C++11

The C++ Programming Language

Fourth Edition

Bjarne Stroustrup

The Creator of C++

The C++ Programming Language

Fourth Edition

Bjarne Stroustrup

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Totonto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Stroustrup, Bjarne.
The C++ programming language / Bjarne Stroustrup.—Fourth edition.
pages cm
Includes bibliographical references and index.
ISBN 978-0-321-56384-2 (pbk. : alk. paper)—ISBN 0-321-56384-0 (pbk. : alk. paper)
1. C++ (Computer programming language) I. Title.

QA76.73.C153 S77 2013
005.13’3—dc23 2013002159

This book was typeset in Times and Helvetica by the author.

ISBN-13: 978-0-321-56384-2
ISBN-10: 0-321-56384-0
Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.
Second printing, June 2013

Contents

Preface

All problems in computer science
can be solved by another level of indirection,
except for the problem of too many layers of indirection.
– David J. Wheeler

C++ feels like a new language. That is, I can express my ideas more clearly, more simply, and more directly in C++11 than I could in C++98. Furthermore, the resulting programs are better checked by the compiler and run faster.

In this book, I aim for *completeness*. I describe every language feature and standard-library component that a professional programmer is likely to need. For each, I provide:

- *Rationale*: What kinds of problems is it designed to help solve? What principles underlie the design? What are the fundamental limitations?
- *Specification*: What is its definition? The level of detail is chosen for the expert programmer; the aspiring language lawyer can follow the many references to the ISO standard.
- *Examples*: How can it be used well by itself and in combination with other features? What are the key techniques and idioms? What are the implications for maintainability and performance?

The use of C++ has changed dramatically over the years and so has the language itself. From the point of view of a programmer, most of the changes have been improvements. The current ISO standard C++ (ISO/IEC 14882-2011, usually called C++11) is simply a far better tool for writing quality software than were previous versions. How is it a better tool? What kinds of programming styles and techniques does modern C++ support? What language and standard-library features support those techniques? What are the basic building blocks of elegant, correct, maintainable, and efficient C++ code? Those are the key questions answered by this book. Many answers are not the same as you would find with 1985, 1995, or 2005 vintage C++: progress happens.

C++ is a general-purpose programming language emphasizing the design and use of type-rich, lightweight abstractions. It is particularly suited for resource-constrained applications, such as those found in software infrastructures. C++ rewards the programmer who takes the time to master techniques for writing quality code. C++ is a language for someone who takes the task of programming seriously. Our civilization depends critically on software; it had better be quality software.

There are billions of lines of C++ deployed. This puts a premium on stability, so 1985 and 1995 C++ code still works and will continue to work for decades. However, for all applications, you can do better with modern C++; if you stick to older styles, you will be writing lower-quality and worse-performing code. The emphasis on stability also implies that standards-conforming code you write today will still work a couple of decades from now. All code in this book conforms to the 2011 ISO C++ standard.

This book is aimed at three audiences:

- C++ programmers who want to know what the latest ISO C++ standard has to offer,
- C programmers who wonder what C++ provides beyond C, and
- People with a background in application languages, such as Java, C#, Python, and Ruby, looking for something ‘‘closer to the machine’’ – something more flexible, something offering better compile-time checking, or something offering better performance.

Naturally, these three groups are not disjoint – a professional software developer masters more than just one programming language.

This book assumes that its readers are programmers. If you ask, ‘‘What’s a for-loop?’’ or ‘‘What’s a compiler?’’ then this book is not (yet) for you; instead, I recommend my *Programming: Principles and Practice Using C++* to get started with programming and C++. Furthermore, I assume that readers have some maturity as software developers. If you ask ‘‘Why bother testing?’’ or say, ‘‘All languages are basically the same; just show me the syntax’’ or are confident that there is a single language that is ideal for every task, this is not the book for you.

What features does C++11 offer over and above C++98? A machine model suitable for modern computers with lots of concurrency. Language and standard-library facilities for doing systems-level concurrent programming (e.g., using multicores). Regular expression handling, resource management pointers, random numbers, improved containers (including, hash tables), and more. General and uniform initialization, a simpler **for**-statement, move semantics, basic Unicode support, lambdas, general constant expressions, control over class defaults, variadic templates, user-defined literals, and more. Please remember that those libraries and language features exist to support programming techniques for developing quality software. They are meant to be used in combination – as bricks in a building set – rather than to be used individually in relative isolation to solve a specific problem. A computer is a universal machine, and C++ serves it in that capacity. In particular, C++’s design aims to be sufficiently flexible and general to cope with future problems undreamed of by its designers.

Acknowledgments

In addition to the people mentioned in the acknowledgment sections of the previous editions, I would like to thank Pete Becker, Hans-J. Boehm, Marshall Clow, Jonathan Coe, Lawrence Crowl, Walter Daugherty, J. Daniel Garcia, Robert Harle, Greg Hickman, Howard Hinnant, Brian Kernighan, Daniel Krügler, Nevin Liber, Michel Michaud, Gary Powell, Jan Christiaan van Winkel, and Leor Zolman. Without their help this book would have been much poorer.

Thanks to Howard Hinnant for answering many questions about the standard library.

Andrew Sutton is the author of the Origin library, which was the testbed for much of the discussion of emulating concepts in the template chapters, and of the matrix library that is the topic of Chapter 29. The Origin library is open source and can be found by searching the Web for ‘‘Origin’’ and ‘‘Andrew Sutton.’’

Thanks to my graduate design class for finding more problems with the ‘‘tour chapters’’ than anyone else.

Had I been able to follow every piece of advice of my reviewers, the book would undoubtedly have been much improved, but it would also have been hundreds of pages longer. Every expert reviewer suggested adding technical details, advanced examples, and many useful development conventions; every novice reviewer (or educator) suggested adding examples; and most reviewers observed (correctly) that the book may be too long.

Thanks to Princeton University’s Computer Science Department, and especially Prof. Brian Kernighan, for hosting me for part of the sabbatical that gave me time to write this book.

Thanks to Cambridge University’s Computer Lab, and especially Prof. Andy Hopper, for hosting me for part of the sabbatical that gave me time to write this book.

Thanks to my editor, Peter Gordon, and his production team at Addison-Wesley for their help and patience.

College Station, Texas *Bjarne Stroustrup*

This page intentionally left blank

Preface to the Third Edition

Programming is understanding.
– Kristen Nygaard

I find using C++ more enjoyable than ever. C++'s support for design and programming has improved dramatically over the years, and lots of new helpful techniques have been developed for its use. However, C++ is not *just* fun. Ordinary practical programmers have achieved significant improvements in productivity, maintainability, flexibility, and quality in projects of just about any kind and scale. By now, C++ has fulfilled most of the hopes I originally had for it, and also succeeded at tasks I hadn't even dreamt of.

This book introduces standard C++† and the key programming and design techniques supported by C++. Standard C++ is a far more powerful and polished language than the version of C++ introduced by the first edition of this book. New language features such as namespaces, exceptions, templates, and run-time type identification allow many techniques to be applied more directly than was possible before, and the standard library allows the programmer to start from a much higher level than the bare language.

About a third of the information in the second edition of this book came from the first. This third edition is the result of a rewrite of even larger magnitude. It offers something to even the most experienced C++ programmer; at the same time, this book is easier for the novice to approach than its predecessors were. The explosion of C++ use and the massive amount of experience accumulated as a result makes this possible.

The definition of an extensive standard library makes a difference to the way C++ concepts can be presented. As before, this book presents C++ independently of any particular implementation, and as before, the tutorial chapters present language constructs and concepts in a ''bottom up'' order so that a construct is used only after it has been defined. However, it is much easier to use a well-designed library than it is to understand the details of its implementation. Therefore, the standard library can be used to provide realistic and interesting examples well before a reader can be assumed to understand its inner workings. The standard library itself is also a fertile source of programming examples and design techniques.

This book presents every major C++ language feature and the standard library. It is organized around language and library facilities. However, features are presented in the context of their use.

† ISO/IEC 14882, Standard for the C++ Programming Language.

That is, the focus is on the language as the tool for design and programming rather than on the language in itself. This book demonstrates key techniques that make C++ effective and teaches the fundamental concepts necessary for mastery. Except where illustrating technicalities, examples are taken from the domain of systems software. A companion, *The Annotated C++ Language Standard*, presents the complete language definition together with annotations to make it more comprehensible.

The primary aim of this book is to help the reader understand how the facilities offered by C++ support key programming techniques. The aim is to take the reader far beyond the point where he or she gets code running primarily by copying examples and emulating programming styles from other languages. Only a good understanding of the ideas behind the language facilities leads to mastery. Supplemented by implementation documentation, the information provided is sufficient for completing significant real-world projects. The hope is that this book will help the reader gain new insights and become a better programmer and designer.

Acknowledgments
In addition to the people mentioned in the acknowledgement sections of the first and second editions, I would like to thank Matt Austern, Hans Boehm, Don Caldwell, Lawrence Crowl, Alan Feuer, Andrew Forrest, David Gay, Tim Griffin, Peter Juhl, Brian Kernighan, Andrew Koenig, Mike Mowbray, Rob Murray, Lee Nackman, Joseph Newcomer, Alex Stepanov, David Vandevoorde, Peter Weinberger, and Chris Van Wyk for commenting on draft chapters of this third edition. Without their help and suggestions, this book would have been harder to understand, contained more errors, been slightly less complete, and probably been a little bit shorter.

I would also like to thank the volunteers on the C++ standards committees who did an immense amount of constructive work to make C++ what it is today. It is slightly unfair to single out individuals, but it would be even more unfair not to mention anyone, so I'd like to especially mention Mike Ball, Dag Brück, Sean Corfield, Ted Goldstein, Kim Knuttila, Andrew Koenig, Dmitry Lenkov, Nathan Myers, Martin O'Riordan, Tom Plum, Jonathan Shopiro, John Spicer, Jerry Schwarz, Alex Stepanov, and Mike Vilot, as people who each directly cooperated with me over some part of C++ and its standard library.

After the initial printing of this book, many dozens of people have mailed me corrections and suggestions for improvements. I have been able to accommodate many of their suggestions within the framework of the book so that later printings benefitted significantly. Translators of this book into many languages have also provided many clarifications. In response to requests from readers, I have added appendices D and E. Let me take this opportunity to thank a few of those who helped: Dave Abrahams, Matt Austern, Jan Bielawski, Janina Mincer Daszkiewicz, Andrew Koenig, Dietmar Kühl, Nicolai Josuttis, Nathan Myers, Paul E. Sevinç, Andy Tenne-Sens, Shoichi Uchida, Ping-Fai (Mike) Yang, and Dennis Yelle.

Murray Hill, New Jersey *Bjarne Stroustrup*

Preface to the Second Edition

The road goes ever on and on.
– Bilbo Baggins

As promised in the first edition of this book, C++ has been evolving to meet the needs of its users. This evolution has been guided by the experience of users of widely varying backgrounds working in a great range of application areas. The C++ user-community has grown a hundredfold during the six years since the first edition of this book; many lessons have been learned, and many techniques have been discovered and/or validated by experience. Some of these experiences are reflected here.

The primary aim of the language extensions made in the last six years has been to enhance C++ as a language for data abstraction and object-oriented programming in general and to enhance it as a tool for writing high-quality libraries of user-defined types in particular. A ‘‘high-quality library,’’ is a library that provides a concept to a user in the form of one or more classes that are convenient, safe, and efficient to use. In this context, *safe* means that a class provides a specific type-safe interface between the users of the library and its providers; *efficient* means that use of the class does not impose significant overheads in run-time or space on the user compared with hand-written C code.

This book presents the complete C++ language. Chapters 1 through 10 give a tutorial introduction; Chapters 11 through 13 provide a discussion of design and software development issues; and, finally, the complete C++ reference manual is included. Naturally, the features added and resolutions made since the original edition are integral parts of the presentation. They include refined overloading resolution, memory management facilities, and access control mechanisms, type-safe linkage, **const** and **static** member functions, abstract classes, multiple inheritance, templates, and exception handling.

C++ is a general-purpose programming language; its core application domain is systems programming in the broadest sense. In addition, C++ is successfully used in many application areas that are not covered by this label. Implementations of C++ exist from some of the most modest microcomputers to the largest supercomputers and for almost all operating systems. Consequently, this book describes the C++ language itself without trying to explain a particular implementation, programming environment, or library.

This book presents many examples of classes that, though useful, should be classified as ‘‘toys.’’ This style of exposition allows general principles and useful techniques to stand out more clearly than they would in a fully elaborated program, where they would be buried in details. Most of the useful classes presented here, such as linked lists, arrays, character strings, matrices, graphics classes, associative arrays, etc., are available in ‘‘bulletproof’’ and/or ‘‘goldplated’’ versions from a wide variety of commercial and non-commercial sources. Many of these ‘‘industrial strength’’ classes and libraries are actually direct and indirect descendants of the toy versions found here.

This edition provides a greater emphasis on tutorial aspects than did the first edition of this book. However, the presentation is still aimed squarely at experienced programmers and endeavors not to insult their intelligence or experience. The discussion of design issues has been greatly expanded to reflect the demand for information beyond the description of language features and their immediate use. Technical detail and precision have also been increased. The reference manual, in particular, represents many years of work in this direction. The intent has been to provide a book with a depth sufficient to make more than one reading rewarding to most programmers. In other words, this book presents the C++ language, its fundamental principles, and the key techniques needed to apply it. Enjoy!

Acknowledgments

In addition to the people mentioned in the acknowledgements section in the preface to the first edition, I would like to thank Al Aho, Steve Buroff, Jim Coplien, Ted Goldstein, Tony Hansen, Lorraine Juhl, Peter Juhl, Brian Kernighan, Andrew Koenig, Bill Leggett, Warren Montgomery, Mike Mowbray, Rob Murray, Jonathan Shopiro, Mike Vilot, and Peter Weinberger for commenting on draft chapters of this second edition. Many people influenced the development of C++ from 1985 to 1991. I can mention only a few: Andrew Koenig, Brian Kernighan, Doug McIlroy, and Jonathan Shopiro. Also thanks to the many participants of the ‘‘external reviews’’ of the reference manual drafts and to the people who suffered through the first year of X3J16.

Murray Hill, New Jersey *Bjarne Stroustrup*

Preface to the First Edition

Language shapes the way we think,
and determines what we can think about.
– B.L.Whorf

C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer. Except for minor details, C++ is a superset of the C programming language. In addition to the facilities provided by C, C++ provides flexible and efficient facilities for defining new types. A programmer can partition an application into manageable pieces by defining new types that closely match the concepts of the application. This technique for program construction is often called *data abstraction.* Objects of some user-defined types contain type information. Such objects can be used conveniently and safely in contexts in which their type cannot be determined at compile time. Programs using objects of such types are often called *object based*. When used well, these techniques result in shorter, easier to understand, and easier to maintain programs.

The key concept in C++ is *class*. A class is a user-defined type. Classes provide data hiding, guaranteed initialization of data, implicit type conversion for user-defined types, dynamic typing, user-controlled memory management, and mechanisms for overloading operators. C++ provides much better facilities for type checking and for expressing modularity than C does. It also contains improvements that are not directly related to classes, including symbolic constants, inline substitution of functions, default function arguments, overloaded function names, free store management operators, and a reference type. C++ retains C's ability to deal efficiently with the fundamental objects of the hardware (bits, bytes, words, addresses, etc.). This allows the user-defined types to be implemented with a pleasing degree of efficiency.

C++ and its standard libraries are designed for portability. The current implementation will run on most systems that support C. C libraries can be used from a C++ program, and most tools that support programming in C can be used with C++.

This book is primarily intended to help serious programmers learn the language and use it for nontrivial projects. It provides a complete description of C++, many complete examples, and many more program fragments.

Acknowledgments

C++ could never have matured without the constant use, suggestions, and constructive criticism of many friends and colleagues. In particular, Tom Cargill, Jim Coplien, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanthi, Doug McIlroy, Dennis Ritchie, Larry Rosler, Jerry Schwarz, and Jon Shopiro provided important ideas for development of the language. Dave Presotto wrote the current implementation of the stream I/O library.

In addition, hundreds of people contributed to the development of C++ and its compiler by sending me suggestions for improvements, descriptions of problems they had encountered, and compiler errors. I can mention only a few: Gary Bishop, Andrew Hume, Tom Karzes, Victor Milenkovic, Rob Murray, Leonie Rose, Brian Schmult, and Gary Walker.

Many people have also helped with the production of this book, in particular, Jon Bentley, Laura Eaves, Brian Kernighan, Ted Kowalski, Steve Mahaney, Jon Shopiro, and the participants in the C++ course held at Bell Labs, Columbus, Ohio, June 26-27, 1985.

Murray Hill, New Jersey *Bjarne Stroustrup*

Part I

Introduction

This introduction gives an overview of the major concepts and features of the C++ programming language and its standard library. It also provides an overview of this book and explains the approach taken to the description of the language facilities and their use. In addition, the introductory chapters present some background information about C++, the design of C++, and the use of C++.

Chapters

“... and you, Marcus, you have given me many things; now I shall give you this good advice. Be many people. Give up the game of being always Marcus Cocoza. You have worried too much about Marcus Cocoza, so that you have been really his slave and prisoner. You have not done anything without first considering how it would affect Marcus Cocoza’s happiness and prestige. You were always much afraid that Marcus might do a stupid thing, or be bored. What would it really have mattered? All over the world people are doing stupid things ... I should like you to be easy, your little heart to be light again. You must from now, be more than one, many people, as many as you can think of ...”

– Karen Blixen,
The Dreamers from *Seven Gothic Tales* (1934)

1

Notes to the Reader

Hurry Slowly
(festina lente).
– Octavius, Caesar Augustus

- The Structure of This Book
  Introduction; Basic Facilities; Abstraction Mechanisms; The Standard Library; Examples and References
- The Design of C++
  Programming Styles; Type Checking; C Compatibility; Language, Libraries, and Systems
- Learning C++
  Programming in C++; Suggestions for C++ Programmers; Suggestions for C Programmers; Suggestions for Java Programmers
- History
  Timeline; The Early Years; The 1998 Standard; The 2011 Standard; What is C++ Used for?
- Advice
- References

1.1 The Structure of This Book

A pure tutorial sorts its topics so that no concept is used before it has been introduced; it must be read linearly starting with page one. Conversely, a pure reference manual can be accessed starting at any point; it describes each topic succinctly with references (forward and backward) to related topics. A pure tutorial can in principle be read without prerequisites – it carefully describes all. A pure reference can be used only by someone familiar with all fundamental concepts and techniques. This book combines aspects of both. If you know most concepts and techniques, you can access it on a per-chapter or even on a per-section basis. If not, you can start at the beginning, but try not to get bogged down in details. Use the index and the cross-references.

Making parts of the book relatively self-contained implies some repetition, but repetition also serves as review for people reading the book linearly. The book is heavily cross-referenced both to itself and to the ISO C++ standard. Experienced programmers can read the (relatively) quick ‘‘tour’’ of C++ to gain the overview needed to use the book as a reference. This book consists of four parts:

Part I	*Introduction*: Chapter 1 (this chapter) is a guide to this book and provides a bit of C++ background. Chapters 2-5 give a quick introduction to the C++ language and its standard library.
Part II	*Basic Facilities*: Chapters 6-15 describe C++'s built-in types and the basic facilities for constructing programs out of them.
Part III	*Abstraction Mechanisms*: Chapters 16-29 describe C++'s abstraction mechanisms and their use for object-oriented and generic programming.
Part IV	Chapters 30-44 provide an overview of the standard library and a discussion of compatibility issues.

1.1.1 Introduction

This chapter, Chapter 1, provides an overview of this book, some hints about how to use it, and some background information about C++ and its use. You are encouraged to skim through it, read what appears interesting, and return to it after reading other parts of the book. Please do not feel obliged to read it all carefully before proceeding.

The following chapters provide an overview of the major concepts and features of the C++ programming language and its standard library:

Chapter 2	*A Tour of C++: The Basics* describes C++'s model of memory, computation, and error handling.
Chapter 3	*A Tour of C++: Abstraction Mechanisms* presents the language features supporting data abstraction, object-oriented programming, and generic programming.
Chapter 4	*A Tour of C++: Containers and Algorithms* introduces strings, simple I/O, containers, and algorithms as provided by the standard library.
Chapter 5	*A Tour of C++: Concurrency and Utilities* outlines the standard-library utilities related to resource management, concurrency, mathematical computation, regular expressions, and more.

This whirlwind tour of C++'s facilities aims to give the reader a taste of what C++ offers. In particular, it should convince readers that C++ has come a long way since the first, second, and third editions of this book.

1.1.2 Basic Facilities

Part II focuses on the subset of C++ that supports the styles of programming traditionally done in C and similar languages. It introduces the notions of type, object, scope, and storage. It presents the fundamentals of computation: expressions, statements, and functions. Modularity – as supported by namespaces, source files, and exception handling – is also discussed:

Chapter 6	*Types and Declarations*: Fundamental types, naming, scopes, initialization, simple type deduction, object lifetimes, and type aliases

Chapter 7	*Pointers, Arrays, and References*
Chapter 8	*Structures, Unions, and Enumerations*
Chapter 9	*Statements*: Declarations as statements, selection statements (**if** and **switch**), iteration statements (**for**, **while**, and **do**), **goto**, and comments
Chapter 10	*Expressions*: A desk calculator example, survey of operators, constant expressions, and implicit type conversion.
Chapter 11	*Select Operations*: Logical operators, the conditional expression, increment and decrement, free store (**new** and **delete**), **{}**-lists, lambda expressions, and explicit type conversion (**static_cast** and **const_cast**)
Chapter 12	*Functions*: Function declarations and definitions, **inline** functions, **constexpr** functions, argument passing, overloaded functions, pre- and postconditions, pointers to functions, and macros
Chapter 13	*Exception Handling*: Styles of error handling, exception guarantees, resource management, enforcing invariants, **throw** and **catch**, a **vector** implementation
Chapter 14	*Namespaces*: **namespace**, modularization and interface, composition using namespaces
Chapter 15	*Source Files and Programs*: Separate compilation, linkage, using header files, and program start and termination

I assume that you are familiar with most of the programming concepts used in Part I. For example, I explain the C++ facilities for expressing recursion and iteration, but I do not go into technical details or spend much time explaining how these concepts are useful.

The exception to this rule is exceptions. Many programmers lack experience with exceptions or got their experience from languages (such as Java) where resource management and exception handling are not integrated. Consequently, the chapter on exception handling (Chapter 13) presents the basic philosophy of C++ exception handling and resource management. It goes into some detail about strategy with a focus on the ‘‘Resource Acquisition Is Initialization’’ technique (RAII).

1.1.3 Abstraction Mechanisms

Part III describes the C++ facilities supporting various forms of abstraction, including object-oriented and generic programming. The chapters fall into three rough categories: classes, class hierarchies, and templates.

The first four chapters concentrate of the classes themselves:

Chapter 16	*Classes*: The notion of a user-defined type, a class, is the foundation of all C++ abstraction mechanisms.
Chapter 17	*Construction, Cleanup, Copy, and Move* shows how a programmer can define the meaning of creation and initialization of objects of a class. Further, the meaning of copy, move, and destruction can be specified.
Chapter 18	*Operator Overloading* presents the rules for giving meaning to operators for user-defined types with an emphasis on conventional arithmetic and logical operators, such as **+**, **∗**, and **&**.
Chapter 19	*Special Operators* discusses the use of user-defined operator for non-arithmetic purposes, such as **[]** for subscripting, **()** for function objects, and **−>** for ‘‘smart pointers.’’

Classes can be organized into hierarchies:

Chapter 20	*Derived Classes* presents the basic language facilities for building hierarchies out of classes and the fundamental ways of using them. We can provide complete separation between an interface (an abstract class) and its implementations (derived classes); the connection between them is provided by virtual functions. The C++ model for access control (**public**, **protected**, and **private**) is presented.
Chapter 21	*Class Hierarchies* discusses ways of using class hierarchies effectively. It also presents the notion of multiple inheritance, that is, a class having more than one direct base class.
Chapter 22	*Run-Time Type Information* presents ways to navigate class hierarchies using data stored in objects. We can use **dynamic_cast** to inquire whether an object of a base class was defined as an object of a derived class and use the **typeid** to gain minimal information from an object (such as the name of its class).

Many of the most flexible, efficient, and useful abstractions involve the parameterization of types (classes) and algorithms (functions) with other types and algorithms:

Chapter 23	*Templates* presents the basic principles behind templates and their use. Class templates, function templates, and template aliases are presented.
Chapter 24	*Generic Programming* introduces the basic techniques for designing generic programs. The technique of *lifting* an abstract algorithm from a number of concrete code examples is central, as is the notion of *concepts* specifying a generic algorithm's requirements on its arguments.
Chapter 25	*Specialization* describes how templates are used to generate classes and functions, *specializations*, given a set of template arguments.
Chapter 26	*Instantiation* focuses on the rules for name binding.
Chapter 27	*Templates and Hierarchies* explains how templates and class hierarchies can be used in combination.
Chapter 28	*Metaprogramming* explores how templates can be used to generate programs. Templates provide a Turing-complete mechanism for generating code.
Chapter 29	*A Matrix Design* gives a longish example to show how language features can be used in combination to solve a complex design problem: the design of an N-dimensional matrix with near-arbitrary element types.

The language features supporting abstraction techniques are described in the context of those techniques. The presentation technique in Part III differs from that of Part II in that I don't assume that the reader knows the techniques described.

1.1.4 The Standard Library

The library chapters are less tutorial than the language chapters. In particular, they are meant to be read in any order and can be used as a user-level manual for the library components:

Chapter 30	*Standard-Library Overview* gives an overview of the standard library, lists the standard-library headers, and presents language support and diagnostics support, such as **exception** and **system_error**.
Chapter 31	*STL Containers* presents the containers from the iterators, containers, and algorithms framework (called *the STL*), including **vector**, **map**, and **unordered_set**.

Chapter 32	*STL Algorithms* presents the algorithms from the STL, including **find()**, **sort()**, and **merge()**.
Chapter 33	*STL Iterators* presents iterators and other utilities from the STL, including **reverse_iterator**, **move_iterator**, and **function**.
Chapter 34	*Memory and Resources* presents utility components related to memory and resource management, such as **array**, **bitset**, **pair**, **tuple**, **unique_ptr**, **shared_ptr**, allocators, and the garbage collector interface.
Chapter 35	*Utilities* presents minor utility components, such as time utilities, type traits, and various type functions.
Chapter 36	*Strings* documents the **string** library, including the character traits that are the basis for the use of different character sets.
Chapter 37	*Regular Expressions* describes the regular expression syntax and the various ways of using it for string matching, including **regex_match()** for matching a complete string, **regex_search()** for finding a pattern in a string, **regex_replace()** for simple replacement, and **regex_iterator** for general traversal of a stream of characters.
Chapter 38	*I/O Streams* documents the stream I/O library. It describes formatted and unformatted input and output, error handling, and buffering.
Chapter 39	*Locales* describes class **locale** and its various **facet**s that provide support for the handling of cultural differences in character sets, formatting of numeric values, formatting of date and time, and more.
Chapter 40	*Numerics* describes facilities for numerical computation (such as **complex**, **valarray**, random numbers, and generalized numerical algorithms).
Chapter 41	*Concurrency* presents the C++ basic memory model and the facilities offered for concurrent programming without locks.
Chapter 42	*Threads and Tasks* presents the classes providing threads-and-locks-style concurrent programming (such as **thread**, **timed_mutex**, **lock_guard**, and **try_lock()**) and the support for task-based concurrency (such as **future** and **async()**).
Chapter 43	*The C Standard Library* documents the C standard library (including **printf()** and **clock()**) as incorporated into the C++ standard library.
Chapter 44	*Compatibility* discusses the relation between C and C++ and between Standard C++ (also called ISO C++) and the versions of C++ that preceded it.

1.1.5 Examples and References

This book emphasizes program organization rather than the design of algorithms. Consequently, I avoid clever or harder-to-understand algorithms. A trivial algorithm is typically better suited to illustrate an aspect of the language definition or a point about program structure. For example, I use a Shell sort where, in real code, a quicksort would be better. Often, reimplementation with a more suitable algorithm is an exercise. In real code, a call of a library function is typically more appropriate than the code used here to illustrate language features.

Textbook examples necessarily give a warped view of software development. By clarifying and simplifying the examples, the complexities that arise from scale disappear. I see no substitute for writing realistically sized programs in order to get an impression of what programming and a programming language are really like. This book concentrates on the language features and the standard-library facilities. These are the basic techniques from which every program is composed. The rules and techniques for such composition are emphasized.

The selection of examples reflects my background in compilers, foundation libraries, and simulations. The emphasis reflects my interest in systems programming. Examples are simplified versions of what is found in real code. The simplification is necessary to keep programming language and design points from getting lost in details. My ideal is the shortest and clearest example that illustrates a design principle, a programming technique, a language construct, or a library feature. There are no ‘‘cute’’ examples without counterparts in real code. For purely language-technical examples, I use variables named `x` and `y`, types called `A` and `B`, and functions called `f()` and `g()`.

Where possible, the C++ language and library features are presented in the context of their use rather than in the dry manner of a manual. The language features presented and the detail in which they are described roughly reflect my view of what is needed for effective use of C++. The purpose is to give you an idea of how a feature can be used, often in combination with other features. An understanding of every language-technical detail of a language feature or library component is neither necessary nor sufficient for writing good programs. In fact, an obsession with understanding every little detail is a prescription for awful – overelaborate and overly clever – code. What is needed is an understanding of design and programming techniques together with an appreciation of application domains.

I assume that you have access to online information sources. The final arbiter of language and standard-library rules is the ISO C++ standard [C++,2011].

References to parts of this book are of the form §2.3.4 (Chapter 2, section 3, subsection 4) and §iso.5.3.1 (ISO C++ standard, §5.3.1). Italics are used sparingly for emphasis (e.g., ‘‘a string literal is *not* acceptable’’), for first occurrences of important concepts (e.g., *polymorphism*), and for comments in code examples.

To save a few trees and to simplify additions, the hundreds of exercises for this book have been moved to the Web. Look for them at www.stroustrup.com.

The language and library used in this book are ‘‘pure C++’’ as defined by the C++ standard [C++,2011]. Therefore, the examples should run on every up-to-date C++ implementation. The major program fragments in this book were tried using several C++ implementations. Examples using features only recently adopted into C++ didn’t compile on every implementation. However, I see no point in mentioning which implementations failed to compile which examples. Such information would soon be out of date because implementers are working hard to ensure that their implementations correctly accept every C++ feature. See Chapter 44 for suggestions on how to cope with older C++ compilers and with code written for C compilers.

I use C++11 features freely wherever I find them most appropriate. For example, I prefer `{}`-style initializers and `using` for type aliases. In places, that usage may startle ‘‘old timers.’’ However, being startled is often a good way to start reviewing material. On the other hand, I don’t use new features just because they are new; my ideal is the most elegant expression of the fundamental ideas – and that may very well be using something that has been in C++ or even in C for ages.

Obviously, if you have to use a pre-C++11 compiler (say, because some of your customers have not yet upgraded to the current standard), you have to refrain from using novel features. However, please don’t assume that ‘‘the old ways’’ are better or simpler just because they are old and familiar. §44.2 summarizes the differences between C++98 and C++11.

1.2 The Design of C++

The purpose of a programming language is to help express ideas in code. In that, a programming language performs two related tasks: it provides a vehicle for the programmer to specify actions to be executed by the machine, and it provides a set of concepts for the programmer to use when thinking about what can be done. The first purpose ideally requires a language that is ‘‘close to the machine’’ so that all important aspects of a machine are handled simply and efficiently in a way that is reasonably obvious to the programmer. The C language was primarily designed with this in mind. The second purpose ideally requires a language that is ‘‘close to the problem to be solved’’ so that the concepts of a solution can be expressed directly and concisely. The facilities added to C to create C++, such as function argument checking, `const`, classes, constructors and destructors, exceptions, and templates, were primarily designed with this in mind. Thus, C++ is based on the idea of providing both

- *direct mappings of built-in operations and types to hardware* to provide efficient memory use and efficient low-level operations, and
- *affordable and flexible abstraction mechanisms* to provide user-defined types with the same notational support, range of uses, and performance as built-in types.

This was initially achieved by applying ideas from Simula to C. Over the years, further application of these simple ideals resulted in a far more general, efficient, and flexible set of facilities. The result supports a synthesis of programming styles that can be simultaneously *efficient* and *elegant*.

The design of C++ has focused on programming techniques dealing with fundamental notions such as memory, mutability, abstraction, resource management, expression of algorithms, error handling, and modularity. Those are the most important concerns of a systems programmer and more generally of programmers of resource-constrained and high-performance systems.

By defining libraries of classes, class hierarchies, and templates, you can write C++ programs at a much higher level than the one presented in this book. For example, C++ is widely used in financial systems, for game development, and for scientific computation (§1.4.5). For high-level applications programming to be effective and convenient, we need libraries. Using just the bare language features makes almost all programming quite painful. That’s true for every general-purpose language. Conversely, given suitable libraries just about any programming task can be pleasant.

My standard introduction of C++ used to start:

- *C++ is a general-purpose programming language with a bias toward systems programming.*

This is still true. What has changed over the years is an increase in the importance, power, and flexibility of C++’s abstraction mechanisms:

- *C++ is a general-purpose programming language providing a direct and efficient model of hardware combined with facilities for defining lightweight abstractions.*

Or terser:

- *C++ is a language for developing and using elegant and efficient abstractions.*

By *general-purpose programming language* I mean a language designed to support a wide variety of uses. C++ has indeed been used for an incredible variety of uses (from microcontrollers to huge distributed commercial applications), but the key point is that C++ is not deliberately specialized for any given application area. No language is ideal for every application and every programmer, but the ideal for C++ is to support the widest possible range of application areas well.

By *systems programming* I mean writing code that directly uses hardware resources, has serious resource constraints, or closely interacts with code that does. In particular, the implementation of software infrastructure (e.g., device drivers, communications stacks, virtual machines, operating systems, operations systems, programming environments, and foundation libraries) is mostly systems programming. The importance of the ‘‘bias toward systems programming’’ qualification in my long-standing characterization of C++ is that C++ has not been simplified (compromised) by ejecting the facilities aimed at the expert-level use of hardware and systems resources in the hope of making it more suitable for other application areas.

Of course, you can also program in ways that completely hide hardware, use expensive abstractions (e.g., every object on the free store and every operation a virtual function), use inelegant styles (e.g., overabstraction), or use essentially no abstractions (‘‘glorified assembly code’’). However, many languages can do that, so those are not distinguishing characteristics of C++.

The *Design and Evolution of C++* book [Stroustrup,1994] (known as *D&E*) outlines the ideas and design aims of C++ in greater detail, but two principles should be noted:

- *Leave no room for a lower-level language below C++* (except for assembly code in rare cases). If you can write more efficient code in a lower-level language then that language will most likely become the systems programming language of choice.
- *What you don’t use you don’t pay for*. If programmers can hand-write reasonable code to simulate a language feature or a fundamental abstraction and provide even slightly better performance, someone will do so, and many will imitate. Therefore, a language feature and a fundamental abstraction must be designed not to waste a single byte or a single processor cycle compared to equivalent alternatives. This is known as *the zero-overhead principle.*

These are Draconian principles, but essential in some (but obviously not all) contexts. In particular, the zero-overhead principle repeatedly led C++ to simpler, more elegant, and more powerful facilities than were first envisioned. The STL is an example (§4.1.1, §4.4, §4.5, Chapter 31, Chapter 32, Chapter 33). These principles have been essential in the effort to raise the level of programming.

1.2.1 Programming Style

Languages features exist to provide support for programming styles. Please don’t look at an individual language feature as a solution, but as one building brick from a varied set which can be combined to express solutions.

The general ideals for design and programming can be expressed simply:

- Express ideas directly in code.
- Express independent ideas independently in code.
- Represent relationships among ideas directly in code.
- Combine ideas expressed in code freely – where and only where combinations make sense.
- Express simple ideas simply.

These are ideals shared by many people, but languages designed to support them can differ dramatically. A fundamental reason for that is that a language embodies a set of engineering tradeoffs reflecting differing needs, tastes, and histories of various individuals and communities. C++’s answers to the general design challenges were shaped by its origins in systems programming (going back to C and BCPL [Richards,1980]), its aim to address issues of program complexity through abstraction (going back to Simula), and its history.

The C++ language features most directly support four programming styles:

- Procedural programming
- Data abstraction
- Object-oriented programming
- Generic programming

However, the emphasis is on the support of effective combinations of those. The best (most maintainable, most readable, smallest, fastest, etc.) solution to most nontrivial problems tends to be one that combines aspects of these styles.

As is usual with important terms in the computing world, a wide variety of definitions of these terms are popular in various parts of the computing industry and academia. For example, what I refer to as a ‘‘programming style,’’ others call a ‘‘programming technique’’ or a ‘‘paradigm.’’ I prefer to use ‘‘programming technique’’ for something more limited and language-specific. I feel uncomfortable with the word ‘‘paradigm’’ as pretentious and (from Kuhn’s original definition) having implied claims of exclusivity.

My ideal is language facilities that can be used elegantly in combination to support a continuum of programming styles and a wide variety of programming techniques.

- *Procedural programming*: This is programming focused on processing and the design of suitable data structures. It is what C was designed to support (and Algol, and Fortran, as well as many other languages). C++’s support comes in the form of the built-in types, operators, statements, functions, `struct`s, `union`s, etc. With minor exceptions, C is a subset of C++. Compared to C, C++ provides further support for procedural programming in the form of many additional language constructs and a stricter, more flexible, and more supportive type system.
- *Data abstraction*: This is programming focused on the design of interfaces, hiding implementation details in general and representations in particular. C++ supports concrete and abstract classes. The facilities for defining classes with private implementation details, constructors and destructors, and associated operations directly support this. The notion of an abstract class provides direct support for complete data hiding.
- *Object-oriented programming*: This is programming focused on the design, implementation, and use of class hierarchies. In addition to allowing the definition lattices of classes, C++ provides a variety of features for navigating class lattices and for simplifying the definition of a class out of existing ones. Class hierarchies provide run-time polymorphism (§20.3.2, §21.2) and encapsulation (§20.4, §20.5).
- *Generic programming*: This is programming focused on the design, implementation, and use of general algorithms. Here, ‘‘general’’ means that an algorithm can be designed to accept a wide variety of types as long as they meet the algorithm’s requirements on its arguments. The template is C++’s main support for generic programming. Templates provide (compile-time) parametric polymorphism.

Just about anything that increases the flexibility or efficiency of classes improves the support of all of those styles. Thus, C++ could be (and has been) called *class oriented*.

Each of these styles of design and programming has contributed to the synthesis that is C++. Focusing exclusively on one of these styles is a mistake: except for toy examples, doing so leads to wasted development effort and suboptimal (inflexible, verbose, poorly performing, unmaintainable, etc.) code.

I wince when someone characterizes C++ exclusively through one of these styles (e.g., ‘‘C++ is an object-oriented language’’) or uses a term (e.g., ‘‘hybrid’’ or ‘‘mixed paradigm’’) to imply that a more restrictive language would be preferable. The former misses the fact that all the styles mentioned have contributed something significant to the synthesis; the latter denies the validity of the synthesis. The styles mentioned are not distinct alternatives: each contributes techniques to a more expressive and effective style of programming, and C++ provides direct language support for their use in combination.

From its inception, the design of C++ aimed at a synthesis of programming and design styles. Even the earliest published account of C++ [Stroustrup,1982] presents examples that use these different styles in combination and presents language features aimed at supporting such combinations:

- *Classes* support all of the mentioned styles; all rely on the user representing ideas as user-defined types or objects of user-defined types.
- *Public/private access control* supports data abstraction and object-oriented programming by making a clear distinction between interface and implementation.
- *Member functions, constructors, destructors, and user-defined assignment* provide a clean functional interface to objects as needed by data abstraction and object-oriented programming. They also provide a uniform notation as needed for generic programming. More general overloading had to wait until 1984 and uniform initialization until 2010.
- *Function declarations* provide specific statically checked interfaces to member functions as well as freestanding functions, so they support all of the mentioned styles. They are necessary for overloading. At the time, C lacked ‘‘function prototypes’’ but Simula had function declarations as well as member functions.
- *Generic functions and parameterized types* (generated from functions and classes using macros) support generic programming. Templates had to wait until 1988.
- *Base and derived classes* provide the foundation for object-oriented programming and some forms of data abstraction. Virtual functions had to wait until 1983.
- *Inlining* made the use of these facilities affordable in systems programming and for building run-time and space efficient libraries.

These early features are general abstraction mechanisms, rather than support for disjoint programming styles. Today’s C++ provides much better support for design and programming based on lightweight abstraction, but the aim of elegant and efficient code was there from the very beginning. The developments since 1981 provide much better support for the synthesis of the programming styles (‘‘paradigms’’) originally considered and significantly improve their integration.

The fundamental object in C++ has identity; that is, it is located in a specific location in memory and can be distinguished from other objects with (potentially) the same value by comparing addresses. Expressions denoting such objects are called *lvalues* (§6.4). However, even from the earliest days of C++’s ancestors [Barron,1963] there have also been objects without identity (objects for which an address cannot be safely stored for later use). In C++11, this notion of *rvalue* has been developed into a notion of a value that can be moved around cheaply (§3.3.2, §6.4.1, §7.7.2). Such objects are the basis of techniques that resemble what is found in functional programming (where the notion of objects with identity is viewed with horror). This nicely complements the techniques and language features (e.g., lambda expressions) developed primarily for generic programming. It also solves classical problems related to ‘‘simple abstract data types,’’ such as how to elegantly and efficiently return a large matrix from an operation (e.g., a matrix **+**).

From the very earliest days, C++ programs and the design of C++ itself have been concerned about resource management. The ideal was (and is) for resource management to be

- simple (for implementers and especially for users),
- general (a resource is anything that has to be acquired from somewhere and later released),
- efficient (obey the zero-overhead principle; §1.2),
- perfect (no leaks are acceptable), and
- statically type-safe.

Many important C++ classes, such as the standard library's **vector**, **string**, **thread**, **mutex**, **unique_ptr**, **fstream**, and **regex**, are resource handles. Foundation and application libraries beyond the standard provided many more examples, such as Matrix and Widget. The initial step in supporting the notion of resource handles was taken with the provision of constructors and destructors in the very first ''C with Classes'' draft. This was soon backed with the ability to control copy by defining assignment as well as copy constructors. The introduction of move constructors and move assignments (§3.3) in C++11 completes this line of thinking by allowing cheap movement of potentially large objects from scope to scope (§3.3.2) and to simply control the lifetime of polymorphic or shared objects (§5.2.1).

The facilities supporting resource management also benefit abstractions that are not resource handles. Any class that establishes and maintains an invariant relies on a subset of those features.

1.2.2 Type Checking

The connection between the language in which we think/program and the problems and solutions we can imagine is very close. For this reason, restricting language features with the intent of eliminating programmer errors is, at best, dangerous. A language provides a programmer with a set of conceptual tools; if these are inadequate for a task, they will be ignored. Good design and the absence of errors cannot be guaranteed merely by the presence or absence of specific language features. However, the language features and the type system are provided for the programmer to precisely and concisely represent a design in code.

The notion of static types and compile-time type checking is central to effective use of C++. The use of static types is key to expressiveness, maintainability, and performance. Following Simula, the design of user-defined types with interfaces that are checked at compile time is key to the expressiveness of C++. The C++ type system is extensible in nontrivial ways (Chapter 3, Chapter 16, Chapter 18, Chapter 19, Chapter 21, Chapter 23, Chapter 28, Chapter 29), aiming for equal support for built-in types and user-defined types.

C++ type-checking and data-hiding features rely on compile-time analysis of programs to prevent accidental corruption of data. They do not provide secrecy or protection against someone who is deliberately breaking the rules: C++ protects against accident, not against fraud. They can, however, be used freely without incurring run-time or space overheads. The idea is that to be useful, a language feature must not only be elegant, it must also be affordable in the context of a real-world program.

C++'s static type system is flexible, and the use of simple user-defined types implies little, if any overhead. The aim is to support a style of programming that represents distinct ideas as distinct types, rather than just using generalizations, such as integer, floating-point number, string, ''raw memory,'' and ''object,'' everywhere. A type-rich style of programming makes code more readable, maintainable, and analyzable. A trivial type system allows only trivial analysis, whereas a type-rich style of programming opens opportunities for nontrivial error detection and optimization. C++ compilers and development tools support such type-based analysis [Stroustrup,2012].

Maintaining most of C as a subset and preserving the direct mapping to hardware needed for the most demanding low-level systems programming tasks implies the ability to break the static type system. However, my ideal is (and always was) complete type safety. In this, I agree with Dennis Ritchie, who said, ‘‘C is a strongly typed, weakly checked language.’’ Note that Simula was both type-safe and flexible. In fact, my ideal when I started on C++ was ‘‘Algol68 with Classes’’ rather than ‘‘C with Classes.’’ However, the list of solid reasons against basing my work on type-safe Algol68 [Woodward,1974] was long and painful. So, perfect type safety is an ideal that C++ as a language can only approximate. But it is an ideal that C++ programmers (especially library builders) can strive for. Over the years, the set of language features, standard-library components, and techniques supporting that ideal has grown. Outside of low-level sections of code (hopefully isolated by type-safe interfaces), code that interfaces to code obeying different language conventions (e.g., an operating system call interface), and the implementations of fundamental abstractions (e.g., `string` and `vector`), there is now little need for type-unsafe code.

1.2.3 C Compatibility

C++ was developed from the C programming language and, with few exceptions, retains C as a subset. The main reasons for relying on C were to build on a proven set of low-level language facilities and to be part of a technical community. Great importance was attached to retaining a high degree of compatibility with C [Koenig,1989] [Stroustrup,1994] (Chapter 44); this (unfortunately) precluded cleaning up the C syntax. The continuing, more or less parallel evolution of C and C++ has been a constant source of concern and requires constant attention [Stroustrup,2002]. Having two committees devoted to keeping two widely used languages ‘‘as compatible as possible’’ is not a particularly good way of organizing work. In particular, there are differences in opinion as to the value of compatibility, differences in opinion on what constitutes good programming, and differences in opinion on what support is needed for good programming. Just keeping up communication between the committees is a large amount of work.

One hundred percent C/C++ compatibility was never a goal for C++ because that would compromise type safety and the smooth integration of user-defined and built-in types. However, the definition of C++ has been repeatedly reviewed to remove gratuitous incompatibilities; C++ is now more compatible with C than it was originally. C++98 adopted many details from C89 (§44.3.1). When C then evolved from C89 [C,1990] to C99 [C,1999], C++ adopted almost all of the new features, leaving out VLAs (variable-length arrays) as a misfeature and designated initializers as redundant. C’s facilities for low-level systems programming tasks are retained and enhanced; for example, see inlining (§3.2.1.1, §12.1.5, §16.2.8) and `constexpr` (§2.2.3, §10.4, §12.1.6).

Conversely, modern C has adopted (with varying degrees of faithfulness and effectiveness) many features from C++ (e.g., `const`, function prototypes, and inlining; see [Stroustrup,2002]).

The definition of C++ has been revised to ensure that a construct that is both legal C and legal C++ has the same meaning in both languages (§44.3).

One of the original aims for C was to replace assembly coding for the most demanding systems programming tasks. When C++ was designed, care was taken not to compromise the gains in this area. The difference between C and C++ is primarily in the degree of emphasis on types and structure. C is expressive and permissive. Through extensive use of the type system, C++ is even more expressive without loss of performance.

Knowing C is not a prerequisite for learning C++. Programming in C encourages many techniques and tricks that are rendered unnecessary by C++ language features. For example, explicit type conversion (casting) is less frequently needed in C++ than it is in C (§1.3.3). However, *good* C programs tend to be C++ programs. For example, every program in Kernighan and Ritchie, *The C Programming Language, Second Edition* [Kernighan,1988], is a C++ program. Experience with any statically typed language will be a help when learning C++.

