类构造函数初始化列表被忽略。

destructor order (析构函数次序)

Derived objects are destroyed in the reverse order from which they were constructed the derived part is destroyed first, then the classes named in the class derivation list are destroyed, starting with the last base class. Classes that serve as base classes in a multiple-inheritance hierarchy ordinarily should define their destructors to be virtual.

应该按照构造次序的逆序撤销派生类对象——首先撤销派生部分，然后，从最后一个基类开始，撤销类派生列表中指定的类。在多重继承层次中作为基类的类通常应该将它们的析构函数数定义为虚函数。

exception handler (异常处理代码)

Another way to refer to a catch clause.

Keyterm Defined Terms

catch 子句的另一个名称。

exception handling (异常处理)

Language-level support for managing run-time anomalies. One independently developed section of code can detect and "raise" an exception that another independently developed part of the program can "handle." The error-detecting part of the program **throws** an exception; the error-handling part handles the exception in a **catch** clause of a **try** block.

管理运行时异常的语言级支持。代码中一个独立开发的部分可以检测并“引发”异常，由程序中另一个独立开发的部分“处理”该异常。也就是说，程序的错误检测部分抛出异常，错误处理部分在 **try** 块的 **catch** 子句中处理异常。

exception object (异常对象)

Object used to communicate between the **throw** and **catch** sides of an exception. The object is created at the point of the throw and is a copy of the thrown expression. The exception object exists until the last handler for the exception completes. The type of the object is the type of the thrown expression.

用于在异常的 **throw** 和 **catch** 方之间进行通信的对象。在抛出点创建该对象，该对象是被抛出表达式的副本。异常对象一直存在，直到该异常最后一个处理代码结束。异常对象的类型是被抛出表达式的类型。

exception safe (异常安全的)

Term used to describe programs that behave correctly when exceptions are thrown.

用于描述在抛出异常时表现正确的程序的术语。

exception specification (异常说明)

Used on a function declaration to indicate what (if any) exception types a function throws. Exception types are named in a parenthesized, comma-separated list following the keyword **throw**, which appears after a function's parameter list. An empty list means that the function throws no exceptions. A function that has no exception specification may throw any exception.

用于函数声明之上，指出函数抛出什么（如果有）异常类型。在用圆括号括住、以逗号分隔、跟在关键字 **throw** 之后的列表中指定异常类型。空列表表示函数不抛出异常，没有异常说明的函数可以抛出任何异常。

exception specifier (异常说明符)

Specifies the types of exceptions that a given catch clause will handle. An exception specifier acts like a parameter list, whose single parameter is initialized by the exception object. Like parameter passing, if the exception specifier is a nonreference type, then the exception object is copied to the **catch**.

说明给定 **catch** 子句将处理的异常的类。异常说明符的行为形参表，由异常对象初始化它的单个形参。像参数传递一样，如果异常说明符是非引用类型，就将异常对象复制到 **catch** 中。

file static (文件静态)

Name local to a file that is declared with the **static** keyword. In C and pre-Standard versions of C++, file statics were used to declare objects that could be used in a single file only. File statics are deprecated in C++, having been replaced by the use of unnamed namespaces.

用关键字 **static** 声明的局部于文件的名字。在 C 语言和标准版本之前的 C++ 中，文件中的静态声明用于声明只能在单个文件中使用的对象，C++ 不赞成文件静态，已经用未命名的命名空间代替它。

function try block (函数测试块)

A **try** block that is a function body. The keyword **try** occurs before the opening curly of the function body and closes with catch clause(s) that appear after the close curly of the function body. Function try blocks are used most often to wrap constructor definitions in order to catch exceptions thrown by constructor initializers.

是函数体的 **try** 块。关键字 **try** 出现在函数体的左花括号之前，以出现在函数体的右花括号之后的 **catch** 子句作为结束。函数测试块最经常用于包围构造函数定义，以便捕获由构造函数初始化式抛出的异常。

global namespace (全局命名空间)

The (implicit) name-space in each program that holds all global definitions.

每个程序中保存所有全局定义的（隐式）命名空间。

multiple inheritance (多重继承)

Inheritance in which a class has more than one immediate base class. The derived class inherits the members of all its base classes. Multiple base classes are defined by naming more than one base class in the class derivation list. A separate access label is required for

Keyterm Defined Terms

each base class.

类有多个直接基类的继承。派生类继承所有基类的成员，通过在类派生列表中指定多个基类而定义多个基类，每个基类需要一个单独的访问标号。

namespace (命名空间)

Mechanism for gathering all the names defined by a library or other collection of programs into a single scope. Unlike other scopes in C++, a namespace scope may be defined in several parts. The namespace may be opened and closed and reopened again in disparate parts of the program.

将一个库或其他程序集合定义的所有名字聚焦到单个作用域的机制。与 C++ 中其他作用域不同，命名空间作用域可以在几个部分中定义，在程序的不同部分，命名空间可以是打开的、关闭的和重新打开的。

namespace alias (命名空间别名)

Mechanism for defining a synonym for a given namespace:

为给定命名空间定义同义词的机制。

```
namespace N1 = N;
```

defines `N1` as another name for the name-space named `N`. A namespace can have multiple aliases, and the namespace name or one of its aliases may be used interchangeably.

将 `N1` 定义为名为 `N` 的命名空间的另一名字。一个命名空间可以有多个别名，并且命名空间名字和它的别名可以互换使用。

namespace pollution (命名空间污染)

Term used to describe what happens when all the names of classes and functions are placed in the global namespace. Large programs that use code written by multiple independent parties often encounter collisions among names if these names are global.

用来描述类和函数的所有名字放在全局命名空间时发生什么情况的术语。如果名字是全局的，则使用由多个独立团队编写的代码的大程序经常遇到名字冲突。

raise (引发)

Often used as a synonym for throw. C++ programmers speak of "throwing" or "raising" an exception interchangably.

经常用作抛出的同义词。C++ 程序员互换地使用“抛出”异常或“引发”异常。

rethrow (重新抛出)

An empty `throw` that does not specify an expression. A rethrow is valid only from inside a catch clause, or in a function called directly or indirectly from a `catch`. Its effect is to rethrow the exception object that it received.

一个空的 `throw`——没有指定 `throw`。只有捕获子句或者从 `catch` 直接或间接调用的函数中的重新抛出才有效，其效果是将接到的异常对象重新抛出。

scope operator (作用域操作符)

Operator used to access names from a namespace or a class.

用于访问命名空间或类中名字的操作符。

stack unwinding (栈展开)

Term used to describe the process whereby the functions leading to a thrown exception are exited in the search for a `catch`. Local objects constructed before the exception are destroyed before entering the corresponding `catch`.

用于描述在查找 `catch` 时退出引起被抛出异常的函数的过程。在进入相应 `catch` 之前，撤销在异常之前构造的局部对象。

terminate

Library function that is called if an exception is not caught or if an exception occurs while a handler is in process. Usually calls `abort` to end the program.

一个库函数。如果没有捕获到异常或者在异常处理过程中发生异常，就调用这个库函数。该函数通常调用 `abort` 函数来结束程序。

throw e

Expression that interrupts the current execution path. Each `throw` transfers control to the nearest enclosing `catch` clause that can handle the type of exception that is thrown. The expression `e` is copied into the exception object.

中断当前执行路径的表达式。每个 `throw` 将控制转到可以处理被抛出异常类型的最近的外围 `catch` 子句，表达式 `e` 被复制到异常对象。

[try block \(测试块\)](#)

A block of statements enclosed by the keyword `try` and one or more catch clauses. If the code inside the try block raises an exception and one of the catch clauses matches the type of the exception, then the exception is handled by that catch. Otherwise, the exception is passed out of the `try` to a catch further up the call chain.

由关键字 `try` 以及一个或多个 `try` 子句包围的语句块。如果 `try` 块内部的代码引发一个异常，而一个 `catch` 子句与异常的类型匹配，则由该 `catch` 处理异常；否则，将异常传出 `try` 之外，传给调用链中更上层的 `catch`。

[unexpected](#)

Library function that is called if an exception is thrown that violates the exception specification of a function.

一个库函数，如果被抛出异常违反函数的异常说明，就调用该函数。

[unnamed namespace \(未命名的命名空间\)](#)

A namespace that is defined without a name. Names defined in an unnamed namespace may be accessed directly without use of the scope operator. Each file has its own unique unnamed namespace. Names in the file are not visible outside that file.

没有定义名字的命名空间。未命名的命名空间中定义的名字可以无须使用作用域操作符而直接访问。每个文件都具有自己的未命名的命名空间，文件中的名字在该文件之外不可见。

[using declaration \(using 声明\)](#)

Mechanism to inject a single name from a namespace into the current scope:

将命名空间中单个名字注入当前作用域的机制。

```
using std::cout;
```

makes the name `cout` from the namespace `std` available in the current scope. The name `cout` can be used without the `std::` qualifier.

使得命名空间 `std` 中的名字 `cout` 在当前作用域中可见，可以无须限定符 `std::` 而使用名字 `cout`。

[using directive \(using 指示\)](#)

Mechanism for making *all* the names in a namespace available in the nearest scope containing both the using directive and the namespace itself.

使一个命名空间中的所有名字在 `using` 指示和命名空间本身的最近作用域中可见的机制。

[virtual base class \(虚基类\)](#)

A base class that was inherited using the `virtual` keyword. A virtual base part occurs only once in a derived object even if the same class appears as a virtual base more than once in the hierarchy. In nonvirtual inheritance a constructor may only initialize its immediate base class(es). When a class is inherited virtually, that class is initialized by the most derived class, which therefore should include an initializer for all of its virtual parent(s).

使用关键字 `virtual` 继承的基类。即使同一类在层次中作为虚基类出现多次，派生类对象中的虚基类部分也只出现一次。在非虚继承中，构造函数只能初始化自己的直接基类，当对一个类进行虚继承的时候，由最低层的派生类初始化那个类，因此最低层的派生类应包含用于其所有虚父类的初始化式。

[virtual inheritance \(虚继承\)](#)

Form of multiple inheritance in which derived classes share a single copy of a base that is included in the hierarchy more than once.

多重继承的形式，这种形式中，派生类共享在层次中被包含多次的基类的一个副本。

Chapter 18. Specialized Tools and Techniques

第十八章 特殊工具与技术

CONTENTS

<u>Section 18.1 Optimizing Memory Allocation</u>	<u>754</u>
<u>Section 18.2 Run-Time Type Identification</u>	<u>772</u>
<u>Section 18.3 Pointer to Class Member</u>	<u>780</u>
<u>Section 18.4 Nested Classes</u>	<u>786</u>
<u>Section 18.5 Union: A Space-Saving Class</u>	<u>792</u>
<u>Section 18.6 Local Classes</u>	<u>796</u>
<u>Section 18.7 Inherently Nonportable Features</u>	<u>797</u>
<u>Chapter Summary</u>	<u>805</u>
<u>Defined Terms</u>	<u>805</u>

The first four parts of this book discussed how to use those parts of C++ that are generally useful. Those features are likely to be used at some point by almost all C++ programmers. In addition, C++ defines some features that are more specialized. Many programmers will never (or only rarely) need to use these features presented in this chapter.

本书前四部分讨论了怎样使用 C++ 中有普遍用途的特征，几乎所有 C++ 程序员都有可能在某些时候使用这些特征。此外，C++ 还定义了一些更为特殊的特征，许多程序员从不（或者很少）需要使用本章所介绍的这些特征。

18.1. Optimizing Memory Allocation

18.1. 优化内存分配

Memory allocation in C++ is a typed operation: `new` ([Section 5.11](#), p. 174) allocates memory for a specified type and constructs an object of that type in the newly allocated memory. A `new` expression automatically runs the appropriate constructor to initialize each dynamically allocated object of class type.

C++ 的内存分配是一种类型化操作：`new` ([第 5.11 节](#)) 为特定类型分配内存，并在新分配的内存中构造该类型的一个对象。`new` 表达式自动运行合适的构造函数来初始化每个动态分配的类类型对象。

The fact that `new` allocates memory on a per-object basis can impose unacceptable run-time overhead on some classes. Such classes might need to make user-level allocation of objects of the class type faster. A common strategy such classes use is to preallocate memory to be used when new objects are created, constructing each new object in preallocated memory as needed.

`new` 基于每个对象分配内存的事实可能会对某些类强加不可接受的运行时开销，这样的类可能需要使用用户级的类类型对象分配能够更快一些。这样的类使用的通用策略是，预先分配用于创建新对象的内存，需要时在预先分配的内存中构造每个新对象。

Other classes want to minimize allocations needed for their own data members. For example, the library `vector` class preallocates ([Section 9.4](#), p. 330) extra memory to hold additional elements if and when they are added. New elements are added in this reserved capacity. Preallocating elements allows `vector` to efficiently add elements while keeping the elements in contiguous memory.

另外一些类希望按最小尺寸为自己的数据成员分配需要的内存。例如，标准库中的 `vector` 类预先分配 ([第 9.4 节](#)) 额外内存以保存加入的附加元素，将新元素加入到这个保留容量中。将元素保持在连续内存中的时候，预先分配的元素使 `vector` 能够高效地加入元素。

In each of these cases preallocating memory to hold user-level objects or to hold the internal data for a class there is the need to decouple memory allocation from object construction. The obvious reason to decouple allocation and construction is that constructing objects in preallocated memory is wasteful. Objects may be created that are never used. Those objects that are used must be reassigned new values when the preallocated object is actually used. More subtly, some classes could not use preallocated memory if it had to be constructed. As an example, consider `vector`, which uses a preallocation strategy. If objects in preallocated memory had to be constructed, then it would not be possible to have `vector`s of types that do not have a default constructor there would be no way for `vector` to know how to construct these objects.

在每种情况下（预先分配内存以保存用户级对象或者保存类的内部数据）都需要将内存分配与对象构造分离开。将内存分配与对象构造分离开的明显的理由是，在预先分配的内存中构造对象很浪费，可能会创建从不使用的对象。当实际使用预先分配的对象的时候，被使用的对象必须重新赋以新值。更微妙的是，如果预先分配的内存必须被构造，某些类就不能使用它。例如，考虑 `vector`，它使用了预先分配策略。如果必须构造预先分配的内存中的对象，就不能有基类型为没有默认构造函数的 `vector`——`vector` 没有办法知道怎样构造这些对象。



The techniques presented in this section are not guaranteed to make all programs faster. Even when they do improve performance, they may carry other costs such as space usage or difficulty of debugging. It is always best to defer optimization until the program is known to work and when run-time measurements indicate that improving memory allocation will solve known performance problems.

本节提出的技术不保证使所有程序更快。即使它们确实能改善性能，也可能带来其他开销，如空间的使用或调试困难。最好将优化推迟到已知程序能够工作，并且运行时测试指出改进内存分配将解决已知的性能问题的时候。

18.1.1. Memory Allocation in C++

18.1.1. C++ 中的内存分配

In C++ memory allocation and object construction are closely intertwined, as are object destruction and memory deallocation. When we use a `new` expression, memory is allocated, and an object is constructed in that memory. When we use a `delete` expression, a destructor is called to destroy the object and the memory used by the object is returned to the system.

C++ 中，内存分配和对象构造紧密纠缠，就像对象和内存回收一样。使用 `new` 表达式的时候，分配内存，并在该内存中构造一个对象；使用 `delete` 表达式的时候，调用析构函数撤销对象，并将对象所用内存返还给系统。

When we take over memory allocation, we must deal with both these tasks. When we allocate raw memory, we must construct object(s) in that memory. Before freeing that memory, we must ensure that the objects are properly destroyed.

接管内存分配时，必须处理这两个任务。分配原始内存时，必须在该内存中构造对象；在释放该内存之前，必须保证适当地撤销这些对象。

Assigning to an object in unconstructed memory rather than initializing it is undefined. For many classes,



doing so causes a crash at run time. Assignment involves obliterating the existing object. If there is no existing object, then the actions in the assignment operator can have disastrous effects.

对未构造的内存中的对象进行赋值而不是初始化，其行为是未定义的。对许多类而言，这样做引起运行时崩溃。赋值涉及删除现有对象，如果没有现存对象，赋值操作符中的动作就会有灾难性效果。

C++ provides two ways to allocate and free unconstructed, raw memory:

C++ 提供下面两种方法分配和释放未构造的原始内存。

1. The `allocator class`, which provides type-aware memory allocation. This class supports an abstract interface to allocating memory and subsequently using that memory to hold objects.

`allocator` 类，它提供可感知类型的内存分配。这个类支持一个抽象接口，以分配内存并随后使用该内存保存对象。

2. The library `operator new` and `operator delete` functions, which allocate and free raw, untyped memory of a requested size.

标准库中的 `operator new` 和 `operator delete`，它们分配和释放需要大小的原始的、未类型化的内存。

C++ also provides various ways to construct and destroy objects in raw memory:

C++ 还提供不同的方法在原始内存中构造和撤销对象。

1. The `allocator` class defines members named `construct` and `destroy`, which operate as their names suggest. The `construct` member initializes objects in unconstructed memory; the `destroy` member runs the appropriate destructor on objects.

`allocator` 类定义了名为 `construct` 和 `destroy` 的成员，其操作正如它们的名字所指出的那样：`construct` 成员在未构造内存中初始化对象，`destroy` 成员在对象上运行适当的析构函数。

2. The placement `new` expression takes a pointer to unconstructed memory and initializes an object or an array in that space.

定位 `new` 表达式接受指向未构造内存的指针，并在该空间中初始化一个对象或一个数组。

3. We can directly call an object's destructor to destroy the object. Running the destructor does not free the memory in which the object resides.

可以直接调用对象的析构函数来撤销对象。运行析构函数并不释放对象所在的内存。

4. The algorithms `uninitialized_fill` and `uninitialized_copy` execute like the `fill` and `copy` algorithms except that they construct objects in their destination rather than assigning to them.

算法 `uninitialized_fill` 和 `uninitialized_copy` 像 `fill` 和 `copy` 算法一样执行，除了它们的目的地构造对象而不是给对象赋值之外。

Modern C++ programs ordinarily ought to use the `allocator` class to allocate memory. It is safer and more flexible. However, when constructing objects, the placement `new` expression is more flexible than the `allocator::construct` member. There are some cases where placement `new` must be used.

现代 C++ 程序一般应该使用 `allocator` 类来分配内存，它更安全更灵活。但是，在构造对象的时候，用 `new` 表达式比 `allocator::construct` 成员更灵活。有几种情况下必须使用 `new`。

18.1.2. The `allocator` Class

18.1.2. `allocator` 类

The `allocator` class is a template that provides typed memory allocation and object construction and destruction. [Table 18.1](#) on the following page outlines the operations that `allocator` supports.

`allocator` 类是一个模板，它提供类型化的内存分配以及对象构造与撤销。[表 18.1](#) 概述了 `allocator` 类支持的操作。

Table 18.1. Standard `allocator` Class and Customized Algorithms

表 18.1. 标准 `allocator` 类与定制算法

<code>allocator<T> a;</code>	Defines an <code>allocator</code> object named <code>a</code> that can allocate
------------------------------------	---

Section 18.1. Optimizing Memory Allocation

	memory or construct objects of type <code>T</code> .
	定义名为 <code>a</code> 的 <code>allocator</code> 对象，可以分配内存或构造 <code>T</code> 类型的对象
<code>a.allocate(n)</code>	Allocates raw, unconstructed memory to hold <code>n</code> objects of type <code>T</code> . 分配原始的未构造内存以保存 <code>T</code> 类型的 <code>n</code> 个对象
<code>a.deallocate(p, n)</code>	Deallocates memory that held <code>n</code> objects of type <code>T</code> starting at address contained in the <code>T*</code> pointer named <code>p</code> . It is the user's responsibility to run <code>destroy</code> on any objects that were constructed in this memory before calling <code>deallocate</code> . 释放内存，在名为 <code>p</code> 的 <code>T*</code> 指针中包含的地址处保存 <code>T</code> 类型的 <code>n</code> 个对象。运行调用 <code>deallocate</code> 之前在该内存中构造的任意对象的 <code>destroy</code> 是用户的责任
<code>a.construct(p, t)</code>	Constructs a new element in the memory pointed to by the <code>T*</code> pointer <code>p</code> . The copy constructor of type <code>T</code> is run to initialize the object from <code>t</code> . 在 <code>T*</code> 指针 <code>p</code> 所指内存中构造一个新元素。运行 <code>T</code> 类型的复制构造函数用 <code>t</code> 初始化该对象
<code>a.destroy(p)</code>	Runs the destructor on the object pointed to by the <code>T*</code> pointer <code>p</code> . 运行 <code>T*</code> 指针 <code>p</code> 所指对象的析构函数
<code>uninitialized_copy(b, e, b2)</code>	Copies elements from the input range denoted by iterators <code>b</code> and <code>e</code> into unconstructed, raw memory beginning at iterator <code>b2</code> . The function constructs elements in the destination, rather than assigning them. The destination denoted by <code>b2</code> is assumed large enough to hold a copy of the elements in the input range. 从迭代器 <code>b</code> 和 <code>e</code> 指出的输入范围将元素复制到从迭代器 <code>b2</code> 开始的未构造的原始内存中。该函数在目的地构造元素，而不是给它们赋值。假定由 <code>b2</code> 指出的目的地足以保存输入范围内元素的副本
<code>uninitialized_fill(b, e, t)</code>	Initializes objects in the range denoted by iterators <code>b</code> and <code>e</code> as a copy of <code>t</code> . The range is assumed to be unconstructed, raw memory. The objects are constructed using the copy constructor. 将由迭代器 <code>b</code> 和 <code>e</code> 指出的范围中的对象初始化为 <code>t</code> 的副本。假定该范围是未构造的原始内存。使用复制构造函数构造对象
<code>uninitialized_fill_n(b, e, t, n)</code>	Initializes at most an integral number <code>n</code> objects in the range denoted by iterators <code>b</code> and <code>e</code> as a copy of <code>t</code> . The range is assumed to be at least <code>n</code> elements in size. The objects are constructed using the copy constructor. 将由迭代器 <code>b</code> 和 <code>e</code> 指出的范围内至多 <code>n</code> 个对象初始化为 <code>t</code> 的副本。假定范围至少为 <code>n</code> 个元素大小。使用复制构造函数构造对象

The `allocator` class separates allocation and object construction. When an `allocator` object allocates memory, it allocates space that is appropriately sized and aligned to hold objects of the given type. However, the memory it allocates is unconstructed. Users of `allocator` must separately `construct` and `destroy` objects placed in the memory it allocates.

`allocator` 类将内存分配和对象构造分开。当 `allocator` 对象分配内存的时候，它分配适当大小并排列成保存给定类型对象的空间。但是，它分配的内存是未构造的，`allocator` 的用户必须分别 `construct` 和 `destroy` 放置在该内存中的对象。

Using `allocator` to Manage Class Member Data

Section 18.1. Optimizing Memory Allocation

使用 `allocator` 管理类成员数据

To understand how we might use a preallocation strategy and the `allocator` class to manage the internal data needs of a class, let's think a bit more about how memory allocation in the `vector` class might work.

为了理解可以怎样使用预分配策略以及 `allocator` 类来管理类的内部数据需要，让我们再想想 `vector` 类中的内存分配会怎样工作。

Recall that the `vector` class stores its elements in contiguous storage. To obtain acceptable performance, `vector` preallocates more elements than are needed ([Section 9.4](#), p. 330). Each `vector` member that adds elements to the container checks whether there is space available for another element. If so, then the member initializes an object in the next available spot in preallocated memory. If there isn't a free element, then the `vector` is reallocated: The `vector` obtains new space, copies the existing elements into that space, adds the new element, and frees the old space.

回忆一下，`vector` 类将元素保存在连续的存储中。为了获得可接受的性能，`vector` 预先分配比所需元素更多的元素（[第 9.4 节](#)）。每个将元素加到容器中的 `vector` 成员检查是否有可用空间以容纳另一元素。如果有，该成员在预分配内存中下一可用位置初始化一个对象；如果没有自由元素，就重新分配 `vector`: `vector` 获取新的空间，将现有元素复制到空间，增加新元素，并释放旧空间。

The storage that `vector` uses starts out as unconstructed memory; it does not yet hold any objects. When the elements are copied to or added in this preallocated space, they must be constructed using the `construct` member of `allocator`.

`vector` 所用存储开始是未构造内存，它还没有保存任何对象。将元素复制或增加到这个预分配空间的时候，必须使用 `allocator` 类的 `construct` 成员构造元素。

To illustrate these concepts we'll implement a small portion of `vector`. We'll name our class `Vector` to distinguish it from the standard `vector` class:

为了说明这些概念，我们将实现 `vector` 的一小部分。将我们的类命名为 `Vector`，以区别于标准类 `vector`:

```
// pseudo-implementation of memory allocation strategy for a vector-like class
template <class T> class Vector {
public:
    Vector(): elements(0), first_free(0), end(0) { }
    void push_back(const T&);
    // ...
private:
    static std::allocator<T> alloc; // object to get raw memory
    void reallocate(); // get more space and copy existing elements
    T* elements; // pointer to first element in the array
    T* first_free; // pointer to first free element in the array
    T* end; // pointer to one past the end of the array
    // ...
};
```

Each `Vector<T>` type defines a `static` data member of type `allocator<T>` to allocate and construct the elements in `vectors` of the given type. Each `Vector` object keeps its elements in a built-in array of the indicated type and maintains three pointers into that array:

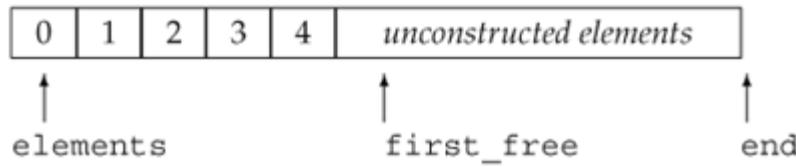
- 每个 `Vector<T>` 类型定义一个 `allocator<T>` 类型的 `static` 数据成员，以便在给定类型的 `Vector` 中分配和构造元素。每个 `Vector` 对象在指定类型的内置数组中保存其元素，并维持该数组的下列三个指针：
- `elements`, which points to the first element in the array
`elements`, 指向数组的第一个元素。
 - `first_free`, which points just after the last actual element
`first_free`, 指向最后一个实际元素之后的那个元素。
 - `end`, which points just after the end of the array itself
`end`, 指向数组本身之后的那个元素。

[Figure 18.1](#) illustrates the meaning of these pointers.

[图 18.1](#) 说明了这些指针的含义。

Figure 18.1. `vector` Memory Allocation Strategy

图 18.1. `vector` 内存分配策略



We can use these pointers to determine the size and capacity of the `Vector`:

可以使用这些指针来确定 `Vector` 的大小和容量:

- The `size` of a `Vector` (the number of elements actually in use) is equal to `first_free - elements`.
`Vector` 的 `size` (实际使用的元素的数目) 等于 `first_free - elements`。
- The `capacity` of a `Vector` (the total number of elements that could be defined before the `Vector` has to be reallocated) is equal to `end - elements`.
`Vector` 的 `capacity` (在必须重新分配 `Vector` 之前, 可以定义的元素的总数) 等于 `end - elements`。
- The free capacity (the number of elements that can be added before a reallocation is necessary) is `end - first_free`.
自由空间 (在需要重新分配之前, 可以增加的元素的数目) 是 `end - first_free`。

Using `construct`

使用 `construct`

The `push_back` member uses these pointers to add a new element to the end of the `Vector`:

`push_back` 成员使用这些指针将新元素加到 `Vector` 末尾:

```
template <class T>
void Vector<T>::push_back(const T& t)
{
    // are we out of space?
    if (first_free == end)
        reallocate(); // gets more space and copies existing elements to it
    alloc.construct(first_free, t);
    ++first_free;
}
```

The `push_back` function starts by determining whether there is space available. If not, it calls `reallocate`. That member allocates new space and copies the existing elements. It resets the pointers to point to the newly allocated space.

`push_back` 函数首先确定是否有可用空间, 如果没有, 就调用 `reallocate` 函数, `reallocate` 分配新空间并复制现存元素, 将指针重置为指向新分配的空间。

Once `push_back` knows that there is room for the new element, it asks the `allocator` object to `construct` a new last element. The `construct` function uses the copy constructor for type `T` to copy the value `t` into the element denoted by `first_free`. It then increments `first_free` to indicate that one more element is in use.

一旦 `push_back` 函数知道还有空间容纳新元素, 它就请求 `allocator` 对象构造一个新的最后元素。`construct` 函数使用类型 `T` 的复制构造函数将 `t` 值复制到由 `first_free` 指出的元素, 然后, 将 `first_free` 加 1 以指出又有一个元素在用。

Reallocating and Copying Elements

重新分配元素与复制元素

The `reallocate` function has the most work to do:

`reallocate` 函数所做的工作最多:

```
template <class T> void Vector<T>::reallocate()
{
    // compute size of current array and allocate space for twice as many elements
    std::ptrdiff_t size = first_free - elements;
```

Section 18.1. Optimizing Memory Allocation

```
std::ptrdiff_t newcapacity = 2 * max(size, 1);
// allocate space to hold newcapacity number of elements of type T
T* newelements = alloc.allocate(newcapacity);

// construct copies of the existing elements in the new space
uninitialized_copy(elements, first_free, newelements);
// destroy the old elements in reverse order
for (T *p = first_free; p != elements; /* empty */)
    alloc.destroy(--p);

// deallocate cannot be called on a 0 pointer
if (elements)
    // return the memory that held the elements
    alloc.deallocate(elements, end - elements);
// make our data structure point to the new elements
elements = newelements;
first_free = elements + size;
end = elements + newcapacity;
}
```

We use a simple but surprisingly effective strategy of allocating twice as much memory each time we reallocate. The function starts by calculating the current number of elements in use, doubling that number, and asking the `allocator` object to obtain the desired amount of space. If the `Vector` is empty, we allocate two elements.

我们使用一个简单但效果惊人的策略：每次重新分配时分配两倍内存。函数首先计算当前在用的元素数目，将该数目翻倍，并请求 `allocator` 对象来获得所需数量的空间。如果 `Vector` 为空，就分配两个元素。

If `Vector` holds `ints`, the call to `allocate` allocates space for `newcapacity` number of `ints`. If it holds `strings`, then it allocates that space for the given number of `strings`.

如果 `Vector` 保存 `int` 值，`allocate` 函数调用为 `newcapacity` 数目的 `int` 值分配空间；如果 `Vector` 保存 `string` 对象，它就为给定数目的 `string` 对象分配空间。

The call to `uninitialized_copy` uses a specialized version of the standard `copy` algorithm. This version expects its destination to be raw, unconstructed memory. Rather than assigning elements from the input range to the destination, it copy-constructs each element in the destination. The `copy` constructor for `T` is used to copy each element from the input range to the destination.

`uninitialized_copy` 调用使用标准 `copy` 算法的特殊版本。这个版本希望目的地是原始的未构造内存，它在目的地复制构造每个元素，而不是将输入范围的元素赋值给目的地，使用 `T` 的复制构造函数从输入范围将每个元素复制到目的地。

The `for` loop calls the `allocator` member `destroy` on each object in the old array. It destroys the elements in reverse order, starting with the last element in the array and finishing with the first. The `destroy` function runs the destructor for type `T` to free any resources used by the old elements.

`for` 循环对旧数组中每个对象调用 `allocator` 的 `destroy` 成员它按逆序撤销元素，从数组中最后一个元素开始，以第一个元素结束。`destroy` 函数运行 `T` 类型的析构函数来释放旧元素所用的任何资源。

Once the elements have been copied and destroyed, we free the space used by the original array. We must check that `elements` actually pointed to an array before calling `deallocate`.

一旦复制和撤销了元素，就释放原来数组所用的空间。在调用 `deallocate` 之前，必须检查 `elements` 是否实际指向一个数组。



`deallocate` expects a pointer that points to space that was allocated by `allocate`. It is not legal to pass `deallocate` a zero pointer.
`deallocate` 期待指向由 `allocate` 分配的空间的指针，传给 `deallocate` 一个零指针是不合法的。

Finally, we have to reset the pointers to address the newly allocated and initialized array. The `first_free` and `end` pointers are set to denote one past the last constructed element and one past the end of the allocated space, respectively.

最后，必须重置指针以指向新分配并初始化的数组。将 `first_free` 和 `end` 指针分别置为指向最后构造的元素之后的单元以及所分配空间末尾的下一单元。

Exercises Section 18.1.2

Exercise 18.1: Implement your own version of the `Vector` class including versions of the `vector` members `reserve` (Section 9.4, p. 330), `resize` (Section 9.3.5, p. 323), and the `const` and non`const` subscript operators (Section 14.5, p. 522).

实现自己的 `Vector` 类的版本，包括 `vector` 成员 `reserve` (第 9.4 节)、`resize` (第 9.3.5 节) 以及 `const` 和非 `const` 下标操作符 (第 14.5 节)。

Exercise 18.2: Define a `typedef` that uses the corresponding pointer type as the `iterator` for your `Vector`.
定义一个类型别名，使用对应指针类型作为 `Vector` 的 `iterator`。

Section 18.1. Optimizing Memory Allocation

Exercise 18.3: To test your `vector` class, reimplement earlier programs you wrote using `vector` to use `vector` instead.

为了测试你的 `vector` 类, 重新实现前面用 `vector` 编写的程序, 用 `vector` 代替 `vector`。

18.1.3. `operator new` and `operator delete` Functions

18.1.3. `operator new` 函数和 `operator delete` 函数

The previous subsection used the `vector` class to show how we could use the `allocator` class to manage a pool of memory for a class' internal data storage. In the next three subsections we'll look at how we might implement the same strategy using the more primitive library facilities.

前几节使用 `vector` 类说明了怎样使用 `allocator` 类来管理用于类的内部数据存储的内存池, 下面三节将介绍怎样用更基本的标准库机制实现相同的策略。

First, we need to understand a bit more about how `new` and `delete` expressions work. When we use a `new` expression

首先, 需要对 `new` 和 `delete` 表达式怎样工作有更多的理解。当使用 `new` 表达式

```
// new expression
string * sp = new string("initialized");
```

three steps actually take place. First, the expression calls a library function named `operator new` to allocate raw, untyped memory large enough to hold an object of the specified type. Next, a constructor for the type is run to construct the object from the specified initializers. Finally, a pointer to the newly allocated and constructed object is returned.

的时候, 实际上发生三个步骤。首先, 该表达式调用名为 `operator new` 的标准库函数, 分配足够大的原始的未类型化的内存, 以保存指定类型的一个对象; 接下来, 运行该类型的一个构造函数, 用指定初始化式构造对象; 最后, 返回指向新分配并构造的对象的指针。

When we use a `delete` expression to delete a dynamically allocated object:

当使用 `delete` 表达式

```
delete sp;
```

two steps happen. First, the appropriate destructor is run on the object to which `sp` points. Then, the memory used by the object is freed by calling a library function named `operator delete`.

删除动态分配对象的时候, 发生两个步骤。首先, 对 `sp` 指向的对象运行适当的析构函数; 然后, 通过调用名为 `operator delete` 的标准库函数释放该对象所用内存。

Terminology: `new` Expression versus `operator new` Function

术语对比: `new` 表达式与 `operator new` 函数

The library functions `operator new` and `operator delete` are misleadingly named. Unlike other `operator` functions, such as `operator=`, these functions do not overload the `new` or `delete` expressions. In fact, we cannot redefine the behavior of the `new` and `delete` expressions.

标准库函数 `operator new` 和 `operator delete` 的命名容易让人误解。与其他 `operator` 函数 (如 `operator=`) 不同, 这些函数没有重载 `new` 或 `delete` 表达式, 实际上, 我们不能重定义 `new` 和 `delete` 表达式的行为。

A `new` expression executes by calling an `operator new` function to obtain memory and then constructs an object in that memory. A `delete` expression executes by destroying an object and then calls an `operator delete` function to free the memory used by the object.

通过调用 `operator new` 函数执行 `new` 表达式获得内存, 并接着在该内存中构造一个对象, 通过撤销一个对象执行 `delete` 表达式, 并接着调用 `operator delete` 函数, 以释放该对象使用的内存。

Because the `new` (or `delete`) expressions and the underlying library functions have the same name, it is easy to confuse the two.

因为 `new` (或 `delete`) 表达式与标准库函数同名, 所以二者容易混淆。



The `operator new` and `operator delete` Interface

`operator new` 和 `operator delete` 接口

There are two overloaded versions of `operator new` and `operator delete` functions. Each version supports the related `new` and `delete` expression:

`operator new` 和 `operator delete` 函数有两个重载版本，每个版本支持相关的 `new` 表达式和 `delete` 表达式：

```
void *operator new(size_t);           // allocate an object
void *operator new[](size_t);         // allocate an array

void *operator delete(void*);         // free an object
void *operator delete[](void*);       // free an array
```

Using the Allocation Operator Functions

使用分配操作符函数

Although the `operator new` and `operator delete` functions are intended to be used by `new` expressions, they are generally available functions in the library. We can use them to obtain unconstructed memory. They are somewhat analogous to the `allocate` and `deallocate` members of the `allocator` class. For example, instead of using an `allocator`, we could have used the `operator new` and `operator delete` functions in our `Vector` class. When we allocated new space we wrote

虽然 `operator new` 和 `operator delete` 函数的设计意图是供 `new` 表达式使用，但它们通常是标准库中的可用函数。可以使用它们获得未构造内存，它们有点类似 `allocate` 类的 `allocator` 和 `deallocate` 成员。例如，代替使用 `allocator` 对象，可以在 `vector` 类中使用 `operator new` 和 `operator delete` 函数。在分配新空间时我们曾编写

```
// allocate space to hold newcapacity number of elements of type T
T* newelements = alloc.allocate(newcapacity);
```

which could be rewritten as

这可以重新编写为

```
// allocate unconstructed memory to hold newcapacity elements of type T
T* newelements = static_cast<T*>
    (operator new[](newcapacity * sizeof(T)));
```

Similarly, when we deallocated the old space pointed to be the `Vector` member `elements` we wrote

类似地，在重新分配由 `Vector` 成员 `elements` 指向的旧空间的时候，我们曾经编写

```
// return the memory that held the elements
alloc.deallocate(elements, end - elements);
```

which could be rewritten as

这可以重新编写为

```
// deallocate the memory that they occupied
operator delete[](elements);
```

These functions behave similarly to the `allocate` and `deallocate` members of the `allocator` class. However, they differ in one important respect: They operate on `void*` pointers rather than typed pointers.

这些函数的表现与 `allocate` 类的 `allocator` 和 `deallocate` 成员类似。但是，它们在一个重要方面有不同：它们在 `void*` 指针而不是类型化的指针上进行操作。



In general, it is more type-safe to use an `allocator` rather than using the `operator new` and `operator delete` functions directly.

一般而言，使用 `allocator` 比直接使用 `operator new` 和 `operator delete` 函数更为类型安全。

Section 18.1. Optimizing Memory Allocation

The `allocate` member allocates typed memory, so programs that use it can avoid the necessity of calculating the byte-count amount of memory needed. They also can avoid casting ([Section 5.12.4](#), p. 183) the return from `operator new`. Similarly, `deallocate` frees memory of a specific type, again avoiding the necessity for converting to `void*`.

`allocate` 成员分配类型化的内存，所以使用它的程序可以不必计算以字节为单位的所需内存量，它们也可以避免对 `operator new` 的返回值进行强制类型转换（[第 5.12.4 节](#)）。类似地，`deallocate` 释放特定类型的内存，也不必转换为 `void*`。

18.1.4. Placement `new` Expressions

18.1.4. 定位 `new` 表达式

The library functions `operator new` and `operator delete` are lower-level versions of the `allocator` members `allocate` and `deallocate`. Each allocates but does not initialize memory.

标准库函数 `operator new` 和 `operator delete` 是 `allocator` 的 `allocate` 和 `deallocate` 成员的低级版本，它们都分配但不初始化内存。

There are also lower-level alternatives to the `allocator` members `construct` and `destroy`. These members initialize and destroy objects in space allocated by an `allocator` object.

`allocator` 的成员 `construct` 和 `destroy` 也有两个低级选择，这些成员在由 `allocator` 对象分配的空间中初始化和撤销对象。

Analogous to the `construct` member, there is a third kind of `new` expression, referred to as `placement new`. The placement `new` expression initializes an object in raw memory that was already allocated. It differs from other versions of `new` in that it does not allocate memory. Instead, it takes a pointer to allocated but unconstructed memory and initializes an object in that memory. In effect, placement `new` allows us to construct an object at a specific, preallocated memory address.

类似于 `construct` 成员，有第三种 `new` 表达式，称为[定位 `new`](#)。定位 `new` 表达式在已分配的原始内存中初始化一个对象，它与 `new` 的其他版本的不同之处在于，它不分配内存。相反，它接受指向已分配但未构造内存的指针，并在该内存中初始化一个对象。实际上，定位 `new` 表达式使我们能够在特定的、预分配的内存地址构造一个对象。

The form of a placement `new` expression is:

定位 `new` 表达式的形式是：

```
new (place_address) type  
new (place_address) type (initializer-list)
```

where `place_address` must be a pointer and the `initializer-list` provides (a possibly empty) list of initializers to use when constructing the newly allocated object.

其中 `place_address` 必须是一个指针，而 `initializer-list` 提供了（可能为空的）初始化列表，以便在构造新分配的对象时使用。

We could use a placement `new` expression to replace the call to `construct` in our `vector` implementation. The original code

可以使用定位 `new` 表达式代替 `vector` 实现中的 `construct` 调用。原来的代码

```
// construct a copy t in the element to which first_free points  
alloc.construct(first_free, t);
```

would be replaced by the equivalent placement `new` expression

可以用等价的定位 `new` 表达式代替

```
// copy t into element addressed by first_free  
new (first_free) T(t);
```

Placement `new` expressions are more flexible than the `construct` member of class `allocator`. When placement `new` initializes an object, it can use any constructor, and builds the object directly. The `construct` function always uses the copy constructor.

定位 `new` 表达式比 `allocator` 类的 `construct` 成员更灵活。定位 `new` 表达式初始化一个对象的时候，它可以使用任何构造函数，并直接建立对象。`construct` 函数总是使用复制构造函数。

For example, we could initialize an allocated but unconstructed `string` from a pair of iterators in either of these two ways:

例如，可以用下面两种方式之一，从一对迭代器初始化一个已分配但未构造的 `string` 对象：

```
allocator<string> alloc;  
string *sp = alloc.allocate(2); // allocate space to hold 2 strings  
// two ways to construct a string from a pair of iterators  
new (sp) string(b, e);           // construct directly in place  
alloc.construct(sp + 1, string(b, e)); // build and copy a temporary
```

The placement `new` expression uses the `string` constructor that takes a pair of iterators to construct the `string` directly in the space to which `sp` points. When we call `construct`, we must first construct the `string` from the iterators to get a `string` object to pass to `construct`. That function then uses the `string` copy constructor to copy that unnamed, temporary `string` into the object to which `sp` points.

定位 `new` 表达式使用了接受一对迭代器的 `string` 构造函数，在 `sp` 指向的空间直接构造 `string` 对象。当调用 `construct` 函数的时候，必须首先从迭代器构造一个 `string`

Section 18.1. Optimizing Memory Allocation

对象，以获得传递给 `construct` 的 `string` 对象，然后，该函数使用 `string` 的复制构造函数，将那个未命名的临时 `string` 对象复制到 `sp` 指向的对象中。

Often the difference is irrelevant: For valuelike classes, there is no observable difference between constructing the object directly in place and constructing a temporary and copying it. And the performance difference is rarely meaningful. But for some classes, using the copy constructor is either impossible (because the copy constructor is private) or should be avoided. In these cases, use of placement `new` may be necessary.

通常，这些区别是不相干的：对值型类而言，在适当的位置直接构造对象与构造临时对象并进行复制之间没有可观察到的区别，而且性能差别基本没有意义。但对某些类而言，使用复制构造函数是不可能的（因为复制构造函数是私有的），或者是应该避免的，在这种情况下，也许有必要使用定位 `new` 表达式。

Exercises Section 18.1.4

Exercise

18.4: Why do you think `construct` is limited to using only the copy constructor for the element type?

你认为为什么限制 `construct` 函数只能使用元素类型的复制构造函数？

Exercise

18.5: Why can placement `new` expressions be more flexible?

为什么定位 `new` 表达式更灵活？

18.1.5. Explicit Destructor Invocation

18.1.5. 显式析构函数的调用

Just as placement `new` is a lower-level alternative to using the `allocate` member of the `allocator` class, we can use an explicit call to a destructor as the lower-level alternative to calling `destroy`.

正如定位 `new` 表达式是使用 `allocator` 类的 `construct` 成员的低级选择，我们可以使用析构函数的显式调用作为调用 `destroy` 函数的低级选择。

In the version of `vector` that used an `allocator`, we clean up each element by calling `destroy`:

在使用 `allocator` 对象的 `vector` 版本中，通过调用 `destroy` 函数清除每个元素：

```
// destroy the old elements in reverse order
for (T *p = first_free; p != elements; /* empty */ )
    alloc.destroy(--p);
```

For programs that use a placement `new` expression to construct the object, we call the destructor explicitly:

对于使用定位 `new` 表达式构造对象的程序，显式调用析构函数：

```
for (T *p = first_free; p != elements; /* empty */ )
    p->~T(); // call the destructor
```

Here we invoke a destructor directly. The arrow operator dereferences the iterator `p` to obtain the object to which `p` points. We then call the destructor, which is the name of the type preceded by a tilde (~).

在这里直接调用析构函数。箭头操作符对迭代器 `p` 解引用以获得 `p` 所指的对象，然后，调用析构函数，析构函数以类名前加 ~ 来命名。

The effect of calling the destructor explicitly is that the object itself is properly cleaned up. However, the memory in which the object resided is not freed. We can reuse the space if desired.

显式调用析构函数的效果是适当地清除对象本身。但是，并没有释放对象所占的内存，如果需要，可以重用该内存空间。



Calling the `operator delete` function does not run the destructor; it only frees the indicated memory.

调用 `operator delete` 函数不会运行析构函数，它只释放指定的内存。

Exercises Section 18.1.5

Section 18.1. Optimizing Memory Allocation

Exercise 18.6: Reimplement your `Vector` class to use `operator new`, `operator delete`, placement `new`, and direct calls to the destructor.

重新实现 `Vector` 类，使用 `operator new`、`operator delete`、定位 `new` 表达式，并直接调用析构函数。

Exercise 18.7: Test your new version by running the same programs that you ran against your initial `Vector` implementation.

运行行为原来的 `Vector` 实现而运行的程序，测试你的新版本。

Exercise 18.8: Which version do you think is better, and why?

你认为哪个版本更好？为什么？

18.1.6. Class Specific `new` and `delete`

18.1.6. 类特定的 `new` 和 `delete`

The previous subsections looked at how a class can take over memory management for its own internal data structure. Another way to optimize memory allocation involves optimizing the behavior of `new` expressions. As an example, consider the `Queue` class from [Chapter 16](#). That class doesn't hold its elements directly. Instead, it uses `new` expressions to allocate objects of type `QueueItem`.

前几节介绍了类怎样能够接管自己的内部数据结构的内存管理，另一种优化内存分配的方法涉及优化 `new` 表达式的行为。作为例子，考虑[第十六章](#)的 `Queue` 类。该类不直接保存它的元素，相反，它使用 `new` 表达式分配 `QueueItem` 类型的对象。

It might be possible to improve the performance of `Queue` by preallocating a block of raw memory to hold `QueueItem` objects. When a new `QueueItem` object is created, it could be constructed in this preallocated space. When `QueueItem` objects are freed, we'd put them back in the block of preallocated objects rather than actually returning memory to the system.

通过预先分配一块原始内存以保存 `QueueItem` 对象，也许有可能改善 `Queue` 的性能。创建新 `QueueItem` 对象的时候，可以在这个预先分配的空间中构造对象。释放 `QueueItem` 对象的时候，将它们放回预先分配对象的块中，而不是将内存真正返回给系统。

The difference between this problem and our `Vector` implementation is that in this case, we want to optimize the behavior of `new` and `delete` expressions when applied to objects of a particular type. By default, `new` expressions allocate memory by calling the version of `operator new` that is defined by the library. A class may manage the memory used for objects of its type by defining its own members named `operator new` and `operator delete`.

这个问题与 `Vector` 的实现之间的区别在于，在这种情况下，我们希望在应用于特定类型的时候优化 `new` 和 `delete` 表达式的行为。默认情况下，`new` 表达式通过调用由标准库定义的 `operator new` 版本分配内存。通过定义自己的名为 `operator new` 和 `operator delete` 的成员，类可以管理用于自身类型的内存。

When the compiler sees a `new` or `delete` expression for a class type, it looks to see if the class has a member `operator new` or `operator delete`. If the class defines (or inherits) its own member `new` and `delete` functions, then those functions are used to allocate and free the memory for the object. Otherwise, the standard library versions of these functions are called.

编译器看到类类型的 `new` 或 `delete` 表达式的时候，它查看该类是否有 `operator new` 或 `operator delete` 成员，如果类定义（或继承）了自己的成员 `new` 和 `delete` 函数，则使用那些函数为对象分配和释放内存；否则，调用这些函数的标准库版本。

When we optimize the behavior of `new` and `delete`, we need only define new versions of the `operator new` and `operator delete`. The `new` and `delete` expressions themselves take care of constructing and destroying the objects.

优化 `new` 和 `delete` 的行为的时候，只需要定义 `operator new` 和 `operator delete` 的新版本，`new` 和 `delete` 表达式自己照管对象的构造和撤销。

Member `new` and `delete` Functions

成员 `new` 和 `delete` 函数

If a class defines either of these members, it should define both of them.

如果类定义了这两个成员中的一个，它也应该定义另一个。



Section 18.1. Optimizing Memory Allocation

A class member `operator new` function must have a return type of `void*` and take a parameter of type `size_t`. The function's `size_t` parameter is initialized by the `new` expression with the size, in bytes, of the amount of memory to allocate.

类成员 `operator new` 函数必须具有返回类型 `void*` 并接受 `size_t` 类型的形参。由 `new` 表达式用以字节计算的分配内存量初始化函数的 `size_t` 形参。

A class member `operator delete` function must have a `void` return type. It can be defined to take a single parameter of type `void*` or to take two parameters, a `void*` and a `size_t`. The `void*` parameter is initialized by the `delete` expression with the pointer that was `deleted`. That pointer might be a null pointer. If present, the `size_t` parameter is initialized automatically by the compiler with the size in bytes of the object addressed by the first parameter.

类成员 `operator delete` 函数必须具有返回类型 `void`。它可以定义为接受单个 `void*` 类型形参，也可以定义为接受两个形参，即 `void*` 和 `size_t` 类型。由 `delete` 表达式用被 `delete` 的指针初始化 `void*` 形参，该指针可以是空指针。如果提供了 `size_t` 形参，就由编译器用第一个形参所指对象的字节大小自动初始化 `size_t` 形参。

The `size_t` parameter is unnecessary unless the class is part of an inheritance hierarchy. When we `delete` a pointer to a type in an inheritance hierarchy, the pointer might point to a base-class object or an object of a derived class. In general, the size of a derived-type object is larger than the size of a base-class object. If the base class has a `virtual` destructor ([Section 15.4.4, p. 587](#)), then the size passed to `operator delete` will vary depending on the dynamic type of the object to which the deleted pointer points. If the base class does not have a virtual destructor, then, as usual, the behavior of deleting a pointer to a derived object through a base-class pointer is undefined.

除非类是某继承层次的一部分，否则形参 `size_t` 不是必需的。当 `delete` 指向继承层次中类型的指针时，指针可以指向基类对象，也可以指向派生类对象。派生类对象的大小一般比基类对象大。如果基类有 `virtual` 析构函数（[第 15.4.4 节](#)），则传给 `operator delete` 的大小将根据被删除指针所指对象的动态类型而变化；如果基类没有 `virtual` 析构函数，那么，通过基类指针删除指向派生类对象的指针的行为，跟往常一样是未定义的。

These functions are implicitly static members ([Section 12.6.1, p. 469](#)). There is no need to declare them `static` explicitly, although it is legal to do so. The member `new` and `delete` functions must be static because they are used either before the object is constructed (`operator new`) or after it has been destroyed (`operator delete`). There are, therefore, no member data for these functions to manipulate. As with any other static member function, `new` and `delete` may access only static members of their class directly.

这些函数隐式地为静态函数（[第 12.6.1 节](#)），不必显式地将它们声明为 `static`，虽然这样做是合法的。成员 `new` 和 `delete` 函数必须是静态的，因为它们要么在构造对象之前使用（`operator new`），要么在撤销对象之后使用（`operator delete`），因此，这些函数没有成员数据可操纵。像任意其他静态成员函数一样，`new` 和 `delete` 只能直接访问所属类的静态成员。

Array Operator `new[]` and Operator `delete[]`

数组操作符 `new[]` 和操作符 `delete[]`

We can also define member `operator new[]` and `operator delete[]` to manage arrays of the class type. If these `operator` functions exist, the compiler uses them in place of the global versions.

也可以定义成员 `operator new[]` 和 `operator delete[]` 来管理类类型的数组。如果这些 `operator` 函数存在，编译器就使用它们代替全局版本。

A class member `operator new[]` must have a return type of `void*` and take a first parameter of type `size_t`. The operator's `size_t` parameter is initialized automatically with a value that represents the number of bytes required to store an array of the given number of elements of the specified type.

类成员 `operator new[]` 必须具有返回类型 `void*`，并且接受的第一个形参类型为 `size_t`。用表示存储特定类型给定数目元素的数组的字节数值自动初始化操作符的 `size_t` 形参。

The member operator `delete[]` must have a `void` return type and a first parameter of type `void*`. The operator's `void*` parameter is initialized automatically with a value that represents the beginning of the storage in which the array is stored.

成员操作符 `operator delete[]` 必须具有返回类型 `void`，并且第一个形参为 `void*` 类型。用表示数组存储起始位置的值自动初始化操作符的 `void*` 形参。

The operator `delete[]` for a class may also have two parameters instead of one, the second parameter being a `size_t`. If present, the additional parameter is initialized automatically by the compiler with the size in bytes of the storage required to store the array.

类的操作符 `delete[]` 也可以有两个形参，第二个形参为 `size_t`。如果提供了附加形参，由编译器用数组所需存储量的字节数自动初始化这个形参。

Overriding Class-Specific Memory Allocation

覆盖类特定的内存分配

A user of a class that defines its own member `new` and `delete` can force a `new` or `delete` expression to use the global library functions through the use of the global scope resolution operator. If the user writes

如果类定义了自己的成员 `new` 和 `delete`，类的用户就可以通过使用全局作用域确定操作符，强制 `new` 或 `delete` 表达式使用全局的库函数。如果用户编写

```
Type *p = ::new Type; // uses global operator new
```

Section 18.1. Optimizing Memory Allocation

```
::delete p;           // uses global operator delete
```

then `new` invokes the global `operator new` even if class `Type` defines its own class-specific `operator new`; similarly for `delete`.

那么，即使类定义了自己的类特定的 `operator new`，也调用全局的 `operator new`; `delete` 类似。



If storage was allocated with a `new` expression invoking the global `operator new` function, then the `delete` expression should also invoke the global `operator delete` function.

如果用 `new` 表达式调用全局 `operator new` 函数分配内存，则 `delete` 表达式也应该调用全局 `operator delete` 函数。

Exercises Section 18.1.6

Exercise Declare members `new` and `delete` for the `QueueItem` class.

18.9:

为 `QueueItem` 类声明成员 `new` 和 `delete`。

18.1.7. A Memory-Allocator Base Class

18.1.7. 一个内存分配器基类



Like the generic handle class ([Section 16.5](#), p. 666) this example represents a fairly sophisticated use of C++. Understanding this example requires (and demonstrates) a good grasp of both inheritance and templates. It may be useful to delay studying this example until you are comfortable with these features.

像泛型句柄类一样（[第 16.5 节](#)），这个例子表示 C++ 的相当复杂的使用，理解这个例子需要（并认证）良好掌握地继承和模板，将对这个例子的研究推迟到你熟悉这些特征之后，也许是有用的。

Having seen how to declare class-specific member `new` and `delete`, we might next implement those members for `QueueItem`. Before doing so, we need to decide how we'll improve over the built-in library `new` and `delete` functions. One common strategy is to preallocate a block of raw memory to hold unconstructed objects. When new elements are created, they could be constructed in one of these preallocated objects. When elements are freed, we'd put them back in the block of preallocated objects rather than actually returning memory to the system. This kind of strategy is often known as maintaining a [freelist](#). The freelist might be implemented as a linked list of objects that have been allocated but not constructed.

已经看过了怎样声明类特定的成员 `new` 和 `delete`，下面就可以为 `QueueItem` 类实现这些成员，在这样做之前，需要决定怎样改进内置库的 `new` 和 `delete` 函数。一个通用策略是预先分配一块原始内存来保存未构造的对象，创建新元素的时候，可以在一个预先分配的对象中构造；释放元素的时候，将它们放回预先分配对象的块中，而不是将内存实际返还给系统。这种策略常被称为维持一个[自由列表](#)。可以将自由列表实现为已分配但未构造的对象的链表。

Rather than implementing a freelist-based allocation strategy for `QueueItem`, we'll observe that `QueueItem` is not unique in wanting to optimize allocation of objects of its type. Many classes might have the same need. Because this behavior might be generally useful, we'll define a new class that we'll name `CachedObj` to handle the freelist. A class, such as `QueueItem`, that wants to optimize allocation of objects of its type could use the `CachedObj` class rather than implementing its own `new` and `delete` members directly.

除了为 `QueueItem` 类实现基于自由列表的分配策略，我们注意到 `QueueItem` 不是唯一希望优化其对象分配的类，许多类都可能有同一需要。因为这个行为也许通常是有用的，所以我们将定义一个名为 `CachedObj` 的新类来处理自由列表。像 `QueueItem` 这样希望优化其对象分配的类可以使用 `CachedObj` 类，而不用直接实现自己的 `new` 和 `delete` 成员。

The `CachedObj` class will have a simple interface: Its only job is to allocate and manage a freelist of allocated but unconstructed objects. This class will define a member `operator new` that will return the next element from the freelist, removing it from the freelist. The `operator new` will allocate new raw memory whenever the freelist becomes empty. The class will also define `operator delete` to put an element back on the freelist when an object is destroyed.

`CachedObj` 类有简单的接口：它的工作只是分配和管理已分配但未构造对象的自由列表。这个类将定义一个成员 `operator new`，返回自由列表的下一个元素，并将该元素从自由列表中删除。当自由列表为空的时候，`operator new` 将分配新的原始内存。这个类还定义 `operator delete`，在撤销对象时将元素放回自由列表。

Classes that wish to use a freelist allocation strategy for their own types will *inherit* from `CachedObj`. Through inheritance, these classes can use the `CachedObj` definition of `operator new` and `operator delete` along with the data members needed to represent the freelist. Because the `CachedObj` class

Section 18.1. Optimizing Memory Allocation

is intended as a base class, we'll give it a `public` virtual destructor.

希望为自己的类型使用自由列表分配策略的类将继承 `CachedObj` 类，通过继承，这些类可以使用 `CachedObj` 类的 `operator new` 和 `operator delete` 定义，以及表示自由列表所需的数据成员。因为打算将 `CachedObj` 类作为基类，所以将给它一个 `public` 虚析构函数。



As we'll see, `CachedObj` may be used only for types that are not involved in an inheritance hierarchy. Unlike the member `new` and `delete` operations, `CachedObj` has no way to allocate different sized objects depending on the actual type of the object: Its freelist holds objects of a single size. Hence, it may be used only for classes, such as `QueueItem`, that do not serve as base classes.

正如我们将看到的，`CachedObj` 只能用于不包含在继承层次中类型。与成员 `new` 和 `delete` 操作不同，`CachedObj` 类没有办法根据对象的实际类型分配不同大小的对象：它的自由列表保存单一大小的对象。因此，它只能用于不作基类使用的类，如 `QueueItem` 类。

The data members defined by the `CachedObj` class, and inherited by its derived classes, are:

由 `CachedObj` 类定义并被它的派生类继承的数据成员是：

- A `static` pointer to the head of the freelist
指向自由列表表头的 `static` 指针。
- A member named `next` that points from one `CachedObj` to the next
名为 `next`、从一个 `CachedObj` 对象指向另一个 `CachedObj` 对象的指针。

The `next` pointer chains the elements together onto the freelist. Each type that we derive from `CachedObj` will contain its own type-specific data plus a single pointer inherited from the `CachedObj` base class. Each object has an extra pointer used by the memory allocator but not by the inherited type itself. When the object is in use, this pointer is meaningless and not used. When the object is available for use and is on the freelist, then the `next` pointer is used to point to the next available object.

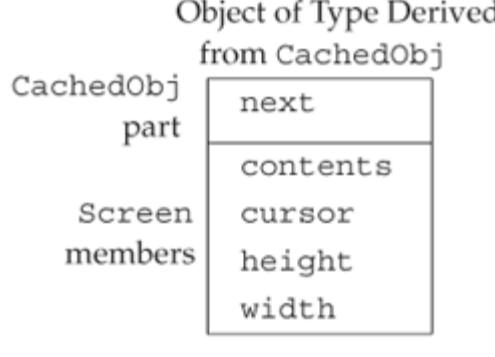
`next` 指针将元素链入自由列表。从 `CachedObj` 类派生的每个类型都包含自己的类型特定的数据，加上一个从 `CachedObj` 基类继承的指针。每个对象具有由内存分配器使用但被继承类型自己不用的一个额外指针，对象在使用的时候，该指针无意义且不使用；对象可供使用并在自由列表中的时候，就使用 `next` 指针来指向下一个可用的对象。

If we used `CachedObj` to optimize allocation of our `Screen` class, objects of type `Screen` (conceptually) would look like the illustration in [Figure 18.2](#).

如果使用 `CachedObj` 类来优化 `Screen` 类的分配，`Screen` 类型的对象（概念上）看起来如图 18.2 所示。

Figure 18.2. Illustration of a `CachedObj` Derived Class

图 18.2. `CachedObj` 派生类举例



The `CachedObj` Class

`CachedObj` 类

The only remaining question is what types to use for the pointers in `CachedObj`. We'd like to use the freelist approach for any type, so the class will be a template. The pointers will point to an object of the template type:

剩下的唯一问题是 `CachedObj` 类中的指针使用什么类型。我们希望为任意类型使用自由列表方法，所以 `CachedObj` 类将是一个模板，指针将指向该模板类型的对象：

Section 18.1. Optimizing Memory Allocation

```
/* memory allocation class: Pre-allocates objects and
 * maintains a freelist of objects that are unused
 * When an object is freed, it is put back on the freelist
 * The memory is only returned when the program exits
 */
template <class T> class CachedObj {
public:
    void *operator new(std::size_t);
    void operator delete(void *, std::size_t);
    virtual ~CachedObj() { }
protected:
    T *next;
private:
    static void add_to_freelist(T*);
    static std::allocator<T> alloc_mem;
    static T *freeStore;
    static const std::size_t chunk;
};
```

The class is quite simple. It provides only three public members: `operator new`, `operator delete`, and a virtual destructor. The `new` and `delete` members take objects off and return objects to the freelist.

这个类相当简单，它只提供三个公用成员：`operator new`、`operator delete` 和虚析构函数。`new` 和 `delete` 成员分别从自由列表取走对象和将对象返回到自由列表。

The `static` members manage the freelist. These members are declared as `static` because there is only one freelist maintained for all the objects of a given type. The `freeStore` pointer points to the head of the freelist. The member named `chunk` specifies the number of objects that will be allocated each time the freelist is empty. Finally, `add_to_freelist` puts objects on the freelist. This function is used by `operator new` to put newly allocated objects onto the freelist. It is also used by `operator delete` to put an object back on the free list when an object is deleted.

