C++ Primer
Fifth Edition

This page intentionally left blank

C++ Primer
Fifth Edition

Stanley B. Lippman
Josée Lajoie
Barbara E. Moo

Addison-Wesley

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

Library of Congress Cataloging-in-Publication Data

Lippman, Stanley B.
C++ primer / Stanley B. Lippman, Josée Lajoie, Barbara E. Moo. – 5th ed.
p. cm.
Includes index.
ISBN 0-321-71411-3 (pbk. : alk. paper) 1. C++ (Computer program language) I. Lajoie, Josée. II. Moo, Barbara E. III. Title.

QA76.73.C153L57697 2013
005.13'3–dc23 2012020184

C++ Primer, Fifth Edition, features an enhanced, layflat binding, which allows the book to stay open more easily when placed on a flat surface. This special binding method—notable by a small space inside the spine—also increases durability.

ISBN-13: 978-0-321-71411-4
ISBN-10: 0-321-71411-3
Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
Third printing, February 2013

To Beth,
who makes this,
and all things,
possible.

———

To Daniel and Anna,
who contain
virtually
all possibilities.
—SBL

To Mark and Mom,
for their
unconditional love and support.
—JL

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

This page intentionally left blank

Contents

This page intentionally left blank

New Features in C++11

This page intentionally left blank

Preface

Countless programmers have learned C++ from previous editions of *C++ Primer*. During that time, C++ has matured greatly: Its focus, and that of its programming community, has widened from looking mostly at *machine* efficiency to devoting more attention to *programmer* efficiency.

In 2011, the C++ standards committee issued a major revision to the ISO C++ standard. This revised standard is latest step in C++'s evolution and continues the emphasis on programmer efficiency. The primary goals of the new standard are to

- Make the language more uniform and easier to teach and to learn
- Make the standard libraries easier, safer, and more efficient to use
- Make it easier to write efficient abstractions and libraries

In this edition, we have completely revised the *C++ Primer* to use the latest standard. You can get an idea of how extensively the new standard has affected C++ by reviewing the New Features Table of Contents, which lists the sections that cover new material and appears on page xxi.

Some additions in the new standard, such as `auto` for type inference, are pervasive. These facilities make the code in this edition easier to read and to understand. Programs (and programmers!) can ignore type details, which makes it easier to concentrate on what the program is intended to do. Other new features, such as smart pointers and move-enabled containers, let us write more sophisticated classes without having to contend with the intricacies of resource management. As a result, we can start to teach how to write your own classes much earlier in the book than we did in the Fourth Edition. We—and you—no longer have to worry about many of the details that stood in our way under the previous standard.

We've marked those parts of the text that cover features defined by the new standard, with a marginal icon. We hope that readers who are already familiar with the core of C++ will find these alerts useful in deciding where to focus their attention. We also expect that these icons will help explain error messages from compilers that might not yet support every new feature. Although nearly all of the examples in this book have been compiled under the current release of the GNU compiler, we realize some readers will not yet have access to completely updated compilers. Even though numerous capabilities have been added by the latest standard, the core language remains unchanged and forms the bulk of the material that we cover. Readers can use these icons to note which capabilities may not yet be available in their compiler.

Why Read This Book?

Modern C++ can be thought of as comprising three parts:

- The low-level language, much of which is inherited from C
- More advanced language features that allow us to define our own types and to organize large-scale programs and systems
- The standard library, which uses these advanced features to provide useful data structures and algorithms

Most texts present C++ in the order in which it evolved. They teach the C subset of C++ first, and present the more abstract features of C++ as advanced topics at the end of the book. There are two problems with this approach: Readers can get bogged down in the details inherent in low-level programming and give up in frustration. Those who do press on learn bad habits that they must unlearn later.

We take the opposite approach: Right from the start, we use the features that let programmers ignore the details inherent in low-level programming. For example, we introduce and use the library `string` and `vector` types along with the built-in arithmetic and array types. Programs that use these library types are easier to write, easier to understand, and much less error-prone.

Too often, the library is taught as an "advanced" topic. Instead of using the library, many books use low-level programming techniques based on pointers to character arrays and dynamic memory management. Getting programs that use these low-level techniques to work correctly is much harder than writing the corresponding C++ code using the library.

Throughout *C++ Primer*, we emphasize good style: We want to help you, the reader, develop good habits immediately and avoid needing to unlearn bad habits as you gain more sophisticated knowledge. We highlight particularly tricky matters and warn about common misconceptions and pitfalls.

We also explain the rationale behind the rules—explaining the why not just the what. We believe that by understanding why things work as they do, readers can more quickly cement their grasp of the language.

Although you do not need to know C in order to understand this book, we assume you know enough about programming to write, compile, and run a program in at least one modern block-structured language. In particular, we assume you have used variables, written and called functions, and used a compiler.

Changes to the Fifth Edition

New to this edition of *C++ Primer* are icons in the margins to help guide the reader. C++ is a large language that offers capabilities tailored to particular kinds of programming problems. Some of these capabilities are of great import for large project teams but might not be necessary for smaller efforts. As a result, not every programmer needs to know every detail of every feature. We've added these marginal icons to help the reader know which parts can be learned later and which topics are more essential.

We've marked sections that cover the fundamentals of the language with an image of a person studying a book. The topics covered in sections marked this

way form the core part of the language. Everyone should read and understand these sections.

We've also indicated those sections that cover advanced or special-purpose topics. These sections can be skipped or skimmed on a first reading. We've marked such sections with a stack of books to indicate that you can safely put down the book at that point. It is probably a good idea to skim such sections so you know that the capability exists. However, there is no reason to spend time studying these topics until you actually need to use the feature in your own programs.

To help readers guide their attention further, we've noted particularly tricky concepts with a magnifying-glass icon. We hope that readers will take the time to understand thoroughly the material presented in the sections so marked. In at least some of these sections, the import of the topic may not be readily apparent; but we think you'll find that these sections cover topics that turn out to be essential to understanding the language.

Another aid to reading this book, is our extensive use of cross-references. We hope these references will make it easier for readers to dip into the middle of the book, yet easily jump back to the earlier material on which later examples rely.

What remains unchanged is that *C++ Primer* is a clear, correct, and thorough tutorial guide to C++. We teach the language by presenting a series of increasingly sophisticated examples, which explain language features and show how to make the best use of C++.

Structure of This Book

We start by covering the basics of the language and the library together in Parts I and II. These parts cover enough material to let you, the reader, write significant programs. Most C++ programmers need to know essentially everything covered in this portion of the book.

In addition to teaching the basics of C++, the material in Parts I and II serves another important purpose: By using the abstract facilities defined by the library, you will become more comfortable with using high-level programming techniques. The library facilities are themselves abstract data types that are usually written in C++. The library can be defined using the same class-construction features that are available to any C++ programmer. Our experience in teaching C++ is that by first using well-designed abstract types, readers find it easier to understand how to build their own types.

Only after a thorough grounding in using the library—and writing the kinds of abstract programs that the library allows—do we move on to those C++ features that will enable you to write your own abstractions. Parts III and IV focus on writing abstractions in the form of classes. Part III covers the fundamentals; Part IV covers more specialized facilities.

In Part III, we cover issues of copy control, along with other techniques to make classes that are as easy to use as the built-in types. Classes are the foundation for object-oriented and generic programming, which we also cover in Part III. *C++ Primer* concludes with Part IV, which covers features that are of most use in structuring large, complicated systems. We also summarize the library algorithms in Appendix A.

Aids to the Reader

Each chapter concludes with a summary, followed by a glossary of defined terms, which together recap the chapter's most important points. Readers should use these sections as a personal checklist: If you do not understand a term, restudy the corresponding part of the chapter.

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

- Important terms are indicated in **bold**; important terms that we assume are already familiar to the reader are indicated in ***bold italics***. Each term appears in the chapter's Defined Terms section.
- Throughout the book, we highlight parts of the text to call attention to important aspects of the language, warn about common pitfalls, suggest good programming practices, and provide general usage tips.
- To make it easier to follow the relationships among features and concepts, we provide extensive forward and backward cross-references.
- We provide sidebar discussions on important concepts and for topics that new C++ programmers often find most difficult.
- Learning any programming language requires writing programs. To that end, the Primer provides extensive examples throughout the text. Source code for the extended examples is available on the Web at the following URL:

```
http://www.informit.com/title/0321714113
```

A Note about Compilers

As of this writing (July, 2012), compiler vendors are hard at work updating their compilers to match the latest ISO standard. The compiler we use most frequently is the GNU compiler, version 4.7.0. There are only a few features used in this book that this compiler does not yet implement: inheriting constructors, reference qualifiers for member functions, and the regular-expression library.

Acknowledgments

In preparing this edition we are very grateful for the help of several current and former members of the standardization committee: Dave Abrahams, Andy Koenig, Stephan T. Lavavej, Jason Merrill, John Spicer, and Herb Sutter. They provided invaluable assistance to us in understanding some of the more subtle parts of the new standard. We'd also like to thank the many folks who worked on updating the GNU compiler making the standard a reality.

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

We extend our deep-felt thanks to our reviewers, whose helpful comments led us to make improvements great and small throughout the book: Marshall Clow, Jon Kalb, Nevin Liber, Dr. C. L. Tondo, Daveed Vandevoorde, and Steve Vinoski.

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

Finally, we thank the fine folks at Addison-Wesley who have shepherded this edition through the publishing process: Peter Gordon, our editor, who provided the impetus for us to revise *C++ Primer* once again; Kim Boedigheimer, who keeps us all on schedule; Barbara Wood, who found lots of editing errors for us during the copy-edit phase, and Elizabeth Ryan, who was again a delight to work with as she guided us through the design and production process.

This page intentionally left blank

CHAPTER 1

GETTING STARTED

CONTENTS

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

After having read this chapter and worked through the exercises, you should be able to write, compile, and execute simple programs. Later chapters will assume that you can use the features introduced in this chapter, and will explain these features in more detail.

The way to learn a new programming language is to write programs. In this chapter, we'll write a program to solve a simple problem for a bookstore.

Our store keeps a file of transactions, each of which records the sale of one or more copies of a single book. Each transaction contains three data elements:

```
0-201-70353-X 4 24.99
```

The first element is an ISBN (International Standard Book Number, a unique book identifier), the second is the number of copies sold, and the last is the price at which each of these copies was sold. From time to time, the bookstore owner reads this file and for each book computes the number of copies sold, the total revenue from that book, and the average sales price.

To be able to write this program, we need to cover a few basic C++ features. In addition, we'll need to know how to compile and execute a program.

Although we haven't yet designed our program, it's easy to see that it must

- Define variables
- Do input and output
- Use a data structure to hold the data
- Test whether two records have the same ISBN
- Contain a loop that will process every record in the transaction file

We'll start by reviewing how to solve these subproblems in C++ and then write our bookstore program.

1.1 Writing a Simple C++ Program

Every C++ program contains one or more ***functions***, one of which must be named **`main`**. The operating system runs a C++ program by calling `main`. Here is a simple version of `main` that does nothing but return a value to the operating system:

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

A function definition has four elements: a ***return type***, a ***function name***, a (possibly empty) ***parameter list*** enclosed in parentheses, and a ***function body***. Although `main` is special in some ways, we define `main` the same way we define any other function.

In this example, `main` has an empty list of parameters (shown by the `()` with nothing inside). § 6.2.5 (p. 218) will discuss the other parameter types that we can define for `main`.

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

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

```
{
    return 0;
}
```

The only statement in this block is a `return`, which is a statement that terminates a function. As is the case here, a `return` can also send a value back to the function's caller. When a `return` statement includes a value, the value returned must have a type that is compatible with the return type of the function. In this case, the return type of `main` is `int` and the return value is `0`, which is an `int`.

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

On most systems, the value returned from `main` is a status indicator. A return value of `0` indicates success. A nonzero return has a meaning that is defined by the system. Ordinarily a nonzero return indicates what kind of error occurred.

KEY CONCEPT: TYPES

Types are one of the most fundamental concepts in programming and a concept that we will come back to over and over in this Primer. A type defines both the contents of a data element and the operations that are possible on those data.

The data our programs manipulate are stored in variables and every variable has a type. When the type of a variable named `v` is `T`, we often say that "`v` has type `T`" or, interchangeably, that "`v` is a `T`."

1.1.1 Compiling and Executing Our Program

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

Many PC-based compilers are run from an integrated development environment (IDE) that bundles the compiler with build and analysis tools. These environments can be a great asset in developing large programs but require a fair bit of time to learn how to use effectively. Learning how to use such environments is well beyond the scope of this book.

Most compilers, including those that come with an IDE, provide a command-line interface. Unless you already know the IDE, you may find it easier to start with the command-line interface. Doing so will let you concentrate on learning C++ first. Moreover, once you understand the language, the IDE is likely to be easier to learn.

Program Source File Naming Convention

Whether you use a command-line interface or an IDE, most compilers expect program source code to be stored in one or more files. Program files are normally

referred to as a ***source files***. On most systems, the name of a source file ends with a suffix, which is a period followed by one or more characters. The suffix tells the system that the file is a C++ program. Different compilers use different suffix conventions; the most common include `.cc`, `.cxx`, `.cpp`, `.cp`, and `.C`.

Running the Compiler from the Command Line

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

```
$ CC prog1.cc
```

where `CC` names the compiler and `$` is the system prompt. The compiler generates an executable file. On a Windows system, that executable file is named `prog1.exe`. UNIX compilers tend to put their executables in files named `a.out`.

To run an executable on Windows, we supply the executable file name and can omit the `.exe` file extension:

```
$ prog1
```

On some systems you must specify the file's location explicitly, even if the file is in the current directory or folder. In such cases, we would write

```
$ .\prog1
```

The "`.`" followed by a backslash indicates that the file is in the current directory.

To run an executable on UNIX, we use the full file name, including the file extension:

```
$ a.out
```

If we need to specify the file's location, we'd use a "`.`" followed by a forward slash to indicate that our executable is in the current directory:

```
$ ./a.out
```

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

On UNIX systems, we obtain the status by writing

```
$ echo $?
```

To see the status on a Windows system, we write

```
$ echo %ERRORLEVEL%
```

Running the GNU or Microsoft Compilers

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

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

Here `$` is the system prompt. The `-o prog1` is an argument to the compiler and names the file in which to put the executable file. This command generates an executable file named `prog1` or `prog1.exe`, depending on the operating system. On UNIX, executable files have no suffix; on Windows, the suffix is `.exe`. If the `-o prog1` is omitted, the compiler generates an executable named `a.out` on UNIX systems and `a.exe` on Windows. (Note: Depending on the release of the GNU compiler you are using, you may need to specify `-std=c++0x` to turn on C++ 11 support.)

The command to run the Microsoft Visual Studio 2010 compiler is `cl`:

```
C:\Users\me\Programs> cl /EHsc prog1.cpp
```

Here `C:\Users\me\Programs>` is the system prompt and `\Users\me\Programs` is the name of the current directory (aka the current folder). The `cl` command invokes the compiler, and `/EHsc` is the compiler option that turns on standard exception handling. The Microsoft compiler automatically generates an executable with a name that corresponds to the first source file name. The executable has the suffix `.exe` and the same name as the source file name. In this case, the executable is named `prog1.exe`.

Compilers usually include options to generate warnings about problematic constructs. It is usually a good idea to use these options. Our preference is to use `-Wall` with the GNU compiler, and to use `/W4` with the Microsoft compilers.

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

Exercises Section 1.1.1

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

Exercise 1.2: Change the program to return `-1`. A return value of `-1` is often treated as an indicator that the program failed. Recompile and rerun your program to see how your system treats a failure indicator from `main`.

1.2 A First Look at Input/Output

The C++ language does not define any statements to do input or output (IO). Instead, C++ includes an extensive **standard library** that provides IO (and many other facilities). For many purposes, including the examples in this book, one needs to know only a few basic concepts and operations from the IO library.

Most of the examples in this book use the **`iostream`** library. Fundamental to the `iostream` library are two types named **`istream`** and **`ostream`**, which represent input and output streams, respectively. A stream is a sequence of characters read from or written to an IO device. The term *stream* is intended to suggest that the characters are generated, or consumed, sequentially over time.

Standard Input and Output Objects

The library defines four IO objects. To handle input, we use an object of type `istream` named **`cin`** (pronounced *see-in*). This object is also referred to as the **standard input**. For output, we use an `ostream` object named **`cout`** (pronounced *see-out*). This object is also known as the **standard output**. The library also defines two other `ostream` objects, named **`cerr`** and **`clog`** (pronounced *see-err* and *see-log*, respectively). We typically use `cerr`, referred to as the **standard error**, for warning and error messages and `clog` for general information about the execution of the program.

Ordinarily, the system associates each of these objects with the window in which the program is executed. So, when we read from `cin`, data are read from the window in which the program is executing, and when we write to `cout`, `cerr`, or `clog`, the output is written to the same window.

A Program That Uses the IO Library

In our bookstore problem, we'll have several records that we'll want to combine into a single total. As a simpler, related problem, let's look first at how we might add two numbers. Using the IO library, we can extend our `main` program to prompt the user to give us two numbers and then print their sum:

```
#include <iostream>
int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1 = 0, v2 = 0;
    std::cin >> v1 >> v2;
    std::cout << "The sum of " << v1 << " and " << v2
              << " is " << v1 + v2 << std::endl;
    return 0;
}
```

This program starts by printing

```
Enter two numbers:
```

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

```
3 7
```

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

```
The sum of 3 and 7 is 10
```

The first line of our program

```
#include <iostream>
```

tells the compiler that we want to use the `iostream` library. The name inside angle brackets (`iostream` in this case) refers to a **header**. Every program that uses a library facility must include its associated header. The `#include` directive

must be written on a single line—the name of the header and the `#include` must appear on the same line. In general, `#include` directives must appear outside any function. Typically, we put all the `#include` directives for a program at the beginning of the source file.

Writing to a Stream

The first statement in the body of `main` executes an **expression**. In C++ an expression yields a result and is composed of one or more operands and (usually) an operator. The expressions in this statement use the output operator (the **« operator**) to print a message on the standard output:

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

The `<<` operator takes two operands: The left-hand operand must be an `ostream` object; the right-hand operand is a value to print. The operator writes the given value on the given `ostream`. The result of the output operator is its left-hand operand. That is, the result is the `ostream` on which we wrote the given value.

Our output statement uses the `<<` operator twice. Because the operator returns its left-hand operand, the result of the first operator becomes the left-hand operand of the second. As a result, we can chain together output requests. Thus, our expression is equivalent to

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

Each operator in the chain has the same object as its left-hand operand, in this case `std::cout`. Alternatively, we can generate the same output using two statements:

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

The first output operator prints a message to the user. That message is a **string literal**, which is a sequence of characters enclosed in double quotation marks. The text between the quotation marks is printed to the standard output.

The second operator prints `endl`, which is a special value called a **manipulator**. Writing `endl` has the effect of ending the current line and flushing the ***buffer*** associated with that device. Flushing the buffer ensures that all the output the program has generated so far is actually written to the output stream, rather than sitting in memory waiting to be written.

Programmers often add print statements during debugging. Such statements should *always* flush the stream. Otherwise, if the program crashes, output may be left in the buffer, leading to incorrect inferences about where the program crashed.

Using Names from the Standard Library

Careful readers will note that this program uses `std::cout` and `std::endl` rather than just `cout` and `endl`. The prefix `std::` indicates that the names `cout` and `endl` are defined inside the **namespace** named **`std`**. Namespaces allow us to

avoid inadvertent collisions between the names we define and uses of those same names inside a library. All the names defined by the standard library are in the `std` namespace.

One side effect of the library's use of a namespace is that when we use a name from the library, we must say explicitly that we want to use the name from the `std` namespace. Writing `std::cout` uses the scope operator (the **`::` operator**) to say that we want to use the name `cout` that is defined in the namespace `std`. § 3.1 (p. 82) will show a simpler way to access names from the library.

Reading from a Stream

Having asked the user for input, we next want to read that input. We start by defining two ***variables*** named `v1` and `v2` to hold the input:

```
int v1 = 0, v2 = 0;
```

We define these variables as type `int`, which is a built-in type representing integers. We also ***initialize*** them to `0`. When we initialize a variable, we give it the indicated value at the same time as the variable is created.

The next statement

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

reads the input. The input operator (the **» operator**) behaves analogously to the output operator. It takes an `istream` as its left-hand operand and an object as its right-hand operand. It reads data from the given `istream` and stores what was read in the given object. Like the output operator, the input operator returns its left-hand operand as its result. Hence, this expression is equivalent to

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

Because the operator returns its left-hand operand, we can combine a sequence of input requests into a single statement. Our input operation reads two values from `std::cin`, storing the first in `v1` and the second in `v2`. In other words, our input operation executes as

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

Completing the Program

What remains is to print our result:

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

This statement, although longer than the one that prompted the user for input, is conceptually similar. It prints each of its operands on the standard output. What is interesting in this example is that the operands are not all the same kinds of values. Some operands are string literals, such as `"The sum of "`. Others are `int` values, such as `v1`, `v2`, and the result of evaluating the arithmetic expression `v1 + v2`. The library defines versions of the input and output operators that handle operands of each of these differing types.

EXERCISES SECTION 1.2

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

Exercise 1.4: Our program used the addition operator, `+`, to add two numbers. Write a program that uses the multiplication operator, `*`, to print the product instead.

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

Exercise 1.6: Explain whether the following program fragment is legal.

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

If the program is legal, what does it do? If the program is not legal, why not? How would you fix it?

1.3 A Word about Comments

Before our programs get much more complicated, we should see how C++ handles *comments*. Comments help the human readers of our programs. They are typically used to summarize an algorithm, identify the purpose of a variable, or clarify an otherwise obscure segment of code. The compiler ignores comments, so they have no effect on the program's behavior or performance.

Although the compiler ignores comments, readers of our code do not. Programmers tend to believe comments even when other parts of the system documentation are out of date. An incorrect comment is worse than no comment at all because it may mislead the reader. When you change your code, be sure to update the comments, too!

Kinds of Comments in C++

There are two kinds of comments in C++: single-line and paired. A single-line comment starts with a double slash (`//`) and ends with a newline. Everything to the right of the slashes on the current line is ignored by the compiler. A comment of this kind can contain any text, including additional double slashes.

The other kind of comment uses two delimiters (`/*` and `*/`) that are inherited from C. Such comments begin with a `/*` and end with the next `*/`. These comments can include anything that is not a `*/`, including newlines. The compiler treats everything that falls between the `/*` and `*/` as part of the comment.

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

Programs typically contain a mixture of both comment forms. Comment pairs

generally are used for multiline explanations, whereas double-slash comments tend to be used for half-line and single-line remarks:

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

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

Comment Pairs Do Not Nest

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

```
/*
 * comment pairs /*   */ cannot nest.
 * ''cannot nest'' is considered source code,
 * as is the rest of the program
 */
int main()
{
    return 0;
}
```

We often need to comment out a block of code during debugging. Because that code might contain nested comment pairs, the best way to comment a block of code is to insert single-line comments at the beginning of each line in the section we want to ignore:

```
// /*
// * everything inside a single-line comment is ignored
// * including nested comment pairs
//  */
```

EXERCISES SECTION 1.3

Exercise 1.7: Compile a program that has incorrectly nested comments.

Exercise 1.8: Indicate which, if any, of the following output statements are legal:

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

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

1.4 Flow of Control

Statements normally execute sequentially: The first statement in a block is executed first, followed by the second, and so on. Of course, few programs—including the one to solve our bookstore problem—can be written using only sequential execution. Instead, programming languages provide various flow-of-control statements that allow for more complicated execution paths.

1.4.1 The `while` Statement

A **`while` statement** repeatedly executes a section of code so long as a given condition is true. We can use a `while` to write a program to sum the numbers from 1 through 10 inclusive as follows:

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

When we compile and execute this program, it prints

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

As before, we start by including the `iostream` header and defining `main`. Inside `main` we define two `int` variables: `sum`, which will hold our summation, and `val`, which will represent each of the values from 1 through 10. We give `sum` an initial value of `0` and start `val` off with the value `1`.

The new part of this program is the `while` statement. A `while` has the form

```
while (condition)
    statement
```

A `while` executes by (alternately) testing the *condition* and executing the associated *statement* until the *condition* is false. A **condition** is an expression that yields a result that is either true or false. So long as *condition* is true, *statement* is executed. After executing *statement*, *condition* is tested again. If *condition* is again true, then *statement* is again executed. The `while` continues, alternately testing the *condition* and executing *statement* until the *condition* is false.

In this program, the `while` statement is

```
// keep executing the while as long as val is less than or equal to 10
while (val <= 10) {
    sum += val;  // assigns sum + val to sum
    ++val;       // add 1 to val
}
```

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

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

A block is a sequence of zero or more statements enclosed by curly braces. A block is a statement and may be used wherever a statement is required. The first statement in this block uses the compound assignment operator (the **+= operator**). This operator adds its right-hand operand to its left-hand operand and stores the result in the left-hand operand. It has essentially the same effect as writing an addition and an **assignment**:

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

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

The next statement

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

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

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

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

EXERCISES SECTION 1.4.1

Exercise 1.9: Write a program that uses a `while` to sum the numbers from 50 to 100.

Exercise 1.10: In addition to the `++` operator that adds `1` to its operand, there is a decrement operator (`--`) that subtracts `1`. Use the decrement operator to write a `while` that prints the numbers from ten down to zero.

Exercise 1.11: Write a program that prompts the user for two integers. Print each number in the range specified by those two integers.

1.4.2 The `for` Statement

In our `while` loop we used the variable `val` to control how many times we executed the loop. We tested the value of `val` in the condition and incremented `val` in the `while` body.

This pattern—using a variable in a condition and incrementing that variable in the body—happens so often that the language defines a second statement, the **`for` statement**, that abbreviates code that follows this pattern. We can rewrite this program using a `for` loop to sum the numbers from 1 through 10 as follows:

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

As before, we define `sum` and initialize it to zero. In this version, we define `val` as part of the `for` statement itself:

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

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

```
int val = 1;
```

defines an `int` object named `val` and gives it an initial value of `1`. The variable `val` exists only inside the `for`; it is not possible to use `val` after this loop terminates. The *init-statement* is executed only once, on entry to the `for`. The *condition*

```
val <= 10
```

compares the current value in `val` to `10`. The *condition* is tested each time through the loop. As long as `val` is less than or equal to `10`, we execute the `for` body. The *expression* is executed after the `for` body. Here, the *expression*

```
++val
```

uses the prefix increment operator, which adds `1` to the value of `val`. After executing the *expression*, the `for` retests the *condition*. If the new value of `val` is still less than or equal to `10`, then the `for` loop body is executed again. After executing the body, `val` is incremented again. The loop continues until the *condition* fails.

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

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

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

1. Create `val` and initialize it to `1`.
2. Test whether `val` is less than or equal to `10`. If the test succeeds, execute the `for` body. If the test fails, exit the loop and continue execution with the first statement following the `for` body.
3. Increment `val`.
4. Repeat the test in step 2, continuing with the remaining steps as long as the condition is true.

EXERCISES SECTION 1.4.2

Exercise 1.12: What does the following `for` loop do? What is the final value of `sum`?

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

Exercise 1.13: Rewrite the first two exercises from § 1.4.1 (p. 13) using `for` loops.

Exercise 1.14: Compare and contrast the loops that used a `for` with those using a `while`. Are there advantages or disadvantages to using either form?

Exercise 1.15: Write programs that contain the common errors discussed in the box on page 16. Familiarize yourself with the messages the compiler generates.

1.4.3 Reading an Unknown Number of Inputs

In the preceding sections, we wrote programs that summed the numbers from 1 through 10. A logical extension of this program would be to ask the user to input a set of numbers to sum. In this case, we won't know how many numbers to add. Instead, we'll keep reading numbers until there are no more numbers to read:

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

If we give this program the input

```
3 4 5 6
```

then our output will be

```
Sum is: 18
```

The first line inside `main` defines two `int` variables, named `sum` and `value`, which we initialize to `0`. We'll use `value` to hold each number as we read it from the input. We read the data inside the condition of the `while`:

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

Evaluating the `while` condition executes the expression

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

That expression reads the next number from the standard input and stores that number in `value`. The input operator (§ 1.2, p. 8) returns its left operand, which in this case is `std::cin`. This condition, therefore, tests `std::cin`.

When we use an `istream` as a condition, the effect is to test the state of the stream. If the stream is valid—that is, if the stream hasn't encountered an error—then the test succeeds. An `istream` becomes invalid when we hit ***end-of-file*** or encounter an invalid input, such as reading a value that is not an integer. An `istream` that is in an invalid state will cause the condition to yield false.

Thus, our `while` executes until we encounter end-of-file (or an input error). The `while` body uses the compound assignment operator to add the current value to the evolving `sum`. Once the condition fails, the `while` ends. We fall through and execute the next statement, which prints the `sum` followed by `endl`.

Entering an End-of-File from the Keyboard

When we enter input to a program from the keyboard, different operating systems use different conventions to allow us to indicate end-of-file. On Windows systems we enter an end-of-file by typing a control-z—hold down the Ctrl key and press z—followed by hitting either the Enter or Return key. On UNIX systems, including on Mac OS X machines, end-of-file is usually control-d.

Compilation Revisited

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

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

```
// error: missing ) in parameter list for main
int main ( {
    // error: used colon, not a semicolon, after endl
    std::cout << "Read each file." << std::endl:
    // error: missing quotes around string literal
    std::cout << Update master. << std::endl;
    // error: second output operator is missing
    std::cout << "Write new master." std::endl;
    // error: missing ; on return statement
    return 0
}
```

Type errors: **Each item of data in C++ has an associated type. The value 10, for example, has a type of `int` (or, more colloquially, "is an `int`"). The word `"hello"`, including the double quotation marks, is a string literal. One example of a type error is passing a string literal to a function that expects an `int` argument.**

Declaration errors: **Every name used in a C++ program must be declared before it is used. Failure to declare a name usually results in an error message. The two most common declaration errors are forgetting to use `std::` for a name from the library and misspelling the name of an identifier:**

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

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

EXERCISES SECTION 1.4.3

Exercise 1.16: Write your own version of a program that prints the sum of a set of integers read from `cin`.

1.4.4 The `if` Statement

Like most languages, C++ provides an **`if` statement** that supports conditional execution. We can use an `if` to write a program to count how many consecutive times each distinct value appears in the input:

```
#include <iostream>
int main()
{
    // currVal is the number we're counting; we'll read new values into val
    int currVal = 0, val = 0;
    // read first number and ensure that we have data to process
    if (std::cin >> currVal) {
        int cnt = 1;  // store the count for the current value we're processing
        while (std::cin >> val) { // read the remaining numbers
            if (val == currVal)    // if the values are the same
                ++cnt;             // add 1 to cnt
            else { // otherwise, print the count for the previous value
                std::cout << currVal << " occurs "
                          << cnt << " times" << std::endl;
                currVal = val;     // remember the new value
                cnt = 1;           // reset the counter
            }
        }  // while loop ends here
        // remember to print the count for the last value in the file
        std::cout << currVal << " occurs "
                  << cnt << " times" << std::endl;
    } // outermost if statement ends here
    return 0;
}
```

If we give this program the following input:

```
42 42 42 42 42 55 55 62 100 100 100
```

then the output should be

```
42 occurs 5 times
55 occurs 2 times
62 occurs 1 times
100 occurs 3 times
```

Much of the code in this program should be familiar from our earlier programs. We start by defining `val` and `currVal`: `currVal` will keep track of which number we are counting; `val` will hold each number as we read it from the input. What's new are the two `if` statements. The first `if`

```
if (std::cin >> currVal) {
    // ...
} // outermost if statement ends here
```

ensures that the input is not empty. Like a `while`, an `if` evaluates a condition. The condition in the first `if` reads a value into `currVal`. If the read succeeds, then the condition is true and we execute the block that starts with the open curly following the condition. That block ends with the close curly just before the `return` statement.

Once we know there are numbers to count, we define `cnt`, which will count how often each distinct number occurs. We use a `while` loop similar to the one in the previous section to (repeatedly) read numbers from the standard input.

The body of the `while` is a block that contains the second `if` statement:

```
if (val == currVal)    // if the values are the same
    ++cnt;             // add 1 to cnt
else { // otherwise, print the count for the previous value
    std::cout << currVal << " occurs "
              << cnt << " times" << std::endl;
    currVal = val;     // remember the new value
    cnt = 1;           // reset the counter
}
```

The condition in this `if` uses the equality operator (the **== operator**) to test whether `val` is equal to `currVal`. If so, we execute the statement that immediately follows the condition. That statement increments `cnt`, indicating that we have seen `currVal` once more.