1.2.4 Language, Libraries, and Systems

The C++ fundamental (built-in) types, operators, and statements are those that computer hardware deals with directly: numbers, characters, and addresses. C++ has no built-in high-level data types and no high-level primitive operations. For example, the C++ language does not provide a matrix type with an inversion operator or a string type with a concatenation operator. If a user wants such a type, it can be defined in the language itself. In fact, defining a new general-purpose or application-specific type is the most fundamental programming activity in C++. A well-designed user-defined type differs from a built-in type only in the way it is defined, not in the way it is used. The C++ standard library (Chapter 4, Chapter 5, Chapter 30, Chapter 31, etc.) provides many examples of such types and their uses. From a user's point of view, there is little difference between a built-in type and a type provided by the standard library. Except for a few unfortunate and unimportant historical accidents, the C++ standard library is written in C++. Writing the C++ standard library in C++ is a crucial test of the C++ type system and abstraction mechanisms: they must be (and are) sufficiently powerful (expressive) and efficient (affordable) for the most demanding systems programming tasks. This ensures that they can be used in large systems that typically consist of layer upon layer of abstraction.

Features that would incur run-time or memory overhead even when not used were avoided. For example, constructs that would make it necessary to store ''housekeeping information'' in every object were rejected, so if a user declares a structure consisting of two 16-bit quantities, that structure will fit into a 32-bit register. Except for the `new`, `delete`, `typeid`, `dynamic_cast`, and `throw` operators, and the `try`-block, individual C++ expressions and statements need no run-time support. This can be essential for embedded and high-performance applications. In particular, this implies that the C++ abstraction mechanisms are usable for embedded, high-performance, high-reliability, and real-time applications. So, programmers of such applications don't have to work with a low-level (error-prone, impoverished, and unproductive) set of language features.

C++ was designed to be used in a traditional compilation and run-time environment: the C programming environment on the UNIX system [UNIX,1985]. Fortunately, C++ was never restricted to UNIX; it simply used UNIX and C as a model for the relationships among language, libraries, compilers, linkers, execution environments, etc. That minimal model helped C++ to be successful on essentially every computing platform. There are, however, good reasons for using C++ in environments that provide significantly more run-time support. Facilities such as dynamic loading, incremental compilation, and a database of type definitions can be put to good use without affecting the language.

Not every piece of code can be well structured, hardware-independent, easy to read, etc. C++ possesses features that are intended for manipulating hardware facilities in a direct and efficient way without concerns for safety or ease of comprehension. It also possesses facilities for hiding such code behind elegant and safe interfaces.

Naturally, the use of C++ for larger programs leads to the use of C++ by groups of programmers. C++'s emphasis on modularity, strongly typed interfaces, and flexibility pays off here. However, as programs get larger, the problems associated with their development and maintenance shift from being language problems to being more global problems of tools and management.

This book emphasizes techniques for providing general-purpose facilities, generally useful types, libraries, etc. These techniques will serve programmers of small programs as well as programmers of large ones. Furthermore, because all nontrivial programs consist of many semi-independent parts, the techniques for writing such parts serve programmers of all applications.

I use the implementation and use of standard-library components, such as **vector**, as examples. This introduces library components and their underlying design concepts and implementation techniques. Such examples show how programmers might design and implement their own libraries. However, if the standard library provides a component that addresses a problem, it is almost always better to use that component than to build your own. Even if the standard component is arguably slightly inferior to a home-built component for a particular problem, the standard component is likely to be more widely applicable, more widely available, and more widely known. Over the longer term, the standard component (possibly accessed through a convenient custom interface) is likely to lower long-term maintenance, porting, tuning, and education costs.

You might suspect that specifying a program by using a more detailed type structure would increase the size of the program source text (or even the size of the generated code). With C++, this is not so. A C++ program declaring function argument types, using classes, etc., is typically a bit shorter than the equivalent C program not using these facilities. Where libraries are used, a C++ program will appear much shorter than its C equivalent, assuming, of course, that a functioning C equivalent could have been built.

C++ supports systems programming. This implies that C++ code is able to effectively interoperate with software written in other languages on a system. The idea of writing all software in a single language is a fantasy. From the beginning, C++ was designed to interoperate simply and efficiently with C, assembler, and Fortran. By that, I meant that a C++, C, assembler, or Fortran function could call functions in the other languages without extra overhead or conversion of data structures passed among them.

C++ was designed to operate within a single address space. The use of multiple processes and multiple address spaces relied on (extralinguistic) operating system support. In particular, I assumed that a C++ programmer would have the operating systems command language available for composing processes into a system. Initially, I relied on the UNIX Shell for that, but just about any "scripting language" will do. Thus, C++ provided no support for multiple address spaces and no support for multiple processes, but it was used for systems relying on those features from the earliest days. C++ was designed to be part of large, concurrent, multilanguage systems.

1.3 Learning C++

No programming language is perfect. Fortunately, a programming language does not have to be perfect to be a good tool for building great systems. In fact, a general-purpose programming language cannot be perfect for all of the many tasks to which it is put. What is perfect for one task is often seriously flawed for another because perfection in one area implies specialization. Thus, C++ was designed to be a good tool for building a wide variety of systems and to allow a wide variety of ideas to be expressed directly.

Not everything can be expressed directly using the built-in features of a language. In fact, that isn't even the ideal. Language features exist to support a variety of programming styles and techniques. Consequently, the task of learning a language should focus on mastering the native and natural styles for that language – not on understanding of every little detail of every language feature. Writing programs is essential; understanding a programming language is not just an intellectual exercise. Practical application of ideas is necessary.

In practical programming, there is little advantage in knowing the most obscure language features or using the largest number of features. A single language feature in isolation is of little interest. Only in the context provided by techniques and by other features does the feature acquire meaning and interest. Thus, when reading the following chapters, please remember that the real purpose of examining the details of C++ is to be able to use language features and library facilities in concert to support good programming styles in the context of sound designs.

No significant system is built exclusively in terms of the language features themselves. We build and use libraries to simplify the task of programming and to increase the quality of our systems. We use libraries to improve maintainability, portability, and performance. Fundamental application concepts are represented as abstractions (e.g., classes, templates, and class hierarchies) in libraries. Many of the most fundamental programming concepts are represented in the standard library. Thus, learning the standard library is an integral part of learning C++. The standard library is the repository of much hard-earned knowledge of how to use C++ well.

C++ is widely used for teaching and research. This has surprised some who – correctly – point out that C++ isn't the smallest or cleanest language ever designed. It is, however:

- Sufficiently clean for successfully teaching basic design and programming concepts
- Sufficiently comprehensive to be a vehicle for teaching advanced concepts and techniques
- Sufficiently realistic, efficient, and flexible for demanding projects
- Sufficiently commercial to be a vehicle for putting what is learned into nonacademic use
- Sufficiently available for organizations and collaborations relying on diverse development and execution environments

C++ is a language that you can grow with.

The most important thing to do when learning C++ is to focus on fundamental concepts (such as type safety, resource management, and invariants) and programming techniques (such as resource management using scoped objects and the use of iterators in algorithms) and not get lost in language-technical details. The purpose of learning a programming language is to become a better programmer, that is, to become more effective at designing and implementing new systems and at maintaining old ones. For this, an appreciation of programming and design techniques is far more important than understanding all the details. The understanding of technical details comes with time and practice.

C++ programming is based on strong static type checking, and most techniques aim at achieving a high level of abstraction and a direct representation of the programmer's ideas. This can usually be done without compromising run-time and space efficiency compared to lower-level techniques. To gain the benefits of C++, programmers coming to it from a different language must learn and internalize idiomatic C++ programming style and technique. The same applies to programmers used to earlier and less expressive versions of C++.

Thoughtlessly applying techniques effective in one language to another typically leads to awkward, poorly performing, and hard-to-maintain code. Such code is also most frustrating to write because every line of code and every compiler error message reminds the programmer that the language used differs from ‘‘the old language.’’ You can write in the style of Fortran, C, Lisp, Java, etc., in any language, but doing so is neither pleasant nor economical in a language with a different philosophy. Every language can be a fertile source of ideas about how to write C++ programs. However, ideas must be transformed into something that fits with the general structure and type system of C++ in order to be effective in C++. Over the basic type system of a language, only Pyrrhic victories are possible.

In the continuing debate on whether one needs to learn C before C++, I am firmly convinced that it is best to go directly to C++. C++ is safer and more expressive, and it reduces the need to focus on low-level techniques. It is easier for you to learn the trickier parts of C that are needed to compensate for its lack of higher-level facilities after you have been exposed to the common subset of C and C++ and to some of the higher-level techniques supported directly in C++. Chapter 44 is a guide for programmers going from C++ to C, say, to deal with legacy code. My opinion on how to teach C++ to novices is represented by [Stroustrup,2008].

There are several independently developed implementations of C++. They are supported by a wealth of tools, libraries, and software development environments. To help master all of this you can find textbooks, manuals, and a bewildering variety of online resources. If you plan to use C++ seriously, I strongly suggest that you obtain access to several such sources. Each has its own emphasis and bias, so use at least two.

1.3.1 Programming in C++

The question ‘‘How does one write good programs in C++?’’ is very similar to the question ‘‘How does one write good English prose?’’ There are two answers: ‘‘Know what you want to say’’ and ‘‘Practice. Imitate good writing.’’ Both appear to be as appropriate for C++ as they are for English – and as hard to follow.

The main ideal for C++ programming – as for programming in most higher-level languages – is to express concepts (ideas, notions, etc.) from a design directly in code. We try to ensure that the concepts we talk about, represent with boxes and arrows on our whiteboard, and find in our (non-programming) textbooks have direct and obvious counterparts in our programs:

[1] Represent ideas directly in code.
[2] Represent relationships among ideas directly in code (e.g., hierarchical, parametric, and ownership relationships).
[3] Represent independent ideas independently in code.
[4] Keep simple things simple (without making complex things impossible).

More specifically:

[5] Prefer statically type-checked solutions (when applicable).
[6] Keep information local (e.g., avoid global variables, minimize the use of pointers).
[7] Don't overabstract (i.e., don't generalize, introduce class hierarchies, or parameterize beyond obvious needs and experience).

More specific suggestions are listed in §1.3.2.

1.3.2 Suggestions for C++ Programmers

By now, many people have been using C++ for a decade or two. Many more are using C++ in a single environment and have learned to live with the restrictions imposed by early compilers and first-generation libraries. Often, what an experienced C++ programmer has failed to notice over the years is not the introduction of new features as such, but rather the changes in relationships between features that make fundamental new programming techniques feasible. In other words, what you didn't think of when first learning C++ or found impractical just might be a superior approach today. You find out only by reexamining the basics.

Read through the chapters in order. If you already know the contents of a chapter, you can be done in minutes. If you don't already know the contents, you'll have learned something unexpected. I learned a fair bit writing this book, and I suspect that hardly any C++ programmer knows every feature and technique presented. Furthermore, to use the language well, you need a perspective that brings order to the set of features and techniques. Through its organization and examples, this book offers such a perspective.

Take the opportunity offered by the new C++11 facilities to modernize your design and programming techniques:

[1] Use constructors to establish invariants (§2.4.3.2, §13.4, §17.2.1).
[2] Use constructor/destructor pairs to simplify resource management (RAII; §5.2, §13.3).
[3] Avoid ‘‘naked’’ `new` and `delete` (§3.2.1.2, §11.2.1).
[4] Use containers and algorithms rather than built-in arrays and ad hoc code (§4.4, §4.5, §7.4, Chapter 32).
[5] Prefer standard-library facilities to locally developed code (§1.2.4).
[6] Use exceptions, rather than error codes, to report errors that cannot be handled locally (§2.4.3, §13.1).
[7] Use move semantics to avoid copying large objects (§3.3.2, §17.5.2).
[8] Use `unique_ptr` to reference objects of polymorphic type (§5.2.1).
[9] Use `shared_ptr` to reference shared objects, that is, objects without a single owner that is responsible for their destruction (§5.2.1).
[10] Use templates to maintain static type safety (eliminate casts) and avoid unnecessary use of class hierarchies (§27.2).

It might also be a good idea to review the advice for C and Java programmers (§1.3.3, §1.3.4).

1.3.3 Suggestions for C Programmers

The better one knows C, the harder it seems to be to avoid writing C++ in C style, thereby losing many of the potential benefits of C++. Please take a look at Chapter 44, which describes the differences between C and C++.

[1] Don't think of C++ as C with a few features added. C++ can be used that way, but only suboptimally. To get really major advantages from C++ as compared to C, you need to apply different design and implementation styles.

[2] Don't write C in C++; that is often seriously suboptimal for both maintenance and performance.

[3] Use the C++ standard library as a teacher of new techniques and programming styles. Note the difference from the C standard library (e.g., `=` rather than `strcpy()` for copying and `==` rather than `strcmp()` for comparing).

[4] Macro substitution is almost never necessary in C++. Use `const` (§7.5), `constexpr` (§2.2.3, §10.4), `enum` or `enum class` (§8.4) to define manifest constants, `inline` (§12.1.5) to avoid function-calling overhead, `template`s (§3.4, Chapter 23) to specify families of functions and types, and `namespace`s (§2.4.2, §14.3.1) to avoid name clashes.

[5] Don't declare a variable before you need it, and initialize it immediately. A declaration can occur anywhere a statement can (§9.3), in `for`-statement initializers (§9.5), and in conditions (§9.4.3).

[6] Don't use `malloc()`. The `new` operator (§11.2) does the same job better, and instead of `realloc()`, try a `vector` (§3.4.2). Don't just replace `malloc()` and `free()` with "naked" `new` and `delete` (§3.2.1.2, §11.2.1).

[7] Avoid `void*`, unions, and casts, except deep within the implementation of some function or class. Their use limits the support you can get from the type system and can harm performance. In most cases, a cast is an indication of a design error. If you must use an explicit type conversion, try using one of the named casts (e.g., `static_cast`; §11.5.2) for a more precise statement of what you are trying to do.

[8] Minimize the use of arrays and C-style strings. C++ standard-library `string`s (§4.2), `array`s (§8.2.4), and `vector`s (§4.4.1) can often be used to write simpler and more maintainable code compared to the traditional C style. In general, try not to build yourself what has already been provided by the standard library.

[9] Avoid pointer arithmetic except in very specialized code (such as a memory manager) and for simple array traversal (e.g., `++p`).

[10] Do not assume that something laboriously written in C style (avoiding C++ features such as classes, templates, and exceptions) is more efficient than a shorter alternative (e.g., using standard-library facilities). Often (but of course not always), the opposite is true.

To obey C linkage conventions, a C++ function must be declared to have C linkage (§15.2.5).

1.3.4 Suggestions for Java Programmers

C++ and Java are rather different languages with similar syntaxes. Their aims are significantly different and so are many of their application domains. Java is *not* a direct successor to C++ in the sense of a language that can do the same as its predecessor, but better and also more. To use C++ well, you need to adopt programming and design techniques appropriate to C++, rather than trying to write Java in C++. It is not just an issue of remembering to `delete` objects that you create with `new` because you can't rely on the presence of a garbage collector:

[1] Don't simply mimic Java style in C++; that is often seriously suboptimal for both maintainability and performance.

[2] Use the C++ abstraction mechanisms (e.g., classes and templates): don't fall back to a C style of programming out of a false feeling of familiarity.
[3] Use the C++ standard library as a teacher of new techniques and programming styles.
[4] Don't immediately invent a unique base for all of your classes (an `Object` class). Typically, you can do better without it for many/most classes.
[5] Minimize the use of reference and pointer variables: use local and member variables (§3.2.1.2, §5.2, §16.3.4, §17.1).
[6] Remember: a variable is never implicitly a reference.
[7] Think of pointers as C++'s equivalent to Java references (C++ references are more limited; there is no reseating of C++ references).
[8] A function is not `virtual` by default. Not every class is meant for inheritance.
[9] Use abstract classes as interfaces to class hierarchies; avoid "brittle base classes," that is, base classes with data members.
[10] Use scoped resource management ("Resource Acquisition Is Initialization"; RAII) whenever possible.
[11] Use a constructor to establish a class invariant (and throw an exception if it can't).
[12] If a cleanup action is needed when an object is deleted (e.g., goes out of scope), use a destructor for that. Don't imitate `finally` (doing so is more ad hoc and in the longer run far more work than relying on destructors).
[13] Avoid "naked" `new` and `delete`; instead, use containers (e.g., `vector`, `string`, and `map`) and handle classes (e.g., `lock` and `unique_ptr`).
[14] Use freestanding functions (nonmember functions) to minimize coupling (e.g., see the standard algorithms), and use namespaces (§2.4.2, Chapter 14) to limit the scope of freestanding functions.
[15] Don't use exception specifications (except `noexcept`; §13.5.1.1).
[16] A C++ nested class does not have access to an object of the enclosing class.
[17] C++ offers only the most minimal run-time reflection: `dynamic_cast` and `typeid` (Chapter 22). Rely more on compile-time facilities (e.g., compile-time polymorphism; Chapter 27, Chapter 28).

Most of this advice applies equally to C# programmers.

1.4 History

I invented C++, wrote its early definitions, and produced its first implementation. I chose and formulated the design criteria for C++, designed its major language features, developed or helped to develop many of the early libraries, and was responsible for the processing of extension proposals in the C++ standards committee.

C++ was designed to provide Simula's facilities for program organization [Dahl,1970] [Dahl,1972] together with C's efficiency and flexibility for systems programming [Kernighan,1978] [Kernighan,1988]. Simula is the initial source of C++'s abstraction mechanisms. The class concept (with derived classes and virtual functions) was borrowed from it. However, templates and exceptions came to C++ later with different sources of inspiration.

The evolution of C++ was always in the context of its use. I spent a lot of time listening to users and seeking out the opinions of experienced programmers. In particular, my colleagues at AT&T Bell Laboratories were essential for the growth of C++ during its first decade.

This section is a brief overview; it does not try to mention every language feature and library component. Furthermore, it does not go into details. For more information, and in particular for more names of people who contributed, see [Stroustrup,1993], [Stroustrup,2007], and [Stroustrup,1994]. My two papers from the ACM History of Programming Languages conference and my *Design and Evolution of C++* book (known as ‘‘D&E’’) describe the design and evolution of C++ in detail and document influences from other programming languages.

Most of the documents produced as part of the ISO C++ standards effort are available online [WG21]. In my FAQ, I try to maintain a connection between the standard facilities and the people who proposed and refined those facilities [Stroustrup,2010]. C++ is not the work of a faceless, anonymous committee or of a supposedly omnipotent ‘‘dictator for life’’; it is the work of many dedicated, experienced, hard-working individuals.

1.4.1 Timeline

The work that led to C++ started in the fall of 1979 under the name ‘‘C with Classes.’’ Here is a simplified timeline:

1979 Work on ‘‘C with Classes’’ started. The initial feature set included classes and derived classes, public/private access control, constructors and destructors, and function declarations with argument checking. The first library supported non-preemptive concurrent tasks and random number generators.

1984 ‘‘C with Classes’’ was renamed to C++. By then, C++ had acquired virtual functions, function and operator overloading, references, and the I/O stream and complex number libraries.

1985 First commercial release of C++ (October 14). The library included I/O streams, complex numbers, and tasks (nonpreemptive scheduling).

1985 *The C++ Programming Language* (‘‘TC++PL,’’ October 14) [Stroustrup,1986].

1989 *The Annotated C++ Reference Manual* (‘‘the ARM’’).

1991 *The C++ Programming Language, Second Edition* [Stroustrup,1991], presenting generic programming using templates and error handling based on exceptions (including the ‘‘Resource Acquisition Is Initialization’’ general resource management idiom).

1997 *The C++ Programming Language, Third Edition* [Stroustrup,1997] introduced ISO C++, including namespaces, **dynamic_cast**, and many refinements of templates. The standard library added the STL framework of generic containers and algorithms.

1998 ISO C++ standard.

2002 Work on a revised standard, colloquially named C++0x, started.

2003 A ‘‘bug fix’’ revision of the ISO C++ standard was issued. A C++ Technical Report introduced new standard-library components, such as regular expressions, unordered containers (hash tables), and resource management pointers, which later became part of C++0x.

2006 An ISO C++ Technical Report on Performance was issued to answer questions of cost, predictability, and techniques, mostly related to embedded systems programming.

2009 C++0x was feature complete. It provided uniform initialization, move semantics, variadic template arguments, lambda expressions, type aliases, a memory model suitable for concurrency, and much more. The standard library added several components, including threads, locks, and most of the components from the 2003 Technical Report.
2011 ISO C++11 standard was formally approved.
2012 The first complete C++11 implementations emerged.
2012 Work on future ISO C++ standards (referred to as C++14 and C++17) started.
2013 *The C++ Programming Language, Fourth Edition* introduced C++11.

During development, C++11 was known as C++0x. As is not uncommon in large projects, we were overly optimistic about the completion date.

1.4.2 The Early Years

I originally designed and implemented the language because I wanted to distribute the services of a UNIX kernel across multiprocessors and local-area networks (what are now known as multicores and clusters). For that, I needed some event-driven simulations for which Simula would have been ideal, except for performance considerations. I also needed to deal directly with hardware and provide high-performance concurrent programming mechanisms for which C would have been ideal, except for its weak support for modularity and type checking. The result of adding Simula-style classes to C, ‘‘C with Classes,’’ was used for major projects in which its facilities for writing programs that use minimal time and space were severely tested. It lacked operator overloading, references, virtual functions, templates, exceptions, and many, many details [Stroustrup,1982]. The first use of C++ outside a research organization started in July 1983.

The name C++ (pronounced ‘‘see plus plus’’) was coined by Rick Mascitti in the summer of 1983 and chosen as the replacement for ‘‘C with Classes’’ by me. The name signifies the evolutionary nature of the changes from C; ‘‘++’’ is the C increment operator. The slightly shorter name ‘‘C+’’ is a syntax error; it had also been used as the name of an unrelated language. Connoisseurs of C semantics find C++ inferior to ++C. The language was not called D, because it was an extension of C, because it did not attempt to remedy problems by removing features, and because there already existed several would-be C successors named D. For yet another interpretation of the name C++, see the appendix of [Orwell,1949].

C++ was designed primarily so that my friends and I would not have to program in assembler, C, or various then-fashionable high-level languages. Its main purpose was to make writing good programs easier and more pleasant for the individual programmer. In the early years, there was no C++ paper design; design, documentation, and implementation went on simultaneously. There was no ‘‘C++ project’’ either, or a ‘‘C++ design committee.’’ Throughout, C++ evolved to cope with problems encountered by users and as a result of discussions among my friends, my colleagues, and me.

1.4.2.1 Language Features and Library Facilities

The very first design of C++ (then called ‘‘C with Classes’’) included function declarations with argument type checking and implicit conversions, classes with the **public**/**private** distinction between the interface and the implementation, derived classes, and constructors and destructors. I used macros to provide primitive parameterization. This was in use by mid-1980. Late that year, I was able to present a set of language facilities supporting a coherent set of programming styles; see §1.2.1. In retrospect, I consider the introduction of constructors and destructors most significant. In the terminology of the time, ‘‘a constructor creates the execution environment for the member functions and the destructor reverses that.’’ Here is the root of C++'s strategies for resource management (causing a demand for exceptions) and the key to many techniques for making user code short and clear. If there were other languages at the time that supported multiple constructors capable of executing general code, I didn't (and don't) know of them. Destructors were new in C++.

C++ was released commercially in October 1985. By then, I had added inlining (§12.1.5, §16.2.8), **const**s (§2.2.3, §7.5, §16.2.9), function overloading (§12.3), references (§7.7), operator overloading (§3.2.1.1, Chapter 18, Chapter 19), and virtual functions (§3.2.3, §20.3.2). Of these features, support for run-time polymorphism in the form of virtual functions was by far the most controversial. I knew its worth from Simula but found it impossible to convince most people in the systems programming world of its value. Systems programmers tended to view indirect function calls with suspicion, and people acquainted with other languages supporting object-oriented programming had a hard time believing that **virtual** functions could be fast enough to be useful in systems code. Conversely, many programmers with an object-oriented background had (and many still have) a hard time getting used to the idea that you use virtual function calls only to express a choice that must be made at run time. The resistance to virtual functions may be related to a resistance to the idea that you can get better systems through more regular structure of code supported by a programming language. Many C programmers seem convinced that what really matters is complete flexibility and careful individual crafting of every detail of a program. My view was (and is) that we need every bit of help we can get from languages and tools: the inherent complexity of the systems we are trying to build is always at the edge of what we can express.

Much of the design of C++ was done on the blackboards of my colleagues. In the early years, the feedback from Stu Feldman, Alexander Fraser, Steve Johnson, Brian Kernighan, Doug McIlroy, and Dennis Ritchie was invaluable.

In the second half of the 1980s, I continued to add language features in response to user comments. The most important of those were templates [Stroustrup,1988] and exception handling [Koenig,1990], which were considered experimental at the time the standards effort started. In the design of templates, I was forced to decide among flexibility, efficiency, and early type checking. At the time, nobody knew how to simultaneously get all three, and to compete with C-style code for demanding systems applications, I felt that I had to choose the first two properties. In retrospect, I think the choice was the correct one, and the search for better type checking of templates continues [Gregor,2006] [Sutton,2011] [Stroustrup,2012a]. The design of exceptions focused on multilevel propagation of exceptions, the passing of arbitrary information to an error handler, and the integrations between exceptions and resource management by using local objects with destructors to represent and release resources (what I clumsily called ‘‘Resource Acquisition Is Initialization’’; §13.3).

I generalized C++'s inheritance mechanisms to support multiple base classes [Stroustrup,1987a]. This was called *multiple inheritance* and was considered difficult and controversial. I considered it far less important than templates or exceptions. Multiple inheritance of abstract classes (often called *interfaces*) is now universal in languages supporting static type checking and object-oriented programming.

The C++ language evolved hand in hand with some of the key library facilities presented in this book. For example, I designed the complex [Stroustrup,1984], vector, stack, and (I/O) stream [Stroustrup,1985] classes together with the operator overloading mechanisms. The first string and list classes were developed by Jonathan Shopiro and me as part of the same effort. Jonathan's string and list classes were the first to see extensive use as part of a library. The string class from the standard C++ library has its roots in these early efforts. The task library described in [Stroustrup,1987b] was part of the first ''C with Classes'' program ever written in 1980. I wrote it and its associated classes to support Simula-style simulations. Unfortunately, we had to wait until 2011 (30 years!) to get concurrency support standardized and universally available (§1.4.4.2, §5.3, Chapter 41). The development of the template facility was influenced by a variety of **vector**, **map**, **list**, and **sort** templates devised by Andrew Koenig, Alex Stepanov, me, and others.

C++ grew up in an environment with a multitude of established and experimental programming languages (e.g., Ada [Ichbiah,1979], Algol 68 [Woodward,1974], and ML [Paulson,1996]). At the time, I was comfortable in about 25 languages, and their influences on C++ are documented in [Stroustrup,1994] and [Stroustrup,2007]. However, the determining influences always came from the applications I encountered. That was a deliberate policy to have the development of C++ ''problem driven'' rather than imitative.

1.4.3 The 1998 Standard

The explosive growth of C++ use caused some changes. Sometime during 1987, it became clear that formal standardization of C++ was inevitable and that we needed to start preparing the ground for a standardization effort [Stroustrup,1994]. The result was a conscious effort to maintain contact between implementers of C++ compilers and major users. This was done through paper and electronic mail and through face-to-face meetings at C++ conferences and elsewhere.

AT&T Bell Labs made a major contribution to C++ and its wider community by allowing me to share drafts of revised versions of the C++ reference manual with implementers and users. Because many of those people worked for companies that could be seen as competing with AT&T, the significance of this contribution should not be underestimated. A less enlightened company could have caused major problems of language fragmentation simply by doing nothing. As it happened, about a hundred individuals from dozens of organizations read and commented on what became the generally accepted reference manual and the base document for the ANSI C++ standardization effort. Their names can be found in *The Annotated C++ Reference Manual* (''the ARM'') [Ellis,1989]. The X3J16 committee of ANSI was convened in December 1989 at the initiative of Hewlett-Packard. In June 1991, this ANSI (American national) standardization of C++ became part of an ISO (international) standardization effort for C++ and named WG21. From 1990, these joint C++ standards committees have been the main forum for the evolution of C++ and the refinement of its definition. I served on these committees throughout. In particular, as the chairman of the working group for extensions (later called the evolution group), I was directly responsible for handling proposals for major changes to C++ and the addition of new language features. An initial draft standard for public review was produced in April 1995. The first ISO C++ standard (ISO/IEC 14882-1998) [C++,1998] was ratified by a 22-0 national vote in 1998. A ''bug fix release'' of this standard was issued in 2003, so you sometimes hear people refer to C++03, but that is essentially the same language as C++98.

1.4.3.1 Language Features

By the time the ANSI and ISO standards efforts started, most major language features were in place and documented in the ARM [Ellis,1989]. Consequently, most of the work involved refinement of features and their specification. The template mechanisms, in particular, benefited from much detailed work. Namespaces were introduced to cope with the increased size of C++ programs and the increased number of libraries. At the initiative of Dmitry Lenkov from Hewett-Packard, minimal facilities to use run-time type information (RTTI; Chapter 22) were introduced. I had left such facilities out of C++ because I had found them seriously overused in Simula. I tried to get a facility for optional conservative garbage collection accepted, but failed. We had to wait until the 2011 standard for that.

Clearly, the 1998 language was far superior in features and in particular in the detail of specification to the 1989 language. However, not all changes were improvements. In addition to the inevitable minor mistakes, two major features were added that in retrospect should not have been:

- Exception specifications provide run-time enforcement of which exceptions a function is allowed to throw. They were added at the energetic initiative of people from Sun Microsystems. Exception specifications turned out to be worse than useless for improving readability, reliability, and performance. They are deprecated (scheduled for future removal) in the 2011 standard. The 2011 standard introduced **noexcept** (§13.5.1.1) as a simpler solution to many of the problems that exception specifications were supposed to address.
- It was always obvious that separate compilation of templates and their uses would be ideal [Stroustrup,1994]. How to achieve that under the constraints from real-world uses of templates was not at all obvious. After a long debate in the committee, a compromise was reached and something called **export** templates were specified as part of the 1998 standard. It was not an elegant solution to the problem, only one vendor implemented **export** (the Edison Design Group), and the feature was removed from the 2011 standard. We are still looking for a solution. My opinion is that the fundamental problem is not separate compilation in itself, but that the distinction between interface and implementation of a template is not well specified. Thus, **export** solved the wrong problem. In the future, language support for ‘‘concepts’’ (§24.3) may help by providing precise specification of template requirements. This is an area of active research and design [Sutton,2011] [Stroustrup,2012a].

1.4.3.2 The Standard Library

The greatest and most important innovation in the 1998 standard was the inclusion of the STL, a framework of algorithms and containers, in the standard library (§4.4, §4.5, Chapter 31, Chapter 32, Chapter 33). It was the work of Alex Stepanov (with Dave Musser, Meng Le, and others) based on more than a decade’s work on generic programming. Andrew Koenig, Beman Dawes, and I did much to help get the STL accepted [Stroustrup,2007]. The STL has been massively influential within the C++ community and beyond.

Except for the STL, the standard library was a bit of a hodgepodge of components, rather than a unified design. I had failed to ship a sufficiently large foundation library with Release 1.0 of C++ [Stroustrup,1993], and an unhelpful (non-research) AT&T manager had prevented my colleagues and me from rectifying that mistake for Release 2.0. That meant that every major organization (such as Borland, IBM, Microsoft, and Texas Instruments) had its own foundation library by the time the standards work started. Thus, the committee was limited to a patchwork of components based on what had always been available (e.g., the **complex** library), what could be added without interfering with the major vendor's libraries, and what was needed to ensure cooperation among different nonstandard libraries.

The standard-library **string** (§4.2, Chapter 36) had its origins in early work by Jonathan Shopiro and me at Bell Labs but was revised and extended by several different individuals and groups during standardization. The **valarray** library for numerical computation (§40.5) is primarily the work of Kent Budge. Jerry Schwarz transformed my streams library (§1.4.2.1) into the **iostream**s library (§4.3, Chapter 38) using Andrew Koenig's manipulator technique (§38.4.5.2) and other ideas. The **iostream**s library was further refined during standardization, where the bulk of the work was done by Jerry Schwarz, Nathan Myers, and Norihiro Kumagai.

By commercial standards the C++98 standard library is tiny. For example, there is no standard GUI, database access library, or Web application library. Such libraries are widely available but are not part of the ISO standard. The reasons for that are practical and commercial, rather than technical. However, the C standard library was (and is) many influential people's measure of a standard library, and compared to that, the C++ standard library is huge.

1.4.4 The 2011 Standard

The current C++, C++11, known for years as C++0x, is the work of the members of WG21. The committee worked under increasingly onerous self-imposed processes and procedures. These processes probably led to a better (and more rigorous) specification, but they also limited innovation [Stroustrup,2007]. An initial draft standard for public review was produced in 2009. The second ISO C++ standard (ISO/IEC 14882-2011) [C++,2011] was ratified by a 21-0 national vote in August 2011.

One reason for the long gap between the two standards is that most members of the committee (including me) were under the mistaken impression that the ISO rules required a ‘‘waiting period’’ after a standard was issued before starting work on new features. Consequently, serious work on new language features did not start until 2002. Other reasons included the increased size of modern languages and their foundation libraries. In terms of pages of standards text, the language grew by about 30% and the standard library by about 100%. Much of the increase was due to more detailed specification, rather than new functionality. Also, the work on a new C++ standard obviously had to take great care not to compromise older code through incompatible changes. There are billions of lines of C++ code in use that the committee must not break.

The overall aims for the C++11 effort were:

- Make C++ a better language for systems programming and library building.
- Make C++ easier to teach and learn.

The aims are documented and detailed in [Stroustrup,2007].

A major effort was made to make concurrent systems programming type-safe and portable. This involved a memory model (§41.2) and a set of facilities for lock-free programming (§41.3), which is primarily the work of Hans Boehm, Brian McKnight, and others. On top of that, we added the **thread**s library. Pete Becker, Peter Dimov, Howard Hinnant, William Kempf, Anthony Williams, and others did massive amounts of work on that. To provide an example of what can be achieved on top of the basic concurrency facilities, I proposed work on ‘‘a way to exchange information between tasks without explicit use of a lock,'' which became futures and async() (§5.3.5); Lawrence Crowl and Detlef Vollmann did most of the work on that. Concurrency is an area where a complete and detailed listing of who did what and why would require a very long paper. Here, I can't even try.

1.4.4.1 Language Features

The list of language features and standard-library facilities added to C++98 to get C++11 is presented in §44.2. With the exception of concurrency support, every addition to the language could be deemed ''minor,'' but doing so would miss the point: language features are meant to be used in combination to write better programs. By ''better'' I mean easier to read, easier to write, more elegant, less error-prone, more maintainable, faster-running, consuming fewer resources, etc.

Here are what I consider the most widely useful new ''building bricks'' affecting the style of C++11 code with references to the text and their primary authors:

- Control of defaults: =delete and =default: §3.3.4, §17.6.1, §17.6.4; Lawrence Crowl and Bjarne Stroustrup.
- Deducing the type of an object from its initializer, auto: §2.2.2, §6.3.6.1; Bjarne Stroustrup. I first designed and implemented auto in 1983 but had to remove it because of C compatibility problems.
- Generalized constant expression evaluation (including literal types), constexpr: §2.2.3, §10.4, §12.1.6; Gabriel Dos Reis and Bjarne Stroustrup [DosReis,2010].
- In-class member initializers: §17.4.4; Michael Spertus and Bill Seymour.
- Inheriting constructors: §20.3.5.1; Bjarne Stroustrup, Michael Wong, and Michel Michaud.
- Lambda expressions, a way of implicitly defining function objects at the point of their use in an expression: §3.4.3, §11.4; Jaakko Jarvi.
- Move semantics, a way of transmitting information without copying: §3.3.2, §17.5.2; Howard Hinnant.
- A way of stating that a function may not throw exceptions noexcept: §13.5.1.1; David Abrahams, Rani Sharoni, and Doug Gregor.
- A proper name for the null pointer, §7.2.2; Herb Sutter and Bjarne Stroustrup.
- The range-for statement: §2.2.5, §9.5.1; Thorsten Ottosen and Bjarne Stroustrup.
- Override controls: final and override: §20.3.4. Alisdair Meredith, Chris Uzdavinis, and Ville Voutilainen.
- Type aliases, a mechanism for providing an alias for a type or a template. In particular, a way of defining a template by binding some arguments of another template: §3.4.5, §23.6; Bjarne Stroustrup and Gabriel Dos Reis.
- Typed and scoped enumerations: enum class: §8.4.1; David E. Miller, Herb Sutter, and Bjarne Stroustrup.
- Universal and uniform initialization (including arbitrary-length initializer lists and protection against narrowing): §2.2.2, §3.2.1.3, §6.3.5, §17.3.1, §17.3.4; Bjarne Stroustrup and Gabriel Dos Reis.
- Variadic templates, a mechanism for passing an arbitrary number of arguments of arbitrary types to a template: §3.4.4, §28.6; Doug Gregor and Jaakko Jarvi.

Many more people than can be listed here deserve to be mentioned. The technical reports to the committee [WG21] and my C++11 FAQ [Stroustrup,2010a] give many of the names. The minutes of the committee's working groups mention more still. The reason my name appears so often is (I hope) not vanity, but simply that I chose to work on what I consider important. These are features that will be pervasive in good code. Their major role is to flesh out the C++ feature set to better support programming styles (§1.2.1). They are the foundation of the synthesis that is C++11.

Much work went into a proposal that did not make it into the standard. ''Concepts'' was a facility for specifying and checking requirements for template arguments [Gregor,2006] based on previous research (e.g., [Stroustrup,1994] [Siek,2000] [DosReis,2006]) and extensive work in the committee. It was designed, specified, implemented, and tested, but by a large majority the committee decided that the proposal was not yet ready. Had we been able to refine ''concepts,'' it would have been the most important single feature in C++11 (its only competitor for that title is concurrency support). However, the committee decided against ''concepts'' on the grounds of complexity, difficulty of use, and compile-time performance [Stroustrup,2010b]. I think we (the committee) did the right thing with ''concepts'' for C++11, but this feature really was ''the one that got away.'' This is currently a field of active research and design [Sutton,2011] [Stroustrup,2012a].

1.4.4.2 Standard Library

The work on what became the C++11 standard library started with a standards committee technical report (''TR1''). Initially, Matt Austern was the head of the Library Working Group, and later Howard Hinnant took over until we shipped the final draft standard in 2011.

As for language features, I'll only list a few standard-library components with references to the text and the names of the individuals most closely associated with them. For a more detailed list, see §44.2.2. Some components, such as **unordered_map** (hash tables), were ones we simply didn't manage to finish in time for the C++98 standard. Many others, such as **unique_ptr** and **function** were part of a technical report (TR1) based on Boost libraries. Boost is a volunteer organization created to provide useful library components based on the STL [Boost].

- Hashed containers, such as **unordered_map**: §31.4.3; Matt Austern.
- The basic concurrency library components, such as **thread**, **mutex**, and **lock**: §5.3, §42.2; Pete Becker, Peter Dimov, Howard Hinnant, William Kempf, Anthony Williams, and more.
- Launching asynchronous computation and returning results, **future**, **promise**, and **async()**: §5.3.5, §42.4.6; Detlef Vollmann, Lawrence Crowl, Bjarne Stroustrup, and Herb Sutter.
- The garbage collection interface: §34.5; Michael Spertus and Hans Boehm.
- A regular expression library, **regexp**: §5.5, Chapter 37; John Maddock.
- A random number library: §5.6.3, §40.7; Jens Maurer and Walter Brown. It was about time. I shipped the first random number library with ''C with Classes'' in 1980.

Several utility components were tried out in Boost:

- A pointer for simply and efficiently passing resources, **unique_ptr**: §5.2.1, §34.3.1; Howard E. Hinnant. This was originally called **move_ptr** and is what **auto_ptr** should have been had we known how to do so for C++98.
- A pointer for representing shared ownership, **shared_ptr**: §5.2.1, §34.3.2; Peter Dimov. A successor to the C++98 **counted_ptr** proposal from Greg Colvin.

- The tuple library: §5.4.3, §28.5, §34.2.4.2; Jaakko Jarvi and Gary Powell. They credit a long list of contributors, including Doug Gregor, David Abrahams, and Jeremy Siek.
- The general bind(): §33.5.1; Peter Dimov. His acknowledgments list a veritable who's who of Boost (including Doug Gregor, John Maddock, Dave Abrahams, and Jaakko Jarvi).
- The function type for holding callable objects: §33.5.3; Doug Gregor. He credits William Kempf and others with contributions.

1.4.5 What is C++ used for?

By now (2013), C++ is used just about everywhere: it is in your computer, your phone, your car, probably even in your camera. You don't usually see it. C++ is a systems programming language, and its most pervasive uses are deep in the infrastructure where we, as users, never look.

C++ is used by millions of programmers in essentially every application domain. Billions (thousands of millions) of lines of C++ are currently deployed. This massive use is supported by half a dozen independent implementations, many thousands of libraries, hundreds of textbooks, and dozens of websites. Training and education at a variety of levels are widely available.

Early applications tended to have a strong systems programming flavor. For example, several early operating systems have been written in C++: [Campbell,1987] (academic), [Rozier,1988] (real time), [Berg,1995] (high-throughput I/O). Many current ones (e.g., Windows, Apple's OS, Linux, and most portable-device OSs) have key parts done in C++. Your cellphone and Internet routers are most likely written in C++. I consider uncompromising low-level efficiency essential for C++. This allows us to use C++ to write device drivers and other software that rely on direct manipulation of hardware under real-time constraints. In such code, predictability of performance is at least as important as raw speed. Often, so is the compactness of the resulting system. C++ was designed so that every language feature is usable in code under severe time and space constraints (§1.2.4) [Stroustrup,1994,§4.5].

Some of today's most visible and widely used systems have their critical parts written in C++. Examples are Amadeus (airline ticketing), Amazon (Web commerce), Bloomberg (financial information), Google (Web search), and Facebook (social media). Many other programming languages and technologies depend critically on C++'s performance and reliability in their implementation. Examples include the most widely used Java Virtual Machines (e.g., Oracle's HotSpot), JavaScript interpreters (e.g., Google's V8), browsers (e.g., Microsoft's Internet Explorer, Mozilla's Firefox, Apple's Safari, and Google's Chrome), and application frameworks (e.g., Microsoft's .NET Web services framework). I consider C++ to have unique strengths in the area of infrastructure software [Stroustrup,2012a].

Most applications have sections of code that are critical for acceptable performance. However, the largest amount of code is not in such sections. For most code, maintainability, ease of extension, and ease of testing are key. C++'s support for these concerns has led to its widespread use in areas where reliability is a must and where requirements change significantly over time. Examples are financial systems, telecommunications, device control, and military applications. For decades, the central control of the U.S. long-distance telephone system has relied on C++, and every 800 call (i.e., a call paid for by the called party) has been routed by a C++ program [Kamath,1993]. Many such applications are large and long-lived. As a result, stability, compatibility, and scalability have been constant concerns in the development of C++. Multimillion-line C++ programs are common.

Games is another area where a multiplicity of languages and tools need to coexist with a language providing uncompromising efficiency (often on ‘‘unusual’’ hardware). Thus, games has been another major applications area for C++.

What used to be called systems programming is widely found in embedded systems, so it is not surprising to find massive use of C++ in demanding embedded systems projects, including computer tomography (CAT scanners), flight control software (e.g., Lockheed-Martin), rocket control, ship’s engines (e.g., the control of the world’s largest marine diesel engines from MAN), automobile software (e.g., BMW), and wind turbine control (e.g., Vesta).

C++ wasn’t specifically designed with numerical computation in mind. However, much numerical, scientific, and engineering computation is done in C++. A major reason for this is that traditional numerical work must often be combined with graphics and with computations relying on data structures that don’t fit into the traditional Fortran mold (e.g., [Root,1995]). I am particularly pleased to see C++ used in major scientific endeavors, such as the Human Genome Project, NASA’s Mars Rovers, CERN’s search for the fundamentals of the universe, and many others.

C++’s ability to be used effectively for applications that require work in a variety of application areas is an important strength. Applications that involve local- and wide-area networking, numerics, graphics, user interaction, and database access are common. Traditionally, such application areas were considered distinct and were served by distinct technical communities using a variety of programming languages. However, C++ is widely used in all of those areas, and more. It is designed so that C++ code can coexist with code written in other languages. Here, again, C++’s stability over decades is important. Furthermore, no really major system is written 100% in a single language. Thus, C++’s original design aim of interoperability becomes significant.

Major applications are not written in just the raw language. C++ is supported by a variety of libraries (beyond the ISO C++ standard library) and tool sets, such as Boost [Boost] (portable foundation libraries), POCO (Web development), QT (cross-platform application development), wxWidgets (a cross-platform GUI library), WebKit (a layout engine library for Web browsers), CGAL (computational geometry), QuickFix (Financial Information eXchange), OpenCV (real-time image processing), and Root [Root,1995] (High-Energy Physics). There are many thousands of C++ libraries, so keeping up with them all is impossible.

1.5 Advice

Each chapter contains an ‘‘Advice’’ section with a set of concrete recommendations related to its contents. Such advice consists of rough rules of thumb, not immutable laws. A piece of advice should be applied only where reasonable. There is no substitute for intelligence, experience, common sense, and good taste.

I find rules of the form ‘‘never do this’’ unhelpful. Consequently, most advice is phrased as suggestions for what to do. Negative suggestions tend not to be phrased as absolute prohibitions and I try to suggest alternatives. I know of no major feature of C++ that I have not seen put to good use. The ‘‘Advice’’ sections do not contain explanations. Instead, each piece of advice is accompanied by a reference to an appropriate section of the book.

For starters, here are a few high-level recommendations derived from the sections on design, learning, and history of C++:

[1] Represent ideas (concepts) directly in code, for example, as a function, a class, or an enumeration; §1.2.
[2] Aim for your code to be both elegant and efficient; §1.2.
[3] Don't overabstract; §1.2.
[4] Focus design on the provision of elegant and efficient abstractions, possibly presented as libraries; §1.2.
[5] Represent relationships among ideas directly in code, for example, through parameterization or a class hierarchy; §1.2.1.
[6] Represent independent ideas separately in code, for example, avoid mutual dependencies among classes; §1.2.1.
[7] C++ is not just object-oriented; §1.2.1.
[8] C++ is not just for generic programming; §1.2.1.
[9] Prefer solutions that can be statically checked; §1.2.1.
[10] Make resources explicit (represent them as class objects); §1.2.1, §1.4.2.1.
[11] Express simple ideas simply; §1.2.1.
[12] Use libraries, especially the standard library, rather than trying to build everything from scratch; §1.2.1.
[13] Use a type-rich style of programming; §1.2.2.
[14] Low-level code is not necessarily efficient; don't avoid classes, templates, and standard-library components out of fear of performance problems; §1.2.4, §1.3.3.
[15] If data has an invariant, encapsulate it; §1.3.2.
[16] C++ is not just C with a few extensions; §1.3.3.

In general: To write a good program takes intelligence, taste, and patience. You are not going to get it right the first time. Experiment!

1.6 References

[Austern,2003] Matt Austern et al.: *Untangling the Balancing and Searching of Balanced Binary Search Trees*. Software – Practice & Experience. Vol 33, Issue 13. November 2003.

[Barron,1963] D. W. Barron et al.: *The main features of CPL*. The Computer Journal. 6 (2): 134. (1963). comjnl.oxfordjournals.org/content/6/2/134.full.pdf+html.

[Barton,1994] J. J. Barton and L. R. Nackman: *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. Addison-Wesley. Reading, Massachusetts. 1994. ISBN 0-201-53393-6.

[Berg,1995] William Berg, Marshall Cline, and Mike Girou: *Lessons Learned from the OS/400 OO Project*. CACM. Vol. 38, No. 10. October 1995.

[Boehm,2008] Hans-J. Boehm and Sarita V. Adve: *Foundations of the C++ concurrency memory model*. ACM PLDI'08.

[Boost] The Boost library collection. www.boost.org.

[Budge,1992] Kent Budge, J. S. Perry, and A. C. Robinson: *High-Performance Scientific Computation Using C++*. Proc. USENIX C++ Conference. Portland, Oregon. August 1992.