`static` 成员管理自由列表。这些成员声明为 `static`，是因为只为所有给定类型的对象维持一个自由列表。`freeStore` 指针指向自由列表的表头。名为 `chunk` 的成员指定每当自由列表为空时将分配的对象的数目。最后，`add_to_freelist` 函数将对象放在自由列表，`operator new` 使用这个函数将新分配的对象放到自由列表，删除对象的时候，`operator delete` 也使用该函数将对象放回自由列表。

Using `CachedObj`

使用 `CachedObj`

The only really tricky part in using `CachedObj` is understanding the template parameter: When we inherit from `CachedObj`, the template type we use to instantiate `CachedObj` will be the derived type itself. We inherit from `CachedObj` in order to reuse its freelist management. However, `CachedObj` holds a pointer to the object type it manages. The type of that pointer is pointer to a type derived from `CachedObj`.

使用 `CachedObj` 类，真正复杂的部分是理解模板形参：当继承 `CachedObj` 类的时候，用来实例化 `CachedObj` 类的模板类型将是派生类型本身。为了重用 `CachedObj` 类的自由列表管理而继承 `CachedObj` 类，但是，`CachedObj` 类保存了指向它管理的对象类型的一个指针，该指针的类型是指向 `CachedObj` 的派生类型的指针。

For example, to optimize memory management for our `Screen` class we would declare `Screen` as

例如，为了优化 `Screen` 类的内存管理，我们将 `Screen` 声明为：

```
class Screen: public CachedObj<Screen> {
    // interface and implementation members of class Screen are unchanged
};
```

This declaration gives `Screen` a new base class, the instance of `CachedObj` that is parameterized by type `Screen`. Each `Screen` now includes an additional inherited member named `next` in addition to its other members defined inside the `Screen` class.

这个声明给了 `Screen` 类一个新的基类：形参为 `Screen` 类型的 `CachedObj` 实例。现在除了 `Screen` 类内部定义的其他成员之外，每个 `Screen` 对象还包含附加的名为 `next` 的继承成员。

Because `QueueItem` is a template type, deriving it from `CachedObj` is a bit complicated:

因为 `QueueItem` 是一个模板类型，从 `CachedObj` 类派生它有点复杂：

```
template <class Type>
class QueueItem: public CachedObj< QueueItem<Type> > {
    // remainder of class declaration and all member definitions unchanged
};
```

This declaration says that `QueueItem` is a class template that is derived from the instantiation of `CachedObj` that holds objects of type `QueueItem<Type>`. For example, if we define a `Queue` of `ints`, then the `QueueItem<int>` class is derived from `CachedObj< QueueItem<int> >`.

这个声明是说，`QueueItem` 是从保存 `QueueItem<Type>` 类型对象的 `CachedObj` 实例派生而来的类模板。例如，如果定义 `int` 值的 `Queue`，就从 `CachedObj< QueueItem<int> >` 派生 `QueueItem<int>` 类。



No other changes are needed in our class. `QueueItem` now has automatic memory allocation that uses a freelist to reduce the number of allocations required when creating new `Queue` elements.

我们的类不需要其他改变。现在 `QueueItem` 类具有自动内存分配，这个内存分配使用自由列表减少创建新的 `Queue` 元素时需要的分配数目。

How Allocation Works

分配怎样工作

Because we derived `QueueItem` from `CachedObj`, any allocation using a `new` expression, such as the call from `Queue::push`:

因为我们从 `CachedObj` 类派生 `QueueItem` 类，任何使用 `new` 表达式的分配，如 `Queue::push` 中的调用：

```
// allocate a new QueueItem object
QueueItem<Type> *pt =
    new QueueItem<Type>(val);
```

allocates and constructs a new `QueueItem`. Each `new` expression:

分配并构造一个新的 `QueueItem` 对象。每个表达式

1. Uses the `QueueItem<T>::operator new` function to allocate an object from the freelist

使用 `QueueItem<T>::operator new` 函数从自由列表分配一个对象。

2. Uses the element type copy constructor for type `T` to construct an object in that storage

为类型 `T` 使用元素类型的复制构造函数，在该内存中构造一个对象。

Similarly, when we `delete` a `QueueItem` pointer such as

类似地，当像 `delete pt;` 这样删除一个 `QueueItem` 指针的时候，运行 `QueueItem` 析构函数

```
delete pt;
```

the `QueueItem` destructor is run to clean up the object to which `pt` points and the class `operator delete` is called. That operator puts the memory the element used back on the freelist.

指向的对象，并调用该类的 `operator delete`，将元素所用的内存放回自由列表。

Defining `operator new`

定义 `operator new`

The `operator new` member returns an object from the freelist. If the freelist is empty, `new` must first allocate a new `chunk` of memory:

`operator new` 成员从自由列表返回一个对象，如果自由列表为空，`new` 必须首先分配 `chunk` 数目的新内存：

```
template <class T>
void *CachedObj<T>::operator new(size_t sz)
{
    // new should only be asked to build a T, not an object
    // derived from T; check that right size is requested
    if (sz != sizeof(T))
        throw std::runtime_error
            ("CachedObj: wrong size object in operator new");
    if (!freeStore) {
        // the list is empty: grab a new chunk of memory
        // allocate allocates chunk number of objects of type T
        T *array = alloc_mem.allocate(chunk);

        // now set the next pointers in each object in the allocated memory
        for (size_t i = 0; i != chunk; ++i)
            add_to_freelist(&array[i]);
    }
    T *p = freeStore;
    freeStore = freeStore->CachedObj<T>::next;
    return p;   // constructor of T will construct the T part of the object
}
```

The function begins by verifying that it is being asked to allocate the right amount of space.

函数首先验证要求它分配正确数量的空间。

This check enforces our design intent that `CachedObj` should be used only for classes that are not base classes. The fact that `CachedObj` allocates objects on its freelist that have a fixed size means that it cannot be used to handle memory allocation for classes in an inheritance hierarchy. Classes related by inheritance almost always define objects of different sizes. A single allocator would have to be much more sophisticated than the one we implement here to handle such classes.

Section 18.1. Optimizing Memory Allocation

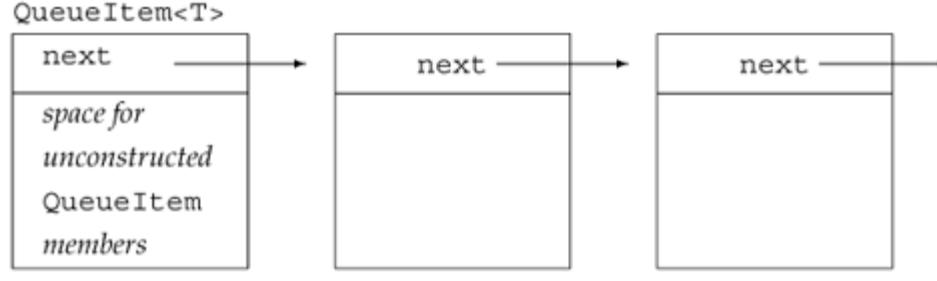
这个检查强调了我们的设计意图：`CachedObj` 类应该只被不是基类的类使用。`CachedObj` 类在固定大小的自由列表上分配对象，这一事实意味着，继承层次中的类不能使用它来处理内存分配。因继承而相关的类几乎总是定义不同大小的对象，处理这些类的单个分配器，可能必须比这里所实现的这个复杂得多。

The `operator new` function next checks whether there are any objects on the freelist. If not, it asks the `allocator` member to allocate `chunk` new, unconstructed objects. It then iterates through the newly allocated objects, setting the `next` pointer. After the call to `add_to_freelist`, each object on the freelist will be unconstructed, except for its `next` pointer, which will hold the address of the next available object. The freelist looks something like the picture in [Figure 18.3](#).

`operator new` 函数接着检查自由列表中是否有对象，如果没有，它就请求 `allocator` 成员分配 `chunk` 个新的未构造对象，然后，它通过新分配的对象进行迭代，设置 `next` 指针。调用了 `add_to_freelist` 函数之后，自由列表上的每个对象除了将保存下一个可用对象的地址 `next` 指针之外，将是未构造的。自由列表如图 18.3 所示。

Figure 18.3. Illustration `CachedObj` Freelist

图 18.3. `CachedObj` 自由列表示例



Having ensured that there are available objects to allocate, `operator new` returns the address of the first object on the freelist, resetting the `freeStore` pointer to address the next element on the freelist. The object returned is unconstructed. Because `operator new` is called from a `new` expression, the `new` expression will take care of constructing the object.

在有可用对象可以分配的保证之下，`operator new` 返回自由列表上第一个对象的地址，将 `freeStore` 指针重置为指向自由列表上下一个元素。被返回的对象是未构造的。因为从 `new` 表达式调用 `operator new`，所以 `new` 表达式将负责构造对象。

Defining `operator delete`

定义 `operator delete`

The member `operator delete` is responsible only for managing the memory. The object itself was already cleaned up in the destructor, which the `delete` expression calls before calling `operator delete`. The `operator delete` member is trivial:

`operator delete` 成员只负责管理内存，在析构函数中已经清除了对象本身，`delete` 表达式在调用 `operator delete` 之前调用析构函数。`operator delete` 成员很简单：

```
template <class T>
void CachedObj<T>::operator delete(void *p, size_t)
{
    if (p != 0)
        // put the "deleted" object back at head of freelist
        add_to_freelist(static_cast<T*>(p));
}
```

It calls `add_to_freelist` to put the deleted object back onto the freelist.

它调用 `add_to_freelist` 成员将被删除对象放回自由列表。

The interesting part is the cast ([Section 5.12.4](#), p. 183). `operator delete` is called when a dynamically allocated object of the class type is `deleted`. The compiler passes the address of that object to `operator delete`. However, the parameter type for the pointer must be `void*`. Before calling `add_to_freelist`, we have to cast the pointer from `void*` back to its actual type. In this case, that type is pointer to `T`, which in turn is a pointer to an object of a type derived from `CachedObj`.

令人感兴趣的部分是强制类型转换（[第 5.12.4 节](#)）。在删除动态分配的类类型对象时调用 `operator delete`，编译器将该对象的地址传给 `operator delete`。但是，指针的形参类型必须是 `void*`，在调用 `add_to_freelist` 之前，必须将指针从 `void*` 强制转换为它的实际类型，本例中，那个类型是指向 `T` 的指针，它是 `CachedObj` 派生类型的对象的指针。

The `add_to_freelist` Member

`add_to_freelist` 成员

The job of this member is to set the `next` pointer and update the `freeStore` pointer when an object is added to the freelist:

Section 18.1. Optimizing Memory Allocation

这个成员的任务是设置 `next` 指针，并且在将对象加到自由列表时更新 `freeStore` 指针：

```
// puts object at head of the freelist
template <class T>
void CachedObj<T>::add_to_freelist(T *p)
{
    p->CachedObj<T>::next = freeStore;
    freeStore = p;
}
```

The only tricky part is the use of the `next` member. Recall that `CachedObj` is intended to be used as a base class. The objects that are allocated aren't of type `CachedObj`. Instead, those objects are of a type derived from `CachedObj`. The type of `T`, therefore, will be the derived type. The pointer `p` is a pointer to `T`, not a pointer to `CachedObj`. If the derived class has its own member named `next`, then writing

唯一复杂的部分是 `next` 成员的使用。回忆一下，我们打算将 `CachedObj` 作为基类使用，被分配对象不是 `CachedObj` 类型的，相反，那些对象是 `CachedObj` 的派生类型的，因此，类型 `T` 将是派生类型，指针 `p` 是指向 `T` 的指针，不是指向 `CachedObj` 的指针。如果派生类型有自己的名为 `next` 的成员，则编写

```
p->next
```

would fetch the `next` member of the derived class! But we want to set the `next` in the base, `CachedObj` class.

将获得派生类的成员！但我们希望在基类——`CachedObj` 类中设置 `next`。



To avoid any possible collision with a member defined in the derived class, we explicitly specify that we are assigning to the base class member `next`.

为了避免任何与派生类中定义的成员可能的冲突，显式指定我们正在给基类成员 `next` 赋值。

Defining the Static Data Members

定义静态数据成员

What remains is to define the static data members:

剩下的是定义静态数据成员：

```
template <class T> allocator< T > CachedObj< T >::alloc_mem;
template <class T> T *CachedObj< T >::freeStore = 0;
template <class T> const size_t CachedObj< T >::chunk = 24;
```

As usual, with static members of a class template there is a different static member for each type used to instantiate `CachedObj`. We initialize `chunk` to an arbitrary value in this case, 24. We initialize the `freeStore` pointer to 0, indicating that the freelist starts out empty. There is no initialization required for the `alloc_mem` member, but we do have to remember to define it.

照常，对于类模板的静态成员，每个类型使用不同的静态成员来实例化 `CachedObj` 类。将 `chunk` 初始化为任意值，本例中为 24。将 `freeStore` 指针初始化为 0，指出自由列表开始时为空。`alloc_mem` 成员不需要初始化，但必须记得定义它。

Exercises Section 18.1.7

Exercise

18.10: Explain each of the following initializations. Indicate if any are errors, and if so, why.

解释下面每个初始化，指出是否有错误的，如果有，为什么错。

```
class iStack {
public:
    iStack(int capacity): stack(capacity), top(0) { }
private:
    int top;
    vector<int> stack;
};
(a) iStack *ps = new iStack(20);
(b) iStack *ps2 = new const iStack(15);
(c) iStack *ps3 = new iStack[ 100 ];
```

Exercise Explain what happens in the following `new` and `delete` expressions:

18.11: 解释下面 `new` 和 `delete` 表达式中发生什么。

```
struct Exercise {
    Exercise();
    ~Exercise();
};
Exercise *pe = new Exercise[20];
delete[] pe;
```

Exercise Implement a class-specific memory allocator for `Queue` or another class of your choice. Measure

Section 18.1. Optimizing Memory Allocation

18.12: the change in performance to see how much it helps, if at all.

为 `Queue` 类或你选择的其他类实现一个类特定的内存分配器。测量性能的改变，看看到底有多大帮助。

18.2. Run-Time Type Identification

18.2. 运行时类型识别

Run-time Type Identification (RTTI) allows programs that use pointers or references to base classes to retrieve the actual derived types of the objects to which these pointers or references refer.

通过运行时类型识别 (RTTI)，程序能够使用基类的指针或引用来检索这些指针或引用所指对象的实际派生类型。

RTTI is provided through two operators:

通过下面两个操作符提供 RTTI：

1. The `typeid` operator, which returns the actual type of the object referred to by a pointer or a reference
`typeid` 操作符，返回指针或引用所指对象的实际类型。
2. The `dynamic_cast` operator, which safely converts from a pointer or reference to a base type to a pointer or reference to a derived type
`dynamic_cast` 操作符，将基类类型的指针或引用安全地转换为派生类型的指针或引用。



These operators return dynamic type information only for classes with one or more virtual functions. For all other types, information for the static (i.e., compile-time) type is returned.

这些操作符只为带有一个或多个虚函数的类返回动态类型信息，对于其他类型，返回静态（即编译时）类型的信息。

The RTTI operators execute at run time for classes with virtual functions, but are evaluated at compile time for all other types.

对于带虚函数的类，在运行时执行 RTTI 操作符，但对于其他类型，在编译时计算 RTTI 操作符。

Dynamic casts are needed when we have a reference or pointer to a base class but need to perform operations from the derived class that are not part of the base class. Ordinarily, the best way to get derived behavior from a pointer to base is to do so through a virtual function. When we use virtual functions, the compiler automatically selects the right function according to the actual type of the object.

当具有基类的引用或指针，但需要执行不是基类组成部分的派生类操作的时候，需要动态的强制类型转换。通常，从基类指针获得派生类行为最好的方法是通过虚函数。当使用虚函数的时候，编译器自动根据对象的实际类型选择正确的函数。

In some situations however, the use of virtual functions is not possible. In these cases, RTTI offers an alternate mechanism. However, this mechanism is more error-prone than using virtual member functions: The programmer must *know* to which type the object should be cast and must check that the cast was performed successfully.

但是，在某些情况下，不可能使用虚函数。在这些情况下，RTTI 提供了可选的机制。然而，这种机制比使用虚函数更容易出错：程序员必须知道应该将对象强制转换为哪种类型，并且必须检查转换是否成功执行了。



Dynamic casts should be used with caution. Whenever possible, it is much better to define and use a virtual function rather than to take over managing the types directly.

使用动态强制类型转换要小心。只要可能，定义和使用虚函数比直接接管类型管理好得多。

18.2.1. The `dynamic_cast` Operator

18.2.1. `dynamic_cast` 操作符

The **`dynamic_cast` operator** can be used to convert a reference or pointer to an object of base type to a reference or pointer to another type in the same hierarchy. The pointer used with a `dynamic_cast` must be valid it must either be 0 or point to an object.

可以使用 **`dynamic_cast` 操作符** 将基类类型对象的引用或指针转换为同一继承层次中其他类型的引用或指针。与 `dynamic_cast` 一起使用的指针必须是有效的——它必须为 0 或者指向一个对象。

Unlike other casts, a `dynamic_cast` involves a run-time type check. If the object bound to the reference or pointer is not an object of the target type, then the `dynamic_cast` fails. If a `dynamic_cast` to a pointer type fails, the result of the `dynamic_cast` is the value 0. If a `dynamic_cast` to a reference type fails, then an exception of type `bad_cast` is thrown.

与其他强制类型转换不同，`dynamic_cast` 涉及运行时类型检查。如果绑定到引用或指针的对象不是目标类型的对象，则 `dynamic_cast` 失败。如果转换到指针类型的 `dynamic_cast` 失败，则 `dynamic_cast` 的结果是 0 值；如果转换到引用类型的 `dynamic_cast` 失败，则抛出一个 `bad_cast` 类型的异常。

The `dynamic_cast` operator therefore performs two operations at once. It begins by verifying that the requested cast is valid. Only if the cast is valid does the operator actually do the cast. In general, the type of the object to which the reference or pointer is bound isn't known at compile-time. A pointer to base can be assigned to point to a derived object. Similarly, a reference to base can be initialized by a derived object. As a result, the verification that the `dynamic_cast` operator performs must be done at run time.

因此，`dynamic_cast` 操作符一次执行两个操作。它首先验证被请求的转换是否有效，只有转换有效，操作符才实际进行转换。一般而言，引用或指针所绑定的对象的类型在编译时是未知的，基类的指针可以赋值为指向派生类对象，同样，基类的引用也可以用派生类对象初始化，因此，`dynamic_cast` 操作符执行的验证必须在运行时进行。

Using the `dynamic_cast` Operator

使用 `dynamic_cast` 操作符

As a simple example, assume that `Base` is a class with at least one virtual function and that class `Derived` is derived from `Base`. If we have a pointer to `Base` named `basePtr`, we can cast it at run time to a pointer to `Derived` as follows:

作为例子，假定 `Base` 是至少带一个虚函数的类，并且 `Derived` 类派生于 `Base` 类。如果有一个名为 `basePtr` 的指向 `Base` 的指针，就可以像这样在运行时将它强制转换为指向 `Derived` 的指针：

```
if (Derived *derivedPtr = dynamic_cast<Derived*>(basePtr))
{
    // use the Derived object to which derivedPtr points
} else { // BasePtr points at a Base object
    // use the Base object to which basePtr points
}
```

At run time, if `basePtr` actually points to a `Derived` object, then the cast will be successful, and `derivedPtr` will be initialized to point to the `Derived` object to which `basePtr` points. Otherwise, the result of the cast is 0, meaning that `derivedPtr` is set to 0, and the condition in the `if` fails.

在运行时，如果 `basePtr` 实际指向 `Derived` 对象，则转换将成功，并且 `derivedPtr` 将被初始化为指向 `basePtr` 所指的 `Derived` 对象；否则，转换的结果是 0，意味着将 `derivedPtr` 置为 0，并且 `if` 中的条件失败。



We can apply a `dynamic_cast` to a pointer whose value is 0. The result of doing so is 0.

可以对值为 0 的指针应用 `dynamic_cast`，这样做的结果是 0。

By checking the value of `derivedPtr`, the code inside the `if` knows that it is operating on a `Derived` object. It is safe for that code to use `Derived` operations. If the `dynamic_cast` fails because `basePtr` refers to a `Base` object, then the `else` clause does processing appropriate to `Base` instead. The other advantage of doing the check inside the `if` condition is that it is not possible to insert code between the `dynamic_cast` and testing the result of the cast. It is, therefore, not possible to use the `derivedPtr` inadvertently before testing that the cast was successful. A third advantage is that the pointer is not accessible outside the `if`. If the cast fails, then the unbound pointer is not available for use in later cases where the test might be forgotten.

通过检查 `derivedPtr` 的值，`if` 内部的代码知道它是在操作 `Derived` 对象，该代码使用 `Derived` 的操作是安全的。如果 `dynamic_cast` 因 `basePtr` 引用了 `Base` 对象而失败，则 `else` 子句进行适应于 `Base` 的处理来代替。在 `if` 条件内部进行检查的另一个好处是，不可能在 `dynamic_cast` 和测试转换结果之间插入代码，因此，不可能在测试转换是否成功之前不经意地使用 `derivedPtr`。第三个好处是，在 `if` 外部不能访问该指针，如果转换失败，则在后面的忘了测试的地方，未绑定的指针是不可用的。

Performing a `dynamic_cast` in a condition ensures that the cast and test of its result are done in a single expression.

在条件中执行 `dynamic_cast` 保证了转换和其结果测试在一个表达式中进行。



Using a `dynamic_cast` and Reference Types

使用 `dynamic_cast` 和引用类型

In the previous example, we used a `dynamic_cast` to convert a pointer to base to a pointer to derived. A `dynamic_cast` can also be used to convert a reference to base to a reference to derived. The form for this a `dynamic_cast` operation is the following,

在前面例子中，使用了 `dynamic_cast` 将基类指针转换为派生类指针，也可以使用 `dynamic_cast` 将基类引用转换为派生类引用，这种 `dynamic_cast` 操作的形式如下：

```
dynamic_cast< Type& >(val)
```

where `Type` is the target type of the conversion, and `val` is an object of base class type.

这里，`Type` 是转换的目标类型，而 `val` 是基类类型的对象。

The `dynamic_cast` operation converts the operand `val` to the desired type `Type&` only if `val` actually refers to an object of the type `Type` or is an object of a type derived from `Type`.

只有当 `val` 实际引用一个 `Type` 类型对象，或者 `val` 是一个 `Type` 派生类型的对象的时候，`dynamic_cast` 操作才将操作数 `val` 转换为想要的 `Type&` 类型。

Because there is no such thing as a null reference, it is not possible to use the same checking strategy for references that is used for pointer casts. Instead, when a cast fails, it throws a `std::bad_cast` exception. This exception is defined in the `typeinfo` library header.

因为不存在空引用，所以不可能对引用使用用于指针强制类型转换的检查策略，相反，当转换失败的时候，它抛出一个 `std::bad_cast` 异常，该异常在库头文件 `typeinfo` 中定义。

We might rewrite the previous example to use references as follows:

可以重写前面的例子如下，以便使用引用：

```
void f(const Base &b)
{
    try {
        const Derived &d = dynamic_cast<const Derived&>(b);
        // use the Derived object to which b referred
    } catch (bad_cast) {
        // handle the fact that the cast failed
    }
}
```

Exercises Section 18.2.1

Exercise

18.13:

Given the following class hierarchy in which each class defines a `public` default constructor and virtual destructor,

给定下面的类层次，其中每个类都定义了 `public` 默认构造函数和虚析构函数。

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public B { /* ... */ };
class D : public B, public A { /* ... */ };
```

which, if any, of the following `dynamic_casts` fail?

如果有，下面哪些 `dynamic_casts` 失败？

(a) `A *pa = new C;`

Section 18.2. Run-Time Type Identification

```
B *pb = dynamic_cast< B* >(pa);  
(b) B *pb = new B;  
C *pc = dynamic_cast< C* >(pb);  
(c) A *pa = new D;  
B *pb = dynamic_cast< B* >(pa);
```

Exercise 18.14: What would happen in the last conversion in the previous exercise if both **D** and **B** inherited from **A** as a virtual base class?

如果 **D** 和 **B** 都以 **A** 为虚基类，上题最后一个转换中将发生什么？

Exercise 18.15: Using the class hierarchy defined in the previous exercise, rewrite the following piece of code to perform a reference `dynamic_cast` to convert the expression `*pa` to the type `C&`:

使用上面习题中定义的类层次，重写下面代码片段，以便执行 `dynamic_cast` 将表达式 `*pa` 转换为 `C&` 类型：

```
if (C *pc = dynamic_cast< C* >(pa))  
    // use C's members  
} else {  
    // use A's members  
}
```

Exercise 18.16: Explain when you would use `dynamic_cast` instead of a virtual function.

解释什么时候可以使用 `dynamic_cast` 代替虚函数。

18.2.2. The `typeid` Operator

18.2.2. `typeid` 操作符

The second operator provided for RTTI is the `typeid` operator. The `typeid` operator allows a program to ask of an expression: What type are you?

为 RTTI 提供的第二个操作符是 `typeid` 操作符。`typeid` 操作符使程序能够问一个表达式：你是什么类型？

A `typeid` expression has the form

`typeid` 表达式形如：

```
typeid(e)
```

where `e` is any expression or a type name.

这里 `e` 是任意表达式或者是类型名。

If the type of the expression is a class type and that class contains one or more virtual functions, then the dynamic type of the expression may differ from its static compile-time type. For example, if the expression dereferences a pointer to a base class, then the static compile-time type of that expression is the base type. However, if the pointer actually addresses a derived object, then the `typeid` operator will say that the type of the expression is the derived type.

如果表达式的类型是类类型且该类包含一个或多个虚函数，则表达式的动态类型可能不同于它的静态编译时类型。例如，如果表达式对基类指针解引用，则该表达式的静态编译时类型是基类类型；但是，如果指针实际指向派生类对象，则 `typeid` 操作符将说表达式的类型是派生类型。

The `typeid` operator can be used with expressions of any type. Expressions of built-in type as well as constants can be used as operands for the `typeid` operator. When the operand is not of class type or is a class without virtual functions, then the `typeid` operator indicates the static type of the operand. When the operand has a class-type that defines at least one virtual function, then the type is evaluated at run time.

`typeid` 操作符可以与任何类型的表达式一起使用。内置类型的表达式以及常量都可以用作 `typeid` 操作符的操作数。如果操作数不是类类型或者是没有虚函数的类，则 `typeid` 操作符指出操作数的静态类型；如果操作数是定义了至少一个虚函数的类类型，则在运行时计算类型。

The result of a `typeid` operation is a reference to an object of a library type named `type_info`. [Section 18.2.4](#) (p. 779) covers this type in more detail. To use the `type_info` class, the library header `typeinfo` must be included.

`typeid` 操作符的结果是名为 `type_info` 的标准库类型的对象引用，[第 18.2.4 节](#) 将更详细地讨论这个类型。要使用 `type_info` 类，必须包含库头文件 `typeinfo`。

Using the `typeid` Operator

使用 `typeid` 操作符

The most common use of `typeid` is to compare the types of two expressions or to compare the type of an expression to a specified type:

`typeid` 最常见的用途是比较两个表达式的类型，或者将表达式的类型与特定类型相比较：

```
Base *bp;
Derived *dp;
// compare type at run time of two objects
if (typeid(*bp) == typeid(*dp)) {
    // bp and dp point to objects of the same type
}
// test whether run time type is a specific type
if (typeid(*bp) == typeid(Derived)) {
    // bp actually points to a Derived
}
```

In the first `if`, we compare the actual types of the objects to which `bp` and `dp` point. If they both point to the same type, then the test succeeds. Similarly, the second `if` succeeds if `bp` currently points to a `Derived` object.

第一个 `if` 中，比较 `bp` 所指对象与 `dp` 所指对象的实际类型，如果它们指向同一类型，则测试成功。类似地，如果 `bp` 当前指向 `Derived` 对象，则第二个 `if` 成功。

Note that the operands to the `typeid` are expressions that are objectswe tested `*bp`, not `bp`:

注意，`typeid` 的操作数是表示对象的表达式——测试 `*bp`，而不是 `bp`:

```
// test always fails: The type of bp is pointer to Base
if (typeid(bp) == typeid(Derived)) {
    // code never executed
}
```

This test compares the type `Base*` to type `Derived`. These types are unequal, so this test will always fail regardless of the type of the object to which `bp` points.

这个测试将 `Base*` 类型与 `Derived` 类型相比较，这两个类型不相等，所以，无论 `bp` 所指对象的类型是什么，这个测试将问题失败。



Dynamic type information is returned only if the operand to `typeid` is an object of a class type with virtual functions. Testing a pointer (as opposed to the object to which the pointer points) returns the static, compile-time type of the pointer.

只有当 `typeid` 的操作数是带虚函数的类类型的对象的时候，才返回动态类型信息。测试指针（相对于指针指向的对象）返回指针的静态的、编译时类型。

If the value of a pointer `p` is 0, then `typeid(*p)` throws a `bad_typeid` exception if the type of `p` is a type with virtual functions. If the type of `p` does not define any virtuals, then the value of `p` is irrelevant. As when evaluating a `sizeof` expression (Section 5.8, p. 167) the compiler does not evaluate `*p`. It uses the static type of `p`, which does not require that `p` itself be a valid pointer.

如果指针 `p` 的值是 0，那么，如果 `p` 的类型是带虚函数的类型，则 `typeid(*p)` 抛出一个 `bad_typeid` 异常；如果 `p` 的类型没有定义任何虚函数，则结果与 `p` 的值是不相关的。正像计算表达式 `sizeof` (第 5.8 节) 一样，编译器不计算 `*p`，它使用 `p` 的静态类型，这并不要求 `p` 本身是有效指针。

Exercises Section 18.2.2

Exercise

18.17:

Write an expression to dynamically cast a pointer to a `Query_base` to a pointer to an `AndQuery`. Test the cast by using objects of `AndQuery` and of another query type. Print a statement indicating whether the cast works and be sure that the output matches your expectations.

编写一个表达式，动态地将 `Query_base` 对象的指针强制转换为 `AndQuery` 对象的指针。通过使用 `AndQuery` 和

Section 18.2. Run-Time Type Identification

其他查询类型的对象测试该转换。显示一个语句指出强制转换是否工作，并确信输出与你的表达式匹配。

Exercise 18.18: Write the same cast, but cast a `Query_base` object to a reference to `AndQuery`. Repeat the test to ensure that your cast works correctly.

编写相同的强制转换，但将 `Query_base` 对象转换为 `AndQuery` 的引用。重复测试以确信你的转换正确工作。

Exercise 18.19: Write a `typeid` expression to see whether two `Query_base` pointers point to the same type. Now check whether that type is an `AndQuery`.

编写一个 `typeid` 表达式，看两个 `Query_base` 指针是否指向相同的类型，然后检查该类型是否为 `AndQuery`。

18.2.3. Using RTTI

18.2.3. RTTI 的使用

As an example of when RTTI might be useful, consider a class hierarchy for which we'd like to implement the equality operator. Two objects are equal if they have the same value for a given set of their data members. Each derived type may add its own data, which we will want to include when testing for equality.

作为说明何时可以使用 RTTI 的例子，考虑一个类层次，我们希望为它实现相等操作符。如果两个对象的给定数据成员集合的值相同，它们就相等。每个派生类型可以增加自己的数据，我们希望在测试相等的时候包含这些数据。

Because the values considered in determining equality for a derived type might differ from those considered for the base type, we'll (potentially) need a different equality operator for each pair of types in the hierarchy. Moreover, we'd like to be able to use a given type as either the left-hand or right-hand operand, so we'll actually need two operators for each pair of types.

因为确定派生类型的相等与确定基类类型的相等所考虑的值不同，所以对层次中的每一对类型（潜在地）需要一个不同的相等操作符。而且，希望能够使用给类型作为左操作数或右操作数，所以实际上对每一对类型将需要两个操作符。

If our hierarchy has only two types, we need four functions:

如果类层次中只有两个类型，就需要四个函数：

```
bool operator==(const Base&, const Base&)
bool operator==(const Derived&, const Derived&)
bool operator==(const Derived&, const Base&);
bool operator==(const Base&, const Derived&);
```

But if our hierarchy has several types, the number of operators we must define expands rapidly for only 3 types we'd need 9 operators. If the hierarchy has 4 types, we'd need 16, and so on.

但是，如果类层次中有几个类型，必须定义的操作符的数目就迅速扩大——仅仅 3 个类型就需要 9 个操作符。如果类层次有 4 个类型，将需要 16 个操作符，以此类推。

We might think we could solve this problem by defining a set of virtual functions that would perform the equality test at each level in the hierarchy. Given those virtuals, we could define a single equality operator that operates on references to the base type. That operator could delegate its work to a virtual `equal` operation that would do the real work.

也许我们认为可以通过定义一个虚函数集合来解决这个问题，这些虚函数可以在类层次中每一层执行相等测试。给定这些虚函数，可以定义单个相等操作符，操作基类类型的引用，该操作符可以将工作委派给可以完成实际工作的虚操作。

Unfortunately, virtual functions are not a good match to this problem. The trouble is deciding on the type for the parameter to the `equal` operation. Virtual functions must have the same parameter type(s) in both the base and derived classes. That implies that a virtual `equal` operation must have a parameter that is a reference to the base class.

但是，虚函数并不是解决这个问题的好办法。麻烦在于决定 `equal` 操作的形参的类型。虚函数在基类类型和派生类型中必须有相同的形参类型，这意味着，虚 `equal` 操作必须有一个形参是基类的引用。

However, when we compare two derived objects, we want to compare data members that might be particular to that derived class. If the parameter is a reference to base, we can use only members that are present in the base class. We cannot access members that are in the derived class but not in the base.

但是，当比较两个派生类对象的时候，我们希望比较可能特定于派生类的数据成员。如果形参是基类的引用，就只能比较基类中出现的成员，我们不能访问在派生类中但不在基类中出现的成员。

Thinking about the problem in this detail, we see that we want to return false if we attempt to compare objects of different types. Given this observation, we can now use RTTI to solve our problem.

Section 18.2. Run-Time Type Identification

仔细考虑这个问题，我们看到，希望在试图比较不同类型的对象时返回假 (`false`)。有了这个观察，现在可以使用 RTTI 解决我们的问题。

We'll define a single equality operator. Each class will define a virtual `equal` function that first casts its operand to the right type. If the cast succeeds, then the real comparison will be performed. If the cast fails, then the `equal` operation will return `false`.

我们将定义单个相等操作符。每个类定义一个虚函数 `equal`，该函数首先将操作数强制转换为正确的类型。如果转换成功，就进行真正的比较；如果转换失败，`equal` 操作就返回 `false`。

The Class Hierarchy

类层次

To make the concept a bit more concrete, let's assume that our classes look something like:

为了使概念更清楚一点，假定类层次是这样的：

```
class Base {
    friend bool operator==(const Base&, const Base&);
public:
    // interface members for Base
protected:
    virtual bool equal(const Base&) const;
    // data and other implementation members of Base
};
class Derived: public Base {
    friend bool operator==(const Base&, const Base&);
public:
    // other interface members for Derived
private:
    bool equal(const Base&) const;
    // data and other implementation members of Derived
};
```

A Type-Sensitive Equality Operator

类型敏感的相等操作符

Next let's look at how we might define the overall equality operator:

下面看看可以怎样定义整体的相等操作符：

```
bool operator==(const Base &lhs, const Base &rhs)
{
    // returns false if typeid's are different otherwise
    // returns lhs.equal(rhs)
    return typeid(lhs) == typeid(rhs) && lhs.equal(rhs);
}
```

This operator returns false if the operands are different types. If they are the same type, then it delegates the real work of comparing the operands to the appropriate virtual `equal` function. If the operands are `Base` objects, then `Base::equal` will be called. If they are `Derived` objects, `Derived::equal` is called.

如果操作数类型不同，这个操作符就返回假；如果操作数类型相同，它就将实际比较操作数的工作委派给适当的虚函数 `equal`。如果操作数是 `Base` 对象，就调用 `Base::equal`；如果操作数是 `Derived` 对象，就调用 `Derived::equal`。

The Virtual `equal` Functions

虚函数 `equal`

Each class in the hierarchy must define its own version of `equal`. The functions in the derived classes will all start the same way: They'll cast their argument to the type of the class itself:

层次中的每个类都必须定义自己的 `equal` 版本。派生类中的 `equal` 函数将以相同的方式开始：它们将实参强制转换为类本身的类型。

```
bool Derived::equal(const Base &rhs) const
{
    if (const Derived *dp
        = dynamic_cast<const Derived*>(&rhs)) {
```

Section 18.2. Run-Time Type Identification

```
// do work to compare two Derived objects and return result
} else
    return false;
}
```

The cast should always succeed after all, the function is called from the equality operator only after testing that the two operands are the same type. However, the cast is necessary so that the function can access the derived members of the right-hand operand. The operand is a `Base&`, so if we want to access members of the `Derived`, we must first do the cast.

这个强制转换应该总是成功——毕竟，只有有测试了两个操作数类型相同之后，才从相等操作符调用该函数。但是，这个强制转换是必要的，以便函数可以访问右操作数的派生类成员。因为操作数是 `Base&`，所以如果想要访问 `Derived` 的成员，就必须首先进行强制转换。

The Base-Class `equal` Function

基类 `equal` 函数

This operation is a bit simpler than the others:

这个操作比其他的简单一点：

```
bool Base::equal(const Base &rhs) const
{
    // do whatever is required to compare to Base objects
}
```

There is no need to cast the parameter before using it. Both `*this` and the parameter are `Base` objects, so all the operations available for this object are also defined for the parameter type.

使用形参之前不必强制转换，`*this` 和形参都是 `Base` 对象，所以对形参类型也定义了该对象可用的所有操作。

18.2.4. The `type_info` Class

18.2.4. `type_info` 类

The exact definition of the `type_info` class varies by compiler, but the standard guarantees that all implementations will provide at least the operations listed in [Table 18.2](#).

`type_info` 类的确切定义随编译器而变化，但是，标准保证所有的实现将至少提供[表 18.2](#) 列出的操作。

Table 18.2. Operations on `type_info`

表 18.2. `type_info` 的操作

<code>t1 == t2</code>	Returns <code>true</code> if the two <code>type_info</code> objects <code>t1</code> and <code>t2</code> refer to the same type; <code>false</code> otherwise. 如果两个对象 <code>t1</code> 和 <code>t2</code> 类型相同，就返回 <code>true</code> ；否则，返回 <code>false</code>
<code>t1 != t2</code>	Returns <code>true</code> if the two <code>type_info</code> objects <code>t1</code> and <code>t2</code> refer to different types; <code>false</code> otherwise. 如果两个对象 <code>t1</code> 和 <code>t2</code> 类型不同，就返回 <code>true</code> ；否则，返回 <code>false</code>
<code>t.name()</code>	Returns a C-style character string that is a printable version of the type name. Type names are generated in a system-dependent way. 返回 C 风格字符串，这是类型名字的可显示版本。类型名字用系统相关的方法产生
<code>t1.before(t2)</code>	Returns a <code>bool</code> that indicates whether <code>t1</code> comes before <code>t2</code> . The ordering imposed by <code>before</code> is compiler-dependent. 返回指出 <code>t1</code> 是否出现在 <code>t2</code> 之前的 <code>bool</code> 值。 <code>before</code> 强制的次序与编译器

有关

The class also provides a public virtual destructor, because it is intended to serve as a base class. If the compiler wants to provide additional type information, it should do so in a class derived from `type_info`.

因为打算作基类使用, `type_info` 类也提供公用虚析构函数。如果编译器想要提供附加的类型信息, 应该在 `type_info` 的派生类中进行。

The default and copy constructors and the assignment operator are all defined as `private`, so we cannot define or copy objects of type `type_info`. The only way to create `type_info` objects in a program is to use the `typeid` operator.

默认构造函数和复制构造函数以及赋值操作符都定义为 `private`, 所以不能定义或复制 `type_info` 类型的对象。程序中创建 `type_info` 对象的唯一方法是使用 `typeid` 操作符。

The `name` function returns a C-style character string for the name of the type represented by the `type_info` object. The value used for a given type depends on the compiler and in particular is not required to match the type names as used in a program. The only guarantee we have about the return from `name` is that it returns a unique string for each type. Nonetheless, the `name` member can be used to print the name of a `type_info` object:

`name` 函数为 `type_info` 对象所表示的类型的名字返回 C 风格字符串。给定类型所用的值取决于编译器, 具体来说, 无须与程序中使用的类型名字匹配。对 `name` 返回值的唯一保证是, 它为每个类型返回唯一的字符串。虽然如此, 仍可以使用 `name` 成员来显示 `type_info` 对象的名字:

```
int iobj;
cout << typeid(iobj).name() << endl
<< typeid(8.16).name() << endl
<< typeid(std::string).name() << endl
<< typeid(Base).name() << endl
<< typeid(Derived).name() << endl;
```

The format and value returned by `name` varies by compiler. This program, when executed on our machine, generates the following output:

`name` 返回的格式和值随编译器而变化。在我们的机器上执行时, 这个程序产生下面的输出:

```
i
d
ss
4Base
7Derived
```



The `type_info` class varies by compiler. Some compilers provide additional member functions that provide additional information about types used in a program. You should consult the reference manual for your compiler to understand the exact `type_info` support provided.

`type_info` 类随编译器而变。一些编译器提供附加的成员函数, 那些函数提供关于程序中所用类型的附加信息。你应该查阅编译器的参考手册来理解所提供的确切的 `type_info` 支持。

Exercises Section 18.2.4

Exercise

18.20:

Given the following class hierarchy in which each class defines a `public` default constructor and virtual destructor, which type name do the following statements print?

给定下面的类层次, 其中每个类都定义了 `public` 默认构造函数及虚析构函数, 下面语句显示哪些类型名?

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public B { /* ... */ };

(a) A *pa = new C;
    cout << typeid(pa).name() << endl;

(b) C cobj;
    A& ra = cobj;
    cout << typeid(&ra).name() << endl;

(c) B *px = new B;
    A& ra = *px;
    cout << typeid(ra).name() << endl;
```

Team LiB

◀ PREVIOUS NEXT ▶

18.3. Pointer to Class Member

18.3. 类成员的指针

We know that, given a pointer to an object, we can fetch a given member from that object using the arrow (`->`) operator. Sometimes it is useful to start from the member. That is, we may want to obtain a pointer to a specific member and then fetch that member from one or another object.

我们知道，给定对象的指针，可以使用箭头 (`->`) 操作符从该对象获得给定成员。有时从成员开始是有用的，也就是说，我们也许希望获得特定成员的指针，然后从一个对象或别的对象获得该成员。

We can do so by using a special kind of pointer known as a **pointer to member**. A pointer to member embodies the type of the class as well as the type of the member. This fact impacts how pointers to member are defined, how they are bound to a function or data member, and how they are used.

可以通过使用称为[成员指针](#)的特殊类型的指针做到这一点。成员指针包含类的类型以及成员的类型。这一事实影响着怎样定义成员指针，怎样将成员指针绑定到函数或数据成员，以及怎样使用它们。

Pointers to member apply only to non`static` members of a class. `static` class members are not part of any object, so no special syntax is needed to point to a `static` member. Pointers to `static` members are ordinary pointers.

成员指针只应用于类的非 `static` 成员。`static` 类成员不是任何对象的组成部分，所以不需要特殊语法来指向 `static` 成员，`static` 成员指针是普通指针。

18.3.1. Declaring a Pointer to Member

18.3.1. 声明成员指针

In exploring pointers to members, we'll use a simplified version of the `Screen` class from [Chapter 12](#).

在研究成员指针时，将使用[第十二章](#)的 `Screen` 类的简化版本。

```
class Screen {
public:
    typedef std::string::size_type index;
    char get() const;
    char get(index ht, index wd) const;
private:
    std::string contents;
    index cursor;
    index height, width;
};
```

Defining a Pointer to Data Member

定义数据成员的指针

The `contents` member of `Screen` has type `std::string`. The complete type of `contents` is "member of class `Screen`, whose type is `std::string`." Consequently, the complete type of a pointer that could point to `contents` is "pointer to member of class `Screen` of type `std::string`." This type is written as

`Screen` 类的 `contents` 成员的类型为 `std::string`。`contents` 的完全类型是“`Screen` 类的成员，其类型是 `std::string`”。因此，可以指向 `contents` 的指针的完全类型是“指向 `std::string` 类型的 `Screen` 类成员的指针”，这个类型可写为

```
string Screen::*
```

We can define a pointer to a `string` member of class `Screen` as

可以将指向 `Screen` 类的 `string` 成员的指针定义为

```
string Screen::*ps_Screen;
```

Section 18.3. Pointer to Class Member

`ps_Screen` could be initialized with the address of `contents` by writing

可以用 `contents` 的地址初始化 `ps_Screen`, 代码为

```
string Screen::*ps_Screen = &Screen::contents;
```

We could also define a pointer that might address the `height`, `width`, or `cursor` members as

还可以将指向 `height`、`width` 或 `cursor` 成员的指针定义为

```
Screen::index Screen::*pindex;
```

which says that `pindex` is a pointer to a member of class `Screen` with type `Screen::index`. We could assign the address of `width` to this pointer as follows:

这是说, `pindex` 是具有 `Screen::index` 类型的 `Screen` 类成员的指针。可以将 `width` 的地址赋给这个指针:

```
pindex = &Screen::width;
```

The pointer `pindex` can be set to any of `width`, `height`, or `cursor` because all three are `Screen` class data members of type `index`.

可以用指针 `pindex` 指向 `width`、`height` 或 `cursor` 中任意一个, 因为这三个都是 `index` 类型的 `Screen` 类成员。

Defining a Pointer to Member Function

定义成员函数的指针

A pointer to a member function must match the type of the function to which it points, in three ways:

成员函数的指针必须在三个方面与它所指函数的类型相匹配:

1. The type and number of the function parameters, including whether the member is `const`

函数形参的类型和数目, 包括成员是否为 `const`。

2. The return type

返回类型。

3. The class type of which it is a member

所属类的类型。

A pointer to member function is defined by specifying the function return type, parameter list, and a class. For example, a pointer to a `Screen` member function capable of referring to the version of `get` that takes no parameters has the following type:

通过指定函数返回类型、形参表和类来定义成员函数的指针。例如, 可引用不接受形参的 `get` 版本的 `Screen` 成员函数的指针具有如下类型:

```
char (Screen::*)( ) const
```

This type specifies a pointer to a `const` member function of class `Screen`, taking no parameters and returning a value of type `char`. A pointer to this version of `get` can be defined and initialized as follows:

这个类型指定 `Screen` 类的 `const` 成员函数的指针, 不接受形参并返回 `char` 类型的值。这个 `get` 函数版本的指针可以像下面这样定义和初始化:

```
// pmf points to the Screen get member that takes no arguments
char (Screen::*pmf)() const = &Screen::get;
```

We might also define a pointer to the two-parameter version of `get` as

也可以将带两个形参的 `get` 函数版本的指针定义为

```
char (Screen::*pmf2)(Screen::index, Screen::index) const;
pmf2 = &Screen::get;
```

The precedence of the call operator is higher than that of the pointer-to-member operators. Therefore, the



parentheses around `Screen::*` are essential. Without them, the compiler treats the following as an (invalid) function declaration:

调用操作符的优先级高于成员指针操作符，因此，包围 `Screen::*` 的括号是必要的，没有这个括号，编译器就将下面代码当作（无效的）函数声明：

```
// error: non-member function p cannot have const qualifier
char Screen::*p() const;
```

Using Typedefs for Member Pointers

为成员指针使用类型别名

Typedefs can make pointers to members easier to read. For example, the following typedef defines `Action` to be an alternative name for the type of the two-parameter version of `get`:

类型别名可以使成员指针更容易阅读。例如，下面的类型别名将 `Action` 定义为带两个形参的 `get` 函数版本的类型的另一名字：

```
// Action is a type name
typedef
char (Screen::*Action)(Screen::index, Screen::index) const;
```

`Action` is the name of the type "pointer to a `const` member function of class `Screen` taking two parameters of type `index` and returning `char`." Using the typedef, we can simplify the definition of a pointer to `get` as follows:

`Action` 是类型“`Screen` 类的接受两个 `index` 类型形参并返回 `char` 的成员函数的指针”的名字。使用类型别名，可以将 `get` 指针的定义简化为

```
Action get = &Screen::get;
```

A pointer-to-member function type may be used to declare function parameters and function return types:

可以使用成员指针函数类型来声明函数形参和函数返回类型：

```
// action takes a reference to a Screen and a pointer to Screen member function
Screen& action(Screen&, Action = &Screen::get);
```

This function is declared as taking two parameters: a reference to a `Screen` object and a pointer to a member function of class `Screen` taking two `index` parameters and returning a `char`. We could call `action` either by passing it a pointer or the address of an appropriate member function in `Screen`:

这个函数声明为接受两个形参：`Screen` 对象的引用，以及 `Screen` 类的接受两个 `index` 类型形参并返回 `char` 的成员函数的指针。可以通过传递 `Screen` 类中适当成员函数的指针或地址调用 `action` 函数：

```
Screen myScreen;
// equivalent calls:
action(myScreen);           // uses the default argument
action(myScreen, get);      // uses the variable get that we previously defined
action(myScreen, &Screen::get); // pass address explicitly
```

Exercises Section 18.3.1

Exercise 18.21: What is the difference between an ordinary data or function pointer and a pointer to data or function member?

普通数据指针或函数指针与数据成员指针或函数成员指针之间的区别是什么？

Section 18.3. Pointer to Class Member

Exercise Define the type that could represent a pointer to the `isbn` member of the `Sales_item` class.

18.22: 定义可以表示 `Sales_item` 类的 `isbn` 成员的指针的类型。

Exercise Define a pointer that could point to the `same_isbn` member.

18.23: 定义可以指向 `same_isbn` 成员的指针。

Exercise Write a `typedef` that is a synonym for a pointer that could point to the `avg_price` member of `Sales_item`.

编写类型别名，作为可指向 `Sales_item` 的 `avg_price` 成员的指针的同义词。

18.3.2. Using a Pointer to Class Member

18.3.2. 使用类成员指针

Analogous to the member access operators, `operators.` and `->`, are two new operators, `.*` and `.->`, that let us bind a pointer to member to an actual object. The left-hand operand of these operators must be an object of or pointer to the class type, respectively. The right-hand operand is a pointer to a member of that type:

类似于成员访问操作符 `.` 和 `->`，`.*` 和 `.->` 是两个新的操作符，它们使我们能够将成员指针绑定到实际对象。这两个操作符的左操作数必须是类类型的对象或类类型的指针，右操作数是该类型的成员指针。

- The pointer-to-member dereference operator (`.*`) fetches the member from an object or reference.

成员指针解引用操作符 (`.*`) 从对象或引用获取成员。

- The pointer-to-member arrow operator (`.->`) fetches the member through a pointer to an object.

成员指针箭头操作符 (`.->`) 通过对象的指针获取成员。

Using a Pointer to Member Function

使用成员函数的指针

Using a pointer to member, we could call the version of `get` that takes no parameters as follows:

使用成员指针，可以这样调用不带形参的 `get` 函数版本：

```
// pmf points to the Screen get member that takes no arguments
char (Screen::*pmf)() const = &Screen::get;
Screen myScreen;
char c1 = myScreen.get();      // call get on myScreen
char c2 = (myScreen.*pmf)();   // equivalent call to get
Screen *pScreen = &myScreen;
c1 = pScreen->get();         // call get on object to which pScreen points
c2 = (pScreen->*pmf)();      // equivalent call to get
```



The calls `(myScreen.*pmf)()` and `(pScreen->*pmf)()` require the parentheses because the precedence of the call operator `()` is higher than the precedence of the pointer to member operators.

因为调用操作符 `()` 比成员指针操作符优先级高，所以调用 `(myScreen.*pmf)()` 和 `(pScreen->*pmf)()` 需要括号。

Without the parentheses,

Section 18.3. Pointer to Class Member

没有括号，就会将

```
myScreen.*pmf()
```

would be interpreted to mean

解释为

```
myScreen.*(pmf())
```

This code says to call the function named `pmf` and bind its return value to the pointer to member object operator (`.*`). Of course, the type of `pmf` does not support such a use, and a compile-time error would be generated.

这个代码是说，调用名为 `pmf` 的函数，并将它的返回值绑定到成员对象操作符 (`.*`) 的指针。当然，`pmf` 的类型不支持这样的使用，且会产生编译时错误。

As with any other function call, we can also pass arguments in a call made through a pointer to member function:

像任何其他函数一样，也可以在通过成员函数指针进行的调用中传递实参：

```
char (Screen::*pmf2)(Screen::index, Screen::index) const;
pmf2 = &Screen::get;
Screen myScreen;
char c1 = myScreen.get(0,0);      // call two-parameter version of get
char c2 = (myScreen.*pmf2)(0,0); // equivalent call to get
```

Using a Pointer to Data Member

使用数据成员的指针

The same pointer-to-member operators are used to access data members:

相同的成员指针操作符用于访问数据成员：

```
Screen::index Screen::*pindex = &Screen::width;
Screen myScreen;
// equivalent ways to fetch width member of myScreen
Screen::index ind1 = myScreen.width;          // directly
Screen::index ind2 = myScreen.*pindex;         // dereference to get width
Screen *pScreen;
// equivalent ways to fetch width member of *pScreen
ind1 = pScreen->width;          // directly
ind2 = pScreen->*pindex;         // dereference pindex to get width
```

Pointer-to-Member Function Tables

成员指针函数表

One common use for function pointers and for pointers to member functions is to store them in a function table. A function table is a collection of function pointers from which a given call is selected at run time.

函数指针和成员函数指针的一个公共用途是，将它们存储在函数表中，函数表是函数指针的集合，在运行时从中选择给定调用。

For a class that has several members of the same type, such a table can be used to select one from the set of these members. Let's assume that our `Screen` class is extended to contain several member functions, each of which moves the cursor in a particular direction:

对具有几个相同类型的成员的类而言，可以使用这样的表来从这些成员的集合中选择一个。假定扩展 `Screen` 类以包含几个成员函数，其中每一个在特定方向移动光标：

```
class Screen {
public:
    // other interface and implementation members as before
    Screen& home();           // cursor movement functions
    Screen& forward();
    Screen& back();
    Screen& up();
    Screen& down();
};
```

Section 18.3. Pointer to Class Member

Each of these new functions takes no parameters and returns a reference to the `Screen` on which it was invoked.

这些新函数的每一个都不接受形参，并返回调用它的对象的引用。

Using the Function-Pointer Table

使用函数指针表

We might want to define a `move` function that could call any one of these functions and perform the indicated action. To support this new function, we'll add a `static` member to `Screen` that will be an array of pointers to the cursor movement functions:

我们可能希望定义一个 `move` 函数，它可以调用这些函数中的任意一个并执行指定动作。为了支持这个新函数，我们将在 `Screen` 中增加一个 `static` 成员，该成员是光标移动函数的指针的数组：

```
class Screen {
public:
    // other interface and implementation members as before
    // Action is pointer that can be assigned any of the cursor movement members
    typedef Screen& (Screen::*Action)();
    static Action Menu[];
    // function table
public:
    // specify which direction to move
    enum Directions { HOME, FORWARD, BACK, UP, DOWN };
    Screen& move(Directions);
};
```

The array named `Menu` will hold pointers to each of the cursor movement functions. Those functions will be stored at the offsets corresponding to the enumerators in `Directions`. The `move` function takes an enumerator and calls the appropriate function:

名为 `Menu` 的数组将保存指向每个光标移动函数的指针，将在对应于 `Directions` 中枚举成员的偏移位置保存那些函数，`move` 函数接受枚举成员并调用适当函数：

```
Screen& Screen::move(Directions cm)
{
    // fetch the element in Menu indexed by cm
    // run that member on behalf of this object
    (this->*Menu[cm])();
    return *this;
}
```

The call inside `move` is evaluated as follows: The `Menu` element indexed by `cm` is fetched. That element is a pointer to a member function of the `Screen` class. We call the member function to which that element points on behalf of the object to which `this` points.

这样计算 `move` 内部的调用：获取由 `cm` 索引的 `Menu` 元素（该元素是 `Screen` 类成员函数的指针），代表 `this` 指向的对象调用该元素指向的成员函数。

When we call `move`, we pass it an enumerator that indicates which direction to move the cursor:

调用 `move` 时，传给它一个枚举成员，指出向哪个方向移动光标：

```
Screen myScreen;

myScreen.move(Screen::HOME);    // invokes myScreen.home
myScreen.move(Screen::DOWN);   // invokes myScreen.down
```

Defining a Table of Member Function Pointers

定义成员函数指针表

What's left is to define and initialize the table itself:

剩下的是定义和初始化表本身：

```
Screen::Action Screen::Menu[] = { &Screen::home,
    &Screen::forward,
    &Screen::back,
    &Screen::up,
    &Screen::down,
};
```

Exercises Section 18.3.2

**Exercise
18.25:**

What is the type of the `Screen` class members `screen` and `cursor`?

`Screen` 类的成员 `screen` 和 `cursor` 的类型是什么？

**Exercise
18.26:**

Define a pointer to member that could point to the `cursor` member of class `Screen`. Fetch the value of `Screen::cursor` through that pointer.

定义一个可以指向 `Screen` 类 `cursor` 成员的成员指针，通过该指针获取 `Screen::cursor` 的值。

**Exercise
18.27:**

Define a `typedef` for each distinct type of `Screen` member function.

为 `Screen` 类成员函数的每个可区分类型定义类型别名。

**Exercise
18.28:**

Pointers to members may also be declared as class data members. Modify the `Screen` class definition to contain a pointer to a `Screen` member function of the same type as `home` and `end`.

也可以将成员指针声明为类的数据成员。修改 `Screen` 类的定义，以便包含与 `home` 和 `end` 类型相同的 `Screen` 成员函数的指针。

**Exercise
18.29:**

Write a `Screen` constructor that takes a parameter of type pointer to `Screen` member function whose parameter list and return type are the same as those for the member functions `home` and `end`.

编写一个 `Screen` 构造函数，该构造函数接受指向 `Screen` 成员函数的指针类型形参，成员函数的形参表和返回类型与成员函数 `home` 和 `end` 的相同。

**Exercise
18.30:**

Provide a default argument for this parameter. Use this parameter to initialize the data member introduced in the previous exercise.

为该形参提供默认实参，使用该形参对上题中引入的数据成员进行初始化。

**Exercise
18.31:**

Provide a `Screen` member function to set this member.

提供一个 `Screen` 成员函数来设置这个成员。

18.4. Nested Classes

18.4. 嵌套类

A class can be defined within another class. Such a class is a **nested class**, also referred to as a **nested type**. Nested classes are most often used to define implementation classes, such as the `QueueItem` class from [Chapter 16](#).

可以在另一个类内部定义一个类，这样的类是**嵌套类**，也称为**嵌套类型**。嵌套类最常用于定义执行类，如[第十六章](#)的 `QueueItem` 类。

Nested classes are independent classes and are largely unrelated to their enclosing class. Objects of the enclosing and nested classes are, therefore, independent from one another. An object of the nested type does not have members defined by the enclosing class. Similarly, an object of the enclosing class does not have members defined by the nested class.

嵌套类是独立的类，基本上与它们的外围类不相关，因此，外围类和嵌套类的对象是互相独立的。嵌套类型的对象不具备外围类所定义的成员，同样，外围类的成员也不具备嵌套类所定义的成员。

The name of a nested class is visible in its enclosing class scope but not in other class scopes or in the scope in which the enclosing class is defined. The name of a nested class will not collide with the same name declared in another scope.

嵌套类的名字在其外围类的作用域中可见，但在其他类作用域或定义外围类的作用域中不可见。嵌套类的名字将不会与另一作用域中声明的名字冲突。

A nested class can have the same kinds of members as a nonnested class. Just like any other class, a nested class controls access to its own members using access labels. Members may be declared `public`, `private`, or `protected`. The enclosing class has no special access to the members of a nested class and the nested class has no special access to members of its enclosing class.

嵌套类可以具有与非嵌套类相同种类的成员。像任何其他类一样，嵌套类使用访问标号控制对自己成员的访问。成员可以声明为 `public`、`private` 或 `protected`。外围类对嵌套类的成员没有特殊访问权，并且嵌套类对其外围类的成员也没有特殊访问权。

A nested class defines a type member in its enclosing class. As with any other member, the enclosing class determines access to this type. A nested class defined in the `public` part of the enclosing class defines a type that may be used anywhere. A nested class defined in the `protected` section defines a type that is accessible only by the enclosing class, its friends, or its derived classes. A `private` nested class defines a type that is accessible only to the members of the enclosing class or its friends.

嵌套类定义了其外围类中的一个类型成员。像任何其他成员一样，外围类决定对这个类型的访问。在外围类的 `public` 部分定义的嵌套类定义了可在任何地方使用的类型，在外围类的 `protected` 部分定义的嵌套类定义了只能由外围类、友元或派生类访问的类型，在外围类的 `private` 部分定义的嵌套类定义了只能被外围类或其友元访问的类型。

18.4.1. A Nested-Class Implementation

18.4.1. 嵌套类的实现

The `Queue` class that we implemented in [Chapter 16](#) defined a companion implementation class named `QueueItem`. That class was a private class it had only `private` members but it was defined at the global scope. General user code cannot use objects of class `QueueItem`: All its members, including constructors, are `private`. However, the name `QueueItem` is visible globally. We cannot define our own type or other entity named `QueueItem`.

在[第十六章](#)中实现的 `Queue` 类定义了一个名为 `QueueItem` 的伙伴执行类，`QueueItem` 类是私有类（它只有 `private` 成员）但它是在全局作用域中定义的。普通用户代码不能使用 `QueueItem` 类的对象：它的所有成员，包括构造函数，均为 `private`。但是，名字 `QueueItem` 是全局可见的，不能定义名为 `QueueItem` 的自有类型或其他实体。

A better design would be to make the `QueueItem` class a `private` member of class `Queue`. That way, the `Queue` class (and its friends) could use `QueueItem`, but the `QueueItem` class type would not be visible to general user code. Once the class itself is `private`, we can make its members publicly `Queue` or the friends of `Queue` can access the `QueueItem` type, so there is no need to protect its members from general program access. We make the members `public` by defining `QueueItem` using the keyword `struct`.

一个更好的设计可能是，将 `QueueItem` 类设为 `Queue` 类的 `private` 成员，那样，`Queue` 类（及其友元）可以使用 `QueueItem`，但 `QueueItem` 类类型对普通用户代码不可见。一旦 `QueueItem` 类本身为 `private`，我们就可以使其成员为 `public` 成员——只有 `Queue` 或 `Queue` 的友元可以访问 `QueueItem` 类型，所以不必防止一般程序访问 `QueueItem` 成员。通过用保留字 `struct` 定义 `QueueItem` 使成员为 `public` 成员。

Our new design looks like:

新的设计如下：

```
template <class Type> class Queue {
    // interface functions to Queue are unchanged
private:
    // public members are ok: QueueItem is a private member of Queue
```

Section 18.4. Nested Classes

```
// only queue and its friends may access the members of QueueItem
struct QueueItem {
    QueueItem(const Type &);
    Type item;           // value stored in this element
    QueueItem *next;     // pointer to next element in the Queue
};

QueueItem *head;      // pointer to first element in Queue
QueueItem *tail;      // pointer to last element in Queue
};
```

Because the class is a `private` member, only members and friends of the `Queue` class can use the `QueueItem` type. Having made the class a `private` member, we can make the `QueueItem` members `public`. Doing so lets us eliminate the friend declarations in `QueueItem`.

因为 `QueueItem` 类是 `private` 成员，所以只有 `Queue` 类的成员和友元可以使用 `QueueItem` 类型。使 `QueueItem` 类成为 `private` 成员之后，就可以使 `QueueItem` 成员 `public`，这样做使我们能够删去 `QueueItem` 中的友元声明。

Classes Nested Inside a Class Template Are Templates

嵌套在类模板内部的类是模板

Because `Queue` is a template, its members are implicitly templates as well. In particular, the nested class `QueueItem` is implicitly a class template. Again, like any other member in `Queue`, the template parameter for `QueueItem` is the same as the template parameter of its enclosing class: class `Queue`.