If the condition is false—that is, if `val` is not equal to `currVal`—then we execute the statement following the `else`. This statement is a block consisting of an output statement and two assignments. The output statement prints the count for the value we just finished processing. The assignments reset `cnt` to `1` and `currVal` to `val`, which is the number we just read.

C++ uses = for assignment and == for equality. Both operators can appear inside a condition. It is a common mistake to write = when you mean == inside a condition.

Exercises Section 1.4.4

Exercise 1.17: What happens in the program presented in this section if the input values are all equal? What if there are no duplicated values?

Exercise 1.18: Compile and run the program from this section giving it only equal values as input. Run it again giving it values in which no number is repeated.

Exercise 1.19: Revise the program you wrote for the exercises in § 1.4.1 (p. 13) that printed a range of numbers so that it handles input in which the first number is smaller than the second.

KEY CONCEPT: INDENTATION AND FORMATTING OF C++ PROGRAMS

C++ programs are largely free-format, meaning that where we put curly braces, indentation, comments, and newlines usually has no effect on what our programs mean. For example, the curly brace that denotes the beginning of the body of `main` could be on the same line as `main`; positioned as we have done, at the beginning of the next line; or placed anywhere else we'd like. The only requirement is that the open curly must be the first nonblank, noncomment character following `main`'s parameter list.

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

Endless debates occur as to the right way to format C or C++ programs. Our belief is that there is no single correct style but that there is value in consistency. Most programmers indent subsidiary parts of their programs, as we've done with the statements inside `main` and the bodies of our loops. We tend to put the curly braces that delimit functions on their own lines. We also indent compound IO expressions so that the operators line up. Other indentation conventions will become clear as our programs become more sophisticated.

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

1.5 Introducing Classes

The only remaining feature we need to understand before solving our bookstore problem is how to define a ***data structure*** to represent our transaction data. In C++ we define our own data structures by defining a **class**. A class defines a type along with a collection of operations that are related to that type. The class mechanism is one of the most important features in C++. In fact, a primary focus of the design of C++ is to make it possible to define **class types** that behave as naturally as the built-in types.

In this section, we'll describe a simple class that we can use in writing our bookstore program. We'll implement this class in later chapters as we learn more about types, expressions, statements, and functions.

To use a class we need to know three things:

- What is its name?
- Where is it defined?
- What operations does it support?

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

As we've seen, to use a library facility, we must include the associated header. Similarly, we use headers to access classes defined for our own applications. Conventionally, header file names are derived from the name of a class defined in that header. Header files that we write usually have a suffix of `.h`, but some programmers use `.H`, `.hpp`, or `.hxx`. The standard library headers typically have no suffix

at all. Compilers usually don't care about the form of header file names, but IDEs sometimes do.

1.5.1 The `Sales_item` Class

The purpose of the `Sales_item` class is to represent the total revenue, number of copies sold, and average sales price for a book. How these data are stored or computed is not our concern. To use a class, we need not care about how it is implemented. Instead, what we need to know is what operations objects of that type can perform.

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

```
Sales_item item;
```

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

In addition to being able to define variables of type `Sales_item`, we can:

- Call a function named `isbn` to fetch the ISBN from a `Sales_item` object.
- Use the input (`>>`) and output (`<<`) operators to read and write objects of type `Sales_item`.
- Use the assignment operator (`=`) to assign one `Sales_item` object to another.
- Use the addition operator (`+`) to add two `Sales_item` objects. The two objects must refer to the same ISBN. The result is a new `Sales_item` object whose ISBN is that of its operands and whose number sold and revenue are the sum of the corresponding values in its operands.
- Use the compound assignment operator (`+=`) to add one `Sales_item` object into another.

KEY CONCEPT: CLASSES DEFINE BEHAVIOR

The important thing to keep in mind when you read these programs is that the author of the `Sales_item` class defines *all* the actions that can be performed by objects of this class. That is, the `Sales_item` class defines what happens when a `Sales_item` object is created and what happens when the assignment, addition, or the input and output operators are applied to `Sales_items`.

In general, the class author determines all the operations that can be used on objects of the class type. For now, the only operations we know we can perform on `Sales_item` objects are the ones listed in this section.

Reading and Writing `Sales_item`s

Now that we know what operations we can use with `Sales_item` objects, we can write programs that use the class. For example, the following program reads data from the standard input into a `Sales_item` object and writes that `Sales_item` back onto the standard output:

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

If the input to this program is

```
0-201-70353-X 4 24.99
```

then the output will be

```
0-201-70353-X 4 99.96 24.99
```

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

This program starts with two `#include` directives, one of which uses a new form. Headers from the standard library are enclosed in angle brackets (`< >`). Those that are not part of the library are enclosed in double quotes (`" "`).

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

Adding `Sales_item`s

A more interesting example adds two `Sales_item` objects:

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item item1, item2;
    std::cin >> item1 >> item2;    // read a pair of transactions
    std::cout << item1 + item2 << std::endl; // print their sum
    return 0;
}
```

If we give this program the following input

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

our output is

```
0-201-78345-X 5 110 22
```

This program starts by including the `Sales_item` and `iostream` headers. Next we define two `Sales_item` objects to hold the transactions. We read data into these objects from the standard input. The output expression does the addition and prints the result.

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

Using File Redirection

It can be tedious to repeatedly type these transactions as input to the programs you are testing. Most operating systems support file redirection, which lets us associate a named file with the standard input and the standard output:

```
$ addItems <infile >outfile
```

Assuming `$` is the system prompt and our addition program has been compiled into an executable file named `addItems.exe` (or `addItems` on UNIX systems), this command will read transactions from a file named `infile` and write its output to a file named `outfile` in the current directory.

Exercises Section 1.5.1

Exercise 1.20: `http://www.informit.com/title/0321714113` contains a copy of `Sales_item.h` in the Chapter 1 code directory. Copy that file to your working directory. Use it to write a program that reads a set of book sales transactions, writing each transaction to the standard output.

Exercise 1.21: Write a program that reads two `Sales_item` objects that have the same ISBN and produces their sum.

Exercise 1.22: Write a program that reads several transactions for the same ISBN. Write the sum of all the transactions that were read.

1.5.2 A First Look at Member Functions

Our program that adds two `Sales_item`s should check whether the objects have the same ISBN. We'll do so as follows:

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

The difference between this program and the previous version is the `if` and its associated `else` branch. Even without understanding the `if` condition, we know what this program does. If the condition succeeds, then we write the same output as before and return `0`, indicating success. If the condition fails, we execute the block following the `else`, which prints a message and returns an error indicator.

What Is a Member Function?

The `if` condition

```
item1.isbn() == item2.isbn()
```

calls a **member function** named `isbn`. A member function is a function that is defined as part of a class. Member functions are sometimes referred to as **methods**.

Ordinarily, we call a member function on behalf of an object. For example, the first part of the left-hand operand of the equality expression

```
item1.isbn
```

uses the dot operator (the **`.` operator**) to say that we want "the `isbn` member of the object named `item1`." The dot operator applies only to objects of class type. The left-hand operand must be an object of class type, and the right-hand operand must name a member of that type. The result of the dot operator is the member named by the right-hand operand.

When we use the dot operator to access a member function, we usually do so to call that function. We call a function using the call operator (the **`()` operator**). The call operator is a pair of parentheses that enclose a (possibly empty) list of ***arguments***. The `isbn` member function does not take an argument. Thus,

```
item1.isbn()
```

calls the `isbn` function that is a member of the object named `item1`. This function returns the ISBN stored in `item1`.

The right-hand operand of the equality operator executes in the same way—it returns the ISBN stored in `item2`. If the ISBNs are the same, the condition is true; otherwise it is false.

Exercises Section 1.5.2

Exercise 1.23: Write a program that reads several transactions and counts how many transactions occur for each ISBN.

Exercise 1.24: Test the previous program by giving multiple transactions representing multiple ISBNs. The records for each ISBN should be grouped together.

1.6 The Bookstore Program

We are now ready to solve our original bookstore problem. We need to read a file of sales transactions and produce a report that shows, for each book, the total number of copies sold, the total revenue, and the average sales price. We'll assume that all the transactions for each ISBN are grouped together in the input.

Our program will combine the data for each ISBN in a variable named `total`. We'll use a second variable named `trans` to hold each transaction we read. If `trans` and `total` refer to the same ISBN, we'll update `total`. Otherwise we'll print `total` and reset it using the transaction we just read:

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item total; // variable to hold data for the next transaction
    // read the first transaction and ensure that there are data to process
    if (std::cin >> total) {
        Sales_item trans; // variable to hold the running sum
        // read and process the remaining transactions
        while (std::cin >> trans) {
            // if we're still processing the same book
            if (total.isbn() == trans.isbn())
                total += trans; // update the running total
            else {
                // print results for the previous book
                std::cout << total << std::endl;
                total = trans;  // total now refers to the next book
            }
        }
        std::cout << total << std::endl; // print the last transaction
    } else {
```

```
            // no input! warn the user
            std::cerr << "No data?!" << std::endl;
            return -1;  // indicate failure
        }
        return 0;
    }
```

This program is the most complicated one we've seen so far, but it uses only facilities that we have already seen.

As usual, we begin by including the headers that we use, `iostream` from the library and our own `Sales_item.h`. Inside `main` we define an object named `total`, which we'll use to sum the data for a given ISBN. We start by reading the first transaction into `total` and testing whether the read was successful. If the read fails, then there are no records and we fall through to the outermost `else` branch, which tells the user that there was no input.

Assuming we have successfully read a record, we execute the block following the outermost `if`. That block starts by defining the object named `trans`, which will hold our transactions as we read them. The `while` statement will read all the remaining records. As in our earlier programs, the `while` condition reads a value from the standard input. In this case, we read a `Sales_item` object into `trans`. As long as the read succeeds, we execute the body of the `while`.

The body of the `while` is a single `if` statement. The `if` checks whether the ISBNs are equal. If so, we use the compound assignment operator to add `trans` to `total`. If the ISBNs are not equal, we print the value stored in `total` and reset `total` by assigning `trans` to it. After executing the `if`, we return to the condition in the `while`, reading the next transaction, and so on until we run out of records.

When the `while` terminates, `total` contains the data for the last ISBN in the file. We write the data for the last ISBN in the last statement of the block that concludes the outermost `if` statement.

EXERCISES SECTION 1.6

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

Chapter Summary

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

Defined Terms

argument Value passed to a function.

assignment Obliterates an object's current value and replaces that value by a new one.

block Sequence of zero or more statements enclosed in curly braces.

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

built-in type Type, such as `int`, defined by the language.

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

character string literal Another term for string literal.

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

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

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

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

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

condition An expression that is evaluated as true or false. A value of zero is false; any other value yields true.

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

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

data structure A logical grouping of data and operations on that data.

edit-compile-debug The process of getting a program to execute properly.

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

expression The smallest unit of computation. An expression consists of one or more operands and usually one or more operators. Expressions are evaluated to produce a result. For example, assuming `i` and `j` are `int`s, then `i + j` is an expression and yields the sum of the two `int` values.

for statement Iteration statement that provides iterative execution. Often used to repeat a calculation a fixed number of times.

function Named unit of computation.

function body Block that defines the actions performed by a function.

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

header Mechanism whereby the definitions of a class or other names are made available to multiple programs. A program uses a header through a `#include` directive.

if statement Conditional execution based on the value of a specified condition. If the condition is true, the `if` body is executed. If not, the `else` body is executed if there is one.

initialize Give an object a value at the same time that it is created.

iostream Header that provides the library types for stream-oriented input and output.

istream Library type providing stream-oriented input.

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

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

manipulator Object, such as `std::endl`, that when read or written "manipulates" the stream itself.

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

method Synonym for member function.

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

ostream Library type providing stream-oriented output.

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

return type Type of the value returned by a function.

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

standard error Output stream used for error reporting. Ordinarily, the standard output and the standard error are tied to the window in which the program is executed.

standard input Input stream usually associated with the window in which the program executes.

standard library Collection of types and functions that every C++ compiler must support. The library provides the types that support IO. C++ programmers tend to talk about "the library," meaning the entire standard library. They also tend to refer to particular parts of the library by referring to a library type, such as the "`iostream` library," meaning the part of the standard library that defines the IO classes.

standard output Output stream usually associated with the window in which the program executes.

statement A part of a program that specifies an action to take place when the program is executed. An expression followed by a semicolon is a statement; other kinds

of statements include blocks and `if`, `for`, and `while` statements, all of which contain other statements within themselves.

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

string literal Sequence of zero or more characters enclosed in double quotes (`"a string literal"`).

uninitialized variable Variable that is not given an initial value. Variables of class type for which no initial value is specified are initialized as specified by the class definition. Variables of built-in type defined inside a function are uninitialized unless explicitly initialized. It is an error to try to use the value of an uninitialized variable. *Uninitialized variables are a rich source of bugs.*

variable A named object.

while statement Iteration statement that provides iterative execution so long as a specified condition is true. The body is executed zero or more times, depending on the truth value of the condition.

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

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

`+=` operator Compound assignment operator that adds the right-hand operand to the left and stores the result in the left-hand operand; `a += b` is equivalent to `a = a + b`.

`.` operator Dot operator. Left-hand operand must be an object of class type and the right-hand operand must be the name of a member of that object. The operator yields the named member of the given object.

`::` operator Scope operator. Among other uses, the scope operator is used to access names in a namespace. For example, `std::cout` denotes the name `cout` from the namespace `std`.

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

`--` operator Decrement operator. Subtracts `1` from the operand; `--i` is equivalent to `i = i - 1`.

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

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

`#include` Directive that makes code in a header available to a program.

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

`!=` operator The inequality operator. Tests whether the left-hand operand is not equal to the right-hand operand.

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

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

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

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

PART I

THE BASICS

CONTENTS

Every widely used programming language provides a common set of features, which differ in detail from one language to another. Understanding the details of how a language provides these features is the first step toward understanding the language. Among the most fundamental of these common features are

- Built-in types such as integers, characters, and so forth
- Variables, which let us give names to the objects we use
- Expressions and statements to manipulate values of these types
- Control structures, such as `if` or `while`, that allow us to conditionally or repeatedly execute a set of actions
- Functions that let us define callable units of computation

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

In C++, as in most programming languages, the type of an object determines what operations can be performed on it. Whether a particular expression is legal depends on the type of the objects in that expression. Some languages, such as Smalltalk and Python, check types at run time. In contrast, C++ is a statically typed language; type checking is done at compile time. As a consequence, the compiler must know the type of every name used in the program.

C++ provides a set of built-in types, operators to manipulate those types, and a small set of statements for program flow control. These elements form an alphabet from which we can write large, complicated, real-world systems. At this basic level, C++ is a simple language. Its expressive power arises from its support for mechanisms that allow the programmer to define new data structures. Using these facilities, programmers can shape the language to their own purposes without the language designers having to anticipate the programmers' needs.

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

The first step in mastering C++—learning the basics of the language and library—is the topic of Part I. Chapter 2 covers the built-in types and looks briefly at the mechanisms for defining our own new types. Chapter 3 introduces two of the most fundamental library types: `string` and `vector`. That chapter also covers arrays, which are a lower-level data structure built into C++ and many other languages. Chapters 4 through 6 cover expressions, statements, and functions. This part concludes in Chapter 7, which describes the basics of building our own class types. As we'll see, defining our own types brings together all that we've learned before, because writing a class entails using the facilities covered in Part I.

CHAPTER 2

VARIABLES AND BASIC TYPES

CONTENTS

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

C++ has extensive support for types. The language defines several primitive types (characters, integers, floating-point numbers, etc.) and provides mechanisms that let us define our own data types. The library uses these mechanisms to define more complicated types such as variable-length character strings, vectors, and so on. This chapter covers the built-in types and begins our coverage of how C++ supports more complicated types.

Types determine the meaning of the data and operations in our programs. The meaning of even as simple a statement as

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

depends on the types of `i` and `j`. If `i` and `j` are integers, this statement has the ordinary, arithmetic meaning of `+`. However, if `i` and `j` are `Sales_item` objects (§ 1.5.1, p. 20), this statement adds the components of these two objects.

2.1 Primitive Built-in Types

C++ defines a set of primitive types that include the **arithmetic types** and a special type named **`void`**. The arithmetic types represent characters, integers, boolean values, and floating-point numbers. The `void` type has no associated values and can be used in only a few circumstances, most commonly as the return type for functions that do not return a value.

2.1.1 Arithmetic Types

The arithmetic types are divided into two categories: **integral types** (which include character and boolean types) and floating-point types.

The size of—that is, the number of bits in—the arithmetic types varies across machines. The standard guarantees minimum sizes as listed in Table 2.1. However, compilers are allowed to use larger sizes for these types. Because the number of bits varies, the largest (or smallest) value that a type can represent also varies.

Table 2.1: C++: Arithmetic Types

Type	Meaning	Minimum Size
`bool`	boolean	NA
`char`	character	8 bits
`wchar_t`	wide character	16 bits
`char16_t`	Unicode character	16 bits
`char32_t`	Unicode character	32 bits
`short`	short integer	16 bits
`int`	integer	16 bits
`long`	long integer	32 bits
`long long`	long integer	64 bits
`float`	single-precision floating-point	6 significant digits
`double`	double-precision floating-point	10 significant digits
`long double`	extended-precision floating-point	10 significant digits

The `bool` type represents the truth values `true` and `false`.

There are several character types, most of which exist to support internationalization. The basic character type is `char`. A `char` is guaranteed to be big enough to hold numeric values corresponding to the characters in the machine's basic character set. That is, a `char` is the same size as a single machine byte.

The remaining character types—`wchar_t`, `char16_t`, and `char32_t`—are used for extended character sets. The `wchar_t` type is guaranteed to be large enough to hold any character in the machine's largest extended character set. The types `char16_t` and `char32_t` are intended for Unicode characters. (Unicode is a standard for representing characters used in essentially any natural language.)

The remaining integral types represent integer values of (potentially) different sizes. The language guarantees that an `int` will be at least as large as `short`, a `long` at least as large as an `int`, and `long long` at least as large as `long`. The type `long long` was introduced by the new standard. C++ 11

Machine-Level Representation of the Built-in Types

Computers store data as a sequence of bits, each holding a 0 or 1, such as

```
00011011011100010110010000111011 ...
```

Most computers deal with memory as chunks of bits of sizes that are powers of 2. The smallest chunk of addressable memory is referred to as a "byte." The basic unit of storage, usually a small number of bytes, is referred to as a "word." In C++ a byte has at least as many bits as are needed to hold a character in the machine's basic character set. On most machines a byte contains 8 bits and a word is either 32 or 64 bits, that is, 4 or 8 bytes.

Most computers associate a number (called an "address") with each byte in memory. On a machine with 8-bit bytes and 32-bit words, we might view a word of memory as follows

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

Here, the byte's address is on the left, with the 8 bits of the byte following the address.

We can use an address to refer to any of several variously sized collections of bits starting at that address. It is possible to speak of the word at address 736424 or the byte at address 736427. To give meaning to memory at a given address, we must know the type of the value stored there. The type determines how many bits are used and how to interpret those bits.

If the object at location 736424 has type `float` and if `float`s on this machine are stored in 32 bits, then we know that the object at that address spans the entire word. The value of that `float` depends on the details of how the machine stores floating-point numbers. Alternatively, if the object at location 736424 is an `unsigned char` on a machine using the ISO-Latin-1 character set, then the byte at that address represents a semicolon.

The floating-point types represent single-, double-, and extended-precision values. The standard specifies a minimum number of significant digits. Most compilers provide more precision than the specified minimum. Typically, `float`s are represented in one word (32 bits), `double`s in two words (64 bits), and `long double`s in either three or four words (96 or 128 bits). The `float` and `double` types typically yield about 7 and 16 significant digits, respectively. The type `long double`

is often used as a way to accommodate special-purpose floating-point hardware; its precision is more likely to vary from one implementation to another.

Signed and Unsigned Types

Except for `bool` and the extended character types, the integral types may be **signed** or **unsigned**. A signed type represents negative or positive numbers (including zero); an unsigned type represents only values greater than or equal to zero.

The types `int`, `short`, `long`, and `long long` are all signed. We obtain the corresponding unsigned type by adding `unsigned` to the type, such as `unsigned long`. The type `unsigned int` may be abbreviated as `unsigned`.

Unlike the other integer types, there are three distinct basic character types: `char`, `signed char`, and `unsigned char`. In particular, `char` is not the same type as `signed char`. Although there are three character types, there are only two representations: signed and unsigned. The (plain) `char` type uses one of these representations. Which of the other two character representations is equivalent to `char` depends on the compiler.

In an unsigned type, all the bits represent the value. For example, an 8-bit `unsigned char` can hold the values from 0 through 255 inclusive.

The standard does not define how signed types are represented, but does specify that the range should be evenly divided between positive and negative values. Hence, an 8-bit `signed char` is guaranteed to be able to hold values from –127 through 127; most modern machines use representations that allow values from –128 through 127.

Advice: Deciding Which Type to Use

C++, like C, is designed to let programs get close to the hardware when necessary. The arithmetic types are defined to cater to the peculiarities of various kinds of hardware. Accordingly, the number of arithmetic types in C++ can be bewildering. Most programmers can (and should) ignore these complexities by restricting the types they use. A few rules of thumb can be useful in deciding which type to use:

- **Use an unsigned type when you know that the values cannot be negative.**
- **Use `int` for integer arithmetic. `short` is usually too small and, in practice, `long` often has the same size as `int`. If your data values are larger than the minimum guaranteed size of an `int`, then use `long long`.**
- **Do not use plain `char` or `bool` in arithmetic expressions. Use them *only* to hold characters or truth values. Computations using `char` are especially problematic because `char` is `signed` on some machines and `unsigned` on others. If you need a tiny integer, explicitly specify either `signed char` or `unsigned char`.**
- **Use `double` for floating-point computations; `float` usually does not have enough precision, and the cost of double-precision calculations versus single-precision is negligible. In fact, on some machines, double-precision operations are faster than single. The precision offered by `long double` usually is unnecessary and often entails considerable run-time cost.**

EXERCISES SECTION 2.1.1

Exercise 2.1: What are the differences between `int`, `long`, `long long`, and `short`? Between an unsigned and a signed type? Between a `float` and a `double`?

Exercise 2.2: To calculate a mortgage payment, what types would you use for the rate, principal, and payment? Explain why you selected each type.

2.1.2 Type Conversions

The type of an object defines the data that an object might contain and what operations that object can perform. Among the operations that many types support is the ability to **convert** objects of the given type to other, related types.

Type conversions happen automatically when we use an object of one type where an object of another type is expected. We'll have more to say about conversions in § 4.11 (p. 159), but for now it is useful to understand what happens when we assign a value of one type to an object of another type.

When we assign one arithmetic type to another:

```
bool b = 42;              // b is true
int i = b;                // i has value 1
i = 3.14;                 // i has value 3
double pi = i;            // pi has value 3.0
unsigned char c = -1;     // assuming 8-bit chars, c has value 255
signed char c2 = 256;     // assuming 8-bit chars, the value of c2 is undefined
```

what happens depends on the range of the values that the types permit:

- When we assign one of the nonbool arithmetic types to a `bool` object, the result is `false` if the value is `0` and `true` otherwise.
- When we assign a `bool` to one of the other arithmetic types, the resulting value is `1` if the `bool` is `true` and `0` if the `bool` is `false`.
- When we assign a floating-point value to an object of integral type, the value is truncated. The value that is stored is the part before the decimal point.
- When we assign an integral value to an object of floating-point type, the fractional part is zero. Precision may be lost if the integer has more bits than the floating-point object can accommodate.
- If we assign an out-of-range value to an object of unsigned type, the result is the remainder of the value modulo the number of values the target type can hold. For example, an 8-bit `unsigned char` can hold values from 0 through 255, inclusive. If we assign a value outside this range, the compiler assigns the remainder of that value modulo 256. Therefore, assigning –1 to an 8-bit `unsigned char` gives that object the value 255.
- If we assign an out-of-range value to an object of signed type, the result is **undefined**. The program might appear to work, it might crash, or it might produce garbage values.

ADVICE: AVOID UNDEFINED AND IMPLEMENTATION-DEFINED BEHAVIOR

Undefined behavior results from errors that the compiler is not required (and sometimes is not able) to detect. Even if the code compiles, a program that executes an undefined expression is in error.

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

Similarly, programs usually should avoid implementation-defined behavior, such as assuming that the size of an `int` is a fixed and known value. Such programs are said to be *nonportable*. When the program is moved to another machine, code that relied on implementation-defined behavior may fail. Tracking down these sorts of problems in previously working programs is, mildly put, unpleasant.

The compiler applies these same type conversions when we use a value of one arithmetic type where a value of another arithmetic type is expected. For example, when we use a nonbool value as a condition (§ 1.4.1, p. 12), the arithmetic value is converted to bool in the same way that it would be converted if we had assigned that arithmetic value to a bool variable:

```
int i = 42;
if (i) //  condition will evaluate as true
    i = 0;
```

If the value is 0, then the condition is false; all other (nonzero) values yield true.

By the same token, when we use a bool in an arithmetic expression, its value always converts to either 0 or 1. As a result, using a bool in an arithmetic expression is almost surely incorrect.

Expressions Involving Unsigned Types

Although we are unlikely to intentionally assign a negative value to an object of unsigned type, we can (all too easily) write code that does so implicitly. For example, if we use both unsigned and int values in an arithmetic expression, the int value ordinarily is converted to unsigned. Converting an int to unsigned executes the same way as if we assigned the int to an unsigned:

```
unsigned u = 10;
int i = -42;
std::cout << i + i << std::endl;   //  prints -84
std::cout << u + i << std::endl;   //  if 32-bit ints, prints 4294967264
```

In the first expression, we add two (negative) int values and obtain the expected result. In the second expression, the int value -42 is converted to unsigned before the addition is done. Converting a negative number to unsigned behaves exactly as if we had attempted to assign that negative value to an unsigned object. The value "wraps around" as described above.

Regardless of whether one or both operands are unsigned, if we subtract a value from an unsigned, we must be sure that the result cannot be negative:

```
unsigned u1 = 42, u2 = 10;
std::cout << u1 - u2 << std::endl; //  ok: result is 32
std::cout << u2 - u1 << std::endl; //  ok: but the result will wrap around
```

The fact that an unsigned cannot be less than zero also affects how we write loops. For example, in the exercises to § 1.4.1 (p. 13), you were to write a loop that used the decrement operator to print the numbers from `10` down to `0`. The loop you wrote probably looked something like

```
for (int i = 10; i >= 0; --i)
    std::cout << i << std::endl;
```

We might think we could rewrite this loop using an `unsigned`. After all, we don't plan to print negative numbers. However, this simple change in type means that our loop will never terminate:

```
//  WRONG: u can never be less than 0; the condition will always succeed
for (unsigned u = 10; u >= 0; --u)
    std::cout << u << std::endl;
```

Consider what happens when `u` is `0`. On that iteration, we'll print `0` and then execute the expression in the `for` loop. That expression, `--u`, subtracts `1` from `u`. That result, `-1`, won't fit in an `unsigned` value. As with any other out-of-range value, `-1` will be transformed to an `unsigned` value. Assuming 32-bit `int`s, the result of `--u`, when `u` is `0`, is `4294967295`.

One way to write this loop is to use a `while` instead of a `for`. Using a `while` lets us decrement before (rather than after) printing our value:

```
unsigned u = 11; //  start the loop one past the first element we want to print
while (u > 0) {
     --u;         //  decrement first, so that the last iteration will print 0
    std::cout << u << std::endl;
}
```

This loop starts by decrementing the value of the loop control variable. On the last iteration, `u` will be `1` on entry to the loop. We'll decrement that value, meaning that we'll print `0` on this iteration. When we next test `u` in the `while` condition, its value will be `0` and the loop will exit. Because we start by decrementing `u`, we have to initialize `u` to a value one greater than the first value we want to print. Hence, we initialize `u` to `11`, so that the first value printed is `10`.

Caution: Don't Mix Signed and Unsigned Types

Expressions that mix signed and unsigned values can yield surprising results when the signed value is negative. It is essential to remember that signed values are automatically converted to unsigned. For example, in an expression like `a * b`, if `a` is `-1` and `b` is `1`, then if both `a` and `b` are `int`s, the value is, as expected `-1`. However, if `a` is `int` and `b` is an `unsigned`, then the value of this expression depends on how many bits an `int` has on the particular machine. On our machine, this expression yields `4294967295`.

Exercises Section 2.1.2

Exercise 2.3: What output will the following code produce?

```
unsigned u = 10, u2 = 42;
std::cout << u2 - u << std::endl;
std::cout << u - u2 << std::endl;
int i = 10, i2 = 42;
std::cout << i2 - i << std::endl;
std::cout << i - i2 << std::endl;
std::cout << i - u << std::endl;
std::cout << u - i << std::endl;
```

Exercise 2.4: Write a program to check whether your predictions were correct. If not, study this section until you understand what the problem is.

2.1.3 Literals

A value, such as `42`, is known as a **literal** because its value self-evident. Every literal has a type. The form and value of a literal determine its type.

Integer and Floating-Point Literals

We can write an integer literal using decimal, octal, or hexadecimal notation. Integer literals that begin with `0` (zero) are interpreted as octal. Those that begin with either `0x` or `0X` are interpreted as hexadecimal. For example, we can write the value `20` in any of the following three ways:

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

The type of an integer literal depends on its value and notation. By default, decimal literals are signed whereas octal and hexadecimal literals can be either signed or unsigned types. A decimal literal has the smallest type of `int`, `long`, or `long long` (i.e., the first type in this list) in which the literal's value fits. Octal and hexadecimal literals have the smallest type of `int`, `unsigned int`, `long`, `unsigned long`, `long long`, or `unsigned long long` in which the literal's value fits. It is an error to use a literal that is too large to fit in the largest related type. There are no literals of type `short`. We'll see in Table 2.2 (p. 40) that we can override these defaults by using a suffix.

Although integer literals may be stored in signed types, technically speaking, the value of a decimal literal is never a negative number. If we write what appears to be a negative decimal literal, for example, `-42`, the minus sign is *not* part of the literal. The minus sign is an operator that negates the value of its (literal) operand.

Floating-point literals include either a decimal point or an exponent specified using scientific notation. Using scientific notation, the exponent is indicated by either `E` or `e`:

```
3.14159    3.14159E0    0.    0e0    .001
```

By default, floating-point literals have type `double`. We can override the default using a suffix from Table 2.2 (overleaf).

Character and Character String Literals

A character enclosed within single quotes is a literal of type `char`. Zero or more characters enclosed in double quotation marks is a string literal:

```
'a'   // character literal
"Hello World!"   // string literal
```

The type of a string literal is ***array*** of constant `char`s, a type we'll discuss in § 3.5.4 (p. 122). The compiler appends a null character (`'\0'`) to every string literal. Thus, the actual size of a string literal is one more than its apparent size. For example, the literal `'A'` represents the single character `A`, whereas the string literal `"A"` represents an array of two characters, the letter `A` and the null character.

Two string literals that appear adjacent to one another and that are separated only by spaces, tabs, or newlines are concatenated into a single literal. We use this form of literal when we need to write a literal that would otherwise be too large to fit comfortably on a single line:

```
// multiline string literal
std::cout << "a really, really long string literal "
             "that spans two lines" << std::endl;
```

Escape Sequences

Some characters, such as backspace or control characters, have no visible image. Such characters are **nonprintable**. Other characters (single and double quotation marks, question mark, and backslash) have special meaning in the language. Our programs cannot use any of these characters directly. Instead, we use an **escape sequence** to represent such characters. An escape sequence begins with a backslash. The language defines several escape sequences:

newline	`\n`	horizontal tab	`\t`	alert (bell)	`\a`
vertical tab	`\v`	backspace	`\b`	double quote	`\"`
backslash	`\\`	question mark	`\?`	single quote	`\'`
carriage return	`\r`	formfeed	`\f`		

We use an escape sequence as if it were a single character:

```
std::cout << '\n';          // prints a newline
std::cout << "\tHi!\n";     // prints a tab followd by "Hi!" and a newline
```

We can also write a generalized escape sequence, which is `\x` followed by one or more hexadecimal digits or a `\` followed by one, two, or three octal digits. The value represents the numerical value of the character. Some examples (assuming the Latin-1 character set):

```
\7 (bell)     \12 (newline)      \40 (blank)
\0 (null)     \115 ('M')         \x4d ('M')
```

As with an escape sequence defined by the language, we use these escape sequences as we would any other character:

```
std::cout << "Hi \x4dO\115!\n"; // prints Hi MOM! followed by a newline
std::cout << '\115' << '\n';    // prints M followed by a newline
```

Note that if a \ is followed by more than three octal digits, only the first three are associated with the \. For example, `"\1234"` represents two characters: the character represented by the octal value 123 and the character 4. In contrast, `\x` uses up all the hex digits following it; `"\x1234"` represents a single, 16-bit character composed from the bits corresponding to these four hexadecimal digits. Because most machines have 8-bit `chars`, such values are unlikely to be useful. Ordinarily, hexadecimal characters with more than 8 bits are used with extended characters sets using one of the prefixes from Table 2.2.