[C,1990]	X3 Secretariat: *Standard – The C Language*. X3J11/90-013. ISO Standard ISO/IEC 9899-1990. Computer and Business Equipment Manufacturers Association. Washington, DC.
[C,1999]	ISO/IEC 9899. *Standard – The C Language*. X3J11/90-013-1999.
[C,2011]	ISO/IEC 9899. *Standard – The C Language*. X3J11/90-013-2011.
[C++,1998]	ISO/IEC JTC1/SC22/WG21: *International Standard – The C++ Language*. ISO/IEC 14882:1998.
[C++Math,2010]	*International Standard – Extensions to the C++ Library to Support Mathematical Special Functions*. ISO/IEC 29124:2010.
[C++,2011]	ISO/IEC JTC1/SC22/WG21: *International Standard – The C++ Language*. ISO/IEC 14882:2011.
[Campbell,1987]	Roy Campbell et al.: *The Design of a Multiprocessor Operating System*. Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.
[Coplien,1995]	James O. Coplien: *Curiously Recurring Template Patterns*. The C++ Report. February 1995.
[Cox,2007]	Russ Cox: *Regular Expression Matching Can Be Simple And Fast*. January 2007. swtch.com/˜rsc/regexp/regexp1.html.
[Czarnecki,2000]	K. Czarnecki and U. Eisenecker: *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley. Reading, Massachusetts. 2000. ISBN 0-201-30977-7.
[Dahl,1970]	O-J. Dahl, B. Myrhaug, and K. Nygaard: *SIMULA Common Base Language.* Norwegian Computing Center S-22. Oslo, Norway. 1970.
[Dahl,1972]	O-J. Dahl and C. A. R. Hoare: *Hierarchical Program Construction* in *Structured Programming.* Academic Press. New York. 1972.
[Dean,2004]	J. Dean and S. Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters*. OSDI'04: Sixth Symposium on Operating System Design and Implementation. 2004.
[Dechev,2010]	D. Dechev, P. Pirkelbauer, and B. Stroustrup: *Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs*. 13th IEEE Computer Society ISORC 2010 Symposium. May 2010.
[DosReis,2006]	Gabriel Dos Reis and Bjarne Stroustrup: *Specifying C++ Concepts*. POPL06. January 2006.
[DosReis,2010]	Gabriel Dos Reis and Bjarne Stroustrup: *General Constant Expressions for System Programming Languages*. SAC-2010. The 25th ACM Symposium On Applied Computing. March 2010.
[DosReis,2011]	Gabriel Dos Reis and Bjarne Stroustrup: *A Principled, Complete, and Efficient Representation of C++*. Journal of Mathematics in Computer Science. Vol. 5, Issue 3. 2011.
[Ellis,1989]	Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley. Reading, Mass. 1990. ISBN 0-201-51459-1.
[Freeman,1992]	Len Freeman and Chris Phillips: *Parallel Numerical Algorithms*. Prentice Hall. Englewood Cliffs, New Jersey. 1992. ISBN 0-13-651597-5.
[Friedl,1997]:	Jeffrey E. F. Friedl: *Mastering Regular Expressions*. O'Reilly Media. Sebastopol, California. 1997. ISBN 978-1565922570.

[Gamma,1995] Erich Gamma et al.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, Massachusetts. 1994. ISBN 0-201-63361-2.

[Gregor,2006] Douglas Gregor et al.: *Concepts: Linguistic Support for Generic Programming in C++*. OOPSLA'06.

[Hennessy,2011] John L. Hennessy and David A. Patterson: *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann. San Francisco, California. 2011. ISBN 978-0123838728.

[Ichbiah,1979] Jean D. Ichbiah et al.: *Rationale for the Design of the ADA Programming Language*. SIGPLAN Notices. Vol. 14, No. 6. June 1979.

[Kamath,1993] Yogeesh H. Kamath, Ruth E. Smilan, and Jean G. Smith: *Reaping Benefits with Object-Oriented Technology*. AT&T Technical Journal. Vol. 72, No. 5. September/October 1993.

[Kernighan,1978] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language.* Prentice Hall. Englewood Cliffs, New Jersey. 1978.

[Kernighan,1988] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language, Second Edition.* Prentice-Hall. Englewood Cliffs, New Jersey. 1988. ISBN 0-13-110362-8.

[Knuth,1968] Donald E. Knuth: *The Art of Computer Programming*. Addison-Wesley. Reading, Massachusetts. 1968.

[Koenig,1989] Andrew Koenig and Bjarne Stroustrup: *C++: As close to C as possible – but no closer*. The C++ Report. Vol. 1, No. 7. July 1989.

[Koenig,1990] A. R. Koenig and B. Stroustrup: *Exception Handling for C++ (revised)*. Proc USENIX C++ Conference. April 1990.

[Kolecki,2002] Joseph C. Kolecki: *An Introduction to Tensors for Students of Physics and Engineering*. NASA/TM-2002-211716.

[Langer,2000] Angelika Langer and Klaus Kreft: *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference*. Addison-Wesley. 2000. ISBN 978-0201183955.

[McKenney] Paul E. McKenney: *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org. Corvallis, Oregon. 2012.
http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html.

[Maddock,2009] John Maddock: *Boost.Regex*. www.boost.org. 2009.

[Orwell,1949] George Orwell: *1984.* Secker and Warburg. London. 1949.

[Paulson,1996] Larry C. Paulson: *ML for the Working Programmer*. Cambridge University Press. Cambridge. 1996. ISBN 0-521-56543-X.

[Pirkelbauer,2009] P. Pirkelbauer, Y. Solodkyy, and B. Stroustrup: *Design and Evaluation of C++ Open Multi-Methods*. Science of Computer Programming. Elsevier Journal. June 2009. doi:10.1016/j.scico.2009.06.002.

[Richards,1980] Martin Richards and Colin Whitby-Strevens: *BCPL – The Language and Its Compiler.* Cambridge University Press. Cambridge. 1980. ISBN 0-521-21965-5.

[Root,1995] *ROOT: A Data Analysis Framework*. root.cern.ch. It seems appropriate to represent a tool from CERN, the birthplace of the World Wide Web, by a

Web address.

[Rozier,1988] M. Rozier et al.: *CHORUS Distributed Operating Systems*. Computing Systems. Vol. 1, No. 4. Fall 1988.

[Siek,2000] Jeremy G. Siek and Andrew Lumsdaine: *Concept checking: Binding parametric polymorphism in C++*. Proc. First Workshop on C++ Template Programming. Erfurt, Germany. 2000.

[Solodkyy,2012] Y. Solodkyy, G. Dos Reis, and B. Stroustrup: *Open and Efficient Type Switch for C++*. Proc. OOPSLA'12.

[Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library*. HP Labs Technical Report HPL-94-34 (R. 1). 1994.

[Stewart,1998] G. W. Stewart: *Matrix Algorithms, Volume I. Basic Decompositions*. SIAM. Philadelphia, Pennsylvania. 1998.

[Stroustrup,1982] B. Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. Sigplan Notices. January 1982. The first public description of "C with Classes."

[Stroustrup,1984] B. Stroustrup: *Operator Overloading in C++*. Proc. IFIP WG2.4 Conference on System Implementation Languages: Experience & Assessment. September 1984.

[Stroustrup,1985] B. Stroustrup: *An Extensible I/O Facility for C++*. Proc. Summer 1985 USENIX Conference.

[Stroustrup,1986] B. Stroustrup: *The C++ Programming Language*. Addison-Wesley. Reading, Massachusetts. 1986. ISBN 0-201-12078-X.

[Stroustrup,1987] B. Stroustrup: *Multiple Inheritance for C++*. Proc. EUUG Spring Conference. May 1987.

[Stroustrup,1987b] B. Stroustrup and J. Shopiro: *A Set of C Classes for Co-Routine Style Programming*. Proc. USENIX C++ Conference. Santa Fe, New Mexico. November 1987.

[Stroustrup,1988] B. Stroustrup: *Parameterized Types for C++*. Proc. USENIX C++ Conference, Denver. 1988.

[Stroustrup,1991] B. Stroustrup: *The C++ Programming Language (Second Edition)*. Addison-Wesley. Reading, Massachusetts. 1991. ISBN 0-201-53992-6.

[Stroustrup,1993] B. Stroustrup: *A History of C++: 1979-1991*. Proc. ACM History of Programming Languages conference (HOPL-2). ACM Sigplan Notices. Vol 28, No 3. 1993.

[Stroustrup,1994] B. Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. Reading, Mass. 1994. ISBN 0-201-54330-3.

[Stroustrup,1997] B. Stroustrup: *The C++ Programming Language, Third Edition*. Addison-Wesley. Reading, Massachusetts. 1997. ISBN 0-201-88954-4. Hardcover ("Special") Edition. 2000. ISBN 0-201-70073-5.

[Stroustrup,2002] B. Stroustrup: *C and C++: Siblings*, *C and C++: A Case for Compatibility*, and *C and C++: Case Studies in Compatibility*. The C/C++ Users Journal. July-September 2002. www.stroustrup.com/papers.html.

[Stroustrup,2007] B. Stroustrup: *Evolving a language in and for the real world: C++ 1991-2006*. ACM HOPL-III. June 2007.

[Stroustrup,2008] B. Stroustrup: *Programming – Principles and Practice Using C++*. Addison-Wesley. 2009. ISBN 0-321-54372-6.

[Stroustrup,2010a] B. Stroustrup: *The C++11 FAQ*. www.stroustrup.com/C++11FAQ.html.

[Stroustrup,2010b] B. Stroustrup: *The C++0x "Remove Concepts" Decision*. Dr. Dobb's Journal. July 2009.

[Stroustrup,2012a] B. Stroustrup and A. Sutton: *A Concept Design for the STL*. WG21 Technical Report N3351==12-0041. January 2012.

[Stroustrup,2012b] B. Stroustrup: *Software Development for Infrastructure*. Computer. January 2012. doi:10.1109/MC.2011.353.

[Sutton,2011] A. Sutton and B. Stroustrup: *Design of Concept Libraries for C++*. Proc. SLE 2011 (International Conference on Software Language Engineering). July 2011.

[Tanenbaum,2007] Andrew S. Tanenbaum: *Modern Operating Systems, Third Edition*. Prentice Hall. Upper Saddle River, New Jersey. 2007. ISBN 0-13-600663-9.

[Tsafrir,2009] Dan Tsafrir et al.: *Minimizing Dependencies within Generic Classes for Faster and Smaller Programs*. ACM OOPSLA'09. October 2009.

[Unicode,1996] The Unicode Consortium: *The Unicode Standard, Version 2.0*. Addison-Wesley. Reading, Massachusetts. 1996. ISBN 0-201-48345-9.

[UNIX,1985] *UNIX Time-Sharing System: Programmer's Manual. Research Version, Tenth Edition.* AT&T Bell Laboratories, Murray Hill, New Jersey. February 1985.

[Vandevoorde,2002] David Vandevoorde and Nicolai M. Josuttis: *C++ Templates: The Complete Guide*. Addison-Wesley. 2002. ISBN 0-201-73484-2.

[Veldhuizen,1995] Todd Veldhuizen: *Expression Templates*. The C++ Report. June 1995.

[Veldhuizen,2003] Todd L. Veldhuizen: *C++ Templates are Turing Complete*. Indiana University Computer Science Technical Report. 2003.

[Vitter,1985] Jefferey Scott Vitter: *Random Sampling with a Reservoir*. ACM Transactions on Mathematical Software, Vol. 11, No. 1. 1985.

[WG21] ISO SC22/WG21 The C++ Programming Language Standards Committee: *Document Archive*. www.open-std.org/jtc1/sc22/wg21.

[Williams,2012] Anthony Williams: *C++ Concurrency in Action – Practical Multithreading*. Manning Publications Co. ISBN 978-1933988771.

[Wilson,1996] Gregory V. Wilson and Paul Lu (editors): *Parallel Programming Using C++*. The MIT Press. Cambridge, Mass. 1996. ISBN 0-262-73118-5.

[Wood,1999] Alistair Wood: *Introduction to Numerical Analysis*. Addison-Wesley. Reading, Massachusetts. 1999. ISBN 0-201-34291-X.

[Woodward,1974] P. M. Woodward and S. G. Bond: *Algol 68-R Users Guide.* Her Majesty's Stationery Office. London. 1974.

2

A Tour of C++: The Basics

The first thing we do, let's
kill all the language lawyers.
– Henry VI, Part II

2.1 Introduction

The aim of this chapter and the next three is to give you an idea of what C++ is, without going into a lot of details. This chapter informally presents the notation of C++, C++'s model of memory and computation, and the basic mechanisms for organizing code into a program. These are the language facilities supporting the styles most often seen in C and sometimes called *procedural programming*. Chapter 3 follows up by presenting C++'s abstraction mechanisms. Chapter 4 and Chapter 5 give examples of standard-library facilities.

The assumption is that you have programmed before. If not, please consider reading a textbook, such as *Programming: Principles and Practice Using C++* [Stroustrup,2009], before continuing here. Even if you have programmed before, the language you used or the applications you wrote may be very different from the style of C++ presented here. If you find this ''lightning tour'' confusing, skip to the more systematic presentation starting in Chapter 6.

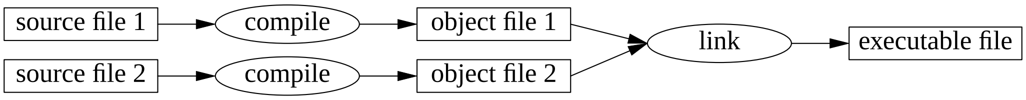
This tour of C++ saves us from a strictly bottom-up presentation of language and library facilities by enabling the use of a rich set of facilities even in early chapters. For example, loops are not discussed in detail until Chapter 10, but they will be used in obvious ways long before that. Similarly, the detailed description of classes, templates, free-store use, and the standard library are spread over many chapters, but standard-library types, such as **vector**, **string**, **complex**, **map**, **unique_ptr**, and **ostream**, are used freely where needed to improve code examples.

As an analogy, think of a short sightseeing tour of a city, such as Copenhagen or New York. In just a few hours, you are given a quick peek at the major attractions, told a few background stories, and usually given some suggestions about what to see next. You do *not* know the city after such a tour. You do *not* understand all you have seen and heard. To really know a city, you have to live in it, often for years. However, with a bit of luck, you will have gained a bit of an overview, a notion of what is special about the city, and ideas of what might be of interest to you. After the tour, the real exploration can begin.

This tour presents C++ as an integrated whole, rather than as a layer cake. Consequently, it does not identify language features as present in C, part of C++98, or new in C++11. Such historical information can be found in §1.4 and Chapter 44.

2.2 The Basics

C++ is a compiled language. For a program to run, its source text has to be processed by a compiler, producing object files, which are combined by a linker yielding an executable program. A C++ program typically consists of many source code files (usually simply called *source files*).



An executable program is created for a specific hardware/system combination; it is not portable, say, from a Mac to a Windows PC. When we talk about portability of C++ programs, we usually mean portability of source code; that is, the source code can be successfully compiled and run on a variety of systems.

The ISO C++ standard defines two kinds of entities:

- *Core language features*, such as built-in types (e.g., **char** and **int**) and loops (e.g., **for**-statements and **while**-statements)
- *Standard-library components*, such as containers (e.g., **vector** and **map**) and I/O operations (e.g., **<<** and **getline()**)

The standard-library components are perfectly ordinary C++ code provided by every C++ implementation. That is, the C++ standard library can be implemented in C++ itself (and is with very minor uses of machine code for things such as thread context switching). This implies that C++ is sufficiently expressive and efficient for the most demanding systems programming tasks.

C++ is a statically typed language. That is, the type of every entity (e.g., object, value, name, and expression) must be known to the compiler at its point of use. The type of an object determines the set of operations applicable to it.

2.2.1 Hello, World!

The minimal C++ program is

```
int main() { }          // the minimal C++ program
```

This defines a function called **main**, which takes no arguments and does nothing (§15.4).

Curly braces, **{ }**, express grouping in C++. Here, they indicate the start and end of the function body. The double slash, **//**, begins a comment that extends to the end of the line. A comment is for the human reader; the compiler ignores comments.

Every C++ program must have exactly one global function named **main()**. The program starts by executing that function. The **int** value returned by **main()**, if any, is the program's return value to "the system." If no value is returned, the system will receive a value indicating successful completion. A nonzero value from **main()** indicates failure. Not every operating system and execution environment make use of that return value: Linux/Unix-based environments often do, but Windows-based environments rarely do.

Typically, a program produces some output. Here is a program that writes **Hello, World!**:

```
#include <iostream>

int main()
{
        std::cout << "Hello, World!\n";
}
```

The line **#include <iostream>** instructs the compiler to *include* the declarations of the standard stream I/O facilities as found in **iostream**. Without these declarations, the expression

```
std::cout << "Hello, World!\n"
```

would make no sense. The operator **<<** ("put to") writes its second argument onto its first. In this case, the string literal **"Hello, World!\n"** is written onto the standard output stream **std::cout**. A string literal is a sequence of characters surrounded by double quotes. In a string literal, the backslash character **** followed by another character denotes a single "special character." In this case, **\n** is the newline character, so that the characters written are **Hello, World!** followed by a newline.

The **std::** specifies that the name **cout** is to be found in the standard-library namespace (§2.4.2, Chapter 14). I usually leave out the **std::** when discussing standard features; §2.4.2 shows how to make names from a namespace visible without explicit qualification.

Essentially all executable code is placed in functions and called directly or indirectly from **main()**. For example:

```
#include <iostream>
using namespace std;            // make names from std visible without std:: (§2.4.2)

double square(double x)         // square a double precision floating-point number
{
        return x∗x;
}
```

```
void print_square(double x)
{
        cout << "the square of " << x << " is " << square(x) << "\n";
}

int main()
{
        print_square(1.234);        // print: the square of 1.234 is 1.52276
}
```

A ‘‘return type’’ `void` indicates that a function does not return a value.

2.2.2 Types, Variables, and Arithmetic

Every name and every expression has a type that determines the operations that may be performed on it. For example, the declaration

```
int inch;
```

specifies that `inch` is of type `int`; that is, `inch` is an integer variable.

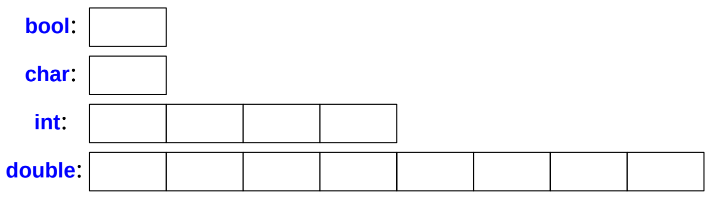
A *declaration* is a statement that introduces a name into the program. It specifies a type for the named entity:

- A *type* defines a set of possible values and a set of operations (for an object).
- An *object* is some memory that holds a value of some type.
- A *value* is a set of bits interpreted according to a type.
- A *variable* is a named object.

C++ offers a variety of fundamental types. For example:

```
bool        // Boolean, possible values are true and false
char        // character, for example, 'a', ' z', and '9'
int         // integer, for example, 1, 42, and 1066
double      // double-precision floating-point number, for example, 3.14 and 299793.0
```

Each fundamental type corresponds directly to hardware facilities and has a fixed size that determines the range of values that can be stored in it:



A `char` variable is of the natural size to hold a character on a given machine (typically an 8-bit byte), and the sizes of other types are quoted in multiples of the size of a `char`. The size of a type is implementation-defined (i.e., it can vary among different machines) and can be obtained by the `sizeof` operator; for example, `sizeof(char)` equals `1` and `sizeof(int)` is often `4`.

The arithmetic operators can be used for appropriate combinations of these types:

```
x+y        // plus
+x         // unary plus
x−y        // minus
−x         // unary minus
x∗y        // multiply
x/y        // divide
x%y        // remainder (modulus) for integers
```

So can the comparison operators:

```
x==y       // equal
x!=y       // not equal
x<y        // less than
x>y        // greater than
x<=y       // less than or equal
x>=y       // greater than or equal
```

In assignments and in arithmetic operations, C++ performs all meaningful conversions (§10.5.3) between the basic types so that they can be mixed freely:

```
void some_function()      // function that doesn’t return a value
{
      double d = 2.2;     // initialize floating-point number
      int i = 7;          // initialize integer
      d = d+i;            // assign sum to d
      i = d∗i;            // assign product to i (truncating the double d*i to an int)
}
```

Note that = is the assignment operator and == tests equality.

C++ offers a variety of notations for expressing initialization, such as the = used above, and a universal form based on curly-brace-delimited initializer lists:

```
double d1 = 2.3;
double d2 {2.3};

complex<double> z = 1;          // a complex number with double-precision floating-point scalars
complex<double> z2 {d1,d2};
complex<double> z3 = {1,2};     // the = is optional with { ... }

vector<int> v {1,2,3,4,5,6};    // a vector of ints
```

The = form is traditional and dates back to C, but if in doubt, use the general {}-list form (§6.3.5.2). If nothing else, it saves you from conversions that lose information (narrowing conversions; §10.5):

```
int i1 = 7.2;        // i1 becomes 7
int i2 {7.2};        // error: floating-point to integer conversion
int i3 = {7.2};      // error: floating-point to integer conversion (the = is redundant)
```

A constant (§2.2.3) cannot be left uninitialized and a variable should only be left uninitialized in extremely rare circumstances. Don’t introduce a name until you have a suitable value for it. User-defined types (such as string, vector, Matrix, Motor_controller, and Orc_warrior) can be defined to be implicitly initialized (§3.2.1.1).

When defining a variable, you don't actually need to state its type explicitly when it can be deduced from the initializer:

```
auto b = true;          // a bool
auto ch = 'x';          // a char
auto i = 123;           // an int
auto d = 1.2;           // a double
auto z = sqrt(y);       // z has the type of whatever sqrt(y) returns
```

With **auto**, we use the **=** syntax because there is no type conversion involved that might cause problems (§6.3.6.2).

We use **auto** where we don't have a specific reason to mention the type explicitly. ‘‘Specific reasons’’ include:

- The definition is in a large scope where we want to make the type clearly visible to readers of our code.
- We want to be explicit about a variable's range or precision (e.g., **double** rather than **float**).

Using **auto**, we avoid redundancy and writing long type names. This is especially important in generic programming where the exact type of an object can be hard for the programmer to know and the type names can be quite long (§4.5.1).

In addition to the conventional arithmetic and logical operators (§10.3), C++ offers more specific operations for modifying a variable:

```
x+=y      // x = x+y
++x       // increment: x = x+1
x−=y      // x = x-y
−−x       // decrement: x = x-1
x∗=y      // scaling: x = x*y
x/=y      // scaling: x = x/y
x%=y      // x = x%y
```

These operators are concise, convenient, and very frequently used.

2.2.3 Constants

C++ supports two notions of immutability (§7.5):

- **const**: meaning roughly ‘‘I promise not to change this value’’ (§7.5). This is used primarily to specify interfaces, so that data can be passed to functions without fear of it being modified. The compiler enforces the promise made by **const**.
- **constexpr**: meaning roughly ‘‘to be evaluated at compile time’’ (§10.4). This is used primarily to specify constants, to allow placement of data in memory where it is unlikely to be corrupted, and for performance.

For example:

```
const int dmv = 17;                          // dmv is a named constant
int var = 17;                                // var is not a constant
constexpr double max1 = 1.4∗square(dmv);     // OK if square(17) is a constant expression
constexpr double max2 = 1.4∗square(var);     // error: var is not a constant expression
const double max3 = 1.4∗square(var);         // OK, may be evaluated at run time
```

```
double sum(const vector<double>&);          // sum will not modify its argument (§2.2.5)
vector<double> v {1.2, 3.4, 4.5};           // v is not a constant
const double s1 = sum(v);                   // OK: evaluated at run time
constexpr double s2 = sum(v);               // error: sum(v) not constant expression
```

For a function to be usable in a *constant expression*, that is, in an expression that will be evaluated by the compiler, it must be defined **constexpr**. For example:

```
constexpr double square(double x) { return x*x; }
```

To be **constexpr**, a function must be rather simple: just a **return**-statement computing a value. A **constexpr** function can be used for non-constant arguments, but when that is done the result is not a constant expression. We allow a **constexpr** function to be called with non-constant-expression arguments in contexts that do not require constant expressions, so that we don't have to define essentially the same function twice: once for constant expressions and once for variables.

In a few places, constant expressions are required by language rules (e.g., array bounds (§2.2.5, §7.3), case labels (§2.2.4, §9.4.2), some template arguments (§25.2), and constants declared using **constexpr**). In other cases, compile-time evaluation is important for performance. Independently of performance issues, the notion of immutability (of an object with an unchangeable state) is an important design concern (§10.4).

2.2.4 Tests and Loops

C++ provides a conventional set of statements for expressing selection and looping. For example, here is a simple function that prompts the user and returns a Boolean indicating the response:

```
bool accept()
{
      cout << "Do you want to proceed (y or n)?\n";       // write question

      char answer = 0;
      cin >> answer;                                       // read answer

      if (answer == 'y') return true;
      return false;
}
```

To match the **<<** output operator (''put to''), the **>>** operator (''get from'') is used for input; **cin** is the standard input stream. The type of the right-hand operand of **>>** determines what input is accepted, and its right-hand operand is the target of the input operation. The **\n** character at the end of the output string represents a newline (§2.2.1).

The example could be improved by taking an **n** (for ''no'') answer into account:

```
bool accept2()
{
      cout << "Do you want to proceed (y or n)?\n";       // write question

      char answer = 0;
      cin >> answer;                                       // read answer
```

```
        switch (answer) {
        case 'y':
                return true;
        case 'n':
                return false;
        default:
                cout << "I'll take that for a no.\n";
                return false;
        }
}
```

A switch-statement tests a value against a set of constants. The case constants must be distinct, and if the value tested does not match any of them, the default is chosen. If no default is provided, no action is taken if the value doesn't match any case constant.

Few programs are written without loops. For example, we might like to give the user a few tries to produce acceptable input:

```
bool accept3()
{
        int tries = 1;
        while (tries<4) {
                cout << "Do you want to proceed (y or n)?\n";       // write question
                char answer = 0;
                cin >> answer;                                      // read answer

                switch (answer) {
                case 'y':
                        return true;
                case 'n':
                        return false;
                default:
                        cout << "Sorry, I don't understand that.\n";
                        ++tries;    // increment
                }
        }
        cout << "I'll take that for a no.\n";
        return false;
}
```

The while-statement executes until its condition becomes false.

2.2.5 Pointers, Arrays, and Loops

An array of elements of type char can be declared like this:

```
char v[6];              // array of 6 characters
```

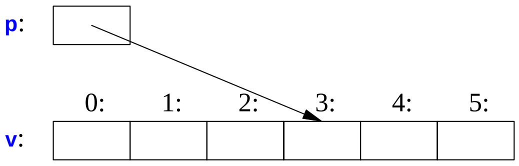
Similarly, a pointer can be declared like this:

```
char* p;                // pointer to character
```

In declarations, [] means ‘‘array of’’ and * means ‘‘pointer to.’’ All arrays have 0 as their lower bound, so **v** has six elements, **v[0]** to **v[5]**. The size of an array must be a constant expression (§2.2.3). A pointer variable can hold the address of an object of the appropriate type:

```
char∗ p = &v[3];          // p points to v's fourth element
char x = ∗p;              // *p is the object that p points to
```

In an expression, prefix unary ∗ means ‘‘contents of’’ and prefix unary **&** means ‘‘address of.’’ We can represent the result of that initialized definition graphically:



Consider copying ten elements from one array to another:

```
void copy_fct()
{
        int v1[10] = {0,1,2,3,4,5,6,7,8,9};
        int v2[10];                    // to become a copy of v1

        for (auto i=0; i!=10; ++i)     // copy elements
                v2[i]=v1[i];
        // ...
}
```

This **for**-statement can be read as ‘‘set **i** to zero; while **i** is not **10**, copy the **i**th element and increment **i**.’’ When applied to an integer variable, the increment operator, **++**, simply adds **1**. C++ also offers a simpler **for**-statement, called a range-**for**-statement, for loops that traverse a sequence in the simplest way:

```
void print()
{
        int v[] = {0,1,2,3,4,5,6,7,8,9};

        for (auto x : v)               // for each x in v
                cout << x << '\n';

        for (auto x : {10,21,32,43,54,65})
                cout << x << '\n';
        // ...
}
```

The first range-**for**-statement can be read as ‘‘for every element of **v**, from the first to the last, place a copy in **x** and print it.’’ Note that we don't have to specify an array bound when we initialize it with a list. The range-**for**-statement can be used for any sequence of elements (§3.4.1).

If we didn't want to copy the values from **v** into the variable **x**, but rather just have **x** refer to an element, we could write:

```
void increment()
{
        int v[] = {0,1,2,3,4,5,6,7,8,9};

        for (auto& x : v)
                ++x;
        // ...
}
```

In a declaration, the unary suffix `&` means ‘‘reference to.’’ A reference is similar to a pointer, except that you don’t need to use a prefix `*` to access the value referred to by the reference. Also, a reference cannot be made to refer to a different object after its initialization. When used in declarations, operators (such as `&`, `*`, and `[]`) are called *declarator operators*:

```
T a[n];     // T[n]: array of n Ts (§7.3)
T* p;       // T*: pointer to T (§7.2)
T& r;       // T&: reference to T   (§7.7)
T f(A);     // T(A): function taking an argument of type A returning a result of type T (§2.2.1)
```

We try to ensure that a pointer always points to an object, so that dereferencing it is valid. When we don’t have an object to point to or if we need to represent the notion of ‘‘no object available’’ (e.g., for an end of a list), we give the pointer the value `nullptr` (‘‘the null pointer’’). There is only one `nullptr` shared by all pointer types:

```
double* pd = nullptr;
Link<Record>* lst = nullptr;     // pointer to a Link to a Record
int x = nullptr;                 // error: nullptr is a pointer not an integer
```

It is often wise to check that a pointer argument that is supposed to point to something, actually points to something:

```
int count_x(char* p, char x)
        // count the number of occurrences of x in p[]
        // p is assumed to point to a zero-terminated array of char (or to nothing)
{
        if (p==nullptr) return 0;
        int count = 0;
        for (; *p!=0; ++p)
                if (*p==x)
                        ++count;
        return count;
}
```

Note how we can move a pointer to point to the next element of an array using `++` and that we can leave out the initializer in a `for`-statement if we don’t need it.

The definition of `count_x()` assumes that the `char*` is a *C-style string*, that is, that the pointer points to a zero-terminated array of `char`.

In older code, `0` or `NULL` is typically used instead of `nullptr` (§7.2.2). However, using `nullptr` eliminates potential confusion between integers (such as `0` or `NULL`) and pointers (such as `nullptr`).

2.3 User-Defined Types

We call the types that can be built from the fundamental types (§2.2.2), the **const** modifier (§2.2.3), and the declarator operators (§2.2.5) *built-in types*. C++'s set of built-in types and operations is rich, but deliberately low-level. They directly and efficiently reflect the capabilities of conventional computer hardware. However, they don't provide the programmer with high-level facilities to conveniently write advanced applications. Instead, C++ augments the built-in types and operations with a sophisticated set of *abstraction mechanisms* out of which programmers can build such high-level facilities. The C++ abstraction mechanisms are primarily designed to let programmers design and implement their own types, with suitable representations and operations, and for programmers to simply and elegantly use such types. Types built out of the built-in types using C++'s abstraction mechanisms are called *user-defined types*. They are referred to as classes and enumerations. Most of this book is devoted to the design, implementation, and use of user-defined types. The rest of this chapter presents the simplest and most fundamental facilities for that. Chapter 3 is a more complete description of the abstraction mechanisms and the programming styles they support. Chapter 4 and Chapter 5 present an overview of the standard library, and since the standard library mainly consists of user-defined types, they provide examples of what can be built using the language facilities and programming techniques presented in Chapter 2 and Chapter 3.

2.3.1 Structures

The first step in building a new type is often to organize the elements it needs into a data structure, a **struct**:

```
struct Vector {
        int sz;              // number of elements
        double∗ elem;   // pointer to elements
};
```

This first version of **Vector** consists of an **int** and a **double∗**.

A variable of type **Vector** can be defined like this:

```
Vector v;
```

However, by itself that is not of much use because **v**'s **elem** pointer doesn't point to anything. To be useful, we must give **v** some elements to point to. For example, we can construct a **Vector** like this:

```
void vector_init(Vector& v, int s)
{
        v.elem = new double[s];   // allocate an array of s doubles
        v.sz = s;
}
```

That is, **v**'s **elem** member gets a pointer produced by the **new** operator and **v**'s **size** member gets the number of elements. The **&** in **Vector&** indicates that we pass **v** by non-**const** reference (§2.2.5, §7.7); that way, **vector_init()** can modify the vector passed to it.

The **new** operator allocates memory from an area called *the free store* (also known as *dynamic memory* and *heap*; §11.2).

A simple use of `Vector` looks like this:

```
double read_and_sum(int s)
     // read s integers from cin and return their sum; s is assumed to be positive
{
     Vector v;
     vector_init(v,s);                    // allocate s elements for v
     for (int i=0; i!=s; ++i)
          cin>>v.elem[i];                 // read into elements

     double sum = 0;
     for (int i=0; i!=s; ++i)
          sum+=v.elem[i];                 // take the sum of the elements
     return sum;
}
```

There is a long way to go before our `Vector` is as elegant and flexible as the standard-library `vector`. In particular, a user of `Vector` has to know every detail of `Vector`'s representation. The rest of this chapter and the next gradually improve `Vector` as an example of language features and techniques. Chapter 4 presents the standard-library `vector`, which contains many nice improvements, and Chapter 31 presents the complete `vector` in the context of other standard-library facilities.

I use `vector` and other standard-library components as examples

- to illustrate language features and design techniques, and
- to help you learn and use the standard-library components.

Don't reinvent standard-library components, such as `vector` and `string`; use them.

We use `.` (dot) to access `struct` members through a name (and through a reference) and `->` to access `struct` members through a pointer. For example:

```
void f(Vector v, Vector& rv, Vector* pv)
{
     int i1 = v.sz;          // access through name
     int i2 = rv.sz;         // access through reference
     int i4 = pv->sz;        // access through pointer
}
```

2.3.2 Classes

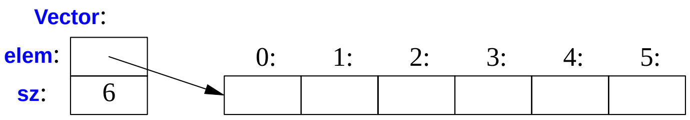
Having the data specified separately from the operations on it has advantages, such as the ability to use the data in arbitrary ways. However, a tighter connection between the representation and the operations is needed for a user-defined type to have all the properties expected of a "real type." In particular, we often want to keep the representation inaccessible to users, so as to ease use, guarantee consistent use of the data, and allow us to later improve the representation. To do that we have to distinguish between the interface to a type (to be used by all) and its implementation (which has access to the otherwise inaccessible data). The language mechanism for that is called a *class*. A class is defined to have a set of *members*, which can be data, function, or type members. The interface is defined by the `public` members of a class, and `private` members are accessible only through that interface. For example:

```
class Vector {
public:
        Vector(int s) :elem{new double[s]}, sz{s} { }   // construct a Vector
        double& operator[](int i) { return elem[i]; }     // element access: subscripting
        int size() { return sz; }
private:
        double∗ elem;  // pointer to the elements
        int sz;             // the number of elements
};
```

Given that, we can define a variable of our new type **Vector**:

```
Vector v(6);     // a Vector with 6 elements
```

We can illustrate a **Vector** object graphically:



Basically, the **Vector** object is a ‘‘handle’’ containing a pointer to the elements (**elem**) plus the number of elements (**sz**). The number of elements (6 in the example) can vary from **Vector** object to **Vector** object, and a **Vector** object can have a different number of elements at different times (§3.2.1.3). However, the **Vector** object itself is always the same size. This is the basic technique for handling varying amounts of information in C++: a fixed-size handle referring to a variable amount of data ‘‘elsewhere’’ (e.g., on the free store allocated by **new**; §11.2). How to design and use such objects is the main topic of Chapter 3.

Here, the representation of a **Vector** (the members **elem** and **sz**) is accessible only through the interface provided by the **public** members: **Vector()**, **operator[]()**, and **size()**. The **read_and_sum()** example from §2.3.1 simplifies to:

```
double read_and_sum(int s)
{
        Vector v(s);                          // make a vector of s elements
        for (int i=0; i!=v.size(); ++i)
                cin>>v[i];                    // read into elements

        double sum = 0;
        for (int i=0; i!=v.size(); ++i)
                sum+=v[i];                    // take the sum of the elements
        return sum;
}
```

A ‘‘function’’ with the same name as its class is called a *constructor*, that is, a function used to construct objects of a class. So, the constructor, **Vector()**, replaces **vector_init()** from §2.3.1. Unlike an ordinary function, a constructor is guaranteed to be used to initialize objects of its class. Thus, defining a constructor eliminates the problem of uninitialized variables for a class.

Vector(int) defines how objects of type Vector are constructed. In particular, it states that it needs an integer to do that. That integer is used as the number of elements. The constructor initializes the Vector members using a member initializer list:

```
:elem{new double[s]}, sz{s}
```

That is, we first initialize elem with a pointer to s elements of type double obtained from the free store. Then, we initialize sz to s.

Access to elements is provided by a subscript function, called operator[]. It returns a reference to the appropriate element (a double&).

The size() function is supplied to give users the number of elements.

Obviously, error handling is completely missing, but we'll return to that in §2.4.3. Similarly, we did not provide a mechanism to ‘‘give back’’ the array of doubles acquired by new; §3.2.1.2 shows how to use a destructor to elegantly do that.

2.3.3 Enumerations

In addition to classes, C++ supports a simple form of user-defined type for which we can enumerate the values:

```
enum class Color { red, blue, green };
enum class Traffic_light { green, yellow, red };

Color col = Color::red;
Traffic_light light = Traffic_light::red;
```

Note that enumerators (e.g., red) are in the scope of their enum class, so that they can be used repeatedly in different enum classes without confusion. For example, Color::red is Color's red which is different from Traffic_light::red.

Enumerations are used to represent small sets of integer values. They are used to make code more readable and less error-prone than it would have been had the symbolic (and mnemonic) enumerator names not been used.

The class after the enum specifies that an enumeration is strongly typed and that its enumerators are scoped. Being separate types, enum classes help prevent accidental misuses of constants. In particular, we cannot mix Traffic_light and Color values:

```
Color x = red;                  // error: which red?
Color y = Traffic_light::red;   // error: that red is not a Color
Color z = Color::red;           // OK
```

Similarly, we cannot implicitly mix Color and integer values:

```
int i = Color::red;             // error: Color::red is not an int
Color c = 2;                    // error: 2 is not a Color
```

If you don't want to explicitly qualify enumerator names and want enumerator values to be ints (without the need for an explicit conversion), you can remove the class from enum class to get a ‘‘plain enum’’ (§8.4.2).

By default, an enum class has only assignment, initialization, and comparisons (e.g., == and <; §2.2.2) defined. However, an enumeration is a user-defined type so we can define operators for it:

```
Traffic_light& operator++(Traffic_light& t)
        // prefix increment: ++
{
        switch (t) {
        case Traffic_light::green:      return t=Traffic_light::yellow;
        case Traffic_light::yellow:     return t=Traffic_light::red;
        case Traffic_light::red:        return t=Traffic_light::green;
        }
}

Traffic_light next = ++light;           // next becomes Traffic_light::green
```

C++ also offers a less strongly typed ‘‘plain’’ `enum` (§8.4.2).

2.4 Modularity

A C++ program consists of many separately developed parts, such as functions (§2.2.1, Chapter 12), user-defined types (§2.3, §3.2, Chapter 16), class hierarchies (§3.2.4, Chapter 20), and templates (§3.4, Chapter 23). The key to managing this is to clearly define the interactions among those parts. The first and most important step is to distinguish between the interface to a part and its implementation. At the language level, C++ represents interfaces by declarations. A *declaration* specifies all that's needed to use a function or a type. For example:

```
double sqrt(double);       // the square root function takes a double and returns a double

class Vector {
public:
        Vector(int s);
        double& operator[](int i);
        int size();
private:
        double∗ elem;  // elem points to an array of sz doubles
        int sz;
};
```

The key point here is that the function bodies, the function *definitions*, are ‘‘elsewhere.’’ For this example, we might like for the representation of `Vector` to be ‘‘elsewhere’’ also, but we will deal with that later (abstract types; §3.2.2). The definition of `sqrt()` will look like this:

```
double sqrt(double d)           // definition of sqrt()
{
        // ... algorithm as found in math textbook ...
}
```

For `Vector`, we need to define all three member functions:

```
Vector::Vector(int s)                   // definition of the constructor
        :elem{new double[s]}, sz{s}     // initialize members
{
}
```

```
double& Vector::operator[](int i)          // definition of subscripting
{
      return elem[i];
}

int Vector::size()                          // definition of size()
{
      return sz;
}
```

We must define `Vector`'s functions, but not `sqrt()` because it is part of the standard library. However, that makes no real difference: a library is simply some ‘‘other code we happen to use’’ written with the same language facilities as we use.

2.4.1 Separate Compilation

C++ supports a notion of separate compilation where user code sees only declarations of types and functions used. The definitions of those types and functions are in separate source files and compiled separately. This can be used to organize a program into a set of semi-independent code fragments. Such separation can be used to minimize compilation times and to strictly enforce separation of logically distinct parts of a program (thus minimizing the chance of errors). A library is often a separately compiled code fragments (e.g., functions).

Typically, we place the declarations that specify the interface to a module in a file with a name indicating its intended use. For example:

```
// Vector.h:

class Vector {
public:
      Vector(int s);
      double& operator[](int i);
      int size();
private:
      double* elem;        // elem points to an array of sz doubles
      int sz;
};
```

This declaration would be placed in a file `Vector.h`, and users will *include* that file, called a *header file*, to access that interface. For example:

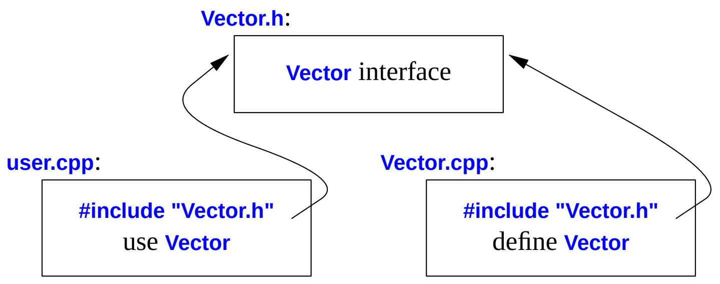
```
// user.cpp:

#include "Vector.h"        // get Vector's interface
#include <cmath>           // get the the standard-library math function interface including sqrt()
using namespace std;       // make std members visible (§2.4.2)
```

```
double sqrt_sum(Vector& v)
{
        double sum = 0;
        for (int i=0; i!=v.size(); ++i)
                sum+=sqrt(v[i]);                        // sum of square roots
        return sum;
}
```

To help the compiler ensure consistency, the .cpp file providing the implementation of Vector will also include the .h file providing its interface:

```
// Vector.cpp:

#include "Vector.h"  // get the interface

Vector::Vector(int s)
        :elem{new double[s]}, sz{s}
{
}

double& Vector::operator[](int i)
{
        return elem[i];
}

int Vector::size()
{
        return sz;
}
```

The code in user.cpp and Vector.cpp shares the Vector interface information presented in Vector.h, but the two files are otherwise independent and can be separately compiled. Graphically, the program fragments can be represented like this:



Strictly speaking, using separate compilation isn't a language issue; it is an issue of how best to take advantage of a particular language implementation. However, it is of great practical importance. The best approach is to maximize modularity, represent that modularity logically through language features, and then exploit the modularity physically through files for effective separate compilation (Chapter 14, Chapter 15).

2.4.2 Namespaces

In addition to functions (§2.2.1, Chapter 12), classes (Chapter 16), and enumerations (§2.3.3, §8.4), C++ offers *namespaces* (Chapter 14) as a mechanism for expressing that some declarations belong together and that their names shouldn't clash with other names. For example, I might want to experiment with my own complex number type (§3.2.1.1, §18.3, §40.4):

```
namespace My_code {
        class complex { /* ... */ };
        complex sqrt(complex);
        // ...
        int main();
}

int My_code::main()
{
        complex z {1,2};
        auto z2 = sqrt(z);
        std::cout << '{' << z2.real() << ',' << z2.imag() << "}\n";
        // ...
};

int main()
{
        return My_code::main();
}
```

By putting my code into the namespace **My_code**, I make sure that my names do not conflict with the standard-library names in namespace **std** (§4.1.2). The precaution is wise, because the standard library does provide support for **complex** arithmetic (§3.2.1.1, §40.4).

The simplest way to access a name in another namespace is to qualify it with the namespace name (e.g., **std::cout** and **My_code::main**). The "real **main()**" is defined in the global namespace, that is, not local to a defined namespace, class, or function. To gain access to names in the standard-library namespace, we can use a **using**-directive (§14.2.3):

```
using namespace std;
```

Namespaces are primarily used to organize larger program components, such as libraries. They simplify the composition of a program out of separately developed parts.

2.4.3 Error Handling

Error handling is a large and complex topic with concerns and ramifications that go far beyond language facilities into programming techniques and tools. However, C++ provides a few features to help. The major tool is the type system itself. Instead of painstakingly building up our applications from the built-in types (e.g., **char**, **int,** and **double**) and statements (e.g., **if**, **while,** and **for**), we build more types that are appropriate for our applications (e.g., **string**, **map**, and **regex**) and algorithms (e.g., **sort()**, **find_if()**, and **draw_all()**). Such higher level constructs simplify our programming, limit our opportunities for mistakes (e.g., you are unlikely to try to apply a tree traversal to a dialog box), and increase the compiler's chances of catching such errors. The majority of C++ constructs are dedicated to the design and implementation of elegant and efficient abstractions (e.g., user-defined types and algorithms using them). One effect of this modularity and abstraction (in particular, the use of libraries) is that the point where a run-time error can be detected is separated from the point where it can be handled. As programs grow, and especially when libraries are used extensively, standards for handling errors become important.

2.4.3.1 Exceptions

Consider again the **Vector** example. What *ought* to be done when we try to access an element that is out of range for the vector from §2.3.2?

- The writer of **Vector** doesn't know what the user would like to have done in this case (the writer of **Vector** typically doesn't even know in which program the vector will be running).
- The user of **Vector** cannot consistently detect the problem (if the user could, the out-of-range access wouldn't happen in the first place).

The solution is for the **Vector** implementer to detect the attempted out-of-range access and then tell the user about it. The user can then take appropriate action. For example, **Vector::operator[]()** can detect an attempted out-of-range access and throw an **out_of_range** exception:

```
double& Vector::operator[](int i)
{
        if (i<0 || size()<=i) throw out_of_range{"Vector::operator[]"};
        return elem[i];
}
```

The **throw** transfers control to a handler for exceptions of type **out_of_range** in some function that directly or indirectly called **Vector::operator[]()**. To do that, the implementation will unwind the function call stack as needed to get back to the context of that caller (§13.5.1). For example:

```
void f(Vector& v)
{
        // ...
        try { // exceptions here are handled by the handler defined below

                v[v.size()] = 7;  // try to access beyond the end of v
        }
        catch (out_of_range) {     // oops: out_of_range error
                // ... handle range error ...
        }
        // ...
}
```

We put code for which we are interested in handling exceptions into a **try**-block. That attempted assignment to **v[v.size()]** will fail. Therefore, the **catch**-clause providing a handler for **out_of_range** will be entered. The **out_of_range** type is defined in the standard library and is in fact used by some standard-library container access functions.

Use of the exception-handling mechanisms can make error handling simpler, more systematic, and more readable. See Chapter 13 for further discussion, details, and examples.

2.4.3.2 Invariants

The use of exceptions to signal out-of-range access is an example of a function checking its argument and refusing to act because a basic assumption, a *precondition*, didn't hold. Had we formally specified **Vector**'s subscript operator, we would have said something like "the index must be in the [**0**:**size()**) range," and that was in fact what we tested in our **operator[]()**. Whenever we define a function, we should consider what its preconditions are and if feasible test them (see §12.4, §13.4).

However, **operator[]()** operates on objects of type **Vector** and nothing it does makes any sense unless the members of **Vector** have "reasonable" values. In particular, we did say "**elem** points to an array of **sz** doubles" but we only said that in a comment. Such a statement of what is assumed to be true for a class is called a *class invariant*, or simply an *invariant*. It is the job of a constructor to establish the invariant for its class (so that the member functions can rely on it) and for the member functions to make sure that the invariant holds when they exit. Unfortunately, our **Vector** constructor only partially did its job. It properly initialized the **Vector** members, but it failed to check that the arguments passed to it made sense. Consider:

```
Vector v(−27);
```

This is likely to cause chaos.