因为 `Queue` 类是一个模板，它的成员也隐含地是模板。具体而言，嵌套类 `QueueItem` 隐含地是一个类模板。像 `Queue` 类中任何其他成员一样，`QueueItem` 的模板形参与其外层类（`Queue` 类）的模板形参相同。

Each instantiation of `Queue` generates its own `QueueItem` class with the appropriate template argument for `Type`. The mapping between an instantiation for the `QueueItem` class template and an instantiation of the enclosing `Queue` class template is one to one.

`Queue` 类的每次实例化用对应于 `Type` 的适当模板实参产生自己的 `QueueItem` 类。`QueueItem` 类模板的实例化与外围 `Queue` 类模板的实例化之间的映射是一对一的。

Defining the Members of a Nested Class

定义嵌套类的成员

In this version of `QueueItem`, we chose not to define the `QueueItem` constructor inside the class. Instead, we'll define it separately. The only trick is where to define it and how to name it.

这个 `QueueItem` 类版本中，我们选择不在类内部定义 `QueueItem` 构造函数，相反，我们单独定义它。唯一复杂的是在哪里定义以及怎样命名。



A nested-class member defined outside its own class must be defined in the same scope as the scope in which the enclosing class is defined. A member of a nested class defined outside its own class may not be defined inside the enclosing class itself. A member of a nested class is not a member of the enclosing class.

在其类外部定义的嵌套类成员，必须定义在定义外围类的同一作用域中。在其类外部定义的嵌套类的成员，不能定义在外围类内部，嵌套类的成员不是外围类的成员。

The constructor for `QueueItem` is not a member of class `Queue`. Therefore, it cannot be defined elsewhere in the body of class `Queue`. It must be defined at the same scope as the `Queue` class but outside that class. To define a member outside the nested-class body, we must remember that its name is not visible outside the class. To define the constructor, we must indicate that `QueueItem` is a nested class within the scope of class `Queue`. We do so by qualifying the class name `QueueItem` with the name of its enclosing class `Queue`:

`QueueItem` 类的构造函数不是 `Queue` 类的成员，因此，不能将它定义在 `Queue` 类定义体中的任何地方，它必须与 `Queue` 类在同一作用域但在 `Queue` 类的外部定义。为了将成员定义在嵌套类定义体外部，必须记住，成员的名字在类外部是不可见的。要定义这个构造函数，必须指出，`QueueItem` 是 `Queue` 类作用域中的嵌套类，通过用外围 `Queue` 类的名字限定类名 `QueueItem` 来做到这一点：

```
// defines the QueueItem constructor
// for class QueueItem nested inside class Queue<Type>
template <class Type>
Queue<Type>::QueueItem::QueueItem(const Type &t):
    item(t), next(0) { }
```

Of course, both `Queue` and `QueueItem` are class templates. The constructor, therefore, is also a template.

当然, `Queue` 和 `QueueItem` 都是类模板, 因此, 这个构造函数也是模板。

This code defines a function template, parameterized by a single type parameter named `Type`. Reading the name of the function from right to left, this function is the constructor for class `QueueItem`, which is a nested in the scope of class `Queue<Type>`.

这段代码定义了一个函数模板, 以名为 `Type` 的单个类型形参化为形参。从右至左读函数的名字, 这个函数是 `QueueItem` 类的构造函数, 它嵌套在 `Queue<Type>` 类的作用域中。

Defining the Nested Class Outside the Enclosing Class

在外围类外部定义嵌套类

Nested classes often support implementation details for the enclosing class. We might want to prevent users of the enclosing class from seeing the code that implements the nested class.

嵌套类通常支持外围类的实现细节。我们可能希望防止外围类的用户看见嵌套类的实现代码。

For example, we might want to put the definition of class `QueueItem` in its own file, which we would include in those files containing the implementation of the `Queue` class and its members. Just as we can define the members of a nested class outside the class body, we can define the entire class outside the body of the enclosing class:

例如, 我们可能希望将 `QueueItem` 类的定义放在它自己的文件中, 我们可以在 `Queue` 类及其成员的实现文件中包含这个文件。正如可以在类定义体外部定义嵌套类的成员一样, 我们也可以在外围类定义体的外部定义整个嵌套类:

```
template <class Type> class Queue {
    // interface functions to Queue are unchanged
private:
    struct QueueItem; // forward declaration of nested type QueueItem
    QueueItem *head; // pointer to first element in Queue
    QueueItem *tail; // pointer to last element in Queue
};

template <class Type>
struct Queue<Type>::QueueItem {
    QueueItem(const Type &t): item(t), next(0) { }
    Type item; // value stored in this element
    QueueItem *next; // pointer to next element in the Queue
};
```

To define the class body outside its enclosing class, we must qualify the name of the nested class by the name of its enclosing class. Note that we must still declare `QueueItem` in the body of class `Queue`.

为了在外围类的外部定义类体, 必须用外围类的名字限定嵌套类的名字。注意, 我们仍然必须在 `Queue` 类的定义体声明 `QueueItem` 类。

A nested class also can be declared and then later defined in the body of the enclosing class. As with other forward declarations, a forward declaration of a nested class allows for nested classes that have members that refer to one another.

也可以在外围类的定义体中声明然后定义嵌套类。像其他前向声明一样, 嵌套类的前向声明使嵌套类能够具有相互引用的成员。



Until the actual definition of a nested class that is defined outside the class body is seen, that class is an incomplete type ([Section 12.1.4](#), p. 437). All the normal restrictions on using an incomplete type apply.

在看到在类定义体外部定义的嵌套类的实际定义之前, 该类是不完全类型 ([第 12.1.4 节](#)), 应用所有使用不完全类型的常规限制。

Nested-Class Static Member Definitions

嵌套类静态成员定义

Section 18.4. Nested Classes

If `QueueItem` had declared a static member, its definition would also need to be defined in the outer scope. Assuming `QueueItem` had a static member, its definition would look something like:

```
// defines an int static member of QueueItem,  
// which is a type nested inside Queue<Type>  
template <class Type>  
int Queue<Type>::QueueItem::static_mem = 1024;
```

Using Members of the Enclosing Class

使用外围类的成员



There is no connection between the objects of an enclosing scope and objects of its nested type(s).

外围作用域的对象与其嵌套类型的对象之间没有联系。

Nonstatic functions in the nested class have an implicit `this` pointer that points to an object of the nested type. A nested-type object contains only the members of the nested type. The `this` pointer may not be used to fetch members of the enclosing class. Similarly, the nonstatic member functions in the enclosing class have a `this` pointer that points to an object of the enclosing type. That object has only the members defined in the enclosing class.

嵌套类中的非静态函数具有隐含的 `this` 指针，指向嵌套类型的对象。嵌套类型对象只包含嵌套类型的成员，不能使用 `this` 指针获取外围类的成员。同样，外围类中的非静态成员函数也具有 `this` 指针，它指向外围类型的对象，该对象只具有外围类中定义的成员。

Any use of a nonstatic data or function member of the enclosing class requires that it be done through a pointer, reference, or object of the enclosing class. The `pop` function in class `Queue` may not use `item` or `next` directly:

外围类的非静态数据或函数成员的任何使用都要求通过外围类的指针、引用或对象进行。`Queue` 类中的 `pop` 函数不能直接使用 `item` 或 `next`:

```
template <class Type>  
void Queue<Type>::pop()  
{  
    // pop is unchecked: popping off an empty Queue is undefined  
    QueueItem* p = head;           // keep pointer to head so can delete it  
    head = head->next;           // head now points to next element  
    delete p;                   // delete old head element  
}
```

Objects of type `Queue` do not have members named `item` or `next`. Function members of `Queue` can use the `head` and `tail` members, which are pointers to `QueueItem` objects, to fetch those `QueueItem` members.

`Queue` 类型的对象没有名为 `item` 或 `next` 成员。`Queue` 类的函数成员可以使用 `head` 和 `tail` 成员（它们是指向 `QueueItem` 对象的指针）来获取那些 `QueueItem` 成员。

Using Static or Other Type Members

使用静态成员或其他类型的成员

A nested class may refer to the static members, type names, and enumerators (Section 2.7, p. 62) of the enclosing class directly. Of course, referring to a type name or static member outside the scope of the enclosing class requires the scope-resolution operator.

嵌套类可以直接引用外围类的静态成员、类型名和枚举成员（第 2.7 节），当然，引用外围类作用域之外的类型名或静态成员，需要作用域确定操作符。

Instantiation for Nested Templates

嵌套模板的实例化

Section 18.4. Nested Classes

A nested class of a class template is not instantiated automatically when the enclosing class template is instantiated. Like any member function, the nested class is instantiated only if it is itself used in a context that requires a complete class type. For example, a definition such as

实例化外围类模板的时候，不会自动实例化类模板的嵌套类。像任何成员函数一样，只有当在需要完整类类型的情况下使用嵌套类本身的时候，才会实例化嵌套类。例如，像

```
Queue<int> qi; // instantiates Queue<int> but not QueueItem<int>
```

instantiates the template `Queue` with type `int` but does not yet instantiate the type `QueueItem<int>`. The `Queue` members `head` and `tail` are pointers to `QueueItem<int>`. There is no need to instantiate `QueueItem<int>` to define pointers to that class.

这样的定义，用 `int` 类型实例化了 `Queue` 模板，但没有实例化 `QueueItem<int>` 类型。成员 `head` 和 `tail` 是指向 `QueueItem<int>` 指针，这里不需要实例化 `QueueItem<int>` 来定义那个类的指针。

Making `QueueItem` a nested class of the class template `Queue` does not change the instantiation of `QueueItem`. The `QueueItem<int>` class will be instantiated only when `QueueItem` is used in this case, only when `head` or `tail` is dereferenced from a member function of class `Queue<int>`.

使 `QueueItem` 类成为类模板 `Queue` 的嵌套类并不改变 `QueueItem` 的实例化。只有在使用 `QueueItem<int>` 的时候——本例中，只有当 `Queue<int>` 类的成员函数中对 `head` 和 `tail` 解引用的时，才实例化 `Queue<int>` 类。

18.4.2. Name Lookup in Nested Class Scope

18.4.2. 嵌套类作用域中的名字查找

Name lookup ([Section 12.3.1, p. 447](#)) for names used in a nested class proceeds in the same manner as for a normal class, the only difference being that now there may be one or more enclosing class scopes to search.

对嵌套类中所用名字的名字查找 ([第 12.3.1 节](#)) 在普通类的名字查找之前进行，现在唯一的区别是可能要查找一个或多个外围类作用域。



When processing the declarations of the class members, any name used must appear prior to its use. When processing definitions, the entire nested and enclosing class(es) are in scope.

当处理类成员声明的时候，所用的任意名字必须在使用之前出现。当处理定义的时候，整个嵌套类和外围类均在作用域中。

As an example of name lookup in a nested class, consider the following class declarations:

作为嵌套类中名字查找的例子，考虑下面的类声明：

```
class Outer {
public:
    struct Inner {
        // ok: reference to incomplete class
        void process(const Outer&);
        Inner2 val; // error: Outer::Inner2 not in scope
    };
    class Inner2 {
public:
        // ok: Inner2::val used in definition
        Inner2(int i = 0): val(i) {}
        // ok: definition of process compiled after enclosing class is complete
        void process(const Outer &out) { out.handle(); }
private:
        int val;
    };
    void handle() const; // member of class Outer
};
```

The compiler first processes the declarations of the members of classes `Outer`, `Outer::Inner`, and `Outer::Inner2`.

编译器首先处理 `Outer` 类成员的声明 `Outer::Inner` 和 `Outer::Inner2`。

The use of the name `Outer` as a parameter to `Inner::process` is bound to the enclosing class. That class is still incomplete when the declaration of `process` is seen, but the parameter is a reference, so this usage is okay.

将名字 `Outer` 作为 `Inner::process` 形参的使用被绑定到外围类，在看到 `process` 的声明时，那个类仍是不完整的，但形参是一个引用，所以这个使用是正确的。

The declaration of the data member `Inner::val` is an error. The type `Inner2` has not yet been seen.

Section 18.4. Nested Classes

数据成员 `Inner2::val` 的声明是错误的，还没有看到 `Inner2` 类型。

The declarations in `Inner2` pose no problems mostly they just use the built-in type `int`. The only exception is the `process` member function. Its parameter resolves to the incomplete type `Outer`. Because the parameter is a reference, the fact that `Outer` is an incomplete type doesn't matter.

`Inner2` 中的声明看来没有问题——它们大多只使用内置类型 `int`。唯一的例外是成员函数 `process`，它的形参确定为不完全类型 `Outer`。因为其形参是一个引用，所以 `Outer` 为不完全类型是无关紧要的。

The definitions of the constructor and `process` member are not processed by the compiler until the remaining declarations in the enclosing class have been seen. Completing the declarations of class `Outer` puts the declaration of the function `handle` in scope.

直到看到了外围类中的其余声明之后，编译器才处理构造函数和 `process` 成员的定义。对 `Outer` 类声明的完成将函数 `handle` 的声明放在作用域中。

When the compiler looks up the names used in the definitions in class `Inner2`, all the names in class `Inner2` and class `Outer` are in scope. The use of `val`, which appears before the declaration of `val`, is okay: That reference is bound to the data member in class `Inner2`. Similarly, the use of `handle` from class `Outer` in the body of the `Inner2::process` member is okay. The entire `Outer` class is in scope when the members of class `Inner2` are compiled.

当编译器查找 `Inner2` 类中的定义所用的名字时，`Inner2` 类和 `Outer` 类中的所有名字都在作用域中。`val` 的使用（出现在 `val` 的声明之前）是正确的：将该引用绑定到 `Inner2` 类中的数据成员。同样，`Inner2::process` 成员函数体中对 `Outer` 类的 `handle` 的使用也正确，当编译 `Inner2` 类的成员的时候，整个 `Outer` 类在作用域中。

Using the Scope Operator to Control Name Lookup

使用作用域操作符控制名字查找

The global version of `handle` can be accessed using the scope operator:

可以使用作用域操作符访问 `handle` 的全局版本。

```
class Inner2 {
public:
    // ...
    // ok: programmer explicitly specifies which handle to call
    void process(const Outer &out) { ::handle(out); }
};
```

Exercises Section 18.4.2

Exercise 18.32: Reimplement the `Queue` and `QueueItem` classes from [Chapter 16](#) making `QueueItem` a nested class inside `Queue`.

重新实现[第十六章](#)的 `Queue` 和 `QueueItem` 类，使 `QueueItem` 成为 `Queue` 内部的嵌套类。

Exercise 18.33: Explain the pros and cons of the original and the nested-class version of the `Queue` design.
解释 `Queue` 设计的原来版本和嵌套类版本的优缺点。

18.5. Union: A Space-Saving Class

18.5. 联合：节省空间的类

A **union** is a special kind of class. A **union** may have multiple data members, but at any point in time, only one of the members may have a value. When a value is assigned to one member of the **union**, all other members become undefined.

联合是一种特殊的类。一个 **union** 对象可以有多个数据成员，但在任何时刻，只有一个成员可以有值。当将一个值赋给 **union** 对象的一个成员的时候，其他所有都变为未定义的。

The amount of storage allocated for a **union** is at least as much as the amount necessary to contain its largest data member. Like any class, a **union** defines a new type.

为 **union** 对象分配的存储的量至少与包含其最大数据成员的一样多。像任何类一样，一个 **union** 定义了一个新的类型。

Defining a Union

定义联合

Unions offer a convenient way to represent a set of mutually exclusive values that may have different types. As an example, we might have a process that handles different kinds of numeric or character data. That process might define a **union** to hold these values:

联合提供了便利的办法表示一组相互排斥的值，这些值可以是不同类型的。作为例子，我们可能有一个处理不同各类数值或字符数据的过程。该过程可以定义一个 **union** 来保存这些值：

```
// objects of type TokenValue have a single member,
// which could be of any of the listed types
union TokenValue {
    char  eval;
    int   ival;
    double dval;
};
```

A **union** is defined starting with the keyword **union**, followed by an (optional) name for the **union** and a set of member declarations enclosed in curly braces. This code defines a **union** named **TokenValue** that can hold a value that is either a **char**, an **int**, a pointer to **char**, or a **double**. [Section 18.5 \(p. 795\)](#) will look at what it means to omit the **union** name.

一个 **union** 定义以关键字 **union** 开始，后接（可选的）**union** 名字，以及一组以花括号括住的成员声明。这段代码定义了名为 **TokenValue** 的 **union**，它可以保存一个 **char**、**int**、**char** 指针或 **double** 值。本节介绍省略 **union** 名字意味着什么。

Like any class, a **union** type defines how much storage is associated with objects of its type. The size of each **union** object is fixed at compile time: It is at least as large as the size of the **union**'s largest data member.

像任何类一样，**union** 类型定义了与 **union** 类型的对象相关联的内存是多少。每个 **union** 对象的大小在编译时固定的：它至少与 **union** 的最大数据成员一样大。

No Static, Reference, or Class Data Members

没有静态数据成员、引用成员或类数据成员

Some, but not all, class features apply equally to **unions**. For example, like any class, a **union** can specify protection labels to make members **public**, **private**, or **protected**. By default, **unions** behave like **structs**: Unless otherwise specified, the members of a **union** are **public**.

某些（但不是全部）类特征同样适用于 **union**。例如，像任何类一样，**union** 可以指定保护标记使成员成为公用的、私有的或受保护的。默认情况下，**union** 表现得像 **struct**：除非另外指定，否则 **union** 的成员都为 **public** 成员。

A **union** may also define member functions, including constructors and destructors. However, a **union** may not serve as a base class, so a member function may not be **virtual**.

union 也可以定义成员函数，包括构造函数和析构函数。但是，**union** 不能作为基类使用，所以成员函数不能为虚数。

Section 18.5. Union: A Space-Saving Class

A `union` cannot have a static data member or a member that is a reference. Moreover, `unions` cannot have a member of a class type that defines a constructor, destructor, or assignment operator:

`union` 不能具有静态数据成员或引用成员，而且，`union` 不能具有定义了构造函数、析构函数或赋值操作符的类类型的成员：

```
union illegal_members {
    Screen s;           // error: has constructor
    static int is;      // error: static member
    int &rfi;           // error: reference member
    Screen *ps;         // ok: ordinary built-in pointer type
};
```

This restriction includes classes with members that have a constructor, destructor, or assignment operator.

这个限制包括了具有带构造函数、析构函数或赋值操作符的成员的类。

Using a Union Type

使用联合类型

The name of a `union` is a type name:

`union` 的名字是一个类型名：

```
TokenValue first_token = {'a'}; // initialized TokenValue
TokenValue last_token;        // uninitialized TokenValue object
TokenValue *pt = new TokenValue; // pointer to a TokenValue object
```

Like other built-in types, by default `unions` are uninitialized. We can explicitly initialize a `union` in the same way that we can explicitly initialize ([Section 12.4.5](#), p. 464) simple classes. However, we can provide an initializer only for the first member. The initializer must be enclosed in a pair of curly braces. The initialization of `first_token` gives a value to its `cval` member.

像其他内置类型一样，默认情况下 `union` 对象是未初始化的。可以用与显式初始化（[第 12.4.5 节](#)）简单类对象一样的方法显式初始化 `union` 对象。但是，只能为第一个成员提供初始化式。该初始化式必须插在一对花括号中。`first_token` 的初始化给它的 `cval` 成员一个值。

Using Members of a Union

使用联合的成员

The members of an object of `union` type are accessed using the normal member access operators (`.` and `->`):

可以使用普通成员访问操作符（`.` 和 `->`）访问 `union` 类型对象的成员：

```
last_token.cval = 'z';
pt->ival = 42;
```

Giving a value to a data member of a `union` object makes the other data members undefined. When using a `union`, we must always know what type of value is currently stored in the `union`. Retrieving the value stored in the `union` through the wrong data member can lead to a crash or other incorrect program behavior.

给 `union` 对象的某个数据成员一个值使得其他数据成员变为未定义的。使用 `union` 对象时，我们必须总是知道 `union` 对象中当前存储的是什么类型的值。通过错误的数据成员检索保存在 `union` 对象中的值，可能会导致程序崩溃或者其他不正确的程序行为。



The best way to avoid accessing the `union` value through the wrong member is to define a separate object that keeps track of what value is stored in the `union`. This additional object is referred to as the [discriminant](#) of the `union`.

避免通过错误成员访问 `union` 值的最佳办法是，定义一个单独的对象跟踪 `union` 中存储了什么值。这个附加对象称为 `union` 的判别式。

Nested Unions

嵌套联合

Most often `unions` are used as nested types, where the discriminant is a member of the enclosing class:

`union` 最经常用作嵌套类型，其中判别式是外围类的一个成员：

```
class Token {
public:
    // indicates which kind of value is in val
    enum TokenKind {INT, CHAR, DBL};
    TokenKind tok;
    union {                                // unnamed union
        char cval;
        int ival;
        double dval;
    } val;                                // member val is a union of the 3 listed types
};
```

In this class, the enumeration object `tok` serves to indicate which kind of value is stored in the `val` member. That member is an (unnamed) `union` that holds either a `char`, `int`, or `double`.

这个类中，用枚举对象 `tok` 指出 `val` 成员中存储了哪种值，`val` 成员是一个（未命名的）`union`，它保存 `char`、`int` 或 `double` 值。

We often use a `switch` statement ([Section 6.6](#), p. 199) to test the discriminant and then do processing dependent on which value is currently stored in the `union`:

经常使用 `switch` 语句（[第 6.6 节](#)）测试判别式，然后根据 `union` 中当前存储的值进行处理：

```
Token token;
switch (token.tok) {
case Token::INT:
    token.val.ival = 42; break;
case Token::CHAR:
    token.val.cval = 'a'; break;
case Token::DBL:
    token.val.dval = 3.14; break;
}
```

Anonymous Unions

匿名联合

An unnamed `union` that is not used to define an object is referred to as an [anonymous union](#). The names of the members of an anonymous `union` appear in the enclosing scope. For example, here is our `Token` class rewritten to use an anonymous `union`:

不用于定义对象的未命名 `union` 称为匿名联合。匿名 `union` 的成员的名字出现在外围作用域中。例如，使用匿名 `union` 重写的 `Token` 类如下：

```
class Token {
public:
    // indicates which kind of token value is in val
    enum TokenKind {INT, CHAR, DBL};
    TokenKind tok;
    union {                                // anonymous union
        char cval;
        int ival;
        double dval;
    };
};
```

Because an anonymous `union` provides no way to access its members, the members are directly accessible as part of the scope where the anonymous `union` is defined. Rewriting our previous `switch` to use the anonymous-`union` version of our class would look like:

因为匿名 `union` 不提供访问其成员的途径，所以将成员作为定义匿名 `union` 的作用域的一部分直接访问。重写前面的 `switch` 以便使用类的匿名 `union` 版本，如下：

```
Token token;
switch (token.tok) {
case Token::INT:
    token.ival = 42; break;
```

Section 18.5. Union: A Space-Saving Class

```
case Token::CHAR:  
    token.cval = 'a'; break;  
case Token::DBL:  
    token.dval = 3.14; break;  
}
```



An anonymous `union` cannot have private or protected members, nor can an anonymous `union` define member functions.

匿名 `union` 不能有私有成员或受保护成员，也不能定义成员函数。

Team LiB

◀ PREVIOUS NEXT ▶

18.6. Local Classes

18.6. 局部类

A class can be defined inside a function body. Such a class is called a **local class**. A local class defines a type that is visible only in the local scope in which it is defined. Unlike nested classes, the members of a local class are severely restricted.

可以在函数体内部定义类，这样的类称为**局部类**。一个局部类定义了一个类型，该类型只在定义它的局部作用域中可见。与嵌套类不同，局部类的成员是严格受限的。



All members, including functions, of a local class must be completely defined inside the class body. As a result, local classes are much less useful than nested classes.

局部类的所有成员（包括函数）必须完全定义在类定义体内部，因此，局部类远不如嵌套类有用。

In practice, the requirement that members be fully defined within the class limits the complexity of the member functions of a local class. Functions in local classes are rarely more than a few lines of code. Beyond that, the code becomes difficult for the reader to understand.

实际上，成员完全定义在类中的要求限制了局部类成员函数的复杂性。局部类中的函数很少超过数行代码，超过的话，阅读者会难以理解代码。

Similarly, a local class is not permitted to declare `static` data members, there being no way to define them.

类似地，不允许局部类声明 `static` 数据成员，没有办法定义它们。

Local Classes May Not Use Variables from the Function's Scope

局部类不能使用函数作用域中的变量

The names from the enclosing scope that a local class can access are limited. A local class can access only type names, `static` variables ([Section 7.5.2, p. 255](#)), and enumerators defined within the enclosing local scopes. A local class may not use the ordinary local variables of the function in which the class is defined:

局部类可以访问的外围作用域中的名字是有限的。局部类只能访问在外围作用域中定义的类型名、`static` 变量（[第 7.5.2 节](#)）和枚举成员，不能使用定义该类的函数中的变量：

```
int a, val;
void foo(int val)
{
    static int si;
    enum Loc { a = 1024, b };
    // Bar is local to foo
    class Bar {
        public:
            Loc locVal; // ok: uses local type name
            int barVal;
            void fooBar(Loc l = a)           // ok: default argument is Loc::a
            {
                barVal = val;             // error: val is local to foo
                barVal = ::val;           // ok: uses global object
                barVal = si;              // ok: uses static local object
                locVal = b;               // ok: uses enumerator
            }
    };
    // ...
}
```

Normal Protection Rules Apply to Local Classes

常规保护规则适用于局部类

The enclosing function has no special access privileges to the private members of the local class. Of course, the local class could make the enclosing function a friend.

外围函数对局部类的私有成员没有特殊访问权，当然，局部类可以将外围函数设为友元。



In practice, `private` members are hardly ever necessary in a local class. Often all members of a local class are `public`.

实际上，局部类中 `private` 成员几乎是不必要的，通常局部类的所有成员都为 `public` 成员。

The portion of a program that can access a local class is very limited. A local class is encapsulated within its local scope. Further encapsulation through information hiding is often overkill.

可以访问局部类的程序部分是非常有限的。局部类封装在它的局部作用域中，进一步通过信息隐藏进行封装通常是不必要的。

Name Lookup within a Local Class

局部类中的名字查找

Name lookup within the body of a local class happens in the same manner as for other classes. Names used in the declarations of the members of the class must be in scope before the use of the name. Names used in definitions of the members can appear anywhere in the scope of the local class. Names not resolved to class members are searched first in the enclosing local scope and then out to the scope enclosing the function itself.

局部类定义体中的名字查找方式与其他类的相同。类成员声明中所用的名字必须在名字使用之前出现在作用域中，成员定义中所用的名字可以出现在局部类作用域的任何地方。没有确定为类成员的名字首先在外围局部作用域中进行查找，然后在包围函数本身的作用域中查找。

Nested Local Classes

嵌套的局部类

It is possible to nest a class inside a local class. In this case, the nested class definition can appear outside the local-class body. However, the nested class must be defined in the same local scope as that in which the local class is defined. As usual, the name of the nested class must be qualified by the name of the enclosing class and a declaration of the nested class must appear in the definition of the local class:

可以将一个类嵌套在局部类内部。这种情况下，嵌套类定义可以出现在局部类定义体之外，但是，嵌套类必须在定义局部类的同一作用域中定义。照常，嵌套类的名字必须用外层类的名字进行限定，并且嵌套类的声明必须出现在局部类的定义中：

```
void foo()
{
    class Bar {
        public:
        // ...
        class Nested; // declares class Nested
    };
    // definition of Nested
    class Bar::Nested {
        // ...
    };
}
```

A class nested in a local class is itself a local class, with all the attendant restrictions. All members of the nested class must be defined inside the body of the nested class itself.

嵌套在局部类中的类本身是一个带有所有附加限制的局部类。嵌套类的所有成员必须在嵌套类本身定义体内部定义。

18.7. Inherently Nonportable Features

18.7. 固有的不可移植的特征

One of the hallmarks of the C programming language is the ability to write low-level programs that can be readily moved from one machine to another. The process of moving a program to a new machine is referred to as "porting," so C programs are said to be [portable](#).

编写可以容易从一个机器移到其他机器的低级程序是 C 程序设计语言的一个特点。将程序移到新机器的过程称为“移植”，所以说 C 程序是可移植的。

To support low-level programming, C defines some features that are inherently nonportable. The fact that the size of the arithmetic types vary across machines ([Section 2.1](#), p. 34) is one such nonportable feature that we have already encountered. In this section we'll cover two additional nonportable features that C++ inherits from C: bit-fields and the `volatile` qualifier. These features make it easier to interface directly to hardware.

为了支持低级编程，C 语言定义了一些固有不可移植的特征。算术类型的大小随机器不同而变化的事实（[第 2.1 节](#)），就是我们已经遇到过的一个这样的不可移植特征。本节将讨论 C++ 的另外两个从 C 语言继承来的不可移植特征：位域和 `volatile` 限定符。这些特征可使与硬件接口的直接通信更容易。

C++ adds another nonportable feature to those that it inherits from C: linkage directives, which make it possible to link to programs written in other languages.

C++ 还增加了另一个不可移植特征（从 C 语言继承来的）：链接指示，它使得可以链接到用其他语言编写的程序。

18.7.1. Bit-fields

18.7.1. 位域

A special class data member, referred to as a [bit-field](#), can be declared to hold a specified number of bits. Bit-fields are normally used when a program needs to pass binary data to another program or hardware device.

可以声明一种特殊的类数据成员，称为[位域](#)，来保存特定的位数。当程序需要将二进制数据传递给另一程序或硬件设备的时候，通常使用位域。



The layout in memory of a bit-field is machine-dependent.

位域在内存中的布局是机器相关的。

A bit-field must be an integral data type. It can be either `signed` or `unsigned`. We indicate that a member is a bit-field by following the member name with a colon and a constant expression specifying the number of bits:

位域必须是整型数据类型，可以是 `signed` 或 `unsigned`。通过在成员名后面接一个冒号以及指定位数的常量表达式，指出成员是一个位域：

```
typedef unsigned int Bit;

class File {
    Bit mode: 2;
    Bit modified: 1;
    Bit prot_owner: 3;
    Bit prot_group: 3;
    Bit prot_world: 3;
    // ...
};
```

The `mode` bit-field has two bits, `modified` only one, and the other members each have three bits. Bit-fields defined in consecutive order within the class body are, if possible, packed within adjacent bits of the same integer, thereby providing for storage compaction. For example, in the preceding declaration, the five bit-fields will be stored in the single `unsigned int` first associated with the bit-field `mode`. Whether and how the bits are packed into the integer is machine-dependent.

`mode` 位域有两个位，`modified` 只有一位，其他每个成员有三个位。（如果可能）将类定义体中按相邻次序定义的位域压缩在同一整数的相邻位，从而提供存储压缩。例如，在前面的声明中，5 个位域将存储在一个首先与位域 `mode` 关联的 `unsigned int` 中。位是否压缩到整数以及如何压缩与机器有关。

Section 18.7. Inherently Nonportable Features



Ordinarily it is best to make a bit-field an `unsigned` type. The behavior of bit-fields stored in a `signed` type is implementation-defined.

通常最好将位域设为 `unsigned` 类型。存储在 `signed` 类型中的位域的行为由实现定义。

Using Bit-fields

使用位域

A bit-field is accessed in much the same manner as the other data members of a class. For example, a bit-field that is a `private` member of its class can be accessed only from within the definitions of the member functions and friends of its class:

用与类的其他数据成员相同的方式访问位域。例如，作为类的 `private` 成员的位域只能从成员函数的定义和类的友元访问：

```
void File::write()
{
    modified = 1;
    // ...
}

void File::close()
{
    if (modified)
        // ... save contents
}
```

Bit-fields with more than one bit are usually manipulated using the built-in bitwise operators ([Section 5.3](#), p. 154):

通常使用内置按位操作符 ([第 5.3 节](#)) 操纵超过一位的位域：

```
enum { READ = 01, WRITE = 02 }; // File modes

int main() {
    File myFile;

    myFile.mode |= READ; // set the READ bit
    if (myFile.mode & READ) // if the READ bit is on
        cout << "myFile.mode READ is set\n";
}
```

Classes that define bit-field members also usually define a set of inline member functions to test and set the value of the bit-field. For example, the class `File` might define the members `isRead` and `isWrite`:

定义了位域成员的类通常也定义一组内联成员函数来测试和设置位域的值。例如，`File` 类可以定义成员 `isRead` 和 `isWrite`：

```
inline int File::isRead() { return mode & READ; }
inline int File::isWrite() { return mode & WRITE; }

if (myFile.isRead()) /* ... */
```

With these member functions, the bit-fields can now be declared as private members of class `File`.

有了这些成员函数，现在就可以将位域声明为 `File` 类的私有成员了。

The address-of operator (`&`) cannot be applied to a bit-field, so there can be no pointers referring to class bit-fields. Nor can a bit-field be a static member of its class.

地址操作符 (`&`) 不能应用于位域，所以不可能有引用类位域的指针，位域也不能是类的静态成员。

18.7.2. `volatile` Qualifier

18.7.2. `volatile` 限定符



The precise meaning of `volatile` is inherently machine-dependent and can be understood only by reading the compiler documentation. Programs that use `volatile` usually must be changed when they are moved to new machines or compilers.

`volatile` 的确切含义与机器相关，只能通过阅读编译器文档来理解。使用 `volatile` 的程序在移到新的机器或编译器时通常必须改变。

Programs that deal directly with hardware often have data elements whose value is controlled by processes outside the direct control of the program itself. For example, a program might contain a variable updated by the system clock. An object should be declared `volatile` when its value might be changed in ways outside either the control or detection of the compiler. The `volatile` keyword is a directive to the compiler that it should not perform optimizations on such objects.

直接处理硬件的程序常具有这样的数据成员，它们的值由程序本身直接控制之外的过程所控制。例如，程序可以包含由系统时钟更新的变量。当可以用编译器的控制或检测之外的方式改变对象值的时候，应该将对象声明为 `volatile`。关键字 `volatile` 是给编译器的指示，指出对这样的对象不应该执行优化。

The `volatile` qualifier is used in much the same way as is the `const` qualifier. It is an additional modifier to a type:

用与 `const` 限定符相同的方式使用 `volatile` 限定符。`volatile` 限定符是一个对类型的附加修饰符：

```
volatile int display_register;
volatile Task *curr_task;
volatile int ixa[max_size];
volatile Screen bitmap_buf;
```

`display_register` 是一个 `volatile` 对象，其类型为 `int`。`curr_task` 是一个指向 `volatile Task` 对象的指针。`ixa` 是一个 `volatile` 整数数组，该数组的每个元素都是 `volatile` 的。`bitmap_buf` 是一个 `volatile Screen` 对象，其所有数据成员都是 `volatile` 的。

`display_register` 是 `int` 类型的 `volatile` 对象；`curr_task` 是 `volatile` 对象的指针；`ixa` 是整数的 `volatile` 数组，该数组的每个元素都认为是 `volatile` 的；`bitmap_buf` 是 `volatile Screen` 对象，它的每个成员都认为是 `volatile` 的。

In the same way that a class may define `const` member functions, it can also define member functions as `volatile`. Only `volatile` member functions may be called on `volatile` objects.

用与定义 `const` 成员函数相同的方式，类也可以将成员函数定义为 `volatile`，`volatile` 对象只能调用 `volatile` 成员函数。

[Section 4.2.5](#) (p. 126) 描述了 `const` 限定符与指针的相互作用。同样的，`volatile` 限定符与指针之间也存在同样的相互作用。可以声明 `volatile` 指针、指向 `volatile` 对象的指针，以及指向 `volatile` 对象的 `volatile` 指针：

```
volatile int v;      // v is a volatile int
int *volatile vip; // vip is a volatile pointer to int
volatile int *ivp;  // ivp is a pointer to volatile int
// ivp is a volatile pointer to volatile int
volatile int *volatile vivp;
int *ip = &v; // error: must use pointer to volatile
*ivp = &v;   // ok: ivp is pointer to volatile
vivp = &v;   // ok: vivp is volatile pointer to volatile
```

As with `const`, we may assign the address of a `volatile` object (or copy a pointer to a `volatile` type) only to a pointer to `volatile`. We may use a `volatile` object to initialize a reference only if the reference is `volatile`.

像用 `const` 一样，只能将 `volatile` 对象的地址赋给指向 `volatile` 的指针，或者将指向 `volatile` 类型的指针复制给指向 `volatile` 的指针。只有当引用为 `volatile` 时，我们才可以使用 `volatile` 对象对引用进行初始化。

Synthesized Copy Control Does Not Apply to Volatile Objects

合成的复制控制不适用于 `volatile` 对象

One important difference between the treatment of `const` and `volatile` is that the synthesized copy and assignment operators cannot be used to

Section 18.7. Inherently Nonportable Features

initialize or assign from a `volatile` object. The synthesized copy-control members take parameters that are `const` references to the class type. However, a `volatile` object cannot be passed to a plain or `const` reference.

对待 `const` 和 `volatile` 的一个重要区别是，不能使用合成的复制和赋值操作符从 `volatile` 对象进行初始化或赋值。合成的复制控制成员接受 `const` 形参，这些形参是对类型的 `const` 引用，但是，不能将 `volatile` 对象传递给普通引用或 `const` 引用。

If a class wants to allow `volatile` objects to be copied or to allow assignment from or to a `volatile` operand, it must define its own versions of the copy constructor and/or assignment operator:

如果类希望允许复制 `volatile` 对象，或者，类希望允许从 `volatile` 操作数或对 `volatile` 操作数进行赋值，它必须定义自己的复制构造函数和／或赋值操作符版本：

```
class Foo {  
public:  
    Foo(const volatile Foo&); // copy from a volatile object  
    // assign from a volatile object to a non volatile object  
    Foo& operator=(volatile const Foo&);  
    // assign from a volatile object to a volatile object  
    Foo& operator=(volatile const Foo&) volatile;  
    // remainder of class Foo  
};
```

By defining the parameter to the copy-control members as a `const volatile` reference, we can copy or assign from any kind of `Foo`: a plain `Foo`, a `const Foo`, a `volatile Foo`, or a `const volatile Foo`.

通过将复制控制成员的形参定义为 `const volatile` 引用，我们可以从任何各类的 `Foo` 对象进行复制或赋值：普通 `Foo` 对象、`const Foo` 对象、`volatile Foo` 对象或 `const volatile Foo` 对象。



Although we can define the copy-control members to handle `volatile` objects, a deeper question is whether it makes any sense to copy a `volatile` object. The answer to that question depends intimately on the reason for using `volatile` in any particular program.

虽然可以定义复制控制成员来处理 `volatile` 对象，但更深入的问题是复制 `volatile` 对象是否有意义，对该问题的回答与任意特定程序中使用 `volatile` 的原因密切相关。

18.7.3. Linkage Directives: `extern "C"`

18.7.3. 链接指示 `extern "C"`

C++ programs sometimes need to call functions written in another programming language. Most often, that other language is C. Like any name, the name of a function written in another language must be declared. That declaration must specify the return type and parameter list. The compiler checks calls to external-language functions in the same way that it handles ordinary C++ functions. However, the compiler typically must generate different code to call functions written in other languages. C++ uses [linkage directives](#) to indicate the language used for any non-C++ function.

C++ 程序有时需要调用用其他程序设计语言编写的函数，最常见的一语言是 C 语言。像任何名字一样，必须声明用其他语言编写的函数的名字，该声明必须指定返回类型和形参表。编译器按处理普通 C++ 函数一样的方式检查对外部语言函数的调用，但是，编译器一般必须产生不同的代码来调用用其他语言编写的函数。C++ 使用[链接指示](#)指出任意非 C++ 函数所用的语言。

Declaring a Non-C++ Function

声明非 C++ 函数

A linkage directive can have one of two forms: single or compound. Linkage directives may not appear inside a class or function definition. The linkage directive must appear on the first declaration of a function.

链接指示有两种形式：单个的或复合的。链接指示不能出现在类定义或函数定义的内部，它必须出现在函数的第一次声明上。

As an example, let's look at some of the C functions declared in the `cstdlib` header. Declarations in that header might look something like

作为例子，看看头文件 `cstdlib` 中声明的一些 C 函数。该头文件中声明形如：

```
// illustrative linkage directives that might appear in the C++ header <cstring>
```

Section 18.7. Inherently Nonportable Features

```
// single statement linkage directive
extern "C" size_t strlen(const char *);
// compound statement linkage directive
extern "C" {
    int strcmp(const char*, const char*);
    char *strcat(char*, const char*);
}
```

The first form consists of the `extern` keyword followed by a string literal, followed by an "ordinary" function declaration. The string literal indicates the language in which the function is written.

第一种形式由关键字 `extern` 后接字符串字面值，再接“普通”函数声明构成。字符串字面值指出编写函数所用的语言。

We can give the same linkage to several functions at once by enclosing their declarations inside curly braces following the linkage directive. These braces serve to group the declarations to which the linkage directive applies. The braces are otherwise ignored, and the names of functions declared within the braces are visible as if the functions were declared outside the braces.

通过将几个函数的声明放在跟在链接指示之后的花括号内部，可以给它们设定相同的链接。花括号的作用是将应用链接指示的声明聚合起来，忽略了花括号，花括号中声明的函数名就是可见的，就像在花括号之外声明函数一样。

Linkage Directives and Header Files

链接指示与头文件

The multiple-declaration form can be applied to an entire header file. For example, the C++ `cstring` header might look like

可以将多重声明形式应用于整个头文件。例如，C++ 的 `cstring` 头文件可以像这样：

```
// compound statement linkage directive
extern "C" {
#include <string.h>      // C functions that manipulate C-style strings
}
```

When a `#include` directive is enclosed in the braces of a compound linkage directive, all ordinary function declarations in the header file are assumed to be functions written in the language of the linkage directive. Linkage directives can be nested, so if the header contained a function with a linkage directive the linkage of that function is unaffected.

当将 `#include` 指示在复合链接指示的花括号中的时候，假定头文件中的所有普通函数声明都是用链接指示的语言编写的函数。链接指示可以嵌套，所以，如果头文件包含了带链接指示的函数，该函数的链接不受影响。



The functions that C++ inherits from the C library are permitted to be defined as C functions but are not required to be C functions—it's up to each C++ implementation to decide whether to implement the C library functions in C or C++.

允许将 C++ 从 C 函数库继承而来的函数定义为 C 函数，但不是必须定义为 C 函数——决定是用 C 还是用 C++ 实现 C 函数库，是每个 C++ 实现的事情。

Exporting Our C++ Functions to Other Languages

导出 C++ 函数到其他语言

By using the linkage directive on a function definition, we can make a C++ function available to a program written in another language:

通过对函数定义使用链接指示，使得用其他语言编写的程序可以使用 C++ 函数：

```
// the calc function can be called from C programs
extern "C" double calc(double dparam) { /* ... */ }
```

When the compiler generates code for this function, it will generate code appropriate to the indicated language.

Section 18.7. Inherently Nonportable Features

当编译器为该函数产生代码的时候，它将产生适合于指定语言的代码。



Every declaration of a function defined with a linkage directive must use the same linkage directive.

用链接指示定义的函数的每个声明都必须使用相同的链接指示。

Languages Supported by Linkage Directives

链接指示支持的语言

A compiler is required to support linkage directives for C. A compiler may provide linkage specifications for other languages. For example, `extern "Ada"`, `extern "FORTRAN"`, and so on.

要求编译器支持对 C 语言的链接指示。编译器可以为其他语言提供链接说明。例如，`extern "Ada"`、`extern "FORTRAN"` 等。



What languages are supported varies by compiler. You must consult the user's guide for further information on any non-C linkage specifications it may provide.

支持什么语言随编译器而变。你必须查阅用户指南，获得关于编译器可以提供的任意非 C 链接说明的进一步信息。

Preprocessor Support for Linking to C

对链接到 C 的预处理器支持

It can be useful sometimes to compile the same source file in both C or C++. The preprocessor name `_cplusplus` (two underscores) is automatically defined when compiling C++, so we can conditionally include code based on whether we are compiling C++.

有时需要在 C 和 C++ 中编译同一源文件。当编译 C++ 时，自动定义预处理器名字 `_cplusplus` (两个下划线)，所以，可以根据是否正在编译 C++ 有条件地包含代码。

```
#ifdef __cplusplus
// ok: we're compiling C++
extern "C"
#endif
int strcmp(const char*, const char*);
```

Overloaded Functions and Linkage Directives

重载函数与链接指示

The interaction between linkage directives and function overloading depends on the target language. If the language supports overloaded functions, then it is likely that a compiler that implements linkage directives for that language would also support overloading of these functions from C++.

链接指示与函数重载之间的相互作用依赖于目标语言。如果语言支持重载函数，则为该语言实现链接指示的编译器很可能也支持 C++ 的这些函数的重载。

The only language guaranteed to be supported by C++ is C. The C language does not support function overloading, so it should not be a surprise that a linkage directive can be specified only for one C function in a set of overloaded functions. It is an error to declare more than one function with C linkage with a given name:

C++ 保证支持的唯一语言是 C。C 语言不支持函数重载，所以，不应该对下面的情况感到惊讶：在一组重载函数中只能为一个 C 函数指定链接指示。用带给定名字的 C 链接声明多于一个函数是错误的：

Section 18.7. Inherently Nonportable Features

```
// error: two extern "C" functions in set of overloaded functions
extern "C" void print(const char*); // C version
extern "C" void print(int); // C++ version
```

In C++ programs, it is fairly common to overload C functions. However, the other functions in the overload set must all be C++ functions:

在 C++ 程序中，重载 C 函数很常见，但是，重载集合中的其他函数必须都是 C++ 函数：

```
class SmallInt { /* ... */ };
class BigNum { /* ... */ };
// the C function can be called from C and C++ programs
// the C++ functions overload that function and are callable from C++
extern "C" double calc(double);
extern SmallInt calc(const SmallInt&);
extern BigNum calc(const BigNum&);
```

The C version of `calc` can be called from C programs and from C++ programs. The additional functions are C++ functions with class parameters that can be called only from C++ programs. The order of the declarations is not significant.

可以从 C 程序和 C++ 程序调用 `calc` 的 C 版本。其余函数是带类型形参的 C++ 函数，只能从 C++ 程序调用。声明的次序不重要。

Pointers to `extern "C"` Functions

`extern "C"` 函数和指针

The language in which a function is written is part of its type. To declare a pointer to a function written in another programming language, we must use a linkage directive:

编写函数所用的语言是函数类型的一部分。为了声明用其他程序设计语言编写的函数的指针，必须使用链接指示：

```
// pf points to a C function returning void taking an int
extern "C" void (*pf)(int);
```

When `pf` is used to call a function, the function call is compiled assuming that the call is to a C function.

使用 `pf` 调用函数的时候，假定该调用是一个 C 函数调用而编译该函数。



A pointer to a C function does not have the same type as a pointer to a C++ function. A pointer to a C function cannot be initialized or be assigned to point to a C++ function (and vice versa).

C 函数的指针与 C++ 函数的指针具有不同的类型，不能将 C 函数的指针初始化或赋值为 C++ 函数的指针（反之亦然）。

When there is such a mismatch, a compile-time error message is issued:

存在这种不匹配的时候，会给出编译时错误：

```
void (*pf1)(int); // points to a C++ function
extern "C" void (*pf2)(int); // points to a C function
pf1 = pf2; // error: pf1 and pf2 have different types
```



Some C++ compilers may accept the preceding assignment as a language extension, even though, strictly speaking, it is illegal.

一些 C++ 编译器可以接受前面的赋值作为语言扩展，尽管严格说来它是非法的。

Linkage Directives Apply to the Entire Declaration

应用于整个声明的链接指示

When we use a linkage directive, it applies to the function and any function pointers used as the return type or as a parameter type:

使用链接指示的时候，它应用于函数和任何函数指针，作为返回类型或形参类型使用：

```
// f1 is a C function; its parameter is a pointer to a C function
extern "C" void f1(void(*)(int));
```

This declaration says that `f1` is a C function that doesn't return a value. It has one parameter, which is a pointer to a function that returns nothing and takes a single `int` parameter. The linkage directive applies to the function pointer as well as to `f1`. When we call `f1`, we must pass it the name of a C function or a pointer to a C function.

这个声明是说，`f1` 是一个不返回值的 C 函数，它有一个形参，该形参是不返回值并接受单个形参的函数的指针。链接指示应用于该函数指针以及 `f1`。调用的时候，必须将 C 函数名字或 C 函数指针传递给它。

Because a linkage directive applies to all the functions in a declaration, we must use a `typedef` to pass a pointer to a C function to a C++ function:

因为链接指示应用于一个声明中的所有函数，所以必须使用类型别名，以便将 C 函数的指针传递给 C++ 函数：

```
// FC is a pointer to C function
extern "C" typedef void FC(int);
// f2 is a C++ function with a parameter that is a pointer to a C function
void f2(FC *);
```

Exercises Section 18.7.3

Exercise Explain these declarations and indicate whether they are legal:

18.34:

解释下面这些声明，并指出它们是否合法：

```
extern "C" int compute(int *, int);
extern "C" double compute(double *, double);
```

Chapter Summary

小结

C++ provides several specialized facilities that are tailored to particular kinds of problems.

C++ 讲述了几个专门针对一些特定问题的特殊设施。

Customized memory management is used by classes in two ways: A class may need to define its own internal memory allocation that allows it to streamline allocation of its own data members. A class might want to define its own, class-specific `operator new` and `operator delete` functions that will be used whenever new objects of the class type are allocated.

类的自定义内存管理有两种方式：定义自己的内部内存分配，以简化自己的数据成员的分配；定义自己的、类特定的 `operator new` 和 `operator delete` 函数，在分配类型的新对象时使用它们。

Some programs need to directly interrogate the dynamic type of an object at run time. Run-time type identification (RTTI) provides language level support for this kind of programming. RTTI applies only to classes that define virtual functions; type information for types that do not define virtual functions is available but reflects the static type.

一些程序需要在运行时直接询问对象的动态类型。运行时类型识别（RTTI）为这类程序设计提供语言级支持。RTTI 只适用于定义了虚函数的类，没有定义虚函数的类型的类型信息是可用的但反映静态类型。

Pointers to ordinary objects are typed. When we define a pointer to a class member, the pointer type must also encapsulate the type of the class to which the pointer points. A pointer to member may be bound to any member of the class that has the same type. When we dereference a pointer to member, an object from which to fetch the member must be specified.

普通对象的指针是有类型的。定义类成员的指针的时候，指针类型必须也封装指针所指向的类类型。可以将成员指针绑定到具有相同类型的任意类成员，引用成员指针的时候，必须指定从中获取成员的对象。

C++ defines several additional aggregate types:

C++ 还定义了另外几个聚焦类型：

- Nested classes, which are classes defined in the scope of another class. Such classes are often defined as implementation classes of its enclosing class.
嵌套类，它是在另一个类的作用域中定义的类，这样的类经常定义为其外围类的具体实现类。
- Unions are a special kind of class that may contain only simple data members. An object of a `union` type may define a value for only one of its data members at any one time. Unions are most often nested inside another class type.
联合，是只能包含简单数据成员的一种特殊类。`union` 类型的对象在任意时刻只能为它的一个数据成员定义值。联合经常嵌套在其他类类型内部。
- Local classes, which are very simple classes defined local to a function. All members of a local class must be defined in the class body. There are no static data members of a local class.
局部类，是局部于函数而定义的非常简单的类。局部类的所有成员必须定义在类定义体中，局部类没有静态数据成员。

C++ also supports several inherently nonportable features including bit-fields and `volatile`, which make it easier to interface to hardware, and linkage directives, which make it easier to interface to programs written in other languages.

C++ 还支持几种固有的不可移植的特征，包括位域和 `volatile`（它们可使与硬件接口更容易）以及链接指示（它使得与用其他语言编写的程序接口更容易）。

Defined Terms

术语

allocator class (allocator 类)

Standard library class that supports type-specific allocation of raw, unconstructed memory. The `allocator` class is a class template that defines member functions to `allocate`, `deallocate`, `construct`, and `destroy` objects of the `allocator`'s template parameter type.

标准库类，支持原始未构造内存的类特定的分配。`allocator` 类是一个类模板，定义了成员函数，对 `allocator` 的模板形参类型的对象进行 `allocate`、`deallocate`、`construct` 和 `destroy`。

anonymous union (匿名联合)

Unnamed union that is not used to define an object. Members of the anonymous union are referred to directly. These unions may not have member functions and may not have private or protected members.

不用于定义对象的未命名联合。直接引用匿名联合的成员。这些联合不能具有成员函数，也不能具有私有或受保护成员。

bit-field (位域)

Class member with an signed or unsigned integral type that specifies the number of bits to allocate to the member. Bit-fields defined in consecutive order in the class are, if possible, compacted into a common integral value.

有符号或无符号整型类成员，它指定了分配给成员的位数。如果可能，将类中以连续次序定义的位域压缩到公共整型值中。

delete expression (delete 表达式)

A `delete` expression destroys a dynamically allocated object of a specified type and frees the memory used by that object. A `delete[]` expression destroys the elements of a dynamically allocated array of a specified type and frees the memory used by the array. These expressions use the corresponding version of the library or class-specific `operator delete` functions to free raw memory that held the object or array.

`delete` 表达式撤销特定类型的动态分配对象，并释放该对象所用的内存。`delete[]` 表达式撤销特定类型动态分配数组的元素，并释放数组使用的内存。这些表达式使用库函数或类特定的 `operator delete` 函数的对应版本来释放保存对象或数组的原始内存。

discriminant (判别式)

Programming technique that uses an object to determine which actual type is held in a union at any given time.

一种编程技术，使用对象来确定任意给定时刻联合中保存的实际类型。

dynamic cast

Operator that performs a checked cast from a base type to a derived type. The base type must define at least one `virtual` function. The operator checks the dynamic type of the object to which the reference or pointer is bound. If the object type is the same as the type of the cast (or a type derived from that type), then the cast is done. Otherwise, a zero pointer is returned for a pointer cast, or an exception is thrown for a cast of a reference.

执行从基类类型到派生类型的带检查强制转换的操作符。基类类型必须定义至少一个 `virtual` 函数。这个操作符检查引用或指针所绑定对象的动态类型。如果对象类型与转换的类型（或者从该类型派生的类型）相同，则进行转换；否则，指针转换返回 0 指针，引用的转换抛出一个异常。

freelist (自由列表)

Memory management technique that involves preallocating unconstructed memory to hold objects that will be created as needed. When objects are freed, their memory is put back on the free list rather than being returned to the system.

一种内存管理技术，涉及预先分配未构造内存以保存在需要时创建的对象。释放对象的时候，将它们的内存放回自由列表，而不是返还给系统。

linkage directive (链接指示)

Mechanism used to allow functions written in a different language to be called from a C++ program. All compilers must support calling C and C++ functions. It is compiler-dependent whether any other languages are supported.

一种机制，用于允许从 C++ 程序调用以不同语言编写的函数。所有编译器都支持调用 C 和 C++ 函数，是否支持任何其他语言随编译器而定。

[local class \(局部类\)](#)

Class defined inside a function. A local class is visible only inside the function in which it is defined. All members of the class must be defined inside the class body. There can be no static members of a local class. Local class members may not access the local variables defined in the enclosing function. They may use type names, static variables, or enumerators defined in the enclosing function.

在函数内部定义的类。局部类只在定义它的函数内部可见，类的所有成员必须定义在类定义体内部。局部类不能有静态成员，局部类成员不能访问外围函数中定义的局部变量，它们可以使用外围函数中定义的类型名、静态变量和枚举成员。

[member operators new and delete \(成员操作符 new 和 delete\)](#)

Class member functions that override the default memory allocation performed by the global library `operator new` and `operator delete` functions. Both object (`new`) and array (`new[]`) forms of these functions may be defined. The member `new` and `delete` functions are implicitly declared as `static`. These operators allocate (deal-locate) memory. They are used automatically by `new` (`delete`) expressions, which handle object initialization and destruction.

类成员函数，覆盖由全局库函数 `operator new` 和 `operator delete` 执行的默认内存分配。可以定义这些函数的对象形式 (`new`) 和数组形式 (`new[]`)。成员 `new` 和 `delete` 函数隐式声明为 `static`，这些操作符分配 (释放) 内存，它们由 `new` (`delete`) 表达式自动使用，`new` (`delete`) 表达式处理对象初始化和撤销。

[nested class \(嵌套类\)](#)

Class defined inside another class. A nested class is defined inside its enclosing scope: Nested-class names must be unique within the class scope in which they are defined but can be reused in scopes outside the enclosing class. Access to the nested class outside the enclosing class requires use of the scope operator to specify the scope(s) in which the class is nested.

在其他类内部定义的类。嵌套类定义在其外围作用域内部：嵌套类名字必须在定义它们的类作用域 中唯一，但可以在外围类之外的作用域中重用。在外围类之外访问嵌套类需要使用作用域操作符指定包含嵌套类的作用域。

[nested type \(嵌套类型\)](#)

Synonym for nested class.

嵌套类的同义词。

[new expression \(new 表达式\)](#)

A `new` expression allocates and constructs an object of a specified type. A `new[]` expression allocates and constructs an array of objects. These expressions use the corresponding version of the library `operator new` functions to allocate raw memory in which the expression constructs an object or array of the specified type.

`new` 表达式分配和构造特定类型的对象。`new[]` 表达式分配和构造对象数组。这些表达式使用库函数 `operator new` 的对应版本分配原始内存，并在该内存中构造特定类型的对象或数组。

[operator delete](#)

A library function that frees untyped, unconstructed memory allocated by `operator new`. The library `operator delete[]` frees memory used to hold an array that was allocated by `operator new[]`.

释放由 `operator new` 分配的未类型化的未构造内存的库函数。库函数 `operator delete[]` 释放由 `operator new[]` 分配的用于保存数组的内存。

[operator new](#)

A library function that allocates untyped, unconstructed memory of a given size. The library function `operator new[]` allocates raw memory for arrays. These library functions provide a more primitive allocation mechanism than the library `allocator` class. Modern C++ programs should use the `allocator` classes rather than these library functions.

一个库函数，它分配给定大小未类型化的未构造内存。库函数 `operator new[]` 为数组分配原始内存。这些库函数提供了比库类 `allocator` 更基本的分配机制。现代 C++ 程序应使用 `allocator` 类而不是这些库函数。

[placement new expression \(定位 new 表达式\)](#)

The form of `new` that constructs its object in specified memory. It does no allocation; instead, it takes an argument that specifies where the object should be constructed. It is a lower-level analog of the behavior provided by the `construct` member of the `allocator` class.

在特定内存中构造对象的 `new` 的形式。它不进行分配，相反，它接受指定在何处构造对象的实参。它是对 `allocator` 类的 `construct` 成员所提供的行为的低级模拟。

[pointer to member \(成员指针\)](#)

Pointer that encapsulates the class type as well as the member type to which the pointer points. The definition of a pointer to member must specify the class name as well as the type of the member(s) to which the pointer may point:

封装类类型以及指针所指向的成员类型的指针。成员指针的定义必须指定类名以及指针可以指向的成员的类型:

```
TC::*pmem = &C::member;
```

This statement defines `pmem` as a pointer that can point to members of the class named `C` that have type `T` and initializes it to point to the member in `C` named `member`. When the pointer is dereferenced, it must be bound to an object of or pointer to type `C`:

这个语句将 `pmem` 定义为指针，它可以指向名为 `C` 的类的类型为 `T` 的成员，并将 `pmem` 初始化为 `C` 中名为 `member` 的成员。对该指针解引用的时候，它必须是绑定到类型 `C` 的对象或指向类型 `C` 的指针：

```
classobj.*pmem;
classptr->*pmem;
```

fetches `member` from the object `classobj` of the object pointed to by `classptr`.

从 `classptr` 所指对象的 `classobj` 对象获取 `member`。

portable (可移植的)

Term used to describe a program that can be moved to a new machine with relatively little effort.

一个用来描述用相对较少的努力就可以移到新机器的程序的术语。

run-time type identification (运行时类型识别)

Term used to describe the language and library facilities that allow the dynamic type of a reference or pointer to be obtained at run time. The RTTI operators, `typeid` and `dynamic_cast`, provide the dynamic type only for references or pointers to class types with virtual functions. When applied to other types, the type returned is the static type of the reference or pointer.

用来描述语言和库设施的术语，这种设施允许在运行时获得引用或指针的动态类型。RTTI 操作符 `typeid` 和 `dynamic_cast` 只为带虚函数的类类型提供动态类型，应用于其他类型的时候，返回引用或指针的静态类型。

typeid

Unary operator that takes an expression and returns a reference to an object of the library type named `type_info` that describes the type of the expression. When the expression is an object of a type that has virtual functions, then the dynamic type of the expression is returned. If the type is a reference, pointer, or other type that does not define virtual functions, then the type returned is the static type of the reference, pointer, or object.

一元操作符，接受一个表达式，并返回描述表达式类型的名为 `type_info` 的库类型的对象引用。如果表达式是具有虚函数的类型的对象，就返回表达式的动态类型；如果该类型是没有定义虚函数的引用、指针或其他类型，就返回引用、指针或对象的静态类型。

type_info

Library type that describes a type. The `type_info` class is inherently machine-dependent, but any library must define `type_info` with members listed in [Table 18.2](#) (p. 779). `type_info` objects may not be copied.

描述类型的库类型。`type_info` 类是固有机器相关的，但任何库都必须定义带有[表 18.2](#) 中所列出成员的 `type_info`。`type_info` 对象不能复制。

union (联合)

Classlike aggregate type that may define multiple data members, only one of which can have a value at any one point. Members of a union must be simple types: They can be a built-in or compound type or a class type that does not define a constructor, destructor, or the assignment operator. Unions may have member functions, including constructors and destructors. A union may not serve as a base class.

类形式的聚合类型，它可以定义多个数据成员，但在任意点只有一个成员可以具有值。联合的成员必须为简单类型：内置类型，复合类型，没有定义构造函数、析构函数或赋值操作符的类类型。联合可以有成员函数，包括构造函数和析构函数。联合不能作基类使用。

volatile

Type qualifier that signifies to the compiler that a variable might be changed outside the direct control of the program. It is a signal to the compiler that it may not perform certain optimizations.

类型限定符，告诉编译器可以在程序的直接控制之外改变一个变量。它是告诉编译器不能执行某些优化的信号。

A.1. Library Names and Headers

A.1. 标准库名字和头文件

Our programs mostly did not show the actual `#include` directives needed to compile the program. As a convenience to our readers, [Table A.1](#) lists the library names our programs used and the header in which they may be found.