Specifying the Type of a Literal

We can override the default type of an integer, floating- point, or character literal by supplying a suffix or prefix as listed in Table 2.2.

```
L'a'      // wide character literal, type is wchar_t
u8"hi!"   // utf-8 string literal (utf-8 encodes a Unicode character in 8 bits)
42ULL     // unsigned integer literal, type is unsigned long long
1E-3F     // single-precision floating-point literal, type is float
3.14159L  // extended-precision floating-point literal, type is long double
```

When you write a `long` literal, use the uppercase `L`; the lowercase letter `l` is too easily mistaken for the digit 1.

Table 2.2: Specifying the Type of a Literal

Character and Character String Literals

Prefix	Meaning	Type
u	Unicode 16 character	char16_t
U	Unicode 32 character	char32_t
L	wide character	wchar_t
u8	utf-8 (string literals only)	char

Integer Literals		Floating-Point Literals	
Suffix	**Minimum Type**	**Suffix**	**Type**
u or U	unsigned	f or F	float
l or L	long	l or L	long double
ll or LL	long long		

We can independently specify the signedness and size of an integral literal. If the suffix contains a `U`, then the literal has an unsigned type, so a decimal, octal, or hexadecimal literal with a `U` suffix has the smallest type of `unsigned int`, `unsigned long`, or `unsigned long long` in which the literal's value fits. If the suffix contains an `L`, then the literal's type will be at least `long`; if the suffix contains `LL`, then the literal's type will be either `long long` or `unsigned long long`.

We can combine `U` with either `L` or `LL`. For example, a literal with a suffix of `UL` will be either `unsigned long` or `unsigned long long`, depending on whether its value fits in `unsigned long`.

Boolean and Pointer Literals

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

```
bool test = false;
```

The word `nullptr` is a pointer literal. We'll have more to say about pointers and `nullptr` in § 2.3.2 (p. 52).

EXERCISES SECTION 2.1.3

Exercise 2.5: Determine the type of each of the following literals. Explain the differences among the literals in each of the four examples:

(a) `'a', L'a', "a", L"a"`
(b) `10, 10u, 10L, 10uL, 012, 0xC`
(c) `3.14, 3.14f, 3.14L`
(d) `10, 10u, 10., 10e-2`

Exercise 2.6: What, if any, are the differences between the following definitions:

```
int month = 9, day = 7;
int month = 09, day = 07;
```

Exercise 2.7: What values do these literals represent? What type does each have?

(a) `"Who goes with F\145rgus?\012"`
(b) `3.14e1L` (c) `1024f` (d) `3.14L`

Exercise 2.8: Using escape sequences, write a program to print `2M` followed by a newline. Modify the program to print `2`, then a tab, then an `M`, followed by a newline.

2.2 Variables

A ***variable*** provides us with named storage that our programs can manipulate. Each variable in C++ has a type. The type determines the size and layout of the variable's memory, the range of values that can be stored within that memory, and the set of operations that can be applied to the variable. C++ programmers tend to refer to variables as "variables" or "objects" interchangeably.

2.2.1 Variable Definitions

A simple variable definition consists of a **type specifier**, followed by a list of one or more variable names separated by commas, and ends with a semicolon. Each name

in the list has the type defined by the type specifier. A definition may (optionally) provide an initial value for one or more of the names it defines:

```
int sum = 0, value, //  sum, value, and units_sold have type int
    units_sold = 0; //  sum and units_sold have initial value 0
Sales_item item;    //  item has type Sales_item (see § 1.5.1 (p. 20))
//  string is a library type, representing a variable-length sequence of characters
std::string book("0-201-78345-X"); //  book initialized from string literal
```

The definition of `book` uses the `std::string` library type. Like `iostream` (§ 1.2, p. 7), `string` is defined in namespace `std`. We'll have more to say about the `string` type in Chapter 3. For now, what's useful to know is that a `string` is a type that represents a variable-length sequence of characters. The `string` library gives us several ways to initialize `string` objects. One of these ways is as a copy of a string literal (§ 2.1.3, p. 39). Thus, `book` is initialized to hold the characters `0-201-78345-X`.

TERMINOLOGY: WHAT IS AN OBJECT?

C++ programmers tend to be cavalier in their use of the term *object*. Most generally, an object is a region of memory that can contain data and has a type.

Some use the term *object* only to refer to variables or values of class types. Others distinguish between named and unnamed objects, using the term *variable* to refer to named objects. Still others distinguish between objects and values, using the term *object* for data that can be changed by the program and the term *value* for data that are read-only.

In this book, we'll follow the more general usage that an object is a region of memory that has a type. We will freely use the term *object* regardless of whether the object has built-in or class type, is named or unnamed, or can be read or written.

Initializers

An object that is **initialized** gets the specified value at the moment it is created. The values used to initialize a variable can be arbitrarily complicated expressions. When a definition defines two or more variables, the name of each object becomes visible immediately. Thus, it is possible to initialize a variable to the value of one defined earlier in the same definition.

```
//  ok: price is defined and initialized before it is used to initialize discount
double price = 109.99, discount = price * 0.16;
//  ok: call applyDiscount and use the return value to initialize salePrice
double salePrice = applyDiscount(price, discount);
```

Initialization in C++ is a surprisingly complicated topic and one we will return to again and again. Many programmers are confused by the use of the = symbol to initialize a variable. It is tempting to think of initialization as a form of assignment, but initialization and assignment are different operations in C++. This concept is particularly confusing because in many languages the distinction is irrelevant

and can be ignored. Moreover, even in C++ the distinction often doesn't matter. Nonetheless, it is a crucial concept and one we will reiterate throughout the text.

Initialization is not assignment. Initialization happens when a variable is given a value when it is created. Assignment obliterates an object's current value and replaces that value with a new one.

List Initialization

One way in which initialization is a complicated topic is that the language defines several different forms of initialization. For example, we can use any of the following four different ways to define an `int` variable named `units_sold` and initialize it to `0`:

```
int units_sold = 0;
int units_sold = {0};
int units_sold{0};
int units_sold(0);
```

The generalized use of curly braces for initialization was introduced as part of the new standard. This form of initialization previously had been allowed only in more restricted ways. For reasons we'll learn about in § 3.3.1 (p. 98), this form of initialization is referred to as **list initialization**. Braced lists of initializers can now be used whenever we initialize an object and in some cases when we assign a new value to an object.

C++ 11

When used with variables of built-in type, this form of initialization has one important property: The compiler will not let us list initialize variables of built-in type if the initializer might lead to the loss of information:

```
long double ld = 3.1415926536;
int a{ld}, b = {ld}; // error: narrowing conversion required
int c(ld), d = ld;   // ok: but value will be truncated
```

The compiler rejects the initializations of `a` and `b` because using a `long double` to initialize an `int` is likely to lose data. At a minimum, the fractional part of `ld` will be truncated. In addition, the integer part in `ld` might be too large to fit in an `int`.

As presented here, the distinction might seem trivial—after all, we'd be unlikely to directly initialize an `int` from a `long double`. However, as we'll see in Chapter 16, such initializations might happen unintentionally. We'll say more about these forms of initialization in § 3.2.1 (p. 84) and § 3.3.1 (p. 98).

Default Initialization

When we define a variable without an initializer, the variable is **default initialized**. Such variables are given the "default" value. What that default value is depends on the type of the variable and may also depend on where the variable is defined.

The value of an object of built-in type that is not explicitly initialized depends on where it is defined. Variables defined outside any function body are initialized to zero. With one exception, which we cover in § 6.1.1 (p. 205), variables of built-in

type defined inside a function are **uninitialized**. The value of an uninitialized variable of built-in type is undefined (§ 2.1.2, p. 36). It is an error to copy or otherwise try to access the value of a variable whose value is undefined.

Each class controls how we initialize objects of that class type. In particular, it is up to the class whether we can define objects of that type without an initializer. If we can, the class determines what value the resulting object will have.

Most classes let us define objects without explicit initializers. Such classes supply an appropriate default value for us. For example, as we've just seen, the library `string` class says that if we do not supply an initializer, then the resulting `string` is the empty string:

```
std::string empty;   //  empty implicitly initialized to the empty string
Sales_item item;     //  default-initialized Sales_item object
```

Some classes require that every object be explicitly initialized. The compiler will complain if we try to create an object of such a class with no initializer.

Uninitialized objects of built-in type defined inside a function body have undefined value. Objects of class type that we do not explicitly initialize have a value that is defined by the class.

Exercises Section 2.2.1

Exercise 2.9: Explain the following definitions. For those that are illegal, explain what's wrong and how to correct it.

(a) `std::cin >> int input_value;`
(b) `int i = { 3.14 };`
(c) `double salary = wage = 9999.99;`
(d) `int i = 3.14;`

Exercise 2.10: What are the initial values, if any, of each of the following variables?

```
std::string global_str;
int global_int;
int main()
{
    int local_int;
    std::string local_str;
}
```

2.2.2 Variable Declarations and Definitions

To allow programs to be written in logical parts, C++ supports what is commonly known as ***separate compilation***. Separate compilation lets us split our programs into several files, each of which can be compiled independently.

When we separate a program into multiple files, we need a way to share code across those files. For example, code defined in one file may need to use a variable defined in another file. As a concrete example, consider `std::cout` and

CAUTION: UNINITIALIZED VARIABLES CAUSE RUN-TIME PROBLEMS

An uninitialized variable has an indeterminate value. Trying to use the value of an uninitialized variable is an error that is often hard to debug. Moreover, the compiler is not required to detect such errors, although most will warn about at least some uses of uninitialized variables.

What happens when we use an uninitialized variable is undefined. Sometimes, we're lucky and our program crashes as soon as we access the object. Once we track down the location of the crash, it is usually easy to see that the variable was not properly initialized. Other times, the program completes but produces erroneous results. Even worse, the results may appear correct on one run of our program but fail on a subsequent run. Moreover, adding code to the program in an unrelated location can cause what we thought was a correct program to start producing incorrect results.

We recommend initializing every object of built-in type. It is not always necessary, but it is easier and safer to provide an initializer until you can be certain it is safe to omit the initializer.

`std::cin`. These are objects defined somewhere in the standard library, yet our programs can use these objects.

To support separate compilation, C++ distinguishes between declarations and definitions. A **declaration** makes a name known to the program. A file that wants to use a name defined elsewhere includes a declaration for that name. A **definition** creates the associated entity.

A variable declaration specifies the type and name of a variable. A variable definition is a declaration. In addition to specifying the name and type, a definition also allocates storage and may provide the variable with an initial value.

To obtain a declaration that is not also a definition, we add the `extern` keyword and may not provide an explicit initializer:

```
extern int i;    // declares but does not define i
int j;           // declares and defines j
```

Any declaration that includes an explicit initializer is a definition. We can provide an initializer on a variable defined as `extern`, but doing so overrides the `extern`. An `extern` that has an initializer is a definition:

```
extern double pi = 3.1416; // definition
```

It is an error to provide an initializer on an `extern` inside a function.

Variables must be defined exactly once but can be declared many times.

The distinction between a declaration and a definition may seem obscure at this point but is actually important. To use a variable in more than one file requires declarations that are separate from the variable's definition. To use the same variable in multiple files, we must define that variable in one—and only one—file. Other files that use that variable must declare—but not define—that variable.

We'll have more to say about how C++ supports separate compilation in § 2.6.3 (p. 76) and § 6.1.3 (p. 207).

Exercises Section 2.2.2

Exercise 2.11: Explain whether each of the following is a declaration or a definition:

(a) `extern int ix = 1024;`
(b) `int iy;`
(c) `extern int iz;`

Key Concept: Static Typing

C++ is a *statically typed* language, which means that types are checked at compile time. The process by which types are checked is referred to as *type checking*.

As we've seen, the type of an object constrains the operations that the object can perform. In C++, the compiler checks whether the operations we write are supported by the types we use. If we try to do things that the type does not support, the compiler generates an error message and does not produce an executable file.

As our programs get more complicated, we'll see that static type checking can help find bugs. However, a consequence of static checking is that the type of every entity we use must be known to the compiler. As one example, we must declare the type of a variable before we can use that variable.

2.2.3 Identifiers

Identifiers in C++ can be composed of letters, digits, and the underscore character. The language imposes no limit on name length. Identifiers must begin with either a letter or an underscore. Identifiers are case-sensitive; upper- and lowercase letters are distinct:

```
// defines four different int variables
int somename, someName, SomeName, SOMENAME;
```

The language reserves a set of names, listed in Tables 2.3 and Table 2.4, for its own use. These names may not be used as identifiers.

The standard also reserves a set of names for use in the standard library. The identifiers we define in our own programs may not contain two consecutive underscores, nor can an identifier begin with an underscore followed immediately by an uppercase letter. In addition, identifiers defined outside a function may not begin with an underscore.

Conventions for Variable Names

There are a number of generally accepted conventions for naming variables. Following these conventions can improve the readability of a program.

- An identifier should give some indication of its meaning.
- Variable names normally are lowercase—`index`, not `Index` or `INDEX`.
- Like `Sales_item`, classes we define usually begin with an uppercase letter.
- Identifiers with multiple words should visually distinguish each word, for example, `student_loan` or `studentLoan`, not `studentloan`.

Naming conventions are most useful when followed consistently.

Table 2.3: C++ Keywords

alignas	continue	friend	register	true
alignof	decltype	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	do	int	signed	typename
break	double	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert	using
char	enum	new	static_cast	virtual
char16_t	explicit	noexcept	struct	void
char32_t	export	nullptr	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr	float	protected	thread_local	
const_cast	for	public	throw	

Table 2.4: C++ Alternative Operator Names

and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bitor	not	or	xor	

EXERCISES SECTION 2.2.3

Exercise 2.12: Which, if any, of the following names are invalid?

(a) `int double = 3.14;` (b) `int _;`
(c) `int catch-22;` (d) `int 1_or_2 = 1;`
(e) `double Double = 3.14;`

2.2.4 Scope of a Name

At any particular point in a program, each name that is in use refers to a specific entity—a variable, function, type, and so on. However, a given name can be reused to refer to different entities at different points in the program.

A **scope** is a part of the program in which a name has a particular meaning. Most scopes in C++ are delimited by curly braces.

The same name can refer to different entities in different scopes. Names are visible from the point where they are declared until the end of the scope in which the declaration appears.

As an example, consider the program from § 1.4.2 (p. 13):

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

This program defines three names—`main`, `sum`, and `val`—and uses the namespace name `std`, along with two names from that namespace—`cout` and `endl`.

The name `main` is defined outside any curly braces. The name `main`—like most names defined outside a function—has **global scope**. Once declared, names at the global scope are accessible throughout the program. The name `sum` is defined within the scope of the block that is the body of the `main` function. It is accessible from its point of declaration throughout the rest of the `main` function but not outside of it. The variable `sum` has **block scope**. The name `val` is defined in the scope of the `for` statement. It can be used in that statement but not elsewhere in `main`.

ADVICE: DEFINE VARIABLES WHERE YOU FIRST USE THEM

It is usually a good idea to define an object near the point at which the object is first used. Doing so improves readability by making it easy to find the definition of the variable. More importantly, it is often easier to give the variable a useful initial value when the variable is defined close to where it is first used.

Nested Scopes

Scopes can contain other scopes. The contained (or nested) scope is referred to as an **inner scope**, the containing scope is the **outer scope**.

Once a name has been declared in a scope, that name can be used by scopes nested inside that scope. Names declared in the outer scope can also be redefined in an inner scope:

```
#include <iostream>
// Program for illustration purposes only: It is bad style for a function
// to use a global variable and also define a local variable with the same name
int reused = 42;  // reused has global scope
int main()
{
    int unique = 0; // unique has block scope
    // output #1: uses global reused; prints 42 0
    std::cout << reused << " " << unique << std::endl;
    int reused = 0; // new, local object named reused hides global reused
    // output #2: uses local reused; prints 0 0
    std::cout << reused << " " <<  unique << std::endl;
    // output #3: explicitly requests the global reused; prints 42 0
    std::cout << ::reused << " " <<  unique << std::endl;
    return 0;
}
```

Output #1 appears before the local definition of `reused`. Therefore, this output statement uses the name `reused` that is defined in the global scope. This statement prints `42 0`. Output #2 occurs after the local definition of `reused`. The local `reused` is now **in scope**. Thus, this second output statement uses the local object named `reused` rather than the global one and prints `0 0`. Output #3 uses the scope operator (§ 1.2, p. 8) to override the default scoping rules. The global scope has no name. Hence, when the scope operator has an empty left-hand side, it is a request to fetch the name on the right-hand side from the global scope. Thus, this expression uses the global `reused` and prints `42 0`.

It is almost always a bad idea to define a local variable with the same name as a global variable that the function uses or might use.

EXERCISES SECTION 2.2.4

Exercise 2.13: What is the value of `j` in the following program?

```
int i = 42;
int main()
{
    int i = 100;
    int j = i;
}
```

Exercise 2.14: Is the following program legal? If so, what values are printed?

```
int i = 100, sum = 0;
for (int i = 0; i != 10; ++i)
     sum += i;
std::cout << i << " " << sum << std::endl;
```

2.3 Compound Types

A **compound type** is a type that is defined in terms of another type. C++ has several compound types, two of which—references and pointers—we'll cover in this chapter.

Defining variables of compound type is more complicated than the declarations we've seen so far. In § 2.2 (p. 41) we said that simple declarations consist of a type followed by a list of variable names. More generally, a declaration is a **base type** followed by a list of **declarators**. Each declarator names a variable and gives the variable a type that is related to the base type.

The declarations we have seen so far have declarators that are nothing more than variable names. The type of such variables is the base type of the declaration. More complicated declarators specify variables with compound types that are built from the base type of the declaration.

2.3.1 References

The new standard introduced a new kind of reference: an "rvalue reference," which we'll cover in § 13.6.1 (p. 532). These references are primarily intended for use inside classes. Technically speaking, when we use the term *reference*, we mean "lvalue reference."

A **reference** defines an alternative name for an object. A reference type "refers to" another type. We define a reference type by writing a declarator of the form `&d`, where `d` is the name being declared:

```
int ival = 1024;
int &refVal = ival;   //  refVal refers to (is another name for) ival
int &refVal2;         //  error: a reference must be initialized
```

Ordinarily, when we initialize a variable, the value of the initializer is copied into the object we are creating. When we define a reference, instead of copying the initializer's value, we **bind** the reference to its initializer. Once initialized, a reference remains bound to its initial object. There is no way to rebind a reference to refer to a different object. Because there is no way to rebind a reference, references *must* be initialized.

A Reference Is an Alias

A reference is not an object. Instead, a reference is *just another name for an already existing object*.

After a reference has been defined, *all* operations on that reference are actually operations on the object to which the reference is bound:

```
refVal = 2;       //  assigns 2 to the object to which refVal refers, i.e., to ival
int ii = refVal; //  same as ii = ival
```

When we assign to a reference, we are assigning to the object to which the reference is bound. When we fetch the value of a reference, we are really fetching the value of the object to which the reference is bound. Similarly, when we use a reference as an initializer, we are really using the object to which the reference is bound:

```
// ok: refVal3 is bound to the object to which refVal is bound, i.e., to ival
int &refVal3 = refVal;
// initializes i from the value in the object to which refVal is bound
int i = refVal;  // ok: initializes i to the same value as ival
```

Because references are not objects, we may not define a reference to a reference.

Reference Definitions

We can define multiple references in a single definition. Each identifier that is a reference must be preceded by the & symbol:

```
int i = 1024, i2 = 2048;  // i and i2 are both ints
int &r = i, r2 = i2;      // r is a reference bound to i; r2 is an int
int i3 = 1024, &ri = i3;  // i3 is an int; ri is a reference bound to i3
int &r3 = i3, &r4 = i2;   // both r3 and r4 are references
```

With two exceptions that we'll cover in § 2.4.1 (p. 61) and § 15.2.3 (p. 601), the type of a reference and the object to which the reference refers must match exactly. Moreover, for reasons we'll explore in § 2.4.1, a reference may be bound only to an object, not to a literal or to the result of a more general expression:

```
int &refVal4 = 10;    // error: initializer must be an object
double dval = 3.14;
int &refVal5 = dval; // error: initializer must be an int object
```

EXERCISES SECTION 2.3.1

Exercise 2.15: Which of the following definitions, if any, are invalid? Why?

```
(a) int ival = 1.01;      (b) int &rval1 = 1.01;
(c) int &rval2 = ival;    (d) int &rval3;
```

Exercise 2.16: Which, if any, of the following assignments are invalid? If they are valid, explain what they do.

```
int i = 0, &r1 = i; double d = 0, &r2 = d;
(a) r2 = 3.14159;     (b) r2 = r1;
(c) i = r2;           (d) r1 = d;
```

Exercise 2.17: What does the following code print?

```
int i, &ri = i;
i = 5; ri = 10;
std::cout << i << " " << ri << std::endl;
```

2.3.2 Pointers

A **pointer** is a compound type that "points to" another type. Like references, pointers are used for indirect access to other objects. Unlike a reference, a pointer is an object in its own right. Pointers can be assigned and copied; a single pointer can point to several different objects over its lifetime. Unlike a reference, a pointer need not be initialized at the time it is defined. Like other built-in types, pointers defined at block scope have undefined value if they are not initialized.

Pointers are often hard to understand. Debugging problems due to pointer errors bedevil even experienced programmers.

We define a pointer type by writing a declarator of the form `*d`, where `d` is the name being defined. The `*` must be repeated for each pointer variable:

```
int *ip1, *ip2;  // both ip1 and ip2 are pointers to int
double dp, *dp2; // dp2 is a pointer to double; dp is a double
```

Taking the Address of an Object

A pointer holds the address of another object. We get the address of an object by usin the address-of operator (the **& operator**):

```
int ival = 42;
int *p = &ival; // p holds the address of ival; p is a pointer to ival
```

The second statement defines `p` as a pointer to `int` and initializes `p` to point to the `int` object named `ival`. Because references are not objects, they don't have addresses. Hence, we may not define a pointer to a reference.

With two exceptions, which we cover in § 2.4.2 (p. 62) and § 15.2.3 (p. 601), the types of the pointer and the object to which it points must match:

```
double dval;
double *pd = &dval;   // ok: initializer is the address of a double
double *pd2 = pd;     // ok: initializer is a pointer to double
int *pi = pd;   // error: types of pi and pd differ
pi = &dval;     // error: assigning the address of a double to a pointer to int
```

The types must match because the type of the pointer is used to infer the type of the object to which the pointer points. If a pointer addressed an object of another type, operations performed on the underlying object would fail.

Pointer Value

The value (i.e., the address) stored in a pointer can be in one of four states:

1. It can point to an object.
2. It can point to the location just immediately past the end of an object.
3. It can be a null pointer, indicating that it is not bound to any object.
4. It can be invalid; values other than the preceding three are invalid.

It is an error to copy or otherwise try to access the value of an invalid pointer. As when we use an uninitialized variable, this error is one that the compiler is unlikely to detect. The result of accessing an invalid pointer is undefined. Therefore, we must always know whether a given pointer is valid.

Although pointers in cases 2 and 3 are valid, there are limits on what we can do with such pointers. Because these pointers do not point to any object, we may not use them to access the (supposed) object to which the pointer points. If we do attempt to access an object through such pointers, the behavior is undefined.

Using a Pointer to Access an Object

When a pointer points to an object, we can use the dereference operator (the **`*` operator**) to access that object:

```
int ival = 42;
int *p = &ival; // p holds the address of ival; p is a pointer to ival
cout << *p;     // * yields the object to which p points; prints 42
```

Dereferencing a pointer yields the object to which the pointer points. We can assign to that object by assigning to the result of the dereference:

```
*p = 0;      // * yields the object; we assign a new value to ival through p
cout << *p; // prints 0
```

When we assign to `*p`, we are assigning to the object to which `p` points.

We may dereference only a valid pointer that points to an object.

KEY CONCEPT: SOME SYMBOLS HAVE MULTIPLE MEANINGS

Some symbols, such as `&` and `*`, are used as both an operator in an expression and as part of a declaration. The context in which a symbol is used determines what the symbol means:

```
int i = 42;
int &r = i;    // & follows a type and is part of a declaration; r is a reference
int *p;        // * follows a type and is part of a declaration; p is a pointer
p = &i;        // & is used in an expression as the address-of operator
*p = i;        // * is used in an expression as the dereference operator
int &r2 = *p; // & is part of the declaration; * is the dereference operator
```

In declarations, `&` and `*` are used to form compound types. In expressions, these same symbols are used to denote an operator. Because the same symbol is used with very different meanings, it can be helpful to ignore appearances and think of them as if they were different symbols.

Null Pointers

A **null pointer** does not point to any object. Code can check whether a pointer is null before attempting to use it. There are several ways to obtain a null pointer:

```
int *p1 = nullptr; // equivalent to int *p1 = 0;
int *p2 = 0;       // directly initializes p2 from the literal constant 0
// must #include cstdlib
int *p3 = NULL;    // equivalent to int *p3 = 0;
```

C++ 11

The most direct approach is to initialize the pointer using the literal **`nullptr`**, which was introduced by the new standard. `nullptr` is a literal that has a special type that can be converted (§ 2.1.2, p. 35) to any other pointer type. Alternatively, we can initialize a pointer to the literal `0`, as we do in the definition of `p2`.

Older programs sometimes use a **preprocessor variable** named `NULL`, which the `cstdlib` header defines as `0`.

We'll describe the preprocessor in a bit more detail in § 2.6.3 (p. 77). What's useful to know now is that the preprocessor is a program that runs before the compiler. Preprocessor variables are managed by the preprocessor, and are not part of the `std` namespace. As a result, we refer to them directly without the `std::` prefix.

When we use a preprocessor variable, the preprocessor automatically replaces the variable by its value. Hence, initializing a pointer to `NULL` is equivalent to initializing it to `0`. Modern C++ programs generally should avoid using `NULL` and use `nullptr` instead.

It is illegal to assign an `int` variable to a pointer, even if the variable's value happens to be `0`.

```
int zero = 0;
pi = zero;          // error: cannot assign an int to a pointer
```

ADVICE: INITIALIZE ALL POINTERS

Uninitialized pointers are a common source of run-time errors.

As with any other uninitialized variable, what happens when we use an uninitialized pointer is undefined. Using an uninitialized pointer almost always results in a run-time crash. However, debugging the resulting crashes can be surprisingly hard.

Under most compilers, when we use an uninitialized pointer, the bits in the memory in which the pointer resides are used as an address. Using an uninitialized pointer is a request to access a supposed object at that supposed location. There is no way to distinguish a valid address from an invalid one formed from the bits that happen to be in the memory in which the pointer was allocated.

Our recommendation to initialize all variables is particularly important for pointers. If possible, define a pointer only after the object to which it should point has been defined. If there is no object to bind to a pointer, then initialize the pointer to `nullptr` or zero. That way, the program can detect that the pointer does not point to an object.

Assignment and Pointers

Both pointers and references give indirect access to other objects. However, there are important differences in how they do so. The most important is that a reference

is not an object. Once we have defined a reference, there is no way to make that reference refer to a different object. When we use a reference, we always get the object to which the reference was initially bound.

There is no such identity between a pointer and the address that it holds. As with any other (nonreference) variable, when we assign to a pointer, we give the pointer itself a new value. Assignment makes the pointer point to a different object:

```
int i = 42;
int *pi = 0;    // pi is initialized but addresses no object
int *pi2 = &i; // pi2 initialized to hold the address of i
int *pi3;       // if pi3 is defined inside a block, pi3 is uninitialized
pi3 = pi2;          // pi3 and pi2 address the same object, e.g., i
pi2 = 0;            // pi2 now addresses no object
```

It can be hard to keep straight whether an assignment changes the pointer or the object to which the pointer points. The important thing to keep in mind is that assignment changes its left-hand operand. When we write

```
pi = &ival; // value in pi is changed; pi now points to ival
```

we assign a new value to `pi`, which changes the address that `pi` holds. On the other hand, when we write

```
*pi = 0;     // value in ival is changed; pi is unchanged
```

then `*pi` (i.e., the value to which `pi` points) is changed.

Other Pointer Operations

So long as the pointer has a valid value, we can use a pointer in a condition. Just as when we use an arithmetic value in a condition (§ 2.1.2, p. 35), if the pointer is 0, then the condition is `false`:

```
int ival = 1024;
int *pi = 0;       // pi is a valid, null pointer
int *pi2 = &ival; // pi2 is a valid pointer that holds the address of ival
if (pi)  // pi has value 0, so condition evaluates as false
    // ...
if (pi2) // pi2 points to ival, so it is not 0; the condition evaluates as true
    // ...
```

Any nonzero pointer evaluates as `true`

Given two valid pointers of the same type, we can compare them using the equality (`==`) or inequality (`!=`) operators. The result of these operators has type `bool`. Two pointers are equal if they hold the same address and unequal otherwise. Two pointers hold the same address (i.e., are equal) if they are both null, if they address the same object, or if they are both pointers one past the same object. Note that it is possible for a pointer to an object and a pointer one past the end of a different object to hold the same address. Such pointers will compare equal.

Because these operations use the value of the pointer, a pointer used in a condition or in a comparsion must be a valid pointer. Using an invalid pointer as a condition or in a comparison is undefined.

§ 3.5.3 (p. 117) will cover additional pointer operations.

`void*` Pointers

The type **`void*`** is a special pointer type that can hold the address of any object. Like any other pointer, a `void*` pointer holds an address, but the type of the object at that address is unknown:

```
double obj = 3.14, *pd = &obj;
// ok: void* can hold the address value of any data pointer type
void *pv = &obj;  // obj can be an object of any type
pv = pd;          // pv can hold a pointer to any type
```

There are only a limited number of things we can do with a `void*` pointer: We can compare it to another pointer, we can pass it to or return it from a function, and we can assign it to another `void*` pointer. We cannot use a `void*` to operate on the object it addresses—we don't know that object's type, and the type determines what operations we can perform on the object.

Generally, we use a `void*` pointer to deal with memory as memory, rather than using the pointer to access the object stored in that memory. We'll cover using `void*` pointers in this way in § 19.1.1 (p. 821). § 4.11.3 (p. 163) will show how we can retrieve the address stored in a `void*` pointer.

Exercises Section 2.3.2

Exercise 2.18: Write code to change the value of a pointer. Write code to change the value to which the pointer points.

Exercise 2.19: Explain the key differences between pointers and references.

Exercise 2.20: What does the following program do?

```
int i = 42;
int *p1 = &i;
*p1 = *p1 * *p1;
```

Exercise 2.21: Explain each of the following definitions. Indicate whether any are illegal and, if so, why.

```
int i = 0;
(a) double* dp = &i;  (b) int *ip = i; (c) int *p = &i;
```

Exercise 2.22: Assuming `p` is a pointer to `int`, explain the following code:

```
if (p) // ...
if (*p) // ...
```

Exercise 2.23: Given a pointer `p`, can you determine whether `p` points to a valid object? If so, how? If not, why not?

Exercise 2.24: Why is the initialization of `p` legal but that of `lp` illegal?

```
int i = 42;    void *p = &i;     long *lp = &i;
```

2.3.3 Understanding Compound Type Declarations

As we've seen, a variable definition consists of a base type and a list of declarators. Each declarator can relate its variable to the base type differently from the other declarators in the same definition. Thus, a single definition might define variables of different types:

```
// i is an int; p is a pointer to int; r is a reference to int
int i = 1024, *p = &i, &r = i;
```

Many programmers are confused by the interaction between the base type and the type modification that may be part of a declarator.

Defining Multiple Variables

It is a common misconception to think that the type modifier (`*` or `&`) applies to all the variables defined in a single statement. Part of the problem arises because we can put whitespace between the type modifier and the name being declared:

```
int* p;   // legal but might be misleading
```

We say that this definition might be misleading because it suggests that `int*` is the type of each variable declared in that statement. Despite appearances, the base type of this declaration is `int`, not `int*`. The `*` modifies the type of `p`. It says nothing about any other objects that might be declared in the same statement:

```
int* p1, p2; // p1 is a pointer to int; p2 is an int
```

There are two common styles used to define multiple variables with pointer or reference type. The first places the type modifier adjacent to the identifier:

```
int *p1, *p2; // both p1 and p2 are pointers to int
```

This style emphasizes that the variable has the indicated compound type.