Here is a more appropriate definition:

```
Vector::Vector(int s)
{
        if (s<0) throw length_error{};
        elem = new double[s];
        sz = s;
}
```

I use the standard-library exception **length_error** to report a non-positive number of elements because some standard-library operations use that exception to report problems of this kind. If operator **new** can't find memory to allocate, it throws a **std::bad_alloc**. We can now write:

```
void test()
{
        try {
                Vector v(−27);
        }
        catch (std::length_error) {
                // handle negative size
        }
        catch (std::bad_alloc) {
                // handle memory exhaustion
        }
}
```

You can define your own classes to be used as exceptions and have them carry arbitrary information from a point where an error is detected to a point where it can be handled (§13.5).

Often, a function has no way of completing its assigned task after an exception is thrown. Then, "handling" an exception simply means doing some minimal local cleanup and rethrowing the exception (§13.5.2.1).

The notion of invariants is central to the design of classes, and preconditions serve a similar role in the design of functions. Invariants

- helps us to understand precisely what we want
- forces us to be specific; that gives us a better chance of getting our code correct (after debugging and testing).

The notion of invariants underlies C++'s notions of resource management supported by constructors (§2.3.2) and destructors (§3.2.1.2, §5.2). See also §13.4, §16.3.1, and §17.2.

2.4.3.3 Static Assertions

Exceptions report errors found at run time. If an error can be found at compile time, it is usually preferable to do so. That's what much of the type system and the facilities for specifying the interfaces to user-defined types are for. However, we can also perform simple checks on other properties that are known at compile time and report failures as compiler error messages. For example:

```
static_assert(4<=sizeof(int), "integers are too small");   // check integer size
```

This will write integers are too small if 4<=sizeof(int) does not hold, that is, if an int on this system does not have at least 4 bytes. We call such statements of expectations *assertions*.

The static_assert mechanism can be used for anything that can be expressed in terms of constant expressions (§2.2.3, §10.4). For example:

```
constexpr double C = 299792.458;                        // km/s

void f(double speed)
{
     const double local_max = 160.0/(60*60);            // 160 km/h == 160.0/(60*60) km/s

     static_assert(speed<C,"can't go that fast");       // error: speed must be a constant
     static_assert(local_max<C,"can't go that fast");   // OK

     // ...
}
```

In general, static_assert(A,S) prints S as a compiler error message if A is not true.

The most important uses of static_assert come when we make assertions about types used as parameters in generic programming (§5.4.2, §24.3).

For runtime-checked assertions, see §13.4.

2.5 Postscript

The topics covered in this chapter roughly correspond to the contents of Part II (Chapters 6–15). Those are the parts of C++ that underlie all programming techniques and styles supported by C++. Experienced C and C++ programmers, please note that this foundation does not closely correspond to the C or C++98 subsets of C++ (that is, C++11).

2.6 Advice

[1] Don’t panic! All will become clear in time; §2.1.
[2] You don’t have to know every detail of C++ to write good programs; §1.3.1.
[3] Focus on programming techniques, not on language features; §2.1.

3

A Tour of C++: Abstraction Mechanisms

Don't Panic!
– Douglas Adams

3.1 Introduction

This chapter aims to give you an idea of C++'s support for abstraction and resource management without going into a lot of detail. It informally presents ways of defining and using new types (*user-defined types*). In particular, it presents the basic properties, implementation techniques, and language facilities used for *concrete classes*, *abstract classes*, and *class hierarchies*. Templates are introduced as a mechanism for parameterizing types and algorithms with (other) types and algorithms. Computations on user-defined and built-in types are represented as functions, sometimes generalized to *template functions* and *function objects*. These are the language facilities supporting the programming styles known as *object-oriented programming* and *generic programming*. The next two chapters follow up by presenting examples of standard-library facilities and their use.

The assumption is that you have programmed before. If not, please consider reading a textbook, such as *Programming: Principles and Practice Using C++* [Stroustrup,2009], before continuing here. Even if you have programmed before, the language you used or the applications you wrote may be very different from the style of C++ presented here. If you find this ‘‘lightning tour’’ confusing, skip to the more systematic presentation starting in Chapter 6.

As in Chapter 2, this tour presents C++ as an integrated whole, rather than as a layer cake. Consequently, it does not identify language features as present in C, part of C++98, or new in C++11. Such historical information can be found in §1.4 and Chapter 44.

3.2 Classes

The central language feature of C++ is the *class*. A class is a user-defined type provided to represent a concept in the code of a program. Whenever our design for a program has a useful concept, idea, entity, etc., we try to represent it as a class in the program so that the idea is there in the code, rather than just in our head, in a design document, or in some comments. A program built out of a well chosen set of classes is far easier to understand and get right than one that builds everything directly in terms of the built-in types. In particular, classes are often what libraries offer.

Essentially all language facilities beyond the fundamental types, operators, and statements exist to help define better classes or to use them more conveniently. By ‘‘better,’’ I mean more correct, easier to maintain, more efficient, more elegant, easier to use, easier to read, and easier to reason about. Most programming techniques rely on the design and implementation of specific kinds of classes. The needs and tastes of programmers vary immensely. Consequently, the support for classes is extensive. Here, we will just consider the basic support for three important kinds of classes:

- Concrete classes (§3.2.1)
- Abstract classes (§3.2.2)
- Classes in class hierarchies (§3.2.4)

An astounding number of useful classes turn out to be of these three kinds. Even more classes can be seen as simple variants of these kinds or are implemented using combinations of the techniques used for these.

3.2.1 Concrete Types

The basic idea of *concrete classes* is that they behave ‘‘just like built-in types.’’ For example, a complex number type and an infinite-precision integer are much like built-in **int**, except of course that they have their own semantics and sets of operations. Similarly, a **vector** and a **string** are much like built-in arrays, except that they are better behaved (§4.2, §4.3.2, §4.4.1).

The defining characteristic of a concrete type is that its representation is part of its definition. In many important cases, such as a **vector**, that representation is only one or more pointers to more data stored elsewhere, but it is present in each object of a concrete class. That allows implementations to be optimally efficient in time and space. In particular, it allows us to

- place objects of concrete types on the stack, in statically allocated memory, and in other objects (§6.4.2);
- refer to objects directly (and not just through pointers or references);
- initialize objects immediately and completely (e.g., using constructors; §2.3.2); and
- copy objects (§3.3).

The representation can be private (as it is for **Vector**; §2.3.2) and accessible only through the member functions, but it is present. Therefore, if the representation changes in any significant way, a user must recompile. This is the price to pay for having concrete types behave exactly like built-in types. For types that don't change often, and where local variables provide much-needed clarity and efficiency, this is acceptable and often ideal. To increase flexibility, a concrete type can keep major parts of its representation on the free store (dynamic memory, heap) and access them through the part stored in the class object itself. That's the way `vector` and `string` are implemented; they can be considered resource handles with carefully crafted interfaces.

3.2.1.1 An Arithmetic Type

The "classical user-defined arithmetic type" is `complex`:

```
class complex {
        double re, im;  // representation: two doubles
public:
        complex(double r, double i) :re{r}, im{i} {}       // construct complex from two scalars
        complex(double r) :re{r}, im{0} {}                 // construct complex from one scalar
        complex() :re{0}, im{0} {}                         // default complex: {0,0}

        double real() const { return re; }
        void real(double d) { re=d; }
        double imag() const { return im; }
        void imag(double d) { im=d; }

        complex& operator+=(complex z) { re+=z.re, im+=z.im; return *this; }   // add to re and im
                                                                                // and return the result
        complex& operator-=(complex z) { re-=z.re, im-=z.im; return *this; }

        complex& operator*=(complex);      // defined out-of-class somewhere
        complex& operator/=(complex);      // defined out-of-class somewhere
};
```

This is a slightly simplified version of the standard-library `complex` (§40.4). The class definition itself contains only the operations requiring access to the representation. The representation is simple and conventional. For practical reasons, it has to be compatible with what Fortran provided 50 years ago, and we need a conventional set of operators. In addition to the logical demands, `complex` must be efficient or it will remain unused. This implies that simple operations must be inlined. That is, simple operations (such as constructors, `+=`, and `imag()`) must be implemented without function calls in the generated machine code. Functions defined in a class are inlined by default. An industrial-strength `complex` (like the standard-library one) is carefully implemented to do appropriate inlining.

A constructor that can be invoked without an argument is called a *default constructor*. Thus, `complex()` is `complex`'s default constructor. By defining a default constructor you eliminate the possibility of uninitialized variables of that type.

The `const` specifiers on the functions returning the real and imaginary parts indicate that these functions do not modify the object for which they are called.

Many useful operations do not require direct access to the representation of `complex`, so they can be defined separately from the class definition:

```
complex operator+(complex a, complex b) { return a+=b; }
complex operator–(complex a, complex b) { return a–=b; }
complex operator–(complex a) { return {–a.real(), –a.imag()}; }      // unary minus
complex operator*(complex a, complex b) { return a*=b; }
complex operator/(complex a, complex b) { return a/=b; }
```

Here, I use the fact that an argument passed by value is copied, so that I can modify an argument without affecting the caller's copy, and use the result as the return value.

The definitions of `==` and `!=` are straightforward:

```
bool operator==(complex a, complex b)        // equal
{
      return a.real()==b.real() && a.imag()==b.imag();
}

bool operator!=(complex a, complex b)        // not equal
{
      return !(a==b);
}

complex sqrt(complex);

// ...
```

Class `complex` can be used like this:

```
void f(complex z)
{
      complex a {2.3};            // construct {2.3,0.0} from 2.3
      complex b {1/a};
      complex c {a+z*complex{1,2.3}};
      // ...
      if (c != b)
            c = –(b/a)+2*b;
}
```

The compiler converts operators involving `complex` numbers into appropriate function calls. For example, `c!=b` means `operator!=(c,b)` and `1/a` means `operator/(complex{1},a)`.

User-defined operators ("overloaded operators") should be used cautiously and conventionally. The syntax is fixed by the language, so you can't define a unary `/`. Also, it is not possible to change the meaning of an operator for built-in types, so you can't redefine `+` to subtract `int`s.

3.2.1.2 A Container

A *container* is an object holding a collection of elements, so we call `Vector` a container because it is the type of objects that are containers. As defined in §2.3.2, `Vector` isn't an unreasonable container of `double`s: it is simple to understand, establishes a useful invariant (§2.4.3.2), provides range-checked access (§2.4.3.1), and provides `size()` to allow us to iterate over its elements. However, it does have a fatal flaw: it allocates elements using `new` but never deallocates them. That's not a good idea because although C++ defines an interface for a garbage collector (§34.5), it is not guaranteed that one is available to make unused memory available for new objects. In some environments you can't use a collector, and sometimes you prefer more precise control of destruction (§13.6.4) for logical or performance reasons. We need a mechanism to ensure that the memory allocated by the constructor is deallocated; that mechanism is a *destructor*:

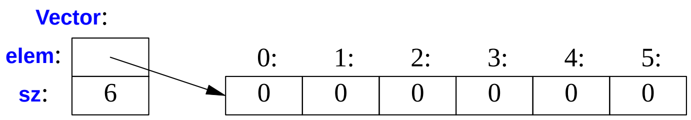
```
class Vector {
private:
      double∗ elem;          // elem points to an array of sz doubles
      int sz;
public:
      Vector(int s) :elem{new double[s]}, sz{s}      // constructor: acquire resources
      {
            for (int i=0; i!=s; ++i) elem[i]=0;       // initialize elements
      }

      ˜Vector() { delete[] elem; }                    // destructor: release resources

      double& operator[](int i);
      int size() const;
};
```

The name of a destructor is the complement operator, ˜, followed by the name of the class; it is the complement of a constructor. Vector's constructor allocates some memory on the free store (also called the *heap* or *dynamic store*) using the new operator. The destructor cleans up by freeing that memory using the delete operator. This is all done without intervention by users of Vector. The users simply create and use Vectors much as they would variables of built-in types. For example:

```
void fct(int n)
{
      Vector v(n);

      // ... use v ...

      {
            Vector v2(2∗n);
            // ... use v and v2 ...
      } // v2 is destroyed here

      // ... use v ..

} // v is destroyed here
```

Vector obeys the same rules for naming, scope, allocation, lifetime, etc., as does a built-in type, such as int and char. For details on how to control the lifetime of an object, see §6.4. This Vector has been simplified by leaving out error handling; see §2.4.3.

The constructor/destructor combination is the basis of many elegant techniques. In particular, it is the basis for most C++ general resource management techniques (§5.2, §13.3). Consider a graphical illustration of a Vector:



The constructor allocates the elements and initializes the **Vector** members appropriately. The destructor deallocates the elements. This *handle-to-data model* is very commonly used to manage data that can vary in size during the lifetime of an object. The technique of acquiring resources in a constructor and releasing them in a destructor, known as *Resource Acquisition Is Initialization* or *RAII*, allows us to eliminate ''naked **new** operations,'' that is, to avoid allocations in general code and keep them buried inside the implementation of well-behaved abstractions. Similarly, ''naked **delete** operations'' should be avoided. Avoiding naked **new** and naked **delete** makes code far less error-prone and far easier to keep free of resource leaks (§5.2).

3.2.1.3 Initializing Containers

A container exists to hold elements, so obviously we need convenient ways of getting elements into a container. We can handle that by creating a **Vector** with an appropriate number of elements and then assigning to them, but typically other ways are more elegant. Here, I just mention two favorites:

- *Initializer-list constructor*: Initialize with a list of elements.
- **push_back()**: Add a new element at the end (at the back of) the sequence.

These can be declared like this:

```
class Vector {
public:
     Vector(std::initializer_list<double>);          // initialize with a list
     // ...
     void push_back(double);                          // add element at end increasing the size by one
     // ...
};
```

The **push_back()** is useful for input of arbitrary numbers of elements. For example:

```
Vector read(istream& is)
{
     Vector v;
     for (double d; is>>d;)          // read floating-point values into d
          v.push_back(d);            // add d to v
     return v;
}
```

The input loop is terminated by an end-of-file or a formatting error. Until that happens, each number read is added to the **Vector** so that at the end, **v**'s size is the number of elements read. I used a **for**-statement rather than the more conventional **while**-statement to keep the scope of **d** limited to the loop. The implementation of **push_back()** is discussed in §13.6.4.3. The way to provide **Vector** with a move constructor, so that returning a potentially huge amount of data from **read()** is cheap, is explained in §3.3.2.

The **std::initializer_list** used to define the initializer-list constructor is a standard-library type known to the compiler: when we use a **{}**-list, such as **{1,2,3,4}**, the compiler will create an object of type **initializer_list** to give to the program. So, we can write:

```
Vector v1 = {1,2,3,4,5};              // v1 has 5 elements
Vector v2 = {1.23, 3.45, 6.7, 8};     // v2 has 4 elements
```

Vector's initializer-list constructor might be defined like this:

```
Vector::Vector(std::initializer_list<double> lst)      // initialize with a list
        :elem{new double[lst.size()]}, sz{lst.size()}
{
        copy(lst.begin(),lst.end(),elem);             // copy from lst into elem
}
```

3.2.2 Abstract Types

Types such as **complex** and **Vector** are called *concrete types* because their representation is part of their definition. In that, they resemble built-in types. In contrast, an *abstract type* is a type that completely insulates a user from implementation details. To do that, we decouple the interface from the representation and give up genuine local variables. Since we don't know anything about the representation of an abstract type (not even its size), we must allocate objects on the free store (§3.2.1.2, §11.2) and access them through references or pointers (§2.2.5, §7.2, §7.7).

First, we define the interface of a class **Container** which we will design as a more abstract version of our **Vector**:

```
class Container {
public:
        virtual double& operator[](int) = 0;        // pure virtual function
        virtual int size() const = 0;               // const member function (§3.2.1.1)
        virtual ˜Container() {}                     // destructor (§3.2.1.2)
};
```

This class is a pure interface to specific containers defined later. The word **virtual** means "may be redefined later in a class derived from this one." Unsurprisingly, a function declared **virtual** is called a *virtual function*. A class derived from **Container** provides an implementation for the **Container** interface. The curious **=0** syntax says the function is *pure virtual*; that is, some class derived from **Container** *must* define the function. Thus, it is not possible to define an object that is just a **Container**; a **Container** can only serve as the interface to a class that implements its **operator[]()** and **size()** functions. A class with a pure virtual function is called an *abstract class*.

This **Container** can be used like this:

```
void use(Container& c)
{
        const int sz = c.size();

        for (int i=0; i!=sz; ++i)
                cout << c[i] << '\n';
}
```

Note how **use()** uses the **Container** interface in complete ignorance of implementation details. It uses **size()** and **[]** without any idea of exactly which type provides their implementation. A class that provides the interface to a variety of other classes is often called a *polymorphic type* (§20.3.2).

As is common for abstract classes, **Container** does not have a constructor. After all, it does not have any data to initialize. On the other hand, **Container** does have a destructor and that destructor is **virtual**. Again, that is common for abstract classes because they tend to be manipulated through references or pointers, and someone destroying a **Container** through a pointer has no idea what resources are owned by its implementation; see also §3.2.4.

A container that implements the functions required by the interface defined by the abstract class **Container** could use the concrete class **Vector**:

```
class Vector_container : public Container {    // Vector_container implements Container
        Vector v;
public:
        Vector_container(int s) : v(s) { }      // Vector of s elements
        ˜Vector_container() {}

        double& operator[](int i) { return v[i]; }
        int size() const { return v.size(); }
};
```

The **:public** can be read as ‘‘is derived from’’ or ‘‘is a subtype of.’’ Class **Vector_container** is said to be *derived* from class **Container**, and class **Container** is said to be a *base* of class **Vector_container**. An alternative terminology calls **Vector_container** and **Container** *subclass* and *superclass*, respectively. The derived class is said to inherit members from its base class, so the use of base and derived classes is commonly referred to as *inheritance*.

The members **operator[]()** and **size()** are said to *override* the corresponding members in the base class **Container** (§20.3.2). The destructor (**˜Vector_container()**) overrides the base class destructor (**˜Container()**). Note that the member destructor (**˜Vector()**) is implicitly invoked by its class's destructor (**˜Vector_container()**).

For a function like **use(Container&)** to use a **Container** in complete ignorance of implementation details, some other function will have to make an object on which it can operate. For example:

```
void g()
{
        Vector_container vc {10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
        use(vc);
}
```

Since **use()** doesn't know about **Vector_container**s but only knows the **Container** interface, it will work just as well for a different implementation of a **Container**. For example:

```
class List_container : public Container { // List_container implements Container
        std::list<double> ld;        // (standard-library) list of doubles (§4.4.2)
public:
        List_container() { }         // empty List
        List_container(initializer_list<double> il) : ld{il} { }
        ˜List_container() {}
        double& operator[](int i);
        int size() const { return ld.size(); }

};

double& List_container::operator[](int i)
{
        for (auto& x : ld) {
                if (i==0) return x;
                −−i;
        }
        throw out_of_range("List container");
}
```

Here, the representation is a standard-library list<double>. Usually, I would not implement a container with a subscript operation using a list, because performance of list subscripting is atrocious compared to vector subscripting. However, here I just wanted to show an implementation that is radically different from the usual one.

A function can create a List_container and have use() use it:

```
void h()
{
        List_container lc = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        use(lc);
}
```

The point is that use(Container&) has no idea if its argument is a Vector_container, a List_container, or some other kind of container; it doesn't need to know. It can use any kind of Container. It knows only the interface defined by Container. Consequently, use(Container&) needn't be recompiled if the implementation of List_container changes or a brand-new class derived from Container is used.

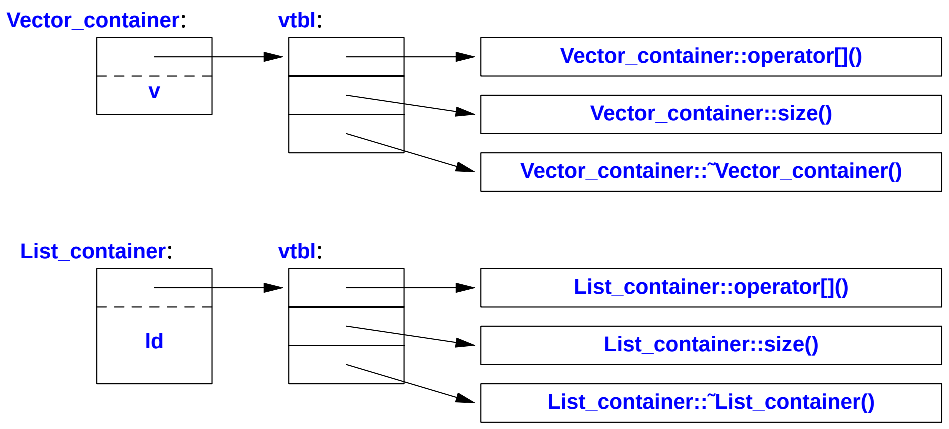
The flip side of this flexibility is that objects must be manipulated through pointers or references (§3.3, §20.4).

3.2.3 Virtual Functions

Consider again the use of Container:

```
void use(Container& c)
{
        const int sz = c.size();

        for (int i=0; i!=sz; ++i)
                cout << c[i] << '\n';
}
```

How is the call c[i] in use() resolved to the right operator[]()? When h() calls use(), List_container's operator[]() must be called. When g() calls use(), Vector_container's operator[]() must be called. To achieve this resolution, a Container object must contain information to allow it to select the right function to call at run time. The usual implementation technique is for the compiler to convert the name of a virtual function into an index into a table of pointers to functions. That table is usually called the *virtual function table* or simply the **vtbl**. Each class with virtual functions has its own **vtbl** identifying its virtual functions. This can be represented graphically like this:

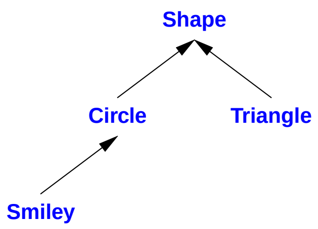


Vector_container:
vtbl:
v
Vector_container::operator[]()
Vector_container::size()
Vector_container::~Vector_container()
List_container:
vtbl:
ld
List_container::operator[]()
List_container::size()
List_container::~List_container()

The functions in the **vtbl** allow the object to be used correctly even when the size of the object and the layout of its data are unknown to the caller. The implementation of the caller needs only to know the location of the pointer to the **vtbl** in a **Container** and the index used for each virtual function. This virtual call mechanism can be made almost as efficient as the ‘‘normal function call’’ mechanism (within 25%). Its space overhead is one pointer in each object of a class with virtual functions plus one **vtbl** for each such class.

3.2.4 Class Hierarchies

The **Container** example is a very simple example of a class hierarchy. A *class hierarchy* is a set of classes ordered in a lattice created by derivation (e.g., **: public**). We use class hierarchies to represent concepts that have hierarchical relationships, such as ‘‘A fire engine is a kind of a truck which is a kind of a vehicle’’ and ‘‘A smiley face is a kind of a circle which is a kind of a shape.’’ Huge hierarchies, with hundreds of classes, that are both deep and wide are common. As a semi-realistic classic example, let’s consider shapes on a screen:



The arrows represent inheritance relationships. For example, class **Circle** is derived from class **Shape**. To represent that simple diagram in code, we must first specify a class that defines the general properties of all shapes:

```
class Shape {
public:
        virtual Point center() const =0;        // pure virtual
        virtual void move(Point to) =0;

        virtual void draw() const = 0;          // draw on current "Canvas"
        virtual void rotate(int angle) = 0;

        virtual ~Shape() {}                     // destructor
        // ...
};
```

Naturally, this interface is an abstract class: as far as representation is concerned, *nothing* (except the location of the pointer to the vtbl) is common for every Shape. Given this definition, we can write general functions manipulating vectors of pointers to shapes:

```
void rotate_all(vector<Shape*>& v, int angle) // rotate v's elements by angle degrees
{
        for (auto p : v)
                p->rotate(angle);
}
```

To define a particular shape, we must say that it is a Shape and specify its particular properties (including its virtual functions):

```
class Circle : public Shape {
public:
        Circle(Point p, int rr);          // constructor

        Point center() const { return x; }
        void move(Point to) { x=to; }

        void draw() const;
        void rotate(int) {}               // nice simple algorithm
private:
        Point x;    // center
        int r;      // radius
};
```

So far, the Shape and Circle example provides nothing new compared to the Container and Vector_container example, but we can build further:

```
class Smiley : public Circle {    // use the circle as the base for a face
public:
        Smiley(Point p, int r) : Circle{p,r}, mouth{nullptr} { }

        ~Smiley()
        {
                delete mouth;
                for (auto p : eyes) delete p;
        }
```

```
          void move(Point to);

          void draw() const;
          void rotate(int);

          void add_eye(Shape* s) { eyes.push_back(s); }
          void set_mouth(Shape* s);
          virtual void wink(int i);        // wink eye number i

          // ...

private:
          vector<Shape*> eyes;             // usually two eyes
          Shape* mouth;
};
```

The push_back() member function adds its argument to the vector (here, eyes), increasing that vector's size by one.

We can now define Smiley::draw() using calls to Smiley's base and member draw()s:

```
void Smiley::draw()
{
          Circle::draw();
          for (auto p : eyes)
                    p->draw();
          mouth->draw();
}
```

Note the way that Smiley keeps its eyes in a standard-library vector and deletes them in its destructor. Shape's destructor is virtual and Smiley's destructor overrides it. A virtual destructor is essential for an abstract class because an object of a derived class is usually manipulated through the interface provided by its abstract base class. In particular, it may be deleted through a pointer to a base class. Then, the virtual function call mechanism ensures that the proper destructor is called. That destructor then implicitly invokes the destructors of its bases and members.

In this simplified example, it is the programmer's task to place the eyes and mouth appropriately within the circle representing the face.

We can add data members, operations, or both as we define a new class by derivation. This gives great flexibility with corresponding opportunities for confusion and poor design. See Chapter 21. A class hierarchy offers two kinds of benefits:

- *Interface inheritance*: An object of a derived class can be used wherever an object of a base class is required. That is, the base class acts as an interface for the derived class. The Container and Shape classes are examples. Such classes are often abstract classes.
- *Implementation inheritance*: A base class provides functions or data that simplifies the implementation of derived classes. Smiley's uses of Circle's constructor and of Circle::draw() are examples. Such base classes often have data members and constructors.

Concrete classes – especially classes with small representations – are much like built-in types: we define them as local variables, access them using their names, copy them around, etc. Classes in class hierarchies are different: we tend to allocate them on the free store using new, and we access them through pointers or references. For example, consider a function that reads data describing shapes from an input stream and constructs the appropriate Shape objects:

```
enum class Kind { circle, triangle, smiley };

Shape* read_shape(istream& is)     // read shape descriptions from input stream is
{
        // ... read shape header from is and find its Kind k ...

        switch (k) {
        case Kind::circle:
                // read circle data {Point,int} into p and r
                return new Circle{p,r};
        case Kind::triangle:
                // read triangle data {Point,Point,Point} into p1, p2, and p3
                return new Triangle{p1,p2,p3};
        case Kind::smiley:
                // read smiley data {Point,int,Shape,Shape,Shape} into p, r, e1 ,e2, and m
                Smiley* ps = new Smiley{p,r};
                ps–>add_eye(e1);
                ps–>add_eye(e2);
                ps–>set_mouth(m);
                return ps;
        }
}
```

A program may use that shape reader like this:

```
void user()
{
        std::vector<Shape*> v;
        while (cin)
                v.push_back(read_shape(cin));
        draw_all(v);                          // call draw() for each element
        rotate_all(v,45);                     // call rotate(45) for each element
        for (auto p : v) delete p;            // remember to delete elements
}
```

Obviously, the example is simplified – especially with respect to error handling – but it vividly illustrates that user() has absolutely no idea of which kinds of shapes it manipulates. The user() code can be compiled once and later used for new Shapes added to the program. Note that there are no pointers to the shapes outside user(), so user() is responsible for deallocating them. This is done with the delete operator and relies critically on Shape's virtual destructor. Because that destructor is virtual, delete invokes the destructor for the most derived class. This is crucial because a derived class may have acquired all kinds of resources (such as file handles, locks, and output streams) that need to be released. In this case, a Smiley deletes its eyes and mouth objects.

Experienced programmers will notice that I left open two obvious opportunities for mistakes:

- A user might fail to delete the pointer returned by read_shape().
- The owner of a container of Shape pointers might not delete the objects pointed to.

In that sense, functions returning a pointer to an object allocated on the free store are dangerous.

One solution to both problems is to return a standard-library `unique_ptr` (§5.2.1) rather than a “naked pointer” and store `unique_ptr`s in the container:

```
unique_ptr<Shape> read_shape(istream& is) // read shape descriptions from input stream is
{
        // read shape header from is and find its Kind k

        switch (k) {
        case Kind::circle:
                // read circle data {Point,int} into p and r
                return unique_ptr<Shape>{new Circle{p,r}};          // §5.2.1
        // ...
}

void user()
{
        vector<unique_ptr<Shape>> v;
        while (cin)
                v.push_back(read_shape(cin));
        draw_all(v);                    // call draw() for each element
        rotate_all(v,45);                    // call rotate(45) for each element
} // all Shapes implicitly destroyed
```

Now the object is owned by the `unique_ptr` which will `delete` the object when it is no longer needed, that is, when its `unique_ptr` goes out of scope.

For the `unique_ptr` version of `user()` to work, we need versions of `draw_all()` and `rotate_all()` that accept `vector<unique_ptr<Shape>>`s. Writing many such `_all()` functions could become tedious, so §3.4.3 shows an alternative.

3.3 Copy and Move

By default, objects can be copied. This is true for objects of user-defined types as well as for built-in types. The default meaning of copy is memberwise copy: copy each member. For example, using `complex` from §3.2.1.1:

```
void test(complex z1)
{
        complex z2 {z1};     // copy initialization
        complex z3;
        z3 = z2;             // copy assignment
        // ...
}
```

Now `z1`, `z2`, and `z3` have the same value because both the assignment and the initialization copied both members.

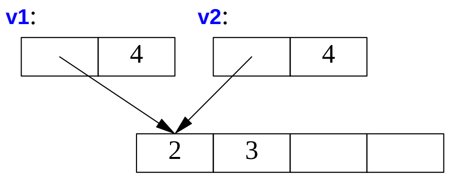
When we design a class, we must always consider if and how an object might be copied. For simple concrete types, memberwise copy is often exactly the right semantics for copy. For some sophisticated concrete types, such as `Vector`, memberwise copy is not the right semantics for copy, and for abstract types it almost never is.

3.3.1 Copying Containers

When a class is a *resource handle*, that is, it is responsible for an object accessed through a pointer, the default memberwise copy is typically a disaster. Memberwise copy would violate the resource handle's invariant (§2.4.3.2). For example, the default copy would leave a copy of a Vector referring to the same elements as the original:

```
void bad_copy(Vector v1)
{
        Vector v2 = v1;          // copy v1's representation into v2
        v1[0] = 2;               // v2[0] is now also 2!
        v2[1] = 3;               // v1[1] is now also 3!
}
```

Assuming that v1 has four elements, the result can be represented graphically like this:



Fortunately, the fact that Vector has a destructor is a strong hint that the default (memberwise) copy semantics is wrong and the compiler should at least warn against this example (§17.6). We need to define better copy semantics.

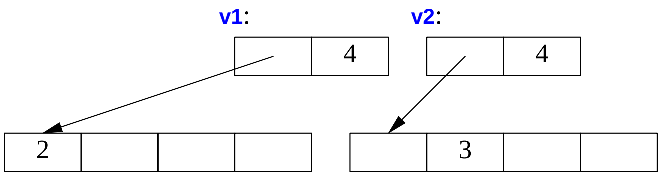
Copying of an object of a class is defined by two members: a *copy constructor* and a *copy assignment*:

```
class Vector {
private:
        double∗ elem;  // elem points to an array of sz doubles
        int sz;
public:
        Vector(int s);                          // constructor: establish invariant, acquire resources
        ˜Vector() { delete[] elem; }            // destructor: release resources

        Vector(const Vector& a);                // copy constructor
        Vector& operator=(const Vector& a);     // copy assignment

        double& operator[](int i);
        const double& operator[](int i) const;

        int size() const;
};
```

A suitable definition of a copy constructor for Vector allocates the space for the required number of elements and then copies the elements into it, so that after a copy each Vector has its own copy of the elements:

```
Vector::Vector(const Vector& a)     // copy constructor
        :elem{new double[sz]},      // allocate space for elements
         sz{a.sz}
{
        for (int i=0; i!=sz; ++i)   // copy elements
                elem[i] = a.elem[i];
}
```

The result of the v2=v1 example can now be presented as:



Of course, we need a copy assignment in addition to the copy constructor:

```
Vector& Vector::operator=(const Vector& a)          // copy assignment
{
        double∗ p = new double[a.sz];
        for (int i=0; i!=a.sz; ++i)
                p[i] = a.elem[i];
        delete[] elem;          // delete old elements
        elem = p;
        sz = a.sz;
        return ∗this;
}
```

The name this is predefined in a member function and points to the object for which the member function is called.

A copy constructor and a copy assignment for a class X are typically declared to take an argument of type const X&.

3.3.2 Moving Containers

We can control copying by defining a copy constructor and a copy assignment, but copying can be costly for large containers. Consider:

```
Vector operator+(const Vector& a, const Vector& b)
{
        if (a.size()!=b.size())
                throw Vector_size_mismatch{};

        Vector res(a.size());
        for (int i=0; i!=a.size(); ++i)
                res[i]=a[i]+b[i];
        return res;
}
```

Returning from a + involves copying the result out of the local variable res and into some place where the caller can access it. We might use this + like this:

```
void f(const Vector& x, const Vector& y, const Vector& z)
{
        Vector r;
        // ...
        r = x+y+z;
        // ...
}
```

That would be copying a Vector at least twice (one for each use of the + operator). If a Vector is large, say, 10,000 doubles, that could be embarrassing. The most embarrassing part is that res in operator+() is never used again after the copy. We didn't really want a copy; we just wanted to get the result out of a function: we wanted to *move* a Vector rather than to *copy* it. Fortunately, we can state that intent:

```
class Vector {
        // ...

        Vector(const Vector& a);                    // copy constructor
        Vector& operator=(const Vector& a);         // copy assignment

        Vector(Vector&& a);                         // move constructor
        Vector& operator=(Vector&& a);              // move assignment
};
```

Given that definition, the compiler will choose the *move constructor* to implement the transfer of the return value out of the function. This means that r=x+y+z will involve no copying of Vectors. Instead, Vectors are just moved.

As is typical, Vector's move constructor is trivial to define:

```
Vector::Vector(Vector&& a)
        :elem{a.elem},          // "grab the elements" from a
         sz{a.sz}
{
        a.elem = nullptr;       // now a has no elements
        a.sz = 0;
}
```

The && means "rvalue reference" and is a reference to which we can bind an rvalue (§6.4.1). The word "rvalue" is intended to complement "lvalue," which roughly means "something that can appear on the left-hand side of an assignment." So an rvalue is – to a first approximation – a value that you can't assign to, such as an integer returned by a function call, and an rvalue reference is a reference to something that nobody else can assign to. The res local variable in operator+() for Vectors is an example.

A move constructor does *not* take a const argument: after all, a move constructor is supposed to remove the value from its argument. A *move assignment* is defined similarly.

A move operation is applied when an rvalue reference is used as an initializer or as the right-hand side of an assignment.

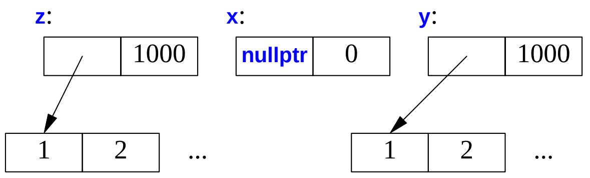
After a move, a moved-from object should be in a state that allows a destructor to be run. Typically, we should also allow assignment to a moved-from object (§17.5, §17.6.2).

Where the programmer knows that a value will not be used again, but the compiler can't be expected to be smart enough to figure that out, the programmer can be specific:

```
Vector f()
{
        Vector x(1000);
        Vector y(1000);
        Vector z(1000);
        // ...
        z = x;                  // we get a copy
        y = std::move(x);       // we get a move
        // ...
        return z;               // we get a move
};
```

The standard-library function `move()` returns an rvalue reference to its argument.

Just before the `return` we have:



When `z` is destroyed, it too has been moved from (by the `return`) so that, like `x`, it is empty (it holds no elements).

3.3.3 Resource Management

By defining constructors, copy operations, move operations, and a destructor, a programmer can provide complete control of the lifetime of a contained resource (such as the elements of a container). Furthermore, a move constructor allows an object to move simply and cheaply from one scope to another. That way, objects that we cannot or would not want to copy out of a scope can be simply and cheaply moved out instead. Consider a standard-library `thread` representing a concurrent activity (§5.3.1) and a `Vector` of a million `double`s. We can't copy the former and don't want to copy the latter.

```
std::vector<thread> my_threads;

Vector init(int n)
{
        thread t {heartbeat};                   // run heartbeat concurrently (on its own thread)
        my_threads.push_back(move(t));          // move t into my_threads
        // ... more initialization ...
        Vector vec(n);
        for (int i=0; i<vec.size(); ++i) vec[i] = 777;
        return vec;                                   // move res out of init()
    }

    auto v = init();  // start heartbeat and initialize v
```

This makes resource handles, such as **Vector** and **thread**, an alternative to using pointers in many cases. In fact, the standard-library ‘‘smart pointers,’’ such as **unique_ptr**, are themselves resource handles (§5.2.1).

I used the standard-library **vector** to hold the **thread**s because we don’t get to parameterize **Vector** with an element type until §3.4.1.

In very much the same way as **new** and **delete** disappear from application code, we can make pointers disappear into resource handles. In both cases, the result is simpler and more maintainable code, without added overhead. In particular, we can achieve *strong resource safety*; that is, we can eliminate resource leaks for a general notion of a resource. Examples are **vector**s holding memory, **thread**s holding system threads, and **fstream**s holding file handles.

3.3.4 Suppressing Operations

Using the default copy or move for a class in a hierarchy is typically a disaster: given only a pointer to a base, we simply don’t know what members the derived class has (§3.2.2), so we can’t know how to copy them. So, the best thing to do is usually to *delete* the default copy and move operations, that is, to eliminate the default definitions of those two operations:

```
class Shape {
public:
        Shape(const Shape&) =delete;                 // no copy operations
        Shape& operator=(const Shape&) =delete;

        Shape(Shape&&) =delete;                      // no move operations
        Shape& operator=(Shape&&) =delete;

        ˜Shape();
        // ...
};
```

Now an attempt to copy a **Shape** will be caught by the compiler. If you need to copy an object in a class hierarchy, write some kind of clone function (§22.2.4).

In this particular case, if you forgot to **delete** a copy or move operation, no harm is done. A move operation is *not* implicitly generated for a class where the user has explicitly declared a destructor. Furthermore, the generation of copy operations is deprecated in this case (§44.2.3). This can be a good reason to explicitly define a destructor even where the compiler would have implicitly provided one (§17.2.3).

A base class in a class hierarchy is just one example of an object we wouldn’t want to copy. A resource handle generally cannot be copied just by copying its members (§5.2, §17.2.2).

The **=delete** mechanism is general, that is, it can be used to suppress any operation (§17.6.4).

3.4 Templates

Someone who wants a vector is unlikely always to want a vector of **doubles**. A vector is a general concept, independent of the notion of a floating-point number. Consequently, the element type of a vector ought to be represented independently. A *template* is a class or a function that we parameterize with a set of types or values. We use templates to represent concepts that are best understood as something very general from which we can generate specific types and functions by specifying arguments, such as the element type **double**.

3.4.1 Parameterized Types

We can generalize our vector-of-doubles type to a vector-of-anything type by making it a **template** and replacing the specific type **double** with a parameter. For example:

```
template<typename T>
class Vector {
private:
        T∗ elem;    // elem points to an array of sz elements of type T
        int sz;
public:
        Vector(int s);                   // constructor: establish invariant, acquire resources
        ˜Vector() { delete[] elem; }     // destructor: release resources

        // ... copy and move operations ...

        T& operator[](int i);
        const T& operator[](int i) const;
        int size() const { return sz; }
};
```

The **template<typename T>** prefix makes **T** a parameter of the declaration it prefixes. It is C++'s version of the mathematical ''for all T'' or more precisely ''for all types T.''

The member functions might be defined similarly:

```
template<typename T>
Vector<T>::Vector(int s)
{
        if (s<0) throw Negative_size{};
        elem = new T[s];
        sz = s;
}

template<typename T>
const T& Vector<T>::operator[](int i) const
{
        if (i<0 || size()<=i)
                throw out_of_range{"Vector::operator[]"};
        return elem[i];
}
```

Given these definitions, we can define Vectors like this:

```
Vector<char> vc(200);          // vector of 200 characters
Vector<string> vs(17);         // vector of 17 strings
Vector<list<int>> vli(45);     // vector of 45 lists of integers
```

The >> in Vector<list<int>> terminates the nested template arguments; it is not a misplaced input operator. It is not (as in C++98) necessary to place a space between the two >s.

We can use Vectors like this:

```
void write(const Vector<string>& vs)           // Vector of some strings
{
     for (int i = 0; i!=vs.size(); ++i)
          cout << vs[i] << '\n';
}
```

To support the range-for loop for our Vector, we must define suitable begin() and end() functions:

```
template<typename T>
T∗ begin(Vector<T>& x)
{
     return &x[0];          // pointer to first element
}

template<typename T>
T∗ end(Vector<T>& x)
{
     return x.begin()+x.size(); // pointer to one-past-last element
}
```

Given those, we can write:

```
void f2(const Vector<string>& vs)   // Vector of some strings
{
     for (auto& s : vs)
          cout << s << '\n';
}
```

Similarly, we can define lists, vectors, maps (that is, associative arrays), etc., as templates (§4.4, §23.2, Chapter 31).

Templates are a compile-time mechanism, so their use incurs no run-time overhead compared to ‘‘handwritten code’’ (§23.2.2).

3.4.2 Function Templates

Templates have many more uses than simply parameterizing a container with an element type. In particular, they are extensively used for parameterization of both types and algorithms in the standard library (§4.4.5, §4.5.5). For example, we can write a function that calculates the sum of the element values of any container like this:

```
template<typename Container, typename Value>
Value sum(const Container& c, Value v)
{
        for (auto x : c)
                v+=x;
        return v;
}
```

The **Value** template argument and the function argument **v** are there to allow the caller to specify the type and initial value of the accumulator (the variable in which to accumulate the sum):

```
void user(Vector<int>& vi, std::list<double>& ld, std::vector<complex<double>>& vc)
{
        int x = sum(vi,0);                    // the sum of a vector of ints (add ints)
        double d = sum(vi,0.0);               // the sum of a vector of ints (add doubles)
        double dd = sum(ld,0.0);              // the sum of a list of doubles
        auto z = sum(vc,complex<double>{});   // the sum of a vector of complex<double>
                                              // the initial value is {0.0,0.0}
}
```

The point of adding **int**s in a **double** would be to gracefully handle a number larger than the largest **int**. Note how the types of the template arguments for **sum<T,V>** are deduced from the function arguments. Fortunately, we do not need to explicitly specify those types.

This **sum()** is a simplified version of the standard-library **accumulate()** (§40.6).

3.4.3 Function Objects

One particularly useful kind of template is the *function object* (sometimes called a *functor*), which is used to define objects that can be called like functions. For example:

```
template<typename T>
class Less_than {
        const T val;     // value to compare against
public:
        Less_than(const T& v) :val(v) { }
        bool operator()(const T& x) const { return x<val; }  // call operator
};
```

The function called **operator()** implements the ‘‘function call,’’ ‘‘call,’’ or ‘‘application’’ operator **()**.

We can define named variables of type **Less_than** for some argument type:

```
Less_than<int> lti {42};               // lti(i) will compare i to 42 using < (i<42)
Less_than<string> lts {"Backus"};      // lts(s) will compare s to "Backus" using < (s<"Backus")
```

We can call such an object, just as we call a function:

```
void fct(int n, const string & s)
{
        bool b1 = lti(n);       // true if n<42
        bool b2 = lts(s);       // true if s<"Backus"
        // ...
}
```

Such function objects are widely used as arguments to algorithms. For example, we can count the occurrences of values for which a predicate returns `true`:

```
template<typename C, typename P>
int count(const C& c, P pred)
{
        int cnt = 0;
        for (const auto& x : c)
                if (pred(x))
                        ++cnt;
        return cnt;
}
```

A *predicate* is something that we can invoke to return `true` or `false`. For example:

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
        cout << "number of values less than " << x
                << ": " << count(vec,Less_than<int>{x})
                << '\n';
        cout << "number of values less than " << s
                << ": " << count(lst,Less_than<string>{s})
                << '\n';
}
```

Here, `Less_than<int>{x}` constructs an object for which the call operator compares to the `int` called `x`; `Less_than<string>{s}` constructs an object that compares to the `string` called `s`. The beauty of these function objects is that they carry the value to be compared against with them. We don't have to write a separate function for each value (and each type), and we don't have to introduce nasty global variables to hold values. Also, for a simple function object like `Less_than` inlining is simple, so that a call of `Less_than` is far more efficient than an indirect function call. The ability to carry data plus their efficiency make function objects particularly useful as arguments to algorithms.

Function objects used to specify the meaning of key operations of a general algorithm (such as `Less_than` for `count()`) are often referred to as *policy objects*.

We have to define `Less_than` separately from its use. That could be seen as inconvenient. Consequently, there is a notation for implicitly generating function objects:

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
        cout << "number of values less than " << x
                << ": " << count(vec,[&](int a){ return a<x; })
                << '\n';
        cout << "number of values less than " << s
                << ": " << count(lst,[&](const string& a){ return a<s; })
                << '\n';
}
```

The notation `[&](int a){ return a<x; }` is called a *lambda expression* (§11.4). It generates a function object exactly like `Less_than<int>{x}`. The `[&]` is a *capture list* specifying that local names used (such as `x`) will be passed by reference. Had we wanted to ''capture'' only `x`, we could have said so: [&x]. Had we wanted to give the generated object a copy of x, we could have said so: [=x]. Capture nothing is [], capture all local names used by reference is [&], and capture all local names used by value is [=].

Using lambdas can be convenient and terse, but also obscure. For nontrivial actions (say, more than a simple expression), I prefer to name the operation so as to more clearly state its purpose and to make it available for use in several places in a program.

In §3.2.4, we noticed the annoyance of having to write many functions to perform operations on elements of vectors of pointers and unique_ptrs, such as draw_all() and rotate_all(). Function objects (in particular, lambdas) can help by allowing us to separate the traversal of the container from the specification of what is to be done with each element.

First, we need a function that applies an operation to each object pointed to by the elements of a container of pointers:

```
template<class C, class Oper>
void for_all(C& c, Oper op)          // assume that C is a container of pointers
{
     for (auto& x : c)
          op(*x);          // pass op() a reference to each element pointed to
}
```

Now, we can write a version of user() from §3.2.4 without writing a set of _all functions:

```
void user()
{
     vector<unique_ptr<Shape>> v;
     while (cin)
          v.push_back(read_shape(cin));
     for_all(v,[](Shape& s){ s.draw(); });          // draw_all()
     for_all(v,[](Shape& s){ s.rotate(45); });      // rotate_all(45)
}
```

I pass a reference to Shape to a lambda so that the lambda doesn't have to care exactly how the objects are stored in the container. In particular, those for_all() calls would still work if I changed v to a vector<Shape*>.

3.4.4 Variadic Templates

A template can be defined to accept an arbitrary number of arguments of arbitrary types. Such a template is called a *variadic template*. For example:

```
template<typename T, typename... Tail>
void f(T head, Tail... tail)
{
     g(head);     // do something to head
     f(tail...);  // try again with tail
}

void f() { }     // do nothing
```

The key to implementing a variadic template is to note that when you pass a list of arguments to it, you can separate the first argument from the rest. Here, we do something to the first argument (the **head**) and then recursively call **f()** with the rest of the arguments (the **tail**). The ellipsis, **...**, is used to indicate ‘‘the rest’’ of a list. Eventually, of course, **tail** will become empty and we need a separate function to deal with that.