本书的示例程序大多没有给出编译程序所需要的actual `#include` 指示。为了方便读者阅读,[表 A.1](#) 列出了示例程序使用过的标准库名字以及包含标准库名字的头文件。

Table A.1. Standard Library Names and Headers

表 A.1. 标准库名字和头文件

Name	Header	Name	Header
名字	头文件	名字	头文件
abort	<cstdlib>	ios_base	<iostream>
accumulate	<numeric>	isalpha	<cctype>
allocator	<memory>	islower	<cctype>
auto_ptr	<memory>	ispunct	<cctype>
back_inserter	<iterator>	isspace	<cctype>
bad_alloc	<new>	istream	<iostream>
bad_cast	<typeinfo>	istream_iterator	<iterator>
bind2nd	<functional>	istringstream	<sstream>
bitset	<bitset>	isupper	<cctype>
boolalpha	<iostream>	left	<iostream>
cerr	<iostream>	less_equal	<functional>
cin	<iostream>	list	<list>
copy	<algorithm>	logic_error	<stdexcept>
count	<algorithm>	lower_bound	<algorithm>
count_if	<algorithm>	make_pair	<utility>
cout	<iostream>	map	<map>
dec	<iostream>	max	<algorithm>
deque	<deque>	min	<algorithm>
endl	<iostream>	multimap	<map>
ends	<iostream>	multiset	<set>
equal_range	<algorithm>	negate	<functional>
exception	<exception>	noboolalpha	<iostream>
fill	<algorithm>	noshowbase	<iostream>
fill_n	<algorithm>	noshowpoint	<iostream>
find	<algorithm>	noskipws	<iostream>
find_end	<algorithm>	not1	<functional>
find_first_of	<algorithm>	nounitbuf	<iostream>
fixed	<iostream>	nouppercase	<iostream>
flush	<iostream>	nth_element	<algorithm>
for_each	<algorithm>	oct	<iostream>
front_inserter	<iterator>	of_stream	<fstream>
fstream	<fstream>	ostream	<iostream>

Section A.1. Library Names and Headers

getline	<string>	ostream_iterator	<iterator>
hex	<iostream>	ostringstream	<sstream>
ifstream	<fstream>	out_of_range	<stdexcept>
inner_product	<numeric>	pair	<utility>
inserter	<iterator>	partial_sort	<algorithm>
internal	<iostream>	plus	<functional>
priority_queue	<queue>	sqrt	<cmath>
ptrdiff_t	<cstddef>	stable_sort	<algorithm>
queue	<queue>	stack	<stack>
range_error	<stdexcept>	strcmp	<cstring>
replace	<algorithm>	strcpy	<cstring>
replace_copy	<algorithm>	string	<string>
reverse_iterator	<iterator>	stringstream	<sstream>
right	<iostream>	strlen	<cstring>
runtime_error	<stdexcept>	strncpy	<cstring>
scientific	<iostream>	terminate	<exception>
set	<set>	tolower	<cctype>
set_difference	<algorithm>	toupper	<cctype>
set_intersection	<algorithm>	type_info	<typeinfo>
set_union	<algorithm>	unexpected	<exception>
setfill	<iomanip>	uninitialized_copy	<memory>
setprecision	<iomanip>	unitbuf	<iostream>
setw	<iomanip>	unique	<algorithm>
showbase	<iostream>	unique_copy	<algorithm>
showpoint	<iostream>	upper_bound	<algorithm>
size_t	<cstddef>	uppercase	<iostream>
skipws	<iostream>	vector	<vector>
sort	<algorithm>		

Team LiB

◀ PREVIOUS NEXT ▶

A.2. A Brief Tour of the Algorithms

A.2. 算法简介

[Chapter 11](#) introduced the generic algorithms and outlined their underlying architecture. The library defines more than 100 algorithms. Learning to use them requires understanding their structure rather than memorizing the details of each algorithm. In this section we describe each of the algorithms. In it, we organize the algorithms by the type of action the algorithm performs.

[第十一章](#)介绍一般算法并列出了它们的底层体系结构。标准库定义了 100 多个算法，学习如何使用它们需要理解它们的结构，而不是记住每个算法的细节。本节描述每个算法，其中，按算法执行行为的类型组织这些算法。

A.2.1. Algorithms to Find an Object

A.2.1.1. 查找对象的算法

The `find` and `count` algorithms search the input range for a specific value. `find` returns an iterator to an element; `count` returns the number of matching elements.

`find` 和 `count` 算法在输入范围中查找指定值。`find` 返回元素的迭代器，`count` 返回匹配元素的数目。

Simple Find Algorithms

简单查找算法

These algorithms require input iterators. The `find` and `count` algorithms look for specific elements. The `find` algorithms return an iterator referring to the first matching element. The `count` algorithms return a count of how many times the element occurs in the input sequence.

这些算法要求输入迭代器。`find` 和 `count` 算法查找特定元素，`find` 算法返回引用第一个匹配元素的迭代器，`count` 算法返回元素在输入序列中出现次数的计数。

```
find(beg, end, val)
count(beg, end, val)
```

Looks for element(s) in input range equal to `val`. Uses the equality (`==`) operator of the underlying type. `find` returns an iterator to the first matching element or `end` if no such element exists. `count` returns a count of how many times `val` occurs.

在输入范围中查找等于 `val` 的元素，使用基础类型的相等 (`==`) 操作符。`find` 返回第一个匹配元素的迭代器，如果不存在在匹配元素就返回 `end`。`count` 返回 `val` 出现次数的计数。

```
find_if(beg, end, unaryPred)
count_if(beg, end, unaryPred)
```

Looks for element(s) in input range for which `unaryPred` is true. The predicate must take a single parameter of the `value_type` of the input range and return a type that can be used as a condition.

在 `unaryPred` 为真的输入范围中查找。谓词必须接受一个形参，形参类型为输入范围的 `value_type`，并且返回可以用作条件的类型。

`find_if` returns an iterator to first element for which `unaryPred` is true, or `end` if no such element exists. `count_if` applies `unaryPred` to each element and returns the number of elements for which `unaryPred` was true.

`find_if` 返回第一个使 `unaryPred` 为真的元素的迭代器，如果不存在这样的元素就返回 `end`。`count_if` 对每个元素应用 `unaryPred`。并返回使 `unaryPred` 为真的元素的数目。

Algorithms to Find One of Many Values

查找许多值中的一个的算法

These algorithms require two pairs of forward iterators. They search for the first (or last) element in the first range that is equal to any element in the second range. The types of `beg1` and `end1` must match exactly, as must the types of `beg2` and `end2`.

这些算法要求两对前向迭代器。它们在第一个范围中查找与第二个范围中任意元素相等的第一个（或最后一个）元素。`beg1` 和 `end1` 的类型必须完全匹配，`beg2` 和 `end2` 的类

Section A.2. A Brief Tour of the Algorithms

型也必须完全匹配。

There is no requirement that the types of `beg1` and `beg2` match exactly. However, it must be possible to compare the element types of the two sequences. So, for example, if the first sequence is a `list<string>`, then the second could be a `vector<char*>`.

不要求 `beg1` 和 `beg2` 的类型完全匹配，但是，必须有可能对这两个序列的元素类型进行比较。例如，如果第一个序列是 `list<string>`，则第二个可以是 `vector<char*>`。

Each algorithm is overloaded. By default, elements are tested using the `==` operator for the element type. Alternatively, we can specify a predicate that takes two parameters and returns a `bool` indicating whether the test between these two elements succeeds or fails.

每个算法都是重载的。默认情况下，使用元素类型的 `==` 操作符测试元素，或者，可以指定一个谓词，该谓词接受两个形参，并返回表示这两个元素间的测试成功或失败的 `bool` 值。

`find_first_of(beg1, end1, beg2, end2)`

Returns an iterator to the first occurrence in the first range of any element from the second range. Returns `end1` if no match found.

返回第二个范围的任意元素在第一个范围的首次出现的迭代器，如果找不到匹配就返回 `end1`。

`find_first_of(beg1, end1, beg2, end2, binaryPred)`

Uses `binaryPred` to compare elements from each sequence. Returns an iterator to the first element in the first range for which the `binaryPred` is true when applied to that element and an element from the second sequence. Returns `end1` if no such element exists.

使用 `binaryPred` 比较来自两个序列的元素，返回第一个范围内第一个这种元素的迭代器：当对该元素和来自第二个范围的一个元素应用 `binaryPred` 的时候，`binaryPred` 为真。如果不存在这样的元素，就返回 `end1`。

`find_end(beg1, end1, beg2, end2)`
`find_end(beg1, end1, beg2, end2, binaryPred)`

Operates like `find_first_of`, except that it searches for the last occurrence of any element from the second sequence.

与 `find_first_of` 想像的操作符，只不过它查找来自第二个序列的任意元素的最后一次出现。

As an example, if the first sequence is 0,1,1,2,2,4,0,1 and the second sequence is 1,3,5,7,9, then `find_end` would return an iterator denoting the last element in the input range, and `find_first_of` would return an iterator to the second element in this example, it returns the first 1 in the input sequence.

作为例子，如果第一个序列是 0, 1, 1, 2, 2, 4, 0, 1 而第二个序列是 1, 3, 5, 7, 9，则 `find_end` 返回表示输入范围内最后一个元素的迭代器，而 `find_first_of` 将返回第二个元素的迭代器——本例中，它返回输入序列中的第一个 1。

Algorithms to Find a Subsequence

查找子序列的算法

These algorithms require forward iterators. They look for a subsequence rather than a single element. If the subsequence is found, an iterator is returned to the first element in the subsequence. If no subsequence is found, the `end` iterator from the input range is returned.

这些算法要求前向迭代器。它们查找子序列而不是单个元素。如果找到了子序列，就返回子序列中第一个元素的迭代器；如果找不到子序列，就返回输入范围的 `end` 迭代器。

Each function is overloaded. By default, the equality (`==`) operator is used to compare elements; the second version allows the programmer to supply a predicate to test instead.

每个函数都是重载的。默认情况下，使用相等操作符 (`==`) 比较元素；第二个版本允许程序员提供一个谓词代替 (`==`) 进行测试。

`adjacent_find(beg, end)`
`adjacent_find(beg, end, binaryPred)`

Returns an iterator to the first adjacent pair of duplicate elements. Returns `end` if there are no adjacent duplicate elements. In the first case, duplicates are found by using `==`. In the second, duplicates are those for which the `binaryPred` is true.

返回重复元素的第一个相邻对。如果没有相邻的重复元素，就返回 `end`。在第一种情况下，使用 `==` 找到重复元素，第二种情况下，重复元素是使 `binaryPred` 为真的那些元素。

`search(beg1, end1, beg2, end2)`
`search(beg1, end1, beg2, end2, binaryPred)`

Returns an iterator to the first position in the input range at which the second range occurs as a subsequence. Returns `end1` if the subsequence is not found. The types of `beg1` and `beg2` may differ but must be compatible: It must be possible to compare elements in the two sequences.

返回输入范围内第二个范围作为子序列出现的第一个位置。如果找不到子序列，就返回 `end1`。`beg1` 和 `beg2` 的类型可以不同，但必须是兼容的：必须能够比较两个序列中的元素。

`search_n(beg, end, count, val)`
`search_n(beg, end, count, val, binaryPred)`

Section A.2. A Brief Tour of the Algorithms

Returns an iterator to the beginning of a subsequence of `count` equal elements. Returns `end` if no such subsequence exists. The first version looks for `count` occurrences of the given `value`; the second version `count` occurrences for which the `binaryPred` is true.

返回 `count` 个相等元素的子串的开关迭代器。如果不存在这样的子串，就返回 `end`。第一个版本查找给定 `val` 的 `count` 次出现，第二个版本查找使 `binaryPred` 为真的 `count` 次出现。

A.2.2. Other Read-Only Algorithms

A.2.2. 其他只读算法

These algorithms require input iterators for their first two arguments. The `equal` and `mismatch` algorithms also take an additional input iterator that denotes a second range. There must be at least as many elements in the second sequence as there are in the first. If there are more elements in the second, they are ignored. If there are fewer, it is an error and results in undefined run-time behavior.

这些算法要求用于前两个实参的输入迭代器。`equal` 和 `mismatch` 算法还接受一个附加输入迭代器，该迭代器表示第二个范围。第二个序列中的元素至少与第一个序列一样多，如果第二个序列元素较多，就忽略多余元素；如果第二个序列元素较少，就会出错并导致未定义的运行时行为。

As usual, the types of the iterators denoting the input range must match exactly. The type of `beg2` must be compatible with the type of `beg1`. That is, it must be possible to compare elements in both sequences.

照常，表示输入范围的迭代器的类型必须完全匹配。`beg2` 的类型必须与 `beg1` 的类型兼容，即必须能够比较两个序列中的元素。

The `equal` and `mismatch` functions are overloaded: One version uses the element equality operator (`==`) to test pairs of elements; the other uses a predicate.

`equal` 和 `mismatch` 函数是重载的：一个版本使用元素相等操作符 (`==`) 测试元素对，另一个使用谓词。

`for_each(beg, end, f)`

Applies the function (or function object (Section 14.8, p. 530)) `f` to each element in its input range. The return value, if any, from `f` is ignored. The iterators are input iterators, so the elements may not be written by `f`. Typically, `for_each` is used with a function that has side effects. For example, `f` might print the values in the range.

对输入范围中的每个元素应用函数（或函数对象（第 14.8 节））`f`。如果 `f` 有返回值，就忽略该返回值。迭代器是输入迭代器，所以 `f` 不能写元素。通常，用有副作用的函数使用 `for_each`。例如，`f` 可以显示范围中的值。

`mismatch(beg1, end1, beg2)`
`mismatch(beg1, end1, beg2, binaryPred)`

Compares the elements in two sequences. Returns a pair of iterators denoting the first elements that do not match. If all the elements match, then the `pair` returned is `end1`, and an iterator into `beg2` offset by the size of the first sequence.

比较两个序列中的元素，返回一对表示第一个不匹配元素的迭代器。如果所有元素都匹配，则返回的 `pair` 是 `end1`，以及 `beg2` 中偏移量为第一个序列长度的迭代器。

`equal(beg1, end1, beg2)`
`equal(beg1, end1, beg2, binaryPred)`

Determines whether two sequences are equal. Returns `true` if each element in the input range equals the corresponding element in the sequence that begins at `beg2`.

确定两个序列是否相等。如果输入范围中的每个元素都与从 `beg2` 开始的序列中的对应元素相等，就返回 `true`。

For example, given the sequences `meet` and `meat`, a call to `mismatch` would return a `pair` containing iterators referring to the second `e` in the first sequence and to the element `a` in the second sequence. If, instead, the second sequence were `meeting`, and we called `equal`, then the `pair` returned would be `end1` and an iterator denoting the element `i` in the second range.

例如，给定序列 `meet` 和 `meat`，对 `mismatch` 的调用将返回一个 `pair` 对象，其中包含指向第一个序列中第二个 `e` 的迭代器，以及指向第二个序列中元素 `a` 的迭代器。如果，第二个序列是 `meeting`，并调用 `equal`，则返回的将是 `end1` 和表示第二个范围中元素 `i` 的迭代器。

A.2.3. Binary-Search Algorithms

A.2.3. 二分查找算法

Although these algorithms may be used with forward iterators, they offer specialized versions that are much faster when used with random-access iterators.

虽然可以与前向迭代器一起使用这些算法，它们还是提供了随机访问迭代器一起使用的特殊版本，它们的速度更快。

These algorithms perform a binary search, which means that the input sequence must be sorted. These algorithms behave similarly to the associative container members of the same name (Section 10.5.2, p. 377). The `equal_range`, `lower_bound`, and `upper_bound` algorithms return an iterator that refers to the positions in the container at which the given element could be inserted while still preserving the container's ordering. If the element is larger than any other in the container, then the iterator that is returned might be the off-the-end iterator.

Section A.2. A Brief Tour of the Algorithms

这些算法执行二分查找，这意味着输入序列必须是已排列的。这些算法的表现类似于同名的关联容器成员（第 10.5.2 节）。`equal_range`、`lower_bound` 和 `upper_bound` 算法返回一个迭代器，该迭代器指向容器中的位置，可以将给定元素插入到这个位置而仍然保持容器的排序。如果元素比容器中任意其他元素都大，则返回的迭代器会是超出末端迭代器。

Each algorithm provides two versions: The first uses the element type's less-than operator (`<`) to test elements; the second uses the specified comparison.

每个算法提供两个版本：第一个使用元素类型的小于操作符（`<`）测试元素，第二个使用指定的比较关系。

```
lower_bound(beg, end, val)
lower_bound(beg, end, val, comp)
```

Returns an iterator to the first position in which `val` can be inserted while preserving the ordering.

返回第一个这种位置的迭代器：可以将 `val` 插入到该位置而仍然保持顺序。

```
upper_bound(beg, end, val)
upper_bound(beg, end, val, comp)
```

Returns an iterator to the last position in which `val` can be inserted while preserving the ordering.

返回最后一个这种位置的迭代器：可以将 `val` 插入到该位置而仍然保持顺序。

```
equal_range(beg, end, val)
equal_range(beg, end, val, comp)
```

Returns an iterator pair indicating the subrange in which `val` could be inserted while preserving the ordering.

返回一个表示子范围的迭代器对，可以将 `val` 插入到该子范围而仍然保持顺序。

```
binary_search(beg, end, val)
binary_search(beg, end, val, comp)
```

Returns a `bool` indicating whether the sequence contains an element that is equal to `val`. Two values `x` and `y` are considered equal if `x < y` and `y <x` both yield false.

返回一个 `bool` 值，表示序列是否包含与 `val` 相等的元素。如果 `x < y` 和 `y <x` 都获得假值，就认为两个值 `x` 和 `y` 相等。

A.2.4. Algorithms that Write Container Elements

A.2.4. 写容器元素的算法

Many algorithms write container elements. These algorithms can be distinguished both by the kinds of iterators on which they operate and by whether they write elements in the input range or write to a specified destination.

许多算法写容器元素。可以根据所操作的迭代器种类，以及是写输入范围的元素还是写到特定目的地，来区分这些算法。

The simplest algorithms read elements in sequence, requiring only input iterators. Those that write back to the input sequence require forward iterators. Some read the sequence backward, thus requiring bidirectional iterators. Algorithms that write to a separate destination, as usual, assume the destination is large enough to hold the output.

最简单的算法读序列中的元素，只要求输入迭代器。那些写回输入序列的算法要求前向迭代器。一些算法反向读取序列，所以要求双向迭代器。写至单独目的地的算法，照常假定目的地足以保存输入。

Algorithms that Write but do Not Read Elements

只写元素不读元素的算法

These algorithms require an output iterator that denotes a destination. They take a second argument that specifies a count and write that number of elements to the destination.

这些算法要求表示目的地的输出迭代器。它们接受指定数量的第二个实参并将该数目的元素写到目的地。

```
fill_n(dest, cnt, val)
generate_n(dest, cnt, Gen)
```

Write `cnt` values to `dest`. The `fill_n` function writes `cnt` copies of the value `val`; `generate_n` evaluates the generator `Gen()` `cnt` times. A generator is a function (or function object (Section 14.8, p. 530)) that is expected to produce a different return value each time it is called.

将 `cnt` 个值写到 `dest`。`fill_n` 函数写 `val` 值的 `cnt` 个副本，`generate_n` 对发生器 `Gen()` 进行 `cnt` 次计算。发生器是一个函数（或函数对象（第 14.8 节）），每次调用它都期待产生一个不同的返回值。

Algorithms that Write Elements Using Input Iterators

使用输入迭代器写元素的算法

Each of these operations reads an input sequence and writes to an output sequence denoted by `dest`. They require `dest` to be an output iterator, and the iterators denoting the input range must be input iterators. The caller is responsible for ensuring that `dest` can hold as many elements as necessary given the input sequence. These algorithms return `dest` incremented to denote one past the last element written.

这些操作每一个读一个输入序列，并写到由 `dest` 表示的输出序列。它们要求 `dest` 是一个输出迭代器，而表示输入范围的迭代器必须是输入迭代器。调用者负责保证 `dest` 可以保存给定输入序列所需数量的元素。这些算法返回 `dest`, `dest` 增量至指向所写最后元素的下一位置。

`copy(beg, end, dest)`

Copies the input range to the sequence beginning at iterator `dest`.

将输入范围复制到从迭代器 `dest` 开始的序列。

`transform(beg, end, dest, unaryOp)`
`transform(beg, end, beg2, dest, binaryOp)`

Applies the specified operation to each element in the input range, writing the result to `dest`. The first version applies a unary operation to each element in the input range. The second applies a binary operation to pairs of elements. It takes the first argument to the binary operation from the sequence denoted by `beg` and `end` and takes the second argument from the sequence beginning at `beg2`. The programmer must ensure that the sequence beginning at `beg2` has at least as many elements as are in the first sequence.

对输入范围内每个元素应用指定操作。将结果写到 `dest`。第一个版本对输入范围内每个元素应用一元操作。第二个版本对元素对应用二元操作，它从由 `beg` 和 `end` 表示的序列接受二元操作的第一个实参，从开始于 `beg2` 的第二个序列接受第二个实参。程序员必须保证开始于 `beg2` 的序列具有至少与第一个序列一样多的元素。

`replace_copy(beg, end, dest, old_val, new_val)`
`replace_copy_if(beg, end, dest, unaryPred, new_val)`

Copies each element to `dest`, replacing specified elements by the `new_val`. The first version replaces those elements that are `==` to `old_val`. The second version replaces those elements for which `unaryPred` is true.

将每个元素复制到 `dest`，用 `new_val` 替代指定元素。第一个版本代替那些 `==old_val` 的元素，第二个版本代替那些使 `unaryPred` 为真的元素。

`merge(beg1, end1, beg2, end2, dest)`
`merge(beg1, end1, beg2, end2, dest, comp)`

Both input sequences must be sorted. Writes a merged sequence to `dest`. The first version compares elements using the `<` operator; the second version uses the given comparison.

两个输入序列都必须是已排序的。将合并后的序列写至 `dest`。第一个版本 `<` 操作符比较元素，第二个版本使用给定的比较关系。

Algorithms that Write Elements Using Forward Iterators

使用前向迭代器写元素的算法

These algorithms require forward iterators because they write elements in their input sequence.

这些算法要求前向迭代器，因为它们修改输入序列中的元素。

`swap(elem1, elem2)`
`iter_swap(iterator1, iterator2)`

Parameters to these functions are references, so the arguments must be writable. Swaps the specified element or elements denoted by the given iterators.

这些函数的形参是引用，所以实参必须是可写的。交换指定元素或由给定迭代器表示的元素。

`swap_ranges(beg1, end1, beg2)`

Swaps the elements in the input range with those in the second sequence beginning at `beg2`. The ranges must not overlap. The programmer must ensure that the sequence starting at `beg2` is at least as large as the input sequence. Returns `beg2` incremented to denote the element just after the last one swapped.

用开始于 `beg2` 的第二个序列中的元素交换输入范围中的元素。范围必须不重叠。程序员必须保证开始于 `beg2` 的序列至少与输入序列一样大。返回 `beg2`, `beg2` 增量到指向被交换的最后一个元素之后的元素。

`fill(beg, end, val)`
`generate(beg, end, Gen)`

Assigns a new value to each element in the input sequence. `fill` assigns the value `val`; `generate` executes `Gen()` to create new values.

将新值赋给输入序列中的每个元素。`fill` 赋 `val` 值，`generate` 执行 `Gen()` 来创建新值。

`replace(beg, end, old_val, new_val)`

Section A.2. A Brief Tour of the Algorithms

```
replace_if(beg, end, unaryPred, new_val)
```

Replace each matching element by `new_val`. The first version uses `==` to compare elements with `old_val`; the second version executes `unaryPred` on each element, replacing those for which `unaryPred` is true.

用 `new_val` 替代每个匹配元素。第一个版本使用 `==` 将元素和 `old_val` 比较，第二个版本对每个元素执行 `unaryPred`，代替使 `unaryPred` 为真的那些元素。

Algorithms that Write Elements Using Bidirectional Iterators

使用双向迭代器写元素的算法

These algorithms require the ability to go backward in the sequence, and so they require bidirectional iterators.

这些算法要求在序列中往回走的能力，所以它们要求双向迭代器。

```
copy_backward(beg, end, dest)
```

Copies elements in reverse order to the output iterator `dest`. Returns `dest` incremented to denote one past the last element copied.

按逆序将元素复制到输出迭代器 `dest`。返回 `dest`, `dest` 增量至指向被复制的最后一个元素的下一位置。

```
inplace_merge(beg, mid, end)
inplace_merge(beg, mid, end, comp)
```

Merges two adjacent subsequences from the same sequence into a single, ordered sequence: The subsequences from `beg` to `mid` and from `mid` to `end` are merged into `beg` to `end`. First version uses `<` to compare elements; second version uses a specified comparison. Returns `void`.

将同一序列中的两个相邻子序列合并为一个有序序列：将从 `beg` 到 `mid` 和从 `mid` 到 `end` 的子序列合并为从 `beg` 到 `end` 的序列。第一个版本使用 `<` 比较元素，第二个版本使用指定的比较关系。返回 `void`。

A.2.5. Partitioning and Sorting Algorithms

A.2.5. 划分与排序算法

The sorting and partitioning algorithms provide various strategies for ordering the elements of a container.

排序和划分算法为容器元素排序提供不同的策略。

A `partition` divides elements in the input range into two groups. The first group consists of those elements that satisfy the specified predicate; the second, those that do not. For example, we can partition elements in a container based on whether the elements are odd, or on whether a word begins with a capital letter, and so forth.

`partition` 将输入范围中的元素划分为两组，第一组由满足给定谓词的元素构成，第二组由不满足谓词的元素构成。例如，可以根据元素是否为奇数划分容器中的元素，或者，根据单词是否以大写字母开头，诸如此类。

Each of the sorting and partitioning algorithms provides stable and unstable versions. A stable algorithm maintains the relative order of equal elements. For example, given the sequence

每个排序和划分算法都提供稳定和不稳定版本，稳定算法维持相等元素的相对次序。例如，给定序列

```
{ "pshew", "Honey", "tigger", "Pooh" }
```

a stable partition based on whether a word begins with a capital letter generates the sequence in which the relative order of the two word categories is maintained:

基于单词是否以大写字母开头的稳定算法，产生维持两个单词类的相对次序的序列：

```
{ "Honey", "Pooh", "pshew", "tigger" }
```

The stable algorithms do more work and so may run more slowly and use more memory than the unstable counterparts.

稳定算法完成更多工作，因此相比于不稳定算法，可能运行慢且使用更多内存。

Partitioning Algorithms

划分算法

These algorithms require bidirectional iterators.

Section A.2. A Brief Tour of the Algorithms

这些算法要求双向迭代器。

```
stable_partition(beg, end, unaryPred)
partition(beg, end, unaryPred)
```

Uses `unaryPred` to partition the input sequence. Elements for which `unaryPred` is true are put at the beginning of the sequence; those for which the predicate is false are at the end. Returns an iterator just past the last element for which `unaryPred` is true.

使用 `unaryPred` 划分输入序列。使 `unaryPred` 为真的元素放在序列开头，使 `unaryPred` 为假的元素放在序列末尾。返回一个迭代器，该迭代器指向使 `unaryPred` 为真的最后元素的下一个位置。

Sorting Algorithms

排序算法

These algorithms require random-access iterators. Each of the sorting algorithms provides two overloaded versions. One version uses element operator `<` to compare elements; the other takes an extra parameter that specifies the comparison. These algorithms require random-access iterators. With one exception, these algorithms return `void`; `partial_sort_copy` returns an iterator into the destination.

这些算法要求随机访问迭代器。每个排序算法都提供两个重载版本，一个版本使用元素操作符 `<` 比较元素，另一个版本接受一个指定比较关系的额外参数。这些算法返回 `void`，除了一个例外，`partial_sort_copy` 返回目的地迭代器。

The `partial_sort` and `nth_element` algorithms do only part of the job of sorting the sequence. They are often used to solve problems that might otherwise be handled by sorting the entire sequence. Because these operations do less work, they typically are faster than sorting the entire input range.

`partial_sort` 和 `nth_element` 算法只完成序列排序的部分工作，经常用它们解决通过对整个序列排序来处理的问题。因为这些操作做的工作较少，所以它们一般比排序整个输入范围要快一些。

```
sort(beg, end)
stable_sort(beg, end)
sort(beg, end, comp)
stable_sort(beg, end, comp)
```

Sorts the entire range.

对整个范围进行排序。

```
partial_sort(beg, mid, end)
partial_sort(beg, mid, end, comp)
```

Sorts a number of elements equal to `mid - beg`. That is, if `mid - beg` is equal to 42, then this function puts the lowest-valued elements in sorted order in the first 42 positions in the sequence. After `partial_sort` completes, the elements in the range from `beg` up to but not including `mid` are sorted. No element in the sorted range is larger than any element in the range after `mid`. The order among the unsorted elements is unspecified.

对 `mid - beg` 个元素进行排序，也就是说，如果 `mid - beg` 等于 42，则该函数将有序次序中的最小值元素放在序列中前 42 个位置。`partial_sort` 完成之后，从 `beg` 到 `mid`（但不包括 `mid`）范围内的元素是有序的。已排序范围内没有元素大于 `mid` 之后的元素。未排序元素之间的次序是未指定的。

As an example, we might have a collection of race scores and want to know what the first-, second- and third-place scores are but don't care about the order of the other times. We might sort such a sequence as follows:

例如，有一个赛跑成绩的集合，我们想要知道前三名的成绩但并不关心其他名次的次序，可以这样对这个序列进行排序：

```
partial_sort(scores.begin(),
            scores.begin() + 3, scores.end());
partial_sort_copy(beg, end, destBeg, destEnd)
partial_sort_copy(beg, end, destBeg, destEnd, comp)
```

Sorts elements from the input range and puts as much of the sorted sequence as fits into the sequence denoted by the iterators `destBeg` and `destEnd`. If the destination range is the same size or has more elements than the input range, then the entire input range is sorted and stored starting at `destBeg`. If the destination size is smaller, then only as many sorted elements as will fit are copied.

对输入序列中的元素进行排序，将已排序序列中适当数目的元素放入由迭代器 `destBeg` 和 `destEnd` 表示的序列。如果目的地范围与输入范围一样大，或者比输入范围大，则将整个输入范围排序且有序序列从 `destBeg` 开始。如果目的地较小，则只复制适当数目的有序元素。

Returns an iterator into the destination that refers just after the last element that was sorted. The returned iterator will be `destEnd` if that destination sequence is smaller or equal in size to the input range.

返回目的地中的迭代器，指向已排序的最后一个元素之后。如果目的地序列比输入范围小或者与输入范围大小相等，返回的迭代器将是 `destEnd`。

```
nth_element(beg, nth, end)
nth_element(beg, nth, end, comp)
```

The argument `nth` must be an iterator positioned on an element in the input sequence. After `nth_element`, the element denoted by that iterator has the value that would be there if the entire sequence were sorted. The elements in the container are also partitioned around `nth`: Those before `nth` are all smaller than or equal to the value denoted by `nth`, and the ones after it are greater than or equal to it. We might use `nth_element` to find the value closest to the median:

Section A.2. A Brief Tour of the Algorithms

实参 `nth` 必须是一个迭代器，定位输入序列中的一个元素。运行 `nth_element` 之后，该迭代器表示的元素的值就是：如果整个序列是已排序的，这个位置上应放置的值。容器中的元素也围绕 `nth` 划分：`nth` 之前的元素都小于或等于 `nth` 所表示的值，`nth` 之后的元素都大于或等于它。可以使用 `nth_element` 查找与中值最接近的值：

```
nth_element(scores.begin(), scores.begin() +  
           scores.size()/2, scores.end());
```

A.2.6. General Reordering Operations

A.2.6. 通用重新排序操作

Several algorithms reorder the elements in a specified way. The first two, `remove` and `unique`, reorder the container so that the elements in the first part of the sequence meet some criteria. They return an iterator marking the end of this subsequence. Others, such as `reverse`, `rotate`, and `random_shuffle` rearrange the entire sequence.

有几个算法用指定方法对元素进行重新排序。最前面的两个 `remove` 和 `unique` 对容器重新排序，以便序列中的第一部分满足一些标准，它们返回标志这个子序列的末尾的迭代器。其他算法，如 `reverse`、`rotate` 和 `random_shuffle`，重新安排整个序列。

These algorithms operate "in place;" they rearrange the elements in the input sequence itself. Three of the reordering algorithms offer "copying" versions. These algorithms, `remove_copy`, `rotate_copy`, and `unique_copy`, write the reordered sequence to a destination rather than rearranging elements directly.

这些算法“就地”操作，它们在输入序列本身中重新安排元素。三个重新排序算法提供“复制”版本。算法 `remove_copy`、`rotate_copy` 和 `unique_copy`，将重新排序之后的序列写至目的地，而不是直接重新安排元素。

Reordering Algorithms Using Forward Iterators

使用前向迭代器的重新排序算法

These algorithms reorder the input sequence. They require that the iterators be at least forward iterators.

这些算法对输入序列进行重新排序。它们要求迭代器至少是前向迭代器。

```
remove(beg, end, val)  
remove_if(beg, end, unaryPred)
```

"Removes" elements from the sequence by overwriting them with elements that are to be kept. The removed elements are those that are == to `val` or for which `unaryPred` is true. Returns an iterator just past the last element that was not removed.

通过用要保存的元素覆盖元素而从序列中“移去”元素。被移支的元素是 ==`val` 或使 `unaryPred` 为真的那些元素。返回一个迭代器，该迭代器指向未移去的最后一个元素的下一位置。

For example, if the input sequence is `hello world` and `val` is `o`, then a call to `remove` will overwrite the two elements that are the letter '`o`' by shifting the sequence to the left twice. The new sequence will be `hell wrldld`. The returned iterator will denote the element after the first `d`.

例如，如果输入序列是 `hello world` 而 `val` 是 `o`，则 `remove` 调用将序列左移两次覆盖两个元素，即字母 '`o`'。新序列将是 `hell wrldld`，返回的迭代器将指向第一个 `d` 之后的元素。

```
unique(beg, end)  
unique(beg, end, binaryPred)
```

"Removes" all but the first of each consecutive group of matching elements. Returns an iterator just past the last unique element. First version uses == to determine whether two elements are the same; second version uses the predicate to test adjacent elements.

“移去”匹配元素的每个连续组，除了第一个之外。返回一个迭代器，该迭代器指向最后一个单一元素的下一位置。第一个版本使用 == 确定两个元素是否相同，第二个版本使用谓词测试相邻元素。

For example, if the input sequence is `boohiss`, then after the call to `unique`, the first sequence will contain `bohisss`. The iterator returned will point to the element after the first `s`. The value of the remaining two elements in the sequence is unspecified.

例如，如果输入序列是 `boohiss`，则调用 `unique` 之后，第一个序列将包含 `bohisss`。返回的迭代器指向第一个 `s` 之后的元素，序列中剩余的两个元素的值是未指定的。

```
rotate(beg, mid, end)
```

Rotates the elements around the element denoted by `mid`. The element at `mid` becomes the first element; those from `mid + 1` through `end` come next, followed by the range from `beg` to `mid`. Returns `void`.

围绕由 `mid` 表示的元素旋转元素。`mid` 处的元素成为第一个元素，从 `mid + 1` 到 `end` 的元素其次，后面是从 `beg` 到 `mid` 的范围。返回 `void`。

For example, given the input sequence `hissboo`, if `mid` denotes the character `b`, then `rotate` would reorder the sequence as `boohiss`.

例如，给定输入序列 `hissboo`，如果 `mid` 表示字符 `b`，则旋转将序列重新排序为 `boohiss`。

Reordering Algorithms Using Bidirectional Iterators

使用双向迭代器的重新排序算法

Because these algorithms process the input sequence backward, they require bidirectional iterators.

因为这些算法向后处理输入序列，所以它们要求双向迭代器。

```
reverse(beg, end)
reverse_copy(beg, end, dest)
```

Reverses the elements in the sequence. `reverse` operates in place; it writes the rearranged elements back into the input sequence. `reverse_copy` copies the elements in reverse order to the output iterator `dest`. As usual, the programmer must ensure that `dest` can be used safely. `reverse` returns `void`; `reverse_copy` returns an iterator just past the last element copied into the destination.

颠倒序列中的元素。`reverse` 就地操作，它将重新安排的元素写回输入序列。`reverse_copy` 将元素按逆序复制到输出迭代器 `dest`。照常，程序员必须保证可以安全块使用 `dest`。`reverse` 返回 `void`, `reverse_copy` 返回一个迭代器，该迭代器指向复制到目的地的最后一个元素的下一位置。

Reordering Algorithms Writing to Output Iterators

写至输出迭代器的重新排序算法

These algorithms require forward iterators for the input sequence and an output iterator for the destination.

这些算法要求输入序列的前向迭代器以及目的地的输出迭代器。

Each of the preceding general reordering algorithms has an `_copy` version. These `_copy` versions perform the same reordering but write the reordered elements to a specified destination sequence rather than changing the input sequence. Except for `rotate_copy`, which requires forward iterators, the input range is specified by input iterators. The `dest` iterator must be an output iterator and, as usual, the programmer must guarantee that the destination can be written safely. The algorithms return the `dest` iterator incremented to denote one past the last element copied.

前面的每个通用重新排序算法都有一个 `_copy` 版本，这些 `_copy` 版本执行相同的重新排序，但是将重新排序之后的元素写至指定目的地序列，而不是改变输入序列。除 `rotate_copy`（它要求前向迭代器）之外，其他的都由迭代器指定输入范围。`dest` 迭代器必须是输出迭代器，而且，程序员也必须保证可以安全地写目的地。这些算法返回 `dest` 迭代器，`dest` 迭代器增量至指向被复制的最后元素的下一位置。

```
remove_copy(beg, end, dest, val)
remove_copy_if(beg, end, dest, unaryPred)
```

Copies elements except those matching `val` or for which `unaryPred` return true into `dest`.

除了与 `val` 匹配或使 `unaryPred` 返回真的元素之外，其他元素都复制到 `dest`。

```
unique_copy(beg, end, dest)
unique_copy(beg, end, dest, binaryPred)
```

Copies unique elements to `dest`.

将唯一元素复制到 `dest`。

```
rotate_copy(beg, mid, end, dest)
```

Like `rotate` except that it leaves its input sequence unchanged and writes the rotated sequence to `dest`. Returns `void`.

除了保持输入序列不变并将旋转后的序列写至 `dest` 之外，与 `rotated` 很像。返回 `void`。

Reordering Algorithms Using Random-Access Iterators

使用随机访问迭代器的重新排序算法

Because these algorithms rearrange the elements in a random order, they require random-access iterators.

因为这些算法按随机次序重新安排元素，所以它们要求随机访问迭代器。

```
random_shuffle(beg, end)
random_shuffle(beg, end, rand)
```

Shuffles the elements in the input sequence. The second version takes a random-number generator. That function must take and return a value of the iterator's `difference_type`. Both versions return `void`.

打乱输入序列中的元素。第二个版本接受随机数发生器，该函数必须接受并返回迭代器的 `difference_type` 值。两个版本都返回 `void`。

A.2.7. Permutation Algorithms

A.2.7. 排列算法

Consider the following sequence of three characters: `abc`. There are six possible permutations on this sequence: `abc`, `acb`, `bac`, `bca`, `cab`, and `cba`. These permutations are listed in lexicographical order based on the less-than operator. That is, `abc` is the first permutation because its first element is less than or equal to the first element in every other permutation, and its second element is smaller than any permutation sharing the same first element. Similarly, `acb` is the next permutation because it begins with `a`, which is smaller than the first element in any remaining permutation. Those permutations that begin with `b` come before those that begin with `c`.

考虑下面的三个字符的序列: `abc`。这个序列有 6 种可能的排列: `abc`, `acb`, `bac`, `bca`, `cab` 和 `cba`。基于小于操作符按字典序列出这些排列, 即, `abc` 是第一排列, 因为它的第一个元素小于或等于其他每个排列中的首元素, 而且, 它的第二个元素小于首元素的任意排列中的第二个元素。类似地, `acb` 是下一个排列, 因为它以 `a` 开头, `a` 小于其余任意排列中的首元素。以 `b` 开头的那些排列出现在以 `c` 开头的那些之前。

For any given permutation, we can say which permutation comes before it and which after it. Given the permutation `bca`, we can say that its previous permutation is `bac` and that its next permutation is `cab`. There is no previous permutation of the sequence `abc`, nor is there a next permutation of `cba`.

对于任意给定排列而言, 可以指出哪个排列出现在它之前以及哪个出现在它之后。给定排列 `bca`, 可以指出它的前一排列是 `bac`, 它的下一排列是 `cab`。序列 `abc` 之前没有排列, `cba` 之后也没有下一排列。

The library provides two permutation algorithms that generate the permutations of a sequence in lexicographical order. These algorithms reorder the sequence to hold the (lexicographically) next or previous permutation of the given sequence. They return a `bool` that indicates whether there was a next or previous permutation.

标准库提供两个排列算法, 按字典序产生序列的排列。这些算法重新排列序列, 以便 (按字典序) 保存给定序列的下一个或前一个排列。它们返回指出是否存在下一个或前一个排列的 `bool` 值。

The algorithms each have two versions: One uses the element type `<` operator, and the other takes an extra argument that specifies a comparison to use to compare the elements. These algorithms assume that the elements in the sequence are unique. That is, the algorithms assume that no two elements in the sequence have the same value.

每个算法有两个版本: 一个使用元素类型的 `<` 操作符, 另一个接受指定用于比较元素的比较关系的实参。这些算法假定序列中的元素是唯一的, 也就是说, 算法假定序列中没有两个元素具有相同值。

Permutation Algorithms Require Bidirectional Iterators

要求双向迭代器的排列算法

To produce the permutation, the sequence must be processed both forward and backward, thus requiring bidirectional iterators.

为了产生排列, 必须对序列进行前向和后向处理, 因此要求双向迭代器。

```
next_permutation(beg, end)
next_permutation(beg, end, comp)
```

If the sequence is already in the last permutation, then `next_permutation` reorders the sequence to be the lowest permutation and returns `false`. Otherwise, it transforms the input sequence into the next permutation, which is the lexicographically next ordered sequence and returns `true`. The first version uses the element `<` operator to compare elements; the second version uses specified comparison.

如果序列已经是在最后一个排列中, 则 `next_permutation` 将序列重新排列为最低排列并返回 `false`; 否则, 它将输入序列变换为下一个排列, 即字典序的下一个排列, 并返回 `true`。第一个版本使用元素的 `<` 操作符比较元素, 第二个版本使用指定的比较关系。

```
prev_permutation(beg, end)
prev_permutation(beg, end, comp)
```

Like `next_permutation`, but transforms the sequence to form the previous permutation. If this is the smallest permutation, then it reorders the sequence to be the largest permutation and returns `false`.

与 `next_permutation` 很像, 但变换序列以形成前一个排列。如果这是最小的排列, 则它将序列重新排列为最大排列, 并返回 `false`。

A.2.8. Set Algorithms for Sorted Sequences

A.2.8. 有序序列的集合算法

The set algorithms implement general set operations on a sequence that is in sorted order.

Section A.2. A Brief Tour of the Algorithms

集合算法实现有序列的通用集合运算。



These algorithms are distinct from the library `set` container and should not be confused with operations on `sets`. Instead, these algorithms provide setlike behavior on an ordinary sequential container (`vector`, `list`, etc.) or other sequence, such as an input stream.

这些算法不同于标准库中的 `set` 容器，不应该与 `set` 的操作相混淆，相反，这些算法提供普通顺序容器（`vector`、`list`，等等）或其他序列（如输入流）上的集合式行为。

With the exception of `includes`, they also take an output iterator. As usual, the programmer must ensure that the destination is large enough to hold the generated sequence. These algorithms return their `dest` iterator incremented to denote the element just after the last one that was written to `dest`.

除了 `include` 之外，它们也接受输出迭代器。程序员照常必须保证目的地足以保存生成的序列。这些算法返回它们的 `dest` 迭代器，`dest` 迭代器增量至指向紧接在写至 `dest` 的最后一个元素之后的元素。

Each algorithm provides two forms: The first uses the `<` operator for the element type to compare elements in the two input sequences. The second takes a comparison, which is used to compare the elements.

每个算法都提供两种形式：第一种形式使用元素类型的 `<` 操作符比较两个输入序列中的元素，第二种形式接受一个用于比较元素的比较关系。

Set Algorithms Require Input Iterators

要求输入迭代器集合算法

These algorithms process elements sequentially, requiring input iterators.

这些算法顺序处理元素，要求输入迭代器。

```
includes(beg, end, beg2, end2)
includes(beg, end, beg2, end2, comp)
```

Returns `true` if every element in the second sequence is contained in the input sequence. Returns `false` otherwise.

如果输入序列包含第二个序列中的每个元素，就返回 `true`；否则，返回 `false`。

```
set_union(beg, end, beg2, end2, dest)
set_union(beg, end, beg2, end2, dest, comp)
```

Creates a sorted sequence of the elements that are in either sequence. Elements that are in both sequences occur in the output sequence only once. Stores the sequence in `dest`.

创建在任一序列中存在的元素的有序序列。两个序列中都存在的元素在输出序列中只出现一次。将序列存储在 `dest` 中。

```
set_intersection(beg, end, beg2, end2, dest)
set_intersection(beg, end, beg2, end2, dest, comp)
```

Creates a sorted sequence of elements present in both sequences. Stores the sequence in `dest`.

创建在两个序列中都存在的元素的有序序列。将序列存储在 `dest` 中。

```
set_difference(beg, end, beg2, end2, dest)
set_difference(beg, end, beg2, end2, dest, comp)
```

Creates a sorted sequence of elements present in the first container but not in the second.

创建在第一个容器中但不在第二个容器中的元素的有序序列。

```
set_symmetric_difference(beg, end, beg2, end2, dest)
set_symmetric_difference(beg, end, beg2, end2, dest, comp)
```

Creates a sorted sequence of elements present in either container but not in both.

创建在任一容器中存在但不在两个容器中同时存在的元素的有序序列。

A.2.9. Minimum and Maximum Values

A.2.9. 最大值和最小值

The first group of these algorithms are unique in the library in that they operate on values rather than sequences. The second set takes a sequence that is denoted by input iterators.

Section A.2. A Brief Tour of the Algorithms

这些算法的第一组在标准库中是独特的，它们操作值而不是序列。第二组接受一个由输入迭代器表示的序列。

```
min(val1, val2)
min(val1, val2, comp)
max(val1, val2)
max(val1, val2, comp)
```

Returns the minimum/maximum of `val1` and `val2`. The arguments must have exactly the same type as each other. Uses either `<` operator for the element type or the specified comparison. Arguments and the return type are both `const` references, meaning that objects are not copied.

返回 `val1` 和 `val2` 的最大值／最小值。实参必须是完全相同的类型。使用元素类型的 `<` 操作符或指定的比较关系。实参和返回类型都是 `const` 引用，表示对象不是复制的。

```
min_element(beg, end)
min_element(beg, end, comp)
max_element(beg, end)
max_element(beg, end, comp)
```

Returns an iterator referring to the smallest/largest element in the input sequence. Uses either `<` operator for the element type or the specified comparison.

返回指向输入序列中最小／最大元素的迭代器。使用元素类型的 `<` 操作符或指定的比较关系。

Lexicographical Comparison

字典序比较关系

Lexicographical comparison examines corresponding elements in two sequences and determines the comparison based on the first unequal pair of elements. Because the algorithms process elements sequentially, they require input iterators. If one sequence is shorter than the other and all its elements match the corresponding elements in the longer sequence, then the shorter sequence is lexicographically smaller. If the sequences are the same size and the corresponding elements match, then neither is lexicographically less than the other.

字典序比较关系检查两个序列中的对应元素，并基于第一个不相等的元素对确定比较关系。因为算法顺序地处理元素，所以它们要求输入迭代器。如果一个序列比另一个短，并且它的元素与较长序列中对应元素相匹配，则较短的序列在字典序上较小。如果序列长短相同且对应元素匹配，则在字典序上两者都不小于另一个。

```
lexicographical_compare(beg1, end1, beg2, end2)
lexicographical_compare(beg1, end1, beg2, end2, comp)
```

Does an element by element comparison of the elements in the two sequences. Returns `true` if the first sequence is lexicographically less than the second sequence. Otherwise, returns `false`. Uses either `<` operator for the element type or the specified comparison.

对两个序列中的元素进行逐个比较。如果第一个序列在字典序上小于第二个序列，就返回 `true`；否则，返回 `false`。使用元素类型的 `<` 操作符或指定的比较关系。

A.2.10. Numeric Algorithms

A.2.10. 算术算法

Numeric algorithms require input iterators; if the algorithm writes output, it uses an output iterator for the destination

算术算法要求输入迭代器，如果算法修改输出，它就使用目的地输出迭代器。

These functions perform simple arithmetic manipulations of their input sequence. To use the numeric algorithms, the `numeric` header must be included.

这些算法执行它们的输入序列的简单算术操纵。要使用算术算法必须包含头文件 `numeric`。

```
accumulate(beg, end, init)
accumulate(beg, end, init, BinaryOp)
```

Returns the sum of all the values in the input range. The summation starts with the initial value specified by `init`. The return type is the same type as the type of `init`.

返回输入范围中所有值的总和。求和从指定的初始值 `init` 开始。返回类型是与 `init` 相同的类型。

Given the sequence `1,1,2,3,5,8` and an initial value of 0, the result is 20. The first version applies the `+` operator for the element type; second version applies the specified binary operation.

给定序列 `1,1,2,3,5,8` 以及初始值 0，结果是 20。第一个版本应用元素类型的 `+` 操作符，第二个版本应用指定的二元操作符。

```
inner_product(beg1, end1, beg2, init)
inner_product(beg1, end1, beg2, init, BinOp1, BinOp2)
```

Returns the sum of the elements generated as the product of two sequences. The two sequences are examined in step, and the elements from each sequence are multiplied. The product of that multiplication is summed. The initial value of the sum is specified by `init`. The second sequence beginning at `beg2` is assumed to have at least as many elements as are in the first sequence; any elements in the second sequence beyond the size of the first sequence are ignored. The type of `init` determines the return type.

Section A.2. A Brief Tour of the Algorithms

返回作为两个序列乘积而生成的元素的总和。步调一致地检查两个序列，将来自两个序列的元素相乘，将相乘的结果求和。由 `init` 指定和的初值。假定从 `beg2` 开始的第二个序列具有至少与第一个序列一样多的元素，忽略第二个序列中超出第一个序列长度的任何元素。`init` 的类型决定返回类型。

The first version uses the element's multiplication (`*`) and addition (`+`) operators: Given the two sequences `2,3,5,8` and `1,2,3,4,5,6,7`, the result is the sum of the initial value plus the following product pairs:

第一个版本使用元素的乘操作符 (`*`) 和加操作符 (`+`)：给定两个序列 `2,3,5,8` 和 `1,2,3,4,5,6,7`，结果是初值加上下面的乘积对：

```
initial_value + (2 * 1) + (3 * 2) + (5 * 3) + (8 * 4)
```

If we provide an initial value of 0, then the result is 55.

如果提供初值 0，则结果是 55。

The second version applies the specified binary operations, using the first operation in place of addition and the second in place of multiplication. As an example, we might use `inner_product` to produce a list of parenthesized namevalue pairs of elements, where the name is taken from the first input sequence and the corresponding value is in the second:

第二个版本应用指定的二元操作，使用第一个操作代替加而第二个操作代替乘。作为例子，可以使用 `inner_product` 来产生以括号括住的元素的名—值对的列表，这里从第一个输入序列获得名字，从第二个序列中获得对应的值：

```
// combine elements into a parenthesized, comma-separated pair
string combine(string x, string y)
{
    return "(" + x + ", " + y + ")";
}

// add two strings, each separated by a comma
string concatenate(string x, string y)
{
    if (x.empty())
        return y;
    return x + ", " + y;
}

cout << inner_product(names.begin(), names.end(),
                      values.begin(), string(),
                      concatenate, combine);
```

If the first sequence contains `if`, `string`, and `sort`, and the second contains `keyword`, `library type`, and `algorithm`, then the output would be

如果第一个序列包含 `if`、`string` 和 `sort`，且第二个序列包含 `keyword`、`library type` 和 `algorithm`，则输出将是

```
(if, keyword), (string, library type), (sort, algorithm)

partial_sum(beg, end, dest)
partial_sum(beg, end, dest, BinaryOp)
```

Writes a new sequence to `dest` in which the value of each new element represents the sum of all the previous elements up to and including its position within the input range. The first version uses the `+` operator for the element type; the second version applies the specified binary operation. The programmer must ensure that `dest` is at least as large as the input sequence. Returns the `dest` iterator incremented to refer just after the last element written.

将新序列写至 `dest`，其中每个新元素的值表示输入范围中在它的位置之前（不包括它的位置）的所有元素的总和。第一个版本使用元素类型 `+` 操作符，第二个版本应用指定的二元操作符。程序员必须保证 `dest` 至少与输入序列一样大。返回 `dest`，`dest` 增量到指向被写入的最后元素的下一位置。

Given the input sequence `0,1,1,2,3,5,8`, the destination sequence will be `0,1,2,4,7,12,20`. The fourth element, for example, is the partial sum of the three previous values (`0,1,1`) plus its own (`2`), yielding a value of 4.

给定输入序列 `0,1,1,2,3,5,8`，目的序列将是 `0,1,2,4,7,12,20`。例如，第四个元素是前三值 (`0,1,1`) 的部分和加上它自己的值 (`2`)，获得值 4。

```
adjacent_difference(beg, end, dest)
adjacent_difference(beg, end, dest, BinaryOp)
```

Writes a new sequence to `dest` in which the value of each new element other than the first represents the difference of the current and previous element. The first version uses the element type's `-` operation; the second version applies the specified binary operation. The programmer must ensure that `dest` is at least as large as the input sequence.

将新序列写至 `dest`，其中除了第一个元素之外每个新元素表示当前元素和前一元素的差。第一个版本使用元素类型的 `-` 操作符，第二个版本应用指定的二元操作。程序员必须保证 `dest` 至少与输入序列一样大。

Given the sequence `0,1,1,2,3,5,8`, the first element of the new sequence is a copy of the first element of the original sequence: 0. The second element is the difference between the first two elements: 1. The third element is the difference between the second and third element, which is 0, and so on. The new sequence is `0,1,0,1,1,2,3`.

给定序列 `0,1,1,2,3,5,8`，新序列的第一个元素是原序列第一个元素的副本：0，第二个元素是前两个元素的差：1，第三个元素是原序列第三个元素和第二个元素的差，为 0，以此类推，新序列是 `0,1,0,1,1,2,3`。

A.3. The IO Library Revisited

A.3. 再谈 IO 库

In [Chapter 8](#) we introduced the basic architecture and most commonly used parts of the IO library. This Appendix completes our coverage of the IO library.

[第八章](#)介绍过 IO 库的基本体系结构以及最常使用的部分，本附录完成对 IO 库的讨论。

A.3.1. Format State

A.3.1. 格式状态

In addition to a condition state ([Section 8.2](#), p. 287), each `iostream` object also maintains a format state that controls the details of how IO is formatted. The format state controls aspects of formatting such as the notational base for an integral value, the precision of a floating-point value, the width of an output element, and so on. The library also defines a set of manipulators (listed in Tables A.2 (p. 829) and A.3 (p. 833) for modifying the format state of an object. Simply speaking, a manipulator is a function or object that can be used as an operand to an input or output operator. A manipulator returns the stream object to which it is applied, so we can output multiple manipulators and data in a single statement.

除了条件状态（[第 8.2 节](#)）之外，每个 `iostream` 对象还维持一个控制 IO 格式化细节的格式状态。格式状态控制格式化特征，如是整型值的基数、浮点值的精度、输出元素的宽度等。标准库还定义了一组操纵符（在[表 A.2](#) 和 [表 A.3](#) 列出）来修改对象的格式状态。简单说来，操纵符是可用作输入或输出操作符操作数的函数或对象。操纵符返回其应用于的流对象，所以可以在一个语句中输出多个操纵符和数据。

将真和假显示为 1, 0

Table A.2. Manipulators Defined in `iostream`

表 A.2. `iostream` 中定义的操纵符

<code>boolalpha</code>	Display true and false as strings 将真和假显示为字符串
<code>noboolalpha</code>	Display true and false as 0, 1
<code>showbase</code>	Generate prefix indicating numeric base 产生指出数的基数的前缀
<code>noshowbase</code>	Do not generate notational base prefix 不产生记数基数前缀
<code>showpoint</code>	Always display decimal point 总是显示小数点
<code>noshowpoint</code>	Only display decimal point if fraction 有小数部分才显示小数点
<code>showpos</code>	Display + in nonnegative numbers 显示非负数中的 +
<code>noshowpos</code>	Do not display + in nonnegative numbers 不显示非负数中的 +
<code>uppercase</code>	Print OX in hexadecimal, E in scientific 在十六进制中打印 OX，科学记数法中打印 E
<code>nouppercase</code>	Print Ox in hexadecimal, e in scientific

Section A.3. The IO Library Revisited

	在十六进制中打印 0x，科学记数法中打印 e
x dec	Display in decimal numeric base 用十进制显示
hex	Display in hexadecimal numeric base 用十六进制显示
oct	Display in octal numeric base 用八进制显示
left	Add fill characters to right of value 在值的右边增加填充字符
right	Add fill characters to left of value 在值的左边增加填充字符
internal	Add fill characters between sign and value 在符号和值之间增加填充字符
fixed	Display floating-point in decimal notation 用小数形式显示浮点数
scientific	Display floating-point in scientific notation 用科学记数法显示浮点数
flush	Flush ostream buffer 刷新 ostream 缓冲区
ends	Insert null, then flush ostream buffer 插入空字符，然后刷新 ostream 缓冲区
endl	Insert newline, then flush ostream buffer 插入换行符，然后刷新 ostream 缓冲区
unitbuf	Flush buffers after every output operation 在每个输出操作之后刷新缓冲区
x nounitbuf	Restore normal buffer flushing 恢复常规缓冲区刷新
x skipws	Skip whitespace with input operators 为输入操作符跳过空白
noskipws	Do not skip whitespace with input operators 不为输入操作符跳过空白
ws	"Eat" whitespace "吃掉"空白

x indicates default stream state

注：带 x 的是默认流状态。

Table A.3. Manipulators Defined in `iomanip`

表 A.3. `iomanip` 中定义的操纵符

setfill(ch)	Fill whitespace with ch 用 ch 填充空白
-------------	--------------------------------------

<code>setprecision(n)</code>	Set floating-point precision to <code>n</code> 将浮点精度置为 <code>n</code>
<code>setw(w)</code>	Read or write value to <code>w</code> characters 读写 <code>w</code> 个字符的值
<code>setbase(b)</code>	Output integers in base <code>b</code> 按基数 <code>b</code> 输出整数

When we read or write a manipulator, no data are read or written. Instead, an action is taken. Our programs have already used one manipulator, `endl`, which we "write" to an output stream as if it were a value. But `endl` isn't a value; instead, it performs an operation: It writes a newline and flushes the buffer.

读写操纵符的时候，不读写数据，相反，会采取某种行动。示例程序已经使用过一个操纵符——`endl`，我们将它“写”至输出流，就好像它是一个值一样。但 `endl` 并不是一个值，相反，它执行一个操作：写换行符并刷新缓冲区。

A.3.2. Many Manipulators Change the Format State

A.3.2. 许多操纵符改变格式状态

Many manipulators change the format state of the stream. They change the format of how floating-pointer numbers are printed or whether a `bool` is displayed as a numeric value or using the `bool` literals, `true` or `false`, and so forth.

许多操纵符改变流的格式状态。它们改变显示浮点数的格式，将 `bool` 值显示为数值还是使用 `bool` 字面值 `true` 和 `false` 的格式，诸如此类。



Manipulators that change the format state of the stream usually leave the format state changed for all subsequent IO.

改变流格式状态的操纵符通常为后续 IO 保留改变后的格式状态。

Most of the manipulators that change the format state provide set/unset pairs; one manipulator sets the format state to a new value and the other unsets it, restoring the normal default formatting.

大多数改变格式状态的操纵符提供设置／复原对，一个操纵符将格式状态置为新值而另一个进行复原，恢复常规默认格式。

The fact that a manipulator makes a persistent change to the format state can be useful when we have a set of IO operations that want to use the same formatting. Indeed, some programs take advantage of this aspect of manipulators to reset the behavior of one or more formatting rules for all its input or output. In such cases, the fact that a manipulator changes the stream is a desirable property.

操纵符进行格式状态的持久改变，在有一组 IO 操作希望使用相同格式化的时候，这一事实有用。事实上，一些各市利用操纵符的这个特征，为自己的所有输入或输出重置一个或多个格式化规则的行为。这种情况下，操纵符改变流是希望得到的性质。

However, many programs (and, more importantly, programmers) expect the state of the stream to match the normal library defaults. In these cases, leaving the state of the stream in a nonstandard state can lead to errors.

但是，许多程序（更重要的是，许多程序员）希望流的状态与标准库默认值匹配。这些情况下，使流状态停留在非标准状态可能会导致错误。



It is usually best to undo any state change made by a manipulator. Ordinarily, a stream should be in its ordinary, default state after every IO operation.

取消操纵符的任何状态改变通常是最好的。一般而言，流应该在每个 IO 操作之后处于通常的默认状态。

Using `flags` Operation to Restore the Format State

用 `flags` 操纵恢复格式状态

An even better approach to managing changes to format state uses the `flags` operations. The `flags` operations are similar to the `rdstate` and `setstate` operations that manage the condition state of the stream. In this case, the library defines a pair of `flags` functions:

管理格式状态改变的一个更好的办法是使用 `flags` 操作。`flags` 操作类似于管理流的条件状态的 `rdstate` 和 `setstate` 操作。这种情况下，标准库定义了一对 `flags` 函数：

- `flags()` with no arguments returns the stream's current format state. The value returned is a library defined type named `fmtflags`.
不带实参的 `flags()` 返回流的当前格式状态。返回值是名为 `fmtflags` 的标准库定义类型。
- `flags(arg)` takes a `fmtflags` argument and sets the stream's format as indicated by the argument.
`flags(arg)` 接受一个实参并将流格式置为实参所指定的格式。

We can use these functions to remember and restore the format state of either an input or output stream:

可以使用这些函数记住并恢复输入或输出流的格式状态：

```
void display(ostream& os)
{
    // remember the current format state
    ostream::fmtflags curr_fmt = os.flags();
    // do output that uses manipulators that change the format state of os
    os.flags(curr_fmt);           // restore the original format state of os
}
```

A.3.3. Controlling Output Formats

A.3.3. 控制输出格式

Many of the manipulators allow us to change the appearance of our output. There are two broad categories of output control: controlling the presentation of numeric values and controlling the amount and placement of padding.

许多操纵符使我们能够改变输出的外观。有两大类的输出控制：控制数值的表示，以及控制填充符的数量和布局。

Controlling the Format of Boolean Values

控制布尔值和格式

One example of a manipulator that changes the formatting state of its object is the `boolalpha` manipulator. By default, `bool` values print as 1 or 0. A `true` value is written as the integer 1 and a `false` value as 0. We can override this formatting by applying the `boolalpha` manipulator to the stream:

改变对象格式化状态的操纵符的一个例子是 `boolalpha` 操纵符。默认情况下，将 `bool` 值显示为 1 或 0，`true` 值显示为 1，而 `false` 值显示为 0。可以通过流的 `boolalpha` 操纵符覆盖这个格式化：

```
cout << "default bool values: "
<< true << " " << false
<< "\nalpha bool values: "
<< boolalpha
<< true << " " << false
<< endl;
```

When executed, the program generates the following:

执行时，这段程序产生下面的输出：

```
default bool values: 1 0
alpha bool values: true false
```

Once we "write" `boolalpha` on `cout`, we've changed how `cout` will print `bool` values from this point on. Subsequent operations that print `bools` will print them as either `true` or `false`.

一旦将 `boolalpha` "写"至 `cout`，从这个点起就改变了 `cout` 将怎样显示 `bool` 值，后续显示 `bool` 值的操作将用 `true` 或 `false` 进行显示。

To undo the format state change to `cout`, we must apply `noboolalpha`:

要取消 `cout` 的格式状态改变，必须应用 `noboolalpha`：

```
bool bool_val;
cout << boolalpha      // sets internal state of cout
<< bool_val
<< noboolalpha; // resets internal state to default formatting
```

Section A.3. The IO Library Revisited

Now we change the formatting of `bool` values only to print of `bool_val` and immediately reset the stream back to its initial state.

现在只改变 `bool` 值的格式化来显示 `bool_val`，并且立即将流重置为原来的状态。

Specifying the Base for Integral Values

指定整型值的基数

By default, integral values are written and read in decimal notation. The programmer can change the notational base to octal or hexadecimal or back to decimal (the representation of floating-point values is unaffected) by using the manipulators `hex`, `oct`, and `dec`:

默认情况下，用十进制读写整型值。通过使用操纵符 `hex`、`oct` 和 `dec`，程序员可以将表示进制改为八进制、十六进制或恢复十进制（浮点值的表示不受影响）：

```
const int ival = 15, jval = 1024; // const, so values never change
cout << "default: ival = " << ival
     << " jval = " << jval << endl;
cout << "printed in octal: ival = " << oct << ival
     << " jval = " << jval << endl;
cout << "printed in hexadecimal: ival = " << hex << ival
     << " jval = " << jval << endl;
cout << "printed in decimal: ival = " << dec << ival
     << " jval = " << jval << endl;
```

When compiled and executed, the program generates the following output:

编译和执行的时候，程序产生下面的输出：

```
default: ival = 15 jval = 1024
printed in octal: ival = 17 jval = 2000
printed in hexadecimal: ival = f jval = 400
printed in decimal: ival = 15 jval = 1024
```

Notice that like `boolalpha`, these manipulators change the format state. They affect the immediately following output, and all subsequent integral output, until the format is reset by invoking another manipulator.