The second places the type modifier with the type but defines only one variable per statement:

```
int* p1;      // p1 is a pointer to int
int* p2;      // p2 is a pointer to int
```

This style emphasizes that the declaration defines a compound type.

There is no single right way to define pointers or references. The important thing is to choose a style and use it consistently.

In this book we use the first style and place the `*` (or the `&`) with the variable name.

Pointers to Pointers

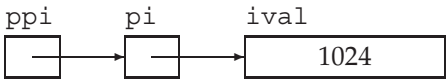
In general, there are no limits to how many type modifiers can be applied to a declarator. When there is more than one modifier, they combine in ways that are logical but not always obvious. As one example, consider a pointer. A pointer is

an object in memory, so like any object it has an address. Therefore, we can store the address of a pointer in another pointer.

We indicate each pointer level by its own `*`. That is, we write `**` for a pointer to a pointer, `***` for a pointer to a pointer to a pointer, and so on:

```
int ival = 1024;
int *pi = &ival;    // pi points to an int
int **ppi = &pi;    // ppi points to a pointer to an int
```

Here `pi` is a pointer to an `int` and `ppi` is a pointer to a pointer to an `int`. We might represent these objects as



Just as dereferencing a pointer to an `int` yields an `int`, dereferencing a pointer to a pointer yields a pointer. To access the underlying object, we must dereference the original pointer twice:

```
cout << "The value of ival\n"
     << "direct value: " << ival << "\n"
     << "indirect value: " << *pi << "\n"
     << "doubly indirect value: " << **ppi
     << endl;
```

This program prints the value of `ival` three different ways: first, directly; then, through the pointer to `int` in `pi`; and finally, by dereferencing `ppi` twice to get to the underlying value in `ival`.

References to Pointers

A reference is not an object. Hence, we may not have a pointer to a reference. However, because a pointer is an object, we can define a reference to a pointer:

```
int i = 42;
int *p;         // p is a pointer to int
int *&r = p;    // r is a reference to the pointer p
r = &i; // r refers to a pointer; assigning &i to r makes p point to i
*r = 0; // dereferencing r yields i, the object to which p points; changes i to 0
```

The easiest way to understand the type of `r` is to read the definition right to left. The symbol closest to the name of the variable (in this case the `&` in `&r`) is the one that has the most immediate effect on the variable's type. Thus, we know that `r` is a reference. The rest of the declarator determines the type to which `r` refers. The next symbol, `*` in this case, says that the type `r` refers to is a pointer type. Finally, the base type of the declaration says that `r` is a reference to a pointer to an `int`.

It can be easier to understand complicated pointer or reference declarations if you read them from right to left.

EXERCISES SECTION 2.3.3

Exercise 2.25: Determine the types and values of each of the following variables.

(a) `int* ip, i, &r = i;` (b) `int i, *ip = 0;` (c) `int* ip, ip2;`

2.4 const Qualifier

Sometimes we want to define a variable whose value we know cannot be changed. For example, we might want to use a variable to refer to the size of a buffer size. Using a variable makes it easy for us to change the size of the buffer if we decided the original size wasn't what we needed. On the other hand, we'd also like to prevent code from inadvertently giving a new value to the variable we use to represent the buffer size. We can make a variable unchangeable by defining the variable's type as **`const`**:

```
const int bufSize = 512;     // input buffer size
```

defines `bufSize` as a constant. Any attempt to assign to `bufSize` is an error:

```
bufSize = 512; // error: attempt to write to const object
```

Because we can't change the value of a `const` object after we create it, it must be initialized. As usual, the initializer may be an arbitrarily complicated expression:

```
const int i = get_size();  // ok: initialized at run time
const int j = 42;          // ok: initialized at compile time
const int k;               // error: k is uninitialized const
```

Initialization and const

As we have observed many times, the type of an object defines the operations that can be performed by that object. A `const` type can use most but not all of the same operations as its nonconst version. The one restriction is that we may use only those operations that cannot change an object. So, for example, we can use a `const int` in arithmetic expressions in exactly the same way as a plain, nonconst `int`. A `const int` converts to `bool` the same way as a plain `int`, and so on.

Among the operations that don't change the value of an object is initialization—when we use an object to initialize another object, it doesn't matter whether either or both of the objects are `const`s:

```
int i = 42;
const int ci = i;     // ok: the value in i is copied into ci
int j = ci;           // ok: the value in ci is copied into j
```

Although `ci` is a `const int`, the value in `ci` is an `int`. The `const`ness of `ci` matters only for operations that might change `ci`. When we copy `ci` to initialize `j`, we don't care that `ci` is a `const`. Copying an object doesn't change that object. Once the copy is made, the new object has no further access to the original object.

By Default, `const` Objects Are Local to a File

When a `const` object is initialized from a compile-time constant, such as in our definition of `bufSize`:

```
const int bufSize = 512;     // input buffer size
```

the compiler will usually replace uses of the variable with its corresponding value during compilation. That is, the compiler will generate code using the value `512` in the places that our code uses `bufSize`.

To substitute the value for the variable, the compiler has to see the variable's initializer. When we split a program into multiple files, every file that uses the `const` must have access to its initializer. In order to see the initializer, the variable must be defined in every file that wants to use the variable's value (§ 2.2.2, p. 45). To support this usage, yet avoid multiple definitions of the same variable, `const` variables are defined as local to the file. When we define a `const` with the same name in multiple files, it is as if we had written definitions for separate variables in each file.

Sometimes we have a `const` variable that we want to share across multiple files but whose initializer is not a constant expression. In this case, we don't want the compiler to generate a separate variable in each file. Instead, we want the `const` object to behave like other (nonconst) variables. We want to define the `const` in one file, and declare it in the other files that use that object.

To define a single instance of a `const` variable, we use the keyword `extern` on both its definition and declaration(s):

```
// file_1.cc defines and initializes a const that is accessible to other files
extern const int bufSize = fcn();

// file_1.h
extern const int bufSize; // same bufSize as defined in file_1.cc
```

In this program, `file_1.cc` defines and initializes `bufSize`. Because this declaration includes an initializer, it is (as usual) a definition. However, because `bufSize` is `const`, we must specify `extern` in order for `bufSize` to be used in other files.

The declaration in `file_1.h` is also `extern`. In this case, the `extern` signifies that `bufSize` is not local to this file and that its definition will occur elsewhere.

To share a `const` object among multiple files, you must define the variable as `extern`.

Exercises Section 2.4

Exercise 2.26: Which of the following are legal? For those that are illegal, explain why.

(a) `const int buf;` (b) `int cnt = 0;`
(c) `const int sz = cnt;` (d) `++cnt; ++sz;`

2.4.1 References to const

As with any other object, we can bind a reference to an object of a const type. To do so we use a **reference to const**, which is a reference that refers to a const type. Unlike an ordinary reference, a reference to const cannot be used to change the object to which the reference is bound:

```
const int ci = 1024;
const int &r1 =  ci;   // ok: both reference and underlying object are const
r1 = 42;               // error: r1 is a reference to const
int &r2 = ci;          // error: nonconst reference to a const object
```

Because we cannot assign directly to ci, we also should not be able to use a reference to change ci. Therefore, the initialization of r2 is an error. If this initialization were legal, we could use r2 to change the value of its underlying object.

TERMINOLOGY: const REFERENCE IS A REFERENCE TO const

C++ programmers tend to abbreviate the phrase "reference to const" as "const reference." This abbreviation makes sense—if you remember that it is an abbreviation.

Technically speaking, there are no const references. A reference is not an object, so we cannot make a reference itself const. Indeed, because there is no way to make a reference refer to a different object, in some sense all references are const. Whether a reference refers to a const or nonconst type affects what we can do with that reference, not whether we can alter the binding of the reference itself.

Initialization and References to const

In § 2.3.1 (p. 51) we noted that there are two exceptions to the rule that the type of a reference must match the type of the object to which it refers. The first exception is that we can initialize a reference to const from any expression that can be converted (§ 2.1.2, p. 35) to the type of the reference. In particular, we can bind a reference to const to a nonconst object, a literal, or a more general expression:

```
int i = 42;
const int &r1 = i;       // we can bind a const int& to a plain int object
const int &r2 = 42;      // ok: r1 is a reference to const
const int &r3 = r1 * 2;  // ok: r3 is a reference to const
int &r4 = r * 2;         // error: r4 is a plain, nonconst reference
```

The easiest way to understand this difference in initialization rules is to consider what happens when we bind a reference to an object of a different type:

```
double dval = 3.14;
const int &ri = dval;
```

Here ri refers to an int. Operations on ri will be integer operations, but dval is a floating-point number, not an integer. To ensure that the object to which ri is bound is an int, the compiler transforms this code into something like

```
const int temp = dval;    // create a temporary const int from the double
const int &ri = temp;     // bind ri to that temporary
```

In this case, `ri` is bound to a **temporary** object. A temporary object is an unnamed object created by the compiler when it needs a place to store a result from evaluating an expression. C++ programmers often use the word temporary as an abbreviation for temporary object.

Now consider what could happen if this initialization were allowed but `ri` was not `const`. If `ri` weren't `const`, we could assign to `ri`. Doing so would change the object to which `ri` is bound. That object is a temporary, not `dval`. The programmer who made `ri` refer to `dval` would probably expect that assigning to `ri` would change `dval`. After all, why assign to `ri` unless the intent is to change the object to which `ri` is bound? Because binding a reference to a temporary is almost surely *not* what the programmer intended, the language makes it illegal.

A Reference to `const` May Refer to an Object That Is Not `const`

It is important to realize that a reference to `const` restricts only what we can do through that reference. Binding a reference to `const` to an object says nothing about whether the underlying object itself is `const`. Because the underlying object might be nonconst, it might be changed by other means:

```
int i = 42;
int &r1 = i;           // r1 bound to i
const int &r2 = i;     // r2 also bound to i; but cannot be used to change i
r1 = 0;                // r1 is not const; i is now 0
r2 = 0;                // error: r2 is a reference to const
```

Binding `r2` to the (nonconst) `int` `i` is legal. However, we cannot use `r2` to change `i`. Even so, the value in `i` still might change. We can change `i` by assigning to it directly, or by assigning to another reference bound to `i`, such as `r1`.

2.4.2 Pointers and `const`

As with references, we can define pointers that point to either `const` or nonconst types. Like a reference to `const`, a **pointer to `const`** (§ 2.4.1, p. 61) may not be used to change the object to which the pointer points. We may store the address of a `const` object only in a pointer to `const`:

```
const double pi = 3.14;    // pi is const; its value may not be changed
double *ptr = &pi;         // error: ptr is a plain pointer
const double *cptr = &pi;  // ok: cptr may point to a double that is const
*cptr = 42;                // error: cannot assign to *cptr
```

In § 2.3.2 (p. 52) we noted that there are two exceptions to the rule that the types of a pointer and the object to which it points must match. The first exception is that we can use a pointer to `const` to point to a nonconst object:

```
double dval = 3.14;        // dval is a double; its value can be changed
cptr = &dval;              // ok: but can't change dval through cptr
```

Like a reference to const, a pointer to const says nothing about whether the object to which the pointer points is const. Defining a pointer as a pointer to const affects only what we can do with the pointer. It is important to remember that there is no guarantee that an object pointed to by a pointer to const won't change.

It may be helpful to think of pointers and references to const as pointers or references "that *think* they point or refer to const."

const Pointers

Unlike references, pointers are objects. Hence, as with any other object type, we can have a pointer that is itself const. Like any other const object, a **const pointer** must be initialized, and once initialized, its value (i.e., the address that it holds) may not be changed. We indicate that the pointer is const by putting the const after the *. This placement indicates that it is the pointer, not the pointed-to type, that is const:

```
int errNumb = 0;
int *const curErr = &errNumb;  // curErr will always point to errNumb
const double pi = 3.14159;
const double *const pip = &pi; // pip is a const pointer to a const object
```

As we saw in § 2.3.3 (p. 58), the easiest way to understand these declarations is to read them from right to left. In this case, the symbol closest to curErr is const, which means that curErr itself will be a const object. The type of that object is formed from the rest of the declarator. The next symbol in the declarator is *, which means that curErr is a const pointer. Finally, the base type of the declaration completes the type of curErr, which is a const pointer to an object of type int. Similarly, pip is a const pointer to an object of type const double.

The fact that a pointer is itself const says nothing about whether we can use the pointer to change the underlying object. Whether we can change that object depends entirely on the type to which the pointer points. For example, pip is a const pointer to const. Neither the value of the object addressed by pip nor the address stored in pip can be changed. On the other hand, curErr addresses a plain, nonconst int. We can use curErr to change the value of errNumb:

```
*pip = 2.72;      // error: pip is a pointer to const
// if the object to which curErr points (i.e., errNumb) is nonzero
if (*curErr) {
    errorHandler();
    *curErr = 0; // ok: reset the value of the object to which curErr is bound
}
```

2.4.3 Top-Level const

As we've seen, a pointer is an object that can point to a different object. As a result, we can talk independently about whether a pointer is const and whether

EXERCISES SECTION 2.4.2

Exercise 2.27: Which of the following initializations are legal? Explain why.

```
(a) int i = -1, &r = 0;          (b) int *const p2 = &i2;
(c) const int i = -1, &r = 0;    (d) const int *const p3 = &i2;
(e) const int *p1 = &i2;         (f) const int &const r2;
(g) const int i2 = i, &r = i;
```

Exercise 2.28: Explain the following definitions. Identify any that are illegal.

```
(a) int i, *const cp;        (b) int *p1, *const p2;
(c) const int ic, &r = ic;   (d) const int *const p3;
(e) const int *p;
```

Exercise 2.29: Uing the variables in the previous exercise, which of the following assignments are legal? Explain why.

```
(a) i = ic;                  (b) p1 = p3;
(c) p1 = &ic;                (d) p3 = &ic;
(e) p2 = p1;                 (f) ic = *p3;
```

the objects to which it can point are `const`. We use the term **top-level `const`** to indicate that the pointer itself is a `const`. When a pointer can point to a `const` object, we refer to that `const` as a **low-level `const`**.

More generally, top-level `const` indicates that an object itself is `const`. Top-level `const` can appear in any object type, i.e., one of the built-in arithmetic types, a class type, or a pointer type. Low-level `const` appears in the base type of compound types such as pointers or references. Note that pointer types, unlike most other types, can have both top-level and low-level `const` independently:

```
int i = 0;
int *const p1 = &i;  // we can't change the value of p1; const is top-level
const int ci = 42;   // we cannot change ci; const is top-level
const int *p2 = &ci; // we can change p2; const is low-level
const int *const p3 = p2; // right-most const is top-level, left-most is not
const int &r = ci;   // const in reference types is always low-level
```

The distinction between top-level and low-level matters when we copy an object. When we copy an object, top-level `const`s are ignored:

```
i = ci;  // ok: copying the value of ci; top-level const in ci is ignored
p2 = p3; // ok: pointed-to type matches; top-level const in p3 is ignored
```

Copying an object doesn't change the copied object. As a result, it is immaterial whether the object copied from or copied into is `const`.

On the other hand, low-level `const` is never ignored. When we copy an object, both objects must have the same low-level `const` qualification or there must be a conversion between the types of the two objects. In general, we can convert a nonconst to `const` but not the other way round:

```
int *p = p3; // error: p3 has a low-level const but p doesn't
p2 = p3;     // ok: p2 has the same low-level const qualification as p3
p2 = &i;     // ok: we can convert int* to const int*
int &r = ci; // error: can't bind an ordinary int& to a const int object
const int &r2 = i; // ok: can bind const int& to plain int
```

p3 has both a top-level and low-level const. When we copy p3, we can ignore its top-level const but not the fact that it points to a const type. Hence, we cannot use p3 to initialize p, which points to a plain (nonconst) int. On the other hand, we can assign p3 to p2. Both pointers have the same (low-level const) type. The fact that p3 is a const pointer (i.e., that it has a top-level const) doesn't matter.

EXERCISES SECTION 2.4.3

Exercise 2.30: For each of the following declarations indicate whether the object being declared has top-level or low-level const.

```
const int v2 = 0;    int v1 = v2;
int *p1 = &v1, &r1 = v1;
const int *p2 = &v2, *const p3 = &i, &r2 = v2;
```

Exercise 2.31: Given the declarations in the previous exercise determine whether the following assignments are legal. Explain how the top-level or low-level const applies in each case.

```
r1 = v2;
p1 = p2;    p2 = p1;
p1 = p3;    p2 = p3;
```

2.4.4 constexpr and Constant Expressions

A **constant expression** is an expression whose value cannot change and that can be evaluated at compile time. A literal is a constant expression. A const object that is initialized from a constant expression is also a constant expression. As we'll see, there are several contexts in the language that require constant expressions.

Whether a given object (or expression) is a constant expression depends on the types and the initializers. For example:

```
const int max_files = 20;     // max_files is a constant expression
const int limit = max_files + 1; // limit is a constant expression
int staff_size = 27;         // staff_size is not a constant expression
const int sz = get_size(); // sz is not a constant expression
```

Although staff_size is initialized from a literal, it is not a constant expression because it is a plain int, not a const int. On the other hand, even though sz is a const, the value of its initializer is not known until run time. Hence, sz is not a constant expression.

`constexpr` Variables

In a large system, it can be difficult to determine (for certain) that an initializer is a constant expression. We might define a `const` variable with an initializer that we think is a constant expression. However, when we use that variable in a context that requires a constant expression we may discover that the initializer was not a constant expression. In general, the definition of an object and its use in such a context can be widely separated.

C++ 11

Under the new standard, we can ask the compiler to verify that a variable is a constant expression by declaring the variable in a **`constexpr`** declaration. Variables declared as `constexpr` are implicitly `const` and must be initialized by constant expressions:

```
constexpr int mf = 20;          //  20 is a constant expression
constexpr int limit = mf + 1; //  mf + 1 is a constant expression
constexpr int sz = size();     //  ok only if size is a constexpr function
```

Although we cannot use an ordinary function as an initializer for a `constexpr` variable, we'll see in § 6.5.2 (p. 239) that the new standard lets us define certain functions as `constexpr`. Such functions must be simple enough that the compiler can evaluate them at compile time. We can use `constexpr` functions in the initializer of a `constexpr` variable.

Generally, it is a good idea to use `constexpr` for variables that you intend to use as constant expressions.

Literal Types

Because a constant expression is one that can be evaluated at compile time, there are limits on the types that we can use in a `constexpr` declaration. The types we can use in a `constexpr` are known as "literal types" because they are simple enough to have literal values.

Of the types we have used so far, the arithmetic, reference, and pointer types are literal types. Our `Sales_item` class and the library IO and `string` types are not literal types. Hence, we cannot define variables of these types as `constexpr`s. We'll see other kinds of literal types in § 7.5.6 (p. 299) and § 19.3 (p. 832).

Although we can define both pointers and reference as `constexpr`s, the objects we use to initialize them are strictly limited. We can initialize a `constexpr` pointer from the `nullptr` literal or the literal (i.e., constant expression) `0`. We can also point to (or bind to) an object that remains at a fixed address.

For reasons we'll cover in § 6.1.1 (p. 204), variables defined inside a function ordinarily are not stored at a fixed address. Hence, we cannot use a `constexpr` pointer to point to such variables. On the other hand, the address of an object defined outside of any function is a constant expression, and so may be used to initialize a `constexpr` pointer. We'll see in § 6.1.1 (p. 205), that functions may define variables that exist across calls to that function. Like an object defined outside any function, these special local objects also have fixed addresses. Therefore, a `constexpr` reference may be bound to, and a `constexpr` pointer may address, such variables.

Pointers and `constexpr`

It is important to understand that when we define a pointer in a `constexpr` declaration, the `constexpr` specifier applies to the pointer, not the type to which the pointer points:

```
const int *p = nullptr;      // p is a pointer to a const int
constexpr int *q = nullptr; // q is a const pointer to int
```

Despite appearances, the types of `p` and `q` are quite different; `p` is a pointer to `const`, whereas `q` is a constant pointer. The difference is a consequence of the fact that `constexpr` imposes a top-level `const` (§ 2.4.3, p. 63) on the objects it defines.

Like any other constant pointer, a `constexpr` pointer may point to a `const` or a nonconst type:

```
constexpr int *np = nullptr; // np is a constant pointer to int that is null
int j = 0;
constexpr int i = 42;  // type of i is const int
// i and j must be defined outside any function
constexpr const int *p = &i; // p is a constant pointer to the const int i
constexpr int *p1 = &j;      // p1 is a constant pointer to the int j
```

EXERCISES SECTION 2.4.4

Exercise 2.32: Is the following code legal or not? If not, how might you make it legal?

```
int null = 0, *p = null;
```

2.5 Dealing with Types

As our programs get more complicated, we'll see that the types we use also get more complicated. Complications in using types arise in two different ways. Some types are hard to "spell." That is, they have forms that are tedious and error-prone to write. Moreover, the form of a complicated type can obscure its purpose or meaning. The other source of complication is that sometimes it is hard to determine the exact type we need. Doing so can require us to look back into the context of the program.

2.5.1 Type Aliases

A **type alias** is a name that is a synonym for another type. Type aliases let us simplify complicated type definitions, making those types easier to use. Type aliases also let us emphasize the purpose for which a type is used.

We can define a type alias in one of two ways. Traditionally, we use a **typedef**:

```
typedef double wages;     // wages is a synonym for double
typedef wages base, *p;   // base is a synonym for double, p for double*
```

The keyword `typedef` may appear as part of the base type of a declaration (§ 2.3, p. 50). Declarations that include `typedef` define type aliases rather than variables. As in any other declaration, the declarators can include type modifiers that define compound types built from the base type of the definition.

C++ 11

The new standard introduced a second way to define a type alias, via an **alias declaration**:

```
using SI = Sales_item;    // SI is a synonym for Sales_item
```

An alias declaration starts with the keyword `using` followed by the alias name and an =. The alias declaration defines the name on the left-hand side of the = as an alias for the type that appears on the right-hand side.

A type alias is a type name and can appear wherever a type name can appear:

```
wages hourly, weekly;     // same as double hourly, weekly;
SI item;                  // same as Sales_item item
```

Pointers, `const`, and Type Aliases

Declarations that use type aliases that represent compound types and `const` can yield surprising results. For example, the following declarations use the type `pstring`, which is an alias for the the type `char*`:

```
typedef char *pstring;
const pstring cstr = 0; // cstr is a constant pointer to char
const pstring *ps;      // ps is a pointer to a constant pointer to char
```

The base type in these declarations is `const pstring`. As usual, a `const` that appears in the base type modifies the given type. The type of `pstring` is "pointer to `char`." So, `const pstring` is a constant pointer to `char`—not a pointer to `const char`.

It can be tempting, albeit incorrect, to interpret a declaration that uses a type alias by conceptually replacing the alias with its corresponding type:

```
const char *cstr = 0;   // wrong interpretation of const pstring cstr
```

However, this interpretation is wrong. When we use `pstring` in a declaration, the base type of the declaration is a pointer type. When we rewrite the declaration using `char*`, the base type is `char` and the `*` is part of the declarator. In this case, `const char` is the base type. This rewrite declares `cstr` as a pointer to `const char` rather than as a `const` pointer to `char`.

2.5.2 The `auto` Type Specifier

It is not uncommon to want to store the value of an expression in a variable. To declare the variable, we have to know the type of that expression. When we write a program, it can be surprisingly difficult—and sometimes even impossible—to determine the type of an expression. Under the new standard, we can let the compiler figure out the type for us by using the **`auto`** type specifier. Unlike type specifiers, such as `double`, that name a specific type, `auto` tells the compiler to deduce

C++ 11

the type from the initializer. By implication, a variable that uses `auto` as its type specifier must have an initializer:

```
// the type of item is deduced from the type of the result of adding val1 and val2
auto item = val1 + val2; // item initialized to the result of val1 + val2
```

Here the compiler will deduce the type of `item` from the type returned by applying `+` to `val1` and `val2`. If `val1` and `val2` are `Sales_item` objects (§ 1.5, p. 19), `item` will have type `Sales_item`. If those variables are type `double`, then `item` has type `double`, and so on.

As with any other type specifier, we can define multiple variables using `auto`. Because a declaration can involve only a single base type, the initializers for all the variables in the declaration must have types that are consistent with each other:

```
auto i = 0, *p = &i;      // ok: i is int and p is a pointer to int
auto sz = 0, pi = 3.14;   // error: inconsistent types for sz and pi
```

Compound Types, `const`, and `auto`

The type that the compiler infers for `auto` is not always exactly the same as the initializer's type. Instead, the compiler adjusts the type to conform to normal initialization rules.

First, as we've seen, when we use a reference, we are really using the object to which the reference refers. In particular, when we use a reference as an initializer, the initializer is the corresponding object. The compiler uses that object's type for `auto`'s type deduction:

```
int i = 0, &r = i;
auto a = r;  // a is an int (r is an alias for i, which has type int)
```

Second, `auto` ordinarily ignores top-level `const`s (§ 2.4.3, p. 63). As usual in initializations, low-level `const`s, such as when an initializer is a pointer to `const`, are kept:

```
const int ci = i, &cr = ci;
auto b = ci;  // b is an int (top-level const in ci is dropped)
auto c = cr;  // c is an int (cr is an alias for ci whose const is top-level)
auto d = &i;  // d is an int* (& of an int object is int*)
auto e = &ci; // e is const int* (& of a const object is low-level const)
```

If we want the deduced type to have a top-level `const`, we must say so explicitly:

```
const auto f = ci;  // deduced type of ci is int; f has type const int
```

We can also specify that we want a reference to the `auto`-deduced type. Normal initialization rules still apply:

```
auto &g = ci;        // g is a const int& that is bound to ci
auto &h = 42;        // error: we can't bind a plain reference to a literal
const auto &j = 42; // ok: we can bind a const reference to a literal
```

When we ask for a reference to an `auto`-deduced type, top-level `const`s in the initializer are not ignored. As usual, `const`s are not top-level when we bind a reference to an initializer.

When we define several variables in the same statement, it is important to remember that a reference or pointer is part of a particular declarator and not part of the base type for the declaration. As usual, the initializers must provide consistent `auto`-deduced types:

```
auto k = ci, &l = i;     // k is int; l is int&
auto &m = ci, *p = &ci; // m is a const int&; p is a pointer to const int
// error: type deduced from i is int; type deduced from &ci is const int
auto &n = i, *p2 = &ci;
```

EXERCISES SECTION 2.5.2

Exercise 2.33: Using the variable definitions from this section, determine what happens in each of these assignments:

```
a = 42;   b = 42;   c = 42;
d = 42;   e = 42;   g = 42;
```

Exercise 2.34: Write a program containing the variables and assignments from the previous exercise. Print the variables before and after the assignments to check whether your predictions in the previous exercise were correct. If not, study the examples until you can convince yourself you know what led you to the wrong conclusion.

Exercise 2.35: Determine the types deduced in each of the following definitions. Once you've figured out the types, write a program to see whether you were correct.

```
const int i = 42;
auto j = i; const auto &k = i; auto *p = &i;
const auto j2 = i, &k2 = i;
```

2.5.3 The `decltype` Type Specifier

Sometimes we want to define a variable with a type that the compiler deduces from an expression but do not want to use that expression to initialize the variable. For such cases, the new standard introduced a second type specifier, **`decltype`**, which returns the type of its operand. The compiler analyzes the expression to determine its type but does not evaluate the expression:

C++ 11

```
decltype(f()) sum = x; // sum has whatever type f returns
```

Here, the compiler does not call `f`, but it uses the type that such a call would return as the type for `sum`. That is, the compiler gives `sum` the same type as the type that would be returned if we were to call `f`.

The way `decltype` handles top-level `const` and references differs subtly from the way `auto` does. When the expression to which we apply `decltype` is a vari-

able, `decltype` returns the type of that variable, including top-level `const` and references:

```
const int ci = 0, &cj = ci;
decltype(ci) x = 0; //  x has type const int
decltype(cj) y = x; //  y has type const int& and is bound to x
decltype(cj) z;     //  error: z is a reference and must be initialized
```

Because `cj` is a reference, `decltype(cj)` is a reference type. Like any other reference, `z` must be initialized.

It is worth noting that `decltype` is the *only* context in which a variable defined as a reference is not treated as a synonym for the object to which it refers.

`decltype` and References

When we apply `decltype` to an expression that is not a variable, we get the type that that expression yields. As we'll see in § 4.1.1 (p. 135), some expressions will cause `decltype` to yield a reference type. Generally speaking, `decltype` returns a reference type for expressions that yield objects that can stand on the left-hand side of the assignment:

```
//  decltype of an expression can be a reference type
int i = 42, *p = &i, &r = i;
decltype(r + 0) b;  //  ok: addition yields an int; b is an (uninitialized) int
decltype(*p) c;     //  error: c is int& and must be initialized
```

Here `r` is a reference, so `decltype(r)` is a reference type. If we want the type to which `r` refers, we can use `r` in an expression, such as `r + 0`, which is an expression that yields a value that has a nonreference type.

On the other hand, the dereference operator is an example of an expression for which `decltype` returns a reference. As we've seen, when we dereference a pointer, we get the object to which the pointer points. Moreover, we can assign to that object. Thus, the type deduced by `decltype(*p)` is `int&`, not plain `int`.

Another important difference between `decltype` and `auto` is that the deduction done by `decltype` *depends on the form of its given expression*. What can be confusing is that enclosing the name of a variable in parentheses affects the type returned by `decltype`. When we apply `decltype` to a variable without any parentheses, we get the type of that variable. If we wrap the variable's name in one or more sets of parentheses, the compiler will evaluate the operand as an expression. A variable is an expression that can be the left-hand side of an assignment. As a result, `decltype` on such an expression yields a reference:

```
//  decltype of a parenthesized variable is always a reference
decltype((i)) d;    //  error: d is int& and must be initialized
decltype(i) e;      //  ok: e is an (uninitialized) int
```

Remember that `decltype((`*variable*`))` (note, double parentheses) is always a reference type, but `decltype(`*variable*`)` is a reference type only if *variable* is a reference.

Exercises Section 2.5.3

Exercise 2.36: In the following code, determine the type of each variable and the value each variable has when the code finishes:

```
int a = 3, b = 4;
decltype(a) c = a;
decltype((b)) d = a;
++c;
++d;
```

Exercise 2.37: Assignment is an example of an expression that yields a reference type. The type is a reference to the type of the left-hand operand. That is, if `i` is an `int`, then the type of the expression `i = x` is `int&`. Using that knowledge, determine the type and value of each variable in this code:

```
int a = 3, b = 4;
decltype(a) c = a;
decltype(a = b) d = a;
```

Exercise 2.38: Describe the differences in type deduction between `decltype` and `auto`. Give an example of an expression where `auto` and `decltype` will deduce the same type and an example where they will deduce differing types.

2.6 Defining Our Own Data Structures

At the most basic level, a data structure is a way to group together related data elements and a strategy for using those data. As one example, our `Sales_item` class groups an ISBN, a count of how many copies of that book had been sold, and the revenue associated with those sales. It also provides a set of operations such as the `isbn` function and the `>>`, `<<`, `+`, and `+=` operators.

In C++ we define our own data types by defining a class. The library types `string`, `istream`, and `ostream` are all defined as classes, as is the `Sales_item` type we used in Chapter 1. C++ support for classes is extensive—in fact, Parts III and IV are largely devoted to describing class-related features. Even though the `Sales_item` class is pretty simple, we won't be able to fully define that class until we learn how to write our own operators in Chapter 14.

2.6.1 Defining the `Sales_data` Type

Although we can't yet write our `Sales_item` class, we can write a more concrete class that groups the same data elements. Our strategy for using this class is that users will be able to access the data elements directly and must implement needed operations for themselves.

Because our data structure does not support any operations, we'll name our version `Sales_data` to distinguish it from `Sales_item`. We'll define our class as follows:

```
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
```

Our class begins with the keyword **`struct`**, followed by the name of the class and a (possibly empty) class body. The class body is surrounded by curly braces and forms a new scope (§ 2.2.4, p. 48). The names defined inside the class must be unique within the class but can reuse names defined outside the class.

The close curly that ends the class body must be followed by a semicolon. The semicolon is needed because we can define variables after the class body:

```
struct Sales_data { /* ... */ } accum, trans, *salesptr;
// equivalent, but better way to define these objects
struct Sales_data { /* ... */ };
Sales_data accum, trans, *salesptr;
```

The semicolon marks the end of the (usually empty) list of declarators. Ordinarily, it is a bad idea to define an object as part of a class definition. Doing so obscures the code by combining the definitions of two different entities—the class and a variable—in a single statement.