We can call this **f()** like this:

```
int main()
{
        cout << "first: ";
        f(1,2.2,"hello");

        cout << "\nsecond: "
        f(0.2,'c',"yuck!",0,1,2);
        cout << "\n";
}
```

This would call **f(1,2.2,"hello")**, which will call **f(2.2,"hello")**, which will call **f("hello")**, which will call **f()**. What might the call **g(head)** do? Obviously, in a real program it will do whatever we wanted done to each argument. For example, we could make it write its argument (here, **head**) to output:

```
template<typename T>
void g(T x)
{
        cout << x << " ";
}
```

Given that, the output will be:

```
first: 1 2.2 hello
second: 0.2 c yuck! 0 1 2
```

It seems that **f()** is a simple variant of **printf()** printing arbitrary lists or values – implemented in three lines of code plus their surrounding declarations.

The strength of variadic templates (sometimes just called *variadics*) is that they can accept any arguments you care to give them. The weakness is that the type checking of the interface is a possibly elaborate template program. For details, see §28.6. For examples, see §34.2.4.2 (N-tuples) and Chapter 29 (N-dimensional matrices).

3.4.5 Aliases

Surprisingly often, it is useful to introduce a synonym for a type or a template (§6.5). For example, the standard header **<cstddef>** contains a definition of the alias **size_t**, maybe:

```
using size_t = unsigned int;
```

The actual type named **size_t** is implementation-dependent, so in another implementation **size_t** may be an **unsigned long**. Having the alias **size_t** allows the programmer to write portable code.

It is very common for a parameterized type to provide an alias for types related to their template arguments. For example:

```
template<typename T>
class Vector {
public:
        using value_type = T;
        // ...
};
```

In fact, every standard-library container provides value_type as the name of its value type (§31.3.1). This allows us to write code that will work for every container that follows this convention. For example:

```
template<typename C>
using Element_type = typename C::value_type;

template<typename Container>
void algo(Container& c)
{
        Vector<Element_type<Container>> vec; // keep results here
        // ...
}
```

The aliasing mechanism can be used to define a new template by binding some or all template arguments. For example:

```
template<typename Key, typename Value>
class Map {
        // ...
};

template<typename Value>
using String_map = Map<string,Value>;

String_map<int> m; // m is a Map<string,int>
```

See §23.6.

3.5 Advice

[1] Express ideas directly in code; §3.2.
[2] Define classes to represent application concepts directly in code; §3.2.
[3] Use concrete classes to represent simple concepts and performance-critical components; §3.2.1.
[4] Avoid ‘‘naked’’ new and delete operations; §3.2.1.2.
[5] Use resource handles and RAII to manage resources; §3.2.1.2.
[6] Use abstract classes as interfaces when complete separation of interface and implementation is needed; §3.2.2.
[7] Use class hierarchies to represent concepts with inherent hierarchical structure; §3.2.4.

[8] When designing a class hierarchy, distinguish between implementation inheritance and interface inheritance; §3.2.4.
[9] Control construction, copy, move, and destruction of objects; §3.3.
[10] Return containers by value (relying on move for efficiency); §3.3.2.
[11] Provide strong resource safety; that is, never leak anything that you think of as a resource; §3.3.3.
[12] Use containers, defined as resource handle templates, to hold collections of values of the same type; §3.4.1.
[13] Use function templates to represent general algorithms; §3.4.2.
[14] Use function objects, including lambdas, to represent policies and actions; §3.4.3.
[15] Use type and template aliases to provide a uniform notation for types that may vary among similar types or among implementations; §3.4.5.

This page intentionally left blank

4

A Tour of C++: Containers and Algorithms

Why waste time learning
when ignorance is instantaneous?
– Hobbes

4.1 Libraries

No significant program is written in just a bare programming language. First, a set of libraries is developed. These then form the basis for further work. Most programs are tedious to write in the bare language, whereas just about any task can be rendered simple by the use of good libraries.

Continuing from Chapters 2 and 3, this chapter and the next give a quick tour of key standard-library facilities. I assume that you have programmed before. If not, please consider reading a textbook, such as *Programming: Principles and Practice Using C++* [Stroustrup,2009], before continuing. Even if you have programmed before, the libraries you used or the applications you wrote may be very different from the style of C++ presented here. If you find this ‘‘lightning tour’’ confusing, you might skip to the more systematic and bottom-up language presentation starting in Chapter 6. Similarly, a more systematic description of the standard library starts in Chapter 30.

I very briefly present useful standard-library types, such as **string**, **ostream**, **vector**, **map** (this chapter), **unique_ptr**, **thread**, **regex**, and **complex** (Chapter 5), as well as the most common ways of using them. Doing this allows me to give better examples in the following chapters. As in Chapter 2 and Chapter 3, you are strongly encouraged not to be distracted or discouraged by an incomplete understanding of details. The purpose of this chapter is to give you a taste of what is to come and to convey a basic understanding of the most useful library facilities.

The specification of the standard library is almost two thirds of the ISO C++ standard. Explore it, and prefer it to home-made alternatives. Much though have gone into its design, more still into its implementations, and much effort will go into its maintenance and extension.

The standard-library facilities described in this book are part of every complete C++ implementation. In addition to the standard-library components, most implementations offer ‘‘graphical user interface’’ systems (GUIs), Web interfaces, database interfaces, etc. Similarly, most application development environments provide ‘‘foundation libraries’’ for corporate or industrial ‘‘standard’’ development and/or execution environments. Here, I do not describe such systems and libraries. The intent is to provide a self-contained description of C++ as defined by the standard and to keep the examples portable, except where specifically noted. Naturally, a programmer is encouraged to explore the more extensive facilities available on most systems.

4.1.1 Standard-Library Overview

The facilities provided by the standard library can be classified like this:

- Run-time language support (e.g., for allocation and run-time type information); see §30.3.
- The C standard library (with very minor modifications to minimize violations of the type system); see Chapter 43.
- Strings and I/O streams (with support for international character sets and localization); see Chapter 36, Chapter 38, and Chapter 39. I/O streams is an extensible framework to which users can add their own streams, buffering strategies, and character sets.
- A framework of containers (such as **vector** and **map**) and algorithms (such as **find()**, **sort()**, and **merge()**); see §4.4, §4.5, Chapters 31-33. This framework, conventionally called the STL [Stepanov,1994], is extensible so users can add their own containers and algorithms.
- Support for numerical computation (such as standard mathematical functions, complex numbers, vectors with arithmetic operations, and random number generators); see §3.2.1.1 and Chapter 40.
- Support for regular expression matching; see §5.5 and Chapter 37.
- Support for concurrent programming, including **thread**s and **lock**s; see §5.3 and Chapter 41. The concurrency support is foundational so that users can add support for new models of concurrency as libraries.
- Utilities to support template metaprogramming (e.g., type traits; §5.4.2, §28.2.4, §35.4), STL-style generic programming (e.g., **pair**; §5.4.3, §34.2.4.1), and general programming (e.g., **clock**; §5.4.1, §35.2).
- ‘‘Smart pointers’’ for resource management (e.g., **unique_ptr** and **shared_ptr**; §5.2.1, §34.3) and an interface to garbage collectors (§34.5).
- Special-purpose containers, such as **array** (§34.2.1), **bitset** (§34.2.2), and **tuple** (§34.2.4.2).

The main criteria for including a class in the library were that:

- it could be helpful to almost every C++ programmer (both novices and experts),
- it could be provided in a general form that did not add significant overhead compared to a simpler version of the same facility, and
- that simple uses should be easy to learn (relative to the inherent complexity of their task).

Essentially, the C++ standard library provides the most common fundamental data structures together with the fundamental algorithms used on them.

4.1.2 The Standard-library Headers and Namespace

Every standard-library facility is provided through some standard header. For example:

```
#include<string>
#include<list>
```

This makes the standard `string` and `list` available.

The standard library is defined in a namespace (§2.4.2, §14.3.1) called `std`. To use standard library facilities, the `std::` prefix can be used:

```
std::string s {"Four legs Good; two legs Baaad!"};
std::list<std::string> slogans {"War is peace", "Freedom is Slavery", "Ignorance is Strength"};
```

For simplicity, I will rarely use the `std::` prefix explicitly in examples. Neither will I always `#include` the necessary headers explicitly. To compile and run the program fragments here, you must `#include` the appropriate headers (as listed in §4.4.5, §4.5.5, and §30.2) and make the names they declare accessible. For example:

```
#include<string>                // make the standard string facilities accessible
using namespace std;             // make std names available without std:: prefix

string s {"C++ is a general−purpose programming language"};      // OK: string is std::string
```

It is generally in poor taste to dump every name from a namespace into the global namespace. However, in this book, I use the standard library almost exclusively and it is good to know what it offers. So, I don't prefix every use of a standard library name with `std::`. Nor do I `#include` the appropriate headers in every example. Assume that done.

Here is a selection of standard-library headers, all supplying declarations in namespace `std`:

Selected Standard Library Headers (continues)

`<algorithm>`	`copy()`, `find()`, `sort()`	§32.2	§iso.25
`<array>`	`array`	§34.2.1	§iso.23.3.2
`<chrono>`	`duration`, `time_point`	§35.2	§iso.20.11.2
`<cmath>`	`sqrt()`, `pow()`	§40.3	§iso.26.8
`<complex>`	`complex`, `sqrt()`, `pow()`	§40.4	§iso.26.8
`<fstream>`	`fstream`, `ifstream`, `ofstream`	§38.2.1	§iso.27.9.1
`<future>`	`future`, `promise`	§5.3.5	§iso.30.6
`<iostream>`	`istream`, `ostream`, `cin`, `cout`	§38.1	§iso.27.4

Selected Standard Library Headers (continued)			
`<map>`	`map`, `multimap`	§31.4.3	§iso.23.4.4
`<memory>`	`unique_ptr`, `shared_ptr`, `allocator`	§5.2.1	§iso.20.6
`<random>`	`default_random_engine`, `normal_distribution`	§40.7	§iso.26.5
`<regex>`	`regex`, `smatch`	Chapter 37	§iso.28.8
`<string>`	`string`, `basic_string`	Chapter 36	§iso.21.3
`<set>`	`set`, `multiset`	§31.4.3	§iso.23.4.6
`<sstream>`	`istrstream`, `ostrstream`	§38.2.2	§iso.27.8
`<thread>`	`thread`	§5.3.1	§iso.30.3
`<unordered_map>`	`unordered_map`, `unordered_multimap`	§31.4.3.2	§iso.23.5.4
`<utility>`	`move()`, `swap()`, `pair`	§35.5	§iso.20.1
`<vector>`	`vector`	§31.4	§iso.23.3.6

This listing is far from complete; see §30.2 for more information.

4.2 Strings

The standard library provides a `string` type to complement the string literals. The `string` type provides a variety of useful string operations, such as concatenation. For example:

```
string compose(const string& name, const string& domain)
{
      return name + '@' + domain;
}

auto addr = compose("dmr","bell–labs.com");
```

Here, `addr` is initialized to the character sequence `dmr@bell–labs.com`. ‘‘Addition’’ of strings means concatenation. You can concatenate a `string`, a string literal, a C-style string, or a character to a `string`. The standard `string` has a move constructor so returning even long `string`s by value is efficient (§3.3.2).

In many applications, the most common form of concatenation is adding something to the end of a `string`. This is directly supported by the `+=` operation. For example:

```
void m2(string& s1, string& s2)
{
      s1 = s1 + '\n';    // append newline
      s2 += '\n';        // append newline
}
```

The two ways of adding to the end of a `string` are semantically equivalent, but I prefer the latter because it is more explicit about what it does, more concise, and possibly more efficient.

A `string` is mutable. In addition to `=` and `+=`, subscripting (using `[]`) and substring operations are supported. The standard-library `string` is described in Chapter 36. Among other useful features, it provides the ability to manipulate substrings. For example:

```
string name = "Niels Stroustrup";

void m3()
{
        string s = name.substr(6,10);          // s = "Stroustrup"
        name.replace(0,5,"nicholas");          // name becomes "nicholas Stroustrup"
        name[0] = toupper(name[0]);            // name becomes "Nicholas Stroustrup"
}
```

The `substr()` operation returns a `string` that is a copy of the substring indicated by its arguments. The first argument is an index into the `string` (a position), and the second is the length of the desired substring. Since indexing starts from `0`, `s` gets the value `Stroustrup`.

The `replace()` operation replaces a substring with a value. In this case, the substring starting at `0` with length `5` is `Niels`; it is replaced by `nicholas`. Finally, I replace the initial character with its uppercase equivalent. Thus, the final value of `name` is `Nicholas Stroustrup`. Note that the replacement string need not be the same size as the substring that it is replacing.

Naturally, `string`s can be compared against each other and against string literals. For example:

```
string incantation;

void respond(const string& answer)
{
        if (answer == incantation) {
                // perform magic
        }
        else if (answer == "yes") {
                // ...
        }
        // ...
}
```

The `string` library is described in Chapter 36. The most common techniques for implementing `string` are presented in the `String` example (§19.3).

4.3 Stream I/O

The standard library provides formatted character input and output through the `iostream` library. The input operations are typed and extensible to handle user-defined types. This section is a very brief introduction to the use of `iostream`s; Chapter 38 is a reasonably complete description of the `iostream` library facilities.

Other forms of user interaction, such as graphical I/O, are handled through libraries that are not part of the ISO standard and therefore not described here.

4.3.1 Output

The I/O stream library defines output for every built-in type. Further, it is easy to define output of a user-defined type (§4.3.3). The operator `<<` (“put to”) is used as an output operator on objects of type **ostream**; **cout** is the standard output stream and **cerr** is the standard stream for reporting errors. By default, values written to **cout** are converted to a sequence of characters. For example, to output the decimal number **10**, we can write:

```
void f()
{
        cout << 10;
}
```

This places the character **1** followed by the character **0** on the standard output stream.

Equivalently, we could write:

```
void g()
{
        int i {10};
        cout << i;
}
```

Output of different types can be combined in the obvious way:

```
void h(int i)
{
        cout << "the value of i is ";
        cout << i;
        cout << '\n';
}
```

For **h(10)**, the output will be:

```
the value of i is 10
```

People soon tire of repeating the name of the output stream when outputting several related items. Fortunately, the result of an output expression can itself be used for further output. For example:

```
void h2(int i)
{
        cout << "the value of i is " << i << '\n';
}
```

This **h2()** produces the same output as **h()**.

A character constant is a character enclosed in single quotes. Note that a character is output as a character rather than as a numerical value. For example:

```
void k()
{
        int b = 'b';        // note: char implicitly converted to int
        char c = 'c';
        cout << 'a' << b << c;
}
```

The integer value of the character **'b'** is **98** (in the ASCII encoding used on the C++ implementation that I used), so this will output **a98c**.

4.3.2 Input

The standard library offers `istream`s for input. Like `ostream`s, `istream`s deal with character string representations of built-in types and can easily be extended to cope with user-defined types.

The operator `>>` (''get from'') is used as an input operator; `cin` is the standard input stream. The type of the right-hand operand of `>>` determines what input is accepted and what is the target of the input operation. For example:

```
void f()
{
        int i;
        cin >> i;          // read an integer into i

        double d;
        cin >> d;          // read a double-precision floating-point number into d
}
```

This reads a number, such as `1234`, from the standard input into the integer variable `i` and a floating-point number, such as `12.34e5`, into the double-precision floating-point variable `d`.

Often, we want to read a sequence of characters. A convenient way of doing that is to read into a `string`. For example:

```
void hello()
{
        cout << "Please enter your name\n";
        string str;
        cin >> str;
        cout << "Hello, " << str << "!\n";
}
```

If you type in `Eric` the response is:

```
Hello, Eric!
```

By default, a whitespace character (§7.3.2), such as a space, terminates the read, so if you enter `Eric Bloodaxe` pretending to be the ill-fated king of York, the response is still:

```
Hello, Eric!
```

You can read a whole line (including the terminating newline character) using the `getline()` function. For example:

```
void hello_line()
{
        cout << "Please enter your name\n";
        string str;
        getline(cin,str);
        cout << "Hello, " << str << "!\n";
}
```

With this program, the input `Eric Bloodaxe` yields the desired output:

```
Hello, Eric Bloodaxe!
```

The newline that terminated the line is discarded, so cin is ready for the next input line.

The standard strings have the nice property of expanding to hold what you put in them; you don't have to precalculate a maximum size. So, if you enter a couple of megabytes of semicolons, the program will echo pages of semicolons back at you.

4.3.3 I/O of User-Defined Types

In addition to the I/O of built-in types and standard strings, the iostream library allows programmers to define I/O for their own types. For example, consider a simple type Entry that we might use to represent entries in a telephone book:

```
struct Entry {
        string name;
        int number;
};
```

We can define a simple output operator to write an Entry using a *{"name",number}* format similar to the one we use for initialization in code:

```
ostream& operator<<(ostream& os, const Entry& e)
{
        return os << "{\"" << e.name << "\", " << e.number << "}";
}
```

A user-defined output operator takes its output stream (by reference) as its first argument and returns it as its result. See §38.4.2 for details.

The corresponding input operator is more complicated because it has to check for correct formatting and deal with errors:

```
istream& operator>>(istream& is, Entry& e)
        // read { "name" , number } pair. Note: formatted with { " " , and }
{
        char c, c2;
        if (is>>c && c=='{' && is>>c2 && c2=='"') { // start with a { "
                string name;                    // the default value of a string is the empty string: ""
                while (is.get(c) && c!='"')     // anything before a " is part of the name
                        name+=c;

                if (is>>c && c==',') {
                        int number = 0;
                        if (is>>number>>c && c=='}') { // read the number and a }
                                e = {name,number};     // assign to the entry
                                return is;
                        }
                }
        }
        is.setf(ios_base::failbit);         // register the failure in the stream
        return is;
}
```

An input operation returns a reference to its istream which can be used to test if the operation succeeded. For example, when used as a condition, is>>c means ‘‘Did we succeed at reading from is into c?’’

The is>>c skips whitespace by default, but is.get(c) does not, so that this Entry-input operator ignores (skips) whitespace outside the name string, but not within it. For example:

```
{ "John Marwood Cleese" , 123456        }
{"Michael Edward Palin",987654}
```

We can read such a pair of values from input into an Entry like this:

```
for (Entry ee; cin>>ee; )  // read from cin into ee
        cout << ee << '\n';  // write ee to cout
```

The output is:

```
{"John Marwood Cleese", 123456}
{"Michael Edward Palin", 987654}
```

See §38.4.1 for more technical details and techniques for writing input operators for user-defined types. See §5.5 and Chapter 37 for a more systematic technique for recognizing patterns in streams of characters (regular expression matching).

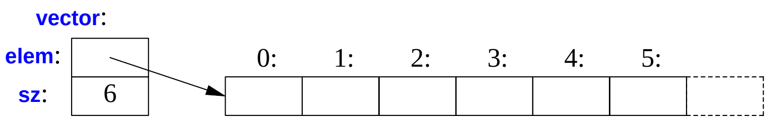
4.4 Containers

Most computing involves creating collections of values and then manipulating such collections. Reading characters into a string and printing out the string is a simple example. A class with the main purpose of holding objects is commonly called a *container*. Providing suitable containers for a given task and supporting them with useful fundamental operations are important steps in the construction of any program.

To illustrate the standard-library containers, consider a simple program for keeping names and telephone numbers. This is the kind of program for which different approaches appear ‘‘simple and obvious’’ to people of different backgrounds. The Entry class from §4.3.3 can be used to hold a simple phone book entry. Here, we deliberately ignore many real-world complexities, such as the fact that many phone numbers do not have a simple representation as a 32-bit int.

4.4.1 vector

The most useful standard-library container is vector. A vector is a sequence of elements of a given type. The elements are stored contiguously in memory:



The Vector examples in §3.2.2 and §3.4 give an idea of the implementation of vector and §13.6 and §31.4 provide an exhaustive discussion.

We can initialize a **vector** with a set of values of its element type:

```
vector<Entry> phone_book = {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Arthur William Russell",345678}
};
```

Elements can be accessed through subscripting:

```
void print_book(const vector<Entry>& book)
{
        for (int i = 0; i!=book.size(); ++i)
                cout << book[i] << '\n';
}
```

As usual, indexing starts at **0** so that **book[0]** holds the entry for **David Hume**. The **vector** member function **size()** gives the number of elements.

The elements of a **vector** constitute a range, so we can use a range-**for** loop (§2.2.5):

```
void print_book(const vector<Entry>& book)
{
        for (const auto& x : book)       // for "auto" see §2.2.2
                cout << x << '\n';
}
```

When we define a **vector**, we give it an initial size (initial number of elements):

```
vector<int> v1 = {1, 2, 3, 4};          // size is 4
vector<string> v2;                      // size is 0
vector<Shape*> v3(23);                  // size is 23; initial element value: nullptr
vector<double> v4(32,9.9);              // size is 32; initial element value: 9.9
```

An explicit size is enclosed in ordinary parentheses, for example, **(23)**, and by default the elements are initialized to the element type's default value (e.g., **nullptr** for pointers and **0** for numbers). If you don't want the default value, you can specify one as a second argument (e.g., **9.9** for the **32** elements of **v4**).

The initial size can be changed. One of the most useful operations on a **vector** is **push_back()**, which adds a new element at the end of a **vector**, increasing its size by one. For example:

```
void input()
{
        for (Entry e; cin>>e;)
                phone_book.push_back(e);
}
```

This reads **Entry**s from the standard input into **phone_book** until either the end-of-input (e.g., the end of a file) is reached or the input operation encounters a format error. The standard-library **vector** is implemented so that growing a **vector** by repeated **push_back()**s is efficient.

A **vector** can be copied in assignments and initializations. For example:

```
vector<Entry> book2 = phone_book;
```

Copying and moving of vectors are implemented by constructors and assignment operators as described in §3.3. Assigning a vector involves copying its elements. Thus, after the initialization of book2, book2 and phone_book hold separate copies of every Entry in the phone book. When a vector holds many elements, such innocent-looking assignments and initializations can be expensive. Where copying is undesirable, references or pointers (§7.2, §7.7) or move operations (§3.3.2, §17.5.2) should be used.

4.4.1.1 Elements

Like all standard-library containers, vector is a container of elements of some type T, that is, a vector<T>. Just about any type qualifies as an element type: built-in numeric types (such as char, int, and double), user-defined types (such as string, Entry, list<int>, and Matrix<double,2>), and pointers (such as const char*, Shape*, and double*). When you insert a new element, its value is copied into the container. For example, when you put an integer with the value 7 into a container, the resulting element really has the value 7. The element is not a reference or a pointer to some object containing 7. This makes for nice compact containers with fast access. For people who care about memory sizes and run-time performance this is critical.

4.4.1.2 Range Checking

The standard-library vector does not guarantee range checking (§31.2.2). For example:

```
void silly(vector<Entry>& book)
{
        int i = book[ph.size()].number;         // book.size() is out of range
        // ...
}
```

That initialization is likely to place some random value in i rather than giving an error. This is undesirable, and out-of-range errors are a common problem. Consequently, I often use a simple range-checking adaptation of vector:

```
template<typename T>
class Vec : public std::vector<T> {
public:
        using vector<T>::vector; // use the constructors from vector (under the name Vec); see §20.3.5.1

        T& operator[](int i)                    // range check
                { return vector<T>::at(i); }

        const T& operator[](int i) const            // range check const objects; §3.2.1.1
                { return vector<T>::at(i); }
};
```

Vec inherits everything from vector except for the subscript operations that it redefines to do range checking. The at() operation is a vector subscript operation that throws an exception of type out_of_range if its argument is out of the vector's range (§2.4.3.1, §31.2.2).

For Vec, an out-of-range access will throw an exception that the user can catch. For example:

```
void checked(Vec<Entry>& book)
{
        try {
                book[book.size()] = {"Joe",999999};        // will throw an exception
                // ...
        }
        catch (out_of_range) {
                cout << "range error\n";
        }
}
```

The exception will be thrown, and then caught (§2.4.3.1, Chapter 13). If the user doesn't catch an exception, the program will terminate in a well-defined manner rather than proceeding or failing in an undefined manner. One way to minimize surprises from uncaught exceptions is to use a main() with a try-block as its body. For example:

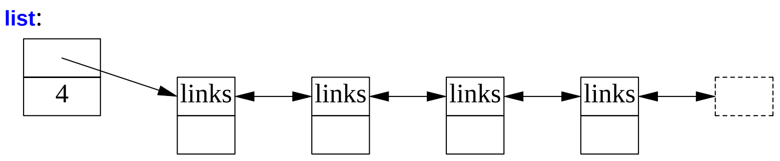
```
int main()
try {
        // your code
}
catch (out_of_range) {
        cerr << "range error\n";
}
catch (...) {
        cerr << "unknown exception thrown\n";
}
```

This provides default exception handlers so that if we fail to catch some exception, an error message is printed on the standard error-diagnostic output stream cerr (§38.1).

Some implementations save you the bother of defining Vec (or equivalent) by providing a range-checked version of vector (e.g., as a compiler option).

4.4.2 list

The standard library offers a doubly-linked list called list:



We use a list for sequences where we want to insert and delete elements without moving other elements. Insertion and deletion of phone book entries could be common, so a list could be appropriate for representing a simple phone book. For example:

```
list<Entry> phone_book = {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Arthur William Russell",345678}
};
```

When we use a linked list, we tend not to access elements using subscripting the way we commonly do for vectors. Instead, we might search the list looking for an element with a given value. To do this, we take advantage of the fact that a list is a sequence as described in §4.5:

```
int get_number(const string& s)
{
        for (const auto& x : phone_book)
                if (x.name==s)
                        return x.number;
        return 0;   // use 0 to represent "number not found"
}
```

The search for s starts at the beginning of the list and proceeds until s is found or the end of phone_book is reached.

Sometimes, we need to identify an element in a list. For example, we may want to delete it or insert a new entry before it. To do that we use an *iterator*: a list iterator identifies an element of a list and can be used to iterate through a list (hence its name). Every standard-library container provides the functions begin() and end(), which return an iterator to the first and to one-past-the-last element, respectively (§4.5, §33.1.1). Using iterators explicitly, we can – less elegantly – write the get_number() function like this:

```
int get_number(const string& s)
{
        for (auto p = phone_book.begin(); p!=phone_book.end(); ++p)
                if (p–>name==s)
                        return p–>number;
        return 0;   // use 0 to represent "number not found"
}
```

In fact, this is roughly the way the terser and less error-prone range-for loop is implemented by the compiler. Given an iterator p, ∗p is the element to which it refers, ++p advances p to refer to the next element, and when p refers to a class with a member m, then p–>m is equivalent to (∗p).m.

Adding elements to a list and removing elements from a list is easy:

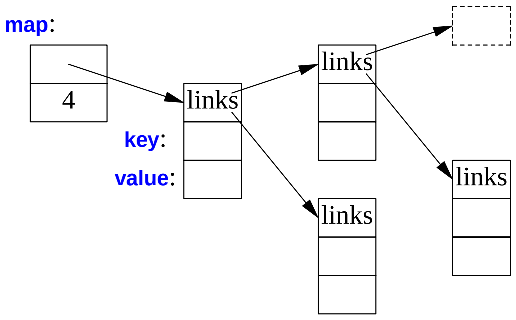
```
void f(const Entry& ee, list<Entry>::iterator p, list<Entry>::iterator q)
{
        phone_book.insert(p,ee);        // add ee before the element referred to by p
        phone_book.erase(q);            // remove the element referred to by q
}
```

For a more complete description of insert() and erase(), see §31.3.7.

These list examples could be written identically using vector and (surprisingly, unless you understand machine architecture) perform better with a small vector than with a small list. When all we want is a sequence of elements, we have a choice between using a vector and a list. Unless you have a reason not to, use a vector. A vector performs better for traversal (e.g., find() and count()) and for sorting and searching (e.g., sort() and binary_search()).

4.4.3 map

Writing code to look up a name in a list of *(name,number)* pairs is quite tedious. In addition, a linear search is inefficient for all but the shortest lists. The standard library offers a search tree (a red-black tree) called map:



In other contexts, a map is known as an associative array or a dictionary. It is implemented as a balanced binary tree.

The standard-library map (§31.4.3) is a container of pairs of values optimized for lookup. We can use the same initializer as for vector and list (§4.4.1, §4.4.2):

```
map<string,int> phone_book {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Arthur William Russell",345678}
};
```

When indexed by a value of its first type (called the *key*), a map returns the corresponding value of the second type (called the *value* or the *mapped type*). For example:

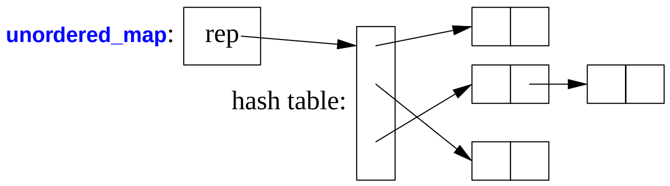
```
int get_number(const string& s)
{
        return phone_book[s];
}
```

In other words, subscripting a map is essentially the lookup we called get_number(). If a key isn't found, it is entered into the map with a default value for its value. The default value for an integer type is 0; the value I just happened to choose represents an invalid telephone number.

If we wanted to avoid entering invalid numbers into our phone book, we could use find() and insert() instead of [] (§31.4.3.1).

4.4.4 unordered_map

The cost of a map lookup is O(log(n)) where n is the number of elements in the map. That's pretty good. For example, for a map with 1,000,000 elements, we perform only about 20 comparisons and indirections to find an element. However, in many cases, we can do better by using a hashed lookup rather than comparison using an ordering function, such as <. The standard-library hashed containers are referred to as “unordered” because they don’t require an ordering function:



For example, we can use an **unordered_map** from **<unordered_map>** for our phone book:

```
unordered_map<string,int> phone_book {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Arthur William Russell",345678}
};
```

As for a **map**, we can subscript an **unordered_map**:

```
int get_number(const string& s)
{
        return phone_book[s];
}
```

The standard-library **unordered_map** provides a default hash function for **string**s. If necessary, you can provide your own (§31.4.3.4).

4.4.5 Container Overview

The standard library provides some of the most general and useful container types to allow the programmer to select a container that best serves the needs of an application:

Standard Container Summary

vector<T>	A variable-size vector (§31.4)
list<T>	A doubly-linked list (§31.4.2)
forward_list<T>	A singly-linked list (§31.4.2)
deque<T>	A double-ended queue (§31.2)
set<T>	A set (§31.4.3)
multiset<T>	A set in which a value can occur many times (§31.4.3)
map<K,V>	An associative array (§31.4.3)
multimap<K,V>	A map in which a key can occur many times (§31.4.3)
unordered_map<K,V>	A map using a hashed lookup (§31.4.3.2)
unordered_multimap<K,V>	A multimap using a hashed lookup (§31.4.3.2)
unordered_set<T>	A set using a hashed lookup (§31.4.3.2)
unordered_multiset<T>	A multiset using a hashed lookup (§31.4.3.2)

The unordered containers are optimized for lookup with a key (often a string); in other words, they are implemented using hash tables.

The standard containers are described in §31.4. The containers are defined in namespace **std** and presented in headers **<vector>**, **<list>**, **<map>**, etc. (§4.1.2, §30.2). In addition, the standard library provides container adaptors **queue<T>** (§31.5.2), **stack<T>** (§31.5.1), **deque<T>** (§31.4), and **priority_queue<T>** (§31.5.3). The standard library also provides more specialized container-like types, such as a fixed-size array **array<T,N>** (§34.2.1) and **bitset<N>** (§34.2.2).

The standard containers and their basic operations are designed to be similar from a notational point of view. Furthermore, the meanings of the operations are equivalent for the various containers. Basic operations apply to every kind of container for which they make sense and can be efficiently implemented. For example:

- **begin()** and **end()** give iterators to the first and one-beyond-the-last elements, respectively.
- **push_back()** can be used (efficiently) to add elements to the end of a **vector**, **forward_list**, **list**, and other containers.
- **size()** returns the number of elements.

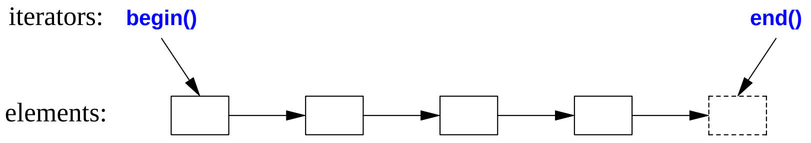
This notational and semantic uniformity enables programmers to provide new container types that can be used in a very similar manner to the standard ones. The range-checked vector, **Vector** (§2.3.2, §2.4.3.1), is an example of that. The uniformity of container interfaces also allows us to specify algorithms independently of individual container types. However, each has strengths and weaknesses. For example, subscripting and traversing a **vector** is cheap and easy. On the other hand, **vector** elements are moved when we insert or remove elements; **list** has exactly the opposite properties. Please note that a **vector** is usually more efficient than a **list** for short sequences of small elements (even for **insert()** and **erase()**). I recommend the standard-library **vector** as the default type for sequences of elements: you need a reason to choose another.

4.5 Algorithms

A data structure, such as a list or a vector, is not very useful on its own. To use one, we need operations for basic access such as adding and removing elements (as is provided for **list** and **vector**). Furthermore, we rarely just store objects in a container. We sort them, print them, extract subsets, remove elements, search for objects, etc. Consequently, the standard library provides the most common algorithms for containers in addition to providing the most common container types. For example, the following sorts a **vector** and places a copy of each unique **vector** element on a **list**:

```
bool operator<(const Entry& x, const Entry& y)    // less than
{
        return x.name<y.name;          // order Entrys by their names
}

void f(vector<Entry>& vec, list<Entry>& lst)
{
        sort(vec.begin(),vec.end());                     // use < for order
        unique_copy(vec.begin(),vec.end(),lst.begin());  // don't copy adjacent equal elements
}
```

The standard algorithms are described in Chapter 32. They are expressed in terms of sequences of elements. A *sequence* is represented by a pair of iterators specifying the first element and the one-beyond-the-last element:



In the example, sort() sorts the sequence defined by the pair of iterators vec.begin() and vec.end() – which just happens to be all the elements of a vector. For writing (output), you need only to specify the first element to be written. If more than one element is written, the elements following that initial element will be overwritten. Thus, to avoid errors, lst must have at least as many elements as there are unique values in vec.

If we wanted to place the unique elements in a new container, we could have written:

```
list<Entry> f(vector<Entry>& vec)
{
        list<Entry> res;
        sort(vec.begin(),vec.end());
        unique_copy(vec.begin(),vec.end(),back_inserter(res)); // append to res
        return res;
}
```

A back_inserter() adds elements at the end of a container, extending the container to make room for them (§33.2.2). Thus, the standard containers plus back_inserter()s eliminate the need to use error-prone, explicit C-style memory management using realloc() (§31.5.1). The standard-library list has a move constructor (§3.3.2, §17.5.2) that makes returning res by value efficient (even for lists of thousands of elements).

If you find the pair-of-iterators style of code, such as sort(vec.begin(),vec.end()), tedious, you can define container versions of the algorithms and write sort(vec) (§4.5.6).

4.5.1 Use of Iterators

When you first encounter a container, a few iterators referring to useful elements can be obtained; begin() and end() are the best examples of this. In addition, many algorithms return iterators. For example, the standard algorithm find looks for a value in a sequence and returns an iterator to the element found:

```
bool has_c(const string& s, char c)        // does s contain the character c?
{
        auto p = find(s.begin(),s.end(),c);
        if (p!=s.end())
                return true;
        else
                return false;
}
```

Like many standard-library search algorithms, find returns end() to indicate “not found.” An equivalent, shorter, definition of has_c() is:

```
bool has_c(const string& s, char c)      // does s contain the character c?
{
        return find(s.begin(),s.end(),c)!=s.end();
}
```

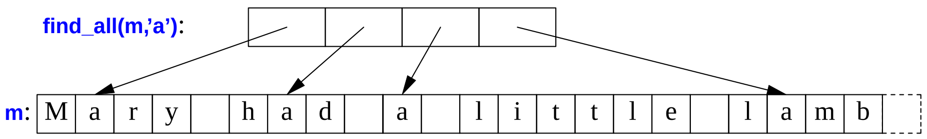
A more interesting exercise would be to find the location of all occurrences of a character in a string. We can return the set of occurrences as a **vector** of **string** iterators. Returning a **vector** is efficient because of **vector** provides move semantics (§3.3.1). Assuming that we would like to modify the locations found, we pass a non-**const** string:

```
vector<string::iterator> find_all(string& s, char c)      // find all occurrences of c in s
{
        vector<string::iterator> res;
        for (auto p = s.begin(); p!=s.end(); ++p)
                if (*p==c)
                        res.push_back(p);
        return res;
}
```

We iterate through the string using a conventional loop, moving the iterator **p** forward one element at a time using **++** and looking at the elements using the dereference operator *. We could test **find_all()** like this:

```
void test()
{
        string m {"Mary had a little lamb"};
        for (auto p : find_all(m,'a'))
                if (*p!='a')
                        cerr << "a bug!\n";
}
```

That call of **find_all()** could be graphically represented like this:



Iterators and standard algorithms work equivalently on every standard container for which their use makes sense. Consequently, we could generalize **find_all()**:

```
template<typename C, typename V>
vector<typename C::iterator> find_all(C& c, V v)      // find all occurrences of v in c
{
        vector<typename C::iterator> res;
        for (auto p = c.begin(); p!=c.end(); ++p)
                if (*p==v)
                        res.push_back(p);
        return res;
}
```

The typename is needed to inform the compiler that C's iterator is supposed to be a type and not a value of some type, say, the integer 7. We can hide this implementation detail by introducing a type alias (§3.4.5) for Iterator:

```
template<typename T>
using Iterator<T> = typename T::iterator;

template<typename C, typename V>
vector<Iterator<C>> find_all(C& c, V v)          // find all occurrences of v in c
{
        vector<Iterator<C>> res;
        for (auto p = c.begin(); p!=c.end(); ++p)
                if (∗p==v)
                        res.push_back(p);
        return res;
}
```

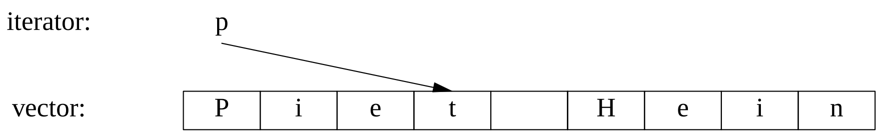
We can now write:

```
void test()
{
        string m {"Mary had a little lamb"};
        for (auto p : find_all(m,'a'))                // p is a string::iterator
                if (∗p!='a')
                        cerr << "string bug!\n";

        list<double> ld {1.1, 2.2, 3.3, 1.1};
        for (auto p : find_all(ld,1.1))
                if (∗p!=1.1)
                        cerr << "list bug!\n";

        vector<string> vs { "red", "blue", "green", "green", "orange", "green" };
        for (auto p : find_all(vs,"green"))
                if (∗p!="green")
                        cerr << "vector bug!\n";

        for (auto p : find_all(vs,"green"))
                ∗p = "vert";
}
```

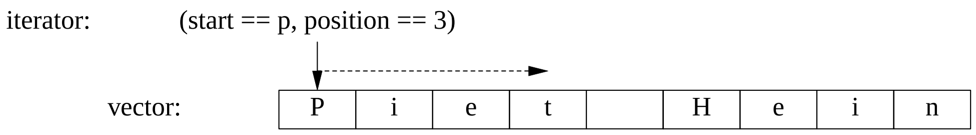
Iterators are used to separate algorithms and containers. An algorithm operates on its data through iterators and knows nothing about the container in which the elements are stored. Conversely, a container knows nothing about the algorithms operating on its elements; all it does is to supply iterators upon request (e.g., begin() and end()). This model of separation between data storage and algorithm delivers very general and flexible software.

4.5.2 Iterator Types

What are iterators really? Any particular iterator is an object of some type. There are, however, many different iterator types, because an iterator needs to hold the information necessary for doing its job for a particular container type. These iterator types can be as different as the containers and the specialized needs they serve. For example, a vector's iterator could be an ordinary pointer, because a pointer is quite a reasonable way of referring to an element of a vector:

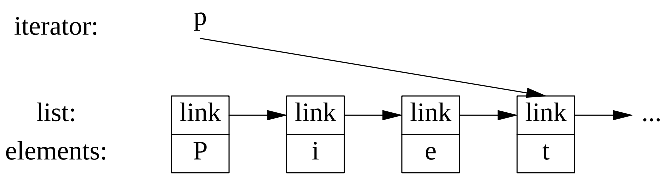


Alternatively, a vector iterator could be implemented as a pointer to the vector plus an index:



Using such an iterator would allow range checking.

A list iterator must be something more complicated than a simple pointer to an element because an element of a list in general does not know where the next element of that list is. Thus, a list iterator might be a pointer to a link:



What is common for all iterators is their semantics and the naming of their operations. For example, applying ++ to any iterator yields an iterator that refers to the next element. Similarly, * yields the element to which the iterator refers. In fact, any object that obeys a few simple rules like these is an iterator (§33.1.4). Furthermore, users rarely need to know the type of a specific iterator; each container "knows" its iterator types and makes them available under the conventional names iterator and const_iterator. For example, list<Entry>::iterator is the general iterator type for list<Entry>. We rarely have to worry about the details of how that type is defined.

4.5.3 Stream Iterators

Iterators are a general and useful concept for dealing with sequences of elements in containers. However, containers are not the only place where we find sequences of elements. For example, an input stream produces a sequence of values, and we write a sequence of values to an output stream. Consequently, the notion of iterators can be usefully applied to input and output.

To make an ostream_iterator, we need to specify which stream will be used and the type of objects written to it. For example:

```
ostream_iterator<string> oo {cout};        // write strings to cout
```

The effect of assigning to ∗oo is to write the assigned value to cout. For example:

```
int main()
{
        ∗oo = "Hello, ";        // meaning cout<<"Hello, "
        ++oo;
        ∗oo = "world!\n";       // meaning cout<<"world!\n"
}
```

This is yet another way of writing the canonical message to standard output. The ++oo is done to mimic writing into an array through a pointer.

Similarly, an istream_iterator is something that allows us to treat an input stream as a read-only container. Again, we must specify the stream to be used and the type of values expected:

```
istream_iterator<string> ii {cin};
```

Input iterators are used in pairs representing a sequence, so we must provide an istream_iterator to indicate the end of input. This is the default istream_iterator:

```
istream_iterator<string> eos {};
```

Typically, istream_iterators and ostream_iterators are not used directly. Instead, they are provided as arguments to algorithms. For example, we can write a simple program to read a file, sort the words read, eliminate duplicates, and write the result to another file:

```
int main()
{
        string from, to;
        cin >> from >> to;                          // get source and target file names

        ifstream is {from};                         // input stream for file "from"
        istream_iterator<string> ii {is};           // input iterator for stream
        istream_iterator<string> eos {};            // input sentinel

        ofstream os{to};                            // output stream for file "to"
        ostream_iterator<string> oo {os,"\n"};      // output iterator for stream

        vector<string> b {ii,eos};                  // b is a vector initialized from input [ii:eos)
        sort(b.begin(),b.end());                    // sort the buffer

        unique_copy(b.begin(),b.end(),oo);          // copy buffer to output, discard replicated values

        return !is.eof() || !os;                    // return error state (§2.2.1, §38.3)
}
```

An ifstream is an istream that can be attached to a file, and an ofstream is an ostream that can be attached to a file. The ostream_iterator's second argument is used to delimit output values.

Actually, this program is longer than it needs to be. We read the strings into a vector, then we sort() them, and then we write them out, eliminating duplicates. A more elegant solution is not to store duplicates at all. This can be done by keeping the strings in a set, which does not keep duplicates and keeps its elements in order (§31.4.3). That way, we could replace the two lines using a vector with one using a set and replace unique_copy() with the simpler copy():

```
    set<string> b {ii,eos};           // collect strings from input
    copy(b.begin(),b.end(),oo);       // copy buffer to output
```

We used the names ii, eos, and oo only once, so we could further reduce the size of the program:

```
int main()
{
    string from, to;
    cin >> from >> to;             // get source and target file names

    ifstream is {from};            // input stream for file "from"
    ofstream os {to};              // output stream for file "to"

    set<string> b {istream_iterator<string>{is},istream_iterator<string>{}}; // read input
    copy(b.begin(),b.end(),ostream_iterator<string>{os,"\n"});              // copy to output

    return !is.eof() || !os;             // return error state (§2.2.1, §38.3)
}
```

It is a matter of taste and experience whether or not this last simplification improves readability.

4.5.4 Predicates

In the examples above, the algorithms have simply ‘‘built in’’ the action to be done for each element of a sequence. However, we often want to make that action a parameter to the algorithm. For example, the find algorithm (§32.4) provides a convenient way of looking for a specific value. A more general variant looks for an element that fulfills a specified requirement, a *predicate* (§3.4.2). For example, we might want to search a map for the first value larger than 42. A map allows us to access its elements as a sequence of *(key,value)* pairs, so we can search a map<string,int>’s sequence for a pair<const string,int> where the int is greater than 42:

```
void f(map<string,int>& m)
{
    auto p = find_if(m.begin(),m.end(),Greater_than{42});
    // ...
}
```

Here, Greater_than is a function object (§3.4.3) holding the value (42) to be compared against:

```
struct Greater_than {
    int val;
    Greater_than(int v) : val{v} { }
    bool operator()(const pair<string,int>& r) { return r.second>val; }
};
```

Alternatively, we could use a lambda expression (§3.4.3):

```
int cxx = count_if(m.begin(), m.end(), [](const pair<string,int>& r) { return r.second>42; });
```

4.5.5 Algorithm Overview

A general definition of an algorithm is ‘‘a finite set of rules which gives a sequence of operations for solving a specific set of problems [and] has five important features: Finiteness ... Definiteness ... Input ... Output ... Effectiveness’’ [Knuth,1968,§1.1]. In the context of the C++ standard library, an algorithm is a function template operating on sequences of elements.

The standard library provides dozens of algorithms. The algorithms are defined in namespace std and presented in the <algorithm> header. These standard-library algorithms all take sequences as inputs (§4.5). A half-open sequence from b to e is referred to as [b:e). Here are a few I have found particularly useful:

Selected Standard Algorithms	
p=find(b,e,x)	p is the first p in [b:e) so that ∗p==x
p=find_if(b,e,f)	p is the first p in [b:e) so that f(∗p)==true
n=count(b,e,x)	n is the number of elements ∗q in [b:e) so that ∗q==x
n=count_if(b,e,f)	n is the number of elements ∗q in [b:e) so that f(∗q,x)
replace(b,e,v,v2)	Replace elements ∗q in [b:e) so that ∗q==v by v2
replace_if(b,e,f,v2)	Replace elements ∗q in [b:e) so that f(∗q) by v2
p=copy(b,e,out)	Copy [b:e) to [out:p)
p=copy_if(b,e,out,f)	Copy elements ∗q from [b:e) so that f(∗q) to [out:p)
p=unique_copy(b,e,out)	Copy [b:e) to [out:p); don’t copy adjacent duplicates
sort(b,e)	Sort elements of [b:e) using < as the sorting criterion
sort(b,e,f)	Sort elements of [b:e) using f as the sorting criterion
(p1,p2)=equal_range(b,e,v)	[p1:p2) is the subsequence of the sorted sequence [b:e) with the value v; basically a binary search for v
p=merge(b,e,b2,e2,out)	Merge two sorted sequences [b:e) and [b2:e2) into [out:p)

These algorithms, and many more (see Chapter 32), can be applied to elements of containers, strings, and built-in arrays.

4.5.6 Container Algorithms

A sequence is defined by a pair of iterators [begin:end). This is general and flexible, but most often, we apply an algorithm to a sequence that is the contents of a container. For example:

```
sort(v.begin(),v.end());
```

Why don’t we just say sort(v)? We can easily provide that shorthand:

```
namespace Estd {
    using namespace std;

    template<class C>
    void sort(C& c)
    {
        sort(c.begin(),c.end());
    }
```

```
        template<class C, class Pred>
        void sort(C& c, Pred p)
        {
                sort(c.begin(),c.end(),p);
        }

        // ...
}
```

I put the container versions of sort() (and other algorithms) into their own namespace Estd (“extended std”) to avoid interfering with other programmers’ uses of namespace std.