注意，像 `boolalpha` 一样，这些操纵符改变格式状态。它们影响紧接在后面的输出，以及所有后续的整型输出，直到通过调用另一操纵符重置格式为止。

Indicating Base on the Output

指出输出的基数

By default, when we print numbers, there is no visual cue as to what notational base was used. Is 20, for example, really 20, or an octal representation of 16? When printing numbers in decimal mode, the number is printed as we expect. If we need to print octal or hexadecimal values, it is likely that we should also use the `showbase` manipulator. The `showbase` manipulator causes the output stream to use the same conventions as used for specifying the base of an integral constant:

默认情况下，显示数值的时候，不存在关于所用基数的可见记号。例如，20 是 20，还是 16 的八进制表示？按十进制模式显示数值的时候，会按我们期待的格式打印数值。如果需要打印八进制或十六进制值，可能应该也使用 `showbase` 操纵符。`showbase` 操纵符导致输出流使用的约定，与指定整型常量基数所用的相同：

- A leading `Ox` indicates hexadecimal
以 `Ox` 为前导表示十六进制。
- A leading `O` indicates octal
以 `O` 为前导表示八进制。
- The absence of either indicates decimal
没有任何前导表示十进制。

Here is the program revised to use `showbase`:

修改程序使用 `showbase` 如下：

```
const int ival = 15, jval = 1024; // const so values never change
cout << showbase; // show base when printing integral values
cout << "default: ival = " << ival
     << " jval = " << jval << endl;
cout << "printed in octal: ival = " << oct << ival
     << " jval = " << jval << endl;
cout << "printed in hexadecimal: ival = " << hex << ival
     << " jval = " << jval << endl;
```

Section A.3. The IO Library Revisited

```
cout << "printed in decimal: ival = " << dec << ival
<< " jval = " << jval << endl;
cout << noshowbase; // reset state of the stream
```

The revised output makes it clear what the underlying value really is:

修改后的输出使得基础值到底是什么很清楚：

```
default: ival = 15 jval = 1024
printed in octal: ival = 017 jval = 02000
printed in hexadecimal: ival = 0xf jval = 0x400
printed in decimal: ival = 15 jval = 1024
```

The `noshowbase` manipulator resets `cout` so that it no longer displays the notational base of integral values.

`noshowbase` 操纵符重置 `cout`，以便它不再显示整型值的表示基数。

By default, hexadecimal values are printed in lowercase with a lowercase x. We could display the x and the hex digits af as uppercase by applying the `uppercase` manipulator.

默认情况下，十六进制值用带小写 x 的小写形式打印。可以应用 `uppercase` 操纵符显示 x 并将十六进制数字 a - f 显示为大写字母。

```
cout << uppercase << showbase << hex
<< "printed in hexadecimal: ival = " << ival
<< " jval = " << jval << endl
<< nouppercase << endl;
```

The preceding program generates the following output:

前面的程序产生下面的输出：

```
printed in hexadecimal: ival = 0XF jval = 0X400
```

To revert back to the lowercase x, we apply the `nouppercase` manipulator.

要恢复小写，就应用 `nouppercase` 操纵符。

Controlling the Format of Floating-Point Values

控制浮点值的格式

There are three aspects of formatting floating-point values that we can control:

对于浮点值的格式化，可以控制下面三个方面：

- Precision: how many digits are printed
精度：显示多少位数字。
- Notation: whether to print in decimal or scientific notation
记数法：用小数还是科学记法显示。
- Handling of the decimal point for floating-point values that are whole numbers
对是整数的浮点值的小数点的处理。

By default, floating-point values are printed using six digits of precision. If the value has no fractional part, then the decimal point is omitted. Whether the number is printed using decimal or scientific notation depends on the value of the floating-point number being printed. The library chooses a format that enhances readability of the number. Very large and very small values are printed using scientific notation. Other values use fixed decimal.

默认情况下，使用六位数字的精度显示浮点值。如果值没有小数部分，则省略小数点。使用小数形式还是科学记数法显示数值取决于被显示的浮点数的值，标准库选择增强数值可读性的格式，非常大和非常小的值使用科学记数法显示，其他值使用小数形式。

Specifying How Much Precision to Print

指定显示精度

By default, precision controls the total number of digits that are printed. When printed, floating-point values are rounded, not truncated, to the current precision. Thus, if the current precision is four, then 3.14159 becomes 3.142；if the precision is three, then it is printed as 3.14。

Section A.3. The IO Library Revisited

默认情况下，精度控制显示的数字总位数。显示的时候，将浮点值四舍五入到当前精度。因此，如果当前精度是 4，则 3.14159 成为 3.142；如果精度是 3，打印为 3.14。

We can change the precision through a member function named `precision` or by using the `setprecision` manipulator. The `precision` member is overloaded ([Section 7.8](#), p. 265): One version takes an `int` value and sets the precision to that new value. It returns the previous precision value. The other version takes no arguments and returns the current precision value. The `setprecision` manipulator takes an argument, which it uses to set the precision.

通过名为 `precision` 的成员函数，或者通过使用 `setprecision` 操纵符，可以改变精度。`precision` 成员是重载的 ([第 7.8 节](#))：一个版本接受一个 `int` 值并将精度设置为那个新值，它返回先前的精度值；另一个版本不接受实参并返回当前精度值。`setprecision` 操纵符接受一个实参，用来设置精度。

The following program illustrates the different ways we can control the precision use when printing floating point values:

下面的程序说明控制显示浮点值所用精度的不同方法：

```
// cout.precision reports current precision value
cout << "Precision: " << cout.precision()
     << ", Value: " << sqrt(2.0) << endl;
// cout.precision(12) asks that 12 digits of precision to be printed
cout.precision(12);
cout << "Precision: " << cout.precision()
     << ", Value: " << sqrt(2.0) << endl;
// alternative way to set precision using setprecision manipulator
cout << setprecision(3);
cout << "Precision: " << cout.precision()
     << ", Value: " << sqrt(2.0) << endl;
```

When compiled and executed, the program generates the following output:

编译并执行后，程序产生下面的输出：

```
Precision: 6, Value: 1.41421
Precision: 12, Value: 1.41421356237
Precision: 3, Value: 1.41
```

This program calls the library `sqrt` function, which is found in the `cmath` header. The `sqrt` function is overloaded and can be called on either a `float`, `double`, or `long double` argument. It returns the square root of its argument.

这个程序调用标准库中的 `sqrt` 函数，可以在头文件 `cmath` 中找到它。`sqrt` 函数量重载的，可以用 `float`、`double` 或 `long double` 实参调用，它返回实参的平方根。



The `setprecision` manipulators and other manipulators that take arguments are defined in the `iomanip` header.

操纵符和其他接受实参的操纵符定义在头文件 `iomanip` 中。

Controlling the Notation

控制记数法

By default, the notation used to print floating-point values depends on the size of the number: If the number is either very large or very small, it will be printed in scientific notation; otherwise, fixed decimal is used. The library chooses the notation that makes the number easiest to read.

默认情况下，用于显示浮点值的记数法取决于数的大小：如果数很大或很小，将按科学记数法显示，否则，使用固定位数的小数。标准库选择使得数容易阅读的记数法。



When printing a floating-point number as a plain number (as opposed to printing money, or a percentage, where we want to control the appearance of the value), it is usually best to let the library choose the notation to use. The one time to force either scientific or fixed decimal is when printing a table in which the decimal points should line up.

将浮点数显示为普通数（相对于显示货币、百分比，那时我们希望控制值的外观）的时候，通常最好让标准库来选择使用的记数法。要强制科学记数法或固定位数小数的一种情况是在显示表的时候，表中的小数点应该对齐。

If we want to force either scientific or fixed notation, we can do so by using the appropriate manipulator: The `scientific` manipulator changes the stream to use scientific notation. As with printing the `x` on hexadecimal integral values, we can also control the case of the `e` in scientific mode through the `uppercase` manipulator. The `fixed` manipulator changes the stream to use fixed decimal.

如果希望强制科学记数法或固定位数小数表示，可以通过使用适当的操纵符做到这一点：`scientific` 操纵符将流变为使用科学记数法。像在十六进制值上显示 `x` 一样，也可以通过 `uppercase` 操纵符控制科学记数法中的 `e`。`fixed` 操纵符将流为使用固定位数小数表示。

These manipulators change the default meaning of the precision for the stream. After executing either `scientific` or `fixed`, the precision value

Section A.3. The IO Library Revisited

controls the number of digits after the decimal point. By default, precision specifies the total number of digits both before and after the decimal point. Using `fixed` or `scientific` lets us print numbers lined up in columns. This strategy ensures that the decimal point is always in a fixed position relative to the fractional part being printed.

这些操纵符改变流精度的默认含义。执行 `scientific` 或 `fixed` 之后，精度值控制小数点之后的数位。默认情况下，精度指定数字的总位数——小数点之前和之后。使用 `fixed` 或 `scientific` 命名我们能够按列对齐来显示数，这一策略保证小数点总是在相对于被显示的小数部分固定的位置。

Reverting to Default Notation for Floating-Point Values

恢复浮点值的默认记数法

Unlike the other manipulators, there is no manipulator to return the stream to its default state in which it chooses a notation based on the value being printed. Instead, we must call the `unsetf` member to undo the change made by either `scientific` or `fixed`. To return the stream to default handling of float values we pass `unsetf` function a library-defined value named `floatfield`:

```
// reset to default handling for notation
cout.unsetf(ostream::floatfield);
```

Except for undoing their effect, using these manipulators is like using any other manipulator:

除了取消它们的效果之外，使用这些操纵符像使用任意其他操纵符一样：

```
cout << sqrt(2.0) << '\n' << endl;
cout << "scientific: " << scientific << sqrt(2.0) << '\n';
    << "fixed decimal: " << fixed << sqrt(2.0) << "\n\n";
cout << uppercase
    << "scientific: " << scientific << sqrt(2.0) << '\n';
    << "fixed decimal: " << fixed << sqrt(2.0) << endl
    << nouppercase;
// reset to default handling for notation
cout.unsetf(ostream::floatfield);
cout << '\n' << sqrt(2.0) << endl;
```

produces the following output:

产生如下输出：

```
1.41421
scientific: 1.414214e+00
fixed decimal: 1.414214

scientific: 1.414214E+00
fixed decimal: 1.414214

1.41421
```

Printing the Decimal Point

显示小数点

By default, when the fractional part of a floating-point value is 0, the decimal point is not displayed. The `showpoint` manipulator forces the decimal point to be printed:

默认情况下，当浮点值的小数部分为 0 的时候，不显示小数点。`showpoint` 操纵符强制显示小数点：

```
cout << 10.0 << endl;           // prints 10
cout << showpoint << 10.0      // prints 10.0000
    << noshowpoint << endl; // revert to default handling of decimal point
```

The `noshowpoint` manipulator reinstates the default behavior. The next output expression will have the default behavior, which is to suppress the decimal point if the floating-point value has a 0 fractional part.

`noshowpoint` 操纵符恢复默认行为。下一个输出表达式将具有默认行为，即，如果浮点值小数部分为 0，就取消小数点。

Padding the Output

填充输出

Section A.3. The IO Library Revisited

When printing data in columns, we often want fairly fine control over how the data are formatted. The library provides several manipulators to help us accomplish the control we might need:

按栏显示数据的时候，经常很希望很好地控制数据的格式化。标准库提供下面几个操纵帮助我们实现需要的控制：

- `setw` to specify the minimum space for the next numeric or string value.
`setw`, 指定下一个数值或字符串的最小间隔。
- `left` to left-justify the output.
`left`, 左对齐输出。
- `right` to right-justify the output. Output is right-justified by default.
`right`, 右对齐输出。输出默认为右对齐。
- `internal` controls placement of the sign on negative values. `internal` left-justifies the sign and right-justifies the value, padding any intervening space with blanks.
`internal`, 控制负值的符号位置。`internal` 左对齐符号且右对齐值，用空格填充介于其间的空间。
- `setfill` lets us specify an alternative character to use when padding the output. By default, the value is a space.
`setfill`, 使我们能够指定填充输出时使用的另一个字符。默认情况下，值是空格。



`setw`, like `endl`, does not change the internal state of the output stream. It determines the size of only the next output.

像 `endl` 一样，`setw` 不改变输出流的内部状态，它只决定下一个输出的长度。

The following program illustrates these manipulators

下面程序段说明了这些操纵符：

```
int i = -16;
double d = 3.14159;

// pad first column to use minimum of 12 positions in the output
cout << "i: " << setw(12) << i << "next col" << '\n';
<< "d: " << setw(12) << d << "next col" << '\n';

// pad first column and left-justify all columns
cout << left
    << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n'
    << right; // restore normal justification

// pad first column and right-justify all columns
cout << right
    << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n';

// pad first column but put the padding internal to the field
cout << internal
    << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n';

// pad first column, using # as the pad character
cout << setfill('#')
    << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n'
    << setfill(' '); // restore normal pad character
```

When executed, this program generates

执行时，该程序段产生如下输出：

```
i:      -16next col
d:      3.14159next col
i: -16      next col
d: 3.14159      next col
i:      -16next col
d:      3.14159next col
i: -      16next col
d:      3.14159next col
i: -######16next col
d: #####3.14159next col
```

A.3.4. Controlling Input Formatting

A.3.4. 控制输入格式化

By default, the input operators ignore whitespace (blank, tab, newline, formfeed, and carriage return). The following loop

默认情况下，输入操作符忽略空白（空格、制表符、换行符、进纸和回车）。对下面的循环：

```
while (cin >> ch)
    cout << ch;
```

given the input sequence

```
a b c
d
```

executes four times to read the characters `a` through `d`, skipping the intervening blanks, possible tabs, and newline characters. The output from this program is

循环执行四次从字符 `a` 读到 `d`，跳过介于其间的空格、可能的制表符和换行符。该程序段的输出是：

```
abcd
```

The `noskipws` manipulator causes the input operator to read, rather than skip, whitespace. To return to the default behavior, we apply `skipws` manipulator:

`noskipws` 操纵符导致输入操作符读（而不是跳过）空白。要返回默认行为，应用 `skipws` 操纵符：

```
cin >> noskipws; // set cin so that it reads whitespace
while (cin >> ch)
    cout << ch;
cin >> skipws; // reset cin to default state so that it discards whitespace
```

Given the same input as before, this loop makes seven iterations, reading white-space as well as the characters in the input. This loop generates
给定与前面相同的输入，该循环进行 7 次迭代，读输入中的空白以及字符。该循环产生如下输出：

```
a b c
d
```

A.3.5. Unformatted Input/Output Operations

A.3.5. 未格式化的输入／输出操作

So far, our programs have used only formatted IO operations. The input and output operators (`<<` and `>>`) format the data they read or write according to the data type being handled. The input operators ignore whitespace; the output operators apply padding, precision, and so on.

迄今为止，示例程序中只使用过格式化的 IO 操作。输入和输出操作符（`<<` 和 `>>`）根据被处理数据的类型格式化所读写的数据。输入操作符忽略空白，输出操作符应用填充、精度等。

The library also provides a rich set of low-level operations that support unformatted IO. These operations let us deal with a stream as a sequence of uninterpreted bytes rather than as a sequence of data types, such as `char`, `int`, `string`, and so on.

标准库还提供了丰富的支持未格式化 IO 的低级操作，这些操作使我们能够将流作为未解释的字节序列处理，而不是作为数据类型（如 `char`、`int`、`string` 等）的序列处理。

A.3.6. Single-Byte Operations

A.3.6. 单字节操作

Several of the unformatted operations deal with a stream one byte at a time. They read rather than ignore whitespace. For example, we could use the unformatted IO operations `get` and `put` to read the characters one at a time:

几个未格式化的操作一次一个字节地处理流，它们不忽略空白地读。例如，可以使用未格式化 IO 操作 `get` 和 `put` 一次读一个字符：

```
char ch;
while (cin.get(ch))
    cout.put(ch);
```

This program preserves the whitespace in the input. Its output is identical to the input. Given the same input as read by the previous program that used `noskipws`, this program generates the same output:

该程序段保持输入中的空白。它的输出与输入相同。给定与前面使用 `noskipws` 的程序段相同的输入，该程序段产生相同的输出：

```
a b c
```

d

Table A.4. Single-Byte Low-Level IO Operations**表 A.4. 单字节低级 IO 操作**

<code>is.get(ch)</code>	Puts next byte from the <code>istream is</code> in character <code>ch</code> . Returns <code>is</code> .
	将 <code>istream is</code> 的下一个字节放入 <code>ch</code> , 返回 <code>is</code>
<code>os.put(ch)</code>	Puts character <code>ch</code> onto the <code>ostream os</code> . Returns <code>os</code> .
	将字符 <code>ch</code> 放入 <code>ostream</code> , 返回 <code>os</code>
<code>is.get()</code>	Returns next byte from <code>is</code> as an <code>int</code> .
	返回 <code>is</code> 的下一字节作为一个 <code>int</code> 值
<code>is.putback(ch)</code>	Puts character <code>ch</code> back on <code>is</code> ; returns <code>is</code> .
	将字符 <code>ch</code> 放回 <code>is</code> , 返回 <code>is</code>
<code>is.unget()</code>	Moves <code>is</code> back one byte; returns <code>is</code> .
	将 <code>is</code> 退回一个字节, 返回 <code>is</code>
<code>is.peek()</code>	Returns the next byte as an <code>int</code> but doesn't remove it.
	将下一字节作为 <code>int</code> 值返回但不移出它

Putting Back onto an Input Stream

在输入流上倒退

Sometimes we need to read a character in order to know that we aren't ready for it yet. In such cases, we'd like to put the character back onto the stream. The library gives us three ways to do so, each of which has subtle differences from the others:

有时我们需要读一个字符才知道还没有为它作好准备，在这种情况下，希望将字符放回流上。标准库给出三种方法做到这一点，它们之间有下列微妙的区别：

- `peek` returns a copy of the next character on the input stream but does not change the stream. The value returned by `peek` stays on the stream and will be the next one retrieved.
`peek`, 返回输入流上下一字符的副本但不改变流。`peek` 返回的值留在流上，且将是下一个被检索的。
- `unget` backs up the input stream so that whatever value was last returned is still on the stream. We can call `unget` even if we do not know what value was last taken from the stream.
`unget`, 使输入流倒退，以便最后返回的值仍在流上。即使不知道最近从流获得的是什么值，也可以调用 `unget`。
- `putback` is a more specialized version of `unget`: It returns the last value read from the stream but takes an argument that must be the same as the one that was last read. Few programs use `putback` because the simpler `unget` does the same job with fewer constraints.
`putback`, `unget` 的更复杂的版本。它返回从流中读到的最后一个值，但接受最后一次读的同一实参。很少有程序使用 `putback`，因为更简单的 `unget` 也可以完成相同工作而约束更少。

In general, we are guaranteed to be able to put back at most one value before the next read. That is, we are not guaranteed to be able to call `putback` or `unget` successively without an intervening read operation.

一般而言，保证能够在下一次读之前放回最多一个值，也就是说，不保证能够连续调用 `putback` 或 `unget` 而没有介于其间的读操作。

`int` Return Values from Input Operations

输入操作的 `int` 返回值

The version of `get` that takes no argument and the `peek` function return a character from the input stream as an `int`. This fact can be surprising; it might seem more natural to have these functions return a `char`.

Section A.3. The IO Library Revisited

不接受实参的 `get` 版本和 `peek` 函数从输入流返回一个字符作为 `int` 值。这个事实可能令人惊讶，因为这些函数返回 `char` 值似乎更自然。

The reason that these functions return an `int` is to allow them to return an end-of-file marker. A given character set is allowed to use every value in the `char` range to represent an actual character. Thus, there is no extra value in that range to use to represent end-of-file.

这些函数返回 `int` 值的原因是为了允许它们返回一个文件结束标记。允许给定字符集使用 `char` 范围的每一个值来表示实际字符，因此，该范围内没有额外值用来表示文件结束符。

Instead, these functions convert the character to `unsigned char` and then promote that value to `int`. As a result, even if the character set has characters that map to negative values, the `int` returned from these operations will be a positive value ([Section 2.1.1, p. 36](#)). By returning end-of-file as a negative value, the library guarantees that end-of-file will be distinct from any legitimate character value. Rather than requiring us to know the actual value returned, the `iostream` header defines a `const` named `EOF` that we can use to test if the value returned from `get` is end-of-file. It is essential that we use an `int` to hold the return from these functions:

相反，这些函数将字符转换为 `unsigned char`，然后将那个值提升为 `int`，因此，即使字符集有映射到负值的字符，从这些操作返回的值也将是一个正值（[第 2.1.1 节](#)）。通过将文件结束符作为负值返回，标准库保证文件结束符区别于任意合法字符值。为了不要求我们知道返回的实际值，头文件 `iostream` 定义了名为 `EOF` 的 `const`，可以使用它来测试 `get` 的返回值是否为文件结束符。实质上我们使用 `int` 对象来保存这些函数的返回值：

```
int ch; // NOTE: int, not char!!!!
// loop to read and write all the data in the input
while ((ch = cin.get()) != EOF)
    cout.put(ch);
```

This program operates identically to one on page [834](#), the only difference being the version of `get` that is used to read the input.

这个程序段与 [A.3.6 节](#) 中的那个程序段同样操作，唯一的区别是读入的 `get` 版本不同。

A.3.7. Multi-Byte Operations

A.3.7. 多字节操作

Other unformatted IO operations deal with chunks of data at a time. These operations can be important if speed is an issue, but like other low-level operations they are error-prone. In particular, these operations require us to allocate and manage the character arrays ([Section 4.3.1, p. 134](#)) used to store and retrieve data.

其他未格式化 IO 操作一次处理数据块。如果速度是一个问题，这些操作可能就很重要。但像其他低级操作一样，它们是容易出错的。尤其是，这些操作要求我们分配和管理用于存储和检索数据的字符数组（[第 4.3.1 节](#)）。

The multi-byte operations are listed in [Table A.5](#) (p. [837](#)). It is worth noting that the `get` member is overloaded; there is a third version that reads a sequence of characters.

多字节操作在 [表 A.5](#) 列出。值得注意的是，`get` 成员是重载的，存在读字符序列的第三个版本。

Table A.5. Multi-Byte Low-Level IO Operations

表 A.5. 多字节低级 IO 操作

<code>is.get(sink, size, delim)</code>	Reads up to <code>size</code> bytes from <code>is</code> and stores them in the character array pointed to by <code>sink</code> . Reads until encountering the <code>delim</code> character or until it has read <code>size</code> bytes or encounters end-of-file. If the <code>delim</code> is present, it is left on the input stream and not read into <code>sink</code> .
<code>is.getline(sink, size, delim)</code>	从 <code>is</code> 中读 <code>size</code> 个字节并将它们存储到 <code>sink</code> 所指向的字符数组中。读操作直到遇到 <code>delim</code> 字符，或已经读入了 <code>size</code> 个字节，或遇到文件结束符才结束。如果出现了 <code>delim</code> ，就将它留在输入流上，不读入到 <code>sink</code> 中。
<code>is.read(sink, size)</code>	与三个实参的 <code>get</code> 行为类似，但读并丢弃 <code>delim</code>
<code>is.gcount()</code>	Returns number of bytes read from the stream <code>is</code> by last call to

Section A.3. The IO Library Revisited

an unformatted read operation.
返回最后一个未格式化读操作从流 <code>is</code> 中读到的字节数
<code>os.write(source, size)</code>
Writes <code>size</code> bytes from the character array <code>source</code> to <code>os</code> . Returns <code>os</code> .
将 <code>size</code> 个字从数组 <code>source</code> 写至 <code>os</code> 。返回 <code>os</code>
<code>is.ignore(size, delim)</code>
Reads and ignores at most <code>size</code> characters up to but not including <code>delim</code> . By default, <code>size</code> is 1 and <code>delim</code> is end-of-file.
读并忽略至多 <code>size</code> 个字符，直到遇到 <code>delim</code> ，但不包括 <code>delim</code> 。默认情况下， <code>size</code> 是 1 而 <code>delim</code> 是文件结束符

Caution: Low-Level Routines Are Error-Prone

警告：低级例程容易出错

In general, we advocate using the higher-level abstractions provided by the library. The IO operations that return `int` are a good example of why.

一般提倡使用标准库提供的高级抽象。返回 `int` 值的 IO 操作是一个很好的例子。

It is a common programming error to assign the return from `get` or one of the other `int` returning functions to a `char` rather than an `int`. Doing so is an error but an error the compiler will not detect. Instead, what happens depends on the machine and on the input data. For example, on a machine in which `char`s are implemented as `unsigned char`s, this loop will run forever:

将 `get` 或其他返回 `int` 值的函数的返回值赋给 `char` 对象而不是 `int` 对象，是常见的错误，但编译器不检测这样的错误，相反，发生什么取决于机器和输入数据。例如，在将 `char` 实现为 `unsigned char` 的机器上，这是一个死循环：

[View full width]

```
char ch; // Using a char here invites disaster!
// return from cin.get is converted from int to char and
→ then compared to an int
while ((ch = cin.get()) != EOF)
    cout.put(ch);
```

The problem is that when `get` returns `EOF`, that value will be converted to an `unsigned char` value. That converted value is no longer equal to the integral value of `EOF`, and the loop will continue forever.

问题在于，当 `get` 返回 `EOF` 的时候，那个值将被转换为 `unsigned char` 值，转换后的值不再等于 `EOF` 的整型值，循环将永远继续。

At least that error is likely to be caught in testing. On machines for which `char`s are implemented as `signed char`s, we can't say with confidence what the behavior of the loop might be. What happens when an out-of-bounds value is assigned to a `signed` value is up to the compiler. On many machines, this loop will appear to work, unless a character in the input matches the `EOF` value. While such characters are unlikely in ordinary data, presumably low-level IO is necessary only when reading binary values that do not map directly to ordinary characters and numeric values. For example, on our machine, if the input contains a character whose value is '`\377`' then the loop terminates prematurely. '`\377`' is the value on our machine to which -1 converts when used as a `signed char`. If the input has this value, then it will be treated as the (premature) end-of-file indicator.

至少还可能在测试中捕获这个错误。在将 `char` 实现为 `signed char` 的机器上，不能确切地说出该循环的行为。当将超出边界的值赋给 `signed` 值时发生什么由编译器负责。在许多机器上，这个循环看起来能工作，除非输入中的字符与 `EOF` 值匹配。虽然这样的字符不可能在普通数据中，但是，大概只有在读不直接映射到普通字符和数值的二进制值的时候，低级 IO 才是必要的。例如，在我们的机器上，如果输入包含值为 '`\377`' 的字符，循环就提前终止。`'\377'` 是我们的机器上 -1 作为 `signed char` 使用的时候所转换的值，如果输入中有这个值，就将它当作（提早的）文件结束指示符对待。

Such bugs do not happen when reading and writing typed values. If you can use the more type-safe, higher-level operations supported by the library, do so.

在读写类型化值的时候不会发生这样的错误。如果可以使用标准库支持的更为类型安全的、更高级的操作，就使用。

The `get` and `getline` functions take the same parameters, and their actions are similar but not identical. In each case, `sink` is a `char` array into which the data are placed. The functions read until one of the following conditions occurs:

Section A.3. The IO Library Revisited

`get` 函数和 `getline` 函数接受相同形参，它们的行为类似但不相同。在每个函数中，`sink` 是一个存放数据的 `char` 数组。函数进行读，直到下面条件中的一个发生：

- `size - 1` characters are read
读到了 `size - 1` 个字符。
- End-of-file is encountered
遇到文件结束符。
- The delimiter character is encountered
遇到分隔符。

Following any of these conditions, a null character is put in the next open position in the array. The difference between these functions is the treatment of the delimiter. `get` leaves the delimiter as the next character of the `istream`. `getline` reads and discards the delimiter. In either case, the delimiter is *not* stored in `sink`.

遵循这些条件中的任意一个，将一个空字符放到数组中下一个开放位置。两个函数之间的区别是对待分隔符的方法。`get` 将分隔符留作 `istream` 的下一个字符，`getline` 读入并丢弃分隔符，两种情况下，都不在 `sink` 中存储分隔符。



It is a common error to intend to remove the delimiter from the stream but to forget to do so.

想要从流中移去分隔符但忘了这样做，是一个常见的错误。

Determining How Many Characters Were Read

确定读了多少字符

Several of the read operations read an unknown number of bytes from the input. We can call `gcount` to determine how many characters the last unformatted input operation read. It is essential to call `gcount` before any intervening unformatted input operation. In particular, the single-character operations that put characters back on the stream are also unformatted input operations. If `peek`, `unget`, or `putback` are called before calling `gcount`, then the return value will be 0!

有几个读操作中从输入中读未知数目的字节。可以调用 `gcount` 操作来确定最后一个未格式化输入操作读了多少字符。有必要在任意插入的未格式化输入操作之前调用 `gcount`。尤其是，将字符放回流上的单字符操作也是未格式化输入操作，如果在调用 `gcount` 之前调用 `peek`、`unget` 或 `putback`，则返回值将是 0！

A.3.8. Random Access to a Stream

A.3.8. 流的随机访问

The various stream types generally support random access to the data in their associated stream. We can reposition the stream so that it skips around, reading first the last line, then the first, and so on. The library provides a pair of functions to `seek` to a given location and to `tell` the current location in the associated stream.

不同的流类型一般支持对相关流中数据的随机访问。可以重新定位流，以便环绕跳过，首先读最后一行，再读第一行，以此类推。标准库提供一对函数定位给定位置并告诉 (`tell`) 相关流中的当前位置。



Random IO is an inherently system-dependent. To understand how to use these features, you must consult your system's documentation.

随机 IO 是一个固有的随系统而定的特征。要理解怎样使用这些特征，必须查阅系统的文档。

Seek and Tell Functions

seek 和 tell 函数

To support random access, the IO types maintain a marker that determines where the next read or write will happen. They also provide two functions: One repositions the marker by *seeking* to a given position; the second *tells* us the current position of the marker. The library actually defines two pairs of *seek* and *tell* functions, which are described in [Table A.6](#). One pair is used by input streams, the other by output streams. The input and output versions are distinguished by a suffix that is either a `g` or a `p`. The `g` versions indicate that we are "getting" (reading) data, and the `p` functions indicate that we are "putting" (writing) data.

为了支持随机访问，IO类型维持一个标记，该标记决定下一个读或写发生在哪。IO类型还提供两个函数：一个通过 `seek` 指定位置重新安置该标记，另一个 `tell` 我们标记的当前位置。标准库实际上定义了两对 `seek` 和 `tell` 函数（表 A.6 给出对它们的描述），一对由输入流使用，另一对由输出流使用。输入和输出版本由后缀 `g` 和 `p` 区分，`g` 版本指出正在“获取”（读）数据，`p` 函数指出正在“放置”（写）数据。

Table A.6. Seek and Tell Functions

表 A.6. seek 和 tell 函数

<code>seekg</code>	Reposition the marker in an input stream 重新定位输入流中的标记
<code>tellg</code>	Return the current position of the marker in an input stream 返回输入流中标记的当前位置
<code>seekp</code>	Reposition the marker for an output stream 重新定位输出流中的标记
<code>tellp</code>	Return the current position of the marker in an output stream 返回输出流中标记的当前位置

Logically enough, we can use only the `g` versions on an `istream` or its derived types `ifstream`, or `istringstream`, and we can use only the `p` versions on an `ostream` or its derived types `ofstream`, and `ostringstream`. An `iostream`, `fstream`, or `stringstream` can both read and write the associated stream; we can use either the `g` or `p` versions on objects of these types.

逻辑上，只能在 `istream` 类型或其派生类型 `ifstream` 或 `istringstream` 之上使用 `g` 版本，并且只能在 `ostream` 类型或其派生类型 `ofstream` 或 `ostringstream` 之上使用 `p` 版本。`iostream` 对象、`fstream` 对象或 `stringstream` 对象对相关流既能读又能写，可以在这些类型的对象上使用 `g` 或 `p` 版本的任何一个。

There Is Only One Marker

只有一个标记

The fact that the library distinguishes between the "putting" and "getting" versions of the `seek` and `tell` functions can be misleading. Even though the library makes this distinction, it maintains only a single marker in the file there is *not* a distinct read marker and write marker.

标准库区分 `seek` 函数和 `tell` 函数的“放置”和“获取”版本，这一事实可能会令人误解。虽然标准库进行这种区分，但它只在文件中维持一个标记——没有可区分的读标记和写标记。

When we're dealing with an input-only or output-only stream, the distinction isn't even apparent. We can use only the `g` or only the `p` versions on such streams. If we attempt to call `tellp` on an `ifstream`, the compiler will complain. Similarly, it will not let us call `seekg` on an `ostringstream`.

处理只输入或只输出的流的时候，区别并不明显。在这样的流上，只能使用 `g` 版本或只能使用 `p` 版本。如果 `ifstream` 对象上调用 `tellp`，编译器将会给出错误提示。类似地，编译器不允许在 `ostringstream` 对象上调用 `seekg`。

When using the `fstream` and `stringstream` types that can both read and write, there is a single buffer that holds data to be read and written and a single marker denoting the current position in the buffer. The library maps both the `g` and `p` positions to this single marker.

使用既能读又能写的 `fstream` 类型和 `stringstream` 类型的时候，只有一个保存数据的缓冲区和一个表示缓冲区中当前位置的标记，标准库将 `g` 位置和 `p` 位置都映射到这个标记。

Because there is only a single marker, we *must* do a `seek` to reposition the marker whenever we switch



between reading and writing.

因为只有一个标记，所以，在读和写之间切换时必须进行 `seek` 来重新定位标记。

Plain `iostreams` Usually Do Not Allow Random Access

普通 `iostream` 对象一般不允许随机访问

The `seek` and `tell` functions are defined for all the stream types. Whether they do anything useful depends on the kind of object to which the stream is bound. On most systems, the streams bound to `cin`, `cout`, `cerr` and `clog` do not support random access after all, what would it mean to jump ten places back when writing directly to `cout`? We can call the `seek` and `tell` functions, but these functions will fail at run time, leaving the stream in an invalid state.

`seek` 函数和 `tell` 函数是为所有流类型定义的，它们是否完成有用的工作取决于流所绑定的对象的类别。在大多数系统上，绑定到 `cin`、`cout`、`cerr` 和 `clog` 的流不支持随机访问——直接写 `cout` 的时候，回跳 10 个位置会意味着什么？可以调用 `seek` 函数和 `tell` 函数，但这些函数将在运行时失败，使得流处于无效状态。



Because the `istream` and `ostream` types usually do not support random access, the remainder of this section should be considered as applicable to only the `fstream` and `sstream` types.

因为 `istream` 和 `ostream` 类型一般不支持随机访问，所以，应该认为本节其余部分只能应用于 `fstream` 和 `sstream` 类型。

Repositioning the Marker

重新定位标记

The `seekg` and `seekp` functions are used to change the read and write positions in a file or a `string`. After a call to `seekg`, the read position in the stream is changed; a call to `seekp` sets the position at which the next write will take place.

`seekg` 函数和 `seekp` 函数用于改变文件或 `string` 对象中的读写位置。调用 `seekg` 函数之后，改变流中的读位置，`seekp` 函数调用将位置置于下一个写将发生的地方。

There are two versions of the seek functions: One moves to an "absolute" address within the file; the other moves to a byte offset from a given position:

`seek` 函数有两个版本：一个移动到文件中的一个“绝对”地址，另一个移动到给定位置的字节偏移量处：

```
// set the indicated marker a fixed position within a file or string
seekg(new_position); // set read marker
seekp(new_position); // set write marker

// offset some distance from the indicated position
seekg(offset, dir); // set read marker
seekp(offset, dir); // set write marker
```

The first version sets the current position to a given location. The second takes an offset and an indicator of where to offset from. The possible values for the offset are listed in [Table A.7](#).

第一个版本将当前位置置于给定地点，第二个版本接受一个偏移量以及从何处偏移的指示器。偏移量的可能值在 [表 A.7](#) 中列出。

Table A.7. Offset From Argument to `seek`

表 A.7. `seek` 实参的偏移量

<code>beg</code>	The beginning of the stream 流的开头
<code>cur</code>	The current position of the stream 流的当前位置

`end`

The end of the stream

流的末尾

The argument and return types for these functions are machine-dependent types defined in both `istream` or `ostream`. The types, named `pos_type` and `off_type`, represent a file position and an offset from that position, respectively. A value of type `off_type` can be positive or negative; we can `seek` forward or backward in the file.

这些函数的实参类型和返回类型是与机器相关的类型，在 `istream` 和 `ostream` 中定义。名为 `pos_type` 和 `off_type` 的类型分别表示文件位置和从该位置的偏移量。`off_type` 类型的值可以为正也可以为负，在文件中可以进行前向或后向 `seek`。

Accessing the Marker

访问标记

The current position is returned by either `tellg` or `tellp`, depending on whether we're looking for the read or write position. As before, the `p` indicates putting (writing) and the `g` indicates getting (reading). The `tell` functions are usually used to remember a location so that we can subsequently `seek` back to it:

当前位置由 `tellg` 或 `tellp` 返回，取决于正在查找读位置还是写位置。像前面一样，`p` 指出放置（写）而 `g` 指出获取（读）。`tell` 函数一般用于记住一个位置，以便随后可以通过 `seek` 回到那里：

```
// remember current write position in mark
ostringstream writeStr; // output stringstream
ostringstream::pos_type mark = writeStr.tellp();
// ...
if (cancelEntry)
    // return to marked position
    writeStr.seekp(mark);
```

The `tell` functions return a value that indicates the position in the associated stream. As with the `size_type` of a `string` or `vector`, we do not know the actual type of the object returned from `tellg` or `tellp`. Instead, we use the `pos_type` member of the appropriate stream class.

`tell` 函数返回一个值，指出相关流中的位置。像 `string` 对象或 `vector` 对象的 `size_type` 一样，我们不知道从 `tellg` 或 `tellp` 返回的对象实际类型，而是使用适当流类的 `pos_type` 成员来代替。

A.3.9. Reading and Writing to the Same File

A.3.9. 读写同一文件

Let's look at a programming example. Assume we are given a file to read. We are to write a new line at the end of the file that contains the relative position at which each line begins. For example, given the following file,

看一个程序示例，假定给定一个文件来读，我们将在文件的末尾写一个新行，该行包含每一行开头的相对位置。例如，给定下面的文件，

```
abcd
efg
hi
j
```

the program should produce the following modified file:

这段程序应产生修改后文件如下：

```
abcd
efg
hi
j
5 9 12 14
```

Note that our program need not write the offset for the first line it always occurs at position 0. It should print the offset that corresponds to the end of the data portion of the file. That is, it should record the position after the end of the input so that we'll know where the original data ends and where our output begins.

注意，程序不必写第一行的偏移量——它总是出现在位置 0，程序应该显示对应于文件数据部分末尾的偏移量，也就是说，它应该记录输入末尾之后的位置，以便我们知道原始数据在何处结束以及我们的输出从何处开始。

We can write this program by writing a loop that reads a line at a time:

通过编写一次读一行的循环可以编写这个程序：

```
int main()
{
```

Section A.3. The IO Library Revisited

```
// open for input and output and pre-position file pointers to end of file
fstream inout("copyOut",
             fstream::ate | fstream::in | fstream::out);
if (!inout) {
    cerr << "Unable to open file!" << endl;
    return EXIT_FAILURE;
}
// inout is opened in ate mode, so it starts out positioned at the end,
// which we must remember as it is the original end-of-file position
ifstream::pos_type end_mark = inout.tellg();
inout.seekg(0, fstream::beg); // reposition to start of the file
int cnt = 0; // accumulator for byte count
string line; // hold each line of input
// while we haven't hit an error and are still reading the original data
// and successfully read another line from the file
while (inout && inout.tellg() != end_mark
       && getline(inout, line))
{
    cnt += line.size() + 1; // add 1 to account for the newline
    // remember current read marker
    ifstream::pos_type mark = inout.tellg();
    inout.seekp(0, fstream::end); // set write marker to end
    inout << cnt; // write the accumulated length
    // print separator if this is not the last line
    if (mark != end_mark) inout << " ";
    inout.seekg(mark); // restore read position
}
inout.clear(); // clear flags in case we hit an error
inout.seekp(0, fstream::end); // seek to end
inout << "\n"; // write a newline at end of file
return 0;
}
```

This program opens the `fstream` using the `in`, `out`, and `ate` modes. The first two modes indicate that we intend to both read and write to the same file. By also opening it in `ate` mode, the file starts out positioned at the end. As usual, we check that the open succeeded, and exit if it did not.

这个程序使用 `in`、`out` 和 `ate` 模式打开 `fstream`。前两种模式指出，我们打算对同一文件进行读写；通过用 `ate` 模式打开，文件首先定位到末端。照常，要检查打开是否成功，如果不成功就退出。

Initial Setup

初始设置

The core of our program will loop through the file a line at a time, recording the relative position of each line as it does so. Our loop should read the contents of the file up to but not including the line that we are adding to hold the line offsets. Because we will be writing to the file, we can't just stop the loop when it encounters end-of-file. Instead, the loop should end when it reaches the point at which the original input ended. To do so, we must first remember the original end-of-file position.

这段程序的核心部分将循环通过文件，一次一行，循环过程中记录每一行的相对位置。循环应该读文件的内容，直到但不包含正在增加的保存行偏移量的那一行。因为将对文件进行写，所以不是在遇到文件结束符时停止循环，相反，应在到达原始输入结束处结束循环。要做到这一点，必须首先记住原来文件结束符的位置。

We opened the file in `ate` mode, so it is already positioned at the end. We store the initial end position in `end_mark`. Of course, having remembered the end position, we must reposition the read marker at the beginning of the file before we attempt to read any data.

因为是以 `ate` 模式打开文件，所以文件已经定位在末端。将原来的末端位置存储在 `end_mark` 中。当然，记住了结束位置之后，在试图读任意数据之前，必须将读标记重新定位于文件开头。

Main Processing Loop

主处理循环

Our `while` loop has a three-part condition.

`while` 循环有三部分条件。

We first check that the stream is valid. Assuming the first test on `inout` succeeds, we then check whether we've exhausted our original input. We do this check by comparing the current read position returned from `tellg` with the position we remembered in `end_mark`. Finally, assuming that both tests succeeded, we call `getline` to read the next line of input. If `getline` succeeds, we perform the body of the loop.

首先检查流是否有效。假定第一个测试 `inout` 成功，接着检查是否已经耗尽了原始输入，通过将从 `tellg` 返回的当前读位置与 `end_mark` 中记录的位置相比较，进行这个检查。最后，假定两个测试都成功，就调用 `getline` 来读下一行输入，如果 `getline` 成功，就执行循环体。

The job that the `while` does is to increment the counter to determine the offset at which the next line starts and write that marker at the end of

Section A.3. The IO Library Revisited

the file. Notice that the end of the file advances on each trip through the loop.

`while` 所做的工作是将计数器增量，以确定下一行开始处的偏移量，并将那个标记写至文件末尾。注意，每通过一次循环都向前推进文件的末尾。

We start by remembering the current position in `mark`. We need to keep that value because we have to reposition the file in order to write the next relative offset. The `seekp` call does this repositioning, resetting the file pointer to the end of the file. We write the counter value and then restore the file position to the value we remembered in `mark`. The effect is that we return the marker to the same place it was after the last read. Having restored the marker, we're ready to repeat the condition in the `while`.

首先将当前位置记在 `mark` 中，需要保存那个值，是因为要写下一个相对偏移量必须重新定位文件。`seekp` 的调用进行这个重新定位，将文件指针重置到文件末尾。写计数器值然后将文件位置恢复为 `mark` 中所记录的值，效果是将标记返回到最后一次读之后的同一地方。恢复了标记之后，就准备重复检测 `while` 中的条件。

Completing the File

完成文件

Once we exit the loop, we have read each line and calculated all the starting offsets. All that remains is to print the offset of the last line. As with the other writes, we call `seekp` to position the file at the end and write the value of `cnt`. The only tricky part is remembering to `clear` the stream. We might exit the loop due to an end-of-file or other input error. If so, `inOut` would be in an error state, and both the `seekp` and the output expression would fail.

一旦退出了循环，就已经读过了每一行并计算了所有行开头的偏移量。剩下的是显示最后一行的偏移量。像对其他写操作一样，调用 `seekp` 将文件定位在末尾，并写 `cnt` 的值。唯一复杂的部分是记得对流进行 `clear`。可能因为文件结束符或其他输入错误而退出循环，如果是这样，`inOut` 将处于错误状态，而 `seekp` 和输出表达式都将失败。

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[!](#) (logical NOT)

[!=](#) (inequality) 2nd

[container](#)

[container adaptor](#)

[iterator](#) 2nd

[string](#)

[#define](#)

[#ifdef](#)

[#ifndef](#)

[#include](#)

[%](#) (modulus)

[&](#) (address-of) 2nd 3rd

[&](#) (bitwise AND) 2nd

[Query](#)

[&&](#) (logical AND)

[operand order of evaluation](#)

[overloaded operator](#)

[\(\)](#) (call operator) 2nd 3rd 4th

[overloaded operator](#)

[* \(dereference\) 2nd 3rd 4th](#)

[iterator](#)

[on map yields pair](#)

[overloaded operator](#)

[pointer](#)

[yields lvalue](#) 2nd

[* \(multiplication\)](#)

[+ \(addition\)](#)

[iterator](#) 2nd

[pointer](#)

[Sales item](#)

[string](#)

[+ \(unary plus\)](#)

[++ \(increment\) 2nd 3rd 4th](#)

[and dereference](#)

[iterator](#) 2nd 3rd

[overloaded operator](#)

[pointer](#)

[prefix yields lvalue](#)

[reverse iterator](#)

[+= \(compound assignment\) 2nd 3rd](#)

[iterator](#)

[overloaded operator](#)

[Sales item](#)

[string](#)

[, \(comma operator\) 2nd](#)

[example](#)

[operand order of evaluation](#)

[overloaded operator](#)

[- \(subtraction\)](#)

SYMBOL

[iterator](#) 2nd
[pointer](#)
- (unary minus)
-- (decrement)
and dereference
iterator
[overloaded operator](#)
prefix yields lvalue
reverse iterator
->* (pointer to member arrow)
... (ellipsis parameter)
[.c file](#)
[.cc file](#)
[.cp file](#)
[.cpp file](#)
[.h file](#)
/ (division)
/* */ (block comment) 2nd
// (single-line comment) 2nd
> (arrow operator)
 class member access
 overloaded operator
:: (scope operator) 2nd 3rd 4th
 base class members
 class member 2nd
 container defined type
 member function definition
 to override name lookup
; (semicolon)
 class definition
< (less-than) 2nd
 overloaded and containers
 used by algorithm
<< (left-shift) 2nd
<< (output operator) 2nd
 bitset
 formatting
 ostream iterator
 overloaded operator
 must be nonmember
 precedence and associativity
 Sales item
 string 2nd
<= (less-than-or-equal) 2nd 3rd
= (assignment) 2nd 3rd
 and conversion
 and equality
 class assignment operator
 container
 overloaded operator 2nd
 and copy constructor
 check for self-assignment
 Message
 multiple inheritance
 reference return 2nd
 rule of three
 use counting 2nd

SYMBOL

[valuelike classes](#)
[pointer](#)
[string](#)
[to signed](#)
[to unsigned](#)
[yields lvalue](#)
[== \(equality\) 2nd](#)
[algorithm](#)
[container](#)
[container adaptor](#)
[iterator 2nd](#)
[string 2nd](#)
[> \(greater-than\) 2nd](#)
[>= \(greater-than-or-equal\) 2nd](#)
[>> \(input operator\) 2nd](#)
[istream iterator](#)
[overloaded operator](#)
 must be nonmember
[precedence and associativity](#)
[Sales item.](#)
[string 2nd](#)
[>> \(right-shift\) 2nd](#)
[?: \(conditional operator\) 2nd](#)
 operand order of evaluation
[\[\] \(subscript\) 2nd 3rd](#)
 and multi-dimensioned array
 and pointer
 array
[bitset](#)
[deque](#)
[map](#)
[overloaded operator](#)
 reference return
[string](#)
[valid subscript range](#)
[vector 2nd](#)
[yields lvalue](#)
[\0 \(null character\)](#)
[\n \(newline character\) 2nd](#)
[\nnn \(octal escape sequence\)](#)
[\t \(tab character\) 2nd](#)
[\xnnn \(hexadecimal escape sequence\) 2nd](#)
[^ \(bitwise XOR\) 2nd](#)
 cplusplus
 DATE
 FILE
 LINE
 TIME
[{} \(curly brace\) 2nd](#)
[| \(bitwise OR\) 2nd](#)
 example
 Query
[|| \(logical OR\)](#)
 operand order of evaluation
 overloaded operator
[~ \(bitwise NOT\) 2nd](#)
 Query

SYMBOL

~classname [See [destructor](#)]

Team LiB

[◀ PREVIOUS](#) [NEXT ▶](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[!](#) (logical NOT)

[!=](#) (inequality) 2nd

[container](#)

[container adaptor](#)

[iterator](#) 2nd

[string](#)

[#define](#)

[#ifdef](#)

[#ifndef](#)

[#include](#)

[%](#) (modulus)

[&](#) (address-of) 2nd 3rd

[&](#) (bitwise AND) 2nd

[Query](#)

[&&](#) (logical AND)

[operand order of evaluation](#)

[overloaded operator](#)

[\(\)](#) (call operator) 2nd 3rd 4th

[overloaded operator](#)

[*](#) (dereference) 2nd 3rd 4th

[iterator](#)

[on map yields pair](#)

[overloaded operator](#)

[pointer](#)

[yields lvalue](#) 2nd

[*](#) (multiplication)

[+](#) (addition)

[iterator](#) 2nd

[pointer](#)

[Sales item](#)

[string](#)

[+ \(unary plus\)](#)

[++](#) (increment) 2nd 3rd 4th

[and dereference](#)

[iterator](#) 2nd 3rd

[overloaded operator](#)

[pointer](#)

[prefix yields lvalue](#)

[reverse iterator](#)

[+=](#) (compound assignment) 2nd 3rd

[iterator](#)

[overloaded operator](#)

[Sales item](#)

[string](#)

[,](#) (comma operator) 2nd

[example](#)

[operand order of evaluation](#)

[overloaded operator](#)

[-](#) (subtraction)

SYMBOL

[iterator](#) 2nd
[pointer](#)
- (unary minus)
-- (decrement)
and dereference
iterator
overloaded operator
prefix yields lvalue
reverse iterator
->* (pointer to member arrow)
... (ellipsis parameter)
.c file
.cc file
.cp file
.cpp file
.h file
/ (division)
/* */ (block comment) 2nd
// (single-line comment) 2nd
> (arrow operator)
 class member access
 overloaded operator
:: (scope operator) 2nd 3rd 4th
 base class members
 class member 2nd
 container defined type
 member function definition
 to override name lookup
; (semicolon)
 class definition
< (less-than) 2nd
 overloaded and containers
 used by algorithm
<< (left-shift) 2nd
<< (output operator) 2nd
 bitset
 formatting
 ostream iterator
 overloaded operator
 must be nonmember
precedence and associativity
Sales item
string 2nd
<= (less-than-or-equal) 2nd 3rd
= (assignment) 2nd 3rd
 and conversion
 and equality
 class assignment operator
 container
 overloaded operator 2nd
 and copy constructor
 check for self-assignment
 Message
 multiple inheritance
 reference return 2nd
 rule of three
 use counting 2nd

SYMBOL

[valuelike classes](#)
[pointer](#)
[string](#)
[to signed](#)
[to unsigned](#)
[yields lvalue](#)
[== \(equality\) 2nd](#)
[algorithm](#)
[container](#)
[container adaptor](#)
[iterator 2nd](#)
[string 2nd](#)
[> \(greater-than\) 2nd](#)
[>= \(greater-than-or-equal\) 2nd](#)
[>> \(input operator\) 2nd](#)
[istream iterator](#)
[overloaded operator](#)
 must be nonmember
[precedence and associativity](#)
[Sales item.](#)
[string 2nd](#)
[>> \(right-shift\) 2nd](#)
[?: \(conditional operator\) 2nd](#)
 operand order of evaluation
[\[\] \(subscript\) 2nd 3rd](#)
 and multi-dimensioned array
 and pointer
 array
 bitset
 deque
 map
 overloaded operator
 reference return
 string
 valid subscript range
 vector 2nd
 yields lvalue
[\0 \(null character\)](#)
[\n \(newline character\) 2nd](#)
[\nnn \(octal escape sequence\)](#)
[\t \(tab character\) 2nd](#)
[\xnnn \(hexadecimal escape sequence\) 2nd](#)
[^ \(bitwise XOR\) 2nd](#)
 cplusplus
 DATE
 FILE
 LINE
 TIME
[{} \(curly brace\) 2nd](#)
[| \(bitwise OR\) 2nd](#)
 example
 Query
[|| \(logical OR\)](#)
 operand order of evaluation
 overloaded operator
[~ \(bitwise NOT\) 2nd](#)
 Query

SYMBOL

~classname [See [destructor](#)]

Team LiB

[◀ PREVIOUS](#) [NEXT ▶](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

[abnormal termination, stream buffers](#) (异常终止, 流缓冲区)

[abort](#) 2nd

[absInt](#)

[abstract base class](#) (抽象基类) 2nd

[example](#) (例子)

[abstract data type](#) (抽象数据类型) 2nd 3rd

[abstraction, data](#) (抽象, 数据) 2nd

[access control](#) (访问控制)

[in base and derived classes](#) (在基类和派生类中)

[local class](#) (局部类)

[nested class](#) (嵌套类)

[using declarations to adjust](#) ([using](#) 声明来调整)

[access label](#) (访问标号) 2nd 3rd 4th

[private](#) 2nd

[protected](#) 2nd

[public](#) 2nd

[Account](#)

[accumulate](#) 2nd

[Action](#)

[adaptor](#) (适配器) 2nd

[container](#) (容器)

[function](#) (函数) 2nd 3rd

[iterator](#) (迭代器)

[addition \(+\)](#) (加)

[iterator](#) (迭代器) 2nd

[pointer](#) (指针)

[Sales item](#)

[string](#)

[address](#) (地址) 2nd

[address-of \(&\)](#) (取地址) 2nd

[overloaded operator](#) (重载操作符)

[adjacent difference](#)

[adjacent find](#)

[algorithm](#) (算法) 2nd

[copy versions](#) ([copy](#) 版本) 2nd

[if versions](#) ([if](#) 版本)

[element type constraints](#) (元素类型约束)

[independent of container](#) (独立于容器)

[iterator argument constraints](#) (迭代器实参约束) 2nd

[iterator category and](#) (迭代器种类和) 2nd

[naming convention](#) (命名规范)

[overloaded versions](#) (重载版本)

[parameter pattern](#) (形参模式)

[passing comparison function](#)

[read-only](#) (只读)

[structure](#) (结构)

[that reorders elements](#) (对元素进行排序的算法)

[that writes elements](#) (写元素的算法)

[type independence \(类型独立性\) 2nd](#)[using function object as argument \(用函数对象作为实参\)](#)[with two input ranges \(带两个输入范围\)](#)[algorithm header \(algorithm 头文件\)](#)[algorithms](#)[binary](#)[library defined](#)[overloaded operator](#)[example](#)[unary](#)[alias, namespace \(别名、命名空间\) 2nd](#)[allocator 2nd 3rd](#)[allocate](#)[compared to operator new \(与 operator new 比较\)](#)[construct 2nd](#)[compared to placement new \(与定位 new 表达式比较\)](#)[deallocate](#)[compared to operator delete \(与 operator delete 比较\)](#)[destroy 2nd](#)[compared to calling destructor \(与调用析构函数比较\)](#)[operations \(操作\)](#)[alternative operator name \(可选择的操作符名字\)](#)[ambiguous \(二义性\)](#)[conversion \(转换\)](#)[multiple inheritance \(多重继承\)](#)[function call \(调用\) 2nd 3rd](#)[multiple base classes \(多个基类\)](#)[overloaded operator \(重载操作符\)](#)[AndQuery](#)[definition \(定义\)](#)[eval function \(eval 函数\)](#)[anonymous union \(匿名联合\) 2nd](#)[app \(file mode\) \(文件模式\)](#)[append, string](#)[arc](#)[argument \(实参\) 2nd 3rd 4th 5th](#)[array type \(数组类型\)](#)[C-style string \(C 风格字符串\)](#)[const reference type \(const 引用类型\)](#)[conversion \(转换\)](#)[with class type conversion \(用类类型转换\)](#)[copied \(复制的\)](#)[uses copy constructor \(使用复制构造函数\)](#)[default \(默认\)](#)[iterator \(迭代器\) 2nd](#)[multi-dimensioned array \(多维数组\)](#)[nonconst reference parameter](#)[passing \(传递\)](#)[pointer to const \(指向 const 对象的指针\)](#)[pointer to nonconst \(指向非 const 对象的指针\)](#)[reference parameter \(引用形参参数\)](#)[template \[See template argument\]](#)[to main \(main 的\)](#)[to member function \(成员函数的\)](#)[type checking \(类型检查\)](#)[ellipsis \(省略符\)](#)[of array type](#)

(数组类型的)

of reference to array (数组引用的)

with class type conversion (用类类型转换的)

argument deduction, template (模板实参推断)

argument list (实参表)

argv

arithmetic (算术)

iterator (迭代器) 2nd 3rd 4th

pointer (指针) 2nd

arithmetic operator (算术操作符)

and compound assignment (和复合赋值)

function object (函数对象)

overloaded operator (重载操作符)

arithmetic type (算术类型) 2nd

conversion (转换) 2nd

from bool (从 bool 转换)

signed to unsigned (signed 到 unsigned)

conversion to bool (转换到 bool)

array (数组) 2nd 3rd

and pointer (和指针)

argument (实参)

as initializer of vector (作为 vector 的初始化式)

assignment (赋值)

associative (关联数组)

conversion to pointer (转换到指针) 2nd

and template argument (和模板实参)

copy (复制)

default initialization (默认初始化)

uses copy constructor (使用复制构造函数)

uses default constructor (使用默认构造函数)

definition (定义)

elements and destructor (元素与析构函数)

function returning (函数返回)

initialization (初始化)

multi-dimensioned (多维数组)

and pointer (和指针)

definition (定义)

initialization (初始化)

parameter (形参)

subscript operator (下标操作符)

of char initialization (char (数组) 的初始化)

parameter (形参)

buffer overflow (缓冲区溢出)

convention (规范)

reference type (引用类型)

size calculation (大小计算)

subscript operator (下标操作符)

arrow operator (>) (箭头操作符)

auto_ptr

class member access (类成员访问)

generic handle (泛型句柄)

overloaded operator (重载操作符)

assert preprocessor macro (`assert` 预处理器宏) 2nd

assign

container (容器)

string

assignment (赋值)

memberwise (逐个成员) 2nd
vs. initialization (与初始化)
assignment (=) (赋值) 2nd 3rd 4th
and conversion (和转换)
and copy constructor (和复制构造函数)
check for self-assignment (检查自身赋值)
container (容器)
for derived class (派生类的)
Message
multiple inheritance (多重继承)
overloaded operator (重载操作符) 2nd 3rd
reference return (引用返回) 2nd
pointer (指针)
rule of three (三法则)
exception for virtual destructors (虚析构函数的异常)
string
synthesized (合成赋值) 2nd
to base from derived (从派生类到基类 (赋值))
to signed (给 signed (赋值))
to unsigned (给 unsigned (赋值))
use counting (使用计数) 2nd
usually not virtual (通常不为虚)
value-like classes (值型类)
yields lvalue (生成左值)
associative array (关联数组) [See map]
associative container (关联容器) 2nd
assignment (=) (赋值)
begin
clear
constructors (构造函数)
count
element type constraints (元素类型约束) 2nd
empty
equal_range
erase
find
insert
key type constraints (键类型约束)
lower_bound
operations (操作)
overriding the default comparison (覆盖默认比较)
rbegin
rend
returning an (返回一个关联容器)
reverse iterator
size
supports relational operators (支持关系操作符)
swap
types defined by (关联容器定义的类型)
upper_bound
associativity (结合性) 2nd 3rd
overloaded operator (重载操作符)
at
deque
vector
ate (file mode) ate

(文件模式)

[auto_ptr](#) [2nd](#)

[constructor](#) (构造函数)

[copy and assignment](#) (复制与赋值)

[default constructor](#) (默认构造函数)

[get member](#) ([get](#) 成员)

[operations](#) (操作)

[pitfalls](#) (缺陷)

[reset member](#) ([reset](#) 成员)

[self-assignment](#) (自身赋值)

[automatic object](#) (自动对象) [2nd](#) [See also [local variable](#), [parameter](#)]

[and destructor](#) (和析构函数)

Team LiB

◀ PREVIOUS NEXT ▶

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[back](#)

[queue](#)

[sequential container](#)

[back inserter](#) 2nd 3rd

[bad](#)

[bad alloc](#) 2nd

[bad cast](#) 2nd

[bad typeid](#)

[badbit](#)

[base](#)

[base class](#) 2nd 3rd 4th [See also [virtual function](#)]

[abstract](#) 2nd

[example](#)

[access control](#) 2nd

[assignment operator, usually not virtual](#)

[can be a derived class](#)

[constructor](#)

[calls virtual function](#)

[not virtual](#)

[conversion from derived](#)

[access control](#)

[definition](#)

[destructor](#)

[calls virtual function](#)

[usually virtual](#)

[friendship not inherited](#)

[handle class](#)

[member hidden by derived](#)

[member operator delete](#)

[multiple](#) [See [multiple base class](#)]

[must be complete type](#)

[no conversion to derived](#)

[object initialized or assigned from derived](#)

[scope](#)

[static members](#)

[user](#)

[virtual](#) [See [virtual base class](#)]

[Basket](#)

[total function](#)

[Bear](#)

[as virtual base](#)

[begin](#)

[container](#)

[map](#)

[set](#)

[vector](#)

[best match](#) 2nd [See also [function matching](#)]

[bidirectional iterator](#) 2nd

[container](#)

[list](#)

[map](#)
[set](#)
[binary \(file mode\)](#)
[binary function object](#)
[binary operator 2nd](#)
[binary search](#)
[BinaryQuery](#)
 [definition](#)
[bind1st](#)
[bind2nd](#)
[binder 2nd](#)
[binding, dynamic 2nd](#)
 [requirements for](#)
[bit-field 2nd](#)
 [access to](#)
[bitset 2nd 3rd](#)
 [any](#)
 [compared to bitwise operator](#)
 [constructor](#)
 [count](#)
 [flip](#)
 [compared to bitwise NOT](#)
 [header](#)
 [none](#)
 [output operator](#)
 [reset](#)
 [set](#)
 [size](#)
 [subscript operator](#)
 [test](#)
 [to ulong](#)
[bitwise AND \(&\) 2nd](#)
 [example](#)
[bitwise exclusive or \(^\) 2nd](#)
[bitwise NOT \(~\) 2nd](#)
 [example](#)
[bitwise operator](#)
 [and compound assignment](#)
 [compared to bitset](#)
 [compound assignment](#)
 [example](#)
 [operand](#)
[bitwise OR \(|\) 2nd](#)
 [example 2nd](#)
[block 2nd 3rd 4th 5th](#)
 [as target of if](#)
 [function](#)
 [TRY 2nd 3rd 4th](#)
[block scope](#)
[body, function 2nd 3rd 4th](#)
[book finding program](#)
 [using equal range](#)
 [using find](#)
 [using upper bound](#)
[bookstore program](#)
 [exception classes](#)
[bool](#)
 [and equality operator](#)