It is a common mistake among new programmers to forget the semicolon at the end of a class definition.

Class Data Members

The class body defines the **members** of the class. Our class has only **data members**. The data members of a class define the contents of the objects of that class type. Each object has its own copy of the class data members. Modifying the data members of one object does not change the data in any other `Sales_data` object.

We define data members the same way that we define normal variables: We specify a base type followed by a list of one or more declarators. Our class has three data members: a member of type `string` named `bookNo`, an `unsigned` member named `units_sold`, and a member of type `double` named `revenue`. Each `Sales_data` object will have these three data members.

Under the new standard, we can supply an **in-class initializer** for a data member. When we create objects, the in-class initializers will be used to initialize the data members. Members without an initializer are default initialized (§ 2.2.1, p. 43). Thus, when we define `Sales_data` objects, `units_sold` and `revenue` will be initialized to 0, and `bookNo` will be initialized to the empty string.

C++ 11

In-class initializers are restricted as to the form (§ 2.2.1, p. 43) we can use: They must either be enclosed inside curly braces or follow an = sign. We may not specify an in-class initializer inside parentheses.

In § 7.2 (p. 268), we'll see that C++ has a second keyword, `class`, that can be used to define our own data structures. We'll explain in that section why we use `struct` here. Until we cover additional class-related features in Chapter 7, you should use `struct` to define your own data structures.

Exercises Section 2.6.1

Exercise 2.39: Compile the following program to see what happens when you forget the semicolon after a class definition. Remember the message for future reference.

```
struct Foo { /* empty */ } // Note: no semicolon
int main()
{
    return 0;
}
```

Exercise 2.40: Write your own version of the `Sales_data` class.

2.6.2 Using the `Sales_data` Class

Unlike the `Sales_item` class, our `Sales_data` class does not provide any operations. Users of `Sales_data` have to write whatever operations they need. As an example, we'll write a version of the program from § 1.5.2 (p. 23) that printed the sum of two transactions. The input to our program will be transactions such as

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

Each transaction holds an ISBN, the count of how many books were sold, and the price at which each book was sold.

Adding Two `Sales_data` Objects

Because `Sales_data` provides no operations, we will have to write our own code to do the input, output, and addition operations. We'll assume that our `Sales_data` class is defined inside `Sales_data.h`. We'll see how to define this header in § 2.6.3 (p. 76).

Because this program will be longer than any we've written so far, we'll explain it in separate parts. Overall, our program will have the following structure:

```
#include <iostream>
#include <string>
#include "Sales_data.h"
int main()
{
    Sales_data data1, data2;
    // code to read into data1 and data2
    // code to check whether data1 and data2 have the same ISBN
    //      and if so print the sum of data1 and data2
}
```

As in our original program, we begin by including the headers we'll need and define variables to hold the input. Note that unlike the `Sales_item` version, our new program includes the `string` header. We need that header because our code will have to manage the `bookNo` member, which has type `string`.

Reading Data into a `Sales_data` Object

Although we won't describe the library `string` type in detail until Chapters 3 and 10, we need to know only a little bit about `string`s in order to define and use our ISBN member. The `string` type holds a sequence of characters. Its operations include the `>>`, `<<`, and `==` operators to read, write, and compare `string`s, respectively. With this knowledge we can write the code to read the first transaction:

```
double price = 0;   // price per book, used to calculate total revenue
// read the first transactions: ISBN, number of books sold, price per book
std::cin >> data1.bookNo >> data1.units_sold >> price;
// calculate total revenue from price and units_sold
data1.revenue = data1.units_sold * price;
```

Our transactions contain the price at which each book was sold but our data structure stores the total revenue. We'll read the transaction data into a `double` named `price`, from which we'll calculate the `revenue` member. The input statement

```
std::cin >> data1.bookNo >> data1.units_sold >> price;
```

uses the dot operator (§ 1.5.2, p. 23) to read into the `bookNo` and `units_sold` members of the object named `data1`.

The last statement assigns the product of `data1.units_sold` and `price` into the `revenue` member of `data1`.

Our program will next repeat the same code to read data into `data2`:

```
// read the second transaction
std::cin >> data2.bookNo >> data2.units_sold >> price;
data2.revenue = data2.units_sold * price;
```

Printing the Sum of Two `Sales_data` Objects

Our other task is to check that the transactions are for the same ISBN. If so, we'll print their sum, otherwise, we'll print an error message:

```
if (data1.bookNo == data2.bookNo) {
    unsigned totalCnt = data1.units_sold + data2.units_sold;
    double totalRevenue = data1.revenue + data2.revenue;
    // print: ISBN, total sold, total revenue, average price per book
    std::cout << data1.bookNo << " " << totalCnt
              << " " << totalRevenue << " ";
    if (totalCnt != 0)
        std::cout << totalRevenue/totalCnt << std::endl;
    else
        std::cout  << "(no sales)" << std::endl;
    return 0;  // indicate success
} else {  // transactions weren't for the same ISBN
    std::cerr << "Data must refer to the same ISBN"
              << std::endl;
    return -1; // indicate failure
}
```

In the first `if` we compare the `bookNo` members of `data1` and `data2`. If those members are the same ISBN, we execute the code inside the curly braces. That code adds the components of our two variables. Because we'll need to print the average price, we start by computing the total of `units_sold` and `revenue` and store those in `totalCnt` and `totalRevenue`, respectively. We print those values. Next we check that there were books sold and, if so, print the computed average price per book. If there were no sales, we print a message noting that fact.

Exercises Section 2.6.2

Exercise 2.41: Use your `Sales_data` class to rewrite the exercises in § 1.5.1 (p. 22), § 1.5.2 (p. 24), and § 1.6 (p. 25). For now, you should define your `Sales_data` class in the same file as your `main` function.

2.6.3 Writing Our Own Header Files

Although as we'll see in § 19.7 (p. 852), we can define a class inside a function, such classes have limited functionality. As a result, classes ordinarily are not defined inside functions. When we define a class outside of a function, there may be only one definition of that class in any given source file. In addition, if we use a class in several different files, the class' definition must be the same in each file.

In order to ensure that the class definition is the same in each file, classes are usually defined in header files. Typically, classes are stored in headers whose name derives from the name of the class. For example, the `string` library type is defined in the `string` header. Similarly, as we've already seen, we will define our `Sales_data` class in a header file named `Sales_data.h`.

Headers (usually) contain entities (such as class definitions and `const` and `constexpr` variables (§ 2.4, p. 60)) that can be defined only once in any given file. However, headers often need to use facilities from other headers. For example, because our `Sales_data` class has a `string` member, `Sales_data.h` must `#include` the `string` header. As we've seen, programs that use `Sales_data` also need to include the `string` header in order to use the `bookNo` member. As a result, programs that use `Sales_data` will include the `string` header twice: once directly and once as a side effect of including `Sales_data.h`. Because a header might be included more than once, we need to write our headers in a way that is safe even if the header is included multiple times.

Whenever a header is updated, the source files that use that header must be recompiled to get the new or changed declarations.

A Brief Introduction to the Preprocessor

The most common technique for making it safe to include a header multiple times relies on the **preprocessor**. The preprocessor—which C++ inherits from C—is a

program that runs before the compiler and changes the source text of our programs. Our programs already rely on one preprocessor facility, `#include`. When the preprocessor sees a `#include`, it replaces the `#include` with the contents of the specified header.

C++ programs also use the preprocessor to define **header guards**. Header guards rely on preprocessor variables (§ 2.3.2, p. 53). Preprocessor variables have one of two possible states: defined or not defined. The **`#define`** directive takes a name and defines that name as a preprocessor variable. There are two other directives that test whether a given preprocessor variable has or has not been defined: **`#ifdef`** is true if the variable has been defined, and **`#ifndef`** is true if the variable has *not* been defined. If the test is true, then everything following the `#ifdef` or `#ifndef` is processed up to the matching **`#endif`**.

We can use these facilities to guard against multiple inclusion as follows:

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
#include <string>
struct Sales_data {
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};
#endif
```

The first time `Sales_data.h` is included, the `#ifndef` test will succeed. The preprocessor will process the lines following `#ifndef` up to the `#endif`. As a result, the preprocessor variable `SALES_DATA_H` will be defined and the contents of `Sales_data.h` will be copied into our program. If we include `Sales_data.h` later on in the same file, the `#ifndef` directive will be false. The lines between it and the `#endif` directive will be ignored.

Preprocessor variable names do not respect C++ scoping rules.

Preprocessor variables, including names of header guards, must be unique throughout the program. Typically we ensure uniqueness by basing the guard's name on the name of a class in the header. To avoid name clashes with other entities in our programs, preprocessor variables usually are written in all uppercase.

Headers should have guards, even if they aren't (yet) included by another header. Header guards are trivial to write, and by habitually defining them you don't need to decide whether they are needed.

Exercises Section 2.6.3

Exercise 2.42: Write your own version of the `Sales_data.h` header and use it to rewrite the exercise from § 2.6.2 (p. 76).

Chapter Summary

Types are fundamental to all programming in C++.

Each type defines the storage requirements and the operations that may be performed on objects of that type. The language provides a set of fundamental built-in types such as `int` and `char`, which are closely tied to their representation on the machine's hardware. Types can be non`const` or `const`; a `const` object must be initialized and, once initialized, its value may not be changed. In addition, we can define compound types, such as pointers or references. A compound type is one that is defined in terms of another type.

The language lets us define our own types by defining classes. The library uses the class facility to provide a set of higher-level abstractions such as the IO and `string` types.

Defined Terms

address Number by which a byte in memory can be found.

alias declaration Defines a synonym for another type: `using` *name* = *type* declares *name* as a synonym for the type *type*.

arithmetic types Built-in types representing boolean values, characters, integers, and floating-point numbers.

array Data structure that holds a collection of unnamed objects that are accessed by an index. Section 3.5 covers arrays in detail.

auto Type specifier that deduces the type of a variable from its initializer.

base type type specifier, possibly qualified by `const`, that precedes the declarators in a declaration. The base type provides the common type on which the declarators in a declaration can build.

bind Associating a name with a given entity so that uses of the name are uses of the underlying entity. For example, a reference is a name that is bound to an object.

byte Smallest addressable unit of memory. On most machines a byte is 8 bits.

class member Part of a class.

compound type A type that is defined in terms of another type.

const Type qualifier used to define objects that may not be changed. `const` objects must be initialized, because there is no way to give them a value after they are defined.

const pointer Pointer that is `const`.

const reference Colloquial synonym for reference to `const`.

constant expression Expression that can be evaluated at compile time.

constexpr Variable that represents a constant expression. § 6.5.2 (p. 239) covers `constexpr` functions.

conversion Process whereby a value of one type is transformed into a value of another type. The language defines conversions among the built-in types.

data member Data elements that constitute an object. Every object of a given class has its own copies of the class' data members. Data members may be initialized when declared inside the class.

declaration Asserts the existence of a variable, function, or type defined elsewhere. Names may not be used until they are defined or declared.

declarator The part of a declaration that includes the name being defined and an optional type modifier.

decltype Type specifier that deduces the type of a variable or an expression.

default initialization How objects are initialized when no explicit initializer is given. How class type objects are initialized is controlled by the class. Objects of built-in type defined at global scope are initialized to 0; those defined at local scope are uninitialized and have undefined values.

definition Allocates storage for a variable of a specified type and optionally initializes the variable. Names may not be used until they are defined or declared.

escape sequence Alternative mechanism for representing characters, particularly for those without printable representations. An escape sequence is a backslash followed by a character, three or fewer octal digits, or an `x` followed by a hexadecimal number.

global scope The scope that is outside all other scopes.

header guard Preprocessor variable used to prevent a header from being included more than once in a single file.

identifier Sequence of characters that make up a name. Identifiers are case-sensitive.

in-class initializer Initializer provided as part of the declaration of a class data member. In-class initializers must follow an `=` symbol or be enclosed inside curly braces.

in scope Name that is visible from the current scope.

initialized A variable given an initial value when it is defined. Variables usually should be initialized.

inner scope Scope that is nested inside another scope.

integral types See arithmetic type.

list initialization Form of initialization that uses curly braces to enclose one or more initializers.

literal A value such as a number, a character, or a string of characters. The value cannot be changed. Literal characters are enclosed in single quotes, literal strings in double quotes.

local scope Colloquial synonym for block scope.

low-level const A `const` that is not top-level. Such `const`s are integral to the type and are never ignored.

member Part of a class.

nonprintable character A character with no visible representation, such as a control character, a backspace, newline, and so on.

null pointer Pointer whose value is 0. A null pointer is valid but does not point to any object.

nullptr Literal constant that denotes the null pointer.

object A region of memory that has a type. A variable is an object that has a name.

outer scope Scope that encloses another scope.

pointer An object that can hold the address of an object, the address one past the end of an object, or zero.

pointer to const Pointer that can hold the address of a `const` object. A pointer to `const` may not be used to change the value of the object to which it points.

preprocessor Program that runs as part of compilation of a C++ program.

preprocessor variable Variable managed by the preprocessor. The preprocessor replaces each preprocessor variable by its value before our program is compiled.

reference An alias for another object.

reference to `const` A reference that may not change the value of the object to which it refers. A reference to `const` may be bound to a `const` object, a nonconst object, or the result of an expression.

scope The portion of a program in which names have meaning. C++ has several levels of scope:

> **global**—names defined outside any other scope
>
> **class**—names defined inside a class
>
> **namespace**—names defined inside a namespace
>
> **block**—names defined inside a block

Scopes nest. Once a name is declared, it is accessible until the end of the scope in which it was declared.

separate compilation Ability to split a program into multiple separate source files.

signed Integer type that holds negative or positive values, including zero.

string Library type representing variable-length sequences of characters.

struct Keyword used to define a class.

temporary Unnamed object created by the compiler while evaluating an expression. A temporary exists until the end of the largest expression that encloses the expression for which it was created.

top-level const The `const` that specifies that an object may not be changed.

type alias A name that is a synonym for another type. Defined through either a `typedef` or an alias declaration.

type checking Term used to describe the process by which the compiler verifies that the way objects of a given type are used is consistent with the definition of that type.

type specifier The name of a type.

typedef Defines an alias for another type. When `typedef` appears in the base type of a declaration, the names defined in the declaration are type names.

undefined Usage for which the language does not specify a meaning. Knowingly or unknowingly relying on undefined behavior is a great source of hard-to-track run-time errors, security problems, and portability problems.

uninitialized Variable defined without an initial value. In general, trying to access the value of an uninitialized variable results in undefined behavior.

unsigned Integer type that holds only values greater than or equal to zero.

variable A named object or reference. In C++, variables must be declared before they are used.

void* Pointer type that can point to any nonconst type. Such pointers may not be dereferenced.

void type Special-purpose type that has no operations and no value. It is not possible to define a variable of type `void`.

word The natural unit of integer computation on a given machine. Usually a word is large enough to hold an address. On a 32-bit machine a word is typically 4 bytes.

`&` operator Address-of operator. Yields the address of the object to which it is applied.

`*` operator Dereference operator. Dereferencing a pointer returns the object to which the pointer points. Assigning to the result of a dereference assigns a new value to the underlying object.

#define Preprocessor directive that defines a preprocessor variable.

#endif Preprocessor directive that ends an `#ifdef` or `#ifndef` region.

#ifdef Preprocessor directive that determines whether a given variable is defined.

#ifndef Preprocessor directive that determines whether a given variable is not defined.

CHAPTER 3
STRINGS, VECTORS, AND ARRAYS

CONTENTS

In addition to the built-in types covered in Chapter 2, C++ defines a rich library of abstract data types. Among the most important library types are `string`, which supports variable-length character strings, and `vector`, which defines variable-size collections. Associated with `string` and `vector` are companion types known as iterators, which are used to access the characters in a `string` or the elements in a `vector`.

The `string` and `vector` types defined by the library are abstractions of the more primitive built-in array type. This chapter covers arrays and introduces the library `vector` and `string` types.

The built-in types that we covered in Chapter 2 are defined directly by the C++ language. These types represent facilities present in most computer hardware, such as numbers or characters. The standard library defines a number of additional types of a higher-level nature that computer hardware usually does not implement directly.

In this chapter, we'll introduce two of the most important library types: `string` and `vector`. A `string` is a variable-length sequence of characters. A `vector` holds a variable-length sequence of objects of a given type. We'll also cover the built-in array type. Like other built-in types, arrays represent facilities of the hardware. As a result, arrays are less convenient to use than the library `string` and `vector` types.

Before beginning our exploration of the library types, we'll look at a mechanism for simplifying access to the names defined in the library.

3.1 Namespace `using` Declarations

Up to now, our programs have explicitly indicated that each library name we use is in the `std` namespace. For example, to read from the standard input, we write `std::cin`. These names use the scope operator (`::`) (§ 1.2, p. 8), which says that the compiler should look in the scope of the left-hand operand for the name of the right-hand operand. Thus, `std::cin` says that we want to use the name `cin` from the namespace `std`.

Referring to library names with this notation can be cumbersome. Fortunately, there are easier ways to use namespace members. The safest way is a **`using` declaration**. § 18.2.2 (p. 793) covers another way to use names from a namespace.

A `using` declaration lets us use a name from a namespace without qualifying the name with a `namespace_name::` prefix. A `using` declaration has the form

```
using namespace::name;
```

Once the `using` declaration has been made, we can access *name* directly:

```
#include <iostream>
// using declaration; when we use the name cin, we get the one from the namespace std
using std::cin;
int main()
{
    int i;
    cin >> i;        // ok: cin is a synonym for std::cin
    cout << i;       // error: no using declaration; we must use the full name
    std::cout << i; // ok: explicitly use cout from namepsace std
    return 0;
}
```

A Separate `using` Declaration Is Required for Each Name

Each `using` declaration introduces a single namespace member. This behavior lets us be specific about which names we're using. As an example, we'll rewrite the program from § 1.2 (p. 6) with `using` declarations for the library names it uses:

```
#include <iostream>
// using declarations for names from the standard library
using std::cin;
using std::cout; using std::endl;
int main()
{
    cout << "Enter two numbers:" << endl;
    int v1, v2;
    cin >> v1 >> v2;
    cout << "The sum of " << v1 << " and " << v2
         << " is " << v1 + v2 << endl;
    return 0;
}
```

The `using` declarations for `cin`, `cout`, and `endl` mean that we can use those names without the `std::` prefix. Recall that C++ programs are free-form, so we can put each `using` declaration on its own line or combine several onto a single line. The important part is that there must be a `using` declaration for each name we use, and each declaration must end in a semicolon.

Headers Should Not Include `using` Declarations

Code inside headers (§ 2.6.3, p. 76) ordinarily should not use `using` declarations. The reason is that the contents of a header are copied into the including program's text. If a header has a `using` declaration, then every program that includes that header gets that same `using` declaration. As a result, a program that didn't intend to use the specified library name might encounter unexpected name conflicts.

A Note to the Reader

From this point on, our examples will assume that `using` declarations have been made for the names we use from the standard library. Thus, we will refer to `cin`, not `std::cin`, in the text and in code examples.

Moreover, to keep the code examples short, we won't show the `using` declarations, nor will we show the necessary `#include` directives. Table A.1 (p. 866) in Appendix A lists the names and corresponding headers for standard library names we use in this Primer.

Readers should be aware that they must add appropriate `#include` and `using` declarations to our examples before compiling them.

Exercises Section 3.1

Exercise 3.1: Rewrite the exercises from § 1.4.1 (p. 13) and § 2.6.2 (p. 76) with appropriate `using` declarations.

3.2 Library `string` Type

A **`string`** is a variable-length sequence of characters. To use the `string` type, we must include the `string` header. Because it is part of the library, `string` is defined in the `std` namespace. Our examples assume the following code:

```
#include <string>
using std::string;
```

This section describes the most common `string` operations; § 9.5 (p. 360) will cover additional operations.

In addition to specifying the operations that the library types provide, the standard also imposes efficiency requirements on implementors. As a result, library types are efficient enough for general use.

3.2.1 Defining and Initializing `strings`

Each class defines how objects of its type can be initialized. A class may define many different ways to initialize objects of its type. Each way must be distinguished from the others either by the number of initializers that we supply, or by the types of those initializers. Table 3.1 lists the most common ways to initialize `string`s. Some examples:

```
string s1;              //  default initialization; s1 is the empty string
string s2 = s1;         //  s2 is a copy of s1
string s3 = "hiya";     //  s3 is a copy of the string literal
string s4(10, 'c');     //  s4 is cccccccccc
```

We can default initialize a `string` (§ 2.2.1, p. 44), which creates an empty `string`; that is, a `string` with no characters. When we supply a string literal (§ 2.1.3, p. 39), the characters from that literal—up to but not including the null character at the end of the literal—are copied into the newly created `string`. When we supply a count and a character, the `string` contains that many copies of the given character.

Direct and Copy Forms of Initialization

In § 2.2.1 (p. 43) we saw that C++ has several different forms of initialization. Using `string`s, we can start to understand how these forms differ from one another. When we initialize a variable using =, we are asking the compiler to **copy initialize** the object by copying the initializer on the right-hand side into the object being created. Otherwise, when we omit the =, we use **direct initialization**.

When we have a single initializer, we can use either the direct or copy form of initialization. When we initialize a variable from more than one value, such as in the initialization of `s4` above, we must use the direct form of initialization:

```
string s5 = "hiya";   //  copy initialization
string s6("hiya");    //  direct initialization
string s7(10, 'c');   //  direct initialization; s7 is cccccccccc
```

When we want to use several values, we can indirectly use the copy form of initialization by explicitly creating a (temporary) object to copy:

```
string s8 = string(10, 'c'); // copy initialization; s8 is ccccccccc
```

The initializer of `s8`—`string(10, 'c')`—creates a `string` of the given size and character value and then copies that value into `s8`. It is as if we had written

```
string temp(10, 'c'); // temp is ccccccccc
string s8 = temp;     // copy temp into s8
```

Although the code used to initialize `s8` is legal, it is less readable and offers no compensating advantage over the way we initialized `s7`.

Table 3.1: Ways to Initialize a `string`

`string s1`	Default initialization; `s1` is the empty string.
`string s2(s1)`	`s2` is a copy of `s1`.
`string s2 = s1`	Equivalent to `s2(s1)`, `s2` is a copy of `s1`.
`string s3("value")`	`s3` is a copy of the string literal, not including the null.
`string s3 = "value"`	Equivalent to `s3("value")`, `s3` is a copy of the string literal.
`string s4(n, 'c')`	Initialize `s4` with `n` copies of the character `'c'`.

3.2.2 Operations on `strings`

Along with defining how objects are created and initialized, a class also defines the operations that objects of the class type can perform. A class can define operations that are called by name, such as the `isbn` function of our `Sales_item` class (§ 1.5.2, p. 23). A class also can define what various operator symbols, such as `<<` or `+`, mean when applied to objects of the class' type. Table 3.2 (overleaf) lists the most common `string` operations.

Reading and Writing `strings`

As we saw in Chapter 1, we use the `iostream` library to read and write values of built-in types such as `int`, `double`, and so on. We use the same IO operators to read and write `string`s:

```
// Note: #include and using declarations must be added to compile this code
int main()
{
    string s;           // empty string
    cin >> s;           // read a whitespace-separated string into s
    cout << s << endl; // write s to the output
    return 0;
}
```

Table 3.2: `string` Operations

`os << s`	Writes `s` onto output stream `os`. Returns `os`.
`is >> s`	Reads whitespace-separated string from `is` into `s`. Returns `is`.
`getline(is, s)`	Reads a line of input from `is` into `s`. Returns `is`.
`s.empty()`	Returns `true` if `s` is empty; otherwise returns `false`.
`s.size()`	Returns the number of characters in `s`.
`s[n]`	Returns a reference to the `char` at position `n` in `s`; positions start at 0.
`s1 + s2`	Returns a `string` that is the concatenation of `s1` and `s2`.
`s1 = s2`	Replaces characters in `s1` with a copy of `s2`.
`s1 == s2`	The `strings` `s1` and `s2` are equal if they contain the same characters.
`s1 != s2`	Equality is case-sensitive.
`<, <=, >, >=`	Comparisons are case-sensitive and use dictionary ordering.

This program begins by defining an empty `string` named `s`. The next line reads the standard input, storing what is read in `s`. The `string` input operator reads and discards any leading whitespace (e.g., spaces, newlines, tabs). It then reads characters until the next whitespace character is encountered.

So, if the input to this program is **`Hello World!`** (note leading and trailing spaces), then the output will be **`Hello`** with no extra spaces.

Like the input and output operations on the built-in types, the `string` operators return their left-hand operand as their result. Thus, we can chain together multiple reads or writes:

```
string s1, s2;
cin >> s1 >> s2; // read first input into s1, second into s2
cout << s1 << s2 << endl; // write both strings
```

If we give this version of the program the same input, **`Hello World!`**, our output would be "**`HelloWorld!`**"

Reading an Unknown Number of `strings`

In § 1.4.3 (p. 14) we wrote a program that read an unknown number of `int` values. We can write a similar program that reads `strings` instead:

```
int main()
{
    string word;
    while (cin >> word)          // read until end-of-file
        cout << word << endl; // write each word followed by a new line
    return 0;
}
```

In this program, we read into a `string`, not an `int`. Otherwise, the `while` condition executes similarly to the one in our previous program. The condition tests the stream after the read completes. If the stream is valid—it hasn't hit end-of-file

or encountered an invalid input—then the body of the while is executed. The body prints the value we read on the standard output. Once we hit end-of-file (or invalid input), we fall out of the while.

Using getline to Read an Entire Line

Sometimes we do not want to ignore the whitespace in our input. In such cases, we can use the **getline** function instead of the >> operator. The getline function takes an input stream and a string. This function reads the given stream up to and including the first newline and stores what it read—*not including* the newline—in its string argument. After getline sees a newline, even if it is the first character in the input, it stops reading and returns. If the first character in the input is a newline, then the resulting string is the empty string.

Like the input operator, getline returns its istream argument. As a result, we can use getline as a condition just as we can use the input operator as a condition (§ 1.4.3, p. 14). For example, we can rewrite the previous program that wrote one word per line to write a line at a time instead:

```
int main()
{
    string line;
    // read input a line at a time until end-of-file
    while (getline(cin, line))
        cout << line << endl;
    return 0;
}
```

Because line does not contain a newline, we must write our own. As usual, we use endl to end the current line and flush the buffer.

The newline that causes getline to return is discarded; the newline is *not* stored in the string.

The string empty and size Operations

The **empty** function does what one would expect: It returns a bool (§ 2.1, p. 32) indicating whether the string is empty. Like the isbn member of Sales_item (§ 1.5.2, p. 23), empty is a member function of string. To call this function, we use the dot operator to specify the object on which we want to run the empty function.

We can revise the previous program to only print lines that are not empty:

```
// read input a line at a time and discard blank lines
while (getline(cin, line))
    if (!line.empty())
        cout << line << endl;
```

The condition uses the logical NOT operator (the **! operator**). This operator returns the inverse of the bool value of its operand. In this case, the condition is true if str is not empty.

The **size** member returns the length of a string (i.e., the number of characters in it). We can use size to print only lines longer than 80 characters:

```
string line;
// read input a line at a time and print lines that are longer than 80 characters
while (getline(cin, line))
    if (line.size() > 80)
        cout << line << endl;
```

The string::size_type Type

It might be logical to expect that size returns an int or, thinking back to § 2.1.1 (p. 34), an unsigned. Instead, size returns a string::size_type value. This type requires a bit of explanation.

The string class—and most other library types—defines several companion types. These companion types make it possible to use the library types in a machine-independent manner. The type **size_type** is one of these companion types. To use the size_type defined by string, we use the scope operator to say that the name size_type is defined in the string class.

Although we don't know the precise type of string::size_type, we do know that it is an unsigned type (§ 2.1.1, p. 32) big enough to hold the size of any string. Any variable used to store the result from the string size operation should be of type string::size_type.

C++ 11

Admittedly, it can be tedious to type string::size_type. Under the new standard, we can ask the compiler to provide the appropriate type by using auto or decltype (§ 2.5.2, p. 68):

```
auto len = line.size(); // len has type string::size_type
```

Because size returns an unsigned type, it is essential to remember that expressions that mix signed and unsigned data can have surprising results (§ 2.1.2, p. 36). For example, if n is an int that holds a negative value, then s.size() < n will almost surely evaluate as true. It yields true because the negative value in n will convert to a large unsigned value.

You can avoid problems due to conversion between unsigned and int by not using ints in expressions that use size().

Comparing strings

The string class defines several operators that compare strings. These operators work by comparing the characters of the strings. The comparisons are case-sensitive—upper- and lowercase versions of a letter are different characters.

The equality operators (== and !=) test whether two strings are equal or unequal, respectively. Two strings are equal if they are the same length and contain the same characters. The relational operators <, <=, >, >= test whether one string is less than, less than or equal to, greater than, or greater than or equal to another. These operators use the same strategy as a (case-sensitive) dictionary:

1. If two strings have different lengths and if every character in the shorter string is equal to the corresponding character of the longer string, then the shorter string is less than the longer one.
2. If any characters at corresponding positions in the two strings differ, then the result of the string comparison is the result of comparing the first character at which the strings differ.

As an example, consider the following strings:

```
string str = "Hello";
string phrase = "Hello World";
string slang  = "Hiya";
```

Using rule 1, we see that str is less than phrase. By applying rule 2, we see that slang is greater than both str and phrase.

Assignment for strings

In general, the library types strive to make it as easy to use a library type as it is to use a built-in type. To this end, most of the library types support assignment. In the case of strings, we can assign one string object to another:

```
string st1(10, 'c'), st2; // st1 is cccccccccc; st2 is an empty string
st1 = st2; // assignment: replace contents of st1 with a copy of st2
           // both st1 and st2 are now the empty string
```

Adding Two strings

Adding two strings yields a new string that is the concatenation of the left-hand followed by the right-hand operand. That is, when we use the plus operator (+) on strings, the result is a new string whose characters are a copy of those in the left-hand operand followed by those from the right-hand operand. The compound assignment operator (+=) (§ 1.4.1, p. 12) appends the right-hand operand to the left-hand string:

```
string s1  = "hello, ", s2 = "world\n";
string s3 = s1 + s2;   // s3 is hello, world\n
s1 += s2;   // equivalent to s1 = s1 + s2
```

Adding Literals and strings

As we saw in § 2.1.2 (p. 35), we can use one type where another type is expected if there is a conversion from the given type to the expected type. The string library lets us convert both character literals and character string literals (§ 2.1.3, p. 39) to strings. Because we can use these literals where a string is expected, we can rewrite the previous program as follows:

```
string s1 = "hello", s2 = "world";  // no punctuation in s1 or s2
string s3 = s1 + ", " + s2 + '\n';
```

When we mix strings and string or character literals, at least one operand to each + operator must be of string type:

```
string s4 = s1 + ", ";            // ok: adding a string and a literal
string s5 = "hello" + ", ";       // error: no string operand
string s6 = s1 + ", " + "world"; // ok: each + has a string operand
string s7 = "hello" + ", " + s2; // error: can't add string literals
```

The initializations of `s4` and `s5` involve only a single operation each, so it is easy to see whether the initialization is legal. The initialization of `s6` may appear surprising, but it works in much the same way as when we chain together input or output expressions (§ 1.2, p. 7). This initialization groups as

```
string s6 = (s1 + ", ") + "world";
```

The subexpression `s1 + ", "` returns a `string`, which forms the left-hand operand of the second + operator. It is as if we had written

```
string tmp = s1 + ", ";   // ok: + has a string operand
s6 = tmp + "world";       // ok: + has a string operand
```

On the other hand, the initialization of `s7` is illegal, which we can see if we parenthesize the expression:

```
string s7 = ("hello" + ", ") + s2;   // error: can't add string literals
```

Now it should be easy to see that the first subexpression adds two string literals. There is no way to do so, and so the statement is in error.

For historical reasons, and for compatibility with C, string literals are *not* standard library `string`s. It is important to remember that these types differ when you use string literals and library `string`s.

Exercises Section 3.2.2

Exercise 3.2: Write a program to read the standard input a line at a time. Modify your program to read a word at a time.

Exercise 3.3: Explain how whitespace characters are handled in the `string` input operator and in the `getline` function.

Exercise 3.4: Write a program to read two `string`s and report whether the `string`s are equal. If not, report which of the two is larger. Now, change the program to report whether the `string`s have the same length, and if not, report which is longer.

Exercise 3.5: Write a program to read `string`s from the standard input, concatenating what is read into one large `string`. Print the concatenated `string`. Next, change the program to separate adjacent input `string`s by a space.

3.2.3 Dealing with the Characters in a `string`

Often we need to deal with the individual characters in a `string`. We might want to check to see whether a `string` contains any whitespace, or to change the characters to lowercase, or to see whether a given character is present, and so on.

One part of this kind of processing involves how we gain access to the characters themselves. Sometimes we need to process every character. Other times we need to process only a specific character, or we can stop processing once some condition is met. It turns out that the best way to deal with these cases involves different language and library facilities.

The other part of processing characters is knowing and/or changing the characteristics of a character. This part of the job is handled by a set of library functions, described in Table 3.3 (overleaf). These functions are defined in the `cctype` header.

ADVICE: USE THE C++ VERSIONS OF C LIBRARY HEADERS

In addition to facilities defined specifically for C++, the C++ library incorporates the C library. Headers in C have names of the form *name*`.h`. The C++ versions of these headers are named `c`*name*—they remove the `.h` suffix and precede the *name* with the letter `c`. The `c` indicates that the header is part of the C library.

Hence, `cctype` has the same contents as `ctype.h`, but in a form that is appropriate for C++ programs. In particular, the names defined in the `c`*name* headers are defined inside the `std` namespace, whereas those defined in the `.h` versions are not.

Ordinarily, C++ programs should use the `c`*name* versions of headers and not the *name*`.h` versions. That way names from the standard library are consistently found in the `std` namespace. Using the `.h` headers puts the burden on the programmer to remember which library names are inherited from C and which are unique to C++.

Processing Every Character? Use Range-Based `for`

If we want to do something to every character in a `string`, by far the best approach is to use a statement introduced by the new standard: the **range `for`** statement. This statement iterates through the elements in a given sequence and performs some operation on each value in that sequence. The syntactic form is

C++ 11

```
for (declaration : expression)
    statement
```

where *expression* is an object of a type that represents a sequence, and *declaration* defines the variable that we'll use to access the underlying elements in the sequence. On each iteration, the variable in *declaration* is initialized from the value of the next element in *expression*.

A `string` represents a sequence of characters, so we can use a `string` as the *expression* in a range `for`. As a simple example, we can use a range `for` to print each character from a `string` on its own line of output:

```
string str("some string");
// print the characters in str one character to a line
for (auto c : str)       // for every char in str
    cout << c << endl;   // print the current character followed by a newline
```

The `for` loop associates the variable `c` with `str`. We define the loop control variable the same way we do any other variable. In this case, we use `auto` (§ 2.5.2,

p. 68) to let the compiler determine the type of `c`, which in this case will be `char`. On each iteration, the next character in `str` will be copied into `c`. Thus, we can read this loop as saying, "For every character `c` in the `string` `str`," do something. The "something" in this case is to print the character followed by a newline.

As a somewhat more complicated example, we'll use a range `for` and the `ispunct` function to count the number of punctuation characters in a `string`:

```
string s("Hello World!!!");
// punct_cnt has the same type that s.size returns; see § 2.5.3 (p. 70)
decltype(s.size()) punct_cnt = 0;
// count the number of punctuation characters in s
for (auto c : s)           // for every char in s
    if (ispunct(c))        // if the character is punctuation
        ++punct_cnt;       // increment the punctuation counter
cout << punct_cnt
     << " punctuation characters in " << s << endl;
```

The output of this program is

```
3 punctuation characters in Hello World!!!
```

Here we use `decltype` (§ 2.5.3, p. 70) to declare our counter, `punct_cnt`. Its type is the type returned by calling `s.size`, which is `string::size_type`. We use a range `for` to process each character in the `string`. This time we check whether each character is punctuation. If so, we use the increment operator (§ 1.4.1, p. 12) to add `1` to the counter. When the range `for` completes, we print the result.

Table 3.3: `cctype` Functions

`isalnum(c)`	`true` if `c` is a letter or a digit.
`isalpha(c)`	`true` if `c` is a letter.
`iscntrl(c)`	`true` if `c` is a control character.
`isdigit(c)`	`true` if `c` is a digit.
`isgraph(c)`	`true` if `c` is not a space but is printable.
`islower(c)`	`true` if `c` is a lowercase letter.
`isprint(c)`	`true` if `c` is a printable character (i.e., a space or a character that has a visible representation).
`ispunct(c)`	`true` if `c` is a punctuation character (i.e., a character that is not a control character, a digit, a letter, or a printable whitespace).
`isspace(c)`	`true` if `c` is whitespace (i.e., a space, tab, vertical tab, return, newline, or formfeed).
`isupper(c)`	`true` if `c` is an uppercase letter.
`isxdigit(c)`	`true` if `c` is a hexadecimal digit.
`tolower(c)`	If `c` is an uppercase letter, returns its lowercase equivalent; otherwise returns `c` unchanged.
`toupper(c)`	If `c` is a lowercase letter, returns its uppercase equivalent; otherwise returns `c` unchanged.

Using a Range for to Change the Characters in a string

If we want to change the value of the characters in a string, we must define the loop variable as a reference type (§ 2.3.1, p. 50). Remember that a reference is just another name for a given object. When we use a reference as our control variable, that variable is bound to each element in the sequence in turn. Using the reference, we can change the character to which the reference is bound.

Suppose that instead of counting punctuation, we wanted to convert a string to all uppercase letters. To do so we can use the library toupper function, which takes a character and returns the uppercase version of that character. To convert the whole string we need to call toupper on each character and put the result back in that character:

```
string s("Hello World!!!");
// convert s to uppercase
for (auto &c : s)    // for every char in s (note: c is a reference)
    c = toupper(c); //  c is a reference, so the assignment changes the char in s
cout << s << endl;
```

The output of this code is

```
HELLO WORLD!!!
```

On each iteration, c refers to the next character in s. When we assign to c, we are changing the underlying character in s. So, when we execute

```
c = toupper(c); //  c is a reference, so the assignment changes the char in s
```

we're changing the value of the character to which c is bound. When this loop completes, all the characters in str will be uppercase.

Processing Only Some Characters?

A range for works well when we need to process every character. However, sometimes we need to access only a single character or to access characters until some condition is reached. For example, we might want to capitalize only the first character or only the first word in a string.

There are two ways to access individual characters in a string: We can use a subscript or an iterator. We'll have more to say about iterators in § 3.4 (p. 106) and in Chapter 9.

The subscript operator (the **[] operator**) takes a string::size_type (§ 3.2.2, p. 88) value that denotes the position of the character we want to access. The operator returns a reference to the character at the given position.

Subscripts for strings start at zero; if s is a string with at least two characters, then s[0] is the first character, s[1] is the second, and the last character is in s[s.size() - 1].

The values we use to subscript a string must be >= 0 and < size().
The result of using an index outside this range is undefined.
By implication, subscripting an empty string is undefined.

The value in the subscript is referred to as "a subscript" or "an **index**." The index we supply can be any expression that yields an integral value. However, if our index has a signed type, its value will be converted to the unsigned type that `string::size_type` represents (§ 2.1.2, p. 36).

The following example uses the subscript operator to print the first character in a `string`:

```
if (!s.empty())              // make sure there's a character to print
    cout << s[0] << endl;    // print the first character in s
```

Before accessing the character, we check that `s` is not empty. Any time we use a subscript, we must ensure that there is a value at the given location. If `s` is empty, then `s[0]` is undefined.

So long as the `string` is not `const` (§ 2.4, p. 59), we can assign a new value to the character that the subscript operator returns. For example, we can capitalize the first letter as follows:

```
string s("some string");
if (!s.empty())               // make sure there's a character in s[0]
    s[0] = toupper(s[0]);     // assign a new value to the first character in s
```

The output of this program is

```
Some string
```

Using a Subscript for Iteration

As a another example, we'll change the first word in `s` to all uppercase:

```
// process characters in s until we run out of characters or we hit a whitespace
for (decltype(s.size()) index = 0;
     index != s.size() && !isspace(s[index]); ++index)
        s[index] = toupper(s[index]); // capitalize the current character
```

This program generates

```
SOME string
```

Our `for` loop (§ 1.4.2, p. 13) uses `index` to subscript `s`. We use `decltype` to give `index` the appropriate type. We initialize `index` to `0` so that the first iteration will start on the first character in `s`. On each iteration we increment `index` to look at the next character in `s`. In the body of the loop we capitalize the current letter.

The new part in this loop is the condition in the `for`. That condition uses the logical AND operator (the **`&&` operator**). This operator yields `true` if both operands are `true` and `false` otherwise. The important part about this operator is that we are guaranteed that it evaluates its right-hand operand *only* if the left-hand operand is `true`. In this case, we are guaranteed that we will not subscript `s` unless we know that `index` is in range. That is, `s[index]` is executed only if `index` is not equal to `s.size()`. Because `index` is never incremented beyond the value of `s.size()`, we know that `index` will always be less than `s.size()`.

CAUTION: SUBSCRIPTS ARE UNCHECKED

When we use a subscript, we must ensure that the subscript is in range. That is, the subscript must be >= 0 and < the size() of the string. One way to simplify code that uses subscripts is *always* to use a variable of type string::size_type as the subscript. Because that type is unsigned, we ensure that the subscript cannot be less than zero. When we use a size_type value as the subscript, we need to check only that our subscript is less than value returned by size().

The library is not required to check the value of an subscript. The result of using an out-of-range subscript is undefined.

Using a Subscript for Random Access

In the previous example we advanced our subscript one position at a time to capitalize each character in sequence. We can also calculate an subscript and directly fetch the indicated character. There is no need to access characters in sequence.

As an example, let's assume we have a number between 0 and 15 and we want to generate the hexadecimal representation of that number. We can do so using a `string` that is initialized to hold the 16 hexadecimal "digits":

```
const string hexdigits = "0123456789ABCDEF";  // possible hex digits
cout << "Enter a series of numbers between 0 and 15"
     << " separated by spaces.  Hit ENTER when finished: "
     << endl;
string result;         // will hold the resulting hexify'd string
string::size_type n;   // hold numbers from the input
while (cin >> n)
    if (n < hexdigits.size())     // ignore invalid input
        result += hexdigits[n];   // fetch the indicated hex digit
cout << "Your hex number is: " << result << endl;
```

If we give this program the input

```
12 0 5 15 8 15
```

the output will be

```
Your hex number is: C05F8F
```

We start by initializing `hexdigits` to hold the hexadecimal digits `0` through `F`. We make that `string` `const` (§ 2.4, p. 59) because we do not want these values to change. Inside the loop we use the input value `n` to subscript `hexdigits`. The value of `hexdigits[n]` is the `char` that appears at position `n` in `hexdigits`. For example, if `n` is `15`, then the result is `F`; if it's `12`, the result is `C`; and so on. We append that digit to `result`, which we print once we have read all the input.

Whenever we use a subscript, we should think about how we know that it is in range. In this program, our subscript, `n`, is a `string::size_type`, which as we know is an unsigned type. As a result, we know that `n` is guaranteed to be greater than or equal to `0`. Before we use `n` to subscript `hexdigits`, we verify that it is less than the `size` of `hexdigits`.

Exercises Section 3.2.3

Exercise 3.6: Use a range `for` to change all the characters in a `string` to `X`.

Exercise 3.7: What would happen if you define the loop control variable in the previous exercise as type `char`? Predict the results and then change your program to use a `char` to see if you were right.

Exercise 3.8: Rewrite the program in the first exercise, first using a `while` and again using a traditional `for` loop. Which of the three approaches do you prefer and why?

Exercise 3.9: What does the following program do? Is it valid? If not, why not?

```
string s;
cout << s[0] << endl;
```

Exercise 3.10: Write a program that reads a string of characters including punctuation and writes what was read but with the punctuation removed.

Exercise 3.11: Is the following range `for` legal? If so, what is the type of `c`?

```
const string s = "Keep out!";
for (auto &c : s) { /* ... */ }
```

3.3 Library `vector` Type

A **`vector`** is a collection of objects, all of which have the same type. Every object in the collection has an associated index, which gives access to that object. A `vector` is often referred to as a **container** because it "contains" other objects. We'll have much more to say about containers in Part II.

To use a `vector`, we must include the appropriate header. In our examples, we also assume that an appropriate `using` declaration is made:

```
#include <vector>
using std::vector;
```

A `vector` is a **class template**. C++ has both class and function templates. Writing a template requires a fairly deep understanding of C++. Indeed, we won't see how to create our own templates until Chapter 16! Fortunately, we can use templates without knowing how to write them.

Templates are not themselves functions or classes. Instead, they can be thought of as instructions to the compiler for generating classes or functions. The process that the compiler uses to create classes or functions from templates is called **instantiation**. When we use a template, we specify what kind of class or function we want the compiler to instantiate.

For a class template, we specify which class to instantiate by supplying additional information, the nature of which depends on the template. How we specify the information is always the same: We supply it inside a pair of angle brackets following the template's name.

In the case of `vector`, the additional information we supply is the type of the objects the `vector` will hold:

```
vector<int> ivec;              // ivec holds objects of type int
vector<Sales_item> Sales_vec; // holds Sales_items
vector<vector<string>> file;  // vector whose elements are vectors
```

In this example, the compiler generates three distinct types from the `vector` template: `vector<int>`, `vector<Sales_item>`, and `vector<vector<string>>`.

`vector` is a template, not a type. Types generated from `vector` must include the element type, for example, `vector<int>`.

We can define `vectors` to hold objects of most any type. Because references are not objects (§ 2.3.1, p. 50), we cannot have a `vector` of references. However, we can have `vectors` of most other (nonreference) built-in types and most class types. In particular, we can have `vectors` whose elements are themselves `vectors`.

It is worth noting that earlier versions of C++ used a slightly different syntax to define a `vector` whose elements are themselves `vectors` (or another template type). In the past, we had to supply a space between the closing angle bracket of the outer `vector` and its element type—`vector<vector<int> >` rather than `vector<vector<int>>`.

C++ 11

Some compilers may require the old-style declarations for a `vector` of `vectors`, for example, `vector<vector<int> >`.

3.3.1 Defining and Initializing `vectors`

As with any class type, the `vector` template controls how we define and initialize `vectors`. Table 3.4 (p. 99) lists the most common ways to define `vectors`.

We can default initialize a `vector` (§ 2.2.1, p. 44), which creates an empty `vector` of the specified type:

```
vector<string> svec; // default initialization; svec has no elements
```

It might seem that an empty `vector` would be of little use. However, as we'll see shortly, we can (efficiently) add elements to a `vector` at run time. Indeed, the most common way of using `vectors` is to define an initially empty `vector` to which elements are added as their values become known at run time.

We can also supply initial value(s) for the element(s) when we define a `vector`. For example, we can copy elements from another `vector`. When we copy a `vector`, each element in the new `vector` is a copy of the corresponding element in the original `vector`. The two `vectors` must be the same type:

```
vector<int> ivec;              // initially empty
// give ivec some values
vector<int> ivec2(ivec);       // copy elements of ivec into ivec2
vector<int> ivec3 = ivec;      // copy elements of ivec into ivec3
vector<string> svec(ivec2);    // error: svec holds strings, not ints
```

List Initializing a `vector`

C++ 11

Another way to provide element values, is that under the new standard, we can list initialize (§ 2.2.1, p. 43) a `vector` from a list of zero or more initial element values enclosed in curly braces:

```
vector<string> articles = {"a", "an", "the"};
```

The resulting `vector` has three elements; the first holds the `string` `"a"`, the second holds `"an"`, and the last is `"the"`.

As we've seen, C++ provides several forms of initialization (§ 2.2.1, p. 43). In many, but not all, cases we can use these forms of initialization interchangably. So far, we have seen two examples where the form of initialization matters: when we use the copy initialization form (i.e., when we use =) (§ 3.2.1, p. 84), we can supply only a single initializer; and when we supply an in-class initializer (§ 2.6.1, p. 73), we must either use copy initialization or use curly braces. A third restriction is that we can supply a list of element values only by using list initialization in which the initializers are enclosed in curly braces. We cannot supply a list of initializers using parentheses:

```
vector<string> v1{"a", "an", "the"};  // list initialization
vector<string> v2("a", "an", "the");  // error
```

Creating a Specified Number of Elements

We can also initialize a `vector` from a count and an element value. The count determines how many elements the `vector` will have; the value provides the initial value for each of those elements:

```
vector<int> ivec(10, -1);        // ten int elements, each initialized to -1
vector<string> svec(10, "hi!"); // ten strings; each element is "hi!"
```

Value Initialization

We can usually omit the value and supply only a size. In this case the library creates a **value-initialized** element initializer for us. This library-generated value is used to initialize each element in the container. The value of the element initializer depends on the type of the elements stored in the `vector`.

If the `vector` holds elements of a built-in type, such as `int`, then the element initializer has a value of 0. If the elements are of a class type, such as `string`, then the element initializer is itself default initialized:

```
vector<int> ivec(10);     // ten elements, each initialized to 0
vector<string> svec(10); // ten elements, each an empty string
```

There are two restrictions on this form of initialization: The first restriction is that some classes require that we always supply an explicit initializer (§ 2.2.1, p. 44). If our `vector` holds objects of a type that we cannot default initialize, then we must supply an initial element value; it is not possible to create `vector`s of such types by supplying only a size.

The second restriction is that when we supply an element count without also supplying an initial value, we must use the direct form of initialization:

```
vector<int> vi = 10;      // error: must use direct initialization to supply a size
```

Here we are using 10 to instruct `vector` how to create the `vector`—we want a `vector` with ten value-initialized elements. We are not "copying" 10 into the `vector`. Hence, we cannot use the copy form of initialization. We'll see more about how this restriction works in § 7.5.4 (p. 296).

Table 3.4: Ways to Initialize a `vector`

`vector<T> v1`	`vector` that holds objects of type `T`. Default initialization; `v1` is empty.
`vector<T> v2(v1)`	`v2` has a copy of each element in `v1`.
`vector<T> v2 = v1`	Equivalent to `v2(v1)`, `v2` is a copy of the elements in `v1`.
`vector<T> v3(n, val)`	`v3` has `n` elements with value `val`.
`vector<T> v4(n)`	`v4` has `n` copies of a value-initialized object.
`vector<T> v5{a,b,c...}`	`v5` has as many elements as there are initializers; elements are initialized by corresponding initializers.
`vector<T> v5 = {a,b,c...}`	Equivalent to `v5{a,b,c...}`.

List Initializer or Element Count?

In a few cases, what initialization means depends upon whether we use curly braces or parentheses to pass the initializer(s). For example, when we initialize a `vector<int>` from a single `int` value, that value might represent the `vector`'s size or it might be an element value. Similarly, if we supply exactly two `int` values, those values could be a size and an initial value, or they could be values for a two-element `vector`. We specify which meaning we intend by whether we use curly braces or parentheses:

```
vector<int> v1(10);      // v1 has ten elements with value 0
vector<int> v2{10};      // v2 has one element with value 10
vector<int> v3(10, 1);   // v3 has ten elements with value 1
vector<int> v4{10, 1};   // v4 has two elements with values 10 and 1
```

When we use parentheses, we are saying that the values we supply are to be used to *construct* the object. Thus, `v1` and `v3` use their initializers to determine the `vector`'s size, and its size and element values, respectively.

When we use curly braces, { . . . }, we're saying that, if possible, we want to *list initialize* the object. That is, if there is a way to use the values inside the curly braces as a list of element initializers, the class will do so. Only if it is not possible to list initialize the object will the other ways to initialize the object be considered. The values we supply when we initialize `v2` and `v4` can be used as element values. These objects are list initialized; the resulting `vectors` have one and two elements, respectively.

On the other hand, if we use braces and there is no way to use the initializers to list initialize the object, then those values will be used to construct the object. For

example, to list initialize a `vector` of `string`s, we must supply values that can be used as `string`s. In this case, there is no confusion about whether to list initialize the elements or construct a `vector` of the given size:

```
vector<string> v5{"hi"}; // list initialization: v5 has one element
vector<string> v6("hi"); // error: can't construct a vector from a string literal
vector<string> v7{10};        // v7 has ten default-initialized elements
vector<string> v8{10, "hi"}; // v8 has ten elements with value "hi"
```

Although we used braces on all but one of these definitions, only `v5` is list initialized. In order to list initialize the `vector`, the values inside braces must match the element type. We cannot use an `int` to initialize a `string`, so the initializers for `v7` and `v8` can't be element initializers. If list initialization isn't possible, the compiler looks for other ways to initialize the object from the given values.

Exercises Section 3.3.1

Exercise 3.12: Which, if any, of the following `vector` definitions are in error? For those that are legal, explain what the definition does. For those that are not legal, explain why they are illegal.

```
(a) vector<vector<int>> ivec;
(b) vector<string> svec = ivec;
(c) vector<string> svec(10, "null");
```

Exercise 3.13: How many elements are there in each of the following `vector`s? What are the values of the elements?

```
(a) vector<int> v1;           (b) vector<int> v2(10);
(c) vector<int> v3(10, 42);   (d) vector<int> v4{10};
(e) vector<int> v5{10, 42};   (f) vector<string> v6{10};
(g) vector<string> v7{10, "hi"};
```

3.3.2 Adding Elements to a `vector`

Directly initializing the elements of a `vector` is feasible only if we have a small number of known initial values, if we want to make a copy of another `vector`, or if we want to initialize all the elements to the same value. More commonly, when we create a `vector`, we don't know how many elements we'll need, or we don't know the value of those elements. Even if we do know all the values, if we have a large number of different initial element values, it can be cumbersome to specify them when we create the `vector`.

As one example, if we need a `vector` with values from 0 to 9, we can easily use list initialization. What if we wanted elements from 0 to 99 or 0 to 999? List initialization would be too unwieldy. In such cases, it is better to create an empty `vector` and use a `vector` member named **`push_back`** to add elements at run time. The `push_back` operation takes a value and "pushes" that value as a new last element onto the "back" of the `vector`. For example:

```
vector<int> v2;          // empty vector
for (int i = 0; i != 100; ++i)
    v2.push_back(i);     // append sequential integers to v2
// at end of loop v2 has 100 elements, values 0 . . . 99
```

Even though we know we ultimately will have 100 elements, we define `v2` as empty. Each iteration adds the next sequential integer as a new element in `v2`.

We use the same approach when we want to create a `vector` where we don't know until run time how many elements the `vector` should have. For example, we might read the input, storing the values we read in the `vector`:

```
// read words from the standard input and store them as elements in a vector
string word;
vector<string> text;         // empty vector
while (cin >> word) {
    text.push_back(word);    // append word to text
}
```

Again, we start with an initially empty `vector`. This time, we read and store an unknown number of values in `text`.

KEY CONCEPT: VECTORS GROW EFFICIENTLY

The standard requires that `vector` implementations can efficiently add elements at run time. Because `vectors` grow efficiently, it is often unnecessary—and can result in poorer performance—to define a `vector` of a specific size. The exception to this rule is if *all* the elements actually need the same value. If differing element values are needed, it is usually more efficient to define an empty `vector` and add elements as the values we need become known at run time. Moreover, as we'll see in § 9.4 (p. 355), `vector` offers capabilities to allow us to further enhance run-time performance when we add elements.

Starting with an empty `vector` and adding elements at run time is distinctly different from how we use built-in arrays in C and in most other languages. In particular, if you are accustomed to using C or Java, you might expect that it would be best to define the `vector` at its expected size. In fact, the contrary is usually the case.

Programming Implications of Adding Elements to a `vector`

The fact that we can easily and efficiently add elements to a `vector` greatly simplifies many programming tasks. However, this simplicity imposes a new obligation on our programs: We must ensure that any loops we write are correct even if the loop changes the size of the `vector`.

Other implications that follow from the dynamic nature of `vectors` will become clearer as we learn more about using them. However, there is one implication that is worth noting already: For reasons we'll explore in § 5.4.3 (p. 188), we cannot use a range `for` if the body of the loop adds elements to the `vector`.

The body of a range `for` must not change the size of the sequence over which it is iterating.

Exercises Section 3.3.2

Exercise 3.14: Write a program to read a sequence of `int`s from `cin` and store those values in a `vector`.

Exercise 3.15: Repeat the previous program but read `string`s this time.

3.3.3 Other `vector` Operations

In addition to `push_back`, `vector` provides only a few other operations, most of which are similar to the corresponding operations on `string`s. Table 3.5 lists the most important ones.

We access the elements of a `vector` the same way that we access the characters in a `string`: through their position in the `vector`. For example, we can use a range `for` (§ 3.2.3, p. 91) to process all the elements in a `vector`:

```
vector<int> v{1,2,3,4,5,6,7,8,9};
for (auto &i : v)      // for each element in v (note: i is a reference)
    i *= i;            // square the element value
for (auto i : v)       // for each element in v
    cout << i << " "; // print the element
cout << endl;
```

In the first loop, we define our control variable, `i`, as a reference so that we can use `i` to assign new values to the elements in `v`. We let `auto` deduce the type of `i`. This loop uses a new form of the compound assignment operator (§ 1.4.1, p. 12). As we've seen, `+=` adds the right-hand operand to the left and stores the result in the left-hand operand. The `*=` operator behaves similarly, except that it multiplies the left- and right-hand operands, storing the result in the left-hand one. The second range `for` prints each element.

The `empty` and `size` members behave as do the corresponding `string` members (§ 3.2.2, p. 87): `empty` returns a `bool` indicating whether the `vector` has any elements, and `size` returns the number of elements in the `vector`. The `size` member returns a value of the `size_type` defined by the corresponding `vector` type.

To use `size_type`, we must name the type in which it is defined. A `vector` type *always* includes its element type (§ 3.3, p. 97):

```
vector<int>::size_type // ok
vector::size_type      // error
```

The equality and relational operators have the same behavior as the corresponding `string` operations (§ 3.2.2, p. 88). Two `vector`s are equal if they have the same number of elements and if the corresponding elements all have the same value. The relational operators apply a dictionary ordering: If the `vector`s have differing sizes, but the elements that are in common are equal, then the `vector` with fewer elements is less than the one with more elements. If the elements have

differing values, then the relationship between the vectors is determined by the relationship between the first elements that differ.

We can compare two vectors only if we can compare the elements in those vectors. Some class types, such as string, define the meaning of the equality and relational operators. Others, such as our Sales_item class, do not. The only operations Sales_item supports are those listed in § 1.5.1 (p. 20). Those operations did not include the equality or relational operators. As a result, we cannot compare two vector<Sales_item> objects.

Table 3.5: vector Operations

v.empty()	Returns true if v is empty; otherwise returns false.
v.size()	Returns the number of elements in v.
v.push_back(t)	Adds an element with value t to end of v.
v[n]	Returns a reference to the element at position n in v.
v1 = v2	Replaces the elements in v1 with a copy of the elements in v2.
v1 = {a,b,c...}	Replaces the elements in v1 with a copy of the elements in the comma-separated list.
v1 == v2 v1 != v2	v1 and v2 are equal if they have the same number of elements and each element in v1 is equal to the corresponding element in v2.
<, <=, >, >=	Have their normal meanings using dictionary ordering.

Computing a vector Index

We can fetch a given element using the subscript operator (§ 3.2.3, p. 93). As with strings, subscripts for vector start at 0; the type of a subscript is the corresponding size_type; and—assuming the vector is nonconst—we can write to the element returned by the subscript operator. In addition, as we did in § 3.2.3 (p. 95), we can compute an index and directly fetch the element at that position.

As an example, let's assume that we have a collection of grades that range from 0 through 100. We'd like to count how many grades fall into various clusters of 10. Between zero and 100 there are 101 possible grades. These grades can be represented by 11 clusters: 10 clusters of 10 grades each plus one cluster for the perfect score of 100. The first cluster will count grades of 0 through 9, the second will count grades from 10 through 19, and so on. The final cluster counts how many scores of 100 were achieved.

Clustering the grades this way, if our input is

```
42 65 95 100 39 67 95 76 88 76 83 92 76 93
```

then the output should be

```
0 0 0 1 1 0 2 3 2 4 1
```

which indicates that there were no grades below 30, one grade in the 30s, one in the 40s, none in the 50s, two in the 60s, three in the 70s, two in the 80s, four in the 90s, and one grade of 100.

We'll use a `vector` with 11 elements to hold the counters for each cluster. We can determine the cluster index for a given grade by dividing that grade by 10. When we divide two integers, we get an integer in which the fractional part is truncated. For example, 42/10 is 4, 65/10 is 6 and 100/10 is 10. Once we've computed the cluster index, we can use it to subscript our `vector` and fetch the counter we want to increment:

```
// count the number of grades by clusters of ten: 0--9, 10--19, . .. 90--99, 100
vector<unsigned> scores(11, 0); // 11 buckets, all initially 0
unsigned grade;
while (cin >> grade) {       // read the grades
    if (grade <= 100)        // handle only valid grades
        ++scores[grade/10]; // increment the counter for the current cluster
}
```

We start by defining a `vector` to hold the cluster counts. In this case, we do want each element to have the same value, so we allocate all 11 elements, each of which is initialized to `0`. The `while` condition reads the grades. Inside the loop, we check that the grade we read has a valid value (i.e., that it is less than or equal to 100). Assuming the grade is valid, we increment the appropriate counter for `grade`.

The statement that does the increment is a good example of the kind of terse code characteristic of C++ programs. This expression

```
++scores[grade/10]; // increment the counter for the current cluster
```

is equivalent to

```
auto ind = grade/10; // get the bucket index
scores[ind] = scores[ind] + 1; // increment the count
```

We compute the bucket index by dividing `grade` by 10 and use the result of the division to index `scores`. Subscripting `scores` fetches the appropriate counter for this grade. We increment the value of that element to indicate the occurrence of a score in the given range.

As we've seen, when we use a subscript, we should think about how we know that the indices are in range (§ 3.2.3, p. 95). In this program, we verify that the input is a valid grade in the range between `0` and `100`. Thus, we know that the indices we can compute are between `0` and `10`. These indices are between `0` and `scores.size() - 1`.

Subscripting Does Not Add Elements

Programmers new to C++ sometimes think that subscripting a `vector` adds elements; it does not. The following code intends to add ten elements to `ivec`:

```
vector<int> ivec;   // empty vector
for (decltype(ivec.size()) ix = 0; ix != 10; ++ix)
    ivec[ix] = ix;  // disaster: ivec has no elements
```

However, it is in error: `ivec` is an empty `vector`; there are no elements to subscript! As we've seen, the right way to write this loop is to use `push_back`:

```
for (decltype(ivec.size()) ix = 0; ix != 10; ++ix)
    ivec.push_back(ix);   //  ok: adds a new element with value ix
```

The subscript operator on `vector` (and `string`) fetches an existing element; it does *not* add an element.

CAUTION: SUBSCRIPT ONLY ELEMENTS THAT ARE KNOWN TO EXIST!

It is crucially important to understand that we may use the subscript operator (the `[]` operator) to fetch only elements that actually exist. For example,

```
vector<int> ivec;       //  empty vector
cout << ivec[0];        //  error: ivec has no elements!

vector<int> ivec2(10); //  vector with ten elements
cout << ivec2[10];      //  error: ivec2 has elements 0 . . . 9
```

It is an error to subscript an element that doesn't exist, but it is an error that the compiler is unlikely to detect. Instead, the value we get at run time is undefined.

Attempting to subscript elements that do not exist is, unfortunately, an extremely common and pernicious programming error. So-called *buffer overflow* errors are the result of subscripting elements that don't exist. Such bugs are the most common cause of security problems in PC and other applications.

A good way to ensure that subscripts are in range is to avoid subscripting altogether by using a range `for` whenever possible.

EXERCISES SECTION 3.3.3

Exercise 3.16: Write a program to print the size and contents of the `vectors` from exercise 3.13. Check whether your answers to that exercise were correct. If not, restudy § 3.3.1 (p. 97) until you understand why you were wrong.

Exercise 3.17: Read a sequence of words from `cin` and store the values a `vector`. After you've read all the words, process the `vector` and change each word to uppercase. Print the transformed elements, eight words to a line.

Exercise 3.18: Is the following program legal? If not, how might you fix it?