4.6 Advice

[1] Don’t reinvent the wheel; use libraries; §4.1.
[2] When you have a choice, prefer the standard library over other libraries; §4.1.
[3] Do not think that the standard library is ideal for everything; §4.1.
[4] Remember to #include the headers for the facilities you use; §4.1.2.
[5] Remember that standard-library facilities are defined in namespace std; §4.1.2.
[6] Prefer strings over C-style strings (a char*; §2.2.5); §4.2, §4.3.2.
[7] iostreams are type sensitive, type-safe, and extensible; §4.3.
[8] Prefer vector<T>, map<K,T>, and unordered_map<K,T> over T[]; §4.4.
[9] Know your standard containers and their tradeoffs; §4.4.
[10] Use vector as your default container; §4.4.1.
[11] Prefer compact data structures; §4.4.1.1.
[12] If in doubt, use a range-checked vector (such as Vec); §4.4.1.2.
[13] Use push_back() or back_inserter() to add elements to a container; §4.4.1, §4.5.
[14] Use push_back() on a vector rather than realloc() on an array; §4.5.
[15] Catch common exceptions in main(); §4.4.1.2.
[16] Know your standard algorithms and prefer them over handwritten loops; §4.5.5.
[17] If iterator use gets tedious, define container algorithms; §4.5.6.

5

A Tour of C++: Concurrency and Utilities

When you wish to instruct,
be brief.
– Cicero

5.1 Introduction

From an end-user's perspective, the ideal standard library would provide components directly supporting essentially every need. For a given application domain, a huge commercial library can come close to that ideal. However, that is not what the C++ standard library is trying to do. A manageable, universally available, library cannot be everything to everybody. Instead, the C++ standard library aims to provide components that are useful to most people in most application areas. That is, it aims to serve the intersection of all needs rather than their union. In addition, support for a few widely important application areas, such as mathematical computation and text manipulation, have crept in.

5.2 Resource Management

One of the key tasks of any nontrivial program is to manage resources. A resource is something that must be acquired and later (explicitly or implicitly) released. Examples are memory, locks, sockets, thread handles, and file handles. For a long-running program, failing to release a resource in a timely manner (‘‘a leak’’) can cause serious performance degradation and possibly even a miserable crash. Even for short programs, a leak can become an embarrassment, say by a resource shortage increasing the run time by orders of magnitude.

The standard library components are designed not to leak resources. To do this, they rely on the basic language support for resource management using constructor/destructor pairs to ensure that a resource doesn’t outlive an object responsible for it. The use of a constructor/destructor pair in Vector to manage the lifetime of its elements is an example (§3.2.1.2) and all standard-library containers are implemented in similar ways. Importantly, this approach interacts correctly with error handling using exceptions. For example, the technique is used for the standard-library lock classes:

```
mutex m; // used to protect access to shared data
// ...
void f()
{
        unique_lock<mutex> lck {m}; // acquire the mutex m
        // ... manipulate shared data ...
}
```

A thread will not proceed until lck’s constructor has acquired its mutex, m (§5.3.4). The corresponding destructor releases the resource. So, in this example, unique_lock’s destructor releases the mutex when the thread of control leaves f() (through a return, by ‘‘falling off the end of the function,’’ or through an exception throw).

This is an application of the ‘‘Resource Acquisition Is Initialization’’ technique (RAII; §3.2.1.2, §13.3). This technique is fundamental to the idiomatic handling of resources in C++. Containers (such as vector and map), string, and iostream manage their resources (such as file handles and buffers) similarly.

5.2.1 unique_ptr and shared_ptr

The examples so far take care of objects defined in a scope, releasing the resources they acquire at the exit from the scope, but what about objects allocated on the free store? In <memory>, the standard library provides two ‘‘smart pointers’’ to help manage objects on the free store:

[1] unique_ptr to represent unique ownership (§34.3.1)
[2] shared_ptr to represent shared ownership (§34.3.2)

The most basic use of these ‘‘smart pointers’’ is to prevent memory leaks caused by careless programming. For example:

```
void f(int i, int j)       // X* vs. unique_ptr<X>
{
        X∗ p = new X;                     // allocate a new X
        unique_ptr<X> sp {new X};         // allocate a new X and give its pointer to unique_ptr
        // ...
        if (i<99) throw Z{};          // may throw an exception
        if (j<77) return;             // may return "early"
        p−>do_something();            // may throw an exception
        sp−>do_something();           // may throw an exception
        // ...
        delete p;                     // destroy *p
    }
```

Here, we ‘‘forgot’’ to delete p if i<99 or if j<77. On the other hand, unique_ptr ensures that its object is properly destroyed whichever way we exit f() (by throwing an exception, by executing return, or by ‘‘falling off the end’’). Ironically, we could have solved the problem simply by *not* using a pointer and *not* using new:

```
    void f(int i, int j)     // use a local variable
    {
        X x;
        // ...
    }
```

Unfortunately, overuse of new (and of pointers and references) seems to be an increasing problem.

However, when you really need the semantics of pointers, unique_ptr is a very lightweight mechanism with no space or time overhead compared to correct use of a built-in pointer. Its further uses include passing free-store allocated objects in and out of functions:

```
    unique_ptr<X> make_X(int i)
        // make an X and immediately give it to a unique_ptr
    {
        // ... check i, etc. ...
        return unique_ptr<X>{new X{i}};
    }
```

A unique_ptr is a handle to an individual object (or an array) in much the same way that a vector is a handle to a sequence of objects. Both control the lifetime of other objects (using RAII) and both rely on move semantics to make return simple and efficient.

The shared_ptr is similar to unique_ptr except that shared_ptrs are copied rather than moved. The shared_ptrs for an object share ownership of an object and that object is destroyed when the last of its shared_ptrs is destroyed. For example:

```
    void f(shared_ptr<fstream>);
    void g(shared_ptr<fstream>);

    void user(const string& name, ios_base::openmode mode)
    {
        shared_ptr<fstream> fp {new fstream(name,mode)};
        if (!∗fp) throw No_file{};    // make sure the file was properly opened

        f(fp);
        g(fp);
        // ...
    }
```

Now, the file opened by fp's constructor will be closed by the last function to (explicitly or implicitly) destroy a copy of fp. Note that f() or g() may spawn a task holding a copy of fp or in some other way store a copy that outlives user(). Thus, shared_ptr provides a form of garbage collection that respects the destructor-based resource management of the memory-managed objects. This is neither cost free nor exorbitantly expensive, but does make the lifetime of the shared object hard to predict. Use shared_ptr only if you actually need shared ownership.

Given unique_ptr and shared_ptr, we can implement a complete ‘‘no naked new’’ policy (§3.2.1.2) for many programs. However, these ‘‘smart pointers’’ are still conceptually pointers and therefore only my second choice for resource management – after containers and other types that manage their resources at a higher conceptual level. In particular, shared_ptrs do not in themselves provide any rules for which of their owners can read and/or write the shared object. Data races (§41.2.4) and other forms of confusion are not addressed simply by eliminating the resource management issues.

Where do we use ‘‘smart pointers’’ (such as unique_ptr) rather than resource handles with operations designed specifically for the resource (such as vector or thread)? Unsurprisingly, the answer is ‘‘when we need pointer semantics.’’

- When we share an object, we need pointers (or references) to refer to the shared object, so a shared_ptr becomes the obvious choice (unless there is an obvious single owner).
- When we refer to a polymorphic object, we need a pointer (or a reference) because we don't know the exact type of the object referred to or even its size), so a unique_ptr becomes the obvious choice.
- A shared polymorphic object typically requires shared_ptrs.

We do *not* need to use a pointer to return a collection of objects from a function; a container that is a resource handle will do that simply and efficiently (§3.3.2).

5.3 Concurrency

Concurrency – the execution of several tasks simultaneously – is widely used to improve throughput (by using several processors for a single computation) or to improve responsiveness (by allowing one part of a program to progress while another is waiting for a response). All modern programming languages provide support for this. The support provided by the C++ standard library is a portable and type-safe variant of what has been used in C++ for more than 20 years and is almost universally supported by modern hardware. The standard-library support is primarily aimed at supporting systems-level concurrency rather than directly providing sophisticated higher-level concurrency models; those can be supplied as libraries built using the standard-library facilities.

The standard library directly supports concurrent execution of multiple threads in a single address space. To allow that, C++ provides a suitable memory model (§41.2) and a set of atomic operations (§41.3). However, most users will see concurrency only in terms of the standard library and libraries built on top of that. This section briefly gives examples of the main standard-library concurrency support facilities: threads, mutexes, lock() operations, packaged_tasks, and futures. These features are built directly upon what operating systems offer and do not incur performance penalties compared with those.

5.3.1 Tasks and threads

We call a computation that can potentially be executed concurrently with other computations a *task*. A *thread* is the system-level representation of a task in a program. A task to be executed concurrently with other tasks is launched by constructing a **std::thread** (found in **<thread>**) with the task as its argument. A task is a function or a function object:

```
void f();                    // function

struct F {                   // function object
        void operator()();   // F's call operator (§3.4.3)
};

void user()
{
        thread t1 {f};       // f() executes in separate thread
        thread t2 {F()};     // F()() executes in separate thread

        t1.join();           // wait for t1
        t2.join();           // wait for t2
}
```

The **join()**s ensure that we don't exit **user()** until the threads have completed. To ''join'' means to ''wait for the thread to terminate.''

Threads of a program share a single address space. In this, threads differ from processes, which generally do not directly share data. Since threads share an address space, they can communicate through shared objects (§5.3.4). Such communication is typically controlled by locks or other mechanisms to prevent data races (uncontrolled concurrent access to a variable).

Programming concurrent tasks can be *very* tricky. Consider possible implementations of the tasks **f** (a function) and **F** (a function object):

```
void f() { cout << "Hello "; }

struct F {
        void operator()() { cout << "Parallel World!\n"; }
};
```

This is an example of a bad error: Here, **f** and **F()** each use the object **cout** without any form of synchronization. The resulting output would be unpredictable and could vary between different executions of the program because the order of execution of the individual operations in the two tasks is not defined. The program may produce ''odd'' output, such as

```
PaHerallllel o World!
```

When defining tasks of a concurrent program, our aim is to keep tasks completely separate except where they communicate in simple and obvious ways. The simplest way of thinking of a concurrent task is as a function that happens to run concurrently with its caller. For that to work, we just have to pass arguments, get a result back, and make sure that there is no use of shared data in between (no data races).

5.3.2 Passing Arguments

Typically, a task needs data to work upon. We can easily pass data (or pointers or references to the data) as arguments. Consider:

```
void f(vector<double>& v);       // function do something with v

struct F {                       // function object: do something with v
        vector<double>& v;
        F(vector<double>& vv) :v{vv} { }
        void operator()();       // application operator; §3.4.3
};

int main()
{
        vector<double> some_vec {1,2,3,4,5,6,7,8,9};
        vector<double> vec2 {10,11,12,13,14};

        thread t1 {f,some_vec};  // f(some_vec) executes in a separate thread
        thread t2 {F{vec2}};     // F(vec2)() executes in a separate thread

        t1.join();
        t2.join();
}
```

Obviously, `F{vec2}` saves a reference to the argument vector in `F`. `F` can now use that array and hopefully no other task accesses `vec2` while `F` is executing. Passing `vec2` by value would eliminate that risk.

The initialization with `{f,some_vec}` uses a `thread` variadic template constructor that can accept an arbitrary sequence of arguments (§28.6). The compiler checks that the first argument can be invoked given the following arguments and builds the necessary function object to pass to the thread. Thus, if `F::operator()()` and `f()` perform the same algorithm, the handling of the two tasks are roughly equivalent: in both cases, a function object is constructed for the `thread` to execute.

5.3.3 Returning Results

In the example in §5.3.2, I pass the arguments by non-`const` reference. I only do that if I expect the task to modify the value of the data referred to (§7.7). That’s a somewhat sneaky, but not uncommon, way of returning a result. A less obscure technique is to pass the input data by `const` reference and to pass the location of a place to deposit the result as a separate argument:

```
void f(const vector<double>& v, double∗ res);// take input from v; place result in *res

class F {
public:
        F(const vector<double>& vv, double∗ p) :v{vv}, res{p} { }
        void operator()();              // place result in *res
private:
        const vector<double>& v;          // source of input
        double* res;                      // target for output
};

int main()
{
        vector<double> some_vec;
        vector<double> vec2;
        // ...

        double res1;
        double res2;

        thread t1 {f,some_vec,&res1}; // f(some_vec,&res1) executes in a separate thread
        thread t2 {F{vec2,&res2}};        // F{vec2,&res2}() executes in a separate thread

        t1.join();
        t2.join();

        cout << res1 << ' ' << res2 << '\n';
}
```

I don’t consider returning results through arguments particularly elegant, so I return to this topic in §5.3.5.1.

5.3.4 Sharing Data

Sometimes tasks need to share data. In that case, the access has to be synchronized so that at most one task at a time has access. Experienced programmers will recognize this as a simplification (e.g., there is no problem with many tasks simultaneously reading immutable data), but consider how to ensure that at most one task at a time has access to a given set of objects.

The fundamental element of the solution is a **mutex**, a “mutual exclusion object.” A **thread** acquires a mutex using a **lock()** operation:

```
mutex m; // controlling mutex
int sh;     // shared data

void f()
{
        unique_lock<mutex> lck {m}; // acquire mutex
        sh += 7;                          // manipulate shared data
}       // release mutex implicitly
```

The **unique_lock**’s constructor acquires the mutex (through a call **m.lock()**). If another thread has already acquired the mutex, the thread waits (“blocks”) until the other thread completes its access. Once a thread has completed its access to the shared data, the **unique_lock** releases the **mutex** (with a call **m.unlock()**). The mutual exclusion and locking facilities are found in **<mutex>**.

The correspondence between the shared data and a mutex is conventional: the programmer simply has to know which mutex is supposed to correspond to which data. Obviously, this is error-prone, and equally obviously we try to make the correspondence clear through various language means. For example:

```
class Record {
public:
        mutex rm;
        // ...
};
```

It doesn't take a genius to guess that for a Record called rec, rec.rm is a mutex that you are supposed to acquire before accessing the other data of rec, though a comment or a better name might have helped a reader.

It is not uncommon to need to simultaneously access several resources to perform some action. This can lead to deadlock. For example, if thread1 acquires mutex1 and then tries to acquire mutex2 while thread2 acquires mutex2 and then tries to acquire mutex1, then neither task will ever proceed further. The standard library offers help in the form of an operation for acquiring several locks simultaneously:

```
void f()
{
        // ...
        unique_lock<mutex> lck1 {m1,defer_lock};     // defer_lock: don't yet try to acquire the mutex
        unique_lock<mutex> lck2 {m2,defer_lock};
        unique_lock<mutex> lck3 {m3,defer_lock};
        // ...
        lock(lck1,lck2,lck3);                        // acquire all three locks
        // ... manipulate shared data ...
} // implicitly release all mutexes
```

This lock() will only proceed after acquiring all its mutex arguments and will never block (''go to sleep'') while holding a mutex. The destructors for the individual unique_locks ensure that the mutexes are released when a thread leaves the scope.

Communicating through shared data is pretty low level. In particular, the programmer has to devise ways of knowing what work has and has not been done by various tasks. In that regard, use of shared data is inferior to the notion of call and return. On the other hand, some people are convinced that sharing must be more efficient than copying arguments and returns. That can indeed be so when large amounts of data are involved, but locking and unlocking are relatively expensive operations. On the other hand, modern machines are very good at copying data, especially compact data, such as vector elements. So don't choose shared data for communication because of ''efficiency'' without thought and preferably not without measurement.

5.3.4.1 Waiting for Events

Sometimes, a thread needs to wait for some kind of external event, such as another thread completing a task or a certain amount of time having passed. The simplest ''event'' is simply time passing. Consider:

```
using namespace std::chrono;        // see §35.2

auto t0 = high_resolution_clock::now();
this_thread::sleep_for(milliseconds{20});
auto t1 = high_resolution_clock::now();
cout << duration_cast<nanoseconds>(t1−t0).count() << " nanoseconds passed\n";
```

Note that I didn't even have to launch a **thread**; by default, **this_thread** refers to the one and only thread (§42.2.6).

I used **duration_cast** to adjust the clock's units to the nanoseconds I wanted. See §5.4.1 and §35.2 before trying anything more complicated than this with time. The time facilities are found in **<chrono>**.

The basic support for communicating using external events is provided by **condition_variable**s found in **<condition_variable>** (§42.3.4). A **condition_variable** is a mechanism allowing one **thread** to wait for another. In particular, it allows a **thread** to wait for some *condition* (often called an *event*) to occur as the result of work done by other **thread**s.

Consider the classical example of two **thread**s communicating by passing messages through a **queue**. For simplicity, I declare the **queue** and the mechanism for avoiding race conditions on that **queue** global to the producer and consumer:

```
class Message {     // object to be communicated
    // ...
};

queue<Message> mqueue;              // the queue of messages
condition_variable mcond;           // the variable communicating events
mutex mmutex;                       // the locking mechanism
```

The types **queue**, **condition_variable**, and **mutex** are provided by the standard library.

The **consumer()** reads and processes **Message**s:

```
void consumer()
{
    while(true) {
        unique_lock<mutex> lck{mmutex};           // acquire mmutex
        while (mcond.wait(lck)) /* do nothing */;  // release lck and wait;
                                                   // re-acquire lck upon wakeup
        auto m = mqueue.front();                   // get the message
        mqueue.pop();
        lck.unlock();                              // release lck
        // ... process m ...
    }
}
```

Here, I explicitly protect the operations on the **queue** and on the **condition_variable** with a **unique_lock** on the **mutex**. Waiting on **condition_variable** releases its lock argument until the wait is over (so that the queue is non-empty) and then reacquires it.

The corresponding **producer** looks like this:

```
void producer()
{
        while(true) {
                Message m;
                // ... fill the message ...
                unique_lock<mutex> lck {mmutex};          // protect operations
                mqueue.push(m);
                mcond.notify_one();                       // notify
        }                                                 // release lock (at end of scope)
}
```

Using **condition_variable**s supports many forms of elegant and efficient sharing, but can be rather tricky (§42.3.4).

5.3.5 Communicating Tasks

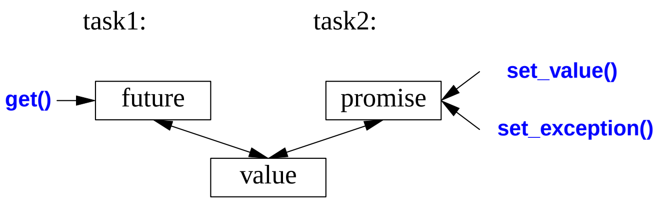
The standard library provides a few facilities to allow programmers to operate at the conceptual level of tasks (work to potentially be done concurrently) rather than directly at the lower level of threads and locks:

[1] **future** and **promise** for returning a value from a task spawned on a separate thread
[2] **packaged_task** to help launch tasks and connect up the mechanisms for returning a result
[3] **async()** for launching of a task in a manner very similar to calling a function.

These facilities are found in **<future>**.

5.3.5.1 future and promise

The important point about **future** and **promise** is that they enable a transfer of a value between two tasks without explicit use of a lock; ‘‘the system’’ implements the transfer efficiently. The basic idea is simple: When a task wants to pass a value to another, it puts the value into a **promise**. Somehow, the implementation makes that value appear in the corresponding **future**, from which it can be read (typically by the launcher of the task). We can represent this graphically:



If we have a **future<X>** called **fx**, we can **get()** a value of type **X** from it:

```
X v = fx.get();   // if necessary, wait for the value to get computed
```

If the value isn’t there yet, our thread is blocked until it arrives. If the value couldn’t be computed, **get()** might throw an exception (from the system or transmitted from the task from which we were trying to **get()** the value).

The main purpose of a **promise** is to provide simple ‘‘put’’ operations (called **set_value()** and **set_exception()**) to match **future**’s **get()**. The names ‘‘future’’ and ‘‘promise’’ are historical; please don’t blame me. They are yet another fertile source of puns.

If you have a **promise** and need to send a result of type **X** to a **future**, you can do one of two things: pass a value or pass an exception. For example:

```
void f(promise<X>& px)   // a task: place the result in px
{
        // ...
        try {
                X res;
                // ... compute a value for res ...
                px.set_value(res);
        }
        catch (...) {               // oops: couldn’t compute res
                // pass the exception to the future’s thread:
                px.set_exception(current_exception());
        }
}
```

The **current_exception()** refers to the caught exception (§30.4.1.2).

To deal with an exception transmitted through a **future**, the caller of **get()** must be prepared to catch it somewhere. For example:

```
void g(future<X>& fx)           // a task: get the result from fx
{
        // ...
        try {
                X v = fx.get();   // if necessary, wait for the value to get computed
                // ... use v ...
        }
        catch (...) {               // oops: someone couldn’t compute v
                // ... handle error ...
        }
}
```

5.3.5.2 packaged_task

How do we get a **future** into the task that needs a result and the corresponding **promise** into the thread that should produce that result? The **packaged_task** type is provided to simplify setting up tasks connected with **future**s and **promise**s to be run on **thread**s. A **packaged_task** provides wrapper code to put the return value or exception from the task into a **promise** (like the code shown in §5.3.5.1). If you ask it by calling **get_future**, a **packaged_task** will give you the **future** corresponding to its **promise**. For example, we can set up two tasks to each add half of the elements of a **vector<double>** using the standard-library **accumulate()** (§3.4.2, §40.6):

```
double accum(double* beg, double * end, double init)
        // compute the sum of [beg:end) starting with the initial value init
{
        return accumulate(beg,end,init);
}

double comp2(vector<double>& v)
{
        using Task_type = double(double*,double*,double);          // type of task

        packaged_task<Task_type> pt0 {accum};                      // package the task (i.e., accum)
        packaged_task<Task_type> pt1 {accum};

        future<double> f0 {pt0.get_future()};                      // get hold of pt0's future
        future<double> f1 {pt1.get_future()};                      // get hold of pt1's future

        double* first = &v[0];
        thread t1 {move(pt0),first,first+v.size()/2,0};            // start a thread for pt0
        thread t2 {move(pt1),first+v.size()/2,first+v.size(),0};   // start a thread for pt1

        // ...

        return f0.get()+f1.get();                       // get the results
}
```

The `packaged_task` template takes the type of the task as its template argument (here `Task_type`, an alias for `double(double*,double*,double)`) and the task as its constructor argument (here, `accum`). The `move()` operations are needed because a `packaged_task` cannot be copied.

Please note the absence of explicit mention of locks in this code: we are able to concentrate on tasks to be done, rather than on the mechanisms used to manage their communication. The two tasks will be run on separate threads and thus potentially in parallel.

5.3.5.3 async()

The line of thinking I have pursued in this chapter is the one I believe to be the simplest yet still among the most powerful: Treat a task as a function that may happen to run concurrently with other tasks. It is far from the only model supported by the C++ standard library, but it serves well for a wide range of needs. More subtle and tricky models, e.g., styles of programming relying on shared memory, can be used as needed.

To launch tasks to potentially run asynchronously, we can use `async()`:

```
double comp4(vector<double>& v)
        // spawn many tasks if v is large enough
{
        if (v.size()<10000) return accum(v.begin(),v.end(),0.0);

        auto v0 = &v[0];
        auto sz = v.size();
        auto f0 = async(accum,v0,v0+sz/4,0.0);           // first quarter
        auto f1 = async(accum,v0+sz/4,v0+sz/2,0.0);      // second quarter
        auto f2 = async(accum,v0+sz/2,v0+sz*3/4,0.0);    // third quarter
        auto f3 = async(accum,v0+sz*3/4,v0+sz,0.0);      // fourth quarter

        return f0.get()+f1.get()+f2.get()+f3.get(); // collect and combine the results
    }
```

Basically, async() separates the ‘‘call part’’ of a function call from the ‘‘get the result part,’’ and separates both from the actual execution of the task. Using async(), you don’t have to think about threads and locks. Instead, you think just in terms of tasks that potentially compute their results asynchronously. There is an obvious limitation: Don’t even think of using async() for tasks that share resources needing locking – with async() you don’t even know how many threads will be used because that’s up to async() to decide based on what it knows about the system resources available at the time of a call. For example, async() may check whether any idle cores (processors) are available before deciding how many threads to use.

Please note that async() is not just a mechanism specialized for parallel computation for increased performance. For example, it can also be used to spawn a task for getting information from a user, leaving the ‘‘main program’’ active with something else (§42.4.6).

5.4 Small Utility Components

Not all standard-library components come as part of obviously labeled facilities, such as ‘‘containers’’ or ‘‘I/O.’’ This section gives a few examples of small, widely useful components:

- clock and duration for measuring time.
- Type functions, such as iterator_traits and is_arithmetic, for gaining information about types.
- pair and tuple for representing small potentially heterogeneous sets of values.

The point here is that a function or a type need not be complicated or closely tied to a mass of other functions and types to be useful. Such library components mostly act as building blocks for more powerful library facilities, including other components of the standard library.

5.4.1 Time

The standard library provides facilities for dealing with time. For example, here is the basic way of timing something:

```
using namespace std::chrono;      // see §35.2

auto t0 = high_resolution_clock::now();
do_work();
auto t1 = high_resolution_clock::now();
cout << duration_cast<milliseconds>(t1−t0).count() << "msec\n";
```

The clock returns a time_point (a point in time). Subtracting two time_points gives a duration (a period of time). Various clocks give their results in various units of time (the clock I used measures nanoseconds), so it is usually a good idea to convert a duration into a known unit. That’s what duration_cast does.

The standard-library facilities for dealing with time are found in the subnamespace **std::chrono** in **<chrono>** (§35.2).

Don't make statements about "efficiency" of code without first doing time measurements. Guesses about performance are most unreliable.

5.4.2 Type Functions

A *type function* is a function that is evaluated at compile-time given a type as its argument or returning a type. The standard library provides a variety of type functions to help library implementers and programmers in general to write code that take advantage of aspects of the language, the standard library, and code in general.

For numerical types, **numeric_limits** from **<limits>** presents a variety of useful information (§5.6.5). For example:

```
constexpr float min = numeric_limits<float>::min();        // smallest positive float (§40.2)
```

Similarly, object sizes can be found by the built-in **sizeof** operator (§2.2.2). For example:

```
constexpr int szi = sizeof(int); // the number of bytes in an int
```

Such type functions are part of C++'s mechanisms for compile-time computation that allow tighter type checking and better performance than would otherwise have been possible. Use of such features is often called *metaprogramming* or (when templates are involved) *template metaprogramming* (Chapter 28). Here, I just present two facilities provided by the standard library: **iterator_traits** (§5.4.2.1) and type predicates (§5.4.2.2).

5.4.2.1 iterator_traits

The standard-library **sort()** takes a pair of iterators supposed to define a sequence (§4.5). Furthermore, those iterators must offer random access to that sequence, that is, they must be *random-access iterators*. Some containers, such as **forward_list**, do not offer that. In particular, a **forward_list** is a singly-linked list so subscripting would be expensive and there is no reasonable way to refer back to a previous element. However, like most containers, **forward_list** offers *forward iterators* that can be used to traverse the sequence by algorithms and **for**-statements (§33.1.1).

The standard library provides a mechanism, **iterator_traits** that allows us to check which kind of iterator is supported. Given that, we can improve the range **sort()** from §4.5.6 to accept either a **vector** or a **forward_list**. For example:

```
void test(vector<string>& v, forward_list<int>& lst)
{
    sort(v);    // sort the vector
    sort(lst);  // sort the singly-linked list
}
```

The techniques needed to make that work are generally useful.

First, I write two helper functions that take an extra argument indicating whether they are to be used for random-access iterators or forward iterators. The version taking random-access iterator arguments is trivial:

```
template<typename Ran>                                          // for random-access iterators
void sort_helper(Ran beg, Ran end, random_access_iterator_tag)  // we can subscript into [beg:end)
{
        sort(beg,end);          // just sort it

}
```

The version for forward iterators is almost as simple; just copy the list into a **vector**, sort, and copy back again:

```
template<typename For>                                  // for forward iterators
void sort_helper(For beg, For end, forward_iterator_tag) // we can traverse [beg:end)
{
        vector<decltype(*beg)> v {beg,end};     // initialize a vector from [beg:end)
        sort(v.begin(),v.end());
        copy(v.begin(),v.end(),beg);            // copy the elements back
}
```

The **decltype()** is a built-in type function that returns the declared type of its argument (§6.3.6.3). Thus, **v** is a **vector<X>** where **X** is the element type of the input sequence.

The real "type magic" is in the selection of helper functions:

```
template<typname C>
void sort(C& c)
{
        using Iter = Iterator_type<C>;
        sort_helper(c.begin(),c.end(),Iterator_category<Iter>{});
}
```

Here, I use two type functions: **Iterator_type<C>** returns the iterator type of **C** (that is, **C::iterator**) and then **Iterator_category<Iter>{}** constructs a "tag" value indicating the kind of iterator provided:

- **std::random_access_iterator_tag** if **C**'s iterator supports random access.
- **std::forward_iterator_tag** if **C**'s iterator supports forward iteration.

Given that, we can select between the two sorting algorithms at compile time. This technique, called *tag dispatch* is one of several used in the standard library and elsewhere to improve flexibility and performance.

The standard-library support for techniques for using iterators, such as tag dispatch, comes in the form of a simple class template **iterator_traits** from **<iterator>** (§33.1.3). This allows simple definitions of the type functions used in **sort()**:

```
template<typename C>
        using Iterator_type = typename C::iterator;     // C's iterator type

template<typename Iter>
        using Iterator_category = typename std::iterator_traits<Iter>::iterator_category;  // Iter's category
```

If you don't want to know what kind of "compile-time type magic" is used to provide the standard-library features, you are free to ignore facilities such as **iterator_traits**. But then you can't use the techniques they support to improve your own code.

5.4.2.2 Type Predicates

A standard-library type predicate is a simple type function that answers a fundamental question about types. For example:

```
bool b1 = Is_arithmetic<int>();       // yes, int is an arithmetic type
bool b2 = Is_arithmetic<string>();    // no, std::string is not an arithmetic type
```

These predicates are found in **<type_traits>** and described in §35.4.1. Other examples are **is_class**, **is_pod**, **is_literal_type**, **has_virtual_destructor**, and **is_base_of**. They are most useful when we write templates. For example:

```
template<typename Scalar>
class complex {
        Scalar re, im;
public:
        static_assert(Is_arithmetic<Scalar>(), "Sorry, I only support complex of arithmetic types");
        // ...
};
```

To improve readability compared to using the standard library directly, I defined a type function:

```
template<typename T>
constexpr bool Is_arithmetic()
{
        return std::is_arithmetic<T>::value ;
}
```

Older programs use **::value** directly instead of **()**, but I consider that quite ugly and it exposes implementation details.

5.4.3 pair and tuple

Often, we need some data that is just data; that is, a collection of values, rather than an object of a class with a well-defined semantics and an invariant for its value (§2.4.3.2, §13.4). In such cases, we could define a simple **struct** with an appropriate set of appropriately named members. Alternatively, we could let the standard library write the definition for us. For example, the standard-library algorithm **equal_range** (§32.6.1) returns a **pair** of iterators specifying a sub-sequence meeting a predicate:

```
template<typename Forward_iterator, typename T, typename Compare>
        pair<Forward_iterator,Forward_iterator>
        equal_range(Forward_iterator first, Forward_iterator last, const T& val, Compare cmp);
```

Given a sorted sequence [**first**:**last**), **equal_range()** will return the **pair** representing the subsequence that matches the predicate **cmp**. We can use that to search in a sorted sequence of **Records**:

```
auto rec_eq = [](const Record& r1, const Record& r2) { return r1.name<r2.name;};// compare names

void f(const vector<Record>& v)          // assume that v is sorted on its "name" field
{
        auto er = equal_range(v.begin(),v.end(),Record{"Reg"},rec_eq);
        for (auto p = er.first; p!=er.second; ++p)        // print all equal records
                cout << *p;                               // assume that << is defined for Record
}
```

The first member of a pair is called first and the second member is called second. This naming is not particularly creative and may look a bit odd at first, but such consistent naming is a boon when we want to write generic code.

The standard-library pair (from <utility>) is quite frequently used in the standard library and elsewhere. A pair provides operators, such as =, ==, and <, if its elements do. The make_pair() function makes it easy to create a pair without explicitly mentioning its type (§34.2.4.1). For example:

```
void f(vector<string>& v)
{
        auto pp = make_pair(v.begin(),2);    // pp is a pair<vector<string>::iterator,int>
        // ...
}
```

If you need more than two elements (or less), you can use tuple (from <utility>; §34.2.4.2). A tuple is a heterogeneous sequence of elements; for example:

```
tuple<string,int,double> t2("Sild",123, 3.14);   // the type is explicitly specified

auto t = make_tuple(string("Herring"),10, 1.23);      // the type is deduced
                                                      // t is a tuple<string,int,double>

string s = get<0>(t); // get first element of tuple
int x = get<1>(t);
double d = get<2>(t);
```

The elements of a tuple are numbered (starting with zero), rather than named the way elements of pairs are (first and second). To get compile-time selection of elements, I must unfortunately use the ugly get<1>(t), rather than get(t,1) or t[1] (§28.5.2).

Like pairs, tuples can be assigned and compared if their elements can be.

A pair is common in interfaces because often we want to return more than one value, such as a result and an indicator of the quality of that result. It is less common to need three or more parts to a result, so tuples are more often found in the implementations of generic algorithms.

5.5 Regular Expressions

Regular expressions are a powerful tool for text processing. They provide a way to simply and tersely describe patterns in text (e.g., a U.S. ZIP code such as TX 77845, or an ISO-style date, such as 2009–06–07) and to efficiently find such patterns in text. In <regex>, the standard library provides support for regular expressions in the form of the std::regex class and its supporting functions. To give a taste of the style of the regex library, let us define and print a pattern:

```
regex pat (R"(\w{2}\s*\d{5}(–\d{4})?)");    // ZIP code pattern: XXddddd-dddd and variants
cout << "pattern: " << pat << '\n';
```

People who have used regular expressions in just about any language will find \w{2}\s*\d{5}(–\d{4})? familiar. It specifies a pattern starting with two letters \w{2} optionally followed by some space \s* followed by five digits \d{5} and optionally followed by a dash and four digits –\d{4}. If you are not familiar with regular expressions, this may be a good time to learn about them ([Stroustrup,2009], [Maddock,2009], [Friedl,1997]). Regular expressions are summarized in §37.1.1.

To express the pattern, I use a *raw string literal* (§7.3.2.1) starting with R"(and terminated by)". This allows backslashes and quotes to be used directly in the string.

The simplest way of using a pattern is to search for it in a stream:

```
int lineno = 0;
for (string line; getline(cin,line);) {          // read into line buffer
        ++lineno;
        smatch matches;                          // matched strings go here
        if (regex_search(line,matches,pat))      // search for pat in line
                cout << lineno << ": " << matches[0] << '\n';
}
```

The regex_search(line,matches,pat) searches the line for anything that matches the regular expression stored in pat and if it finds any matches, it stores them in matches. If no match was found, regex_search(line,matches,pat) returns false. The matches variable is of type smatch. The ‘‘s’’ stands for ‘‘sub’’ and an smatch is a vector of sub-matches. The first element, here matches[0], is the complete match.

For a more complete description see Chapter 37.

5.6 Math

C++ wasn’t designed primarily with numerical computation in mind. However, C++ is heavily used for numerical computation and the standard library reflects that.

5.6.1 Mathematical Functions and Algorithms

In <cmath>, we find the ‘‘usual mathematical functions,’’ such as sqrt(), log(), and sin() for arguments of type float, double, and long double (§40.3). Complex number versions of these functions are found in <complex> (§40.4).

In <numeric>, we find a small set of generalized numerical algorithms, such as accumulate(). For example:

```
void f()
{
        list<double> lst {1, 2, 3, 4, 5, 9999.99999};
        auto s = accumulate(lst.begin(),lst.end(),0.0); // calculate the sum
        cout << s << '\n';                              // print 10014.9999
}
```

These algorithms work for every standard-library sequence and can have operations supplied as arguments (§40.6).

5.6.2 Complex Numbers

The standard library supports a family of complex number types along the lines of the complex class described in §2.3. To support complex numbers where the scalars are single-precision floating-point numbers (floats), double-precision floating-point numbers (doubles), etc., the standard library complex is a template:

```
template<typename Scalar>
class complex {
public:
        complex(const Scalar& re ={}, const Scalar& im ={});
        // ...
};
```

The usual arithmetic operations and the most common mathematical functions are supported for complex numbers. For example:

```
void f(complex<float> fl, complex<double> db)
{
        complex<long double> ld {fl+sqrt(db)};
        db += fl*3;
        fl = pow(1/fl,2);
        // ...
}
```

The sqrt() and pow() (exponentiation) functions are among the usual mathematical functions defined in <complex>. For more details, see §40.4.

5.6.3 Random Numbers

Random numbers are useful in many contexts, such as testing, games, simulation, and security. The diversity of application areas is reflected in the wide selection of random number generators provided by the standard library in <random>. A random number generator consists of two parts:

[1] an *engine* that produces a sequence of random or pseudo-random values.
[2] a *distribution* that maps those values into a mathematical distribution in a range.

Examples of distributions are uniform_int_distribution (where all integers produced are equally likely), normal_distribution (“the bell curve”), and exponential_distribution (exponential growth); each for some specified range. For example:

```
using my_engine = default_random_engine;                // type of engine
using my_distribution = uniform_int_distribution<>;     // type of distribution

my_engine re {};                                        // the default engine
my_distribution one_to_six {1,6};                       // distribution that maps to the ints 1..6
auto die = bind(one_to_six,re);                         // make a generator

int x = die();                                          // roll the die: x becomes a value in [1:6]
```

The standard-library function bind() makes a function object that will invoke its first argument (here, one_to_six) given its second argument (here, re) as its argument (§33.5.1). Thus a call die() is equivalent to a call one_to_six(re).

Thanks to its uncompromising attention to generality and performance one expert has deemed the standard-library random number component "what every random number library wants to be when it grows up." However, it can hardly be deemed "novice friendly." The **using** statements makes what is being done a bit more obvious. Instead, I could just have written:

```
auto die = bind(uniform_int_distribution<>{1,6}, default_random_engine{});
```

Which version is the more readable depends entirely on the context and the reader.

For novices (of any background) the fully general interface to the random number library can be a serious obstacle. A simple uniform random number generator is often sufficient to get started. For example:

```
Rand_int rnd {1,10};                    // make a random number generator for [1:10]
int x = rnd();                  // x is a number in [1:10]
```

So, how could we get that? We have to get something like **die()** inside a class **Rand_int**:

```
class Rand_int {
public:
        Rand_int(int low, int high) :dist{low,high} { }
        int operator()() { return dist(re); }           // draw an int
private:
        default_random_engine re;
        uniform_int_distribution<> dist;
};
```

That definition is still "expert level," but the *use* of **Rand_int()** is manageable in the first week of a C++ course for novices. For example:

```
int main()
{
        Rand_int rnd {0,4};         // make a uniform random number generator

        vector<int> histogram(5);               // make a vector of size 5
        for (int i=0; i!=200; ++i)
                ++histogram[rnd()];             // fill histogram with the frequencies of numbers [0:4]

        for (int i = 0; i!=mn.size(); ++i) {    // write out a bar graph
                cout << i << '\t';
                for (int j=0; j!=mn[i]; ++j) cout << '*';
                cout << endl;
        }
}
```

The output is a (reassuringly boring) uniform distribution (with reasonable statistical variation):

```
0       ********************************************
1       ******************************************
2       *******************************
3       ******************************************
4       *****************************************
```

There is no standard graphics library for C++, so I use ‘‘ASCII graphics.’’ Obviously, there are lots of open source and commercial graphics and GUI libraries for C++, but in this book I’ll restrict myself to ISO standard facilities.

For more information about random numbers, see §40.7.

5.6.4 Vector Arithmetic

The **vector** described in §4.4.1 was designed to be a general mechanism for holding values, to be flexible, and to fit into the architecture of containers, iterators, and algorithms. However, it does not support mathematical vector operations. Adding such operations to **vector** would be easy, but its generality and flexibility precludes optimizations that are often considered essential for serious numerical work. Consequently, the standard library provides (in **<valarray>**) a **vector**-like template, called **valarray**, that is less general and more amenable to optimization for numerical computation:

```
template<typename T>
class valarray {
      // ...
};
```

The usual arithmetic operations and the most common mathematical functions are supported for **valarray**s. For example:

```
void f(valarray<double>& a1, valarray<double>& a2)
{
      valarray<double> a = a1*3.14+a2/a1;          // numeric array operators *, +, /, and =
      a2 += a1*3.14;
      a = abs(a);
      double d = a2[7];
      // ...
}
```

For more details, see §40.5. In particular, **valarray** offers stride access to help implement multidimensional computations.

5.6.5 Numeric Limits

In **<limits>**, the standard library provides classes that describe the properties of built-in types – such as the maximum exponent of a **float** or the number of bytes in an **int**; see §40.2. For example, we can assert that a **char** is signed:

```
static_assert(numeric_limits<char>::is_signed,"unsigned characters!");
static_assert(100000<numeric_limits<int>::max(),"small ints!");
```

Note that the second assert (only) works because **numeric_limits<int>::max()** is a **constexpr** function (§2.2.3, §10.4).

5.7 Advice

[1] Use resource handles to manage resources (RAII); §5.2.
[2] Use unique_ptr to refer to objects of polymorphic type; §5.2.1.
[3] Use shared_ptr to refer to shared objects; §5.2.1.
[4] Use type-safe mechanisms for concurrency; §5.3.
[5] Minimize the use of shared data; §5.3.4.
[6] Don’t choose shared data for communication because of ‘‘efficiency’’ without thought and preferably not without measurement; §5.3.4.
[7] Think in terms of concurrent tasks, rather than threads; §5.3.5.
[8] A library doesn’t have to be large or complicated to be useful; §5.4.
[9] Time your programs before making claims about efficiency; §5.4.1.
[10] You can write code to explicitly depend on properties of types; §5.4.2.
[11] Use regular expressions for simple pattern matching; §5.5.
[12] Don’t try to do serious numeric computation using only the language; use libraries; §5.6.
[13] Properties of numeric types are accessible through numeric_limits; §5.6.5.

Part II

Basic Facilities

This part describes C++'s built-in types and the basic facilities for constructing programs out of them. The C subset of C++ is presented together with C++'s additional support for traditional styles of programming. It also discusses the basic facilities for composing a C++ program out of logical and physical parts.

Chapters

“... I have long entertained a suspicion, with regard to the decisions of philosophers upon all subjects, and found in myself a greater inclination to dispute, than assent to their conclusions. There is one mistake, to which they seem liable, almost without exception; they confine too much their principles, and make no account of that vast variety, which nature has so much affected in all her operations. When a philosopher has once laid hold of a favourite principle, which perhaps accounts for many natural effects, he extends the same principle over the whole creation, and reduces to it every phænomenon, though by the most violent and absurd reasoning. ...”

– David Hume,
Essays, Moral, Political, and Literary. PART I. (1752)

6

Types and Declarations

Perfection is achieved
only on the point of collapse.
– C. N. Parkinson

6.1 The ISO C++ Standard

The C++ language and standard library are defined by their ISO standard: ISO/IEC 14882:2011. In this book, references to the standard are of the form §iso.23.3.6.1. In cases where the text of this book is considered imprecise, incomplete, or possibly wrong, consult the standard. But don't expect the standard to be a tutorial or to be easily accessible by non-experts.

Strictly adhering to the C++ language and library standard doesn't by itself guarantee good code or even portable code. The standard doesn't say whether a piece of code is good or bad; it simply says what a programmer can and cannot rely on from an implementation. It is easy to write perfectly awful standard-conforming programs, and most real-world programs rely on features that the standard does not guarantee to be portable. They do so to access system interfaces and hardware features that cannot be expressed directly in C++ or require reliance on specific implementation details.

Many important things are deemed *implementation-defined* by the standard. This means that each implementation must provide a specific, well-defined behavior for a construct and that behavior must be documented. For example:

```
unsigned char c1 = 64;        // well defined: a char has at least 8 bits and can always hold 64
unsigned char c2 = 1256;      // implementation-defined: truncation if a char has only 8 bits
```

The initialization of `c1` is well defined because a `char` must be at least 8 bits. However, the behavior of the initialization of `c2` is implementation-defined because the number of bits in a `char` is implementation-defined. If the `char` has only 8 bits, the value `1256` will be truncated to `232` (§10.5.2.1). Most implementation-defined features relate to differences in the hardware used to run a program.

Other behaviors are *unspecified*; that is, a range of possible behaviors are acceptable, but the implementer is not obliged to specify which actually occur. Usually, the reason for deeming something unspecified is that the exact behavior is unpredictable for fundamental reasons. For example, the exact value returned by `new` is unspecified. So is the value of a variable assigned to from two threads unless some synchronization mechanism has been employed to prevent a data race (§41.2).

When writing real-world programs, it is usually necessary to rely on implementation-defined behavior. Such behavior is the price we pay for the ability to operate effectively on a large range of systems. For example, C++ would have been much simpler if all characters had been 8 bits and all pointers 32 bits. However, 16-bit and 32-bit character sets are not uncommon, and machines with 16-bit and 64-bit pointers are in wide use.

To maximize portability, it is wise to be explicit about what implementation-defined features we rely on and to isolate the more subtle examples in clearly marked sections of a program. A typical example of this practice is to present all dependencies on hardware sizes in the form of constants and type definitions in some header file. To support such techniques, the standard library provides `numeric_limits` (§40.2). Many assumptions about implementation-defined features can be checked by stating them as static assertions (§2.4.3.3). For example:

```
static_assert(4<=sizeof(int),"sizeof(int) too small");
```

Undefined behavior is nastier. A construct is deemed *undefined* by the standard if no reasonable behavior is required by an implementation. Typically, some obvious implementation technique will cause a program using an undefined feature to behave very badly. For example:

```
const int size = 4∗1024;
char page[size];

void f()
{
        page[size+size] = 7; // undefined
}
```

Plausible outcomes of this code fragment include overwriting unrelated data and triggering a hardware error/exception. An implementation is not required to choose among plausible outcomes. Where powerful optimizers are used, the actual effects of undefined behavior can become quite unpredictable. If a set of plausible and easily implementable alternatives exist, a feature is deemed unspecified or implementation-defined rather than undefined.

It is worth spending considerable time and effort to ensure that a program does not use something deemed unspecified or undefined by the standard. In many cases, tools exist to help do this.

6.1.1 Implementations

A C++ implementation can be either *hosted* or *freestanding* (§iso.17.6.1.3). A hosted implementation includes all the standard-library facilities as described in the standard (§30.2) and in this book. A freestanding implementation may provide fewer standard-library facilities, as long as the following are provided:

Freestanding Implementation Headers		
Types	<cstddef>	§10.3.1
Implementation properties	<cfloat> <limits> <climits>	§40.2
Integer types	<cstdint>	§43.7
Start and termination	<cstdlib>	§43.7
Dynamic memory management	<new>	§11.2.3
Type identification	<typeinfo>	§22.5
Exception handling	<exception>	§30.4.1.1
Initializer lists	<initializer_list>	§30.3.1
Other run-time support	<cstdalign> <cstdarg> <cstdbool>	§12.2.4, §44.3.4
Type traits	<type_traits>	§35.4.1
Atomics	<atomic>	§41.3

Freestanding implementations are meant for code running with only the most minimal operating system support. Many implementations also provide a (non-standard) option for not using exceptions for really minimal, close-to-the-hardware, programs.

6.1.2 The Basic Source Character Set

The C++ standard and the examples in this book are written using the *basic source character set* consisting of the letters, digits, graphical characters, and whitespace characters from the U.S. variant of the international 7-bit character set ISO 646-1983 called ASCII (ANSI3.4-1968). This can cause problems for people who use C++ in an environment with a different character set:

- ASCII contains punctuation characters and operator symbols (such as], {, and !) that are not available in some character sets.
- We need a notation for characters that do not have a convenient character representation (such as newline and ‘‘the character with value 17’’).
- ASCII doesn’t contain characters (such as ñ, Þ, and Æ) that are used for writing languages other than English.

To use an extended character set for source code, a programming environment can map the extended character set into the basic source character set in one of several ways, for example, by using universal character names (§6.2.3.2).

6.2 Types

Consider:

```
x = y+f(2);
```

For this to make sense in a C++ program, the names **x**, **y**, and **f** must be suitably declared. That is, the programmer must specify that entities named **x**, **y**, and **f** exist and that they are of types for which **=** (assignment), **+** (addition), and **()** (function call), respectively, are meaningful.