[conversion to arithmetic type](#)

[literal](#)

[boolalpha manipulator](#)

[brace, curly 2nd](#)

[break statement 2nd](#)

[and switch](#)

[buffer 2nd](#)

[flushing](#)

[buffer overflow](#)

[and C-style string](#)

[array parameter](#)

[built-in type 2nd 3rd](#)

[class member default initialization](#)

[conversion](#)

[initialization of](#)

[Bulk_item](#)

[class definition](#)

[constructor](#)

[constructor using default arguments](#)

[derived from Disc_item](#)

[interface](#)

[member functions](#)

[byte 2nd](#)

Team LiB

◀ PREVIOUS NEXT ▶

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[C library header](#)

[C with classes](#)

C++

[calling C function from C++](#)

[compiling C and C++](#)

[using C linkage](#)

[C-style cast](#)

[C-style string](#) 2nd 3rd 4th 5th

[and char*](#)

[and string literal](#)

[compared to `string`](#) 2nd

[definition](#)

[dynamically allocated](#)

[initialization](#)

[parameter](#)

[pitfalls with generic programs](#)

[c_str](#)

[example](#)

[CachedObj](#)

[add to freelist](#)

[allocation explained](#)

[definition](#)

[definition of `static` members](#)

[design](#)

[illustration](#)

[inheriting from](#)

[operator delete](#)

[operator new](#)

[call operator \(\)](#) 2nd 3rd 4th

[execution flow](#)

[overloaded operator](#)

[calling C function from C++](#)

[candidate function](#) 2nd

[and function templates](#)

[namespaces](#)

[overloaded operator](#)

[capacity](#)

[string](#)

[vector](#)

[case label](#) 2nd 3rd

[default](#)

[cassert header](#)

[cast](#) 2nd

checked [See [dynamic_cast](#)]

[old-style](#)

[catch clause](#) 2nd 3rd 4th 5th

[catch\(...\)](#) 2nd

[example](#)

[exception specifier](#)

[matching](#)

[ordering of parameter](#)
[category iterator](#)
[cctype 2nd header](#)
[cerr 2nd](#)
[char literal](#)
char string literal [See [string literal](#)]

character

[newline \(\n\)](#)
[nonprintable 2nd](#)
[null \(\0\)](#)
[printable](#)
[tab \(\t\)](#)

checked cast [See [dynamic cast](#)]

[CheckedPtr](#)

[children's story program revisited](#)

[cin 2nd by default tied to cout](#)

[cl](#)

[class 2nd 3rd 4th 5th](#)

[abstract base](#)

[example](#)

[access labels 2nd](#)

[as friend](#)

[2nd 3rd \[See base class\]](#)

[concrete](#)

[conversion](#)

[multiple conversions lead to ambiguities](#)

[conversion constructor](#)

[function matching](#)

[with standard conversion](#)

[data member 2nd](#)

[const vs. mutable](#)

[const, initialization](#)

[constraints on type](#)

[definition](#)

[initialization](#)

[mutable](#)

[reference, initialization](#)

[static](#)

[data member definition](#)

[default access label](#)

[default inheritance access label](#)

[definition 2nd](#)

[and header 2nd](#)

[2nd 3rd \[See derived class\]](#)

[destructor definition](#)

direct base [See [immediate base class](#)]

[explicit constructor](#)

[forward declaration](#)

[generic handle 2nd](#)

[2nd \[See handle class\]](#)

[immediate base 2nd](#)

[indirect base 2nd](#)

local [See [local class](#)]

[member 2nd 3rd](#)

[member access](#)

[2nd \[See member function\]](#)

[member : constant expression \[See bit-field\]](#)

[multiple inheritance \[See multiple base class\]](#)

[nested \[See nested class\]](#)

[nonvirtual function, calls resolved at compile time](#)

[operator delete \[See member operator\]](#)

[operator new \[See member operator new\]](#)

[pointer member](#)

[copy control](#)

[copy control strategies](#)

[default copy behavior](#)

[pointer to member](#)

[definition](#)

[pointer to member function, definition](#)

[preventing copies](#)

[private member](#)

[private member](#)

[inheritance](#)

[protected member](#)

[public member 2nd](#)

[inheritance](#)

[static member 2nd](#)

[as default argument](#)

[data member as constant expression](#)

[example](#)

[inheritance](#)

[template member \[See member template\]](#)

[type member](#)

[undefined member](#)

[user 2nd](#)

[virtual base](#)

[virtual function, calls resolved at run time](#)

[class declaration 2nd](#)

[of derived class](#)

[class derivation list 2nd](#)

[access control](#)

[default access label](#)

[multiple base classes](#)

[virtual base](#)

[class keyword](#)

[class member : constant expression \[See bit-field\]](#)

[class scope 2nd 3rd](#)

[friend declaration](#)

[inheritance](#)

[member definition](#)

[name lookup](#)

[static members](#)

[virtual functions](#)

[class template 2nd 3rd 4th \[See also template parameter, template argument, instantiation\]](#)

[compiler error detection](#)

[declaration](#)

[definition](#)

[error detection](#)

[explicit template argument](#)

[export](#)

[friend](#)

[declaration dependencies](#)

[explicit template instantiation](#)

[nontemplate class or function](#)

[template class or function](#)

[member function](#)

[defined outside class body](#)

[instantiation](#)

[member specialization](#)

member template [See [member template](#)]

[nontype template parameter](#)

[static member](#)

[accessed through an instantiation](#)

[definition](#)

[type includes template argument\(s\)](#) 2nd

[type-dependent code](#)

[uses of template parameter](#)

class template specialization

[definition](#)

[member defined outside class body](#)

[member, declaration](#)

[namespaces](#)

[class type](#) 2nd 3rd 4th 5th 6th

[class member default initialization](#)

[conversion](#)

[design considerations](#)

[example](#)

[initialization of](#)

[multiple conversions lead to ambiguities](#)

[object definition](#)

[operator](#) 2nd 3rd

[operator and function matching](#)

[parameter and overloaded operator](#)

[used implicitly](#)

[variable vs. function declaration](#)

[with standard conversion](#)

[class, keyword](#)

[compared to](#) [typename](#)

[in template parameter](#)

[in variable definition](#)

cleanup, object [See [destructor](#)]

[clear](#) 2nd

[associative container](#)

[example](#) 2nd

[sequential container](#)

[close](#) 2nd

[close](#)

[comma operator \(, \)](#) 2nd

[example](#)

[operand order of evaluation](#)

[overloaded operator](#)

[comment](#) 2nd

[block \(/* * */ \)](#) 2nd

[single-line \(// \)](#) 2nd

[compare](#)

[plain function](#)

[string](#)

[template version](#)

[instantiated with pointer](#)

[specialization](#)

compilation

[and header](#)

[conditional](#)
[inclusion model for templates](#)
[needed when class changes](#)
[needed when inline function changes](#)
[separate 2nd
of templates](#)
[separate model for templates](#)

compiler
[extension](#)
[flag for inclusion compilation model](#)
[GNU](#)
[Microsoft](#)
[template errors diagnosed at link time](#)

compiler extension
[compiling C and C++](#)
composition vs. inheritance
[compound assignment \(e.g., +=\) 2nd 3rd](#)

[bitwise operator](#)
[iterator](#)
[overloaded operator 2nd](#)
[Sales item](#)
[string](#)
[compound expression 2nd](#)
[compound statement 2nd](#)
[compound type 2nd 3rd](#)
[compute](#)
[overloaded version](#)

concatenation
[Screen operations](#)
[string](#)
[string literal](#)

[concrete class](#)
[initialization](#)

[condition 2nd](#)
[and conversion](#)
[assignment in](#)
[in do while statement](#)
[in for statement 2nd](#)
[in if statement 2nd](#)
[in logical operator](#)
[in while statement](#)
[stream type as 2nd 3rd](#)
[string input operation as](#)

[condition state 2nd](#)
[conditional compilation](#)
[conditional operator \(? :\) 2nd](#)

[operand order of evaluation](#)
[console window](#)
[const](#)
[and dynamically allocated array](#)
[conversion to 2nd](#)
[and template argument](#)
[iterator vs. const iterator](#)
[object scope 2nd](#)
[overloading and 2nd](#)
[parameter](#)
[pointer](#)
[reference](#)
[initialization](#)

const data member
compared to [mutable](#)
[initialization](#)
[static data member](#)

const member function 2nd 3rd 4th 5th

const object, constructor

const pointer [See also [pointer to const](#)]
conversion from [non const](#)

const reference
argument
conversion from [non const](#)
parameter
overloading
return type

const void* 2nd

const cast 2nd

const iterator 2nd
compared to [const iterator](#)
container

const reference

const reverse iterator
container

constant expression 2nd
and header file
array index
bit-field
enumerator
nontype template parameter
static data member

construction, order of 2nd
derived objects 2nd
multiple base classes
virtual base classes

constructor 2nd 3rd 4th 5th 6th
const objects
conversion 2nd
function matching
with standard conversion

copy 2nd
base from derived
multiple inheritance

default 2nd 3rd 4th 5th 6th
default argument in
derived class
initializes immediate base class
initializes virtual base

execution flow
explicit 2nd
copy-initialization

for associative container
for sequential container
function matching
function [try](#) block
in constructor initializer list
inheritance
initializer
may not be virtual
object creation

[order of construction](#)

[derived objects 2nd](#)

[multiple base classes](#)

[virtual base classes](#)

[overloaded](#)

[pair](#)

[resource allocation](#)

[synthesized copy 2nd](#)

[synthesized default 2nd 3rd 4th](#)

[virtual inheritance](#)

[with standard conversion](#)

[constructor initializer list 2nd 3rd 4th 5th](#)

[compared to assignment](#)

[derived classes](#)

[function `try` block](#)

[initializers](#)

[multiple base classes](#)

[sometimes required](#)

[virtual base class](#)

[container 2nd 3rd 4th](#) [See also [sequential container](#), [associative container](#)]

[and generic algorithms](#)

[as element type](#)

[assignment \(=\)](#)

[associative 2nd](#)

[begin](#)

[clear](#)

[const iterator](#)

[const reference](#)

[const reverse iterator](#)

[element type constraints 2nd](#)

[elements and destructor](#)

[elements are copies](#)

[empty](#)

[end](#)

[erase](#)

[has bidirectional iterator](#)

[inheritance](#)

[insert](#)

[iterator](#)

[rbegin 2nd](#)

[reference](#)

[rend 2nd](#)

[returning a](#)

[reverse iterator 2nd](#)

[sequential 2nd](#)

[size](#)

[size_type](#)

[supports relational operators](#)

[swap](#)

[types defined by](#)

[continue statement 2nd](#)

[example](#)

[control flow of 2nd](#)

[conversion 2nd](#)

[ambiguous](#)

[and assignment](#)

[argument](#)

[with class type conversion](#)

[arithmetic type 2nd](#)
[array to pointer 2nd](#)
 and template argument
[conversion constructor](#)
[copy](#)
[copy constructor 2nd 3rd](#)
 and assignment operator
 argument passing
 base from derived
 for derived class
 initialization
[Message](#)

[copy control 2nd](#)
 handle class
 inheritance
 message handling example
 multiple inheritance
 of pointer members

[copy-initialization](#)
 using constructor

[copy backward](#)

[count](#)
 book finding program
 map
 multimap
 multiset
 set

[count, use 2nd](#)
[count if 2nd](#)
 with function object argument

[cout 2nd](#)
 by default tied to [cin](#)

[cstddef header 2nd](#)
[cstdlib header](#)
[cstring header](#)

[ctrl-d \(Unix end-of-file\)](#)
[ctrl-z \(Windows end-of-file\)](#)

[curly brace 2nd](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[dangling else](#) 2nd

[dangling pointer](#) 2nd

- [returning pointer to local variable](#)

- [synthesized copy control](#)

[data abstraction](#) 2nd

- [advantages](#)

[data hiding](#)

[data structure](#) 2nd

[data type, abstract](#)

[ddd.ddd L or ddd.dddI \(long double literal\)](#)

[dec manipulator](#)

[decimal literal](#)

[declaration](#) 2nd

- [class](#) 2nd

- [class template member specialization](#)

- [dependencies and template friends](#)

- [derived class](#)

- [export](#)

- [forward](#) 2nd

- [function](#)

- [exception specification](#)

- [function template specialization](#) 2nd

- [member template](#)

- [template](#)

- [using](#) 2nd 3rd 4th

- [access control](#)

- [class member access](#)

- [overloaded inherited functions](#)

[declaration statement](#) 2nd

[decrement \(- - \)](#)

- [iterator](#)

- [overloaded operator](#)

- [prefix yields lvalue](#)

- [reverse iterator](#)

[deduction, template argument](#)

[default argument](#)

- [and header file](#)

- [function matching](#)

- [in constructor](#)

- [initializer](#)

- [overloaded function](#)

- [virtual functions](#)

[default case label](#) 2nd

[default constructor](#) 2nd 3rd 4th 5th 6th

- [default argument](#)

- [Sales item](#)

- [string](#) 2nd

- [synthesized](#) 2nd 3rd 4th

- [used implicitly](#)

- [variable definition](#)

[definition 2nd](#)
[array](#)
[base class](#)
[C-style string](#)
[class 2nd](#)
[class data member 2nd](#)
[class static member](#)
[class template](#)
 [static member](#)
[class template specialization](#)
 [member defined outside class body](#)
[class type object](#)
[derived class](#)
[destructor](#)
[dynamically allocated array](#)
[dynamically allocated object](#)
[function](#)
[inside a switch expression](#)
[inside a while condition](#)
[inside an if condition](#)
[map 2nd](#)
[multi-dimensioned array](#)
[namespace](#)
 [can be discontiguous](#)
 [member](#)
[of variable after case label](#)
[overloaded operator](#)
[pair](#)
[pointer](#)
[pointer to function](#)
[static data member](#)
[variable](#)
[delete 2nd 3rd 4th](#)
 [compared to operator delete](#)
 [const object](#)
 [execution flow](#)
 [member operator](#)
 [and inheritance](#)
 [interface](#)
[memory leak 2nd](#)
[null pointer](#)
[runs destructor](#)
[delete \[\]](#)
 [and dynamically allocated array](#)
[dequeue](#)
 [as element type](#)
 [assign](#)
 [assignment \(=\)](#)
 [at](#)
 [back](#)
 [begin](#)
 [clear](#)
 [const iterator](#)
 [const reference](#)
 [const reverse iterator](#)
[constructor from element count, uses copy constructor](#)
[constructors](#)
[difference type](#)

[element type constraints 2nd](#)

[empty](#)

[end](#)

[erase](#)

[invalidates iterator](#)

[front](#)

[insert](#)

[invalidates iterator](#)

[iterator](#)

[iterator supports arithmetic](#)

[performance characteristics](#)

[pop back](#)

[pop front](#)

[push back](#)

[invalidates iterator](#)

[push front](#)

[invalidates iterator](#)

[random-access iterator](#)

[rbegin 2nd](#)

[reference](#)

[relational operators](#)

[rend 2nd](#)

[resize](#)

[reverse iterator 2nd](#)

[size](#)

[size type](#)

[subscript \(\[\]\)](#)

[supports relational operators](#)

[swap](#)

[types defined by](#)

[value type](#)

[dereference \(*\) 2nd 3rd 4th](#)

[and increment](#)

[auto_ptr](#)

[iterator](#)

[on map iterator yields pair](#)

[overloaded operator](#)

[pointer](#)

[yields lvalue 2nd](#)

[derivation list, class 2nd](#)

[access control](#)

[default access label](#)

[derived class 2nd 3rd 4th \[See also virtual function\]](#)

[access control 2nd](#)

[as base class](#)

[assigned or copied to base object](#)

[assignment \(=\)](#)

[constructor](#)

[calls virtual function](#)

[for remote virtual base](#)

[initializes immediate base class](#)

[constructor initializer list](#)

[conversion to base](#)

[access control](#)

[copy constructor](#)

[default derivation label](#)

[definition](#)

[destructor](#)

[calls virtual function](#)
[friendship not inherited](#)
[handle class](#)
[member hides member in base](#)
[member operator delete](#)
[multiple base classes](#)
[no conversion from base](#)
[scope](#)
[scope \(::\) to access base class member](#)
[static members](#)

[using declaration](#)
[inherited functions](#)
[member access](#)
[with remote virtual base](#)

derived object

[contains base part](#)
[multiple base classes, contains base part for each](#)

[derived to base 2nd](#)

[access control](#)
[enumeration type to integer](#)
[from istream](#)
[function matching of template and nontemplate functions](#)
[function to pointer](#)
[and template argument](#)
[implicit](#)
[inheritance](#)
[integral promotion](#)
[multi-dimensioned array to pointer](#)
[multiple inheritance](#)
[nontemplate type argument](#)
[of return value](#)
[rank for function matching](#)
[rank of class type conversions](#)
[signed to unsigned](#)
[signed type](#)
[template argument](#)
[to const](#)
[and template argument](#)
[parameter matching](#)
[to const pointer](#)
[virtual base](#)

design

[CachedObj](#)
[class member access control](#)
[class type conversions](#)
[consistent definitions of equality and relational operators](#)
[is-a relationship](#)
[Message class](#)
[namespace](#)
[of handle classes](#)
[of header files](#)
[export](#)
[inclusion compilation model](#)
[separate compilation model](#)
[optimizing new and delete](#)
[using freelist](#)
[overloaded operator](#)
[overview of use counting](#)
[Query classes](#)

[Queue](#)

[resource allocation is initialization](#)

[Sales item handle class](#)

[TextQuery class](#)

[vector memory allocation strategy](#)

[writing generic code](#)

[pointer template argument](#)

[destruction, order of](#)

[derived objects](#)

[multiple base classes](#)

[virtual base classes](#)

[destructor 2nd 3rd](#)

[called during exception handling](#)

[container elements](#)

[definition](#)

[derived class](#)

[explicit call to](#)

[implicitly called](#)

[library classes](#)

[Message](#)

[multiple inheritance](#)

[order of destruction](#)

[derived objects](#)

[multiple base classes](#)

[virtual base classes](#)

[resource deallocation](#)

[rule of three](#)

[exception for virtual destructors](#)

[should not throw exception](#)

[synthesized 2nd](#)

[use counting 2nd](#)

[valuelike classes](#)

[virtual in base class](#)

[virtual, multiple inheritance](#)

[development environment, integrated](#)

[difference type 2nd 3rd](#)

[dimension 2nd](#)

[direct base class \[See \[immediate base class\]\(#\)\]](#)

[direct-initialization](#)

[using constructor](#)

[directive, using 2nd](#)

[pitfalls](#)

[Disc item](#)

[class definition](#)

[discriminant 2nd](#)

[divides<T>](#)

[division \(/\) 2nd](#)

[do while statement](#)

[condition in](#)

[domain error](#)

[dot \(.\) 2nd](#)

[class member access](#)

[dot operator \(.\) 2nd](#)

[class member access](#)

[double](#)

[literal \(numE num or numE num\)](#)

[long double](#)

[notation output format control](#)

[output format control](#)

[duplicate word program](#)

[revisited](#)

[dynamic binding 2nd](#)

[in C++](#)

[requirements for](#)

[dynamic type 2nd](#)

[dynamic cast 2nd 3rd](#)

[example](#)

[throws bad cast](#)

[to pointer](#)

[to reference](#)

[dynamically allocated](#)

[array 2nd](#)

[definition](#)

[delete](#)

[const object](#)

[initialization](#)

[of const](#)

Team LiB

◀ PREVIOUS NEXT ▶

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[edit-compile-debug](#) 2nd

 errors at link time

[else](#) [See [if statement](#)]

 dangling 2nd

[empty](#)

[associative container](#)

[container](#)

[priority queue](#)

[queue](#)

[stack](#)

[string](#) 2nd

[vector](#) 2nd

[encapsulation](#) 2nd

 advantages

[end](#)

[container](#)

[map](#)

[set](#)

[vector](#)

[end-of-file](#) 2nd 3rd

 entering from keyboard

[Endangered](#)

[endl](#)

 manipulator flushes the buffer

[ends](#), manipulator flushes the buffer

[enum](#) keyword

[enumeration](#) 2nd

 conversion to integer

 function matching

[enumerator](#) 2nd

 conversion to integer

[environment, integrated development](#)

[eof](#)

[eofbit](#)

[equal](#)

[equal member function](#)

[equal range](#)

[associative container](#)

[book finding program](#)

[equal to<T>](#)

[equality \(==\)](#) 2nd 3rd 4th

[algorithm](#) 2nd

 and assignment

[container](#) 2nd

[container adaptor](#) 2nd

[iterator](#) 2nd 3rd 4th

 overloaded operator 2nd

 consistent with equality

[string](#) 2nd 3rd 4th

[erase](#)

[associative container](#)

[container](#)

[invalidates iterator](#)

[map](#)

[multimap](#)

[multiset](#)

[sequential container](#)

[set](#)

[string](#)

[error, standard](#)

[escape sequence 2nd](#)

[hexadecimal \(\xnnn\)](#)

[octal \(\ nnn\)](#)

[evaluation](#)

[order of 2nd](#)

[short-circuit](#)

[exception](#)

[class 2nd](#)

[class hierarchy](#)

[constructor](#)

[extending the hierarchy](#)

[header](#)

[what member 2nd](#)

[exception handling 2nd](#) [See also [throw](#), [catch clause](#)]

[and terminate](#)

[compared to assert](#)

[exception in destructor](#)

[finding a catch clause](#)

[function try block 2nd](#)

[handler](#) [See [catch clause](#)]

[library class destructors](#)

[local objects destroyed](#)

[specifier 2nd 3rd 4th](#)

[nonreference](#)

[reference](#)

[types related by inheritance](#)

[stack unwinding](#)

[uncaught exception](#)

[unhandled exception](#)

[exception object 2nd](#)

[array or function](#)

[initializes catch parameter](#)

[must be copyable](#)

[pointer to local object](#)

[rethrow](#)

[exception safety 2nd](#)

[exception specification 2nd](#)

[function pointers](#)

[tHROW\(\)](#)

[unexpected](#)

[violation](#)

[virtual functions](#)

[exception, raise](#) [See [throw](#)]

[executable file](#)

[EXIT_FAILURE](#)

[EXIT_SUCCESS](#)

[explicit constructor 2nd](#)

[copy-initialization](#)

[export](#)
[and header design](#)
[keyword 2nd](#)
[exporting C++ to C](#)
[expression 2nd 3rd 4th](#)
[and operand conversion](#)
[compound 2nd](#)
[constant 2nd](#)
[throw 2nd](#)
[expression statement 2nd](#)
[extended compute](#)
[extension compiler](#)
[extern](#)
[extern 'C' \[See \[linkage directive\]\(#\)\]](#)
[extern const](#)

Team LiB

◀ PREVIOUS NEXT ▶

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[factorial program](#)

[fail](#)

[failbit](#)

[file](#)

[executable](#)

[object](#)

[source 2nd](#)

[file mode 2nd](#)

[combinations](#)

[example](#)

[file static 2nd](#)

[fill](#)

[fill n](#)

[find 2nd](#)

[book finding program](#)

[map](#)

[multimap](#)

[multiset](#)

[set](#)

[string](#)

[find last word program](#)

[find end](#)

[find first not of, string](#)

[find first of 2nd 3rd](#)

[string](#)

[find if 2nd](#)

[find last not of, string](#)

[find last of, string](#)

[find val program](#)

[fixed manipulator](#)

[float](#)

[literal \(num f or num f\)](#)

[floating point](#)

[notation output format control](#)

[output format control](#)

[floating point literal \[See \[double literal\]\(#\)\]](#)

[flow of control 2nd](#)

[flush, manipulator flushes the buffer](#)

[Folder \[See \[Message\]\(#\)\]](#)

[for statement 2nd](#)

[condition in](#)

[execution flow](#)

[expression](#)

[for header](#)

[initialization statement](#)

[scope](#)

[for statement](#) [for statement](#)

[for each](#)

[format state](#)

[forward declaration of class type](#)

[forward iterator 2nd](#)

[fp compute](#)

[free store 2nd](#)

[freelist 2nd](#)

[friend 2nd](#)

[class](#)

[class template](#)

[explicit template instantiation](#)

[nontemplate class or function](#)

[template class or function](#)

[function template, example](#)

[inheritance](#)

[member function](#)

[overloaded function](#)

[overloaded operator](#)

[scope considerations](#)

[namespaces](#)

[template example](#)

[friend keyword](#)

[front](#)

[queue](#)

[sequential container](#)

[front inserter 2nd](#)

[compared to inserter](#)

[fstream 2nd 3rd](#) [See also [istream](#), [ostream](#)]

[close](#)

[constructor](#)

[file marker](#)

[file mode](#)

[combinations](#)

[example](#)

[file random access](#)

[header 2nd](#)

[off type](#)

[open](#)

[pos type](#)

[random IO sample program](#)

[seek and tell members](#)

[function 2nd 3rd 4th](#)

[calls resolved at run time](#)

[candidate 2nd](#)

[compared to run-time type identification](#)

[conversion to pointer](#)

[and template argument](#)

[default argument](#)

[derived classes](#)

[destructor](#)

[destructor and multiple inheritance](#)

[equal member](#)

[exception specifications](#)

[function returning](#)

[in constructors](#)

[in destructor](#)

[inline 2nd](#)

[inline and header](#)

[introduction](#)

[2nd 3rd](#) [See [member function](#)]

[multiple inheritance](#)

[no virtual constructor](#)
[nonvirtual, calls resolved at compile time](#)
[overloaded 2nd 3rd](#)
 [compared to redeclaration](#)
 [friend declaration](#)
 [scope](#)
 [virtual](#)
[overloaded operator](#)
[overriding run-time binding](#)
[pure virtual 2nd](#)
 [example](#)
[recursive 2nd](#)
[return type](#)
[run-time type identification](#)
[scope](#)
[to copy unknown type](#)
[type-sensitive equality](#)
[viable 2nd](#)
[virtual 2nd 3rd](#)
 [assignment operator](#)
[function adaptor 2nd 3rd](#)
 [bind1st](#)
 [bind2nd](#)
 [binder](#)
 [negator](#)
 [not1](#)
 [not2](#)
[function body 2nd 3rd 4th](#)
[function call](#)
 [ambiguous 2nd](#)
 [execution flow](#)
 [overhead](#)
 [through pointer to function](#)
 [through pointer to member](#)
 [to overloaded operator](#)
 [to overloaded postfix operator](#)
 [using default argument](#)
[function declaration](#)
 [and header file](#)
 [exception specification](#)
[function definition](#)
[function matching 2nd](#)
 [and overloaded function templates](#)
 [examples](#)
[argument conversion](#)
[conversion operator](#)
[conversion rank](#)
 [class type conversions](#)
[enumeration parameter](#)
[integral promotion](#)
[multiple parameters](#)
[namespaces](#)
[of member functions](#)
[overloaded operator](#)
[function name 2nd](#)
[function object 2nd](#)
[function pointer](#)
 [and template argument deduction](#)
 [definition](#)
 [exception specifications](#)

[function returning initialization](#)
[overloaded functions parameter](#)
[return type typedef](#)
[function prototype 2nd](#)
[function return type 2nd 3rd const reference](#)
[no implicit return type nonreference](#)
[uses copy constructor reference](#)
[reference yields lvalue void](#)
[function scope](#)
[function table pointer to member](#)
[function template 2nd](#) [See also [template parameter](#), [template argument](#), [instantiation](#)]
[as friend compiler error detection declaration](#)
[error detection explicit template argument and function pointer specifying](#)
[export inline instantiation template argument deduction type-dependent code](#)
function template specialization
[compared to overloaded function declaration 2nd example namespaces scope](#)
function [TRY block 2nd](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[q++](#)

[acd program](#)

[generate](#)

[generate_n](#)

generic algorithm [See [algorithm](#)]

[generic handle class](#) 2nd

generic memory management [See [CachedObj](#)]

[generic programming](#) 2nd

[and pointer template argument](#)

[type-independent code](#)

[getline](#) 2nd

[example](#) 2nd

[global namespace](#) 2nd

[global scope](#) 2nd

[global variable, lifetime](#)

[GNU compiler](#)

[good](#)

[goto statement](#) 2nd

[greater-than \(>\)](#) 2nd 3rd 4th

[greater-than-or-equal \(>=\)](#) 2nd 3rd 4th

[greater<T>](#)

[greater_equal<T>](#)

[GT6 program](#)

[GT_cls](#)

[guard header](#) 2nd

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

[Handle](#)

[int instantiation](#)
[operations](#)
[Sales item instantiation](#)

[handle class](#) 2nd

[copy control](#)
[copying unknown type](#)
[design](#)
[generic 2nd](#)
[that hides inheritance hierarchy](#)
[using a](#)

[handler](#) [See [catch clause](#)]

[has-a relationship](#)

[HasPtr](#)

[as a smart pointer](#)
[using synthesized copy control](#)
[with value semantics](#)

[header](#) 2nd 3rd 4th

[algorithms](#)
[and constant expression](#)
[and library names](#)
[bitset](#)
[C library](#)
[cassert](#)
[cctype](#) 2nd
[class definition](#) 2nd
[cstddef](#) 2nd
[cstdlib](#)
[cstring](#)
[default argument](#)
[dequeue](#)
[design](#)
[export](#)
[inclusion compilation model](#)
[namespace members](#)
[separate compilation model](#)
[exception](#)

[fstream](#) 2nd

[function declaration](#)
[inline function](#)
[inline member function definition](#)

[iomanip](#)

[iostream](#)

[iterator](#)

[list](#)

[map](#) 2nd

[new](#)

[numeric](#)

[programmer-defined](#)

[queue](#)

[Sales item](#) 2nd 3rd

[set](#) 2nd

[sstream](#) 2nd

[stack](#)

[stdexcept](#) 2nd

[string](#)

[type info](#)

[using declaration](#)

[utility](#)

[vector](#) 2nd

[header file, naming convention](#)

[header guard](#) 2nd

[heap](#) 2nd

[hex manipulator](#)

[hexadecimal escape sequence](#) (`\xnnn`)

[hexadecimal literal](#) (`0x num` or `0x num`).

[hides, names in base hidden by names in derived](#)

[hierarchy, inheritance](#) 2nd 3rd

[high-order bits](#) 2nd

Team LiB

◀ PREVIOUS NEXT ▶

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

[IDE](#)

[identification, run-time type](#) [2nd](#)

[identifier](#) [2nd](#)

[naming convention](#)

[reserved](#)

[if statement](#) [2nd](#) [3rd](#) [4th](#)

[compared to](#) [switch](#)

[dangling else](#)

[else branch](#) [2nd](#)

[if statement, else branch](#)

[ifstream](#) [2nd](#) [See also [istream](#)]

[close](#)

[constructor](#)

[file marker](#)

[file mode](#)

[combinations](#)

[example](#)

[file random access](#)

[off type](#)

[open](#)

[pos type](#)

[random IO sample program](#)

[seek and tell members](#)

[immediate base class](#) [2nd](#)

[implementation](#) [2nd](#) [3rd](#)

[implementation inheritance](#)

[implicit conversion](#) [See [conversion](#)]

[implicit return](#)

[from main allowed](#)

[implicit this pointer](#) [2nd](#) [3rd](#) [4th](#)

[in and overloaded operator](#)

[static member functions](#)

[implicit this pointer, overloaded operator](#)

[in \(file mode\)](#)

[include](#) [See [#include](#)]

[includes](#)

[inclusion compilation model](#) [2nd](#)

[incomplete type](#) [2nd](#)

[restriction on use](#) [2nd](#) [3rd](#)

[increment](#) [\(++\)](#) [2nd](#) [3rd](#) [4th](#) [5th](#) [6th](#)

[and dereference](#) [2nd](#)

[iterator](#) [2nd](#) [3rd](#) [4th](#) [5th](#)

[overloaded operator](#)

[pointer](#) [2nd](#)

[prefix yields lvalue](#) [2nd](#)

[reverse iterator](#) [2nd](#)

[indentation](#) [2nd](#)

[index](#) [2nd](#)

[indirect base class](#) [2nd](#)

[inequality \(!=\)](#) [2nd](#) [3rd](#) [4th](#)

[container](#) [2nd](#)

[container adaptor](#) [2nd](#)

[iterator](#) [2nd](#) [3rd](#) [4th](#)

[overloaded operator](#) [2nd](#)

[string](#) [2nd](#)

[inheritance](#) [2nd](#)

[containers](#)

[conversions](#)

[default access label](#)

[friends](#)

[handle class](#)

[implementation](#)

[interface](#)

[iostream](#) [diagram](#)

[multiple](#) [See [multiple base class](#)]

[private](#)

[static members](#)

[virtual](#) [2nd](#)

[inheritance hierarchy](#) [2nd](#) [3rd](#)

[inheritance vs. composition](#)

[initialization](#) [2nd](#) [3rd](#) [4th](#)

[array](#)

[array of](#) [char](#)

[built-in type](#)

[C-style string](#)

[class data member](#)

[class member of built-in type](#)

[class member of class type](#)

[class type](#) [2nd](#)

[const static data member](#)

definitions and [goto](#)

[constructor](#)

[dynamically allocated array](#)

[dynamically allocated object](#)

[local](#) [2nd](#)

[map](#)

[memberwise](#) [2nd](#)

[multi-dimensioned array](#)

[objects of concrete class type](#)

[pair](#)

[parameter](#)

[pointer](#)

[pointer to function](#)

[return value](#)

[scope](#)

[value](#) [2nd](#)

[variable](#) [2nd](#) [3rd](#)

[vs. assignment](#)

[initialization vs. assignment](#)

[initialized](#) [2nd](#)

[initializer list, constructor](#) [2nd](#) [3rd](#) [4th](#) [5th](#)

[inline function](#) [2nd](#)

[and header](#)

[function template](#)

[member function](#)

[and header](#)

[inner product](#)

[inplace merge](#)

[input \(>>\)](#) 2nd 3rd 4th
[istream iterator](#)
[istream iterator.](#)
[overloaded operator](#)
[error handling](#)
[must be nonmember](#)
[precedence and associativity](#) 2nd
[Sales item](#)
[Sales item.](#)
[string](#) 2nd 3rd

[input iterator](#) 2nd

[input standard](#)

[insert](#)

[inserter](#)
[invalidates iterator](#)
[map](#)
[multimap](#)
[multiset](#)
[return type from set::insert](#)
[sequential container](#)
[set](#)
[string](#)

[insert iterator](#) 2nd 3rd

[inserter](#)

[inserter](#)
compared to [front inserter](#)

[instantiation](#) 2nd

[class template](#) 2nd 3rd
[member function](#)
[nontype parameter](#)
[type](#)
[error detection](#)
[function template](#)
from function pointer
nontemplate argument conversion
nontype template parameter
template argument conversion

[member template](#)

[nested class template](#) 2nd

[on use](#)

[static class member](#)

[int](#)

[literal](#)

[Integral](#)

[integral promotion](#) 2nd

[function matching](#)

[integral type](#) 2nd

[integrated development environment](#)

[interface](#) 2nd 3rd

[interface inheritance](#)

[internal manipulator](#)

[interval, left-inclusive](#) 2nd

[invalid argument](#)

[invalidated iterator](#) 2nd

IO stream [See [stream](#)]

[iomanip header](#)

[iostate](#)

[iostream](#) 2nd 3rd [See also [istream](#), [ostream](#)]

[header](#)
[inheritance hierarchy](#)
[seek and tell members](#)
[is-a relationship](#)
[isalnum](#)
[isalpha](#)
[ISBN](#)
[isbn mismatch](#)
 [destructor explained](#)
[iscntrl](#)
[isdigit](#)
[isgraph](#)
[islower](#)
[isprint](#)
[ispunct](#)
[isShorter program 2nd](#)
[isspace](#)
[istream 2nd 3rd \[See also manipulator\]](#)
 [condition state](#)
 [flushing input buffer](#)
 [format state](#)
 [aCount](#)
 [get](#)
 [multi-byte version](#)
 [returns int 2nd](#)
 [getline 2nd](#)
 [getline, example](#)
 [ignore](#)
[inheritance hierarchy](#)
[input \(>>\)](#)
 [precedence and associativity](#)
[no containers of](#)
[no copy or assign](#)
[peek](#)
[put](#)
[putback](#)
[read](#)
[seek and tell members](#)
[unformatted operation](#)
 [multi-byte](#)
 [single-byte](#)
[unget](#)
[write](#)
[istream iterator 2nd](#)
 [and class type](#)
 [constructors](#)
 [input iterator](#)
 [input operator \(>>\)](#)
 [limitations](#)
 [operations](#)
 [used with algorithms](#)
[istringstream 2nd 3rd \[See also istream\]](#)
 [str](#)
 [word per line processing 2nd 3rd](#)
[isupper](#)
[isxdigit](#)
[Item_base](#)
 [class definition](#)

[constructor](#)
[interface](#)
[member functions](#)
[iterator swap](#)
[iterator 2nd 3rd 4th 5th](#)
[iterator 2nd](#)
iterator
 [argument](#)
 [arrow \(->\)](#)
 [bidirectional 2nd](#)
 [compared to reverse iterator 2nd](#)
[iterator](#)
 [container](#)
iterator
 [destination 2nd](#)
 [equality 2nd](#)
 [forward 2nd](#)
 [generic algorithms](#)
 [inequality 2nd](#)
 [input 2nd](#)
 [insert 2nd 3rd](#)
 [invalidated 2nd](#)
invalidated by
 [assign](#)
 [erase](#)
 [insert](#)
 [push back](#)
 [push front](#)
 [resize](#)
[off-the-end 2nd 3rd](#)
[operations](#)
[output 2nd](#)
[parameter 2nd](#)
[random-access 2nd](#)
[relational operators](#)
[reverse 2nd 3rd](#)
[stream](#)
[iterator arithmetic 2nd 3rd 4th](#)
 [relational operators](#)
[iterator category 2nd](#)
 [algorithm and 2nd](#)
 [bidirectional iterator](#)
 [forward iterator](#)
 [hierarchy](#)
 [input iterator](#)
 [output iterator](#)
 [random-access iterator](#)
[iterator header](#)
[iterator range 2nd 3rd](#)
 [algorithms constraints on 2nd](#)
 [erase](#)
 [generic algorithms](#)
 [insert](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

key type

[associative containers](#)

keyword

[enum](#)

[export](#)

[friend](#)

[namespace](#)

[protected](#)

[template](#)

[try](#)

[union](#)

[virtual](#)

[keyword table](#)

[Koenig lookup](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

[L' c' \(wchar_t literal\)](#)

[label](#)

[access](#) [2nd](#) [3rd](#) [4th](#)

[case](#) [2nd](#) [3rd](#)

[statement](#)

[labeled statement](#) [2nd](#)

[left manipulator](#)

[left-inclusive interval](#) [2nd](#)

[left-shift \(<<\)](#) [2nd](#) [3rd](#) [4th](#)

[length error](#)

[less-than \(<\)](#) [2nd](#) [3rd](#) [4th](#)

[overloaded and containers](#)

[used by algorithm](#) [2nd](#)

[less-than-or-equal \(<=\)](#) [2nd](#) [3rd](#) [4th](#) [5th](#) [6th](#)

[less<T>](#)

[less_equal<T>](#)

[lexicographical compare](#)

[library names to header table](#)

[library type](#)

[library, standard](#) [2nd](#)

[lifetime, object](#) [2nd](#)

[link time errors from template](#)

[linkage directive](#) [2nd](#)

[C++ to C](#)

[compound](#)

[overloaded function](#)

[parameter or return type](#)

[pointer to function](#)

[single](#)

[linking](#) [2nd](#)

[list](#)

[as element type](#)

[assign](#)

[assignment \(=\)](#)

[back](#)

[begin](#)

[bidirectional iterator](#)

[clear](#)

[const iterator](#)

[const reference](#)

[const reverse iterator](#)

[constructor from element count, uses copy constructor](#)

[constructors](#)

[element type constraints](#) [2nd](#)

[empty](#)

[end](#)

[erase](#)

[front](#)

[insert](#)

[iterator](#)
[merge](#)
[performance characteristics](#)
[pop back](#)
[pop front](#)
[push back](#)
[push front](#)
[rbegin 2nd](#)
[reference](#)
[relational operators](#)
[remove](#)
[remove_if](#)
[rend 2nd](#)
[resize](#)
[reverse](#)
[reverse_iterator 2nd](#)
[size](#)
[size_type](#)
[specific algorithms](#)
[splice](#)
[swap](#)
[types defined by](#)
[unique](#)
[value_type](#)
[literal 2nd 3rd](#)
[bool](#)
[char](#)
[decimal](#)
[double \(numE num or nume num\)](#)
[float \(numF or numf\)](#)
[hexadecimal \(0x num or 0x num\).](#)
[int](#)
[long \(numL or numl\)](#)
[long double \(ddd.dddL or ddd.ddd1\)](#)
[multi-line](#)
[octal \(0 num\)](#)
[string 2nd 3rd](#)
[unsigned \(numU or numu\)](#)
[wchar_t](#)
[local class 2nd](#)
[access control](#)
[name lookup](#)
[nested class in](#)
[restrictions on](#)
[local scope 2nd](#)
[local static object 2nd](#)
[local variable 2nd](#)
[destructor](#)
[lifetime](#)
[reference return type](#)
[logic_error](#)
[logical AND \(&&\) 2nd](#)
[operand order of evaluation 2nd](#)
[overloaded operator](#)
[logical NOT \(!\) 2nd](#)
[logical operator](#)

[function object](#)
[logical OR \(||\) 2nd](#)
[operand order of evaluation 2nd](#)
[overloaded operator](#)
[logical and<T>](#)
[logical not<T>](#)
[logical or<T>](#)
[long](#)
[literal \(numL or num1\)](#)
[long double](#)
[long double, literal \(ddd.dddL or ddd.ddd1\)](#)
[lookup_name 2nd](#)
[and templates](#)
[before type checking 2nd](#)
[multiple inheritance](#)
[class member declaration](#)
[class member definition 2nd](#)
[class member definition, examples](#)
[collisions under inheritance](#)
[depends on static type](#)
[multiple inheritance](#)
[inheritance 2nd](#)
[local class](#)
[multiple inheritance](#)
[ambiguous names](#)
[namespace names](#)
[argument-dependent lookup](#)
[nested class](#)
[overloaded virtual functions](#)
[virtual inheritance](#)
[low-order bits 2nd](#)
[lower bound](#)
[associative container](#)
[book finding program](#)
[lvalue 2nd](#)
[assignment](#)
[dereference](#)
[function reference return type](#)
[prefix decrement](#)
[prefix increment](#)
[subscript](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

machine-dependent

- [bitfield layout](#)
- [char representation](#)
- [division and modulus result](#)
- [end-of-file character](#)
- [iostate type](#)
- [linkage directive language](#)
- [nonzero return from main](#)
- [pre-compiled headers](#)
- [random file access](#)
- [reinterpret cast](#)
- [representation of enum type](#)
- [return from exception what operation](#)
- [signed and out-of-range value](#)
- [signed types and bitwise operators](#)
- [size of arithmetic types](#)
- [template compilation optimization](#)
- [terminate function](#)
- [type info members](#)
- [vector memory allocation size](#)
- [volatile implementation](#)

[magic number](#) 2nd

[main](#) 2nd

- [arguments to](#)
- [not recursive](#)
- [return type](#)
- [return value](#) 2nd
- [returns 0 by default](#)

[make pair](#)

[make plural](#) program

[manip](#)

[manipulator](#) 2nd 3rd

- [boolalpha](#) 2nd
- [change format state](#)
- [dec](#) 2nd
- [endl flushes the buffer](#)
- [ends flushes the buffer](#)
- [fixed](#) 2nd
- [flush flushes the buffer](#)
- [hex](#) 2nd
- [internal](#) 2nd
- [left](#) 2nd
- [noboolalpha](#) 2nd
- [noshowbase](#) 2nd
- [noshowpoint](#) 2nd
- [noskipws](#) 2nd
- [nouppercase](#) 2nd
- [oct](#) 2nd

[right](#) 2nd
[scientific](#) 2nd
[setfill](#) 2nd
[setprecision](#) 2nd
[setw](#) 2nd
[showbase](#) 2nd
[showpoint](#) 2nd
[skipws](#) 2nd
[unitbuf](#) flushes the buffer
[uppercase](#) 2nd

[map](#) 2nd
 as element type
 [assignment \(=\)](#)
 [begin](#)
 bidirectional iterator
 [clear](#)
 constructors
 [count](#)
 definition
 dereference yields [pair](#)
 element type constraints
 [empty](#)
 [end](#)
 [equal](#) range
 [erase](#) 2nd
 [find](#)
 header
 [insert](#)
 [iterator](#)
 key type constraints
 key type
 lower bound
 mapped type 2nd
 operations
 overriding the default comparison
 [rbegin](#)
 [rend](#)
 return type from [insert](#)
 [reverse iterator](#)
 size
 subscript operator
 supports relational operators
 [swap](#)
 upper bound
 value type

[mapped type](#), [map](#), [multimap](#)

[match](#), [best](#) 2nd
[max](#)

[member](#) [See also [class member](#)]
 [mutable data](#)
 pointer to 2nd

[member function](#) 2nd 3rd 4th
 as friend
 base member hidden by derived
 class template
 defined outside class body

[instantiation](#)
[const](#) 2nd 3rd
[defined outside class body](#) 2nd

[definition](#)
in class scope
name lookup
name lookup, examples

[equal](#)
function template [See [member template](#)]

[implicitly inline](#)

[inline](#)
and header

[overloaded](#)

[overloaded on const](#)

[overloaded operator](#) 2nd

[pointer to definition](#)

[returning *this](#)

[static](#)

[this pointer](#)

[undefined](#)

[member operator delete](#) 2nd

[and inheritance](#)

[CachedObj](#)

[example](#)

[interface](#)

[member operator delete \[\]](#)

[member operator new](#) 2nd

[CachedObj](#)

[example](#)

[interface](#)

[member operator new \[\]](#)

[member template](#) 2nd

[declaration](#)

[defined outside class body](#)

[examples](#)

[instantiation](#)

[template parameters](#)

[memberwise assignment](#) 2nd

[memberwise initialization](#) 2nd

[memory and object construction](#)

[memory exhaustion](#)

[memory leak](#) 2nd

after exception

memory management, generic [See [CachedObj](#)]

[merge](#)

[list](#)

[Message](#)

[assignment operator](#)

[class definition](#)

[copy constructor](#)

[design](#)

[destructor](#)

[put Msg in Folder](#)

[remove Msg from Folder](#)

method [See [member function](#)]

[Microsoft compiler](#)

[min](#)

[min_element](#)

[minus<T>](#)
[mismatch](#)
[mode_file 2nd](#)
[modulus \(%\) 2nd](#)
[modulus<T>](#)
[multi-dimensioned array](#)

[and pointer](#)
[conversion to pointer](#)
[definition](#)
[initialization](#)
[parameter](#)
[subscript operator](#)

[multi-line literal](#)

[multimap 2nd](#)

[assignment \(=\)](#)

[begin](#)

[clear](#)

[constructors](#)

[count](#)

[dereference yields pair](#)

[element type constraints](#)

[empty](#)

[equal range](#)

[erase 2nd](#)

[find](#)

[has no subscript operator](#)

[insert](#)

[iterator 2nd](#)

[key type constraints](#)

[key type](#)

[lower bound](#)

[mapped type](#)

[operations 2nd](#)

[overriding the default comparison](#)

[rbegin](#)

[rend](#)

[return type from insert](#)

[reverse iterator](#)

[size](#)

[supports relational operators](#)

[swap](#)

[upper bound](#)

[value type](#)

[multiple base class](#) [See also [virtual base class](#)]

[ambiguities](#)

[ambiguous conversion](#)

[avoiding potential name ambiguities](#)

[conversions](#)

[definition](#)

[destructor usually virtual](#)

[name lookup](#)

[object composition](#)

[order of construction](#)

[scope](#)

[virtual functions](#)

[multiple inheritance](#) [See [multiple base class](#)]

[multiplication \(*\) 2nd](#)

[multiplies<T>](#),
[multiset](#) 2nd
[assignment \(=\)](#)
[begin](#)
[clear](#)
[constructors](#)
[count](#)
[element type constraints](#)
[end](#)
[equal range.](#)
[erase](#) 2nd
[find](#)
[insert](#)
[iterator](#)
[key type constraints](#)
[lower bound.](#)
[operations](#) 2nd
[overriding the default comparison](#)
[rbegin](#)
[rend](#)
[return type from insert](#)
[reverse iterator.](#)
[Sales item.](#)
[supports relational operators](#)
[swap](#)
[upper bound.](#)
[example](#)
[value type.](#)
[mutable](#) data member 2nd

Team LiB

◀ PREVIOUS NEXT ▶

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

[name lookup](#) 2nd

[and templates](#)

[before type checking](#) 2nd

[multiple inheritance](#)

[class member declaration](#)

[class member definition](#) 2nd

[class member definition, examples](#)

[collisions under inheritance](#)

[depends on static type](#)

[multiple inheritance](#)

[inheritance](#) 2nd

[local class](#)

[multiple inheritance](#)

[ambiguous names](#)

[namespace names](#)

[argument-dependent lookup](#)

[nested class](#)

[overloaded virtual functions](#)

[virtual inheritance](#)

[name resolution](#) [See [name lookup](#)]

[namespace](#) 2nd 3rd 4th

[class friend declaration scope](#)

[cplusplus primer](#).

[definition](#)

[design](#)

[discontiguous definition](#)

[function matching](#)

[global](#)

[member](#)

[member definition](#)

[outside namespace](#)

[restrictions](#)

[nested](#)

[scope](#)

[unnamed](#)

[local to file](#)

[replace file](#) [static](#)

[namespace alias](#) 2nd

[namespace keyword](#)

[namespace pollution](#) 2nd

[naming convention](#)

[header file](#)

[source file](#)

[NDEBUG](#)

[negate<T>](#)

[negator](#) 2nd

[nested class](#) 2nd

[access control](#)

[class defined outside enclosing class](#)

[in class template](#)

[in local class](#)

[member defined outside class body](#)

[name lookup](#)

[QueueItem example](#)

[relationship to enclosing class 2nd](#)

[scope](#)

[static members](#)

[union](#)

[nested namespace](#)

nested type [See [nested class](#)]

[new 2nd 3rd 4th](#)

[compared to operator new](#)

[execution flow](#)

[header](#)

[member operator](#)

[member operator, interface](#)

[placement 2nd](#)

[compared to construct](#)

[new \[\]](#)

[new failure](#)

[next permutation](#)

[noboolalpha manipulator](#)

[NoDefault](#)

[nonconst reference](#)

[parameter](#)

[limitations](#)

[nonportable](#)

[nonprintable character 2nd](#)

[nonreference](#)

[parameter](#)

[uses copy constructor](#)

[return type](#)

[uses copy constructor](#)

[nontype template parameter 2nd 3rd 4th](#) [See also [template parameter](#)]

[class template](#)

[must be constant expression](#)

[nonvirtual function, calls resolved at compile time](#)

[noshowbase manipulator](#)

[noshowpoint manipulator](#)

[noskipws manipulator](#)

not equal [See [inequality](#)]

[not1](#)

[not2](#)

[not_equal_to<T>](#)

[NotQuery](#)

[definition](#)

[eval function](#)

[nouppercase manipulator](#)

[nth_element](#)

[NULL](#)

[null pointer](#)

[delete of](#)

[null statement 2nd](#)

null-terminated array [See [C-style string](#)]

[number, magic 2nd](#)

[numEnum or nume num \(double literal\)](#)

[numeric header](#)

numeric literal

[float \(numF or numf\).](#)

[long \(num_L or num₁\)](#),
[long double \(ddd.ddd_L or ddd.ddd₁\)](#),
[unsigned \(num_U or num_u\)](#),
[num_F or num_f \(float literal\)](#),
[num_L or num₁ \(long literal\)](#),
[num_U or num_u \(unsigned literal\)](#)

Team LiB

[PREVIOUS](#) [NEXT](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

[object](#) 2nd 3rd

[auto_ptr](#)

[automatic](#) 2nd

[constructor](#)

[destructor](#)

[exception](#)

[function](#)

[is not polymorphic](#)

[local static](#) 2nd

[temporary](#)

[object cleanup](#) [See [destructor](#)]

[object creation](#)

[constructor](#)

[order of construction](#)

[derived objects](#) 2nd

[multiple base classes](#)

[virtual base classes](#)

[order of destruction](#)

[derived objects](#)

[multiple base classes](#)

[virtual base classes](#)

[object file](#)

[object lifetime](#) 2nd

[and destructor](#)

[compared to scope](#)

[object-oriented programming](#) 2nd 3rd

[key ideas in](#)

[oct manipulator](#)

[octal escape sequence](#) (`\ nnn`)

[octal, literal](#) (`0 num`)

[off-the-end iterator](#) 2nd 3rd

[istream iterator](#).

[off-the-end pointer](#)

[ofstream](#) 2nd [See also [ostream](#)]

[close](#)

[constructor](#)

[file marker](#)

[file mode](#)

[combinations](#)

[example](#)

[file random access](#)

[off type](#).

[open](#)

[pos type](#).

[random IO sample program](#)

[open](#)

[open file program](#)

[open file, example of](#) 2nd

[operand](#) 2nd

[order of evaluation](#)

[comma operator](#)
[conditional operator](#)
[logical operator](#)
[operator 2nd](#)
 [addition \(+\)](#)
 [iterator 2nd](#)
 [pointer](#)
 [string](#)
 [address-of \(&\)](#)
 [arrow \(->\)](#)
 [class member access](#)
 [assignment \(=\) 2nd 3rd](#)
 [and conversion](#)
 [and equality](#)
 [container](#)
 [multiple inheritance](#)
 [pointer](#)
 [string](#)
 [to signed](#)
 [to unsigned](#)
 [yields lvalue](#)
 [binary 2nd](#)
 [bitwise AND \(&\)](#)
 [bitwise exclusive or \(^\)](#)
 [bitwise not \(~\)](#)
 [bitwise OR \(|\)](#)
 [bitwise OR \(|\). example](#)
 [call \(\) 2nd](#)
 [comma \(,\)](#)
 [operand order of evaluation](#)
 [comma \(,\). example](#)
 [compound assginment \(e.g., +=\), iterator](#)
 [compound assignment \(e.g., +=\) 2nd 3rd](#)
 [arithmetic](#)
 [bitwise](#)
 [string](#)
 [conditional \(?:\)](#)
 [operand order of evaluation](#)
 [decrement \(- -\)](#)
 [iterator](#)
 [prefix yields lvalue](#)
 [reverse iterator](#)
 [dereference \(*\)](#)
 [and increment](#)
 [iterator](#)
 [on map yields pair](#)
 [pointer](#)
 [sizeof](#)
 [typeid 2nd](#)
 [yields lvalue 2nd](#)
[operator alternative name](#)
[operator delete \[\] member](#)
[operator delete function 2nd](#)
 [compared to deallocate](#)
 [compared to delete expression](#)
[operator delete member](#)
 [and inheritance](#)

[CachedObj](#)

[example](#)

[interface](#)

[operator new \[\] member](#)

[operator new function 2nd](#)

[compared to allocate](#)

[compared to new expression](#)

[operator new member](#)

[CachedObj](#)

[example](#)

[interface](#)

operator overloading [See [overloaded operator](#)]

[options to main](#)

[order of construction 2nd](#)

[derived objects 2nd](#)

[multiple base classes](#)

[virtual base classes](#)

[order of destruction 2nd](#)

[derived objects](#)

[multiple base classes](#)

[virtual base classes](#)

[order of evaluation 2nd](#)

[comma operator](#)

[conditional operator](#)

[logical operator](#)

[ordering, strict weak 2nd](#)

[OrQuery](#)

[definition](#)

[eval function](#)

[ostream 2nd 3rd](#) [See also [manipulator](#)]

[condition state](#)

[floatfield member](#)

[flushing output buffer](#)

[format state](#)

[inheritance hierarchy](#)

[no containers of](#)

[no copy or assign](#)

[not flushed if program crashes](#)

[output \(<<\)](#)

[precedence and associativity](#)

[precision member](#)

[seek and tell members](#)

[tie member](#)

[unsetf member](#)

[ostream iterator. 2nd](#)

[and class type](#)

[constructors](#)

[limitations](#)

[operations](#)

[output iterator](#)

[output operator \(<<\)](#)

[used with algorithms](#)

[ostringstream 2nd](#) [See also [ostream](#)]

[str](#)

[out \(file mode\)](#)

[out of stock.](#)

[out of range. 2nd](#)

[output \(<<\) 2nd 3rd 4th](#)

[bitset](#) 2nd
[ostream iterator](#), 2nd
[overloaded operator](#)
 [formatting](#)
 [must be nonmember](#)
[precedence and associativity](#) 2nd
[Sales item.](#)
[string](#) 2nd 3rd
[output iterator](#) 2nd
[output, standard](#)
[overflow](#)
[overflow error.](#)
overload resolution [See [function matching](#)]
[overloaded](#) 2nd 3rd 4th
[overloaded function](#) 2nd
 [compared to redeclaration](#)
 [compared to template specialization](#)
 [friend declaration](#)
 [linkage directive](#)
 [namespaces](#)
 [scope](#)
 [using declarations](#)
 [using directive](#)
 [virtual](#)
[overloaded member function](#)
 [on const](#)
[overloaded operator](#) 2nd 3rd
 [& \(address-of\)](#)
 [&& \(logical AND\)](#)
 [\(\) \(call operator\)](#)
 [* \(dereference\)](#)
 [, \(comma operator\)](#)
 [-> \(arrow operator\)](#)
 [<< \(output operator\)](#)
 [formatting](#)
 [must be nonmember](#)
 [Sales item.](#)
 [= \(assignment\)](#) 2nd 3rd
 [and copy constructor](#)
 [check for self-assignment](#)
 [Message](#)
 [reference return](#) 2nd
 [rule of three](#)
 [use counting](#) 2nd
 [valuelike classes](#)
 >> (input operator)
 [error handling](#)
 [must be nonmember](#)
 [\[\]\(subscript\)](#)
 [reference return](#)
 [addition \(+\).](#) [Sales item.](#)
 [ambiguous](#)
 [arithmetic operators](#)
 [as virtual function](#)
 [binary operator](#)
 [candidate functions](#)
 [compound assignment \(e.g., +=\)](#)
 [Sales item.](#)

[consistency between relational and equality operators](#)

[definition 2nd](#)

[design](#)

[equality operators 2nd](#)

[explicit call to](#)

[explicit call to postfix operators](#)

[function matching](#)

[member and `this` pointer](#)

[member vs. nonmember function 2nd](#)

[postfix increment \(`++`\) and decrement \(`--`\) operators](#)

[precedence and associativity](#)

[prefix increment \(`++`\) and decrement \(`--`\) operators](#)

[relational operators 2nd](#)

[require class-type parameter](#)

[unary operator](#)

[`||` \(logical OR\)](#)

overloading [See [overloaded function](#)]

operator [See [overloaded operator](#)]

Team LiB

◀ PREVIOUS NEXT ▶

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[pair](#) 2nd

 as return type from [map::insert](#)

 as return type from [set::insert](#).

 default constructor

 definition

 initialization

[make_pair](#).

 operations

 public data members

[Panda](#)

 virtual inheritance

[parameter](#) 2nd 3rd 4th

 and [main](#)

 array and buffer overflow

 array type

 C-style string

[const](#)

[const](#) reference

 overloading

 ellipsis

 function pointer

 linkage directive

 initialization of

 iterator 2nd

 library container

 lifetime

 local copy

 matching

 ellipsis

 template specialization

 with class type conversion

 multi-dimensioned array

 non[const](#) reference

 nonreference type

 uses copy constructor

 of member function

 passing

 pointer members

 pointer to [const](#)

 overloading

 pointer to function

 linkage directive

 pointer to non[const](#).

 pointer type 2nd

 reference

 to array type

 to pointer

 rule of three

 exception for virtual destructors

 synthesized 2nd

template [See [template parameter](#)]

type checking

[and template argument
of reference to array](#)

[use counting](#) 2nd

[valuelike classes](#)

[vector type](#)

[parameter list](#) 2nd 3rd 4th

[member function definition](#)

[parentheses, override precedence](#)

[partial specialization](#) 2nd

[partial sort](#),

[partial sort copy](#),

[partial sum](#),

[partition](#)

[placement new](#) 2nd

[compared to construct](#)

[plus<T>](#),

[pointer](#) 2nd 3rd

[array](#)

[arrow \(->\)](#)

[as initializer of vector](#)

[as parameter](#)

[assignment](#)

char* [See [C-style string](#)]

[class member copy control](#)

[copy constructor](#)

[destructor](#)

[strategies](#)

[compared to iterator](#)

[compared to reference](#)

[const](#)

[const pointer to const](#)

[container constructor from](#)

[conversion from derived to multiple base](#)

[conversion fromderived to base](#)

[conversion to bool](#)

[conversion to void](#)

[dangling](#) 2nd

[synthesized copy control](#)

[declaration style](#)

[definition](#)

[delete](#)

[dynamic cast, example](#)

[function returning](#)

[implicit this](#) 2nd

[initialization](#)

[is polymorphic](#)

[multi-dimensioned array](#)

[new](#)

[null](#)

[off-the-end](#)

[pitfalls with generic programs](#)

[reference parameter](#)

[relational operator](#)

[return type and local variable](#)

[smart](#) 2nd 3rd

[handle class](#)

[overloaded \(++\) and \(*\).](#)

[overloaded -> \(arrowoperator\) and * \(dereference\)](#)

[subscript operator](#)

[to pointer](#)

[typedef](#)

[typeid operator](#)

[uninitialized](#)

[volatile](#)

[pointer arithmetic 2nd](#)

[pointer to const](#)

[argument](#)

[conversion from non const.](#)

[parameter](#)

[overloading](#)

[pointer to function](#)

[definition](#)

[exception specifications](#)

[function returning](#)

[initialization](#)

[linkage directive](#)

[overloaded functions](#)

[parameter](#)

[return type](#)

[typedef](#)

[pointer to member 2nd](#)

[and typedef](#)

[arrow \(->*\) 2nd](#)

[definition](#)

[dot \(.* \) 2nd](#)

[function pointer](#)

[function table](#)

[pointer to non const](#)

[argument](#)

[parameter](#)

[polymorphism 2nd](#)

[compile time polymorphism via templates](#)

[run time polymorphism in C++](#)

[pop](#)

[priority queue.](#)

[queue](#)

[stack](#)

[pop back, sequential container](#)

[pop front, sequential container](#)

[portable](#)

[postfix decrement \(- -\)](#)

[overloaded operator](#)

[yields rvalue](#)

[postfix increment \(++\)](#)

[and dereference](#)

[overloaded operator](#)

[precedence 2nd 3rd 4th 5th](#)

[of assignment](#)

[of conditional](#)

[of dot and dereference](#)

[of increment and dereference](#)

[of IO operator](#)

[of pointer to member and call operator](#)

[overloaded operator](#)

[pointer parameter declaration](#)

[precedence table](#)
[predicate 2nd](#)
[prefix decrement \(--\)](#)
 [overloaded operator](#)
 [yields lvalue](#)
[prefix increment \(++\)](#)
 [and dereference](#)
 [overloaded operator](#)
 [yields lvalue](#)
[preprocessor 2nd](#)
 [directive 2nd](#)
 [macro 2nd](#)
 [variable](#)
[prev permutation.](#)
[preventing copies of class objects](#)
[print total.](#)
 [explained](#)
[printable character](#)
[printValues program 2nd 3rd](#)
[priority queue. 2nd](#)
 [constructors](#)
 [relational operator](#)
[private](#)
 [class](#)
 [copy constructor](#)
 [inheritance](#)
 [member 2nd](#)
[private access label 2nd](#)
 [inheritance](#)
[private inheritance](#)
[program](#)
 [book finding](#)
 [using equal range.](#)
 [using find](#)
 [using upper bound.](#)
 [bookstore](#)
 [bookstore exception classes](#)
 [CachedObj](#)
 [duplicate words](#)
 [revisited](#)
 [factorial](#)
 [find last word](#)
 [find val.](#)
 [acd](#)
 [GT6](#)
 [Handle class](#)
 [int instantiation](#)
 [operations](#)
 [Sales item instantiation](#)
 [isShorter 2nd](#)
 [make plural.](#)
 [message handling classes](#)
 [open file.](#)
 [printValues 2nd 3rd](#)
 [ptr swap.](#)
[Query](#)
 [design](#)