```
vector<int> ivec;
ivec[0] = 42;
```

Exercise 3.19: List three ways to define a `vector` and give it ten elements, each with the value 42. Indicate whether there is a preferred way to do so and why.

Exercise 3.20: Read a set of integers into a `vector`. Print the sum of each pair of adjacent elements. Change your program so that it prints the sum of the first and last elements, followed by the sum of the second and second-to-last, and so on.

3.4 Introducing Iterators

Although we can use subscripts to access the characters of a `string` or the elements in a `vector`, there is a more general mechanism—known as **iterators**—that we can use for the same purpose. As we'll see in Part II, in addition to `vector`, the library defines several other kinds of containers. All of the library containers have iterators, but only a few of them support the subscript operator. Technically speaking, a `string` is not a container type, but `string` supports many of the container operations. As we've seen `string`, like `vector` has a subscript operator. Like `vectors`, `strings` also have iterators.

Like pointers (§ 2.3.2, p. 52), iterators give us indirect access to an object. In the case of an iterator, that object is an element in a container or a character in a `string`. We can use an iterator to fetch an element and iterators have operations to move from one element to another. As with pointers, an iterator may be valid or invalid. A valid iterator either denotes an element or denotes a position one past the last element in a container. All other iterator values are invalid.

3.4.1 Using Iterators

Unlike pointers, we do not use the address-of operator to obtain an iterator. Instead, types that have iterators have members that return iterators. In particular, these types have members named **`begin`** and **`end`**. The `begin` member returns an iterator that denotes the first element (or first character), if there is one:

```
// the compiler determines the type of b and e; see § 2.5.2 (p. 68)
// b denotes the first element and e denotes one past the last element in v
auto b = v.begin(), e = v.end(); // b and e have the same type
```

The iterator returned by `end` is an iterator positioned "one past the end" of the associated container (or `string`). This iterator denotes a nonexistent element "off the end" of the container. It is used as a marker indicating when we have processed all the elements. The iterator returned by `end` is often referred to as the **off-the-end iterator** or abbreviated as "the `end` iterator." If the container is empty, `begin` returns the same iterator as the one returned by `end`.

If the container is empty, the iterators returned by `begin` and `end` are equal—they are both off-the-end iterators.

In general, we do not know (or care about) the precise type that an iterator has. In this example, we used `auto` to define `b` and `e` (§ 2.5.2, p. 68). As a result, these variables have whatever type is returned by the `begin` and `end` members, respectively. We'll have more to say about those types on page 108.

Iterator Operations

Iterators support only a few operations, which are listed in Table 3.6. We can compare two valid iterators using `==` or `!=`. Iterators are equal if they denote the same element or if they are both off-the-end iterators for the same container. Otherwise, they are unequal.

As with pointers, we can dereference an iterator to obtain the element denoted by an iterator. Also, like pointers, we may dereference only a valid iterator that denotes an element (§ 2.3.2, p. 53). Dereferencing an invalid iterator or an off-the-end iterator has undefined behavior.

As an example, we'll rewrite the program from § 3.2.3 (p. 94) that capitalized the first character of a `string` using an iterator instead of a subscript:

```
string s("some string");
if (s.begin() != s.end()) { //  make sure s is not empty
    auto it = s.begin();    //  it denotes the first character in s
    *it = toupper(*it);     //  make that character uppercase
}
```

As in our original program, we first check that `s` isn't empty. In this case, we do so by comparing the iterators returned by `begin` and `end`. Those iterators are equal if the `string` is empty. If they are unequl, there is at least one character in `s`.

Inside the `if` body, we obtain an iterator to the first character by assigning the iterator returned by `begin` to `it`. We dereference that iterator to pass that character to `toupper`. We also dereference `it` on the left-hand side of the assignment in order to assign the character returned from `toupper` to the first character in `s`. As in our original program, the output of this loop will be:

```
Some string
```

Table 3.6: Standard Container Iterator Operations

`*iter`	Returns a reference to the element denoted by the iterator `iter`.
`iter->mem`	Dereferences `iter` and fetches the member named `mem` from the underlying element. Equivalent to `(*iter).mem`.
`++iter`	Increments `iter` to refer to the next element in the container.
`--iter`	Decrements `iter` to refer to the previous element in the container.
`iter1 == iter2` `iter1 != iter2`	Compares two iterators for equality (inequality). Two iterators are equal if they denote the same element or if they are the off-the-end iterator for the same container.

Moving Iterators from One Element to Another

Iterators use the increment (++) operator (§ 1.4.1, p. 12) to move from one element to the next. Incrementing an iterator is a logically similar operation to incrementing an integer. In the case of integers, the effect is to "add `1`" to the integer's value. In the case of iterators, the effect is to "advance the iterator by one position."

Because the iterator returned from `end` does not denote an element, it may not be incremented or dereferenced.

Using the increment operator, we can rewrite our program that changed the case of the first word in a `string` to use iterators instead:

```
// process characters in s until we run out of characters or we hit a whitespace
for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it)
    *it = toupper(*it); // capitalize the current character
```

This loop, like the one in § 3.2.3 (p. 94), iterates through the characters in `s`, stopping when we encounter a whitespace character. However, this loop accesses these characters using an iterator, not a subscript.

The loop starts by initializing `it` from `s.begin`, meaning that `it` denotes the first character (if any) in `s`. The condition checks whether `it` has reached the `end` of `s`. If not, the condition next dereferences `it` to pass the current character to `isspace` to see whether we're done. At the end of each iteration, we execute `++it` to advance the iterator to access the next character in `s`.

The body of this loop, is the same as the last statement in the previous `if`. We dereference `it` to pass the current character to `toupper` and assign the resulting uppercase letter back into the character denoted by `it`.

KEY CONCEPT: GENERIC PROGRAMMING

Programmers coming to C++ from C or Java might be surprised that we used `!=` rather than `<` in our `for` loops such as the one above and in the one on page 94. C++ programmers use `!=` as a matter of habit. They do so for the same reason that they use iterators rather than subscripts: This coding style applies equally well to various kinds of containers provided by the library.

As we've seen, only a few library types, `vector` and `string` being among them, have the subscript operator. Similarly, all of the library containers have iterators that define the `==` and `!=` operators. Most of those iterators do not have the `<` operator. By routinely using iterators and `!=`, we don't have to worry about the precise type of container we're processing.

Iterator Types

Just as we do not know the precise type of a `vector`'s or `string`'s `size_type` member (§ 3.2.2, p. 88), so too, we generally do not know—and do not need to know—the precise type of an iterator. Instead, as with `size_type`, the library types that have iterators define types named `iterator` and `const_iterator` that represent actual iterator types:

```
vector<int>::iterator it; // it can read and write vector<int> elements
string::iterator it2;     // it2 can read and write characters in a string
vector<int>::const_iterator it3; // it3 can read but not write elements
string::const_iterator it4;      // it4 can read but not write characters
```

A `const_iterator` behaves like a `const` pointer (§ 2.4.2, p. 62). Like a `const` pointer, a `const_iterator` may read but not write the element it denotes; an object of type `iterator` can both read and write. If a `vector` or `string` is `const`, we may use only its `const_iterator` type. With a nonconst `vector` or `string`, we can use either `iterator` or `const_iterator`.

TERMINOLOGY: ITERATORS AND ITERATOR TYPES

The term iterator is used to refer to three different entities. We might mean the *concept* of an iterator, or we might refer to the `iterator` *type* defined by a container, or we might refer to an *object* as an iterator.

What's important to understand is that there is a collection of types that are related conceptually. A type is an iterator if it supports a common set of actions. Those actions let us access an element in a container and let us move from one element to another.

Each container class defines a type named `iterator`; that `iterator` type supports the actions of an (conceptual) iterator.

The `begin` and `end` Operations

The type returned by `begin` and `end` depends on whether the object on which they operator is `const`. If the object is `const`, then `begin` and `end` return a `const_iterator`; if the object is not `const`, they return `iterator`:

```
vector<int> v;
const vector<int> cv;
auto it1 = v.begin();  // it1 has type vector<int>::iterator
auto it2 = cv.begin(); // it2 has type vector<int>::const_iterator
```

Often this default behavior is not what we want. For reasons we'll explain in § 6.2.3 (p. 213), it is usually best to use a `const` type (such as `const_iterator`) when we need to read but do not need to write to an object. To let us ask specifically for the `const_iterator` type, the new standard introduced two new functions named `cbegin` and `cend`:

C++ 11

```
auto it3 = v.cbegin(); // it3 has type vector<int>::const_iterator
```

As do the `begin` and `end` members, these members return iterators to the first and one past the last element in the container. However, regardless of whether the `vector` (or `string`) is `const`, they return a `const_iterator`.

Combining Dereference and Member Access

When we dereference an iterator, we get the object that the iterator denotes. If that object has a class type, we may want to access a member of that object. For example, we might have a `vector` of `string`s and we might need to know whether a given element is empty. Assuming `it` is an iterator into this `vector`, we can check whether the `string` that `it` denotes is empty as follows:

```
(*it).empty()
```

For reasons we'll cover in § 4.1.2 (p. 136), the parentheses in `(*it).empty()` are necessary. The parentheses say to apply the dereference operator to `it` and to apply the dot operator (§ 1.5.2, p. 23) to the result of dereferencing `it`. Without parentheses, the dot operator would apply to `it`, not to the resulting object:

```
(*it).empty() // dereferences it and calls the member empty on the resulting object
*it.empty()   // error: attempts to fetch the member named empty from it
              //        but it is an iterator and has no member named empty
```

The second expression is interpreted as a request to fetch the `empty` member from the object named `it`. However, `it` is an iterator and has no member named `empty`. Hence, the second expression is in error.

To simplify expressions such as this one, the language defines the arrow operator (the **`->` operator**). The arrow operator combines dereference and member access into a single operation. That is, `it->mem` is a synonym for `(*it).mem`.

For example, assume we have a `vector<string>` named `text` that holds the data from a text file. Each element in the `vector` is either a sentence or an empty `string` representing a paragraph break. If we want to print the contents of the first paragraph from `text`, we'd write a loop that iterates through `text` until we encounter an element that is empty:

```
// print each line in text up to the first blank line
for (auto it = text.cbegin();
     it != text.cend() && !it->empty(); ++it)
    cout << *it << endl;
```

We start by initializing `it` to denote the first element in `text`. The loop continues until either we process every element in `text` or we find an element that is empty. So long as there are elements and we haven't seen an empty element, we print the current element. It is worth noting that because the loop reads but does not write to the elements in `text`, we use `cbegin` and `cend` to control the iteration.

Some `vector` Operations Invalidate Iterators

In § 3.3.2 (p. 101) we noted that there are implications of the fact that `vector`s can grow dynamically. We also noted that one such implication is that we cannot add elements to a `vector` inside a range `for` loop. Another implication is that any operation, such as `push_back`, that changes the size of a `vector` potentially invalidates all iterators into that `vector`. We'll explore how iterators become invalid in more detail in § 9.3.6 (p. 353).

For now, it is important to realize that loops that use iterators should not add elements to the container to which the iterators refer.

Exercises Section 3.4.1

Exercise 3.21: Redo the first exercise from § 3.3.3 (p. 105) using iterators.

Exercise 3.22: Revise the loop that printed the first paragraph in `text` to instead change the elements in `text` that correspond to the first paragraph to all uppercase. After you've updated `text`, print its contents.

Exercise 3.23: Write a program to create a `vector` with ten `int` elements. Using an iterator, assign each element a value that is twice its current value. Test your program by printing the `vector`.

3.4.2 Iterator Arithmetic

Incrementing an iterator moves the iterator one element at a time. All the library containers have iterators that support increment. Similarly, we can use `==` and `!=` to compare two valid iterators (§ 3.4, p. 106) into any of the library container types.

Iterators for `string` and `vector` support additional operations that can move an iterator multiple elements at a time. They also support all the relational operators. These operations, which are often referred to as **iterator arithmetic**, are described in Table 3.7.

Table 3.7: Operations Supported by `vector` and `string` Iterators

`iter + n` `iter - n`	Adding (subtracting) an integral value `n` to (from) an iterator yields an iterator that many elements forward (backward) within the container. The resulting iterator must denote elements in, or one past the end of, the same container.
`iter1 += n` `iter1 -= n`	Compound-assignment for iterator addition and subtraction. Assigns to `iter1` the value of adding `n` to, or subtracting `n` from, `iter1`.
`iter1 - iter2`	Subtracting two iterators yields the number that when added to the right-hand iterator yields the left-hand iterator. The iterators must denote elements in, or one past the end of, the same container.
`>, >=, <, <=`	Relational operators on iterators. One iterator is less than another if it refers to an element that appears in the container before the one referred to by the other iterator. The iterators must denote elements in, or one past the end of, the same container.

Arithmetic Operations on Iterators

We can add (or subtract) an integral value and an iterator. Doing so returns an iterator positioned forward (or backward) that many elements. When we add or subtract an integral value and an iterator, the result must denote an element in the same `vector` (or `string`) or denote one past the end of the associated `vector` (or `string`). As an example, we can compute an iterator to the element nearest the middle of a `vector`:

```
// compute an iterator to the element closest to the midpoint of vi
auto mid = vi.begin() + vi.size() / 2;
```

If `vi` has 20 elements, then `vi.size()/2` is `10`. In this case, we'd set `mid` equal to `vi.begin() + 10`. Remembering that subscripts start at 0, this element is the same as `vi[10]`, the element ten past the first.

In addition to comparing two iterators for equality, we can compare `vector` and `string` iterators using the relational operators (`<`, `<=`, `>`, `>=`). The iterators must be valid and must denote elements in (or one past the end of) the same `vector` or `string`. For example, assuming `it` is an iterator into the same `vector` as `mid`, we can check whether `it` denotes an element before or after `mid` as follows:

```
if (it < mid)
    // process elements in the first half of vi
```

We can also subtract two iterators so long as they refer to elements in, or one off the end of, the same `vector` or `string`. The result is the distance between the iterators. By distance we mean the amount by which we'd have to change one iterator to get the other. The result type is a signed integral type named **`difference_type`**. Both `vector` and `string` define `difference_type`. This type is signed, because subtraction might have a negative result.

Using Iterator Arithmetic

A classic algorithm that uses iterator arithmetic is binary search. A binary search looks for a particular value in a sorted sequence. It operates by looking at the element closest to the middle of the sequence. If that element is the one we want, we're done. Otherwise, if that element is smaller than the one we want, we continue our search by looking only at elements after the rejected one. If the middle element is larger than the one we want, we continue by looking only in the first half. We compute a new middle element in the reduced range and continue looking until we either find the element or run out of elements.

We can do a binary search using iterators as follows:

```
// text must be sorted
// beg and end will denote the range we're searching
auto beg = text.begin(), end = text.end();
auto mid = text.begin() + (end - beg)/2; // original midpoint
// while there are still elements to look at and we haven't yet found sought
while (mid != end && *mid != sought) {
    if (sought < *mid)      // is the element we want in the first half?
        end = mid;          // if so, adjust the range to ignore the second half
    else                    // the element we want is in the second half
        beg = mid + 1;      // start looking with the element just after mid
    mid = beg + (end - beg)/2;  // new midpoint
}
```

We start by defining three iterators: `beg` will be the first element in the range, `end` one past the last element, and `mid` the element closest to the middle. We initialize these iterators to denote the entire range in a `vector<string>` named `text`.

Our loop first checks that the range is not empty. If `mid` is equal to the current value of `end`, then we've run out of elements to search. In this case, the condition fails and we exit the `while`. Otherwise, `mid` refers to an element and we check whether `mid` denotes the one we want. If so, we're done and we exit the loop.

If we still have elements to process, the code inside the `while` adjusts the range by moving `end` or `beg`. If the element denoted by `mid` is greater than `sought`, we know that if `sought` is in `text`, it will appear before the element denoted by `mid`. Therefore, we can ignore elements after `mid`, which we do by assigning `mid` to `end`. If `*mid` is smaller than `sought`, the element must be in the range of elements after the one denoted by `mid`. In this case, we adjust the range by making `beg` denote the element just after `mid`. We already know that `mid` is not the one we want, so we can eliminate it from the range.

At the end of the `while`, `mid` will be equal to `end` or it will denote the element for which we are looking. If `mid` equals `end`, then the element was not in `text`.

EXERCISES SECTION 3.4.2

Exercise 3.24: Redo the last exercise from § 3.3.3 (p. 105) using iterators.

Exercise 3.25: Rewrite the grade clustering program from § 3.3.3 (p. 104) using iterators instead of subscripts.

Exercise 3.26: In the binary search program on page 112, why did we write `mid = beg + (end - beg) / 2;` instead of `mid = (beg + end) /2;`?

3.5 Arrays

An array is a data structure that is similar to the library `vector` type (§ 3.3, p. 96) but offers a different trade-off between performance and flexibility. Like a `vector`, an array is a container of unnamed objects of a single type that we access by position. Unlike a `vector`, arrays have fixed size; we cannot add elements to an array. Because arrays have fixed size, they sometimes offer better run-time performance for specialized applications. However, that run-time advantage comes at the cost of lost flexibility.

If you don't know exactly how many elements you need, use a `vector`.

3.5.1 Defining and Initializing Built-in Arrays

Arrays are a compound type (§ 2.3, p. 50). An array declarator has the form `a[d]`, where `a` is the name being defined and `d` is the dimension of the array. The dimension specifies the number of elements and must be greater than zero. The number of elements in an array is part of the array's type. As a result, the dimension must be known at compile time, which means that the dimension must be a constant expression (§ 2.4.4, p. 65):

```
unsigned cnt = 42;           // not a constant expression
constexpr unsigned sz = 42; // constant expression
                             // constexpr see § 2.4.4 (p. 66)
int arr[10];               // array of ten ints
int *parr[sz];             // array of 42 pointers to int
string bad[cnt];           // error: cnt is not a constant expression
string strs[get_size()]; // ok if get_size is constexpr, error otherwise
```

By default, the elements in an array are default initialized (§ 2.2.1, p. 43).

As with variables of built-in type, a default-initialized array of built-in type that is defined inside a function will have undefined values.

When we define an array, we must specify a type for the array. We cannot use `auto` to deduce the type from a list of initializers. As with `vector`, arrays hold objects. Thus, there are no arrays of references.

Explicitly Initializing Array Elements

We can list initialize (§ 3.3.1, p. 98) the elements in an array. When we do so, we can omit the dimension. If we omit the dimension, the compiler infers it from the number of initializers. If we specify a dimension, the number of initializers must not exceed the specified size. If the dimension is greater than the number of initializers, the initializers are used for the first elements and any remaining elements are value initialized (§ 3.3.1, p. 98):

```
const unsigned sz = 3;
int ia1[sz] = {0,1,2};          //  array of three ints with values 0, 1, 2
int a2[] = {0, 1, 2};           //  an array of dimension 3
int a3[5] = {0, 1, 2};          //  equivalent to a3[] = {0, 1, 2, 0, 0}
string a4[3] = {"hi", "bye"}; //  same as a4[] = {"hi", "bye", ""}
int a5[2] = {0,1,2};            //  error: too many initializers
```

Character Arrays Are Special

Character arrays have an additional form of initialization: We can initialize such arrays from a string literal (§ 2.1.3, p. 39). When we use this form of initialization, it is important to remember that string literals end with a null character. That null character is copied into the array along with the characters in the literal:

```
char a1[] = {'C', '+', '+'};        //  list initialization, no null
char a2[] = {'C', '+', '+', '\0'}; //  list initialization, explicit null
char a3[] = "C++";                  //  null terminator added automatically
const char a4[6] = "Daniel";        //  error: no space for the null!
```

The dimension of `a1` is 3; the dimensions of `a2` and `a3` are both 4. The definition of `a4` is in error. Although the literal contains only six explicit characters, the array size must be at least seven—six to hold the literal and one for the null.

No Copy or Assignment

We cannot initialize an array as a copy of another array, nor is it legal to assign one array to another:

```
int a[] = {0, 1, 2}; //  array of three ints
int a2[] = a;        //  error: cannot initialize one array with another
a2 = a;              //  error: cannot assign one array to another
```

Some compilers allow array assignment as a **compiler extension**. It is usually a good idea to avoid using nonstandard features. Programs that use such features, will not work with a different compiler.

Understanding Complicated Array Declarations

Like `vectors`, arrays can hold objects of most any type. For example, we can have an array of pointers. Because an array is an object, we can define both pointers and references to arrays. Defining arrays that hold pointers is fairly straightforward, defining a pointer or reference to an array is a bit more complicated:

```
int *ptrs[10];              // ptrs is an array of ten pointers to int
int &refs[10] = /* ? */;    // error: no arrays of references
int (*Parray)[10] = &arr;   // Parray points to an array of ten ints
int (&arrRef)[10] = arr;    // arrRef refers to an array of ten ints
```

By default, type modifiers bind right to left. Reading the definition of `ptrs` from right to left (§ 2.3.3, p. 58) is easy: We see that we're defining an array of size 10, named `ptrs`, that holds pointers to `int`.

Reading the definition of `Parray` from right to left isn't as helpful. Because the array dimension follows the name being declared, it can be easier to read array declarations from the inside out rather than from right to left. Reading from the inside out makes it much easier to understand the type of `Parray`. We start by observing that the parentheses around `*Parray` mean that `Parray` is a pointer. Looking right, we see that `Parray` points to an array of size 10. Looking left, we see that the elements in that array are `int`s. Thus, `Parray` is a pointer to an array of ten `int`s. Similarly, `(&arrRef)` says that `arrRef` is a reference. The type to which it refers is an array of size 10. That array holds elements of type `int`.

Of course, there are no limits on how many type modifiers can be used:

```
int *(&arry)[10] = ptrs; // arry is a reference to an array of ten pointers
```

Reading this declaration from the inside out, we see that `arry` is a reference. Looking right, we see that the object to which `arry` refers is an array of size 10. Looking left, we see that the element type is pointer to `int`. Thus, `arry` is a reference to an array of ten pointers.

It can be easier to understand array declarations by starting with the array's name and reading them from the inside out.

Exercises Section 3.5.1

Exercise 3.27: Assuming `txt_size` is a function that takes no arguments and returns an `int` value, which of the following definitions are illegal? Explain why.

```
unsigned buf_size = 1024;
(a) int ia[buf_size];      (b) int ia[4 * 7 - 14];
(c) int ia[txt_size()];    (d) char st[11] = "fundamental";
```

Exercise 3.28: What are the values in the following arrays?

```
string sa[10];
int ia[10];
int main() {
    string sa2[10];
    int    ia2[10];
}
```

Exercise 3.29: List some of the drawbacks of using an array instead of a `vector`.

3.5.2 Accessing the Elements of an Array

As with the library `vector` and `string` types, we can use a range `for` or the subscript operator to access elements of an array. As usual, the indices start at 0. For an array of ten elements, the indices are 0 through 9, not 1 through 10.

When we use a variable to subscript an array, we normally should define that variable to have type **`size_t`**. `size_t` is a machine-specific unsigned type that is guaranteed to be large enough to hold the size of any object in memory. The `size_t` type is defined in the `cstddef` header, which is the C++ version of the `stddef.h` header from the C library.

With the exception that arrays are fixed size, we use arrays in ways that are similar to how we use `vectors`. For example, we can reimplement our grading program from § 3.3.3 (p. 104) to use an array to hold the cluster counters:

```
// count the number of grades by clusters of ten: 0--9, 10--19, . . . 90--99, 100
unsigned scores[11] = {};  // 11 buckets, all value initialized to 0
unsigned grade;
while (cin >> grade) {
    if (grade <= 100)
        ++scores[grade/10]; // increment the counter for the current cluster
}
```

The only obvious difference between this program and the one on page 104 is the declaration of `scores`. In this program `scores` is an array of 11 `unsigned` elements. The not so obvious difference is that the subscript operator in this program is the one that is defined as part of the language. This operator can be used on operands of array type. The subscript operator used in the program on page 104 was defined by the library `vector` template and applies to operands of type `vector`.

As in the case of `string` or `vector`, it is best to use a range `for` when we want to traverse the entire array. For example, we can print the resulting `scores` as follows:

```
for (auto i : scores)        // for each counter in scores
    cout << i << " ";        // print the value of that counter
cout << endl;
```

Because the dimension is part of each array type, the system knows how many elements are in `scores`. Using a range `for` means that we don't have to manage the traversal ourselves.

Checking Subscript Values

As with `string` and `vector`, it is up to the programmer to ensure that the subscript value is in range—that is, that the index value is equal to or greater than zero and less than the size of the array. Nothing stops a program from stepping across an array boundary except careful attention to detail and thorough testing of the code. It is possible for programs to compile and execute yet still be fatally wrong.

The most common source of security problems are buffer overflow bugs. Such bugs occur when a program fails to check a subscript and mistakenly uses memory outside the range of an array or similar data structure.

Exercises Section 3.5.2

Exercise 3.30: Identify the indexing errors in the following code:

```
constexpr size_t array_size = 10;
int ia[array_size];
for (size_t ix = 1; ix <= array_size; ++ix)
      ia[ix] = ix;
```

Exercise 3.31: Write a program to define an array of ten `int`s. Give each element the same value as its position in the array.

Exercise 3.32: Copy the array you defined in the previous exercise into another array. Rewrite your program to use `vector`s.

Exercise 3.33: What would happen if we did not initialize the `scores` array in the program on page 116?

3.5.3 Pointers and Arrays

In C++ pointers and arrays are closely intertwined. In particular, as we'll see, when we use an array, the compiler ordinarily converts the array to a pointer.

Normally, we obtain a pointer to an object by using the address-of operator (§ 2.3.2, p. 52). Generally speaking, the address-of operator may be applied to any object. The elements in an array are objects. When we subscript an array, the result is the object at that location in the array. As with any other object, we can obtain a pointer to an array element by taking the address of that element:

```
string nums[] = {"one", "two", "three"};  // array of strings
string *p = &nums[0];   // p points to the first element in nums
```

However, arrays have a special property—in most places when we use an array, the compiler automatically substitutes a pointer to the first element:

```
string *p2 = nums;       // equivalent to p2 = &nums[0]
```

In most expressions, when we use an object of array type, we are really using a pointer to the first element in that array.

There are various implications of the fact that operations on arrays are often really operations on pointers. One such implication is that when we use an array as an initializer for a variable defined using `auto` (§ 2.5.2, p. 68), the deduced type is a pointer, not an array:

```
int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia is an array of ten ints
auto ia2(ia); // ia2 is an int* that points to the first element in ia
ia2 = 42;     // error: ia2 is a pointer, and we can't assign an int to a pointer
```

Although `ia` is an array of ten `int`s, when we use `ia` as an initializer, the compiler treats that initialization as if we had written

```
auto ia2(&ia[0]);  // now it's clear that ia2 has type int*
```

It is worth noting that this conversion does not happen when we use `decltype` (§ 2.5.3, p. 70). The type returned by `decltype(ia)` is array of ten `int`s:

```
// ia3 is an array of ten ints
decltype(ia) ia3 = {0,1,2,3,4,5,6,7,8,9};
ia3 = p;     // error: can't assign an int* to an array
ia3[4] = i; // ok: assigns the value of i to an element in ia3
```

Pointers Are Iterators

Pointers that address elements in an array have additional operations beyond those we described in § 2.3.2 (p. 52). In particular, pointers to array elements support the same operations as iterators on `vector`s or `string`s (§ 3.4, p. 106). For example, we can use the increment operator to move from one element in an array to the next:

```
int arr[] = {0,1,2,3,4,5,6,7,8,9};
int *p = arr; // p points to the first element in arr
++p;          // p points to arr[1]
```

Just as we can use iterators to traverse the elements in a `vector`, we can use pointers to traverse the elements in an array. Of course, to do so, we need to obtain pointers to the first and one past the last element. As we've just seen, we can obtain a pointer to the first element by using the array itself or by taking the address-of the first element. We can obtain an off-the-end pointer by using another special property of arrays. We can take the address of the nonexistent element one past the last element of an array:

```
int *e = &arr[10]; // pointer just past the last element in arr
```

Here we used the subscript operator to index a nonexisting element; `arr` has ten elements, so the last element in `arr` is at index position 9. The only thing we can do with this element is take its address, which we do to initialize `e`. Like an off-the-end iterator (§ 3.4.1, p. 106), an off-the-end pointer does not point to an element. As a result, we may not dereference or increment an off-the-end pointer.

Using these pointers we can write a loop to print the elements in `arr` as follows:

```
for (int *b = arr; b != e; ++b)
    cout << *b << endl; // print the elements in arr
```

The Library `begin` and `end` Functions

C++ 11

Although we can compute an off-the-end pointer, doing so is error-prone. To make it easier and safer to use pointers, the new library includes two functions, named `begin` and `end`. These functions act like the similarly named container members (§ 3.4.1, p. 106). However, arrays are not class types, so these functions are not member functions. Instead, they take an argument that is an array:

```
int ia[] = {0,1,2,3,4,5,6,7,8,9}; // ia is an array of ten ints
int *beg = begin(ia); // pointer to the first element in ia
int *last = end(ia);  // pointer one past the last element in ia
```

`begin` returns a pointer to the first, and `end` returns a pointer one past the last element in the given array: These functions are defined in the `iterator` header.

Using `begin` and `end`, it is easy to write a loop to process the elements in an array. For example, assuming `arr` is an array that holds `int` values, we might find the first negative value in `arr` as follows:

```
// pbeg points to the first and pend points just past the last element in arr
int *pbeg = begin(arr), *pend = end(arr);
// find the first negative element, stopping if we've seen all the elements
while (pbeg != pend && *pbeg >= 0)
    ++pbeg;
```

We start by defining two `int` pointers named `pbeg` and `pend`. We position `pbeg` to denote the first element and `pend` to point one past the last element in `arr`. The `while` condition uses `pend` to know whether it is safe to dereference `pbeg`. If `pbeg` does point at an element, we dereference and check whether the underlying element is negative. If so, the condition fails and we exit the loop. If not, we increment the pointer to look at the next element.

A pointer "one past" the end of a built-in array behaves the same way as the iterator returned by the `end` operation of a `vector`. In particular, we may not dereference or increment an off-the-end pointer.

Pointer Arithmetic

Pointers that address array elements can use all the iterator operations listed in Table 3.6 (p. 107) and Table 3.7 (p. 111). These operations—dereference, increment, comparisons, addition of an integral value, subtraction of two pointers—have the same meaning when applied to pointers that point at elements in a built-in array as they do when applied to iterators.

When we add (or subtract) an integral value to (or from) a pointer, the result is a new pointer. That new pointer points to the element the given number ahead of (or behind) the original pointer:

```
constexpr size_t sz = 5;
int arr[sz] = {1,2,3,4,5};
int *ip = arr;      // equivalent to int *ip = &arr[0]
int *ip2 = ip + 4; // ip2 points to arr[4], the last element in arr
```

The result of adding `4` to `ip` is a pointer that points to the element four elements further on in the array from the one to which `ip` currently points.

The result of adding an integral value to a pointer must be a pointer to an element in the same array, or a pointer just past the end of the array:

```
// ok: arr is converted to a pointer to its first element; p points one past the end of arr
int *p = arr + sz; // use caution -- do not dereference!
int *p2 = arr + 10; // error: arr has only 5 elements; p2 has undefined value
```

When we add `sz` to `arr`, the compiler converts `arr` to a pointer to the first element in `arr`. When we add `sz` to that pointer, we get a pointer that points `sz` positions

(i.e., 5 positions) past the first one. That is, it points one past the last element in arr. Computing a pointer more than one past the last element is an error, although the compiler is unlikely to detect such errors.

As with iterators, subtracting two pointers gives us the distance between those pointers. The pointers must point to elements in the same array:

```
auto n = end(arr) - begin(arr); //  n is 5, the number of elements in arr
```

The result of subtracting two pointers is a library type named **ptrdiff_t**. Like size_t, the ptrdiff_t type is a machine-specific type and is defined in the cstddef header. Because subtraction might yield a negative distance, ptrdiff_t is a signed integral type.

We can use the relational operators to compare pointers that point to elements of an array, or one past the last element in that array. For example, we can traverse the elements in arr as follows:

```
int *b = arr, *e = arr + sz;
while (b < e) {
    //  use *b
    ++b;
}
```

We cannot use the relational operators on pointers to two unrelated objects:

```
int i = 0, sz = 42;
int *p = &i, *e = &sz;
//  undefined:  p and e are unrelated; comparison is meaningless!
while (p < e)
```

Although the utility may be obscure at this point, it is worth noting that pointer arithmetic is also valid for null pointers (§ 2.3.2, p. 53) and for pointers that point to an object that is not an array. In the latter case, the pointers must point to the same object, or one past that object. If p is a null pointer, we can add or subtract an integral constant expression (§ 2.4.4, p. 65) whose value is 0 to p. We can also subtract two null pointers from one another, in which case the result is 0.

Interaction between Dereference and Pointer Arithmetic

The result of adding an integral value to a pointer is itself a pointer. Assuming the resulting pointer points to an element, we can dereference the resulting pointer:

```
int ia[] = {0,2,4,6,8};  //  array with 5 elements of type int
int last = *(ia + 4);  //  ok: initializes last to 8, the value of ia[4]
```

The expression *(ia + 4) calculates the address four elements past ia and dereferences the resulting pointer. This expression is equivalent to writing ia[4].

Recall that in § 3.4.1 (p. 109) we noted that parentheses are required in expressions that contain dereference and dot operators. Similarly, the parentheses around this pointer addition are essential. Writing

```
last = *ia + 4;    //  ok: last = 4, equivalent to ia[0] + 4
```

means dereference ia and add 4 to the dereferenced value. We'll cover the reasons for this behavior in § 4.1.2 (p. 136).

Subscripts and Pointers

As we've seen, in most places when we use the name of an array, we are really using a pointer to the first element in that array. One place where the compiler does this transformation is when we subscript an array. Given

```
int ia[] = {0,2,4,6,8};  // array with 5 elements of type int
```

if we write `ia[0]`, that is an expression that uses the name of an array. When we subscript an array, we are really subscripting a pointer to an element in that array:

```
int i = ia[2];  // ia is converted to a pointer to the first element in ia
                // ia[2] fetches the element to which (ia + 2) points
int *p = ia;    // p points to the first element in ia
i = *(p + 2);   // equivalent to i = ia[2]
```

We can use the subscript operator on any pointer, as long as that pointer points to an element (or one past the last element) in an array:

```
int *p = &ia[2];  // p points to the element indexed by 2
int j = p[1];     // p[1] is equivalent to *(p + 1),
                  // p[1] is the same element as ia[3]
int k = p[-2];    // p[-2] is the same element as ia[0]
```

This last example points out an important difference between arrays and library types such as `vector` and `string` that have subscript operators. The library types force the index used with a subscript to be an unsigned value. The built-in subscript operator does not. The index used with the built-in subscript operator can be a negative value. Of course, the resulting address must point to an element in (or one past the end of) the array to which the original pointer points.

Unlike subscripts for `vector` and `string`, the index of the built-in subscript operator is not an `unsigned` type.

EXERCISES SECTION 3.5.3

Exercise 3.34: Given that `p1` and `p2` point to elements in the same array, what does the following code do? Are there values of `p1` or `p2` that make this code illegal?