Every name (identifier) in a C++ program has a type associated with it. This type determines what operations can be applied to the name (that is, to the entity referred to by the name) and how such operations are interpreted. For example:

```
float x;          // x is a floating-point variable
int y = 7;        // y is an integer variable with the initial value 7
float f(int);     // f is a function taking an argument of type int and returning a floating-point number
```

These declarations would make the example meaningful. Because **y** is declared to be an **int**, it can be assigned to, used as an operand for **+**, etc. On the other hand, **f** is declared to be a function that takes an **int** as its argument, so it can be called given the interger **2**.

This chapter presents fundamental types (§6.2.1) and declarations (§6.3). Its examples just demonstrate language features; they are not intended to do anything useful. More extensive and realistic examples are saved for later chapters. This chapter simply provides the most basic elements from which C++ programs are constructed. You must know these elements, plus the terminology and simple syntax that go with them, in order to complete a real project in C++ and especially to read code written by others. However, a thorough understanding of every detail mentioned in this chapter is not a requirement for understanding the following chapters. Consequently, you may prefer to skim through this chapter, observing the major concepts, and return later as the need for understanding more details arises.

6.2.1 Fundamental Types

C++ has a set of *fundamental types* corresponding to the most common basic storage units of a computer and the most common ways of using them to hold data:

§6.2.2 A Boolean type (**bool**)
§6.2.3 Character types (such as **char** and **wchar_t**)
§6.2.4 Integer types (such as **int** and **long long**)
§6.2.5 Floating-point types (such as **double** and **long double**)
§6.2.7 A type, **void**, used to signify the absence of information

From these types, we can construct other types using declarator operators:

§7.2 Pointer types (such as **int***)
§7.3 Array types (such as **char[]**)
§7.7 Reference types (such as **double&** and **vector<int>&&**)

In addition, a user can define additional types:

§8.2 Data structures and classes (Chapter 16)
§8.4 Enumeration types for representing specific sets of values (**enum** and **enum class**)

The Boolean, character, and integer types are collectively called *integral types*. The integral and floating-point types are collectively called *arithmetic types*. Enumerations and classes (Chapter 16) are called *user-defined types* because they must be defined by users rather than being available for use without previous declaration, the way fundamental types are. In contrast, fundamental types, pointers, and references are collectively referred to as *built-in types*. The standard library provides many user-defined types (Chapter 4, Chapter 5).

The integral and floating-point types are provided in a variety of sizes to give the programmer a choice of the amount of storage consumed, the precision, and the range available for computations (§6.2.8). The assumption is that a computer provides bytes for holding characters, words for holding and computing integer values, some entity most suitable for floating-point computation, and addresses for referring to those entities. The C++ fundamental types together with pointers and arrays present these machine-level notions to the programmer in a reasonably implementation-independent manner.

For most applications, we could use **bool** for logical values, **char** for characters, **int** for integer values, and **double** for floating-point values. The remaining fundamental types are variations for optimizations, special needs, and compatibility that are best ignored until such needs arise.

6.2.2 Booleans

A Boolean, **bool**, can have one of the two values **true** or **false**. A Boolean is used to express the results of logical operations. For example:

```
void f(int a, int b)
{
      bool b1 {a==b};
      // ...
}
```

If **a** and **b** have the same value, **b1** becomes **true**; otherwise, **b1** becomes **false**.

A common use of **bool** is as the type of the result of a function that tests some condition (a predicate). For example:

```
bool is_open(File*);

bool greater(int a, int b) { return a>b; }
```

By definition, **true** has the value **1** when converted to an integer and **false** has the value **0**. Conversely, integers can be implicitly converted to **bool** values: nonzero integers convert to **true** and **0** converts to **false**. For example:

```
bool b1 = 7;      // 7!=0, so b becomes true
bool b2 {7};      // error: narrowing (§2.2.2, §10.5)

int i1 = true;    // i1 becomes 1
int i2 {true};    // i2 becomes 1
```

If you prefer to use the **{}**-initializer syntax to prevent narrowing, yet still want to convert an **int** to a **bool**, you can be explicit:

```
void f(int i)
{
        bool b {i!=0};
        // ...
};
```

In arithmetic and logical expressions, bools are converted to ints; integer arithmetic and logical operations are performed on the converted values. If the result needs to be converted back to bool, a 0 is converted to false and a nonzero value is converted to true. For example:

```
bool a = true;
bool b = true;

bool x = a+b;    // a+b is 2, so x becomes true
bool y = a||b;   // a||b is 1, so y becomes true ("||" means "or")
bool z = a−b;    // a-b is 0, so z becomes false
```

A pointer can be implicitly converted to a bool (§10.5.2.5). A non-null pointer converts to true; pointers with the value nullptr convert to false. For example:

```
void g(int∗ p)
{
        bool b = p;                   // narrows to true or false
        bool b2 {p!=nullptr};         // explicit test against nullptr

        if (p) {        // equivalent to p!=nullptr
                // ...
        }
}
```

I prefer if (p) over if (p!=nullptr) because it more directly expresses the notion ‘‘if p is valid’’ and also because it is shorter. The shorter form leaves fewer opportunities for mistakes.

6.2.3 Character Types

There are many character sets and character set encodings in use. C++ provides a variety of character types that reflect that – often bewildering – variety:

- char: The default character type, used for program text. A char is used for the implementation's character set and is usually 8 bits.
- signed char: Like char, but guaranteed to be signed, that is, capable of holding both positive and negative values.
- unsigned char: Like char, but guaranteed to be unsigned.
- wchar_t: Provided to hold characters of a larger character set such as Unicode (see §7.3.2.2). The size of wchar_t is implementation-defined and large enough to hold the largest character set supported by the implementation's locale (Chapter 39).
- char16_t: A type for holding 16-bit character sets, such as UTF-16.
- char32_t: A type for holding 32-bit character sets, such as UTF-32.

These are six distinct types (despite the fact that the _t suffix is often used to denote aliases; §6.5). On each implementation, the char type will be identical to that of either signed char or unsigned char, but these three names are still considered separate types.

A char variable can hold a character of the implementation's character set. For example:

```
char ch = 'a';
```

Almost universally, a char has 8 bits so that it can hold one of 256 different values. Typically, the character set is a variant of ISO-646, for example ASCII, thus providing the characters appearing on your keyboard. Many problems arise from the fact that this set of characters is only partially standardized.

Serious variations occur between character sets supporting different natural languages and between character sets supporting the same natural language in different ways. Here, we are interested only in how such differences affect the rules of C++. The larger and more interesting issue of how to program in a multilingual, multi-character-set environment is beyond the scope of this book, although it is alluded to in several places (§6.2.3, §36.2.1, Chapter 39).

It is safe to assume that the implementation character set includes the decimal digits, the 26 alphabetic characters of English, and some of the basic punctuation characters. It is *not* safe to assume that:

- There are no more than 127 characters in an 8-bit character set (e.g., some sets provide 255 characters).
- There are no more alphabetic characters than English provides (most European languages provide more, e.g., æ, þ, and ß).
- The alphabetic characters are contiguous (EBCDIC leaves a gap between 'i' and 'j').
- Every character used to write C++ is available (e.g., some national character sets do not provide {, }, [,], |, and \).
- A char fits in 1 byte. There are embedded processors without byte accessing hardware for which a char is 4 bytes. Also, one could reasonably use a 16-bit Unicode encoding for the basic chars.

Whenever possible, we should avoid making assumptions about the representation of objects. This general rule applies even to characters.

Each character has an integer value in the character set used by the implementation. For example, the value of 'b' is 98 in the ASCII character set. Here is a loop that outputs the the integer value of any character you care to input:

```
void intval()
{
        for (char c; cin >> c; )
                cout << "the value of '" << c << "' is " << int{c} << '\n';
}
```

The notation int{c} gives the integer value for a character c ("the int we can construct from c"). The possibility of converting a char to an integer raises the question: is a char signed or unsigned? The 256 values represented by an 8-bit byte can be interpreted as the values 0 to 255 or as the values –127 to 127. No, not –128 to 127 as one might expect: the C++ standard leaves open the possibility of one's-complement hardware and that eliminates one value; thus, a use of –128 is nonportable. Unfortunately, the choice of signed or unsigned for a plain char is implementation-defined. C++ provides two types for which the answer is definite: signed char, which can hold at least the values –127 to 127, and unsigned char, which can hold at least the values 0 to 255.

Fortunately, the difference matters only for values outside the 0 to 127 range, and the most common characters are within that range.

Values outside that range stored in a plain char can lead to subtle portability problems. See §6.2.3.1 if you need to use more than one type of char or if you store integers in char variables.

Note that the character types are integral types (§6.2.1) so that arithmetic and bitwise logical operations (§10.3) apply. For example:

```
void digits()
{
        for (int i=0; i!=10; ++i)
                cout << static_cast<char>('0'+i);
}
```

This is a way of writing the ten digits to cout. The character literal '0' is converted to its integer value and i is added. The resulting int is then converted to a char and written to cout. Plain '0'+i is an int, so if I had left out the static_cast<char>, the output would have been something like 48, 49, and so on, rather than 0, 1, and so on.

6.2.3.1 Signed and Unsigned Characters

It is implementation-defined whether a plain char is considered signed or unsigned. This opens the possibility for some nasty surprises and implementation dependencies. For example:

```
char c = 255;    // 255 is "all ones," hexadecimal 0xFF
int i = c;
```

What will be the value of i? Unfortunately, the answer is undefined. On an implementation with 8-bit bytes, the answer depends on the meaning of the ‘‘all ones’’ char bit pattern when extended into an int. On a machine where a char is unsigned, the answer is 255. On a machine where a char is signed, the answer is –1. In this case, the compiler might warn about the conversion of the literal 255 to the char value –1. However, C++ does not offer a general mechanism for detecting this kind of problem. One solution is to avoid plain char and use the specific char types only. Unfortunately, some standard-library functions, such as strcmp(), take plain chars only (§43.4).

A char must behave identically to either a signed char or an unsigned char. However, the three char types are distinct, so you can’t mix pointers to different char types. For example:

```
void f(char c, signed char sc, unsigned char uc)
{
        char∗ pc = &uc;                    // error: no pointer conversion
        signed char∗ psc = pc;             // error: no pointer conversion
        unsigned char∗ puc = pc;           // error: no pointer conversion
        psc = puc;                         // error: no pointer conversion
}
```

Variables of the three char types can be freely assigned to each other. However, assigning a too-large value to a signed char (§10.5.2.1) is still undefined. For example:

```
void g(char c, signed char sc, unsigned char uc)
{
        c = 255;    // implementation-defined if plain chars are signed and have 8 bits
        c = sc;     // OK
        c = uc;     // implementation-defined if plain chars are signed and if uc's value is too large
        sc = uc;    // implementation defined if uc's value is too large
        uc = sc;    // OK: conversion to unsigned
        sc = c;     // implementation-defined if plain chars are unsigned and if c's value is too large
        uc = c;     // OK: conversion to unsigned
    }
```

To be concrete, assume that a **char** is 8 bits:

```
signed char sc = –160;
unsigned char uc = sc;     // uc == 116 (because 256-160==116)
cout << uc;                // print 't'

char count[256];           // assume 8-bit chars
++count[sc];               // likely disaster: out-of-range access
++count[uc];               // OK
```

None of these potential problems and confusions occur if you use plain **char** throughout and avoid negative character values.

6.2.3.2 Character Literals

A *character literal* is a single character enclosed in single quotes, for example, **'a'** and **'0'**. The type of a character literal is **char**. A character literal can be implicitly converted to its integer value in the character set of the machine on which the C++ program is to run. For example, if you are running on a machine using the ASCII character set, the value of **'0'** is **48**. The use of character literals rather than decimal notation makes programs more portable.

A few characters have standard names that use the backslash, **** , as an escape character:

Name	ASCII Name	C++ Name
Newline	NL (LF)	\n
Horizontal tab	HT	\t
Vertical tab	VT	\v
Backspace	BS	\b
Carriage return	CR	\r
Form feed	FF	\f
Alert	BEL	\a
Backslash	\	\\
Question mark	?	\?
Single quote	′	\'
Double quote	"	\"
Octal number	*ooo*	\ooo
Hexadecimal number	*hhh*	\x*hhh* ...

Despite their appearance, these are single characters.

We can represent a character from the implementation character set as a one-, two-, or three-digit octal number (**** followed by octal digits) or as a hexadecimal number (**\x** followed by hexadecimal digits). There is no limit to the number of hexadecimal digits in the sequence. A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. For example:

Octal	Hexadecimal	Decimal	ASCII
'\6'	'\x6'	6	ACK
'\60'	'\x30'	48	'0'
'\137'	'\x05f'	95	'_'

This makes it possible to represent every character in the machine's character set and, in particular, to embed such characters in character strings (see §7.3.2). Using any numeric notation for characters makes a program nonportable across machines with different character sets.

It is possible to enclose more than one character in a character literal, for example, 'ab'. Such uses are archaic, implementation-dependent, and best avoided. The type of such a multicharacter literal is int.

When embedding a numeric constant in a string using the octal notation, it is wise always to use three digits for the number. The notation is hard enough to read without having to worry about whether or not the character after a constant is a digit. For hexadecimal constants, use two digits. Consider these examples:

```
char v1[] = "a\xah\129";      // 6 chars: 'a' '\xa' 'h' '\12' '9' '\0'
char v2[] = "a\xah\127";      // 5 chars: 'a' '\xa' 'h' '\127' '\0'
char v3[] = "a\xad\127";      // 4 chars: 'a' '\xad' '\127' '\0'
char v4[] = "a\xad\0127";     // 5 chars: 'a' '\xad' '\012' '7' '\0'
```

Wide character literals are of the form L'ab' and are of type wchar_t. The number of characters between the quotes and their meanings are implementation-defined.

A C++ program can manipulate character sets that are much richer than the 127-character ASCII set, such as Unicode. Literals of such larger character sets are presented as sequences of four or eight hexadecimal digits preceded by a U or a u. For example:

```
U'\UFADEBEEF'
u'\uDEAD'
u'\xDEAD'
```

The shorter notation u'\uXXXX' is equivalent to U'\U0000XXXX' for any hexadecimal digit X. A number of hexadecimal digits different from four or eight is a lexical error. The meaning of the hexadecimal number is defined by the ISO/IEC 10646 standard and such values are called *universal character names*. In the C++ standard, universal character names are described in §iso.2.2, §iso.2.3, §iso.2.14.3, §iso.2.14.5, and §iso.E.

6.2.4 Integer Types

Like char, each integer type comes in three forms: "plain" int, signed int, and unsigned int. In addition, integers come in four sizes: short int, "plain" int, long int, and long long int. A long int can be referred to as plain long, and a long long int can be referred to as plain long long. Similarly, short is a synonym for short int, unsigned for unsigned int, and signed for signed int. No, there is no long short int equivalent to int.

The unsigned integer types are ideal for uses that treat storage as a bit array. Using an unsigned instead of an int to gain one more bit to represent positive integers is almost never a good idea. Attempts to ensure that some values are positive by declaring variables unsigned will typically be defeated by the implicit conversion rules (§10.5.1, §10.5.2.1).

Unlike plain chars, plain ints are always signed. The signed int types are simply more explicit synonyms for their plain int counterparts, rather than different types.

If you need more detailed control over integer sizes, you can use aliases from <cstdint> (§43.7), such as int64_t (a signed integer with exactly 64 bits), uint_fast16_t (an unsigned integer with exactly 8 bits, supposedly the fastest such integer), and int_least32_t (a signed integer with at least 32 bits, just like plain int). The plain integer types have well-defined minimal sizes (§6.2.8), so the <cstdint> are sometimes redundant and can be overused.

In addition to the standard integer types, an implementation may provide *extended integer types* (signed and unsigned). These types must behave like integers and are considered integer types when considering conversions and integer literal values, but they usually have greater range (occupy more space).

6.2.4.1 Integer Literals

Integer literals come in three guises: decimal, octal, and hexadecimal. Decimal literals are the most commonly used and look as you would expect them to:

```
7    1234    976    12345678901234567890
```

The compiler ought to warn about literals that are too long to represent, but an error is only guaranteed for {} initializers (§6.3.5).

A literal starting with zero followed by x or X (0x or 0X) is a hexadecimal (base 16) number. A literal starting with zero but not followed by x or X is an octal (base 8) number. For example:

Decimal	Octal	Hexadecimal
	0	0x0
2	02	0x2
63	077	0x3f
83	0123	0x63

The letters a, b, c, d, e, and f, or their uppercase equivalents, are used to represent 10, 11, 12, 13, 14, and 15, respectively. Octal and hexadecimal notations are most useful for expressing bit patterns. Using these notations to express genuine numbers can lead to surprises. For example, on a machine on which an int is represented as a two's complement 16-bit integer, 0xffff is the negative decimal number −1. Had more bits been used to represent an integer, it would have been the positive decimal number 65535.

The suffix U can be used to write explicitly unsigned literals. Similarly, the suffix L can be used to write explicitly long literals. For example, 3 is an int, 3U is an unsigned int, and 3L is a long int.

Combinations of suffixes are allowed. For example:

```
cout << 0xF0UL << ' ' << 0LU << '\n';
```

If no suffix is provided, the compiler gives an integer literal a suitable type based on its value and the implementation's integer sizes (§6.2.4.2).

It is a good idea to limit the use of nonobvious constants to a few well-commented **const** (§7.5), **constexpr** (§10.4), and enumerator (§8.4) initializers.

6.2.4.2 Types of Integer Literals

In general, the type of an integer literal depends on its form, value, and suffix:

- If it is decimal and has no suffix, it has the first of these types in which its value can be represented: **int**, **long int**, **long long int**.
- If it is octal or hexadecimal and has no suffix, it has the first of these types in which its value can be represented: **int**, **unsigned int**, **long int**, **unsigned long int**, **long long int**, **unsigned long long int**.
- If it is suffixed by **u** or **U**, its type is the first of these types in which its value can be represented: **unsigned int**, **unsigned long int**, **unsigned long long int**.
- If it is decimal and suffixed by **l** or **L**, its type is the first of these types in which its value can be represented: **long int**, **long long int**.
- If it is octal or hexadecimal and suffixed by **l** or **L**, its type is the first of these types in which its value can be represented: **long int**, **unsigned long int**, **long long int**, **unsigned long long int**.
- If it is suffixed by **ul**, **lu**, **uL**, **Lu**, **Ul**, **lU**, **UL**, or **LU**, its type is the first of these types in which its value can be represented: **unsigned long int**, **unsigned long long int**.
- If it is decimal and is suffixed by **ll** or **LL**, its type is **long long int**.
- If it is octal or hexadecimal and is suffixed by **ll** or **LL**, its type is the first of these types in which its value can be represented: **long long int**, **unsigned long long int**.
- If it is suffixed by **llu**, **llU**, **ull**, **Ull**, **LLu**, **LLU**, **uLL**, or **ULL**, its type is **unsigned long long int**.

For example, **100000** is of type **int** on a machine with 32-bit **int**s but of type **long int** on a machine with 16-bit **int**s and 32-bit **long**s. Similarly, **0XA000** is of type **int** on a machine with 32-bit **int**s but of type **unsigned int** on a machine with 16-bit **int**s. These implementation dependencies can be avoided by using suffixes: **100000L** is of type **long int** on all machines and **0XA000U** is of type **unsigned int** on all machines.

6.2.5 Floating-Point Types

The floating-point types represent floating-point numbers. A floating-point number is an approximation of a real number represented in a fixed amount of memory. There are three floating-point types: **float** (single-precision), **double** (double-precision), and **long double** (extended-precision).

The exact meaning of single-, double-, and extended-precision is implementation-defined. Choosing the right precision for a problem where the choice matters requires significant understanding of floating-point computation. If you don't have that understanding, get advice, take the time to learn, or use **double** and hope for the best.

6.2.5.1 Floating-Point Literals

By default, a floating-point literal is of type **double**. Again, a compiler ought to warn about floating-point literals that are too large to be represented. Here are some floating-point literals:

```
1.23   .23   0.23   1.   1.0   1.2e10   1.23e–15
```

Note that a space cannot occur in the middle of a floating-point literal. For example, 65.43 e–21 is not a floating-point literal but rather four separate lexical tokens (causing a syntax error):

```
65.43   e   –   21
```

If you want a floating-point literal of type float, you can define one using the suffix f or F:

```
3.14159265f   2.0f   2.997925F   2.9e–3f
```

If you want a floating-point literal of type long double, you can define one using the suffix l or L:

```
3.14159265L   2.0L   2.997925L   2.9e–3L
```

6.2.6 Prefixes and Suffixes

There is a minor zoo of suffixes indicating types of literals and also a few prefixes:

Arithmetic Literal Prefixes and Suffixes

Notation		∗fix	Meaning	Example	Reference	ISO
0		prefix	octal	0776	§6.2.4.1	§iso.2.14.2
0x	0X	prefix	hexadecimal	0xff	§6.2.4.1	§iso.2.14.2
u	U	suffix	unsigned	10U	§6.2.4.1	§iso.2.14.2
l	L	suffix	long	20000L	§6.2.4.1	§iso.2.14.2
ll	LL	suffix	long long	20000LL	§6.2.4.1	§iso.2.14.2
f	F	suffix	float	10f	§6.2.5.1	§iso.2.14.4
e	E	infix	floating-point	10e–4	§6.2.5.1	§iso.2.14.4
.		infix	floating-point	12.3	§6.2.5.1	§iso.2.14.4
'		prefix	char	'c'	§6.2.3.2	§iso.2.14.3
u'		prefix	char16_t	u'c'	§6.2.3.2	§iso.2.14.3
U'		prefix	char32_t	U'c'	§6.2.3.2	§iso.2.14.3
L'		prefix	wchar_t	L'c'	§6.2.3.2	§iso.2.14.3
"		prefix	string	"mess"	§7.3.2	§iso.2.14.5
R"		prefix	raw string	R"(\b)"	§7.3.2.1	§iso.2.14.5
u8"	u8R"	prefix	UTF-8 string	u8"foo"	§7.3.2.2	§iso.2.14.5
u"	uR"	prefix	UTF-16 string	u"foo"	§7.3.2.2	§iso.2.14.5
U"	UR"	prefix	UTF-32 string	U"foo"	§7.3.2.2	§iso.2.14.5
L"	LR"	prefix	wchar_t string	L"foo"	§7.3.2.2	§iso.2.14.5

Note that ‘‘string’’ here means ‘‘string literal’’ (§7.3.2) rather than ‘‘of type std::string.’’

Obviously, we could also consider . and e as infix and R" and u8" as the first part of a set of delimiters. However, I consider the nomenclature less important than giving an overview of the bewildering variety of literals.

The suffixes l and L can be combined with the suffixes u and U to express unsigned long types. For example:

```
1LU         // unsigned long
2UL         // unsigned long
3ULL        // unsigned long long
4LLU        // unsigned long long
5LUL        // error
```

The suffixes l and L can be used for floating-point literals to express long double. For example:

```
1L          // long int
1.0L        // long double
```

Combinations of R, L, and u prefixes are allowed, for example, uR"**(foo\(bar))**". Note the dramatic difference in the meaning of a U prefix for a character (unsigned) and for a string UTF-32 encoding (§7.3.2.2).

In addition, a user can define new suffixes for user-defined types. For example, by defining a user-defined literal operator (§19.2.6), we can get

```
"foo bar"s      // a literal of type std::string
123_km          // a literal of type Distance
```

Suffixes not starting with _ are reserved for the standard library.

6.2.7 void

The type void is syntactically a fundamental type. It can, however, be used only as part of a more complicated type; there are no objects of type void. It is used either to specify that a function does not return a value or as the base type for pointers to objects of unknown type. For example:

```
void x;         // error: there are no void objects
void& r;        // error: there are no references to void
void f();       // function f does not return a value (§12.1.4)
void* pv;       // pointer to object of unknown type (§7.2.1)
```

When declaring a function, you must specify the type of the value returned. Logically, you would expect to be able to indicate that a function didn't return a value by omitting the return type. However, that would make a mess of the grammar (§iso.A). Consequently, void is used as a "pseudo return type" to indicate that a function doesn't return a value.

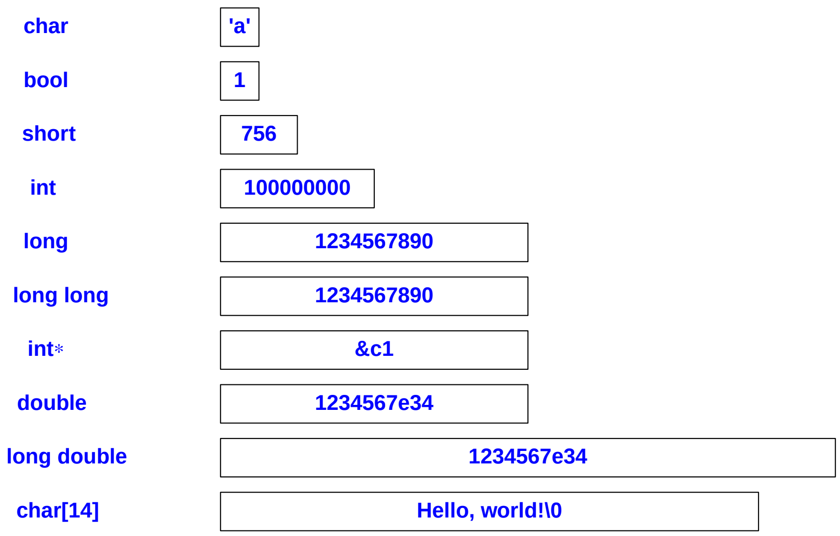
6.2.8 Sizes

Some of the aspects of C++'s fundamental types, such as the size of an int, are implementation-defined (§6.1). I point out these dependencies and often recommend avoiding them or taking steps to minimize their impact. Why should you bother? People who program on a variety of systems or use a variety of compilers care a lot because if they don't, they are forced to waste time finding and fixing obscure bugs. People who claim they don't care about portability usually do so because they use only a single system and feel they can afford the attitude that "the language is what my compiler implements." This is a narrow and shortsighted view. If your program is a success, it will be ported, so someone will have to find and fix problems related to implementation-dependent features. In addition, programs often need to be compiled with other compilers for the same system, and even a future release of your favorite compiler may do some things differently from the current one. It is far easier to know and limit the impact of implementation dependencies when a program is written than to try to untangle the mess afterward.

It is relatively easy to limit the impact of implementation-dependent language features. Limiting the impact of system-dependent library facilities is far harder. Using standard-library facilities wherever feasible is one approach.

The reason for providing more than one integer type, more than one unsigned type, and more than one floating-point type is to allow the programmer to take advantage of hardware characteristics. On many machines, there are significant differences in memory requirements, memory access times, and computation speed among the different varieties of fundamental types. If you know a machine, it is usually easy to choose, for example, the appropriate integer type for a particular variable. Writing truly portable low-level code is harder.

Here is a graphical representation of a plausible set of fundamental types and a sample string literal (§7.3.2):



On the same scale (.2 inch to a byte), a megabyte of memory would stretch about 3 miles (5 km) to the right.

Sizes of C++ objects are expressed in terms of multiples of the size of a char, so by definition the size of a char is 1. The size of an object or type can be obtained using the sizeof operator (§10.3). This is what is guaranteed about sizes of fundamental types:

- 1 ≡ sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)
- 1 ≤ sizeof(bool) ≤ sizeof(long)
- sizeof(char) ≤ sizeof(wchar_t) ≤ sizeof(long)
- sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)
- sizeof(N) ≡ sizeof(signed N) ≡ sizeof(unsigned N)

In that last line, N can be char, short, int, long, or long long. In addition, it is guaranteed that a char has at least 8 bits, a short at least 16 bits, and a long at least 32 bits. A char can hold a character of the machine's character set. The char type is supposed to be chosen by the implementation to be the most suitable type for holding and manipulating characters on a given computer; it is typically an 8-bit byte. Similarly, the int type is supposed to be chosen to be the most suitable for holding and manipulating integers on a given computer; it is typically a 4-byte (32-bit) word. It is unwise to assume more. For example, there are machines with 32-bit chars. It is extremely unwise to assume that the size of an int is the same as the size of a pointer; many machines (“64-bit architectures”) have pointers that are larger than integers. Note that it is not guaranteed that sizeof(long)<sizeof(long long) or that sizeof(double)<sizeof(long double).

Some implementation-defined aspects of fundamental types can be found by a simple use of sizeof, and more can be found in <limits>. For example:

```
#include <limits>    // §40.2
#include <iostream>

int main()
{
     cout << "size of long " << sizeof(1L) << '\n';
     cout << "size of long long " << sizeof(1LL) << '\n';

     cout << "largest float == " << std::numeric_limits<float>::max() << '\n';
     cout << "char is signed == " << std::numeric_limits<char>::is_signed << '\n';
}
```

The functions in <limits> (§40.2) are constexpr (§10.4) so that they can be used without run-time overhead and in contexts that require a constant expression.

The fundamental types can be mixed freely in assignments and expressions. Wherever possible, values are converted so as not to lose information (§10.5).

If a value v can be represented exactly in a variable of type T, a conversion of v to T is value-preserving. Conversions that are not value-preserving are best avoided (§2.2.2, §10.5.2.6).

If you need a specific size of integer, say, a 16-bit integer, you can #include the standard header <cstdint> that defines a variety of types (or rather type aliases; §6.5). For example:

```
int16_t x {0xaabb};                    // 2 bytes
int64_t xxxx {0xaaaabbbbccccdddd};     // 8 bytes
int_least16_t y;                       // at least 2 bytes (just like int)
int_least32_t yy                       // at least 4 bytes (just like long)
int_fast32_t z;                        // the fastest int type with at least 4 bytes
```

The standard header <cstddef> defines an alias that is very widely used in both standard-library declarations and user code: size_t is an implementation-defined unsigned integer type that can hold the size in bytes of every object. Consequently, it is used where we need to hold an object size. For example:

```
void∗ allocate(size_t n);  // get n bytes
```

Similarly, <cstddef> defines the signed integer type ptrdiff_t for holding the result of subtracting two pointers to get a number of elements.

6.2.9 Alignment

An object doesn't just need enough storage to hold its representation. In addition, on some machine architectures, the bytes used to hold it must have proper *alignment* for the hardware to access it efficiently (or in extreme cases to access it at all). For example, a 4-byte **int** often has to be aligned on a word (4-byte) boundary, and sometimes an 8-byte **double** has to be aligned on a word (8-byte) boundary. Of course, this is all very implementation specific, and for most programmers completely implicit. You can write good C++ code for decades without needing to be explicit about alignment. Where alignment most often becomes visible is in object layouts: sometimes **struct**s contain ''holes'' to improve alignment (§8.2.1).

The **alignof()** operator returns the alignment of its argument expression. For example:

```
auto ac = alignof('c');       // the alignment of a char
auto ai = alignof(1);         // the alignment of an int
auto ad = alignof(2.0);       // the alignment of a double

int a[20];
auto aa = alignof(a);         // the alignment of an int
```

Sometimes, we have to use alignment in a declaration, where an expression, such as **alignof(x+y)** is not allowed. Instead, we can use the type specifier **alignas**: **alignas(T)** means ''align just like a **T**.'' For example, we can set aside uninitialized storage for some type **X** like this:

```
void user(const vector<X>& vx)
{
      constexpr int bufmax = 1024;
      alignas(X) buffer[bufmax];      // uninitialized

      const int max = min(vx.size(),bufmax/sizeof(X));
      uninitialized_copy(vx.begin(),vx.begin()+max,buffer);
      // ...
}
```

6.3 Declarations

Before a name (identifier) can be used in a C++ program, it must be declared. That is, its type must be specified to inform the compiler what kind of entity the name refers to. For example:

```
char ch;
string s;
auto count = 1;
const double pi {3.1415926535897};
extern int error_number;

const char* name = "Njal";
const char* season[] = { "spring", "summer", "fall", "winter" };
vector<string> people { name, "Skarphedin", "Gunnar" };
```

```
struct Date { int d, m, y; };
int day(Date∗ p) { return p–>d; }
double sqrt(double);
template<class T> T abs(T a) { return a<0 ? –a : a; }

constexpr int fac(int n) { return (n<2)?1:n∗fac(n–1); }   // possible compile-time evaluation (§2.2.3)
constexpr double zz { ii∗fac(7) };                         // compile-time initialization

using Cmplx = std::complex<double>;                        // type alias (§3.4.5, §6.5)
struct User;                                               // type name
enum class Beer { Carlsberg, Tuborg, Thor };
namespace NS { int a; }
```

As can be seen from these examples, a declaration can do more than simply associate a type with a name. Most of these *declarations* are also *definitions*. A definition is a declaration that supplies all that is needed in a program for the use of an entity. In particular, if it takes memory to represent something, that memory is set aside by its definition. A different terminology deems declarations parts of an interface and definitions parts of an implementation. When taking that view, we try to compose interfaces out of declarations that can be replicated in separate files (§15.2.2); definitions that set aside memory do not belong in interfaces.

Assuming that these declarations are in the global scope (§6.3.4), we have:

```
char ch;                              // set aside memory for a char and initialize it to 0
auto count = 1;                       // set aside memory for an int initialized to 1
const char∗ name = "Njal";            // set aside memory for a pointer to char
                                      // set aside memory for a string literal "Njal"
                                      // initialize the pointer with the address of that string literal

struct Date { int d, m, y; };         // Date is a struct with three members
int day(Date∗ p) { return p–>d; }     // day is a function that executes the specified code

using Point = std::complex<short>;// Point is a name for std::complex<short>
```

Of the declarations above, only three are not also definitions:

```
double sqrt(double);          // function declaration
extern int error_number;      // variable declaration
struct User;                  // type name declaration
```

That is, if used, the entity they refer to must be defined elsewhere. For example:

```
double sqrt(double d) { /* ... */ }
int error_number = 1;
struct User { /* ... */ };
```

There must always be exactly one definition for each name in a C++ program (for the effects of **#include**, see §15.2.3). However, there can be many declarations.

All declarations of an entity must agree on its type. So, this fragment has two errors:

```
int count;
int count;        // error: redefinition
```

```
extern int error_number;
extern short error_number;          // error: type mismatch
```

This has no errors (for the use of **extern**, see §15.2):

```
extern int error_number;
extern int error_number; // OK: redeclaration
```

Some definitions explicitly specify a ‘‘value’’ for the entities they define. For example:

```
struct Date { int d, m, y; };
using Point = std::complex<short>;       // Point is a name for std::complex<short>
int day(Date* p) { return p–>d; }
const double pi {3.1415926535897};
```

For types, aliases, templates, functions, and constants, the ‘‘value’’ is permanent. For non-**const** data types, the initial value may be changed later. For example:

```
void f()
{
        int count {1};                  // initialize count to 1
        const char* name {"Bjarne"};    // name is a variable that points to a constant (§7.5)
        count = 2;                      // assign 2 to count
        name = "Marian";
}
```

Of the definitions, only two do not specify values:

```
char ch;
string s;
```

See §6.3.5 and §17.3.3 for explanations of how and when a variable is assigned a default value. Any declaration that specifies a value is a definition.

6.3.1 The Structure of Declarations

The structure of a declaration is defined by the C++ grammar (§iso.A). This grammar evolved over four decades, starting with the early C grammars, and is quite complicated. However, without too many radical simplifications, we can consider a declaration as having five parts (in order):

- Optional prefix specifiers (e.g., **static** or **virtual**)
- A base type (e.g., **vector<double>** or **const int**)
- A declarator optionally including a name (e.g., **p[7]**, **n**, or ***(*)[]**)
- Optional suffix function specifiers (e.g., **const** or **noexcept**)
- An optional initializer or function body (e.g., **={7,5,3}** or **{return x;}**)

Except for function and namespace definitions, a declaration is terminated by a semicolon. Consider a definition of an array of C-style strings:

```
const char* kings[] = { "Antigonus", "Seleucus", "Ptolemy" };
```

Here, the base type is **const char**, the declarator is ***kings[]**, and the initializer is the **=** followed by the **{}**-list.

A specifier is an initial keyword, such as **virtual** (§3.2.3, §20.3.2), **extern** (§15.2), or **constexpr** (§2.2.3), that specifies some non-type attribute of what is being declared.

A declarator is composed of a name and optionally some declarator operators. The most common declarator operators are:

Declarator Operators		
prefix	*	pointer
prefix	*const	constant pointer
prefix	*volatile	volatile pointer
prefix	&	lvalue reference (§7.7.1)
prefix	&&	rvalue reference (§7.7.2)
prefix	auto	function (using suffix return type)
postfix	[]	array
postfix	()	function
postfix	–>	returns from function

Their use would be simple if they were all either prefix or postfix. However, *, [], and () were designed to mirror their use in expressions (§10.3). Thus, * is prefix and [] and () are postfix. The postfix declarator operators bind tighter than the prefix ones. Consequently, char*kings[] is an array of pointers to char, whereas char(*kings)[] is a pointer to an array of char. We have to use parentheses to express types such as ‘‘pointer to array’’ and ‘‘pointer to function’’; see the examples in §7.2.

Note that the type cannot be left out of a declaration. For example:

```
const c = 7;     // error: no type

gt(int a, int b) // error: no return type
{
      return (a>b) ? a : b;
}

unsigned ui;     // OK: “unsigned”means “unsigned int”
long li;         // OK: “long” means “long int”
```

In this, standard C++ differs from early versions of C and C++ that allowed the first two examples by considering int to be the type when none was specified (§44.3). This ‘‘implicit int’’ rule was a source of subtle errors and much confusion.

Some types have names composed out of multiple keywords, such as long long and volatile int. Some type names don’t even look much like names, such as decltype(f(x)) (the return type of a call f(x); §6.3.6.3).

The volatile specifier is described in §41.4.

The alignas() specifier is described in §6.2.9.

6.3.2 Declaring Multiple Names

It is possible to declare several names in a single declaration. The declaration simply contains a list of comma-separated declarators. For example, we can declare two integers like this:

```
int x, y;        // int x; int y;
```

Operators apply to individual names only – and not to any subsequent names in the same declaration. For example:

```
int∗ p, y;          // int* p; int y;    NOT int* y;
int x, ∗q;          // int x; int* q;
int v[10], ∗pv;     // int v[10]; int* pv;
```

Such declarations with multiple names and nontrivial declarators make a program harder to read and should be avoided.

6.3.3 Names

A name (identifier) consists of a sequence of letters and digits. The first character must be a letter. The underscore character, _, is considered a letter. C++ imposes no limit on the number of characters in a name. However, some parts of an implementation are not under the control of the compiler writer (in particular, the linker), and those parts, unfortunately, sometimes do impose limits. Some run-time environments also make it necessary to extend or restrict the set of characters accepted in an identifier. Extensions (e.g., allowing the character $ in a name) yield nonportable programs. A C++ keyword (§6.3.3.1), such as new or int, cannot be used as a name of a user-defined entity. Examples of names are:

```
hello      this_is_a_most_unusually_long_identifier_that_is_better_avoided
DEFINED    foO      bAr        u_name     HorseSense
var0       var1     CLASS      _class     ___
```

Examples of character sequences that cannot be used as identifiers are:

```
012        a fool     $sys       class      3var
pay.due    foo˜bar    .name      if
```

Nonlocal names starting with an underscore are reserved for special facilities in the implementation and the run-time environment, so such names should not be used in application programs. Similarly, names starting with a double underscore (__) or an underscore followed by an uppercase letter (e.g., _Foo) are reserved (§iso.17.6.4.3).

When reading a program, the compiler always looks for the longest string of characters that could make up a name. Hence, var10 is a single name, not the name var followed by the number 10. Also, elseif is a single name, not the keyword else followed by the keyword if.

Uppercase and lowercase letters are distinct, so Count and count are different names, but it is often unwise to choose names that differ only by capitalization. In general, it is best to avoid names that differ only in subtle ways. For example, in some fonts, the uppercase ‘‘o’’ (O) and zero (0) can be hard to tell apart, as can the lowercase ‘‘L’’ (l), uppercase ‘‘i’’ (I), and one (1). Consequently, l0, lO, l1, ll, and I1l are poor choices for identifier names. Not all fonts have the same problems, but most have some.

Names from a large scope ought to have relatively long and reasonably obvious names, such as vector, Window_with_border, and Department_number. However, code is clearer if names used only in a small scope have short, conventional names such as x, i, and p. Functions (Chapter 12), classes (Chapter 16), and namespaces (§14.3.1) can be used to keep scopes small. It is often useful to keep frequently used names relatively short and reserve really long names for infrequently used entities.

Choose names to reflect the meaning of an entity rather than its implementation. For example, **phone_book** is better than **number_vector** even if the phone numbers happen to be stored in a **vector** (§4.4). Do not encode type information in a name (e.g., **pcname** for a name that's a **char*** or **icount** for a count that's an **int**) as is sometimes done in languages with dynamic or weak type systems:

- Encoding types in names lowers the abstraction level of the program; in particular, it prevents generic programming (which relies on a name being able to refer to entities of different types).
- The compiler is better at keeping track of types than you are.
- If you want to change the type of a name (e.g., use a **std::string** to hold the name), you'll have to change every use of the name (or the type encoding becomes a lie).
- Any system of type abbreviations you can come up with will become overelaborate and cryptic as the variety of types you use increases.

Choosing good names is an art.

Try to maintain a consistent naming style. For example, capitalize names of user-defined types and start names of non-type entities with a lowercase letter (for example, **Shape** and **current_token**). Also, use all capitals for macros (if you must use macros (§12.6); for example, **HACK**) and never for non-macros (not even for non-macro constants). Use underscores to separate words in an identifier; **number_of_elements** is more readable than **numberOfElements**. However, consistency is hard to achieve because programs are typically composed of fragments from different sources and several different reasonable styles are in use. Be consistent in your use of abbreviations and acronyms. Note that the language and the standard library use lowercase for types; this can be seen as a hint that they are part of the standard.

6.3.3.1 Keywords

The C++ keywords are:

C++ Keywords

alignas	alignof	and	and_eq	asm	auto
bitand	bitor	bool	break	case	catch
char	char16_t	char32_t	class	compl	const
constexpr	const_cast	continue	decltype	default	delete
do	double	dynamic_cast	else	enum	explicit
extern	false	float	for	friend	goto
if	inline	int	long	mutable	namespace
new	noexcept	not	not_eq	nullptr	operator
or	or_eq	private	protected	public	register
reinterpret_cast	return	short	signed	sizeof	static
static_assert	static_cast	struct	switch	template	this
thread_local	throw	true	try	typedef	typeid
typename	union	unsigned	using	virtual	void
volatile	wchar_t	while	xor	xor_eq	

In addition, the word **export** is reserved for future use.

6.3.4 Scope

A declaration introduces a name into a scope; that is, a name can be used only in a specific part of the program text.

- *Local scope*: A name declared in a function (Chapter 12) or lambda (§11.4) is called a *local name*. Its scope extends from its point of declaration to the end of the block in which its declaration occurs. A *block* is a section of code delimited by a `{}` pair. Function and lambda parameter names are considered local names in the outermost block of their function or lambda.
- *Class scope*: A name is called a *member name* (or a *class member name*) if it is defined in a class outside any function, class (Chapter 16), enum class (§8.4.1), or other namespace. Its scope extends from the opening `{` of the class declaration to the end of the class declaration.
- *Namespace scope*: A name is called a *namespace member name* if it is defined in a namespace (§14.3.1) outside any function, lambda (§11.4), class (Chapter 16), enum class (§8.4.1), or other namespace. Its scope extends from the point of declaration to the end of its namespace. A namespace name may also be accessible from other translation units (§15.2).
- *Global scope*: A name is called a *global name* if it is defined outside any function, class (Chapter 16), enum class (§8.4.1), or namespace (§14.3.1). The scope of a global name extends from the point of declaration to the end of the file in which its declaration occurs. A global name may also be accessible from other translation units (§15.2). Technically, the global namespace is considered a namespace, so a global name is an example of a namespace member name.
- *Statement scope*: A name is in a statement scope if it is defined within the `()` part of a `for`-, `while`-, `if`-, or `switch`-statement. Its scope extends from its point of declaration to the end of its statement. All names in statement scope are local names.
- *Function scope*: A label (§9.6) is in scope from its point of declaration until the end of the function.

A declaration of a name in a block can hide a declaration in an enclosing block or a global name. That is, a name can be redefined to refer to a different entity within a block. After exit from the block, the name resumes its previous meaning. For example:

```
int x;                  // global x

void f()
{
        int x;          // local x hides global x
        x = 1;          // assign to local x
        {
                int x;  // hides first local x
                x = 2;  // assign to second local x
        }
        x = 3;          // assign to first local x
}

int* p = &x;            // take address of global x
```

Hiding names is unavoidable when writing large programs. However, a human reader can easily fail to notice that a name has been hidden (also known as *shadowed*). Because such errors are relatively rare, they can be very difficult to find. Consequently, name hiding should be minimized. Using names such as **i** and **x** for global variables or for local variables in a large function is asking for trouble.

A hidden global name can be referred to using the scope resolution operator, **::**. For example:

```
int x;

void f2()
{
        int x = 1;  // hide global x
        ::x = 2;    // assign to global x
        x = 2;      // assign to local x
        // ...
}
```

There is no way to use a hidden local name.

The scope of a name that is not a class member starts at its point of declaration, that is, after the complete declarator and before the initializer. This implies that a name can be used even to specify its own initial value. For example:

```
int x = 97;

void f3()
{
        int x = x;        // perverse: initialize x with its own (uninitialized) value
}
```

A good compiler warns if a variable is used before it has been initialized.

It is possible to use a single name to refer to two different objects in a block without using the **::** operator. For example:

```
int x = 11;

void f4()                 // perverse: use of two different objects both called x in a single scope
{
        int y = x;        // use global x: y = 11
        int x = 22;
        y = x;            // use local x: y = 22
}
```

Again, such subtleties are best avoided.

The names of function arguments are considered declared in the outermost block of a function. For example:

```
void f5(int x)
{
        int x;      // error
}
```

This is an error because x is defined twice in the same scope.

Names introduced in a for-statement are local to that statement (in statement scope). This allows us to use conventional names for loop variables repeatedly in a function. For example:

```
void f(vector<string>& v, list<int>& lst)
{
        for (const auto& x : v) cout << x << '\n';
        for (auto x : lst) cout << x << '\n';
        for (int i = 0, i!=v.size(), ++i) cout << v[i] << '\n';
        for (auto i : {1, 2, 3, 4, 5, 6, 7}) cout << i << '\n';
}
```

This contains no name clashes.

A declaration is not allowed as the only statement on the branch of an if-statement (§9.4.1).

6.3.5 Initialization

If an initializer is specified for an object, that initializer determines the initial value of an object. An initializer can use one of four syntactic styles:

```
X a1 {v};
X a2 = {v};
X a3 = v;
X a4(v);
```

Of these, only the first can be used in every context, and I strongly recommend its use. It is clearer and less error-prone than the alternatives. However, the first form (used for a1) is new in C++11, so the other three forms are what you find in older code. The two forms using = are what you use in C. Old habits die hard, so I sometimes (inconsistently) use = when initializing a simple variable with a simple value. For example:

```
int x1 = 0;
char c1 = 'z';
```

However, anything much more complicated than that is better done using {}. Initialization using {}, *list initialization*, does not allow narrowing (§iso.8.5.4). That is:

- An integer cannot be converted to another integer that cannot hold its value. For example, char to int is allowed, but not int to char.
- A floating-point value cannot be converted to another floating-point type that cannot hold its value. For example, float to double is allowed, but not double to float.
- A floating-point value cannot be converted to an integer type.
- An integer value cannot be converted to a floating-point type.

For example:

```
void f(double val, int val2)
{
        int x2 = val;           // if val==7.9, x2 becomes 7
        char c2 = val2;         // if val2==1025, c2 becomes 1
        int x3 {val};           // error: possible truncation
        char c3 {val2};         // error: possible narrowing

        char c4 {24};           // OK: 24 can be represented exactly as a char
        char c5 {264};          // error (assuming 8-bit chars): 264 cannot be represented as a char

        int x4 {2.0};           // error: no double to int value conversion

        // ...
}
```

See §10.5 for the conversion rules for built-in types.

There is no advantage to using {} initialization, and one trap, when using auto to get the type determined by the initializer. The trap is that if the initializer is a {}-list, we may not want its type deduced (§6.3.6.2). For example:

```
auto z1 {99};    // z1 is an initializer_list<int>
auto z2 = 99;    // z2 is an int
```

So prefer = when using auto.