[interface](#)
[operations](#)
[Queue](#)
[copy elems member](#)
[destroy member](#)
[pop member](#)
[push member](#)
[random IO example](#)
[restricted word count](#)
[rqcd](#)
[Sales item handle class](#)
[Screen class template](#)
[swap 2nd](#)
[TextQuery](#)
[class definition](#)
[design](#)
[interface](#)
[vector capacity](#)
[vector, capacity](#)
[vowel counting](#)
[word count](#)
[word transformation](#)
[ZooAnimal class hierarchy](#)
[programmer-defined header](#)
programming
[generic 2nd](#)
[object-oriented 2nd 3rd](#)
[promotion, integral 2nd](#)
[protected access label 2nd](#)
[protected keyword](#)
[protected, inheritance 2nd](#)
[prototype, function 2nd](#)
[ptr swap program](#)
[ptrdiff_t, 2nd](#)
public
[inheritance 2nd](#)
[member 2nd](#)
[public access label 2nd](#)
[inheritance](#)
[pure virtual function 2nd](#)
[example](#)
push
[priority queue.](#)
[queue](#)
[stack](#)
[push back, 2nd](#)
[back inserter,](#)
[sequential container](#)
[vector](#)
push_front
[front inserter,](#)
[sequential container](#)
[put Msg in Folder.](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[Query](#)

[& \(bitwise AND\)](#)

[definition](#)

[<< \(output operator\)](#)

[definition](#)

[design](#)

[interface](#)

[operations](#)

[| \(bitwise OR\)](#)

[definition](#)

[~ \(bitwise NOT\)](#)

[definition](#)

[Query base](#)

[definition](#)

[member functions](#)

[queue 2nd](#)

[Queue](#)

[<< \(output operator\)](#)

[assignment](#)

[queue](#)

[constructors](#)

[Queue](#)

[copy elems member 2nd](#)

[definition](#)

[design](#)

[destroy member](#)

[final class definition](#)

[interface](#)

[member template declarations](#)

[operations](#)

[pop member](#)

[push member](#)

[push specialized](#)

[queue](#)

[relational operator](#)

[Queue](#)

[template version, char*](#)

[QueueItem](#)

[as nested class](#)

[constructor](#)

[definition](#)

[CachedObj](#)

[allocation explained](#)

[friendship](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

[Raccoon](#) as virtual base

[RAII](#) [See [resource allocation is initialization](#)]

[raise](#)

raise exception [See [throw](#)]

[random file IO](#)

[random-access iterator](#) 2nd

[deque](#)

[string](#)

[vector](#)

[random shuffle.](#)

[range](#)

[iterator](#) 2nd 3rd

[left-inclusive](#)

[range error.](#)

[rbegin](#), [container](#) 2nd

[rdstate](#)

[recursive function](#) 2nd

[refactoring](#) 2nd

[reference](#)

[reference](#) 2nd

[and pointer](#)

[const](#) [reference](#)

[initialization](#)

[conversion from derived to multiple base](#)

[conversion from derived to base](#)

[dynamic cast operator](#), [example](#)

[is polymorphic](#)

[nonconst](#) [reference](#)

[parameter](#)

[pointer parameter](#)

[return type and class object](#)

[return type and local variable](#)

[return type, is lvalue](#)

[return value](#)

[to array parameter](#)

[reference count](#) [See [use count](#)]

[reference data member](#), [initialization](#)

[reference return](#)

[reference to const](#) [See [const reference](#)]

[reinterpret cast](#), 2nd

[relational operator](#)

[associative container](#)

[container](#)

[container adaptor](#)

[function object](#)

[overloaded operator](#) 2nd

[consistent with equality](#)

[pointer](#)

[string](#)

[remove](#)

R

list
remove copy.
remove copy if.
remove if.
 list
remove Msg from Folder.
rend, container 2nd
replace 2nd
 string
replace copy, 2nd
replace copy if.
replace if.
reserve
 string
 vector
reserved identifier
resize, sequential container
Resource
resource allocation is initialization
 auto_ptr.
restricted word count program
result 2nd
rethrow 2nd
return statement
 from main
 implicit
 local variable 2nd
return type 2nd 3rd 4th
 const reference
 function
 function pointer
 linkage directive
 member function definition
 no implicit return type
 nonreference
 of virtual function
 pointer to function
 reference
 reference yields lvalue
 uses copy constructor
 void
return value
 conversion
 copied
return, container
reverse
 list
reverse iterator 2nd 3rd
 ++ (increment)
 -- (decrement)
 base
 compared to iterator 2nd
 example
 requires -- (decrement)
reverse copy.
reverse iterator.
 container

[rfind](#) [string](#)
[rwd](#) [program](#)
[right manipulator](#)
[right-shift \(>>\) 2nd](#) [3rd](#) [4th](#)
[scope \(::\)](#) [2nd](#) [3rd](#)
 [class member](#) [2nd](#)
 [container defined type](#)
 [member function definition](#)
 [to override name lookup](#)
[shift](#) [2nd](#)
[sizeof](#)
[subscript \(\[\]\)](#)
 [and multi-dimensioned array](#)
 [and pointer](#)
 [array](#)
 [bitset](#)
 [deque](#)
 [map](#)
 [string](#)
 [valid subscript range](#)
 [vector](#) [2nd](#)
 [yields lvalue](#)
[subtraction \(-\)](#)
 [iterator](#) [2nd](#)
 [pointer](#)
[unary](#) [2nd](#)
[unary minus \(-\)](#)
[unary plus \(+\)](#)
[rotate](#)
[rotate copy](#)
[rule of three](#) [2nd](#)
 [exception for virtual destructors](#)
[run time](#)
 [error](#)
[run-time type identification](#) [2nd](#)
 [classes with virtual functions](#)
 [compared to virtual functions](#)
 [dynamic cast](#)
 [example](#)
 [throws bad cast](#)
 [to pointer](#)
 [to reference](#)
[type-sensitive equality](#)
[typeid](#)
 [and virtual functions](#)
 [example](#)
 [returns type info](#)
[runtime error](#) [2nd](#)
 [constructor from string](#)
[rvalue](#) [2nd](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

[safety_exception](#)

[Sales_item](#)

[addition \(+\) 2nd](#)

[throws exception 2nd](#)

[avg_price definition](#)

[class definition 2nd](#)

[compare function](#)

[compound assignment \(e.g., +=\)](#)

[conversion](#)

[default constructor](#)

[equality operators \(==\), \(!=\)](#)

[explicit constructor](#)

[handle class](#)

[clone function](#)

[constructor 2nd](#)

[definition](#)

[design](#)

[multiset of](#)

[using generic Handle](#)

[header 2nd 3rd](#)

[input \(>>\)](#)

[istream constructor](#)

[no relational operators](#)

[operations](#)

[output \(<<\)](#)

[same isbn 2nd](#)

[string constructor](#)

[scientific manipulator](#)

[scope 2nd](#)

[block](#)

[class 2nd 3rd](#)

[compared to object lifetime](#)

[const object 2nd](#)

[for statement](#)

[friend declaration](#)

[function](#)

[function template specialization](#)

[global 2nd](#)

[local 2nd](#)

[multiple inheritance](#)

[namespace](#)

[statement](#)

[template parameter](#)

[using declaration](#)

[using directive](#)

[example](#)

[name collisions](#)

[scope \(::\)](#)

[base class members](#)

S

[namespace member](#)
[scope operator \(::\)](#) 2nd 3rd 4th
[class member](#) 2nd
[container defined type](#)
[member function definition](#)
[namespace member](#)
[to override class-specific memory allocation](#)
[to override name lookup](#)

[Screen](#)

[CachedObj](#)
[class template](#)
[concatenating operations](#)
[display](#)
[do display](#)
[friends](#)
[get definition](#)
[get members](#)
[get cursor definition](#)
[Menu function table](#)
[move members](#)
[set members](#)
[simplified](#)
[size type](#)

[ScreenPtr](#)

[arrow operator \(->\)](#)
[dereference \(*\)](#)
[use counted](#)

[ScrPtr](#)

[search](#)
[search n](#)
[seek and tell members](#)

self-assignment

[auto ptr](#)
[check](#)
[use counting](#)

[semantics, value](#) 2nd

[semicolon \(;\)](#)

[semicolon \(;\), class definition](#)

[sentinel](#) 2nd

[separate compilation](#) 2nd

[inclusion model for templates](#)
[of templates](#)
[separate compilation model for templates](#) 2nd

[sequence \(\xnnn\), hexadecimal escape](#)

[sequence, escape](#)

[sequential container](#) 2nd

[assign](#)
[assignment \(=\)](#)
[back](#)
[clear](#)
[const iterator](#)
[const reverse iterator](#)

constructor from element count

[uses copy constructor](#)
[uses element default constructor](#)

[constructors](#)

[dequeue](#)

[element type constraints 2nd](#)

[empty](#)

[erase](#)

[front](#)

[insert](#)

[iterator](#)

[list](#)

[operations](#)

[performance characteristics](#)

[pop back](#)

[pop front](#)

[priority queue](#)

[push back](#)

[push front](#)

[queue](#)

[rbegin](#)

[rend](#)

[resize](#)

[returning a](#)

[reverse iterator 2nd](#)

[size](#)

[size type](#)

[stack](#)

[supports relational operators](#)

[swap](#)

[types defined by](#)

[value type](#)

[vector](#)

[set 2nd](#)

[as element type](#)

[assignment \(=\)](#)

[begin](#)

[bidirectional iterator](#)

[clear](#)

[constructors](#)

[count](#)

[element type constraints](#)

[empty](#)

[end](#)

[equal range](#)

[erase 2nd](#)

[find](#)

[insert](#)

[iterator](#)

[key type constraints](#)

[lower bound](#)

[operations](#)

[overriding the default comparison](#)

[rbegin](#)

[rend](#)

[return alternatives](#)

[return type from insert](#)

[reverse iterator](#)

[size](#)

[supports relational operators](#)

[swap](#)

S

[upper bound](#)
[value type](#)
[set difference](#)
[set intersection 2nd](#)
[set symmetric difference](#)
[set union](#)
[setfill manipulator](#)
[setprecision manipulator](#)
[setstate 2nd](#)

[setw manipulator](#)
[shift operator 2nd](#)

[short](#)
[short-circuit evaluation](#)
 [overloaded operator](#)

[shorterString](#)

[showbase manipulator](#)
[showpoint manipulator](#)

[signed 2nd](#)
 [conversion to unsigned 2nd](#)

[size](#)
 [associative container](#)
 [priority queue](#)
 [queue](#)
 [sequential container](#)
 [stack](#)
 [string](#)
 [vector](#)

[size_t 2nd 3rd](#)
 [and array](#)

[size_type 2nd](#)
 [container](#)
 [string](#)
 [vector](#)

[sizeof operator](#)
[skipws manipulator](#)

[sliced 2nd](#)
[SmallInt 2nd](#)

[conversion operator](#)
[smart pointer 2nd 3rd](#)
 [handle class](#)
 [overloaded \(++\) and \(*\)](#)
 [overloaded -> \(arrow operator\) and * \(dereference\)](#)

[sort 2nd](#)
[source file 2nd](#)
 [naming convention](#)

specialization

[class template](#)
 [definition](#)
 [member defined outside class body](#)
 [partial](#)

[partial specialization](#)

[class template member](#)
 [declaration](#)

[function template](#)
 [compared to overloaded function](#)
 [declaration 2nd](#)
 [example](#)

[scope](#)
[template, namespaces](#)

[specifier, type](#) 2nd

[splice, list](#)

[sstream](#)

[header](#) 2nd

[str](#)

[stable partition](#)

[stable sort](#) 2nd

[stack](#) 2nd

[constructors](#)

[relational operator](#)

[stack unwinding](#) 2nd

[standard error](#) 2nd

[standard input](#) 2nd

[standard library](#) 2nd

[standard output](#) 2nd

[state, condition](#)

[statement](#) 2nd

[break](#) 2nd

[compound](#) 2nd

[continue](#) 2nd

[declaration](#) 2nd

[do while](#)

[expression](#) 2nd

[for](#) 2nd

[for statement](#)_{for}

[goto](#) 2nd

[if](#) 2nd 3rd 4th

[labeled](#) 2nd

[null](#) 2nd

[return](#)

[return, local variable](#) 2nd

[switch](#) 2nd

[while](#) 2nd 3rd 4th

[statement block](#) [See [block](#)]

[statement label](#)

[statement scope](#)

[statement](#)_{for} [statement, for](#)

[static](#)

[static \(file static\)](#)

[static class member](#) 2nd

[as default argument](#)

[class template](#)

[accessed through an instantiation](#)

[definition](#)

[const](#) [data member, initialization](#)

[const](#) [member function](#)

[data member](#)

[as constant expression](#)

[inheritance](#)

[member function](#)

[this pointer](#)

[static object, local](#) 2nd

[static type](#) 2nd

[determines name lookup](#)

[multiple inheritance](#)

[static type checking](#) 2nd

S

[argument](#)
[function return value](#)

[static cast](#) 2nd

[std](#) 2nd

[stdexcept header](#) 2nd

[store, free](#) 2nd

[str](#)

[strcat](#)

[strcmp](#)

[strcpy](#)

[stream](#)

[flushing buffer](#)

[istream iterator](#)

[iterator](#) 2nd

[and class type](#)

[limitations](#)

[used with algorithms](#)

[not flushed if program crashes](#)

[ostream iterator](#)

[type as condition](#)

[stream iterator](#)

[strict weak ordering](#) 2nd

[string](#)

[addition](#)

[addition to string literal](#)

[and string literal](#) 2nd

[append](#)

[are case sensitive](#)

[as sequential container](#)

[assign](#)

[assignment \(=\)](#)

[c_str](#)

[c_str, example](#)

[capacity](#)

[compare](#)

[compared to C-style string](#)

[compound assignment](#)

[concatenation](#)

[constructor](#) 2nd

[default constructor](#)

[empty](#)

[equality \(==\)](#)

[equality operator](#)

[erase](#)

[find](#)

[find first not of](#)

[find first of](#)

[find last not of](#)

[find last of](#)

[getline](#)

[getline, example](#)

[header](#)

[input operation as condition](#)

[input operator](#)

[insert](#)

[output operator](#)

[random-access iterator](#)

[relational operator](#) 2nd

[replace](#)
[reserve](#)
[rfind](#)
[size](#)
[size_type](#)
[subscript operator](#)
[substr](#)

[string literal](#) 2nd 3rd
 [addition to string](#)
 [and C-style string](#)
 [and string library type](#) 2nd
 [concatenation](#)

[string, C-style](#) [See [C-style string](#)]

[stringstream](#) 2nd 3rd [See also [istream](#), [ostream](#)]

[str](#)
[strlen](#)
[strncat](#)
[strncpy](#)

[struct](#) [See also [class](#)]

[default access label](#)
 [default inheritance access label](#)

[struct](#), keyword 2nd 3rd
 [in variable definition](#)

[structure, data](#) 2nd

[Studio, Visual](#)

[subscript \(\[\] \)](#) 2nd 3rd 4th

[and multi-dimensioned array](#)
 [and pointer](#)
 [array](#)
 [bitset](#)
 [deque](#)

[map](#)
 [overloaded operator](#)
 [reference return](#)

[string](#)
 [valid subscript range](#)
 [vector](#) 2nd
 [yields lvalue](#)

[subscript range](#)

[array](#)
 [string](#)
 [vector](#)

[substr, string](#)

[subtraction \(-\)](#)
 [iterator](#) 2nd
 [pointer](#)

[swap](#) 2nd
 [container](#)
 [swap program](#) 2nd

[swap ranges](#)

[switch statement](#) 2nd

[and break](#)
 [case label](#)
 [compared to if](#)
 [default label](#)
 [execution flow](#)
 [expression](#)

[variable definition](#)

[synthesized assignment \(=\) 2nd](#)

[multiple inheritance](#)

[pointer members](#)

[synthesized copy constructor 2nd](#)

[multiple inheritance](#)

[pointer members](#)

[virtual base class](#)

[synthesized copy control, volatile](#)

[synthesized default constructor 2nd 3rd 4th](#)

[inheritance](#)

[synthesized destructor 2nd](#)

[multiple inheritance](#)

[pointer members](#)

Team LiB

◀ PREVIOUS NEXT ▶

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[table of library name and header](#)

[template](#) [See also [class template](#), [function template](#), [instantiation](#)]

[class 2nd](#)

[class member](#) [See [member template](#)]

[link time errors](#)

[overview](#)

[template argument 2nd](#)

[and function argument type checking](#)

[class template](#)

[conversion](#)

[deduction](#)

[from function pointer](#)

[deduction for class template member function](#)

[deduction for function template](#)

[explicit and class template](#)

[explicit and function template](#)

[and function pointer](#)

[specifying](#)

[pointer](#)

[template argument deduction](#)

[template class](#) [See [class template](#)]

[template function](#) [See [function template](#)]

[template keyword](#)

[template parameter 2nd 3rd](#)

[and member templates](#)

[name](#)

[restrictions on use](#)

[nontype parameter 2nd 3rd 4th](#)

[class template](#)

[must be constant expression](#)

[scope](#)

[type parameter 2nd 3rd 4th](#)

[uses of inside class definition](#)

[template parameter list 2nd](#)

[template specialization 2nd](#)

[class member declaration](#)

[compared to overloaded function](#)

[definition](#)

[example](#)

[function declaration 2nd](#)

[member defined outside class body](#)

[member of class template](#)

[parameter matching](#)

[partial specialization 2nd](#)

[scope](#)

[template<>](#) [See [template specialization](#)]

[temporary object](#)

[terminate 2nd 3rd 4th 5th](#)

[TextQuery](#)

[class definition](#)

[main program using](#)

[program design](#)
[program interface](#)
[revisited](#)
[this pointer](#)
[implicit 2nd](#)
[implicit parameter 2nd](#)
[in overloaded operator](#)
[overloaded operator](#)
[static member functions](#)

[three, rule of 2nd](#)

[tHRow 2nd 3rd 4th 5th](#)

[example 2nd](#)

[execution flow 2nd](#)

[pointer to local object](#)

[rethrow](#)

[tolower](#)

[top](#)

[priority queue](#)

[stack](#)

[toupper](#)

[TRansform](#)

[transformation program, word](#)

[translation unit \[See \[source file\]\(#\)\]](#)

[trunc \(file mode\)](#)

[TRy block 2nd 3rd 4th](#)

[TRy keyword](#)

[type](#)

[abstract data 2nd](#)

[arithmetic 2nd](#)

[built-in 2nd 3rd](#)

[class 2nd 3rd](#)

[compound 2nd 3rd](#)

[dynamic 2nd](#)

[function return](#)

[incomplete 2nd](#)

[integral 2nd](#)

[library](#)

[nested \[See \[nested class\]\(#\)\]](#)

[return 2nd 3rd 4th](#)

[static 2nd](#)

[determines name lookup](#)

[name lookup and multiple inheritance](#)

[type checking](#)

[argument](#)

[with class type conversion](#)

[ellipsis parameter](#)

[name lookup](#)

[reference to array argument](#)

[type identification, run-time 2nd](#)

[type specifier 2nd](#)

[type template parameter 2nd 3rd \[See also \[template parameter\]\(#\)\]](#)

[type info](#)

[header](#)

[name member](#)

[no copy or assign](#)

[operations](#)

[returned from \[typeid\]\(#\)](#)

[typedef 2nd](#)

[typedef](#)

[and pointer](#)
[and pointer to member](#)
[pointer to function](#)
[typeid operator 2nd](#)
[and virtual functions](#)
[example](#)
[returns type info](#)

`typename`, keyword

[compared to class](#)

[in template parameter](#)

[inside template definition](#)

Team LiB

[!\[\]\(1d304e9d41cfa06aa6c1f595e5c320a2_img.jpg\) PREVIOUS](#) [!\[\]\(070de65041cedd1f7e1ad7493d3f9fbb_img.jpg\) NEXT !\[\]\(86e35f281f0e37c5bc9f46965f621e2a_img.jpg\)](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

[U_Ptr](#)

[unary function object](#)

[unary minus \(-\)](#)

[unary operator 2nd](#)

[unary plus \(+\)](#)

[uncaught exception](#)

[undefined behavior 2nd](#)

[dangling pointer](#)

[synthesized copy control](#)

[invalidated iterator](#)

[uninitialized class data member](#)

[uninitialized pointer](#)

[uninitialized variable](#)

[underflow error](#)

[unexpected 2nd](#)

[uninitialized 2nd 3rd 4th](#)

[uninitialized pointer](#)

[uninitialized copy 2nd](#)

[uninitialized fill](#)

[union 2nd](#)

[anonymous 2nd](#)

[as nested type](#)

[example](#)

[limitations on](#)

[union keyword](#)

[unique 2nd](#)

[list](#)

[unique copy 2nd](#)

[unitbuf, manipulator flushes the buffer](#)

[unnamed namespace 2nd](#)

[local to file](#)

[replace file static](#)

[unsigned 2nd](#)

[conversion to signed 2nd](#)

[literal \(numu or numu\)](#)

[unsigned char](#)

[unwinding, stack 2nd](#)

[upper bound](#)

[associative container](#)

[book finding program](#)

[example](#)

[uppercase manipulator](#)

[use count 2nd](#)

[design overview](#)

[generic class](#)

[held in companion class](#)

[pointer to](#)

[self-assignment check](#)

[user 2nd](#)

[using declaration 2nd 3rd 4th](#)

[access control](#)

[class member access](#)

[in header](#)

[overloaded function](#)

[overloaded inherited functions](#)

[scope](#)

[using directive 2nd](#)

[overloaded function](#)

[pitfalls](#)

[scope](#)

[example](#)

[name collisions](#)

[utility header](#)

Team LiB

◀ PREVIOUS NEXT ▶

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[value initialization](#) 2nd

[and dynamically allocated array](#)

[and resize](#)

[deque](#)[dequeue](#)

[list](#)[list](#)

[map subscript operator](#)

[of dynamically allocated object](#)

[sequential container](#)

[vector](#)

[vector](#)[vector](#)

[value semantics](#) 2nd

[value type](#)

[map](#), [multimap](#)

[sequential container](#)

[set](#), [multiset](#)

[varargs](#)

[variable](#) 2nd 3rd

[define before use](#)

[defined after case label](#)

[definition](#)

[definitions and goto](#)

[vector](#) 2nd

[Vector](#)

[vector](#)

[argument](#)

[as element type](#)

[assig_n](#)

[assignment \(=\)](#)

[at](#)

[back](#)

[begin](#) 2nd

[capacity](#)

[Vector](#)

[capacity](#)

[vector](#)

[clear](#)

[const iterator](#) 2nd

[const reference](#)

[const reverse iterator](#)

[constructor from element count, uses copy constructor](#)

[constructor taking iterators](#)

[constructors](#) 2nd

[difference type](#)

[element type constraints](#) 2nd

[empty](#) 2nd

[end](#) 2nd

[erase](#) 2nd

[invalidates iterator](#)

front
header
initialization from pointer
insert
 invalidates iterator
iterator 2nd
iterator supports arithmetic
memory allocation strategy

Vector
 memory allocation strategy

vector
 memory management strategy
parameter
performance characteristics
pop back
 push back 2nd

Vector
 push back

vector
 push_back
 invalidates iterator
 random-access iterator
 rbegin 2nd

Vector
 reallocate

vector
 reference
 relational operators
 rend 2nd
 reserve
 resize
 reverse iterator 2nd
 size 2nd

Vector
 size

vector
 size_type 2nd
 subscript ([])
 subscript operator
 supports relational operators
 swap
 type
 types defined by

Vector
 using explicit destructor call
 using operator new and delete
 using placement new

vector
 value type

vector capacity program

viable function 2nd
 with class type conversion

virtual base class 2nd
 ambiguities
 conversion
 defining base as
 derived class constructor
 name lookup

[order of construction](#)
[stream types](#)
[virtual function 2nd 3rd](#)
 [assignment operator](#)
 [calls resolved at run time](#)
 [compared to run-time type identification](#)
 [default argument](#)
 [derived classes](#)
 [destructor](#)
 [multiple inheritance](#)
 [exception specifications](#)
 [in constructors](#)
 [in destructor](#)
 [introduction](#)
 [multiple inheritance](#)
 [no virtual constructor](#)
 [overloaded](#)
 [overloaded operator](#)
 [overriding run-time binding](#)
 [pure 2nd](#)
 [example](#)
 [return type](#)
 [run-time type identification](#)
 [scope](#)
 [static](#)
 [to copy unknown type](#)
 [type-sensitive equality](#)
[virtual inheritance 2nd](#)
[virtual keyword](#)
[Visual Studio](#)
[void 2nd](#)
 [return type](#)
[void* 2nd](#)
 [const void* 2nd](#)
[volatile 2nd](#)
 [pointer](#)
 [synthesized copy control](#)
[vowel counting program](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

[wcerr](#)

[wchar_t](#)

[literal](#)

[wchar_t streams](#)

[wcin](#)

[wcout](#)

[weak ordering, strict 2nd](#)

[wfstream](#)

[what](#) [See [exception](#)]

[while statement](#) 2nd 3rd 4th

[condition in](#)

[whitespace](#)

[wide character streams](#)

[wifstream](#)

[window, console](#)

[Window Mar](#)

[wiostream](#)

[wistream](#)

[wistringstream](#)

[wofstream](#)

[word 2nd](#)

[word count program](#)

[restricted](#)

[word per line processing](#)

[istringstream](#)

[istringstream](#)[istringstream](#)

[word transformation program](#)

[WordQuery](#)

[definition](#)

[woostream](#)

[wostringstream](#)

[wrap around](#)

[wstringstream](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

[ZooAnimal class hierarchy](#)([ZooAnimal](#), 使用虚继承)

[ZooAnimal, using virtual inheritance](#)([ZooAnimal](#) 类层次)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[abnormal termination, stream buffers](#) (异常终止, 流缓冲区)

[abort](#) 2nd

[absInt](#)

[abstract base class](#) (抽象基类) 2nd

[example](#) (例子)

[abstract data type](#) (抽象数据类型) 2nd 3rd

[abstraction, data](#) (抽象, 数据) 2nd

[access control](#) (访问控制)

[in base and derived classes](#) (在基类和派生类中)

[local class](#) (局部类)

[nested class](#) (嵌套类)

[using declarations to adjust](#) ([using](#) 声明来调整)

[access label](#) (访问标号) 2nd 3rd 4th

[private](#) 2nd

[protected](#) 2nd

[public](#) 2nd

[Account](#)

[accumulate](#) 2nd

[Action](#)

[adaptor](#) (适配器) 2nd

[container](#) (容器)

[function](#) (函数) 2nd 3rd

[iterator](#) (迭代器)

[addition \(+\)](#) (加)

[iterator](#) (迭代器) 2nd

[pointer](#) (指针)

[Sales item](#)

[string](#)

[address](#) (地址) 2nd

[address-of \(&\)](#) (取地址) 2nd

[overloaded operator](#) (重载操作符)

[adjacent difference](#)

[adjacent find](#)

[algorithm](#) (算法) 2nd

[copy versions](#) ([copy](#) 版本) 2nd

[if versions](#) ([if](#) 版本)

[element type constraints](#) (元素类型约束)

[independent of container](#) (独立于容器)

[iterator argument constraints](#) (迭代器实参约束) 2nd

[iterator category and](#) (迭代器种类和) 2nd

[naming convention](#) (命名规范)

[overloaded versions](#) (重载版本)

[parameter pattern](#) (形参模式)

[passing comparison function](#)

[read-only](#) (只读)

[structure](#) (结构)

[that reorders elements](#) (对元素进行排序的算法)

[that writes elements](#) (写元素的算法)

[type independence \(类型独立性\) 2nd](#)[using function object as argument \(用函数对象作为实参\)](#)[with two input ranges \(带两个输入范围\)](#)[algorithm header \(algorithm 头文件\)](#)[algorithms](#)[binary](#)[library defined](#)[overloaded operator](#)[example](#)[unary](#)[alias, namespace \(别名, 命名空间\) 2nd](#)[allocator 2nd 3rd](#)[allocate](#)[compared to operator new \(与 operator new 比较\)](#)[construct 2nd](#)[compared to placement new \(与定位 new 表达式比较\)](#)[deallocate](#)[compared to operator delete \(与 operator delete 比较\)](#)[destroy 2nd](#)[compared to calling destructor \(与调用析构函数比较\)](#)[operations \(操作\)](#)[alternative operator name \(可选择的操作符名字\)](#)[ambiguous \(二义性\)](#)[conversion \(转换\)](#)[multiple inheritance \(多重继承\)](#)[function call \(调用\) 2nd 3rd](#)[multiple base classes \(多个基类\)](#)[overloaded operator \(重载操作符\)](#)[AndQuery](#)[definition \(定义\)](#)[eval function \(eval 函数\)](#)[anonymous union \(匿名联合\) 2nd](#)[app \(file mode\) \(文件模式\)](#)[append, string](#)[arc](#)[argument \(实参\) 2nd 3rd 4th 5th](#)[array type \(数组类型\)](#)[C-style string \(C 风格字符串\)](#)[const reference type \(const 引用类型\)](#)[conversion \(转换\)](#)[with class type conversion \(用类类型转换\)](#)[copied \(复制的\)](#)[uses copy constructor \(使用复制构造函数\)](#)[default \(默认\)](#)[iterator \(迭代器\) 2nd](#)[multi-dimensioned array \(多维数组\)](#)[nonconst reference parameter](#)[passing \(传递\)](#)[pointer to const \(指向 const 对象的指针\)](#)[pointer to nonconst \(指向非 const 对象的指针\)](#)[reference parameter \(引用形参参数\)](#)[template \[See template argument\]](#)[to main \(main 的\)](#)[to member function \(成员函数的\)](#)[type checking \(类型检查\)](#)[ellipsis \(省略符\)](#)[of array type](#)

(数组类型的)

of reference to array (数组引用的)

with class type conversion (用类类型转换的)

argument deduction, template (模板实参推断)

argument list (实参表)

argv

arithmetic (算术)

iterator (迭代器) 2nd 3rd 4th

pointer (指针) 2nd

arithmetic operator (算术操作符)

and compound assignment (和复合赋值)

function object (函数对象)

overloaded operator (重载操作符)

arithmetic type (算术类型) 2nd

conversion (转换) 2nd

from bool (从 bool 转换)

signed to unsigned (signed 到 unsigned)

conversion to bool (转换到 bool)

array (数组) 2nd 3rd

and pointer (和指针)

argument (实参)

as initializer of vector (作为 vector 的初始化式)

assignment (赋值)

associative (关联数组)

conversion to pointer (转换到指针) 2nd

and template argument (和模板实参)

copy (复制)

default initialization (默认初始化)

uses copy constructor (使用复制构造函数)

uses default constructor (使用默认构造函数)

definition (定义)

elements and destructor (元素与析构函数)

function returning (函数返回)

initialization (初始化)

multi-dimensioned (多维数组)

and pointer (和指针)

definition (定义)

initialization (初始化)

parameter (形参)

subscript operator (下标操作符)

of char initialization (char (数组) 的初始化)

parameter (形参)

buffer overflow (缓冲区溢出)

convention (规范)

reference type (引用类型)

size calculation (大小计算)

subscript operator (下标操作符)

arrow operator (>) (箭头操作符)

auto_ptr

class member access (类成员访问)

generic handle (泛型句柄)

overloaded operator (重载操作符)

assert preprocessor macro (`assert` 预处理宏) 2nd

assign

container (容器)

string

assignment (赋值)

[memberwise \(逐个成员\) 2nd](#)[vs. initialization \(与初始化\)](#)[assignment \(=\) \(赋值\) 2nd 3rd 4th](#)[and conversion \(和转换\)](#)[and copy constructor \(和复制构造函数\)](#)[check for self-assignment \(检查自身赋值\)](#)[container \(容器\)](#)[for derived class \(派生类的\)](#)[Message](#)[multiple inheritance \(多重继承\)](#)[overloaded operator \(重载操作符\) 2nd 3rd](#)[reference return \(引用返回\) 2nd](#)[pointer \(指针\)](#)[rule of three \(三法则\)](#)[exception for virtual destructors \(虚析构函数的异常\)](#)[string](#)[synthesized \(合成赋值\) 2nd](#)[to base from derived \(从派生类到基类 \(赋值\) \)](#)[to signed \(给 signed \(赋值\) \)](#)[to unsigned \(给 unsigned \(赋值\) \)](#)[use counting \(使用计数\) 2nd](#)[usually not virtual \(通常不为虚\)](#)[value-like classes \(值型类\)](#)[yields lvalue \(生成左值\)](#)[associative array \(关联数组\) \[See \[map\]\(#\)\]](#)[associative container \(关联容器\) 2nd](#)[assignment \(=\) \(赋值\)](#)[begin](#)[clear](#)[constructors \(构造函数\)](#)[count](#)[element type constraints \(元素类型约束\) 2nd](#)[empty](#)[equal_range](#)[erase](#)[find](#)[insert](#)[key type constraints \(键类型约束\)](#)[lower_bound](#)[operations \(操作\)](#)[overriding the default comparison \(覆盖默认比较\)](#)[rbegin](#)[rend](#)[returning an \(返回一个关联容器\)](#)[reverse_iterator](#)[size](#)[supports relational operators \(支持关系操作符\)](#)[swap](#)[types defined by \(关联容器定义的类型\)](#)[upper_bound](#)[associativity \(结合性\) 2nd 3rd](#)[overloaded operator \(重载操作符\)](#)[at](#)[deque](#)[vector](#)[ate \(file mode\) ate](#)

(文件模式)

[auto_ptr](#) [2nd](#)

[constructor](#) (构造函数)

[copy and assignment](#) (复制与赋值)

[default constructor](#) (默认构造函数)

[get member](#) ([get](#) 成员)

[operations](#) (操作)

[pitfalls](#) (缺陷)

[reset member](#) ([reset](#) 成员)

[self-assignment](#) (自身赋值)

[automatic object](#) (自动对象) [2nd](#) [See also [local variable](#), [parameter](#)]

[and destructor](#) (和析构函数)

Team LiB

◀ PREVIOUS NEXT ▶

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

[back](#)

[queue](#)

[sequential container](#)

[back inserter](#) 2nd 3rd

[bad](#)

[bad alloc](#) 2nd

[bad cast](#) 2nd

[bad typeid](#)

[badbit](#)

[base](#)

[base class](#) 2nd 3rd 4th [See also [virtual function](#)]

[abstract](#) 2nd

[example](#)

[access control](#) 2nd

[assignment operator, usually not virtual](#)

[can be a derived class](#)

[constructor](#)

[calls virtual function](#)

[not virtual](#)

[conversion from derived](#)

[access control](#)

[definition](#)

[destructor](#)

[calls virtual function](#)

[usually virtual](#)

[friendship not inherited](#)

[handle class](#)

[member hidden by derived](#)

[member operator delete](#)

[multiple](#) [See [multiple base class](#)]

[must be complete type](#)

[no conversion to derived](#)

[object initialized or assigned from derived](#)

[scope](#)

[static members](#)

[user](#)

[virtual](#) [See [virtual base class](#)]

[Basket](#)

[total function](#)

[Bear](#)

[as virtual base](#)

[begin](#)

[container](#)

[map](#)

[set](#)

[vector](#)

[best match](#) 2nd [See also [function matching](#)]

[bidirectional iterator](#) 2nd

[container](#)

[list](#)

[map](#)
[set](#)
[binary \(file mode\)](#)
[binary function object](#)
[binary operator 2nd](#)
[binary search](#)
[BinaryQuery](#)
 [definition](#)
[bind1st](#)
[bind2nd](#)
[binder 2nd](#)
[binding, dynamic 2nd](#)
 [requirements for](#)
[bit-field 2nd](#)
 [access to](#)
[bitset 2nd 3rd](#)
 [any](#)
 [compared to bitwise operator](#)
 [constructor](#)
 [count](#)
 [flip](#)
 [compared to bitwise NOT](#)
 [header](#)
 [none](#)
 [output operator](#)
 [reset](#)
 [set](#)
 [size](#)
 [subscript operator](#)
 [test](#)
 [to ulong](#)
[bitwise AND \(&\) 2nd](#)
 [example](#)
[bitwise exclusive or \(^\) 2nd](#)
[bitwise NOT \(~\) 2nd](#)
 [example](#)
[bitwise operator](#)
 [and compound assignment](#)
 [compared to bitset](#)
 [compound assignment](#)
 [example](#)
 [operand](#)
[bitwise OR \(|\) 2nd](#)
 [example 2nd](#)
[block 2nd 3rd 4th 5th](#)
 [as target of if](#)
 [function](#)
 [TRY 2nd 3rd 4th](#)
[block scope](#)
[body, function 2nd 3rd 4th](#)
[book finding program](#)
 [using equal range](#)
 [using find](#)
 [using upper bound](#)
[bookstore program](#)
 [exception classes](#)
[bool](#)
 [and equality operator](#)

[conversion to arithmetic type](#)

[literal](#)

[boolalpha manipulator](#)

[brace, curly 2nd](#)

[break statement 2nd](#)

[and switch](#)

[buffer 2nd](#)

[flushing](#)

[buffer overflow](#)

[and C-style string](#)

[array parameter](#)

[built-in type 2nd 3rd](#)

[class member default initialization](#)

[conversion](#)

[initialization of](#)

[Bulk_item](#)

[class definition](#)

[constructor](#)

[constructor using default arguments](#)

[derived from Disc_item](#)

[interface](#)

[member functions](#)

[byte 2nd](#)

Team LiB

◀ PREVIOUS NEXT ▶

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[C library header](#)

[C with classes](#)

C++

[calling C function from C++](#)

[compiling C and C++](#)

[using C linkage](#)

[C-style cast](#)

[C-style string](#) 2nd 3rd 4th 5th

[and char*](#)

[and string literal](#)

[compared to `string`](#) 2nd

[definition](#)

[dynamically allocated](#)

[initialization](#)

[parameter](#)

[pitfalls with generic programs](#)

[c_str](#)

[example](#)

[CachedObj](#)

[add to freelist](#)

[allocation explained](#)

[definition](#)

[definition of `static` members](#)

[design](#)

[illustration](#)

[inheriting from](#)

[operator delete](#)

[operator new](#)

[call operator \(\)](#) 2nd 3rd 4th

[execution flow](#)

[overloaded operator](#)

[calling C function from C++](#)

[candidate function](#) 2nd

[and function templates](#)

[namespaces](#)

[overloaded operator](#)

[capacity](#)

[string](#)

[vector](#)

[case label](#) 2nd 3rd

[default](#)

[cassert header](#)

[cast](#) 2nd

checked [See [dynamic_cast](#)]

[old-style](#)

[catch clause](#) 2nd 3rd 4th 5th

[catch\(...\)](#) 2nd

[example](#)

[exception specifier](#)

[matching](#)

[ordering of parameter](#)
[category iterator](#)
[cctype 2nd header](#)
[cerr 2nd](#)
[char literal](#)
char string literal [See [string literal](#)]

character

[newline \(\n\)](#)
[nonprintable 2nd](#)
[null \(\0\)](#)
[printable](#)
[tab \(\t\)](#)

checked cast [See [dynamic cast](#)]

[CheckedPtr](#)

[children's story program revisited](#)

[cin 2nd by default tied to cout](#)

[cl](#)

[class 2nd 3rd 4th 5th](#)

[abstract base](#)

[example](#)

[access labels 2nd](#)

[as friend](#)

[2nd 3rd \[See base class\]](#)

[concrete](#)

[conversion](#)

[multiple conversions lead to ambiguities](#)

[conversion constructor](#)

[function matching](#)

[with standard conversion](#)

[data member 2nd](#)

[const vs. mutable](#)

[const, initialization](#)

[constraints on type](#)

[definition](#)

[initialization](#)

[mutable](#)

[reference, initialization](#)

[static](#)

[data member definition](#)

[default access label](#)

[default inheritance access label](#)

[definition 2nd](#)

[and header 2nd](#)

[2nd 3rd \[See derived class\]](#)

[destructor definition](#)

direct base [See [immediate base class](#)]

[explicit constructor](#)

[forward declaration](#)

[generic handle 2nd](#)

[2nd \[See handle class\]](#)

[immediate base 2nd](#)

[indirect base 2nd](#)

local [See [local class](#)]

[member 2nd 3rd](#)

[member access](#)

[2nd \[See member function\]](#)

[member : constant expression \[See bit-field\]](#)

[multiple inheritance \[See multiple base class\]](#)

[nested \[See nested class\]](#)

[nonvirtual function, calls resolved at compile time](#)

[operator delete \[See member operator\]](#)

[operator new \[See member operator new\]](#)

[pointer member](#)

[copy control](#)

[copy control strategies](#)

[default copy behavior](#)

[pointer to member](#)

[definition](#)

[pointer to member function, definition](#)

[preventing copies](#)

[private member](#)

[private member](#)

[inheritance](#)

[protected member](#)

[public member 2nd](#)

[inheritance](#)

[static member 2nd](#)

[as default argument](#)

[data member as constant expression](#)

[example](#)

[inheritance](#)

[template member \[See member template\]](#)

[type member](#)

[undefined member](#)

[user 2nd](#)

[virtual base](#)

[virtual function, calls resolved at run time](#)

[class declaration 2nd](#)

[of derived class](#)

[class derivation list 2nd](#)

[access control](#)

[default access label](#)

[multiple base classes](#)

[virtual base](#)

[class keyword](#)

[class member : constant expression \[See bit-field\]](#)

[class scope 2nd 3rd](#)

[friend declaration](#)

[inheritance](#)

[member definition](#)

[name lookup](#)

[static members](#)

[virtual functions](#)

[class template 2nd 3rd 4th \[See also template parameter, template argument, instantiation\]](#)

[compiler error detection](#)

[declaration](#)

[definition](#)

[error detection](#)

[explicit template argument](#)

[export](#)

[friend](#)

[declaration dependencies](#)

[explicit template instantiation](#)

[nontemplate class or function](#)

[template class or function](#)

[member function](#)

[defined outside class body](#)

[instantiation](#)

[member specialization](#)

member template [See [member template](#)]

[nontype template parameter](#)

[static member](#)

[accessed through an instantiation](#)

[definition](#)

[type includes template argument\(s\)](#) 2nd

[type-dependent code](#)

[uses of template parameter](#)

class template specialization

[definition](#)

[member defined outside class body](#)

[member, declaration](#)

[namespaces](#)

[class type](#) 2nd 3rd 4th 5th 6th

[class member default initialization](#)

[conversion](#)

[design considerations](#)

[example](#)

[initialization of](#)

[multiple conversions lead to ambiguities](#)

[object definition](#)

[operator](#) 2nd 3rd

[operator and function matching](#)

[parameter and overloaded operator](#)

[used implicitly](#)

[variable vs. function declaration](#)

[with standard conversion](#)

[class, keyword](#)

[compared to](#) [typename](#)

[in template parameter](#)

[in variable definition](#)

cleanup, object [See [destructor](#)]

[clear](#) 2nd

[associative container](#)

[example](#) 2nd

[sequential container](#)

[close](#) 2nd

[close](#)

[comma operator \(, \)](#) 2nd

[example](#)

[operand order of evaluation](#)

[overloaded operator](#)

[comment](#) 2nd

[block \(/* * */ \)](#) 2nd

[single-line \(// \)](#) 2nd

[compare](#)

[plain function](#)

[string](#)

[template version](#)

[instantiated with pointer](#)

[specialization](#)

compilation

[and header](#)

[conditional](#)
[inclusion model for templates](#)
[needed when class changes](#)
[needed when inline function changes](#)
[separate 2nd
of templates](#)
[separate model for templates](#)

compiler
[extension](#)
[flag for inclusion compilation model](#)
[GNU](#)
[Microsoft](#)
[template errors diagnosed at link time](#)

compiler extension
[compiling C and C++](#)
composition vs. inheritance
[compound assignment \(e.g., +=\) 2nd 3rd](#)

[bitwise operator](#)
[iterator](#)
[overloaded operator 2nd](#)
[Sales item](#)
[string](#)

[compound expression 2nd](#)
[compound statement 2nd](#)
[compound type 2nd 3rd](#)
[compute](#)

[overloaded version](#)

concatenation

[Screen operations](#)
[string](#)
[string literal](#)

[concrete class](#)
[initialization](#)

[condition 2nd](#)
[and conversion](#)
[assignment in](#)
[in do while statement](#)
[in for statement 2nd](#)
[in if statement 2nd](#)
[in logical operator](#)
[in while statement](#)
[stream type as 2nd 3rd](#)
[string input operation as](#)

[condition state 2nd](#)
[conditional compilation](#)
[conditional operator \(? :\) 2nd](#)

[operand order of evaluation](#)
[console window](#)
[const](#)
[and dynamically allocated array](#)
[conversion to 2nd](#)
[and template argument](#)
[iterator vs. const iterator](#)
[object scope 2nd](#)
[overloading and 2nd](#)
[parameter](#)
[pointer](#)
[reference](#)
[initialization](#)

const data member
[compared to mutable](#)
[initialization](#)
[static data member](#)

const member function 2nd 3rd 4th 5th

const object, constructor

const pointer [See also [pointer to const](#)]
[conversion from non const](#)

const reference
[argument](#)
[conversion from non const](#)
[parameter](#)
[overloading](#)
[return type](#)

const void* 2nd

const cast 2nd

const iterator 2nd
[compared to const iterator](#)
[container](#)

const reference

const reverse iterator
[container](#)

constant expression 2nd
[and header file](#)
[array index](#)
[bit-field](#)
[enumerator](#)
[nontype template parameter](#)
[static data member](#)

construction, order of 2nd
[derived objects 2nd](#)
[multiple base classes](#)
[virtual base classes](#)

constructor 2nd 3rd 4th 5th 6th
[const objects](#)
[conversion 2nd](#)
[function matching](#)
[with standard conversion](#)

copy 2nd
[base from derived](#)
[multiple inheritance](#)

default 2nd 3rd 4th 5th 6th
[default argument in](#)
[derived class](#)
[initializes immediate base class](#)
[initializes virtual base](#)

execution flow
[explicit 2nd](#)
[copy-initialization](#)

for associative container
for sequential container
[function matching](#)
[function try block](#)

in constructor initializer list
[inheritance](#)
[initializer](#)
[may not be virtual](#)
[object creation](#)

[order of construction](#)
[derived objects 2nd](#)
[multiple base classes](#)
[virtual base classes](#)
[overloaded](#)
[pair](#)
[resource allocation](#)
[synthesized copy 2nd](#)
[synthesized default 2nd 3rd 4th](#)
[virtual inheritance](#)
[with standard conversion](#)
[constructor initializer list 2nd 3rd 4th 5th](#)
[compared to assignment](#)
[derived classes](#)
[function `try` block](#)
[initializers](#)
[multiple base classes](#)
[sometimes required](#)
[virtual base class](#)
[container 2nd 3rd 4th](#) [See also [sequential container](#), [associative container](#)]
[and generic algorithms](#)
[as element type](#)
[assignment \(`=`\)](#)
[associative 2nd](#)
[begin](#)
[clear](#)
[const iterator](#)
[const reference](#)
[const reverse iterator](#)
[element type constraints 2nd](#)
[elements and destructor](#)
[elements are copies](#)
[empty](#)
[end](#)
[erase](#)
[has bidirectional iterator](#)
[inheritance](#)
[insert](#)
[iterator](#)
[rbegin 2nd](#)
[reference](#)
[rend 2nd](#)
[returning a](#)
[reverse iterator 2nd](#)
[sequential 2nd](#)
[size](#)
[size type](#)
[supports relational operators](#)
[swap](#)
[types defined by](#)
[continue statement 2nd](#)
[example](#)
[control flow of 2nd](#)
[conversion 2nd](#)
[ambiguous](#)
[and assignment](#)
[argument](#)
[with class type conversion](#)

[arithmetic type 2nd](#)
[array to pointer 2nd](#)
 and template argument
[conversion constructor](#)
[copy](#)
[copy constructor 2nd 3rd](#)
 and assignment operator
 argument passing
 base from derived
 for derived class
 initialization
[Message](#)

[copy control 2nd](#)
 handle class
 inheritance
 message handling example
 multiple inheritance
 of pointer members

[copy-initialization](#)
 using constructor

[copy backward](#)

[count](#)
 book finding program
 map
 multimap
 multiset
 set

[count, use 2nd](#)
[count if 2nd](#)
 with function object argument

[cout 2nd](#)
 by default tied to [cin](#)

[cstddef header 2nd](#)
[cstdlib header](#)
[cstring header](#)

[ctrl-d \(Unix end-of-file\)](#)
[ctrl-z \(Windows end-of-file\)](#)

[curly brace 2nd](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[dangling else](#) 2nd

[dangling pointer](#) 2nd

- [returning pointer to local variable](#)

- [synthesized copy control](#)

[data abstraction](#) 2nd

- [advantages](#)

[data hiding](#)

[data structure](#) 2nd

[data type, abstract](#)

[ddd.ddd L or ddd.dddI \(long double literal\)](#)

[dec manipulator](#)

[decimal literal](#)

[declaration](#) 2nd

- [class](#) 2nd

- [class template member specialization](#)

- [dependencies and template friends](#)

- [derived class](#)

- [export](#)

- [forward](#) 2nd

- [function](#)

- [exception specification](#)

- [function template specialization](#) 2nd

- [member template](#)

- [template](#)

- [using](#) 2nd 3rd 4th

- [access control](#)

- [class member access](#)

- [overloaded inherited functions](#)

[declaration statement](#) 2nd

[decrement \(- - \)](#)

- [iterator](#)

- [overloaded operator](#)

- [prefix yields lvalue](#)

- [reverse iterator](#)

[deduction, template argument](#)

[default argument](#)

- [and header file](#)

- [function matching](#)

- [in constructor](#)

- [initializer](#)

- [overloaded function](#)

- [virtual functions](#)

[default case label](#) 2nd

[default constructor](#) 2nd 3rd 4th 5th 6th

- [default argument](#)

- [Sales item](#)

- [string](#) 2nd

- [synthesized](#) 2nd 3rd 4th

- [used implicitly](#)

- [variable definition](#)

[definition 2nd](#)
[array](#)
[base class](#)
[C-style string](#)
[class 2nd](#)
[class data member 2nd](#)
[class static member](#)
[class template](#)
 [static member](#)
[class template specialization](#)
 [member defined outside class body](#)
[class type object](#)
[derived class](#)
[destructor](#)
[dynamically allocated array](#)
[dynamically allocated object](#)
[function](#)
[inside a switch expression](#)
[inside a while condition](#)
[inside an if condition](#)
[map 2nd](#)
[multi-dimensioned array](#)
[namespace](#)
 [can be discontiguous](#)
 [member](#)
[of variable after case label](#)
[overloaded operator](#)
[pair](#)
[pointer](#)
[pointer to function](#)
[static data member](#)
[variable](#)
[delete 2nd 3rd 4th](#)
 [compared to operator delete](#)
 [const object](#)
 [execution flow](#)
 [member operator](#)
 [and inheritance](#)
 [interface](#)
[memory leak 2nd](#)
[null pointer](#)
[runs destructor](#)
[delete \[\]](#)
 [and dynamically allocated array](#)
[dequeue](#)
 [as element type](#)
 [assign](#)
 [assignment \(=\)](#)
 [at](#)
 [back](#)
 [begin](#)
 [clear](#)
 [const iterator](#)
 [const reference](#)
 [const reverse iterator](#)
[constructor from element count, uses copy constructor](#)
[constructors](#)
[difference type](#)

[element type constraints 2nd](#)

[empty](#)

[end](#)

[erase](#)

[invalidates iterator](#)

[front](#)

[insert](#)

[invalidates iterator](#)

[iterator](#)

[iterator supports arithmetic](#)

[performance characteristics](#)

[pop back](#)

[pop front](#)

[push back](#)

[invalidates iterator](#)

[push front](#)

[invalidates iterator](#)

[random-access iterator](#)

[rbegin 2nd](#)

[reference](#)

[relational operators](#)

[rend 2nd](#)

[resize](#)

[reverse iterator 2nd](#)

[size](#)

[size type](#)

[subscript \(\[\]\)](#)

[supports relational operators](#)

[swap](#)

[types defined by](#)

[value type](#)

[dereference \(*\) 2nd 3rd 4th](#)

[and increment](#)

[auto_ptr](#)

[iterator](#)

[on map iterator yields pair](#)

[overloaded operator](#)

[pointer](#)

[yields lvalue 2nd](#)

[derivation list, class 2nd](#)

[access control](#)

[default access label](#)

[derived class 2nd 3rd 4th \[See also virtual function\]](#)

[access control 2nd](#)

[as base class](#)

[assigned or copied to base object](#)

[assignment \(=\)](#)

[constructor](#)

[calls virtual function](#)

[for remote virtual base](#)

[initializes immediate base class](#)

[constructor initializer list](#)

[conversion to base](#)

[access control](#)

[copy constructor](#)

[default derivation label](#)

[definition](#)

[destructor](#)

[calls virtual function](#)
[friendship not inherited](#)
[handle class](#)
[member hides member in base](#)
[member operator delete](#)
[multiple base classes](#)
[no conversion from base](#)
[scope](#)
[scope \(::\) to access base class member](#)
[static members](#)

[using declaration](#)
[inherited functions](#)
[member access](#)
[with remote virtual base](#)

derived object

[contains base part](#)
[multiple base classes, contains base part for each](#)

[derived to base 2nd](#)

[access control](#)
[enumeration type to integer](#)
[from istream](#)
[function matching of template and nontemplate functions](#)
[function to pointer](#)
[and template argument](#)
[implicit](#)
[inheritance](#)
[integral promotion](#)
[multi-dimensioned array to pointer](#)
[multiple inheritance](#)
[nontemplate type argument](#)
[of return value](#)
[rank for function matching](#)
[rank of class type conversions](#)
[signed to unsigned](#)
[signed type](#)
[template argument](#)
[to const](#)
[and template argument](#)
[parameter matching](#)
[to const pointer](#)
[virtual base](#)

design

[CachedObj](#)
[class member access control](#)
[class type conversions](#)
[consistent definitions of equality and relational operators](#)
[is-a relationship](#)
[Message class](#)
[namespace](#)
[of handle classes](#)
[of header files](#)
[export](#)
[inclusion compilation model](#)
[separate compilation model](#)
[optimizing new and delete](#)
[using freelist](#)
[overloaded operator](#)
[overview of use counting](#)
[Query classes](#)

[Queue](#)

[resource allocation is initialization](#)

[Sales item handle class](#)

[TextQuery class](#)

[vector memory allocation strategy](#)

[writing generic code](#)

[pointer template argument](#)

[destruction, order of](#)

[derived objects](#)

[multiple base classes](#)

[virtual base classes](#)

[destructor 2nd 3rd](#)

[called during exception handling](#)

[container elements](#)

[definition](#)

[derived class](#)

[explicit call to](#)

[implicitly called](#)

[library classes](#)

[Message](#)

[multiple inheritance](#)

[order of destruction](#)

[derived objects](#)

[multiple base classes](#)

[virtual base classes](#)

[resource deallocation](#)

[rule of three](#)

[exception for virtual destructors](#)

[should not throw exception](#)

[synthesized 2nd](#)

[use counting 2nd](#)

[valuelike classes](#)

[virtual in base class](#)

[virtual, multiple inheritance](#)

[development environment, integrated](#)

[difference type 2nd 3rd](#)

[dimension 2nd](#)

[direct base class \[See \[immediate base class\]\(#\)\]](#)

[direct-initialization](#)

[using constructor](#)

[directive, using 2nd](#)

[pitfalls](#)

[Disc item](#)

[class definition](#)

[discriminant 2nd](#)

[divides<T>](#)

[division \(/\) 2nd](#)

[do while statement](#)

[condition in](#)

[domain error](#)

[dot \(.\) 2nd](#)

[class member access](#)

[dot operator \(.\) 2nd](#)

[class member access](#)

[double](#)

[literal \(numE num or numE num\)](#)

[long double](#)

[notation output format control](#)

[output format control](#)

[duplicate word program](#)

[revisited](#)

[dynamic binding 2nd](#)

[in C++](#)

[requirements for](#)

[dynamic type 2nd](#)

[dynamic cast 2nd 3rd](#)

[example](#)

[throws bad cast](#)

[to pointer](#)

[to reference](#)

[dynamically allocated](#)

[array 2nd](#)

[definition](#)

[delete](#)

[const object](#)

[initialization](#)

[of const](#)

Team LiB

◀ PREVIOUS NEXT ▶

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[edit-compile-debug](#) 2nd

 errors at link time

[else](#) [See [if statement](#)]

 dangling 2nd

[empty](#)

[associative container](#)

[container](#)

[priority queue](#)

[queue](#)

[stack](#)

[string](#) 2nd

[vector](#) 2nd

[encapsulation](#) 2nd

 advantages

[end](#)

[container](#)

[map](#)

[set](#)

[vector](#)

[end-of-file](#) 2nd 3rd

 entering from keyboard

[Endangered](#)

[endl](#)

 manipulator flushes the buffer

[ends](#), manipulator flushes the buffer

[enum](#) keyword

[enumeration](#) 2nd

 conversion to integer

 function matching

[enumerator](#) 2nd

 conversion to integer

[environment, integrated development](#)

[eof](#)

[eofbit](#)

[equal](#)

[equal member function](#)

[equal range](#)

[associative container](#)

[book finding program](#)

[equal to<T>](#)

[equality \(==\)](#) 2nd 3rd 4th

[algorithm](#) 2nd

 and assignment

[container](#) 2nd

[container adaptor](#) 2nd

[iterator](#) 2nd 3rd 4th

 overloaded operator 2nd

 consistent with equality

[string](#) 2nd 3rd 4th

[erase](#)

[associative container](#)

[container](#)

[invalidates iterator](#)

[map](#)

[multimap](#)

[multiset](#)

[sequential container](#)

[set](#)

[string](#)

[error, standard](#)

[escape sequence 2nd](#)

[hexadecimal \(\xnnn\)](#)

[octal \(\ nnn\)](#)

[evaluation](#)

[order of 2nd](#)

[short-circuit](#)

[exception](#)

[class 2nd](#)

[class hierarchy](#)

[constructor](#)

[extending the hierarchy](#)

[header](#)

[what member 2nd](#)

[exception handling 2nd](#) [See also [throw](#), [catch clause](#)]

[and terminate](#)

[compared to assert](#)

[exception in destructor](#)

[finding a catch clause](#)

[function try block 2nd](#)

[handler](#) [See [catch clause](#)]

[library class destructors](#)

[local objects destroyed](#)

[specifier 2nd 3rd 4th](#)

[nonreference](#)

[reference](#)

[types related by inheritance](#)

[stack unwinding](#)

[uncaught exception](#)

[unhandled exception](#)

[exception object 2nd](#)

[array or function](#)

[initializes catch parameter](#)

[must be copyable](#)

[pointer to local object](#)

[rethrow](#)

[exception safety 2nd](#)

[exception specification 2nd](#)

[function pointers](#)

[tHROW\(\)](#)

[unexpected](#)

[violation](#)

[virtual functions](#)

[exception, raise](#) [See [throw](#)]

[executable file](#)

[EXIT_FAILURE](#)

[EXIT_SUCCESS](#)

[explicit constructor 2nd](#)

[copy-initialization](#)

[export](#)
[and header design](#)
[keyword 2nd](#)
[exporting C++ to C](#)
[expression 2nd 3rd 4th](#)
[and operand conversion](#)
[compound 2nd](#)
[constant 2nd](#)
[throw 2nd](#)
[expression statement 2nd](#)
[extended compute](#)
[extension compiler](#)
[extern](#)
[extern 'C' \[See \[linkage directive\]\(#\)\]](#)
[extern const](#)

Team LiB

◀ PREVIOUS NEXT ▶

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[factorial program](#)

[fail](#)

[failbit](#)

[file](#)

[executable](#)

[object](#)

[source 2nd](#)

[file mode 2nd](#)

[combinations](#)

[example](#)

[file static 2nd](#)

[fill](#)

[fill n](#)

[find 2nd](#)

[book finding program](#)

[map](#)

[multimap](#)

[multiset](#)

[set](#)

[string](#)

[find last word program](#)

[find end](#)

[find first not of, string](#)

[find first of 2nd 3rd](#)

[string](#)

[find if 2nd](#)

[find last not of, string](#)

[find last of, string](#)

[find val program](#)

[fixed manipulator](#)

[float](#)

[literal \(num f or num f\)](#)

[floating point](#)

[notation output format control](#)

[output format control](#)

[floating point literal \[See \[double literal\]\(#\)\]](#)

[flow of control 2nd](#)

[flush, manipulator flushes the buffer](#)

[Folder \[See \[Message\]\(#\)\]](#)

[for statement 2nd](#)

[condition in](#)

[execution flow](#)

[expression](#)

[for header](#)

[initialization statement](#)

[scope](#)

[for statement](#)[for statement](#)

[for each](#)

[format state](#)

[forward declaration of class type](#)

[forward iterator 2nd](#)

[fp compute](#)

[free store 2nd](#)

[freelist 2nd](#)

[friend 2nd](#)

[class](#)

[class template](#)

[explicit template instantiation](#)

[nontemplate class or function](#)

[template class or function](#)

[function template, example](#)

[inheritance](#)

[member function](#)

[overloaded function](#)

[overloaded operator](#)

[scope considerations](#)

[namespaces](#)

[template example](#)

[friend keyword](#)

[front](#)

[queue](#)

[sequential container](#)

[front inserter 2nd](#)

[compared to inserter](#)

[fstream 2nd 3rd](#) [See also [istream](#), [ostream](#)]

[close](#)

[constructor](#)

[file marker](#)

[file mode](#)

[combinations](#)

[example](#)

[file random access](#)

[header 2nd](#)

[off type](#)

[open](#)

[pos type](#)

[random IO sample program](#)

[seek and tell members](#)

[function 2nd 3rd 4th](#)

[calls resolved at run time](#)

[candidate 2nd](#)

[compared to run-time type identification](#)

[conversion to pointer](#)

[and template argument](#)

[default argument](#)

[derived classes](#)

[destructor](#)

[destructor and multiple inheritance](#)

[equal member](#)

[exception specifications](#)

[function returning](#)

[in constructors](#)

[in destructor](#)

[inline 2nd](#)

[inline and header](#)

[introduction](#)

[2nd 3rd](#) [See [member function](#)]

[multiple inheritance](#)

[no virtual constructor](#)
[nonvirtual, calls resolved at compile time](#)
[overloaded 2nd 3rd](#)
 [compared to redeclaration](#)
 [friend declaration](#)
 [scope](#)
 [virtual](#)
[overloaded operator](#)
[overriding run-time binding](#)
[pure virtual 2nd](#)
 [example](#)
[recursive 2nd](#)
[return type](#)
[run-time type identification](#)
[scope](#)
[to copy unknown type](#)
[type-sensitive equality](#)
[viable 2nd](#)
[virtual 2nd 3rd](#)
 [assignment operator](#)
[function adaptor 2nd 3rd](#)
 [bind1st](#)
 [bind2nd](#)
 [binder](#)
 [negator](#)
 [not1](#)
 [not2](#)
[function body 2nd 3rd 4th](#)
[function call](#)
 [ambiguous 2nd](#)
 [execution flow](#)
 [overhead](#)
 [through pointer to function](#)
 [through pointer to member](#)
 [to overloaded operator](#)
 [to overloaded postfix operator](#)
 [using default argument](#)
[function declaration](#)
 [and header file](#)
 [exception specification](#)
[function definition](#)
[function matching 2nd](#)
 [and overloaded function templates](#)
 [examples](#)
[argument conversion](#)
[conversion operator](#)
[conversion rank](#)
 [class type conversions](#)
[enumeration parameter](#)
[integral promotion](#)
[multiple parameters](#)
[namespaces](#)
[of member functions](#)
[overloaded operator](#)
[function name 2nd](#)
[function object 2nd](#)
[function pointer](#)
 [and template argument deduction](#)
 [definition](#)
 [exception specifications](#)