It is possible to define a class so that an object can be initialized by a list of values and alternatively be constructed given a couple of arguments that are not simply values to be stored. The classical example is a vector of integers:

```
vector<int> v1 {99};        // v1 is a vector of 1 element with the value 99
vector<int> v2(99);         // v2 is a vector of 99 elements each with the default value 0
```

I use the explicit invocation of a constructor, (99), to get the second meaning. Most types do not offer such confusing alternatives – even most vectors do not; for example:

```
vector<string> v1{"hello!"};        // v1 is a vector of 1 element with the value "hello!"
vector<string> v2("hello!");        // error: no vector constructor takes a string literal
```

So, prefer {} initialization over alternatives unless you have a strong reason not to.

The empty initializer list, {}, is used to indicate that a default value is desired. For example:

```
int x4 {};              // x4 becomes 0
double d4 {};           // d4 becomes 0.0
char* p {};             // p becomes nullptr
vector<int> v4{};       // v4 becomes the empty vector
string s4 {};           // s4 becomes ""
```

Most types have a default value. For integral types, the default value is a suitable representation of zero. For pointers, the default value is nullptr (§7.2.2). For user-defined types, the default value (if any) is determined by the type's constructors (§17.3.3).

For user-defined types, there can be a distinction between direct initialization (where implicit conversions are allowed) and copy initialization (where they are not); see §16.2.6.

Initialization of particular kinds of objects is discussed where appropriate:

- Pointers: §7.2.2, §7.3.2, §7.4
- References: §7.7.1 (lvalues), §7.7.2 (rvalues)

- Arrays: §7.3.1, §7.3.2
- Constants: §10.4
- Classes: §17.3.1 (not using constructors), §17.3.2 (using constructors), §17.3.3 (default), §17.4 (member and base), §17.5 (copy and move)
- User-defined containers: §17.3.4

6.3.5.1 Missing Initializers

For many types, including all built-in types, it is possible to leave out the initializer. If you do that – and that has unfortunately been common – the situation is more complicated. If you don't like the complications, just initialize consistently. The only really good case for an uninitialized variable is a large input buffer. For example:

```
constexpr int max = 1024*1024;
char buf[max];
some_stream.get(buf,max);    // read at most max characters into buf
```

We could easily have initialized buf:

```
char buf[max] {};            // initialize every char to 0
```

By redundantly initializing, we would have suffered a performance hit which just might have been significant. Avoid such low-level use of buffers where you can, and don't leave such buffers uninitialized unless you know (e.g., from measurement) that the optimization compared to using an initialized array is significant.

If no initializer is specified, a global (§6.3.4), namespace (§14.3.1), local static (§12.1.8), or static member (§16.2.12) (collectively called *static objects*) is initialized to {} of the appropriate type. For example:

```
int a;          // means "int a{};" so that a becomes 0
double d;       // means "double d{};" so that d becomes 0.0
```

Local variables and objects created on the free store (sometimes called *dynamic objects* or *heap objects*; §11.2) are not initialized by default unless they are of user-defined types with a default constructor (§17.3.3). For example:

```
void f()
{
     int x;                        // x does not have a well-defined value
     char buf[1024];               // buf[i] does not have a well-defined value

     int* p {new int};             // *p does not have a well-defined value
     char* q {new char[1024]};     // q[i] does not have a well-defined value

     string s;                     // s=="" because of string's default constructor
     vector<char> v;               // v=={} because of vector's default constructor

     string* ps {new string};      // *ps is "" because of string's default constructor
     // ...
}
```

If you want initialization of local variables of built-in type or objects of built-in type created with **new**, use **{}**. For example:

```
void ff()
{
        int x {};                      // x becomes 0
        char buf[1024]{};              // buf[i] becomes 0 for all i

        int∗ p {new int{10}};          // *p becomes 10
        char∗ q {new char[1024]{}};    // q[i] becomes 0 for all i

        // ...
}
```

A member of an array or a class is default initialized if the array or structure is.

6.3.5.2 Initializer Lists

So far, we have considered the cases of no initializer and one initializer value. More complicated objects can require more than one value as an initializer. This is primarily handled by initializer lists delimited by **{** and **}**. For example:

```
int a[] = { 1, 2 };                         // array initializer
struct S { int x, string s };
S s = { 1, "Helios" };                      // struct initializer
complex<double> z = { 0, pi };              // use constructor
vector<double> v = { 0.0, 1.1, 2.2, 3.3 };  // use list constructor
```

For C-style initialization of arrays, see §7.3.1. For C-style structures, see §8.2. For user-defined types with constructors, see §2.3.2 or §16.2.5. For initializer-list constructors, see §17.3.4.

In the cases above, the **=** is redundant. However, some prefer to add it to emphasize that a set of values are used to initialize a set of member variables.

In some cases, function-style argument lists can also be used (§2.3, §16.2.5). For example:

```
complex<double> z(0,pi);      // use constructor
vector<double> v(10,3.3);     // use constructor: v gets 10 elements initialized to 3.3
```

In a declaration, an empty pair of parentheses, **()**, always means ‘‘function’’ (§12.1). So, if you want to be explicit about ‘‘use default initialization’’ you need **{}**. For example:

```
complex<double> z1(1,2);      // function-style initializer (initialization by constructor)
complex<double> f1();         // function declaration

complex<double> z2 {1,2};     // initialization by constructor to {1,2}
complex<double> f2 {};        // initialization by constructor to the default value {0,0}
```

Note that initialization using the **{}** notation does not narrow (§6.3.5).

When using **auto**, a **{}**-list has its type deduced to **std::initializer_list<T>**. For example:

```
auto x1 {1,2,3,4};           // x1 is an initializer_list<int>
auto x2 {1.0, 2.25, 3.5 };   // x2 is an initializer_list of<double>
auto x3 {1.0,2};             // error: cannot deduce the type of {1.0,2} (§6.3.6.2)
```

6.3.6 Deducing a Type: auto and decltype()

The language provides two mechanisms for deducing a type from an expression:

- auto for deducing a type of an object from its initializer; the type can be the type of a variable, a const, or a constexpr.
- decltype(expr) for deducing the type of something that is not a simple initializer, such as the return type for a function or the type of a class member.

The deduction done here is very simple: auto and decltype() simply report the type of an expression already known to the compiler.

6.3.6.1 The auto Type Specifier

When a declaration of a variable has an initializer, we don't need to explicitly specify a type. Instead, we can let the variable have the type of its initializer. Consider:

```
int a1 = 123;
char a2 = 123;
auto a3 = 123;  // the type of a3 is "int"
```

The type of the integer literal 123 is int, so a3 is an int. That is, auto is a placeholder for the type of the initializer.

There is not much advantage in using auto instead of int for an expression as simple as 123. The harder the type is to write and the harder the type is to know, the more useful auto becomes. For example:

```
template<class T> void f1(vector<T>& arg)
{
      for (vector<T>::iterator p = arg.begin(); p!=arg.end(); ++p)
            *p = 7;

      for (auto p = arg.begin(); p!=arg.end(); ++p)
            *p = 7;
}
```

The loop using auto is the more convenient to write and the easier to read. Also, it is more resilient to code changes. For example, if I changed arg to be a list, the loop using auto would still work correctly whereas the first loop would need to be rewritten. So, unless there is a good reason not to, use auto in small scopes.

If a scope is large, mentioning a type explicitly can help localize errors. That is, compared to using a specific type, using auto can delay the detection of type errors. For example:

```
void f(double d)
{
      constexpr auto max = d+7;
      int a[max];              // error: array bound not an integer
      // ...
}
```

If auto causes surprises, the best cure is typically to make functions smaller, which most often is a good idea anyway (§12.1).

We can decorate a deduced type with specifiers and modifiers (§6.3.1), such as const and & (reference; §7.7). For example:

```
void f(vector<int>& v)
{
        for (const auto& x : v) {    // x is a const int&
                // ...
        }
}
```

Here, auto is determined by the element type of v, that is, int.

Note that the type of an expression is never a reference because references are implicitly dereferenced in expressions (§7.7). For example:

```
void g(int& v)
{
        auto x = v;        // x is an int (not an int&)
        auto& y = v;       // y is an int&
}
```

6.3.6.2 auto and {}-lists

When we explicitly mention the type of an object we are initializing, we have two types to consider: the type of the object and the type of the initializer. For example:

```
char v1 = 12345;        // 12345 is an int
int v2 = 'c';           // 'c' is a char
T v3 = f();
```

By using the {}-initializer syntax for such definitions, we minimize the chances for unfortunate conversions:

```
char v1 {12345};        // error: narrowing
int v2 {'c'};           // fine: implicit char->int conversion
T v3 {f()};             // works if and only if the type of f() can be implicitly converted to a T
```

When we use auto, there is only one type involved, the type of the initializer, and we can safely use the = syntax:

```
auto v1 = 12345;        // v1 is an int
auto v2 = 'c';          // v2 is a char
auto v3 = f();          // v3 is of some appropriate type
```

In fact, it can be an advantage to use the = syntax with auto, because the {}-list syntax might surprise someone:

```
auto v1 {12345};        // v1 is a list of int
auto v2 {'c'};          // v2 is a list of char
auto v3 {f()};          // v3 is a list of some appropriate type
```

This is logical. Consider:

```
auto x0 {};          // error: cannot deduce a type
auto x1 {1};         // list of int with one element
auto x2 {1,2};       // list of int with two elements
auto x3 {1,2,3};     // list of int with three elements
```

The type of a homogeneous list of elements of type T is taken to be of type initializer_list<T> (§3.2.1.3, §11.3.3). In particular, the type of x1 is *not* deduced to be int. Had it been, what would be the types of x2 and x3?

Consequently, I recommend using = rather than {} for objects specified auto whenever we don't mean "list."

6.3.6.3 The decltype() Specifier

We can use auto when we have a suitable initializer. But sometimes, we want to have a type deduced without defining an initialized variable. Then, we can use a declaration type specifier: decltype(expr) is the declared type of expr. This is mostly useful in generic programming. Consider writing a function that adds two matrices with potentially different element types. What should be the type of the result of the addition? A matrix, of course, but what might its element type be? The obvious answer is that the element type of the sum is the type of the sum of the elements. So, I can declare:

```
template<class T, class U>
auto operator+(const Matrix<T>&  a, const Matrix<U>&  b) −> Matrix<decltype(T{}+U{})>;
```

I use the suffix return type syntax (§12.1) to be able to express the return type in terms of the arguments: Matrix<decltype(T{}+U{})>. That is, the result is a Matrix with the element type being what you get from adding a pair of elements from the argument Matrixes: T{}+U{}.

In the definition, I again need decltype() to express Matrix's element type:

```
template<class T, class U>
auto operator+(const Matrix<T>&  a, const Matrix<U>&  b) −> Matrix<decltype(T{}+U{})>
{
    Matrix<decltype(T{}+U{})> res;
    for (int i=0; i!=a.rows(); ++i)
        for (int j=0; j!=a.cols(); ++j)
            res(i,j) += a(i,j) + b(i,j);
    return res;
}
```

6.4 Objects and Values

We can allocate and use objects that do not have names (e.g., created using new), and it is possible to assign to strange-looking expressions (e.g., ∗p[a+10]=7). Consequently, we need a name for "something in memory." This is the simplest and most fundamental notion of an object. That is, an *object* is a contiguous region of storage; an *lvalue* is an expression that refers to an object. The word "lvalue" was originally coined to mean "something that can be on the left-hand side of an assignment." However, not every lvalue may be used on the left-hand side of an assignment; an lvalue can refer to a constant (§7.7). An lvalue that has not been declared `const` is often called a *modifiable lvalue*. This simple and low-level notion of an object should not be confused with the notions of class object and object of polymorphic type (§3.2.2, §20.3.2).

6.4.1 Lvalues and Rvalues

To complement the notion of an lvalue, we have the notion of an *rvalue*. Roughly, rvalue means "a value that is not an lvalue," such as a temporary value (e.g., the value returned by a function).

If you need to be more technical (say, because you want to read the ISO C++ standard), you need a more refined view of lvalue and rvalue. There are two properties that matter for an object when it comes to addressing, copying, and moving:

- *Has identity*: The program has the name of, pointer to, or reference to the object so that it is possible to determine if two objects are the same, whether the value of the object has changed, etc.
- *Movable*: The object may be moved from (i.e., we are allowed to move its value to another location and leave the object in a valid but unspecified state, rather than copying; §17.5).

It turns out that three of the four possible combinations of those two properties are needed to precisely describe the C++ language rules (we have no need for objects that do not have identity and cannot be moved). Using "`m` for movable" and "`i` for has identity," we can represent this classification of expressions graphically:

$$\begin{tikzcd}
\text{lvalue } \{\texttt{i\&!m}\} & & \text{xvalue } \{\texttt{i\&m}\} & & \text{prvalue } \{\texttt{!i\&m}\} \\
& \text{glvalue } \{\texttt{i}\} \arrow[ul] \arrow[ur] & & \text{rvalue } \{\texttt{m}\} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

So, a classical lvalue is something that has identity and cannot be moved (because we could examine it after a move), and a classical rvalue is anything that we are allowed to move from. The other alternatives are *prvalue* ("pure rvalue"), *glvalue* ("generalized lvalue"), and *xvalue* ("x" for "extraordinary" or "expert only"; the suggestions for the meaning of this "x" have been quite imaginative). For example:

```
void f(vector<string>& vs)
{
      vector<string>& v2 = std::move(vs);      // move vs to v2
      // ...
}
```

Here, `std::move(vs)` is an xvalue: it clearly has identity (we can refer to it as `vs`), but we have explicitly given permission for it to be moved from by calling `std::move()` (§3.3.2, §35.5.1).

For practical programming, thinking in terms of rvalue and lvalue is usually sufficient. Note that every expression is either an lvalue or an rvalue, but not both.

6.4.2 Lifetimes of Objects

The *lifetime* of an object starts when its constructor completes and ends when its destructor starts executing. Objects of types without a declared constructor, such as an `int`, can be considered to have default constructors and destructors that do nothing.

We can classify objects based on their lifetimes:

- *Automatic*: Unless the programmer specifies otherwise (§12.1.8, §16.2.12), an object declared in a function is created when its definition is encountered and destroyed when its name goes out of scope. Such objects are sometimes called *automatic* objects. In a typical implementation, automatic objects are allocated on the stack; each call of the function gets its own *stack frame* to hold its automatic objects.
- *Static*: Objects declared in global or namespace scope (§6.3.4) and `static`s declared in functions (§12.1.8) or classes (§16.2.12) are created and initialized once (only) and ''live'' until the program terminates (§15.4.3). Such objects are called *static* objects. A static object has the same address throughout the life of a program execution. Static objects can cause serious problems in a multi-threaded program because they are shared among all threads and typically require locking to avoid data races (§5.3.1, §42.3).
- *Free store*: Using the `new` and `delete` operators, we can create objects whose lifetimes are controlled directly (§11.2).
- *Temporary objects* (e.g., intermediate results in a computation or an object used to hold a value for a reference to `const` argument): their lifetime is determined by their use. If they are bound to a reference, their lifetime is that of the reference; otherwise, they ''live'' until the end of the full expression of which they are part. A *full expression* is an expression that is not part of another expression. Typically, temporary objects are automatic.
- *Thread-local* objects; that is, objects declared `thread_local` (§42.2.8): such objects are created when their thread is and destroyed when their thread is.

Static and *automatic* are traditionally referred to as *storage classes*.

Array elements and nonstatic class members have their lifetimes determined by the object of which they are part.

6.5 Type Aliases

Sometimes, we need a new name for a type. Possible reasons include:

- The original name is too long, complicated, or ugly (in some programmer's eyes).
- A programming technique requires different types to have the same name in a context.
- A specific type is mentioned in one place only to simplify maintenance.

For example:

```
using Pchar = char*;              // pointer to character
using PF = int(*)(double);        // pointer to function taking a double and returning an int
```

Similar types can define the same name as a member alias:

```
template<class T>
class vector {
      using value_type = T;          // every container has a value_type
      // ...
};
```

```
template<class T>
class list {
        using value_type = T;          // every container has a value_type
        // ...
};
```

For good and bad, type aliases are synonyms for other types rather than distinct types. That is, an alias refers to the type for which it is an alias. For example:

```
Pchar p1 = nullptr;          // p1 is a char*
char∗ p3 = p1;               // fine
```

People who would like to have distinct types with identical semantics or identical representation should look at enumerations (§8.4) and classes (Chapter 16).

An older syntax using the keyword **typedef** and placing the name being declared where it would have been in a declaration of a variable can equivalently be used in many contexts. For example:

```
typedef int int32_t;             // equivalent to "using int32_t = int;"
typedef short int16_t;           // equivalent to "using int16_t = short;"
typedef void(∗PtoF)(int);        // equivalent to "using PtoF = void(*)(int);"
```

Aliases are used when we want to insulate our code from details of the underlying machine. The name **int32_t** indicates that we want it to represent a 32-bit integer. Having written our code in terms of **int32_t**, rather than ‘‘plain **int**,’’ we can port our code to a machine with **sizeof(int)==2** by redefining the single occurrence of **int32_t** in our code to use a longer integer:

```
using int32_t = long;
```

The **_t** suffix is conventional for aliases (‘‘typedefs’’). The **int16_t**, **int32_t**, and other such aliases can be found in **<stdint>** (§43.7). Note that naming a type after its representation rather than its purpose is not necessarily a good idea (§6.3.3).

The **using** keyword can also be used to introduce a **template** alias (§23.6). For example:

```
template<typename T>
        using Vector = std::vector<T, My_allocator<T>>;
```

We cannot apply type specifiers, such as **unsigned**, to an alias. For example:

```
using Char = char;
using Uchar = unsigned Char;          // error
using Uchar = unsigned char;          // OK
```

6.6 Advice

[1] For the final word on language definition issues, see the ISO C++ standard; §6.1.
[2] Avoid unspecified and undefined behavior; §6.1.
[3] Isolate code that must depend on implementation-defined behavior; §6.1.
[4] Avoid unnecessary assumptions about the numeric value of characters; §6.2.3.2, §10.5.2.1.
[5] Remember that an integer starting with a **0** is octal; §6.2.4.1.

[6] Avoid “magic constants”; §6.2.4.1.
[7] Avoid unnecessary assumptions about the size of integers; §6.2.8.
[8] Avoid unnecessary assumptions about the range and precision of floating-point types; §6.2.8.
[9] Prefer plain `char` over `signed char` and `unsigned char`; §6.2.3.1.
[10] Beware of conversions between signed and unsigned types; §6.2.3.1.
[11] Declare one name (only) per declaration; §6.3.2.
[12] Keep common and local names short, and keep uncommon and nonlocal names longer; §6.3.3.
[13] Avoid similar-looking names; §6.3.3.
[14] Name an object to reflect its meaning rather than its type; §6.3.3.
[15] Maintain a consistent naming style; §6.3.3.
[16] Avoid `ALL_CAPS` names; §6.3.3.
[17] Keep scopes small; §6.3.4.
[18] Don’t use the same name in both a scope and an enclosing scope; §6.3.4.
[19] Prefer the `{}`-initializer syntax for declarations with a named type; §6.3.5.
[20] Prefer the `=` syntax for the initialization in declarations using `auto`; §6.3.5.
[21] Avoid uninitialized variables; §6.3.5.1.
[22] Use an alias to define a meaningful name for a built-in type in cases in which the built-in type used to represent a value might change; §6.5.
[23] Use an alias to define synonyms for types; use enumerations and classes to define new types; §6.5.

This page intentionally left blank

7

Pointers, Arrays, and References

The sublime and the ridiculous
are often so nearly related that
it is difficult to class them separately.
– Thomas Paine

7.1 Introduction

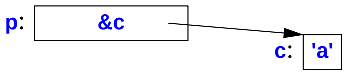
This chapter deals with the basic language mechanisms for referring to memory. Obviously, we can refer to an object by name, but in C++ (most) objects "have identity." That is, they reside at a specific address in memory, and an object can be accessed if you know its address and its type. The language constructs for holding and using addresses are pointers and references.

7.2 Pointers

For a type **T**, **T*** is the type ‘‘pointer to **T**.’’ That is, a variable of type **T*** can hold the address of an object of type **T**. For example:

```
char c = 'a';
char* p = &c;          // p holds the address of c; & is the address-of operator
```

or graphically:



The fundamental operation on a pointer is *dereferencing*, that is, referring to the object pointed to by the pointer. This operation is also called *indirection*. The dereferencing operator is (prefix) unary *. For example:

```
char c = 'a';
char* p = &c;    // p holds the address of c; & is the address-of operator
char c2 = *p;    // c2 == 'a'; * is the dereference operator
```

The object pointed to by **p** is **c**, and the value stored in **c** is **'a'**, so the value of ***p** assigned to **c2** is **'a'**.

It is possible to perform some arithmetic operations on pointers to array elements (§7.4).

The implementation of pointers is intended to map directly to the addressing mechanisms of the machine on which the program runs. Most machines can address a byte. Those that can't tend to have hardware to extract bytes from words. On the other hand, few machines can directly address an individual bit. Consequently, the smallest object that can be independently allocated and pointed to using a built-in pointer type is a **char**. Note that a **bool** occupies at least as much space as a **char** (§6.2.8). To store smaller values more compactly, you can use the bitwise logical operations (§11.1.1), bit-fields in structures (§8.2.7), or a **bitset** (§34.2.2).

The *, meaning ‘‘pointer to,’’ is used as a suffix for a type name. Unfortunately, pointers to arrays and pointers to functions need a more complicated notation:

```
int* pi;               // pointer to int
char** ppc;            // pointer to pointer to char
int* ap[15];           // array of 15 pointers to ints
int (*fp)(char*);      // pointer to function taking a char* argument; returns an int
int* f(char*);         // function taking a char* argument; returns a pointer to int
```

See §6.3.1 for an explanation of the declaration syntax and §iso.A for the complete grammar.

Pointers to functions can be useful; they are discussed in §12.5. Pointers to class members are presented in §20.6.

7.2.1 void*

In low-level code, we occasionally need to store or pass along an address of a memory location without actually knowing what type of object is stored there. A **void*** is used for that. You can read **void*** as ‘‘pointer to an object of unknown type.’’

A pointer to any type of object can be assigned to a variable of type void*, but a pointer to function (§12.5) or a pointer to member (§20.6) cannot. In addition, a void* can be assigned to another void*, void*s can be compared for equality and inequality, and a void* can be explicitly converted to another type. Other operations would be unsafe because the compiler cannot know what kind of object is really pointed to. Consequently, other operations result in compile-time errors. To use a void*, we must explicitly convert it to a pointer to a specific type. For example:

```
void f(int* pi)
{
      void* pv = pi;  // ok: implicit conversion of int* to void*
      *pv;            // error: can't dereference void*
      ++pv;           // error: can't increment void* (the size of the object pointed to is unknown)

      int* pi2 = static_cast<int*>(pv);          // explicit conversion back to int*

      double* pd1 = pv;                          // error
      double* pd2 = pi;                          // error
      double* pd3 = static_cast<double*>(pv);    // unsafe (§11.5.2)
}
```

In general, it is not safe to use a pointer that has been converted (‘‘cast’’) to a type that differs from the type of the object pointed to. For example, a machine may assume that every double is allocated on an 8-byte boundary. If so, strange behavior could arise if pi pointed to an int that wasn’t allocated that way. This form of explicit type conversion is inherently unsafe and ugly. Consequently, the notation used, static_cast (§11.5.2), was designed to be ugly and easy to find in code.

The primary use for void* is for passing pointers to functions that are not allowed to make assumptions about the type of the object and for returning untyped objects from functions. To use such an object, we must use explicit type conversion.

Functions using void* pointers typically exist at the very lowest level of the system, where real hardware resources are manipulated. For example:

```
void* my_alloc(size_t n);      // allocate n bytes from my special heap
```

Occurrences of void*s at higher levels of the system should be viewed with great suspicion because they are likely indicators of design errors. Where used for optimization, void* can be hidden behind a type-safe interface (§27.3.1).

Pointers to functions (§12.5) and pointers to members (§20.6) cannot be assigned to void*s.

7.2.2 nullptr

The literal nullptr represents the null pointer, that is, a pointer that does not point to an object. It can be assigned to any pointer type, but not to other built-in types:

```
int* pi = nullptr;
double* pd = nullptr;
int i = nullptr;            // error: i is not a pointer
```

There is just one nullptr, which can be used for every pointer type, rather than a null pointer for each pointer type.

Before **nullptr** was introduced, zero (**0**) was used as a notation for the null pointer. For example:

```
int* x = 0; // x gets the value nullptr
```

No object is allocated with the address **0**, and **0** (the all-zeros bit pattern) is the most common representation of **nullptr**. Zero (**0**) is an **int**. However, the standard conversions (§10.5.2.3) allow **0** to be used as a constant of pointer or pointer-to-member type.

It has been popular to define a macro **NULL** to represent the null pointer. For example:

```
int* p = NULL; // using the macro NULL
```

However, there are differences in the definition of **NULL** in different implementations; for example, **NULL** might be **0** or **0L**. In C, **NULL** is typically **(void*)0**, which makes it illegal in C++ (§7.2.1):

```
int* p = NULL; // error: can't assign a void* to an int*
```

Using **nullptr** makes code more readable than alternatives and avoids potential confusion when a function is overloaded to accept either a pointer or an integer (§12.3.1).

7.3 Arrays

For a type **T**, **T[size]** is the type ‘‘array of **size** elements of type **T**.’’ The elements are indexed from **0** to **size–1**. For example:

```
float v[3];      // an array of three floats: v[0], v[1], v[2]
char* a[32];     // an array of 32 pointers to char: a[0] .. a[31]
```

You can access an array using the subscript operator, **[]**, or through a pointer (using operator * or operator **[]**; §7.4). For example:

```
void f()
{
      int aa[10];
      aa[6] = 9;         // assign to aa's 7th element
      int x = aa[99];    // undefined behavior
}
```

Access out of the range of an array is undefined and usually disastrous. In particular, run-time range checking is neither guaranteed nor common.

The number of elements of the array, the array bound, must be a constant expression (§10.4). If you need variable bounds, use a **vector** (§4.4.1, §31.4). For example:

```
void f(int n)
{
      int v1[n];             // error: array size not a constant expression
      vector<int> v2(n);     // OK: vector with n int elements
}
```

Multidimensional arrays are represented as arrays of arrays (§7.4.2).

An array is C++'s fundamental way of representing a sequence of objects in memory. If what you want is a simple fixed-length sequence of objects of a given type in memory, an array is the ideal solution. For every other need, an array has serious problems.

An array can be allocated statically, on the stack, and on the free store (§6.4.2). For example:

```
int a1[10];                       // 10 ints in static storage

void f()
{
        int a2 [20];              // 20 ints on the stack
        int∗p = new int[40];      // 40 ints on the free store
        // ...
}
```

The C++ built-in array is an inherently low-level facility that should primarily be used inside the implementation of higher-level, better-behaved, data structures, such as the standard-library **vector** or **array**. There is no array assignment, and the name of an array implicitly converts to a pointer to its first element at the slightest provocation (§7.4). In particular, avoid arrays in interfaces (e.g., as function arguments; §7.4.3, §12.2.2) because the implicit conversion to pointer is the root cause of many common errors in C code and C-style C++ code. If you allocate an array on the free store, be sure to **delete[]** its pointer once only and only after its last use (§11.2.2). That's most easily and most reliably done by having the lifetime of the free-store array controlled by a resource handle (e.g., **string** (§19.3, §36.3), **vector** (§13.6, §34.2), or **unique_ptr** (§34.3.1)). If you allocate an array statically or on the stack, be sure never to **delete[]** it. Obviously, C programmers cannot follow these pieces of advice because C lacks the ability to encapsulate arrays, but that doesn't make the advice bad in the context of C++.

One of the most widely used kinds of arrays is a zero-terminated array of **char**. That's the way C stores strings, so a zero-terminated array of **char** is often called a *C-style string*. C++ string literals follow that convention (§7.3.2), and some standard-library functions (e.g., **strcpy()** and **strcmp()**; §43.4) rely on it. Often, a **char∗** or a **const char∗** is assumed to point to a zero-terminated sequence of characters.

7.3.1 Array Initializers

An array can be initialized by a list of values. For example:

```
int v1[] = { 1, 2, 3, 4 };
char v2[] = { 'a', 'b', 'c', 0 };
```

When an array is declared without a specific size, but with an initializer list, the size is calculated by counting the elements of the initializer list. Consequently, **v1** and **v2** are of type **int[4]** and **char[4]**, respectively. If a size is explicitly specified, it is an error to give surplus elements in an initializer list. For example:

```
char v3[2] = { 'a', 'b', 0 };     // error: too many initializers
char v4[3] = { 'a', 'b', 0 };     // OK
```

If the initializer supplies too few elements for an array, **0** is used for the rest. For example:

```
int v5[8] = { 1, 2, 3, 4 };
```

is equivalent to

```
int v5[] = { 1, 2, 3, 4 , 0, 0, 0, 0 };
```

There is no built-in copy operation for arrays. You cannot initialize one array with another (not even of exactly the same type), and there is no array assignment:

```
int v6[8] = v5;  // error: can’t copy an array (cannot assign an int* to an array)
v6 = v5;         // error: no array assignment
```

Similarly, you can’t pass arrays by value. See also §7.4.

When you need assignment to a collection of objects, use a **vector** (§4.4.1, §13.6, §34.2), an **array** (§8.2.4), or a **valarray** (§40.5) instead.

An array of characters can be conveniently initialized by a string literal (§7.3.2).

7.3.2 String Literals

A *string literal* is a character sequence enclosed within double quotes:

```
"this is a string"
```

A string literal contains one more character than it appears to have; it is terminated by the null character, **'\0'**, with the value **0**. For example:

```
sizeof("Bohr")==5
```

The type of a string literal is ‘‘array of the appropriate number of **const** characters,’’ so **"Bohr"** is of type **const char[5]**.

In C and in older C++ code, you could assign a string literal to a non-**const char***:

```
void f()
{
      char* p = "Plato";     // error, but accepted in pre-C++11-standard code
      p[4] = 'e';            // error: assignment to const
}
```

It would obviously be unsafe to accept that assignment. It was (and is) a source of subtle errors, so please don’t grumble too much if some old code fails to compile for this reason. Having string literals immutable is not only obvious but also allows implementations to do significant optimizations in the way string literals are stored and accessed.

If we want a string that we are guaranteed to be able to modify, we must place the characters in a non-**const** array:

```
void f()
{
      char p[] = "Zeno";     // p is an array of 5 char
      p[0] = 'R';            // OK
}
```

A string literal is statically allocated so that it is safe to return one from a function. For example:

```
const char* error_message(int i)
{
      // ...
      return "range error";
}
```

The memory holding "range error" will not go away after a call of error_message().

Whether two identical string literals are allocated as one array or as two is implementation-defined (§6.1). For example:

```
const char* p = "Heraclitus";
const char* q = "Heraclitus";

void g()
{
        if (p == q) cout << "one!\n";           // the result is implementation-defined
        // ...
}
```

Note that == compares addresses (pointer values) when applied to pointers, and not the values pointed to.

The empty string is written as a pair of adjacent double quotes, "", and has the type const char[1]. The one character of the empty string is the terminating '\0'.

The backslash convention for representing nongraphic characters (§6.2.3.2) can also be used within a string. This makes it possible to represent the double quote (") and the escape character backslash (\) within a string. The most common such character by far is the newline character, '\n'. For example:

```
cout<<"beep at end of message\a\n";
```

The escape character, '\a', is the ASCII character BEL (also known as *alert*), which causes a sound to be emitted.

It is not possible to have a ‘‘real’’ newline in a (nonraw) string literal:

```
"this is not a string
but a syntax error"
```

Long strings can be broken by whitespace to make the program text neater. For example:

```
char alpha[] = "abcdefghijklmnopqrstuvwxyz"
               "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

The compiler will concatenate adjacent strings, so alpha could equivalently have been initialized by the single string

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

It is possible to have the null character in a string, but most programs will not suspect that there are characters after it. For example, the string "Jens\000Munk" will be treated as "Jens" by standard-library functions such as strcpy() and strlen(); see §43.4.

7.3.2.1 Raw Character Strings

To represent a backslash (\) or a double quote (") in a string literal, we have to precede it with a backslash. That’s logical and in most cases quite simple. However, if we need a lot of backslashes and a lot of quotes in string literals, this simple technique becomes unmanageable. In particular, in regular expressions a backslash is used both as an escape character and to introduce characters representing character classes (§37.1.1). This is a convention shared by many programming languages, so we can't just change it. Therefore, when you write regular expressions for use with the standard **regex** library (Chapter 37), the fact that a backslash is an escape character becomes a notable source of errors. Consider how to write the pattern representing two words separated by a backslash (\):

```
string s = "\\w\\\\w";          // I hope I got that right
```

To prevent the frustration and errors caused by this clash of conventions, C++ provides *raw string literals*. A raw string literal is a string literal where a backslash is just a backslash (and a double quote is just a double quote) so that our example becomes:

```
string s = R"(\w\\w)";          // I'm pretty sure I got that right
```

Raw string literals use the **R"(ccc)"** notation for a sequence of characters **ccc**. The initial **R** is there to distinguish raw string literals from ordinary string literals. The parentheses are there to allow ("unescaped") double quotes. For example:

```
R"("quoted string")"            // the string is "quoted string"
```

So, how do we get the character sequence **)"** into a raw string literal? Fortunately, that's a rare problem, but **"(** and **)"** is only the default delimiter pair. We can add delimiters before the **(** and after the **)** in **"(...)"**. For example:

```
R"***("quoted string containing the usual terminator ("))")***"
     // "quoted string containing the usual terminator ("))"
```

The character sequence after the **)** must be identical to the sequence before the **(**. This way we can cope with (almost) arbitrarily complicated patterns.

Unless you work with regular expressions, raw string literals are probably just a curiosity (and one more thing to learn), but regular expressions are useful and widely used. Consider a real-world example:

```
"('(?:[^\\\\']|\\\\.)*'|\"(?:[^\\\\\"]|\\\\.)*\")|"      // Are the five backslashes correct or not?
```

With examples like that, even experts easily become confused, and raw string literals provide a significant service.

In contrast to nonraw string literals, a raw string literal can contain a newline. For example:

```
string counts {R"(1
22
333)"};
```

is equivalent to

```
string x {"1\n22\n333"};
```

7.3.2.2 Larger Character Sets

A string with the prefix **L**, such as **L"angst"**, is a string of wide characters (§6.2.3). Its type is **const wchar_t[]**. Similarly, a string with the prefix **LR**, such as **LR"(angst)"**, is a raw string (§7.3.2.1) of wide characters of type **const wchar_t[]**. Such a string is terminated by a **L'\0'** character.

There are six kinds of character literals supporting Unicode (*Unicode literals*). This sounds excessive, but there are three major encodings of Unicode: UTF-8, UTF-16, and UTF-32. For each of these three alternatives, both raw and ‘‘ordinary’’ strings are supported. All three UTF encodings support all Unicode characters, so which you use depends on the system you need to fit into. Essentially all Internet applications (e.g., browsers and email) rely on one or more of these encodings.

UTF-8 is a variable-width encoding: common characters fit into 1 byte, less frequently used characters (by some estimate of use) into 2 bytes, and rarer characters into 3 or 4 bytes. In particular, the ASCII characters fit into 1 byte with the same encodings (integer values) in UTF-8 as in ASCII. The various Latin alphabets, Greek, Cyrillic, Hebrew, Arabic, and more fit into 2 bytes.

A UTF-8 string is terminated by `'\0'`, a UTF-16 string by `u'\0'`, and a UTF-32 string by `U'\0'`.

We can represent an ordinary English character string in a variety of ways. Consider a file name using a backslash as the separator:

```
"folder\\file"          // implementation character set string
R"(folder\file)"        // implementation character raw set string
u8"folder\\file"        // UTF-8 string
u8R"(folder\file)"      // UTF-8 raw string
u"folder\\file"         // UTF-16 string
uR"(folder\file)"               // UTF-16 raw string
U"folder\\file"         // UTF-32 string
UR"(folder\file)"               // UTF-32 raw string
```

If printed, these strings will all look the same, but except for the ‘‘plain’’ and UTF-8 strings their internal representations are likely to differ.

Obviously, the real purpose of Unicode strings is to be able to put Unicode characters into them. For example:

```
u8"The official vowels in Danish are: a, e, i, o, u, \u00E6, \u00F8, \u00E5 and y."
```

Printing that string appropriately gives you

```
The official vowels in Danish are: a, e, i, o, u, æ, ø, å and y.
```

The hexadecimal number after the `\u` is a Unicode code point (§iso.2.14.3) [Unicode,1996]. Such a code point is independent of the encoding used and will in fact have different representations (as bits in bytes) in different encodings. For example, `u'0430'` (Cyrillic lowercase letter ‘‘a’’) is the 2-byte hexadecimal value `D0B0` in UTF-8, the 2-byte hexadecimal value `0403` in UTF-16, and the 4-byte hexadecimal value `00000403` in UTF-32. These hexadecimal values are referred to as *universal character names*.

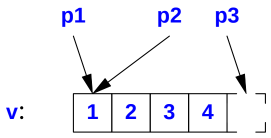
The order of the `u`s and `R`s and their cases are significant: `RU` and `Ur` are not valid string prefixes.

7.4 Pointers into Arrays

In C++, pointers and arrays are closely related. The name of an array can be used as a pointer to its initial element. For example:

```
int v[] = { 1, 2, 3, 4 };
int* p1 = v;            // pointer to initial element (implicit conversion)
int* p2 = &v[0];        // pointer to initial element
int* p3 = v+4;          // pointer to one-beyond-last element
```

or graphically:



Taking a pointer to the element one beyond the end of an array is guaranteed to work. This is important for many algorithms (§4.5, §33.1). However, since such a pointer does not in fact point to an element of the array, it may not be used for reading or writing. The result of taking the address of the element before the initial element or beyond one-past-the-last element is undefined and should be avoided. For example:

```
int* p4 = v−1;  // before the beginning, undefined: don't do it
int* p5 = v+7;  // beyond the end, undefined: don't do it
```

The implicit conversion of an array name to a pointer to the initial element of the array is extensively used in function calls in C-style code. For example:

```
extern "C" int strlen(const char*);         // from <string.h>

void f()
{
        char v[] = "Annemarie";
        char* p = v;       // implicit conversion of char[] to char*
        strlen(p);
        strlen(v);         // implicit conversion of char[] to char*
        v = p;             // error: cannot assign to array
}
```

The same value is passed to the standard-library function `strlen()` in both calls. The snag is that it is impossible to avoid the implicit conversion. In other words, there is no way of declaring a function so that the array `v` is copied when the function is called. Fortunately, there is no implicit or explicit conversion from a pointer to an array.

The implicit conversion of the array argument to a pointer means that the size of the array is lost to the called function. However, the called function must somehow determine the size to perform a meaningful operation. Like other C standard-library functions taking pointers to characters, `strlen()` relies on zero to indicate end-of-string; `strlen(p)` returns the number of characters up to and not including the terminating `0`. This is all pretty low-level. The standard-library `vector` (§4.4.1, §13.6, §31.4), `array` (§8.2.4, §34.2.1), and `string` (§4.2) don't suffer from this problem. These library types give their number of elements as their `size()` without having to count elements each time.

7.4.1 Navigating Arrays

Efficient and elegant access to arrays (and similar data structures) is the key to many algorithms (see §4.5, Chapter 32). Access can be achieved either through a pointer to an array plus an index or through a pointer to an element. For example:

```
void fi(char v[])
{
        for (int i = 0; v[i]!=0; ++i)
                use(v[i]);
}

void fp(char v[])
{
        for (char∗ p = v; ∗p!=0; ++p)
                use(∗p);
}
```

The prefix ∗ operator dereferences a pointer so that ∗p is the character pointed to by p, and ++ increments the pointer so that it refers to the next element of the array.

There is no inherent reason why one version should be faster than the other. With modern compilers, identical code should be (and usually is) generated for both examples. Programmers can choose between the versions on logical and aesthetic grounds.

Subscripting a built-in array is defined in terms of the pointer operations + and ∗. For every built-in array a and integer j within the range of a, we have:

```
a[j] == ∗(&a[0]+j) == ∗(a+j) == ∗(j+a) == j[a]
```

It usually surprises people to find that a[j]==j[a]. For example, 3["Texas"]=="Texas"[3]=='a'. Such cleverness has no place in production code. These equivalences are pretty low-level and do not hold for standard-library containers, such as array and vector.

The result of applying the arithmetic operators +, −, ++, or −− to pointers depends on the type of the object pointed to. When an arithmetic operator is applied to a pointer p of type T∗, p is assumed to point to an element of an array of objects of type T; p+1 points to the next element of that array, and p−1 points to the previous element. This implies that the integer value of p+1 will be sizeof(T) larger than the integer value of p. For example:

```
template<typename T>
int byte_diff(T∗ p, T∗ q)
{
        return reinterpret_cast<char∗>(q)−reinterpret_cast<char∗>(p);
}

void diff_test()
{
        int vi[10];
        short vs[10];
        cout << vi << ' ' << &vi[1] << ' ' << &vi[1]−&vi[0] << ' ' << byte_diff(&vi[0],&vi[1]) << '\n';
        cout << vs << ' ' << &vs[1] << ' ' << &vs[1]−&vs[0] << ' ' << byte_diff(&vs[0],&vs[1]) << '\n';
}
```

This produced:

```
0x7fffaef0 0x7fffaef4 1 4
0x7fffaedc 0x7fffaede 1 2
```

The pointer values were printed using the default hexadecimal notation. This shows that on my implementation, `sizeof(short)` is `2` and `sizeof(int)` is `4`.

Subtraction of pointers is defined only when both pointers point to elements of the same array (although the language has no fast way of ensuring that is the case). When subtracting a pointer `p` from another pointer `q`, `q−p`, the result is the number of array elements in the sequence [`p`:`q`) (an integer). One can add an integer to a pointer or subtract an integer from a pointer; in both cases, the result is a pointer value. If that value does not point to an element of the same array as the original pointer or one beyond, the result of using that value is undefined. For example:

```
void f()
{
        int v1[10];
        int v2[10];

        int i1 = &v1[5]−&v1[3];      // i1 = 2
        int i2 = &v1[5]−&v2[3];      // result undefined

        int∗ p1 = v2+2;              // p1 = &v2[2]
        int∗ p2 = v2−2;              // *p2 undefined
}
```

Complicated pointer arithmetic is usually unnecessary and best avoided. Addition of pointers makes no sense and is not allowed.

Arrays are not self-describing because the number of elements of an array is not guaranteed to be stored with the array. This implies that to traverse an array that does not contain a terminator the way C-style strings do, we must somehow supply the number of elements. For example:

```
void fp(char v[], int size)
{
        for (int i=0; i!=size; ++i)
                use(v[i]);           // hope that v has at least size elements
        for (int x : v)
                use(x);              // error: range-for does not work for pointers

        const int N = 7;
        char v2[N];
        for (int i=0; i!=N; ++i)
                use(v2[i]);
        for (int x : v2)
                use(x);              // range-for works for arrays of known size
}
```

This array concept is inherently low-level. Most advantages of the built-in array and few of the disadvantages can be obtained through the use of the standard-library container **array** (§8.2.4, §34.2.1). Some C++ implementations offer optional range checking for arrays. However, such checking can be quite expensive, so it is often used only as a development aid (rather than being included in production code). If you are not using range checking for individual accesses, try to maintain a consistent policy of accessing elements only in well-defined ranges. That is best done when arrays are manipulated through the interface of a higher-level container type, such as **vector**, where it is harder to get confused about the range of valid elements.

7.4.2 Multidimensional Arrays

Multidimensional arrays are represented as arrays of arrays; a 3-by-5 array is declared like this:

```
int ma[3][5];     // 3 arrays with 5 ints each
```

We can initialize **ma** like this:

```
void init_ma()
{
        for (int i = 0; i!=3; i++)
                for (int j = 0; j!=5; j++)
                        ma[i][j] = 10∗i+j;
}
```

or graphically:

ma:	00	01	02	03	04	10	11	12	13	14	20	21	22	23	24

The array **ma** is simply 15 **int**s that we access as if it were 3 arrays of 5 **int**s. In particular, there is no single object in memory that is the matrix **ma** – only the elements are stored. The dimensions **3** and **5** exist in the compiler source only. When we write code, it is our job to remember them somehow and supply the dimensions where needed. For example, we might print **ma** like this:

```
void print_ma()
{
        for (int i = 0; i!=3; i++) {
                for (int j = 0; j!=5; j++)
                        cout << ma[i][j] << '\t';
                cout << '\n';
        }
}
```

The comma notation used for array bounds in some languages cannot be used in C++ because the comma (**,**) is a sequencing operator (§10.3.2). Fortunately, most mistakes are caught by the compiler. For example:

```
int bad[3,5];             // error: comma not allowed in constant expression
int good[3][5];           // 3 arrays with 5 ints each
int ouch = good[1,4];     // error: int initialized by int* (good[1,4] means good[4], which is an int*)
int nice = good[1][4];
```

7.4.3 Passing Arrays

Arrays cannot directly be passed by value. Instead, an array is passed as a pointer to its first element. For example:

```
void comp(double arg[10])           // arg is a double*
{
        for (int i=0; i!=10; ++i)
                arg[i]+=99;
}

void f()
{
        double a1[10];
        double a2[5];
        double a3[100];

        comp(a1);
        comp(a2);        // disaster!
        comp(a3);        // uses only the first 10 elements
};
```

This code looks sane, but it is not. The code compiles, but the call `comp(a2)` will write beyond the bounds of `a2`. Also, anyone who guessed that the array was passed by value will be disappointed: the writes to `arg[i]` are writes directly to the elements of `comp()`'s argument, rather than to a copy. The function could equivalently have been written as

```
void comp(double* arg)
{
        for (int i=0; i!=10; ++i)
                arg[i]+=99;
}
```

Now the insanity is (hopefully) obvious. When used as a function argument, the first dimension of an array is simply treated as a pointer. Any array bound specified is simply ignored. This implies that if you want to pass a sequence of elements without losing size information, you should not pass a built-in array. Instead, you can place the array inside a class as a member (as is done for `std::array`) or define a class that acts as a handle (as is done for `std::string` and `std::vector`).

If you insist on using arrays directly, you will have to deal with bugs and confusion without getting noticeable advantages in return. Consider defining a function to manipulate a two-dimensional matrix. If the dimensions are known at compile time, there is no problem:

```
void print_m35(int m[3][5])
{
        for (int i = 0; i!=3; i++) {
                for (int j = 0; j!=5; j++)
                        cout << m[i][j] << '\t';
                cout << '\n';
        }
}
```

A matrix represented as a multidimensional array is passed as a pointer (rather than copied; §7.4). The first dimension of an array is irrelevant to finding the location of an element; it simply states how many elements (here, 3) of the appropriate type (here, int[5]) are present. For example, look at the layout of ma above and note that by knowing only that the second dimension is 5, we can locate ma[i][5] for any i. The first dimension can therefore be passed as an argument:

```
void print_mi5(int m[][5], int dim1)
{
        for (int i = 0; i!=dim1; i++) {
                for (int j = 0; j!=5; j++)
                        cout << m[i][j] << '\t';
                cout << '\n';
        }
}
```

When both dimensions need to be passed, the ‘‘obvious solution’’ does not work:

```
void print_mij(int m[][], int dim1, int dim2)      // doesn't behave as most people would think
{
        for (int i = 0; i!=dim1; i++) {
                for (int j = 0; j!=dim2; j++)
                        cout << m[i][j] << '\t';       // surprise!
                cout << '\n';
        }
}
```

Fortunately, the argument declaration m[][] is illegal because the second dimension of a multidimensional array must be known in order to find the location of an element. However, the expression m[i][j] is (correctly) interpreted as *(*(m+i)+j), although that is unlikely to be what the programmer intended. A correct solution is:

```
void print_mij(int* m, int dim1, int dim2)
{
        for (int i = 0; i!=dim1; i++) {
                for (int j = 0; j!=dim2; j++)
                        cout << m[i*dim2+j] << '\t'; // obscure
                cout << '\n';
        }
}
```

The expression used for accessing the members in print_mij() is equivalent to the one the compiler generates when it knows the last dimension.

To call this function, we pass a matrix as an ordinary pointer:

```
int test()
{
        int v[3][5] = {
                {0,1,2,3,4}, {10,11,12,13,14}, {20,21,22,23,24}
        };
```