[function returning initialization](#)
[overloaded functions](#)
[parameter](#)
[return type](#)
[typedef](#)
[function prototype 2nd](#)
[function return type 2nd 3rd](#)
 [const reference](#)
 [no implicit return type](#)
 [nonreference](#)
 [uses copy constructor](#)
 [reference](#)
 [reference yields lvalue](#)
 [void](#)
[function scope](#)
[function table](#)
 [pointer to member](#)
[function template 2nd](#) [See also [template parameter](#), [template argument](#), [instantiation](#)]
 [as friend](#)
 [compiler error detection](#)
 [declaration](#)
 [error detection](#)
 [explicit template argument](#)
 [and function pointer](#)
 [specifying](#)
 [export](#)
 [inline](#)
 [instantiation](#)
 [template argument deduction](#)
 [type-dependent code](#)
[function template specialization](#)
 [compared to overloaded function](#)
 [declaration 2nd](#)
 [example](#)
 [namespaces](#)
 [scope](#)
[function TRy block 2nd](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[q++](#)

[acd program](#)

[generate](#)

[generate_n](#)

generic algorithm [See [algorithm](#)]

[generic handle class](#) 2nd

generic memory management [See [CachedObj](#)]

[generic programming](#) 2nd

[and pointer template argument](#)

[type-independent code](#)

[getline](#) 2nd

[example](#) 2nd

[global namespace](#) 2nd

[global scope](#) 2nd

[global variable, lifetime](#)

[GNU compiler](#)

[good](#)

[goto statement](#) 2nd

[greater-than \(>\)](#) 2nd 3rd 4th

[greater-than-or-equal \(>=\)](#) 2nd 3rd 4th

[greater<T>](#)

[greater_equal<T>](#)

[GT6 program](#)

[GT_cls](#)

[guard header](#) 2nd

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[Handle](#)

[int instantiation](#)

[operations](#)

[Sales item instantiation](#)

[handle class](#) 2nd

[copy control](#)

[copying unknown type](#)

[design](#)

[generic](#) 2nd

[that hides inheritance hierarchy](#)

[using a](#)

[handler](#) [See [catch clause](#)]

[has-a relationship](#)

[HasPtr](#)

[as a smart pointer](#)

[using synthesized copy control](#)

[with value semantics](#)

[header](#) 2nd 3rd 4th

[algorithm](#)

[and constant expression](#)

[and library names](#)

[bitset](#)

[C library](#)

[cassert](#)

[cctype](#) 2nd

[class definition](#) 2nd

[cstddef](#) 2nd

[cstdlib](#)

[cstring](#)

[default argument](#)

[dequeue](#)

[design](#)

[export](#)

[inclusion compilation model](#)

[namespace members](#)

[separate compilation model](#)

[exception](#)

[fstream](#) 2nd

[function declaration](#)

[inline function](#)

[inline member function definition](#)

[iomanip](#)

[iostream](#)

[iterator](#)

[list](#)

[map](#) 2nd

[new](#)

[numeric](#)

[programmer-defined](#)

[queue](#)
[Sales item](#) 2nd 3rd
[set](#) 2nd
[sstream](#) 2nd
[stack](#)
[stdexcept](#) 2nd
[string](#)
[type info](#)
[using declaration](#)
[utility](#)
[vector](#) 2nd

[header file, naming convention](#)

[header guard](#) 2nd

[heap](#) 2nd

[hex manipulator](#)

[hexadecimal escape sequence \(\xnnn\)](#)

[hexadecimal literal \(0x num or 0x num\).](#)

[hides, names in base hidden by names in derived](#)

[hierarchy, inheritance](#) 2nd 3rd

[high-order bits](#) 2nd

Team LiB

◀ PREVIOUS NEXT ▶

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

IDE

[identification, run-time type](#) [2nd](#)

[identifier](#) [2nd](#)

[naming convention](#)

[reserved](#)

[if statement](#) [2nd](#) [3rd](#) [4th](#)

[compared to](#) [switch](#)

[dangling else](#)

[else branch](#) [2nd](#)

[if statement, else branch](#)

[ifstream](#) [2nd](#) [See also [istream](#)]

[close](#)

[constructor](#)

[file marker](#)

[file mode](#)

[combinations](#)

[example](#)

[file random access](#)

[off type](#)

[open](#)

[pos type](#)

[random IO sample program](#)

[seek and tell members](#)

[immediate base class](#) [2nd](#)

[implementation](#) [2nd](#) [3rd](#)

[implementation inheritance](#)

[implicit conversion](#) [See [conversion](#)]

[implicit return](#)

[from main allowed](#)

[implicit this pointer](#) [2nd](#) [3rd](#) [4th](#)

[in and overloaded operator](#)

[static member functions](#)

[implicit this pointer, overloaded operator](#)

[in \(file mode\)](#)

[include](#) [See [#include](#)]

[includes](#)

[inclusion compilation model](#) [2nd](#)

[incomplete type](#) [2nd](#)

[restriction on use](#) [2nd](#) [3rd](#)

[increment](#) [\(++\)](#) [2nd](#) [3rd](#) [4th](#) [5th](#) [6th](#)

[and dereference](#) [2nd](#)

[iterator](#) [2nd](#) [3rd](#) [4th](#) [5th](#)

[overloaded operator](#)

[pointer](#) [2nd](#)

[prefix yields lvalue](#) [2nd](#)

[reverse iterator](#) [2nd](#)

[indentation](#) [2nd](#)

[index](#) [2nd](#)

[indirect base class](#) [2nd](#)

[inequality \(!=\)](#) 2nd 3rd 4th

[container](#) 2nd

[container adaptor](#) 2nd

[iterator](#) 2nd 3rd 4th

[overloaded operator](#) 2nd

[string](#) 2nd

[inheritance](#) 2nd

[containers](#)

[conversions](#)

[default access label](#)

[friends](#)

[handle class](#)

[implementation](#)

[interface](#)

[iostream diagram](#)

[multiple](#) [See [multiple base class](#)]

[private](#)

[static members](#)

[virtual](#) 2nd

[inheritance hierarchy](#) 2nd 3rd

[inheritance vs. composition](#)

[initialization](#) 2nd 3rd 4th

[array](#)

[array of char](#)

[built-in type](#)

[C-style string](#)

[class data member](#)

[class member of built-in type](#)

[class member of class type](#)

[class type](#) 2nd

[const static data member](#)

[definitions and goto](#)

[constructor](#)

[dynamically allocated array](#)

[dynamically allocated object](#)

[local](#) 2nd

[map](#)

[memberwise](#) 2nd

[multi-dimensioned array](#)

[objects of concrete class type](#)

[pair](#)

[parameter](#)

[pointer](#)

[pointer to function](#)

[return value](#)

[scope](#)

[value](#) 2nd

[variable](#) 2nd 3rd

[vs. assignment](#)

[initialization vs. assignment](#)

[initialized](#) 2nd

[initializer list, constructor](#) 2nd 3rd 4th 5th

[inline function](#) 2nd

[and header](#)

[function template](#)

[member function](#)

[and header](#)

[inner product](#)

[inplace merge](#)

[input \(>>\)](#) 2nd 3rd 4th
[istream iterator](#)
[istream iterator.](#)
[overloaded operator](#)
[error handling](#)
[must be nonmember](#)
[precedence and associativity](#) 2nd
[Sales item](#)
[Sales item.](#)
[string](#) 2nd 3rd

[input iterator](#) 2nd
[input standard](#)
[insert](#)
[inserter](#)
[invalidates iterator](#)
[map](#)
[multimap](#)
[multiset](#)
[return type from set::insert](#)
[sequential container](#)
[set](#)
[string](#)

[insert iterator](#) 2nd 3rd
[inserter](#)
[inserter](#)
[compared to front inserter](#)

[instantiation](#) 2nd
[class template](#) 2nd 3rd
[member function](#)
[nontype parameter](#)
[type](#)
[error detection](#)
[function template](#)
[from function pointer](#)
[nontemplate argument conversion](#)
[nontype template parameter](#)
[template argument conversion](#)
[member template](#)
[nested class template](#) 2nd
[on use](#)
[static class member](#)

[int](#)
[literal](#)
[Integral](#)

[integral promotion](#) 2nd
[function matching](#)

[integral type](#) 2nd
[integrated development environment](#)
[interface](#) 2nd 3rd
[interface inheritance](#)
[internal manipulator](#)
[interval, left-inclusive](#) 2nd
[invalid argument](#)
[invalidated iterator](#) 2nd
IO stream [See [stream](#)]
[iomanip header](#)
[iostate](#)
[iostream](#) 2nd 3rd [See also [istream](#), [ostream](#)]

[header](#)
[inheritance hierarchy](#)
[seek and tell members](#)
[is-a relationship](#)
[isalnum](#)
[isalpha](#)
[ISBN](#)
[isbn mismatch](#)
 [destructor explained](#)
[iscntrl](#)
[isdigit](#)
[isgraph](#)
[islower](#)
[isprint](#)
[ispunct](#)
[isShorter program 2nd](#)
[isspace](#)
[istream 2nd 3rd \[See also manipulator\]](#)
 [condition state](#)
 [flushing input buffer](#)
 [format state](#)
 [aCount](#)
 [get](#)
 [multi-byte version](#)
 [returns int 2nd](#)
 [getline 2nd](#)
 [getline, example](#)
 [ignore](#)
[inheritance hierarchy](#)
[input \(>>\)](#)
 [precedence and associativity](#)
[no containers of](#)
[no copy or assign](#)
[peek](#)
[put](#)
[putback](#)
[read](#)
[seek and tell members](#)
[unformatted operation](#)
 [multi-byte](#)
 [single-byte](#)
[unget](#)
[write](#)
[istream iterator 2nd](#)
 [and class type](#)
 [constructors](#)
 [input iterator](#)
 [input operator \(>>\)](#)
 [limitations](#)
 [operations](#)
 [used with algorithms](#)
[istringstream 2nd 3rd \[See also istream\]](#)
 [str](#)
 [word per line processing 2nd 3rd](#)
[isupper](#)
[isxdigit](#)
[Item_base](#)
 [class definition](#)

[constructor](#)
[interface](#)
[member functions](#)
[iterator swap](#)
[iterator 2nd 3rd 4th 5th](#)
[iterator 2nd](#)
iterator
 [argument](#)
 [arrow \(->\)](#)
 [bidirectional 2nd](#)
 [compared to reverse iterator 2nd](#)
[iterator](#)
 [container](#)
iterator
 [destination 2nd](#)
 [equality 2nd](#)
 [forward 2nd](#)
 [generic algorithms](#)
 [inequality 2nd](#)
 [input 2nd](#)
 [insert 2nd 3rd](#)
 [invalidated 2nd](#)
invalidated by
 [assign](#)
 [erase](#)
 [insert](#)
 [push back](#)
 [push front](#)
 [resize](#)
[off-the-end 2nd 3rd](#)
[operations](#)
[output 2nd](#)
[parameter 2nd](#)
[random-access 2nd](#)
[relational operators](#)
[reverse 2nd 3rd](#)
[stream](#)
[iterator arithmetic 2nd 3rd 4th](#)
 [relational operators](#)
[iterator category 2nd](#)
 [algorithm and 2nd](#)
 [bidirectional iterator](#)
 [forward iterator](#)
 [hierarchy](#)
 [input iterator](#)
 [output iterator](#)
 [random-access iterator](#)
[iterator header](#)
[iterator range 2nd 3rd](#)
 [algorithms constraints on 2nd](#)
 [erase](#)
 [generic algorithms](#)
 [insert](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

key type

[associative containers](#)

keyword

[enum](#)

[export](#)

[friend](#)

[namespace](#)

[protected](#)

[template](#)

[try](#)

[union](#)

[virtual](#)

[keyword table](#)

[Koenig lookup](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

[L' c' \(wchar_t literal\)](#)

[label](#)

[access](#) [2nd](#) [3rd](#) [4th](#)

[case](#) [2nd](#) [3rd](#)

[statement](#)

[labeled statement](#) [2nd](#)

[left manipulator](#)

[left-inclusive interval](#) [2nd](#)

[left-shift \(<<\)](#) [2nd](#) [3rd](#) [4th](#)

[length error](#)

[less-than \(<\)](#) [2nd](#) [3rd](#) [4th](#)

[overloaded and containers](#)

[used by algorithm](#) [2nd](#)

[less-than-or-equal \(<=\)](#) [2nd](#) [3rd](#) [4th](#) [5th](#) [6th](#)

[less<T>](#)

[less_equal<T>](#)

[lexicographical compare](#)

[library names to header table](#)

[library type](#)

[library, standard](#) [2nd](#)

[lifetime, object](#) [2nd](#)

[link time errors from template](#)

[linkage directive](#) [2nd](#)

[C++ to C](#)

[compound](#)

[overloaded function](#)

[parameter or return type](#)

[pointer to function](#)

[single](#)

[linking](#) [2nd](#)

[list](#)

[as element type](#)

[assign](#)

[assignment \(=\)](#)

[back](#)

[begin](#)

[bidirectional iterator](#)

[clear](#)

[const iterator](#)

[const reference](#)

[const reverse iterator](#)

[constructor from element count, uses copy constructor](#)

[constructors](#)

[element type constraints](#) [2nd](#)

[empty](#)

[end](#)

[erase](#)

[front](#)

[insert](#)

L

[iterator](#)
[merge](#)
[performance characteristics](#)
[pop back](#)
[pop front](#)
[push back](#)
[push front](#)
[rbegin 2nd](#)
[reference](#)
[relational operators](#)
[remove](#)
[remove_if](#)
[rend 2nd](#)
[resize](#)
[reverse](#)
[reverse_iterator 2nd](#)
[size](#)
[size_type](#)
[specific algorithms](#)
[splice](#)
[swap](#)
[types defined by](#)
[unique](#)
[value_type](#)

[literal 2nd 3rd](#)
[bool](#)
[char](#)
[decimal](#)
[double \(numE num or nume num\)](#)
[float \(numF or numf\)](#)
[hexadecimal \(0x num or 0x num\).](#)
[int](#)
[long \(numL or numl\)](#)
[long double \(ddd.dddL or ddd.ddd1\)](#)
[multi-line](#)
[octal \(0 num\)](#)
[string 2nd 3rd](#)
[unsigned \(numU or numu\)](#)
[wchar_t](#)

[local class 2nd](#)
[access control](#)
[name lookup](#)
[nested class in](#)
[restrictions on](#)

[local scope 2nd](#)
[local static object 2nd](#)

[local variable 2nd](#)
[destructor](#)
[lifetime](#)
[reference return type](#)

[logic_error](#)

[logical AND \(&&\) 2nd](#)
[operand order of evaluation 2nd](#)
[overloaded operator](#)

[logical NOT \(!\) 2nd](#)
[logical operator](#)

[function object](#)
[logical OR \(||\) 2nd](#)
 [operand order of evaluation 2nd](#)
 [overloaded operator](#)
[logical_and<T>](#)
[logical_not<T>](#)
[logical_or<T>](#)
[long](#)
 [literal \(num_L or num₁\)](#)
[long double](#)
 [long double, literal \(ddd.ddd_L or ddd.ddd₁\)](#)
[lookup_name 2nd](#)
 [and templates](#)
 [before type checking 2nd](#)
 [multiple inheritance](#)
 [class member declaration](#)
 [class member definition 2nd](#)
 [class member definition, examples](#)
 [collisions under inheritance](#)
 [depends on static type](#)
 [multiple inheritance](#)
 [inheritance 2nd](#)
 [local class](#)
 [multiple inheritance](#)
 [ambiguous names](#)
 [namespace names](#)
 [argument-dependent lookup](#)
 [nested class](#)
 [overloaded virtual functions](#)
 [virtual inheritance](#)
[low-order bits 2nd](#)
[lower bound](#)
 [associative container](#)
 [book finding program](#)
[lvalue 2nd](#)
 [assignment](#)
 [dereference](#)
 [function reference return type](#)
 [prefix decrement](#)
 [prefix increment](#)
 [subscript](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

machine-dependent

- [bitfield layout](#)
- [char representation](#)
- [division and modulus result](#)
- [end-of-file character](#)
- [iostate type](#)
- [linkage directive language](#)
- [nonzero return from main](#)
- [pre-compiled headers](#)
- [random file access](#)
- [reinterpret cast](#)
- [representation of enum type](#)
- [return from exception what operation](#)
- [signed and out-of-range value](#)
- [signed types and bitwise operators](#)
- [size of arithmetic types](#)
- [template compilation optimization](#)
- [terminate function](#)

- [type info members](#)
- [vector memory allocation size](#)
- [volatile implementation](#)

[magic number](#) 2nd

[main](#) 2nd

- [arguments to](#)
- [not recursive](#)
- [return type](#)
- [return value](#) 2nd
- [returns 0 by default](#)

[make pair](#)

[make plural](#) program

[manip](#)

[manipulator](#) 2nd 3rd

- [boolalpha](#) 2nd
- [change format state](#)
- [dec](#) 2nd
- [endl flushes the buffer](#)
- [ends flushes the buffer](#)
- [fixed](#) 2nd
- [flush flushes the buffer](#)

- [hex](#) 2nd
- [internal](#) 2nd
- [left](#) 2nd
- [noboolalpha](#) 2nd
- [noshowbase](#) 2nd
- [noshowpoint](#) 2nd
- [noskipws](#) 2nd
- [nouppercase](#) 2nd
- [oct](#) 2nd

[right](#) 2nd
[scientific](#) 2nd
[setfill](#) 2nd
[setprecision](#) 2nd
[setw](#) 2nd
[showbase](#) 2nd
[showpoint](#) 2nd
[skipws](#) 2nd
[unitbuf](#) flushes the buffer
[uppercase](#) 2nd

[map](#) 2nd
as element type
assignment (=)
[begin](#)
[bidirectional iterator](#)
[clear](#)
[constructors](#)
[count](#)
[definition](#)
[dereference yields pair](#)
[element type constraints](#)
[empty](#)
[end](#)
[equal range](#)
[erase](#) 2nd
[find](#)
[header](#)
[insert](#)
[iterator](#)
[key type constraints](#)
[key type](#)
[lower bound](#)
[mapped type](#) 2nd
[operations](#)
[overriding the default comparison](#)
[rbegin](#)
[rend](#)
[return type from insert](#)
[reverse iterator](#)
[size](#)
[subscript operator](#)
[supports relational operators](#)
[swap](#)
[upper bound](#)
[value type](#)

[mapped type, map, multimap](#)
[match, best](#) 2nd
[max](#)

[member](#) [See also [class member](#)]
[mutable data](#)
[pointer to](#) 2nd

[member function](#) 2nd 3rd 4th
as friend
[base member hidden by derived](#)
[class template](#)
[defined outside class body](#)

[instantiation](#)
[const](#) 2nd 3rd
[defined outside class body](#) 2nd

[definition](#)
in class scope
name lookup
name lookup, examples

[equal](#)

function template [See [member template](#)]

[implicitly inline](#)

[inline](#)
and header

[overloaded](#)

[overloaded on const](#)

[overloaded operator](#) 2nd

[pointer to definition](#)

[returning *this](#)

[static](#)

[this pointer](#)

[undefined](#)

[member operator delete](#) 2nd

[and inheritance](#)

[CachedObj](#)

[example](#)

[interface](#)

[member operator delete \[\]](#)

[member operator new](#) 2nd

[CachedObj](#)

[example](#)

[interface](#)

[member operator new \[\]](#)

[member template](#) 2nd

[declaration](#)

[defined outside class body](#)

[examples](#)

[instantiation](#)

[template parameters](#)

[memberwise assignment](#) 2nd

[memberwise initialization](#) 2nd

[memory and object construction](#)

[memory exhaustion](#)

[memory leak](#) 2nd

after exception

memory management, generic [See [CachedObj](#)]

[merge](#)

[list](#)

[Message](#)

[assignment operator](#)

[class definition](#)

[copy constructor](#)

[design](#)

[destructor](#)

[put Msg in Folder](#)

[remove Msg from Folder](#)

method [See [member function](#)]

[Microsoft compiler](#)

[min](#)

[min_element](#)

[minus<T>](#)
[mismatch](#)
[mode_file 2nd](#)
[modulus \(%\) 2nd](#)
[modulus<T>](#)
[multi-dimensioned array](#)

[and pointer](#)
[conversion to pointer](#)
[definition](#)
[initialization](#)
[parameter](#)
[subscript operator](#)

[multi-line literal](#)

[multimap 2nd](#)

[assignment \(=\)](#)

[begin](#)

[clear](#)

[constructors](#)

[count](#)

[dereference yields pair](#)

[element type constraints](#)

[empty](#)

[equal range](#)

[erase 2nd](#)

[find](#)

[has no subscript operator](#)

[insert](#)

[iterator 2nd](#)

[key type constraints](#)

[key type](#)

[lower bound](#)

[mapped type](#)

[operations 2nd](#)

[overriding the default comparison](#)

[rbegin](#)

[rend](#)

[return type from insert](#)

[reverse iterator](#)

[size](#)

[supports relational operators](#)

[swap](#)

[upper bound](#)

[value type](#)

[multiple base class](#) [See also [virtual base class](#)]

[ambiguities](#)

[ambiguous conversion](#)

[avoiding potential name ambiguities](#)

[conversions](#)

[definition](#)

[destructor usually virtual](#)

[name lookup](#)

[object composition](#)

[order of construction](#)

[scope](#)

[virtual functions](#)

[multiple inheritance](#) [See [multiple base class](#)]

[multiplication \(*\) 2nd](#)

[multiplies<T>](#),
[multiset](#) 2nd
[assignment \(=\)](#)
[begin](#)
[clear](#)
[constructors](#)
[count](#)
[element type constraints](#)
[end](#)
[equal range.](#)
[erase](#) 2nd
[find](#)
[insert](#)
[iterator](#)
[key type constraints](#)
[lower bound.](#)
[operations](#) 2nd
[overriding the default comparison](#)
[rbegin](#)
[rend](#)
[return type from insert](#)
[reverse iterator.](#)
[Sales item.](#)
[supports relational operators](#)
[swap](#)
[upper bound.](#)
[example](#)
[value type.](#)
[mutable](#) data member 2nd

Team LiB

◀ PREVIOUS NEXT ▶

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

[name lookup](#) 2nd
 [and templates](#)
 [before type checking](#) 2nd
 [multiple inheritance](#)
 [class member declaration](#)
 [class member definition](#) 2nd
 [class member definition, examples](#)
 [collisions under inheritance](#)
 [depends on static type](#)
 [multiple inheritance](#)
 [inheritance](#) 2nd
 [local class](#)
 [multiple inheritance](#)
 [ambiguous names](#)
 [namespace names](#)
 [argument-dependent lookup](#)
 [nested class](#)
 [overloaded virtual functions](#)
 [virtual inheritance](#)
[name resolution](#) [See [name lookup](#)]
[namespace](#) 2nd 3rd 4th
 [class friend declaration scope](#)
 [cplusplus primer](#).
 [definition](#)
 [design](#)
 [discontiguous definition](#)
 [function matching](#)
 [global](#)
 [member](#)
 [member definition](#)
 [outside namespace](#)
 [restrictions](#)
 [nested](#)
 [scope](#)
 [unnamed](#)
 [local to file](#)
 [replace file](#) [static](#)
[namespace alias](#) 2nd
[namespace keyword](#)
[namespace pollution](#) 2nd
[naming convention](#)
 [header file](#)
 [source file](#)
[NDEBUG](#)
[negate<T>](#).
[negator](#) 2nd
[nested class](#) 2nd
 [access control](#)
 [class defined outside enclosing class](#)
 [in class template](#)
 [in local class](#)

[member defined outside class body](#)

[name lookup](#)

[QueueItem example](#)

[relationship to enclosing class 2nd](#)

[scope](#)

[static members](#)

[union](#)

[nested namespace](#)

nested type [See [nested class](#)]

[new 2nd 3rd 4th](#)

[compared to operator new](#)

[execution flow](#)

[header](#)

[member operator](#)

[member operator, interface](#)

[placement 2nd](#)

[compared to construct](#)

[new \[\]](#)

[new failure](#)

[next permutation](#)

[noboolalpha manipulator](#)

[NoDefault](#)

[nonconst reference](#)

[parameter](#)

[limitations](#)

[nonportable](#)

[nonprintable character 2nd](#)

[nonreference](#)

[parameter](#)

[uses copy constructor](#)

[return type](#)

[uses copy constructor](#)

[nontype template parameter 2nd 3rd 4th](#) [See also [template parameter](#)]

[class template](#)

[must be constant expression](#)

[nonvirtual function, calls resolved at compile time](#)

[noshowbase manipulator](#)

[noshowpoint manipulator](#)

[noskiws manipulator](#)

not equal [See [inequality](#)]

[not1](#)

[not2](#)

[not_equal_to<T>](#)

[NotQuery](#)

[definition](#)

[eval function](#)

[nouppercase manipulator](#)

[nth_element](#)

[NULL](#)

[null pointer](#)

[delete of](#)

[null statement 2nd](#)

null-terminated array [See [C-style string](#)]

[number, magic 2nd](#)

[numEnum or nume num \(double literal\)](#)

[numeric header](#)

numeric literal

[float \(numF or numf\).](#)

[long \(num_L or num₁\)](#),
[long double \(ddd.ddd_L or ddd.ddd₁\)](#),
[unsigned \(num_U or num_u\)](#),
[num_F or num_f \(float literal\)](#),
[num_L or num₁ \(long literal\)](#),
[num_U or num_u \(unsigned literal\)](#)

Team LiB

[!\[\]\(74a7694bcbe43f5bfd9a341716246624_img.jpg\) PREVIOUS](#) [!\[\]\(6471ff2c4a60be5cef5b1fd49451cbfe_img.jpg\) NEXT !\[\]\(9d7220db366f487ab76b77737bb80e3a_img.jpg\)](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

[object](#) 2nd 3rd

[auto_ptr](#)

[automatic](#) 2nd

[constructor](#)

[destructor](#)

[exception](#)

[function](#)

[is not polymorphic](#)

[local static](#) 2nd

[temporary](#)

[object cleanup](#) [See [destructor](#)]

[object creation](#)

[constructor](#)

[order of construction](#)

[derived objects](#) 2nd

[multiple base classes](#)

[virtual base classes](#)

[order of destruction](#)

[derived objects](#)

[multiple base classes](#)

[virtual base classes](#)

[object file](#)

[object lifetime](#) 2nd

[and destructor](#)

[compared to scope](#)

[object-oriented programming](#) 2nd 3rd

[key ideas in](#)

[oct manipulator](#)

[octal escape sequence](#) (`\ nnn`)

[octal, literal](#) (`0 num`)

[off-the-end iterator](#) 2nd 3rd

[istream iterator](#).

[off-the-end pointer](#)

[ofstream](#) 2nd [See also [ostream](#)]

[close](#)

[constructor](#)

[file marker](#)

[file mode](#)

[combinations](#)

[example](#)

[file random access](#)

[off type](#).

[open](#)

[pos type](#).

[random IO sample program](#)

[open](#)

[open file program](#)

[open file, example of](#) 2nd

[operand](#) 2nd

[order of evaluation](#)

[comma operator](#)
[conditional operator](#)
[logical operator](#)
[operator 2nd](#)
 [addition \(+\)](#)
 [iterator 2nd](#)
 [pointer](#)
 [string](#)
 [address-of \(&\)](#)
 [arrow \(->\)](#)
 [class member access](#)
 [assignment \(=\) 2nd 3rd](#)
 [and conversion](#)
 [and equality](#)
 [container](#)
 [multiple inheritance](#)
 [pointer](#)
 [string](#)
 [to signed](#)
 [to unsigned](#)
 [yields lvalue](#)
 [binary 2nd](#)
 [bitwise AND \(&\)](#)
 [bitwise exclusive or \(^\)](#)
 [bitwise not \(~\)](#)
 [bitwise OR \(|\)](#)
 [bitwise OR \(|\). example](#)
 [call \(\) 2nd](#)
 [comma \(,\)](#)
 [operand order of evaluation](#)
 [comma \(,\). example](#)
 [compound assginment \(e.g., +=\), iterator](#)
 [compound assignment \(e.g., +=\) 2nd 3rd](#)
 [arithmetic](#)
 [bitwise](#)
 [string](#)
 [conditional \(?:\)](#)
 [operand order of evaluation](#)
 [decrement \(- -\)](#)
 [iterator](#)
 [prefix yields lvalue](#)
 [reverse iterator](#)
 [dereference \(*\)](#)
 [and increment](#)
 [iterator](#)
 [on map yields pair](#)
 [pointer](#)
 [sizeof](#)
 [typeid 2nd](#)
 [yields lvalue 2nd](#)
[operator alternative name](#)
[operator delete \[\] member](#)
[operator delete function 2nd](#)
 [compared to deallocate](#)
 [compared to delete expression](#)
[operator delete member](#)
 [and inheritance](#)

[CachedObj](#)

[example](#)

[interface](#)

[operator new \[\] member](#)

[operator new function 2nd](#)

[compared to allocate](#)

[compared to new expression](#)

[operator new member](#)

[CachedObj](#)

[example](#)

[interface](#)

operator overloading [See [overloaded operator](#)]

[options to main](#)

[order of construction 2nd](#)

[derived objects 2nd](#)

[multiple base classes](#)

[virtual base classes](#)

[order of destruction 2nd](#)

[derived objects](#)

[multiple base classes](#)

[virtual base classes](#)

[order of evaluation 2nd](#)

[comma operator](#)

[conditional operator](#)

[logical operator](#)

[ordering, strict weak 2nd](#)

[OrQuery](#)

[definition](#)

[eval function](#)

[ostream 2nd 3rd](#) [See also [manipulator](#)]

[condition state](#)

[floatfield member](#)

[flushing output buffer](#)

[format state](#)

[inheritance hierarchy](#)

[no containers of](#)

[no copy or assign](#)

[not flushed if program crashes](#)

[output \(<<\)](#)

[precedence and associativity](#)

[precision member](#)

[seek and tell members](#)

[tie member](#)

[unsetf member](#)

[ostream iterator. 2nd](#)

[and class type](#)

[constructors](#)

[limitations](#)

[operations](#)

[output iterator](#)

[output operator \(<<\)](#)

[used with algorithms](#)

[ostringstream 2nd](#) [See also [ostream](#)]

[str](#)

[out \(file mode\)](#)

[out of stock.](#)

[out of range. 2nd](#)

[output \(<<\) 2nd 3rd 4th](#)

[bitset](#) 2nd
[ostream iterator](#), 2nd
[overloaded operator](#)
 [formatting](#)
 [must be nonmember](#)
[precedence and associativity](#) 2nd
[Sales item.](#)
[string](#) 2nd 3rd
[output iterator](#) 2nd
[output, standard](#)
[overflow](#)
[overflow error.](#)
overload resolution [See [function matching](#)]
[overloaded](#) 2nd 3rd 4th
[overloaded function](#) 2nd
 [compared to redeclaration](#)
 [compared to template specialization](#)
 [friend declaration](#)
 [linkage directive](#)
 [namespaces](#)
 [scope](#)
 [using declarations](#)
 [using directive](#)
 [virtual](#)
[overloaded member function](#)
 [on const](#)
[overloaded operator](#) 2nd 3rd
 [& \(address-of\)](#)
 [&& \(logical AND\)](#)
 [\(\) \(call operator\)](#)
 [* \(dereference\)](#)
 [, \(comma operator\)](#)
 [-> \(arrow operator\)](#)
 [<< \(output operator\)](#)
 [formatting](#)
 [must be nonmember](#)
 [Sales item.](#)
 [= \(assignment\)](#) 2nd 3rd
 [and copy constructor](#)
 [check for self-assignment](#)
 [Message](#)
 [reference return](#) 2nd
 [rule of three](#)
 [use counting](#) 2nd
 [valuelike classes](#)
 >> (input operator)
 [error handling](#)
 [must be nonmember](#)
 [_ _ \(subscript\)](#)
 [reference return](#)
 [addition \(+\).](#) [Sales item.](#)
 [ambiguous](#)
 [arithmetic operators](#)
 [as virtual function](#)
 [binary operator](#)
 [candidate functions](#)
 [compound assignment \(e.g., +=\)](#)
 [Sales item.](#)

[consistency between relational and equality operators](#)

[definition 2nd](#)

[design](#)

[equality operators 2nd](#)

[explicit call to](#)

[explicit call to postfix operators](#)

[function matching](#)

[member and `this` pointer](#)

[member vs. nonmember function 2nd](#)

[postfix increment \(`++`\) and decrement \(`--`\) operators](#)

[precedence and associativity](#)

[prefix increment \(`++`\) and decrement \(`--`\) operators](#)

[relational operators 2nd](#)

[require class-type parameter](#)

[unary operator](#)

[`||` \(logical OR\)](#)

overloading [See [overloaded function](#)]

operator [See [overloaded operator](#)]

Team LiB

◀ PREVIOUS NEXT ▶

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[pair](#) 2nd

 as return type from [map::insert](#)

 as return type from [set::insert](#).

 default constructor

 definition

 initialization

[make_pair](#).

 operations

 public data members

[Panda](#)

 virtual inheritance

[parameter](#) 2nd 3rd 4th

 and [main](#)

 array and buffer overflow

 array type

 C-style string

[const](#)

[const](#) reference

 overloading

 ellipsis

 function pointer

 linkage directive

 initialization of

 iterator 2nd

 library container

 lifetime

 local copy

 matching

 ellipsis

 template specialization

 with class type conversion

 multi-dimensioned array

 non[const](#) reference

 nonreference type

 uses copy constructor

 of member function

 passing

 pointer members

 pointer to [const](#)

 overloading

 pointer to function

 linkage directive

 pointer to non[const](#).

 pointer type 2nd

 reference

 to array type

 to pointer

 rule of three

 exception for virtual destructors

 synthesized 2nd

template [See [template parameter](#)]

type checking

[and template argument
of reference to array](#)

[use counting](#) 2nd

[valuelike classes](#)

[vector type](#)

[parameter list](#) 2nd 3rd 4th

[member function definition](#)

[parentheses, override precedence](#)

[partial specialization](#) 2nd

[partial sort](#),

[partial sort copy](#),

[partial sum](#),

[partition](#)

[placement new](#) 2nd

[compared to construct](#)

[plus<T>](#),

[pointer](#) 2nd 3rd

[array](#)

[arrow \(->\)](#)

[as initializer of vector](#)

[as parameter](#)

[assignment](#)

char* [See [C-style string](#)]

[class member copy control](#)

[copy constructor](#)

[destructor](#)

[strategies](#)

[compared to iterator](#)

[compared to reference](#)

[const](#)

[const pointer to const](#)

[container constructor from](#)

[conversion from derived to multiple base](#)

[conversion fromderived to base](#)

[conversion to bool](#)

[conversion to void](#)

[dangling](#) 2nd

[synthesized copy control](#)

[declaration style](#)

[definition](#)

[delete](#)

[dynamic cast, example](#)

[function returning](#)

[implicit this](#) 2nd

[initialization](#)

[is polymorphic](#)

[multi-dimensioned array](#)

[new](#)

[null](#)

[off-the-end](#)

[pitfalls with generic programs](#)

[reference parameter](#)

[relational operator](#)

[return type and local variable](#)

[smart](#) 2nd 3rd

[handle class](#)

[overloaded \(++\) and \(*\).](#)
[overloaded -> \(arrowoperator\) and * \(dereference\)](#)

[subscript operator](#)

[to pointer](#)

[typedef](#)

[typeid operator](#)

[uninitialized](#)

[volatile](#)

[pointer arithmetic 2nd](#)

[pointer to const](#)

[argument](#)

[conversion from non const.](#)

[parameter](#)

[overloading](#)

[pointer to function](#)

[definition](#)

[exception specifications](#)

[function returning](#)

[initialization](#)

[linkage directive](#)

[overloaded functions](#)

[parameter](#)

[return type](#)

[typedef](#)

[pointer to member 2nd](#)

[and typedef](#)

[arrow \(->*\) 2nd](#)

[definition](#)

[dot \(.* \) 2nd](#)

[function pointer](#)

[function table](#)

[pointer to non const](#)

[argument](#)

[parameter](#)

[polymorphism 2nd](#)

[compile time polymorphism via templates](#)

[run time polymorphism in C++](#)

[pop](#)

[priority queue.](#)

[queue](#)

[stack](#)

[pop back, sequential container](#)

[pop front, sequential container](#)

[portable](#)

[postfix decrement \(- -\)](#)

[overloaded operator](#)

[yields rvalue](#)

[postfix increment \(++\)](#)

[and dereference](#)

[overloaded operator](#)

[precedence 2nd 3rd 4th 5th](#)

[of assignment](#)

[of conditional](#)

[of dot and dereference](#)

[of increment and dereference](#)

[of IO operator](#)

[of pointer to member and call operator](#)

[overloaded operator](#)

[pointer parameter declaration](#)

[precedence table](#)
[predicate 2nd](#)
[prefix decrement \(--\)](#)
 overloaded operator
 yields lvalue
[prefix increment \(++\)](#)
 and dereference
 overloaded operator
 yields lvalue
[preprocessor 2nd](#)
 directive 2nd
 macro 2nd
 variable
[prev permutation.](#)
[preventing copies of class objects](#)
[print total.](#)
 explained
[printable character](#)
[printValues program 2nd 3rd](#)
[priority queue. 2nd](#)
 constructors
 relational operator
[private](#)
 class
 copy constructor
 inheritance
 member 2nd
[private access label 2nd](#)
 inheritance
[private inheritance](#)
[program](#)
 book finding
 [using equal range.](#)
 [using find](#)
 [using upper bound.](#)
 [bookstore](#)
 [bookstore exception classes](#)
 [CachedObj](#)
 [duplicate words](#)
 [revisited](#)
 [factorial](#)
 [find last word](#)
 [find val.](#)
 [acd](#)
 [GT6](#)
 [Handle class](#)
 [int instantiation](#)
 [operations](#)
 [Sales item instantiation](#)
 [isShorter 2nd](#)
 [make plural.](#)
 [message handling classes](#)
 [open file.](#)
 [printValues 2nd 3rd](#)
 [ptr swap.](#)
 [Query](#)
 [design](#)

[interface](#)
[operations](#)
[Queue](#)
[copy elems member](#)
[destroy member](#)
[pop member](#)
[push member](#)
[random IO example](#)
[restricted word count](#)
[rqcd](#)
[Sales item handle class](#)
[Screen class template](#)
[swap 2nd](#)
[TextQuery](#)
[class definition](#)
[design](#)
[interface](#)
[vector capacity](#)
[vector, capacity](#)
[vowel counting](#)
[word count](#)
[word transformation](#)
[ZooAnimal class hierarchy](#)
[programmer-defined header](#)
programming
[generic 2nd](#)
[object-oriented 2nd 3rd](#)
[promotion, integral 2nd](#)
[protected access label 2nd](#)
[protected keyword](#)
[protected, inheritance 2nd](#)
[prototype, function 2nd](#)
[ptr swap program](#)
[ptrdiff_t, 2nd](#)
public
[inheritance 2nd](#)
[member 2nd](#)
[public access label 2nd](#)
[inheritance](#)
[pure virtual function 2nd](#)
[example](#)
push
[priority queue.](#)
[queue](#)
[stack](#)
[push back, 2nd](#)
[back inserter,](#)
[sequential container](#)
[vector](#)
push_front
[front inserter,](#)
[sequential container](#)
[put Msg in Folder.](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

Query

[& \(bitwise AND\)](#)

[definition](#)

[<< \(output operator\)](#)

[definition](#)

[design](#)

[interface](#)

[operations](#)

[| \(bitwise OR\)](#)

[definition](#)

[~ \(bitwise NOT\)](#)

[definition](#)

Query base

[definition](#)

[member functions](#)

queue 2nd

[Queue](#)

[<< \(output operator\)](#)

[assignment](#)

[queue](#)

[constructors](#)

[Queue](#)

[copy elems member 2nd](#)

[definition](#)

[design](#)

[destroy member](#)

[final class definition](#)

[interface](#)

[member template declarations](#)

[operations](#)

[pop member](#)

[push member](#)

[push specialized](#)

[queue](#)

[relational operator](#)

[Queue](#)

[template version, char*](#)

QueueItem

[as nested class](#)

[constructor](#)

[definition](#)

[CachedObj](#)

[allocation explained](#)

[friendship](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[Raccoon](#) as virtual base

[RAII](#) [See [resource allocation is initialization](#)]

[raise](#)

raise exception [See [throw](#)]

[random file IO](#)

[random-access iterator](#) 2nd

[deque](#)

[string](#)

[vector](#)

[random shuffle.](#)

[range](#)

[iterator](#) 2nd 3rd

[left-inclusive](#)

[range error.](#)

[rbegin](#), [container](#) 2nd

[rdstate](#)

[recursive function](#) 2nd

[refactoring](#) 2nd

[reference](#)

[reference](#) 2nd

[and pointer](#)

[const](#) [reference](#)

[initialization](#)

[conversion from derived to multiple base](#)

[conversion from derived to base](#)

[dynamic cast operator](#), [example](#)

[is polymorphic](#)

[nonconst](#) [reference](#)

[parameter](#)

[pointer parameter](#)

[return type and class object](#)

[return type and local variable](#)

[return type, is lvalue](#)

[return value](#)

[to array parameter](#)

[reference count](#) [See [use count](#)]

[reference data member](#), [initialization](#)

[reference return](#)

[reference to const](#) [See [const reference](#)]

[reinterpret cast](#), 2nd

[relational operator](#)

[associative container](#)

[container](#)

[container adaptor](#)

[function object](#)

[overloaded operator](#) 2nd

[consistent with equality](#)

[pointer](#)

[string](#)

[remove](#)

R

list
remove copy.
remove copy if.
remove if.
 list
remove Msg from Folder.
rend, container 2nd
replace 2nd
 string
replace copy, 2nd
replace copy if.
replace if.
reserve
 string
 vector
reserved identifier
resize, sequential container
Resource
resource allocation is initialization
 auto_ptr.
restricted word count program
result 2nd
rethrow 2nd
return statement
 from main
 implicit
 local variable 2nd
return type 2nd 3rd 4th
 const reference
 function
 function pointer
 linkage directive
 member function definition
 no implicit return type
 nonreference
 of virtual function
 pointer to function
 reference
 reference yields lvalue
 uses copy constructor
 void
return value
 conversion
 copied
return, container
reverse
 list
reverse iterator 2nd 3rd
 ++ (increment)
 -- (decrement)
 base
 compared to iterator 2nd
 example
 requires -- (decrement)
reverse copy.
reverse iterator.
 container

[rfind](#) [string](#)
[rwd](#) [program](#)
[right manipulator](#)
[right-shift \(>>\) 2nd](#) [3rd](#) [4th](#)
[scope \(::\) 2nd](#) [3rd](#)
 [class member](#) [2nd](#)
 [container defined type](#)
 [member function definition](#)
 [to override name lookup](#)
[shift 2nd](#)
[sizeof](#)
[subscript \(\[\]\)](#)
 [and multi-dimensioned array](#)
 [and pointer](#)
 [array](#)
 [bitset](#)
 [dequeue](#)
 [map](#)
 [string](#)
 [valid subscript range](#)
 [vector 2nd](#)
 [yields lvalue](#)
[subtraction \(-\)](#)
 [iterator 2nd](#)
 [pointer](#)
[unary 2nd](#)
[unary minus \(-\)](#)
[unary plus \(+\)](#)
[rotate](#)
[rotate copy](#)
[rule of three 2nd](#)
 [exception for virtual destructors](#)
[run time](#)
 [error](#)
[run-time type identification 2nd](#)
 [classes with virtual functions](#)
 [compared to virtual functions](#)
 [dynamic cast](#)
 [example](#)
 [throws bad cast](#)
 [to pointer](#)
 [to reference](#)
[type-sensitive equality](#)
[typeid](#)
 [and virtual functions](#)
 [example](#)
 [returns type info](#)
[runtime error 2nd](#)
 [constructor from string](#)
[rvalue 2nd](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

[safety_exception](#)

[Sales_item](#)

[addition \(+\)](#) 2nd

[throws exception](#) 2nd

[avg_price](#) definition

[class](#) definition 2nd

[compare](#) function

[compound assignment \(e.g., +=\)](#)

[conversion](#)

[default constructor](#)

[equality operators \(==\), \(!=\)](#)

[explicit constructor](#)

[handle class](#)

[clone](#) function

[constructor](#) 2nd

[definition](#)

[design](#)

[multiset](#) of

[using generic Handle](#)

[header](#) 2nd 3rd

[input \(>>\)](#)

[istream](#) constructor

[no relational operators](#)

[operations](#)

[output \(<<\)](#)

[same isbn](#) 2nd

[string](#) constructor

[scientific manipulator](#)

[scope](#) 2nd

[block](#)

[class](#) 2nd 3rd

[compared to object lifetime](#)

[const](#) object 2nd

[for statement](#)

[friend declaration](#)

[function](#)

[function template specialization](#)

[global](#) 2nd

[local](#) 2nd

[multiple inheritance](#)

[namespace](#)

[statement](#)

[template parameter](#)

[using](#) declaration

[using](#) directive

[example](#)

[name collisions](#)

[scope \(::\)](#)

[base class members](#)

S

[namespace member](#)

[scope operator \(::\) 2nd 3rd 4th](#)

[class member 2nd](#)

[container defined type](#)

[member function definition](#)

[namespace member](#)

[to override class-specific memory allocation](#)

[to override name lookup](#)

[Screen](#)

[CachedObj](#)

[class template](#)

[concatenating operations](#)

[display](#)

[do display](#)

[friends](#)

[get definition](#)

[get members](#)

[get cursor definition](#)

[Menu function table](#)

[move members](#)

[set members](#)

[simplified](#)

[size type](#)

[ScreenPtr](#)

[arrow operator \(->\)](#)

[dereference \(*\)](#)

[use counted](#)

[ScrPtr](#)

[search](#)

[search n](#)

[seek and tell members](#)

self-assignment

[auto ptr](#)

[check](#)

[use counting](#)

[semantics, value 2nd](#)

[semicolon \(;\)](#)

[semicolon \(;\), class definition](#)

[sentinel 2nd](#)

[separate compilation 2nd](#)

[inclusion model for templates](#)

[of templates](#)

[separate compilation model for templates 2nd](#)

[sequence \(\xnnn\), hexadecimal escape](#)

[sequence, escape](#)

[sequential container 2nd](#)

[assign](#)

[assignment \(=\)](#)

[back](#)

[clear](#)

[const iterator](#)

[const reverse iterator](#)

constructor from element count

[uses copy constructor](#)

[uses element default constructor](#)

[constructors](#)

[dequeue](#)

[element type constraints 2nd](#)

[empty](#)

[erase](#)

[front](#)

[insert](#)

[iterator](#)

[list](#)

[operations](#)

[performance characteristics](#)

[pop back](#)

[pop front](#)

[priority queue](#)

[push back](#)

[push front](#)

[queue](#)

[rbegin](#)

[rend](#)

[resize](#)

[returning a](#)

[reverse iterator 2nd](#)

[size](#)

[size type](#)

[stack](#)

[supports relational operators](#)

[swap](#)

[types defined by](#)

[value type](#)

[vector](#)

[set 2nd](#)

[as element type](#)

[assignment \(=\)](#)

[begin](#)

[bidirectional iterator](#)

[clear](#)

[constructors](#)

[count](#)

[element type constraints](#)

[empty](#)

[end](#)

[equal range](#)

[erase 2nd](#)

[find](#)

[insert](#)

[iterator](#)

[key type constraints](#)

[lower bound](#)

[operations](#)

[overriding the default comparison](#)

[rbegin](#)

[rend](#)

[return alternatives](#)

[return type from insert](#)

[reverse iterator](#)

[size](#)

[supports relational operators](#)

[swap](#)

S

[upper bound](#)
[value type](#)
[set difference](#)
[set intersection 2nd](#)
[set symmetric difference](#)
[set union](#)
[setfill manipulator](#)
[setprecision manipulator](#)
[setstate 2nd](#)

[setw manipulator](#)

[shift operator 2nd](#)

[short](#)

[short-circuit evaluation](#)

[overloaded operator](#)

[shorterString](#)

[showbase manipulator](#)

[showpoint manipulator](#)

[signed 2nd](#)

[conversion to unsigned 2nd](#)

[size](#)

[associative container](#)

[priority queue](#)

[queue](#)

[sequential container](#)

[stack](#)

[string](#)

[vector](#)

[size_t 2nd 3rd](#)

[and array](#)

[size_type 2nd](#)

[container](#)

[string](#)

[vector](#)

[sizeof operator](#)

[skipws manipulator](#)

[sliced 2nd](#)

[SmallInt 2nd](#)

[conversion operator](#)

[smart pointer 2nd 3rd](#)

[handle class](#)

[overloaded \(++\) and \(*\)](#)

[overloaded -> \(arrow operator\) and * \(dereference\)](#)

[sort 2nd](#)

[source file 2nd](#)

[naming convention](#)

[specialization](#)

[class template](#)

[definition](#)

[member defined outside class body](#)

[partial](#)

[partial specialization](#)

[class template member](#)

[declaration](#)

[function template](#)

[compared to overloaded function](#)

[declaration 2nd](#)

[example](#)

[scope](#)
[template, namespaces](#)

[specifier, type](#) 2nd

[splice, list](#)

[sstream](#)

[header](#) 2nd

[str](#)

[stable partition](#)

[stable sort](#) 2nd

[stack](#) 2nd

[constructors](#)

[relational operator](#)

[stack unwinding](#) 2nd

[standard error](#) 2nd

[standard input](#) 2nd

[standard library](#) 2nd

[standard output](#) 2nd

[state, condition](#)

[statement](#) 2nd

[break](#) 2nd

[compound](#) 2nd

[continue](#) 2nd

[declaration](#) 2nd

[do while](#)

[expression](#) 2nd

[for](#) 2nd

[for statement](#)_{for}

[goto](#) 2nd

[if](#) 2nd 3rd 4th

[labeled](#) 2nd

[null](#) 2nd

[return](#)

[return, local variable](#) 2nd

[switch](#) 2nd

[while](#) 2nd 3rd 4th

[statement block](#) [See [block](#)]

[statement label](#)

[statement scope](#)

[statement](#)_{for} [statement, for](#)

[static](#)

[static \(file static\)](#)

[static class member](#) 2nd

[as default argument](#)

[class template](#)

[accessed through an instantiation](#)

[definition](#)

[const](#) [data member, initialization](#)

[const](#) [member function](#)

[data member](#)

[as constant expression](#)

[inheritance](#)

[member function](#)

[this pointer](#)

[static object, local](#) 2nd

[static type](#) 2nd

[determines name lookup](#)

[multiple inheritance](#)

[static type checking](#) 2nd

[argument](#)
[function return value](#)

[static cast](#) 2nd

[std](#) 2nd

[stdexcept header](#) 2nd

[store, free](#) 2nd

[str](#)

[strcat](#)

[strcmp](#)

[strcpy](#)

[stream](#)

[flushing buffer](#)

[istream iterator](#)

[iterator](#) 2nd

[and class type](#)

[limitations](#)

[used with algorithms](#)

[not flushed if program crashes](#)

[ostream iterator](#)

[type as condition](#)

[stream iterator](#)

[strict weak ordering](#) 2nd

[string](#)

[addition](#)

[addition to string literal](#)

[and string literal](#) 2nd

[append](#)

[are case sensitive](#)

[as sequential container](#)

[assign](#)

[assignment \(=\)](#)

[c_str](#)

[c_str, example](#)

[capacity](#)

[compare](#)

[compared to C-style string](#)

[compound assignment](#)

[concatenation](#)

[constructor](#) 2nd

[default constructor](#)

[empty](#)

[equality \(==\)](#)

[equality operator](#)

[erase](#)

[find](#)

[find first not of](#)

[find first of](#)

[find last not of](#)

[find last of](#)

[getline](#)

[getline, example](#)

[header](#)

[input operation as condition](#)

[input operator](#)

[insert](#)

[output operator](#)

[random-access iterator](#)

[relational operator](#) 2nd

[replace](#)
[reserve](#)
[rfind](#)
[size](#)
[size_type](#)
[subscript operator](#)
[substr](#)

[string literal](#) 2nd 3rd
 [addition to string](#)
 [and C-style string](#)
 [and string library type](#) 2nd
 [concatenation](#)

[string, C-style](#) [See [C-style string](#)]

[stringstream](#) 2nd 3rd [See also [istream](#), [ostream](#)]

[str](#)
[strlen](#)
[strncat](#)
[strncpy](#)

[struct](#) [See also [class](#)]

[default access label](#)
 [default inheritance access label](#)

[struct](#), keyword 2nd 3rd
 [in variable definition](#)

[structure, data](#) 2nd

[Studio, Visual](#)

[subscript \(\[\] \)](#) 2nd 3rd 4th

[and multi-dimensioned array](#)
 [and pointer](#)
 [array](#)
 [bitset](#)
 [deque](#)
 [map](#)

[overloaded operator](#)
 [reference return](#)
 [string](#)
 [valid subscript range](#)
 [vector](#) 2nd
 [yields lvalue](#)

[subscript range](#)

[array](#)
 [string](#)
 [vector](#)

[substr, string](#)

[subtraction \(-\)](#)
 [iterator](#) 2nd
 [pointer](#)

[swap](#) 2nd
 [container](#)
 [swap program](#) 2nd

[swap ranges](#)
[switch statement](#) 2nd
 [and break](#)
 [case label](#)
 [compared to if](#)
 [default label](#)
 [execution flow](#)
 [expression](#)

[variable definition](#)

[synthesized assignment \(=\) 2nd](#)

[multiple inheritance](#)

[pointer members](#)

[synthesized copy constructor 2nd](#)

[multiple inheritance](#)

[pointer members](#)

[virtual base class](#)

[synthesized copy control, volatile](#)

[synthesized default constructor 2nd 3rd 4th](#)

[inheritance](#)

[synthesized destructor 2nd](#)

[multiple inheritance](#)

[pointer members](#)

Team LiB

◀ PREVIOUS NEXT ▶

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[table of library name and header](#)

[template](#) [See also [class template](#), [function template](#), [instantiation](#)]

[class 2nd](#)

[class member](#) [See [member template](#)]

[link time errors](#)

[overview](#)

[template argument 2nd](#)

[and function argument type checking](#)

[class template](#)

[conversion](#)

[deduction](#)

[from function pointer](#)

[deduction for class template member function](#)

[deduction for function template](#)

[explicit and class template](#)

[explicit and function template](#)

[and function pointer](#)

[specifying](#)

[pointer](#)

[template argument deduction](#)

[template class](#) [See [class template](#)]

[template function](#) [See [function template](#)]

[template keyword](#)

[template parameter 2nd 3rd](#)

[and member templates](#)

[name](#)

[restrictions on use](#)

[nontype parameter 2nd 3rd 4th](#)

[class template](#)

[must be constant expression](#)

[scope](#)

[type parameter 2nd 3rd 4th](#)

[uses of inside class definition](#)

[template parameter list 2nd](#)

[template specialization 2nd](#)

[class member declaration](#)

[compared to overloaded function](#)

[definition](#)

[example](#)

[function declaration 2nd](#)

[member defined outside class body](#)

[member of class template](#)

[parameter matching](#)

[partial specialization 2nd](#)

[scope](#)

[template<>](#) [See [template specialization](#)]

[temporary object](#)

[terminate 2nd 3rd 4th 5th](#)

[TextQuery](#)

[class definition](#)

[main program using](#)

[program design](#)

[program interface](#)

[revisited](#)

[this pointer](#)

[implicit 2nd](#)

[implicit parameter 2nd](#)

[in overloaded operator](#)

[overloaded operator](#)

[static member functions](#)

[three, rule of 2nd](#)

[tHRow 2nd 3rd 4th 5th](#)

[example 2nd](#)

[execution flow 2nd](#)

[pointer to local object](#)

[rethrow](#)

[tolower](#)

[top](#)

[priority queue](#)

[stack](#)

[toupper](#)

[TRansform](#)

[transformation program, word](#)

[translation unit \[See \[source file\]\(#\)\]](#)

[trunc \(file mode\)](#)

[TRY block 2nd 3rd 4th](#)

[TRY keyword](#)

[type](#)

[abstract data 2nd](#)

[arithmetic 2nd](#)

[built-in 2nd 3rd](#)

[class 2nd 3rd](#)

[compound 2nd 3rd](#)

[dynamic 2nd](#)

[function return](#)

[incomplete 2nd](#)

[integral 2nd](#)

[library](#)

[nested \[See \[nested class\]\(#\)\]](#)

[return 2nd 3rd 4th](#)

[static 2nd](#)

[determines name lookup](#)

[name lookup and multiple inheritance](#)

[type checking](#)

[argument](#)

[with class type conversion](#)

[ellipsis parameter](#)

[name lookup](#)

[reference to array argument](#)

[type identification, run-time 2nd](#)

[type specifier 2nd](#)

[type template parameter 2nd 3rd \[See also \[template parameter\]\(#\)\]](#)

[type info](#)

[header](#)

[name member](#)

[no copy or assign](#)

[operations](#)

[returned from typeid](#)

[typedef 2nd](#)

[typedef](#)

[and pointer](#)
[and pointer to member](#)
[pointer to function](#)
[typeid operator 2nd](#)
[and virtual functions](#)
[example](#)
[returns type info](#)

`typename`, keyword

[compared to class](#)

[in template parameter](#)

[inside template definition](#)

Team LiB

[!\[\]\(8484929b446e90b0da0e90c3d0ef7f3d_img.jpg\) PREVIOUS](#) [!\[\]\(d1746b416113b3cb1f3b856fd4297ca2_img.jpg\) NEXT !\[\]\(c3c0999d457ebb3f22f88d696104ad2a_img.jpg\)](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

[U_Ptr](#)

[unary function object](#)

[unary minus \(-\)](#)

[unary operator 2nd](#)

[unary plus \(+\)](#)

[uncaught exception](#)

[undefined behavior 2nd](#)

[dangling pointer](#)

[synthesized copy control](#)

[invalidated iterator](#)

[uninitialized class data member](#)

[uninitialized pointer](#)

[uninitialized variable](#)

[underflow error](#)

[unexpected 2nd](#)

[uninitialized 2nd 3rd 4th](#)

[uninitialized pointer](#)

[uninitialized copy 2nd](#)

[uninitialized fill](#)

[union 2nd](#)

[anonymous 2nd](#)

[as nested type](#)

[example](#)

[limitations on](#)

[union keyword](#)

[unique 2nd](#)

[list](#)

[unique copy 2nd](#)

[unitbuf, manipulator flushes the buffer](#)

[unnamed namespace 2nd](#)

[local to file](#)

[replace file static](#)

[unsigned 2nd](#)

[conversion to signed 2nd](#)

[literal \(numu or numu\)](#)

[unsigned char](#)

[unwinding, stack 2nd](#)

[upper bound](#)

[associative container](#)

[book finding program](#)

[example](#)

[uppercase manipulator](#)

[use count 2nd](#)

[design overview](#)

[generic class](#)

[held in companion class](#)

[pointer to](#)

[self-assignment check](#)

[user 2nd](#)

[using declaration 2nd 3rd 4th](#)

[access control](#)

[class member access](#)

[in header](#)

[overloaded function](#)

[overloaded inherited functions](#)

[scope](#)

[using directive 2nd](#)

[overloaded function](#)

[pitfalls](#)

[scope](#)

[example](#)

[name collisions](#)

[utility header](#)

Team LiB

◀ PREVIOUS NEXT ▶

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[value initialization](#) 2nd

[and dynamically allocated array](#)

[and resize](#)

[deque](#)[dequeue](#)

[list](#)[list](#)

[map subscript operator](#)

[of dynamically allocated object](#)

[sequential container](#)

[vector](#)

[vector](#)[vector](#)

[value semantics](#) 2nd

[value type](#)

[map](#), [multimap](#)

[sequential container](#)

[set](#), [multiset](#)

[varargs](#)

[variable](#) 2nd 3rd

[define before use](#)

[defined after case label](#)

[definition](#)

[definitions and goto](#)

[vector](#) 2nd

[Vector](#)

[vector](#)

[argument](#)

[as element type](#)

[assig_n](#)

[assignment \(=\)](#)

[at](#)

[back](#)

[begin](#) 2nd

[capacity](#)

[Vector](#)

[capacity](#)

[vector](#)

[clear](#)

[const iterator](#) 2nd

[const reference](#)

[const reverse iterator](#)

[constructor from element count, uses copy constructor](#)

[constructor taking iterators](#)

[constructors](#) 2nd

[difference type](#)

[element type constraints](#) 2nd

[empty](#) 2nd

[end](#) 2nd

[erase](#) 2nd

[invalidates iterator](#)

V

front
header
initialization from pointer
insert

invalidates iterator

iterator 2nd

iterator supports arithmetic

memory allocation strategy

Vector

memory allocation strategy

vector

memory management strategy

parameter

performance characteristics

pop back

push back 2nd

Vector

push back

vector

push_back

invalidates iterator

random-access iterator

rbegin 2nd

Vector

reallocate

vector

reference

relational operators

rend 2nd

reserve

resize

reverse iterator 2nd

size 2nd

Vector

size

vector

size_type 2nd

subscript ([])

subscript operator

supports relational operators

swap

type

types defined by

Vector

using explicit destructor call

using operator new and delete

using placement new

vector

value_type

vector capacity program

viable function 2nd

with class type conversion

virtual base class 2nd

ambiguities

conversion

defining base as

derived class constructor

name lookup

[order of construction](#)
[stream types](#)
[virtual function 2nd 3rd](#)
 [assignment operator](#)
 [calls resolved at run time](#)
 [compared to run-time type identification](#)
 [default argument](#)
 [derived classes](#)
 [destructor](#)
 [multiple inheritance](#)
 [exception specifications](#)
 [in constructors](#)
 [in destructor](#)
 [introduction](#)
 [multiple inheritance](#)
 [no virtual constructor](#)
 [overloaded](#)
 [overloaded operator](#)
 [overriding run-time binding](#)
 [pure 2nd](#)
 [example](#)
 [return type](#)
 [run-time type identification](#)
 [scope](#)
 [static](#)
 [to copy unknown type](#)
 [type-sensitive equality](#)
[virtual inheritance 2nd](#)
[virtual keyword](#)
[Visual Studio](#)
[void 2nd](#)
 [return type](#)
[void* 2nd](#)
 [const void* 2nd](#)
[volatile 2nd](#)
 [pointer](#)
 [synthesized copy control](#)
[vowel counting program](#)

Index

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[Z\]](#)

[wcerr](#)
[wchar_t](#)
[literal](#)
[wchar_t streams](#)
[wcin](#)
[wcout](#)
[weak ordering, strict 2nd](#)
[wfstream](#)
[what](#) [See [exception](#)]
[while statement](#) 2nd 3rd 4th
[condition in](#)
[whitespace](#)
[wide character streams](#)
[wifstream](#)
[window, console](#)
[Window Mar](#)
[wiostream](#)
[wistream](#)
[wistringstream](#)
[wofstream](#)
[word 2nd](#)
[word count program](#)
[restricted](#)
[word per line processing](#)
[istringstream](#)
[istringstream istringstream](#)
[word transformation program](#)
[WordQuery](#)
[definition](#)
[wostream](#)
[wostringstream](#)
[wrap around](#)
[wstringstream](#)

Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[Z](#)]

[ZooAnimal class hierarchy](#)([ZooAnimal](#), 使用虚继承)

[ZooAnimal, using virtual inheritance](#)([ZooAnimal](#) 类层次)