```
p1 += p2 - p1;
```

Exercise 3.35: Using pointers, write a program to set the elements in an array to zero.

Exercise 3.36: Write a program to compare two arrays for equality. Write a similar program to compare two `vectors`.

3.5.4 C-Style Character Strings

Although C++ supports C-style strings, they should not be used by C++ programs. C-style strings are a surprisingly rich source of bugs and are the root cause of many security problems. They're also harder to use!

Character string literals are an instance of a more general construct that C++ inherits from C: **C-style character strings**. C-style strings are not a type. Instead, they are a convention for how to represent and use character strings. Strings that follow this convention are stored in character arrays and are **null terminated**. By null-terminated we mean that the last character in the string is followed by a null character (`'\0'`). Ordinarily we use pointers to manipulate these strings.

C Library String Functions

The Standard C library provides a set of functions, listed in Table 3.8, that operate on C-style strings. These functions are defined in the `cstring` header, which is the C++ version of the C header `string.h`.

The functions in Table 3.8 do not verify their string parameters.

The pointer(s) passed to these routines must point to null-terminated array(s):

```
char ca[] = {'C', '+', '+'};   // not null terminated
cout << strlen(ca) << endl;    // disaster: ca isn't null terminated
```

In this case, `ca` is an array of `char` but is not null terminated. The result is undefined. The most likely effect of this call is that `strlen` will keep looking through the memory that follows `ca` until it encounters a null character.

Table 3.8: C-Style Character String Functions

`strlen(p)`	Returns the length of `p`, *not counting the null*.
`strcmp(p1, p2)`	Compares `p1` and `p2` for equality. Returns `0` if `p1 == p2`, a positive value if `p1 > p2`, a negative value if `p1 < p2`.
`strcat(p1, p2)`	Appends `p2` to `p1`. Returns `p1`.
`strcpy(p1, p2)`	Copies `p2` into `p1`. Returns `p1`.

Comparing Strings

Comparing two C-style strings is done quite differently from how we compare library `strings`. When we compare two library `strings`, we use the normal relational or equality operators:

```
string s1 = "A string example";
string s2 = "A different string";
if (s1 < s2)  // false: s2 is less than s1
```

Using these operators on similarly defined C-style strings compares the pointer values, not the strings themselves:

```
const char ca1[] = "A string example";
const char ca2[] = "A different string";
if (ca1 < ca2)  // undefined: compares two unrelated addresses
```

Remember that when we use an array, we are really using a pointer to the first element in the array (§ 3.5.3, p. 117). Hence, this condition actually compares two `const char*` values. Those pointers do not address the same object, so the comparison is undefined.

To compare the strings, rather than the pointer values, we can call `strcmp`. That function returns 0 if the strings are equal, or a positive or negative value, depending on whether the first string is larger or smaller than the second:

```
if (strcmp(ca1, ca2) < 0) // same effect as string comparison s1 < s2
```

Caller Is Responsible for Size of a Destination String

Concatenating or copying C-style strings is also very different from the same operations on library `string`s. For example, if we wanted to concatenate the two `string`s `s1` and `s2` defined above, we can do so directly:

```
// initialize largeStr as a concatenation of s1, a space, and s2
string largeStr = s1 + " " + s2;
```

Doing the same with our two arrays, `ca1` and `ca2`, would be an error. The expression `ca1 + ca2` tries to add two pointers, which is illegal and meaningless.

Instead we can use `strcat` and `strcpy`. However, to use these functions, we must pass an array to hold the resulting string. The array we pass *must* be large enough to hold the generated string, including the null character at the end. The code we show here, although a common usage pattern, is fraught with potential for serious error:

```
// disastrous if we miscalculated the size of largeStr
strcpy(largeStr, ca1);      // copies ca1 into largeStr
strcat(largeStr, " ");      // adds a space at the end of largeStr
strcat(largeStr, ca2);      // concatenates ca2 onto largeStr
```

The problem is that we can easily miscalculate the size needed for `largeStr`. Moreover, any time we change the values we want to store in `largeStr`, we have to remember to double-check that we calculated its size correctly. Unfortunately, programs similar to this code are widely distributed. Programs with such code are error-prone and often lead to serious security leaks.

For most applications, in addition to being safer, it is also more efficient to use library `string`s rather than C-style strings.

Exercises Section 3.5.4

Exercise 3.37: What does the following program do?

```
const char ca[] = {'h', 'e', 'l', 'l', 'o'};
const char *cp = ca;
while (*cp) {
    cout << *cp << endl;
    ++cp;
}
```

Exercise 3.38: In this section, we noted that it was not only illegal but meaningless to try to add two pointers. Why would adding two pointers be meaningless?

Exercise 3.39: Write a program to compare two `string`s. Now write a program to compare the values of two C-style character strings.

Exercise 3.40: Write a program to define two character arrays initialized from string literals. Now define a third character array to hold the concatenation of the two arrays. Use `strcpy` and `strcat` to copy the two arrays into the third.

3.5.5 Interfacing to Older Code

Many C++ programs predate the standard library and do not use the `string` and `vector` types. Moreover, many C++ programs interface to programs written in C or other languages that cannot use the C++ library. Hence, programs written in modern C++ may have to interface to code that uses arrays and/or C-style character strings. The C++ library offers facilities to make the interface easier to manage.

Mixing Library `string`s and C-Style Strings

In § 3.2.1 (p. 84) we saw that we can initialize a `string` from a string literal:

```
string s("Hello World");  // s holds Hello World
```

More generally, we can use a null-terminated character array anywhere that we can use a string literal:

- We can use a null-terminated character array to initialize or assign a `string`.
- We can use a null-terminated character array as one operand (but not both operands) to the `string` addition operator or as the right-hand operand in the `string` compound assignment (+=) operator.

The reverse functionality is not provided: There is no direct way to use a library `string` when a C-style string is required. For example, there is no way to initialize a character pointer from a `string`. There is, however, a `string` member function named `c_str` that we can often use to accomplish what we want:

```
char *str = s; // error: can't initialize a char* from a string
const char *str = s.c_str(); // ok
```

The name `c_str` indicates that the function returns a C-style character string. That is, it returns a pointer to the beginning of a null-terminated character array that holds the same data as the characters in the `string`. The type of the pointer is `const char*`, which prevents us from changing the contents of the array.

The array returned by `c_str` is not guaranteed to be valid indefinitely. Any subsequent use of `s` that might change the value of `s` can invalidate this array.

If a program needs continuing access to the contents of the array returned by `str()`, the program must copy the array returned by `c_str`.

Using an Array to Initialize a `vector`

In § 3.5.1 (p. 114) we noted that we cannot initialize a built-in array from another array. Nor can we initialize an array from a `vector`. However, we can use an array to initialize a `vector`. To do so, we specify the address of the first element and one past the last element that we wish to copy:

```
int int_arr[] = {0, 1, 2, 3, 4, 5};
// ivec has six elements; each is a copy of the corresponding element in int_arr
vector<int> ivec(begin(int_arr), end(int_arr));
```

The two pointers used to construct `ivec` mark the range of values to use to initialize the elements in `ivec`. The second pointer points one past the last element to be copied. In this case, we used the library `begin` and `end` functions (§ 3.5.3, p. 118) to pass pointers to the first and one past the last elements in `int_arr`. As a result, `ivec` will have six elements each of which will have the same value as the corresponding element in `int_arr`.

The specified range can be a subset of the array:

```
// copies three elements: int_arr[1], int_arr[2], int_arr[3]
vector<int> subVec(int_arr + 1, int_arr + 4);
```

This initialization creates `subVec` with three elements. The values of these elements are copies of the values in `int_arr[1]` through `int_arr[3]`.

ADVICE: USE LIBRARY TYPES INSTEAD OF ARRAYS

Pointers and arrays are surprisingly error-prone. Part of the problem is conceptual: Pointers are used for low-level manipulations and it is easy to make bookkeeping mistakes. Other problems arise because of the syntax, particularly the declaration syntax used with pointers.

Modern C++ programs should use `vector`s and iterators instead of built-in arrays and pointers, and use `string`s rather than C-style array-based character strings.

3.6 Multidimensional Arrays

Strictly speaking, there are no multidimensional arrays in C++. What are commonly referred to as multidimensional arrays are actually arrays of arrays. It can

Exercises Section 3.5.5

Exercise 3.41: Write a program to initialize a `vector` from an array of `int`s.

Exercise 3.42: Write a program to copy a `vector` of `int`s into an array of `int`s.

be helpful to keep this fact in mind when you use what appears to be a multidimensional array.

We define an array whose elements are arrays by providing two dimensions: the dimension of the array itself and the dimension of its elements:

```
int ia[3][4]; //  array of size 3; each element is an array of ints of size 4
//  array of size 10; each element is a 20-element array whose elements are arrays of 30 ints
int arr[10][20][30] = {0}; //  initialize all elements to 0
```

As we saw in § 3.5.1 (p. 115), we can more easily understand these definitions by reading them from the inside out. We start with the name we're defining (`ia`) and see that `ia` is an array of size 3. Continuing to look to the right, we see that the elements of `ia` also have a dimension. Thus, the elements in `ia` are themselves arrays of size 4. Looking left, we see that the type of those elements is `int`. So, `ia` is an array of size 3, each of whose elements is an array of four `int`s.

We read the definition for `arr` in the same way. First we see that `arr` is an array of size 10. The elements of that array are themselves arrays of size 20. Each of those arrays has 30 elements that are of type `int`. There is no limit on how many subscripts are used. That is, we can have an array whose elements are arrays of elements that are arrays, and so on.

In a two-dimensional array, the first dimension is usually referred to as the row and the second as the column.

Initializing the Elements of a Multidimensional Array

As with any array, we can initialize the elements of a multidimensional array by providing a bracketed list of initializers. Multidimensional arrays may be initialized by specifying bracketed values for each row:

```
int ia[3][4] = {     //  three elements; each element is an array of size 4
    {0, 1, 2, 3},    //  initializers for the row indexed by 0
    {4, 5, 6, 7},    //  initializers for the row indexed by 1
    {8, 9, 10, 11}   //  initializers for the row indexed by 2
};
```

The nested braces are optional. The following initialization is equivalent, although considerably less clear:

```
//  equivalent initialization without the optional nested braces for each row
int ia[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

As is the case for single-dimension arrays, elements may be left out of the initializer list. We can initialize only the first element of each row as follows:

```
// explicitly initialize only element 0 in each row
int ia[3][4] = {{ 0 }, { 4 }, { 8 }};
```

The remaining elements are value initialized in the same way as ordinary, single-dimension arrays (§ 3.5.1, p. 114). If the nested braces were omitted, the results would be very different. This code

```
// explicitly initialize row 0; the remaining elements are value initialized
int ix[3][4] = {0, 3, 6, 9};
```

initializes the elements of the first row. The remaining elements are initialized to 0.

Subscripting a Multidimensional Array

As with any array, we can use a subscript to access the elements of a multidimensional array. To do so, we use a separate subscript for each dimension.

If an expression provides as many subscripts as there are dimensions, we get an element with the specified type. If we supply fewer subscripts than there are dimensions, then the result is the inner-array element at the specified index:

```
// assigns the first element of arr to the last element in the last row of ia
ia[2][3] = arr[0][0][0];
int (&row)[4] = ia[1]; // binds row to the second four-element array in ia
```

In the first example we supply indices for all the dimensions for both arrays. On the left-hand side, `ia[2]` returns the last row in `ia`. It does not fetch an element from that array but returns the array itself. We subscript that array, fetching element `[3]`, which is the last element in that array.

Similarly, the right-hand operand has three dimensions. We first fetch the array at index `0` from the outermost array. The result of that operation is a (multidimensional) array of size 20. We take the first element from that 20-element array, yielding an array of size 30. We then fetch the first element from that array.

In the second example, we define `row` as a reference to an array of four `int`s. We bind that reference to the second row in `ia`.

As another example, it is common to use a pair of nested `for` loops to process the elements in a multidimensional array:

```
constexpr size_t rowCnt = 3, colCnt = 4;
int ia[rowCnt][colCnt];    // 12 uninitialized elements
// for each row
for (size_t i = 0; i != rowCnt; ++i) {
    // for each column within the row
    for (size_t j = 0; j != colCnt; ++j) {
        // assign the element's positional index as its value
        ia[i][j] = i * colCnt + j;
    }
}
```

The outer `for` loops through each of the array elements in `ia`. The inner `for` loops through the elements of those interior arrays. In this case, we set the value of each element as its index in the overall array.

Using a Range `for` with Multidimensional Arrays

Under the new standard we can simplify the previous loop by using a range `for`:

```
size_t cnt = 0;
for (auto &row : ia)          // for every element in the outer array
    for (auto &col : row) {   // for every element in the inner array
        col = cnt;            // give this element the next value
        ++cnt;                // increment cnt
    }
```

This loop gives the elements of `ia` the same values as the previous loop, but this time we let the system manage the indices for us. We want to change the value of the elements, so we declare our control variables, `row` and `col`, as references (§ 3.2.3, p. 93). The first `for` iterates through the elements in `ia`. Those elements are arrays of size 4. Thus, the type of `row` is a reference to an array of four `int`s. The second `for` iterates through one of those 4-element arrays. Hence, `col` is `int&`. On each iteration we assign the value of `cnt` to the next element in `ia` and increment `cnt`.

In the previous example, we used references as our loop control variables because we wanted to change the elements in the array. However, there is a deeper reason for using references. As an example, consider the following loop:

```
for (const auto &row : ia)  // for every element in the outer array
    for (auto col : row)    // for every element in the inner array
        cout << col << endl;
```

This loop does not write to the elements, yet we still define the control variable of the outer loop as a reference. We do so in order to avoid the normal array to pointer conversion (§ 3.5.3, p. 117). Had we neglected the reference and written these loops as:

```
for (auto row : ia)
    for (auto col : row)
```

our program would not compile. As before, the first `for` iterates through `ia`, whose elements are arrays of size 4. Because `row` is not a reference, when the compiler initializes `row` it will convert each array element (like any other object of array type) to a pointer to that array's first element. As a result, in this loop the type of `row` is `int*`. The inner `for` loop is illegal. Despite our intentions, that loop attempts to iterate over an `int*`.

To use a multidimensional array in a range `for`, the loop control variable for all but the innermost array must be references.

Pointers and Multidimensional Arrays

As with any array, when we use the name of a multidimensional array, it is automatically converted to a pointer to the first element in the array.

When you define a pointer to a multidimensional array, remember that a multidimensional array is really an array of arrays.

Because a multidimensional array is really an array of arrays, the pointer type to which the array converts is a pointer to the first inner array:

```
int ia[3][4];      // array of size 3; each element is an array of ints of size 4
int (*p)[4] = ia; // p points to an array of four ints
p = &ia[2];        // p now points to the last element in ia
```

Applying the strategy from § 3.5.1 (p. 115), we start by noting that `(*p)` says `p` is a pointer. Looking right, we see that the object to which `p` points has a dimension of size `4`, and looking left that the element type is `int`. Hence, `p` is a pointer to an array of four `int`s.

The parentheses in this declaration are essential:

```
int *ip[4];     // array of pointers to int
int (*ip)[4];   // pointer to an array of four ints
```

With the advent of the new standard, we can often avoid having to write the type of a pointer into an array by using `auto` or `decltype` (§ 2.5.2, p. 68):

C++ 11

```
// print the value of each element in ia, with each inner array on its own line
// p points to an array of four ints
for (auto p = ia; p != ia + 3; ++p) {
    // q points to the first element of an array of four ints; that is, q points to an int
    for (auto q = *p; q != *p + 4; ++q)
         cout << *q << ' ';
    cout << endl;
}
```

The outer `for` loop starts by initializing `p` to point to the first array in `ia`. That loop continues until we've processed all three rows in `ia`. The increment, `++p`, has the effect of moving `p` to point to the next row (i.e., the next element) in `ia`.

The inner `for` loop prints the values of the inner arrays. It starts by making `q` point to the first element in the array to which `p` points. The result of `*p` is an array of four `int`s. As usual, when we use an array, it is converted automatically to a pointer to its first element. The inner `for` loop runs until we've processed every element in the inner array. To obtain a pointer just off the end of the inner array, we again dereference `p` to get a pointer to the first element in that array. We then add 4 to that pointer to process the four elements in each inner array.

Of course, we can even more easily write this loop using the library `begin` and `end` functions (§ 3.5.3, p. 118):

```
// p points to the first array in ia
for (auto p = begin(ia); p != end(ia); ++p) {
    // q points to the first element in an inner array
    for (auto q = begin(*p); q != end(*p); ++q)
        cout << *q << ' ';    // prints the int value to which q points
    cout << endl;
}
```

Here we let the library determine the end pointer, and we use `auto` to avoid having to write the type returned from `begin`. In the outer loop, that type is a pointer to an array of four `int`s. In the inner loop, that type is a pointer to `int`.

Type Aliases Simplify Pointers to Multidimensional Arrays

A type alias (§ 2.5.1, p. 67) can make it easier to read, write, and understand pointers to multidimensional arrays. For example:

```
using int_array = int[4]; //  new style type alias declaration; see § 2.5.1 (p. 68)
typedef int int_array[4]; //  equivalent typedef declaration; § 2.5.1 (p. 67)
// print the value of each element in ia, with each inner array on its own line
for (int_array *p = ia; p != ia + 3; ++p) {
    for (int *q = *p; q != *p + 4; ++q)
         cout << *q << ' ';
    cout << endl;
}
```

Here we start by defining `int_array` as a name for the type "array of four `int`s." We use that type name to define our loop control variable in the outer `for` loop.

EXERCISES SECTION 3.6

Exercise 3.43: Write three different versions of a program to print the elements of `ia`. One version should use a range `for` to manage the iteration, the other two should use an ordinary `for` loop in one case using subscripts and in the other using pointers. In all three programs write all the types directly. That is, do not use a type alias, `auto`, or `decltype` to simplify the code.

Exercise 3.44: Rewrite the programs from the previous exercises using a type alias for the type of the loop control variables.

Exercise 3.45: Rewrite the programs again, this time using `auto`.

Chapter Summary

Among the most important library types are `vector` and `string`. A `string` is a variable-length sequence of characters, and a `vector` is a container of objects of a single type.

Iterators allow indirect access to objects stored in a container. Iterators are used to access and navigate between the elements in `string`s and `vector`s.

Arrays and pointers to array elements provide low-level analogs to the `vector` and `string` libraries. In general, the library classes should be used in preference to low-level array and pointer alternatives built into the language.

Defined Terms

begin Member of `string` and `vector` that returns an iterator to the first element. Also, free-standing library function that takes an array and returns a pointer to the first element in the array.

buffer overflow Serious programming bug that results when we use an index that is out-of-range for a container, such as a `string`, `vector`, or an array.

C-style strings Null-terminated character array. String literals are C-style strings. C-style strings are inherently error-prone.

class template A blueprint from which specific clas types can be created. To use a class template, we must specify additional information. For example, to define a `vector`, we specify the element type: `vector<int>` holds `int`s.

compiler extension Feature that is added to the language by a particular compiler. Programs that rely on compiler extensions cannot be moved easily to other compilers.

container A type whose objects hold a collection of objects of a given type. `vector` is a container type.

copy initialization Form of initialization that uses an `=`. The newly created object is a copy of the given initializer.

difference_type A `signed` integral type defined by `vector` and `string` that can hold the distance between any two iterators.

direct initialization Form of initialization that does not include an `=`.

empty Member of `string` and `vector`. Returns `bool`, which is `true` if `size` is zero, `false` otherwise.

end Member of `string` and `vector` that returns an off-the-end iterator. Also, free-standing library function that takes an array and returns a pointer one past the last element in the array.

getline Function defined in the `string` header that takes an `istream` and a `string`. The function reads the stream up to the next newline, storing what it read into the `string`, and returns the `istream`. The newline is read and discarded.

index Value used in the subscript operator to denote the element to retrieve from a `string`, `vector`, or array.

instantiation Compiler process that generates a specific template class or function.

iterator A type used to access and navigate among the elements of a container.

iterator arithmetic Operations on `vector` or `string` iterators: Adding or subtracting an integral value and an iterator yields an iterator that many elements ahead of or behind the original iterator. Subtracting one iterator from another yields the distance between them. Iterators must refer to elements in, or off-the-end of the same container.

null-terminated string String whose last character is followed by the null character (`'\0'`).

off-the-end iterator The iterator returned by `end` that refers to a nonexistent element one past the end of a container.

pointer arithmetic The arithmetic operations that can be applied to pointers. Pointers to arrays support the same operations as iterator arithmetic.

ptrdiff_t Machine-dependent signed integral type defined in the `cstddef` header that is large enough to hold the difference between two pointers into the largest possible array.

push_back Member of `vector`. Appends elements to the back of a `vector`.

range for Control statement that iterates through a specified collection of values.

size Member of `string` and `vector`. Returns the number of characters or elements, respectively. Returns a value of the `size_type` for the type.

size_t Machine-dependent unsigned integral type defined in the `cstddef` header that is large enough to hold the size of the largest possible array.

size_type Name of types defined by the `string` and `vector` classes that are capable of containing the size of any `string` or `vector`, respectively. Library classes that define `size_type` define it as an `unsigned` type.

string Library type that represents a sequence of characters.

using declarations Make a name from a namespace accessible directly.

```
using namespace::name;
```

makes *name* accessible without the *namespace*`::` prefix.

value initialization Initialization in which built-in types are initialized to zero and class types are initialized by the class's default constructor. Objects of a class type can be value initialized only if the class has a default constructor. Used to initialize a container's elements when a size, but not an element initializer, is specified. Elements are initialized as a copy of this compiler-generated value.

vector Library type that holds a collection of elements of a specified type.

`++` operator The iterator types and pointers define the increment operator to "add one" by moving the iterator to refer to the next element.

`[ ]` operator Subscript operator. `obj[i]` yields the element at position `i` from the container object `obj`. Indices count from zero—the first element is element `0` and the last is the element indexed by `obj.size() - 1`. Subscript returns an object. If `p` is a pointer and `n` an integer, `p[n]` is a synonym for `*(p+n)`.

`->` operator Arrow operator. Combines the operations of dereference and dot operators: `a->b` is a synonym for `(*a).b`.

`<<` operator The `string` library type defines an output operator. The `string` operator prints the characters in a `string`.

`>>` operator The `string` library type defines an input operator. The `string` operator reads whitespace-delimited chunks of characters, storing what is read into the right-hand (`string`) operand.

`!` operator Logical NOT operator. Returns the inverse of the `bool` value of its operand. Result is `true` if operand is `false` and vice versa.

`&&` operator Logical AND operator. Result is `true` if both operands are `true`. The right-hand operand is evaluated *only* if the left-hand operand is `true`.

`||` operator Logical OR operator. Yields `true` if either operand is `true`. The right-hand operand is evaluated *only* if the left-hand operand is `false`.

CHAPTER 4

EXPRESSIONS

CONTENTS

C++ provides a rich set of operators and defines what these operators do when applied to operands of built-in type. It also allows us to define the meaning of most of the operators when applied to operands of class types. This chapter focuses on the operators as defined in the language and applied to operands of built-in type. We will also look at some of the operators defined by the library. Chapter 14 will show how we can define operators for our own types.

An expression is composed of one or more **operands** and yields a **result** when it is evaluated. The simplest form of an **expression** is a single literal or variable. The result of such an expression is the value of the variable or literal. More complicated expressions are formed from an **operator** and one or more operands.

4.1 Fundamentals

There are a few fundamental concepts that affect how expressions are evaluated. We start by briefly discussing the concepts that apply to most (if not all) expressions. Subsequent sections will cover these topics in more detail.

4.1.1 Basic Concepts

There are both ***unary operators*** and ***binary operators***. Unary operators, such as address-of (`&`) and dereference (`*`), act on one operand. Binary operators, such as equality (`==`) and multiplication (`*`), act on two operands. There is also one ternary operator that takes three operands, and one operator, function call, that takes an unlimited number of operands.

Some symbols, such as `*`, are used as both a unary (dereference) and a binary (multiplication) operator. The context in which a symbol is used determines whether the symbol represents a unary or binary operator. The uses of such symbols are independent; it can be helpful to think of them as two different symbols.

Grouping Operators and Operands

Understanding expressions with multiple operators requires understanding the ***precedence*** and ***associativity*** of the operators and may depend on the ***order of evaluation*** of the operands. For example, the result of the following expression depends on how the operands are grouped to the operators:

```
5 + 10 * 20/2;
```

The operands to the `*` operator could be `10` and `20`, or `10` and `20/2`, or `15` and `20`, or `15` and `20/2`. Understanding such expressions is the topic of the next section.

Operand Conversions

As part of evaluating an expression, operands are often converted from one type to another. For example, the binary operators usually expect operands with the same type. These operators can be used on operands with differing types so long as the operands can be converted (§ 2.1.2, p. 35) to a common type.

Although the rules are somewhat complicated, for the most part conversions happen in unsurprising ways. For example, we can convert an integer to floating-point, and vice versa, but we cannot convert a pointer type to floating-point. What may be a bit surprising is that small integral type operands (e.g., `bool`, `char`, `short`, etc.) are generally **promoted** to a larger integral type, typically `int`. We'll look in detail at conversions in § 4.11 (p. 159).

Overloaded Operators

The language defines what the operators mean when applied to built-in and compound types. We can also define what most operators mean when applied to class types. Because such definitions give an alternative meaning to an existing operator symbol, we refer to them as **overloaded operators**. The IO library `>>` and `<<` operators and the operators we used with `string`s, `vector`s, and iterators are all overloaded operators.

When we use an overloaded operator, the meaning of the operator—including the type of its operand(s) and the result—depend on how the operator is defined. However, the number of operands and the precedence and the associativity of the operator cannot be changed.

Lvalues and Rvalues

Every expression in C++ is either an **rvalue** (pronounced "are-value") or an **lvalue** (pronounced "ell-value"). These names are inherited from C and originally had a simple mnemonic purpose: lvalues could stand on the left-hand side of an assignment whereas rvalues could not.

In C++, the distinction is less simple. In C++, an lvalue expression yields an object or a function. However, some lvalues, such as `const` objects, may not be the left-hand operand of an assignment. Moreover, some expressions yield objects but return them as rvalues, not lvalues. Roughly speaking, when we use an object as an rvalue, we use the object's value (its contents). When we use an object as an lvalue, we use the object's identity (its location in memory).

Operators differ as to whether they require lvalue or rvalue operands and as to whether they return lvalues or rvalues. The important point is that (with one exception that we'll cover in § 13.6 (p. 531)) we can use an lvalue when an rvalue is required, but we cannot use an rvalue when an lvalue (i.e., a location) is required. When we use an lvalue in place of an rvalue, the object's contents (its value) are used. We have already used several operators that involve lvalues.

- Assignment requires a (non`const`) lvalue as its left-hand operand and yields its left-hand operand as an lvalue.
- The address-of operator (§ 2.3.2, p. 52) requires an lvalue operand and returns a pointer to its operand as an rvalue.
- The built-in dereference and subscript operators (§ 2.3.2, p. 53, and § 3.5.2, p. 116) and the iterator dereference and `string` and `vector` subscript operators (§ 3.4.1, p. 106, § 3.2.3, p. 93, and § 3.3.3, p. 102) all yield lvalues.
- The built-in and iterator increment and decrement operators (§ 1.4.1, p. 12, and § 3.4.1, p. 107) require lvalue operands and the prefix versions (which are the ones we have used so far) also yield lvalues.

As we present the operators, we will note whether an operand must be an lvalue and whether the operator returns an lvalue.

Lvalues and rvalues also differ when used with `decltype` (§ 2.5.3, p. 70). When we apply `decltype` to an expression (other than a variable), the result is

a reference type if the expression yields an lvalue. As an example, assume `p` is an `int*`. Because dereference yields an lvalue, `decltype(*p)` is `int&`. On the other hand, because the address-of operator yields an rvalue, `decltype(&p)` is `int**`, that is, a pointer to a pointer to type `int`.

4.1.2 Precedence and Associativity

An expression with two or more operators is a **compound expression**. Evaluating a compound expression involves grouping the operands to the operators. Precedence and associativity determine how the operands are grouped. That is, they determine which parts of the expression are the operands for each of the operators in the expression. Programmers can override these rules by parenthesizing compound expressions to force a particular grouping.

In general, the value of an expression depends on how the subexpressions are grouped. Operands of operators with higher precedence group more tightly than operands of operators at lower precedence. Associativity determines how to group operands with the same precedence. For example, multiplication and division have the same precedence as each other, but they have higher precedence than addition. Therefore, operands to multiplication and division group before operands to addition and subtraction. The arithmetic operators are left associative, which means operators at the same precdence group left to right:

- Because of precedence, the expression `3+4*5` is `23`, not `35`.
- Because of associativity, the expression `20-15-3` is `2`, not `8`.

As a more complicated example, a left-to-right evaluation of the following expression yields 20:

```
6 + 3 * 4 / 2 + 2
```

Other imaginable results include 9, 14, and 36. In C++, the result is 14, because this expression is equivalent to

```
// parentheses in this expression match default precedence and associativity
((6 + ((3 * 4) / 2)) + 2)
```

Parentheses Override Precedence and Associativity

We can override the normal grouping with parentheses. Parenthesized expressions are evaluated by treating each parenthesized subexpression as a unit and otherwise applying the normal precedence rules. For example, we can parenthesize the expression above to force the result to be any of the four possible values:

```
// parentheses result in alternative groupings
cout << (6 + 3) * (4 / 2 + 2) << endl;      // prints 36
cout << ((6 + 3) * 4) / 2 + 2 << endl;      // prints 20
cout << 6 + 3 * 4 / (2 + 2) << endl;        // prints 9
```

When Precedence and Associativity Matter

We have already seen examples where precedence affects the correctness of our programs. For example, consider the discussion in § 3.5.3 (p. 120) about dereference and pointer arithmetic:

```
int ia[] = {0,2,4,6,8}; //  array with five elements of type int
int last = *(ia + 4);   //  initializes last to 8, the value of ia[4]
last = *ia + 4;          //  last = 4, equivalent to ia[0] + 4
```

If we want to access the element at the location `ia + 4`, then the parentheses around the addition are essential. Without parentheses, `*ia` is grouped first and `4` is added to the value in `*ia`.

The most common case that we've seen in which associativity matters is in input and output expressions. As we'll see in § 4.8 (p. 155), the operators used for IO are left associative. This associativity means we can combine several IO operations in a single expression:

```
cin >> v1 >> v2;   // read into v1 and then into v2
```

Table 4.12 (p. 166) lists all the operators organized into segments separated by double lines. Operators in each segment have the same precedence, and have higher precedence than operators in subsequent segments. For example, the prefix increment and dereference operators share the same precedence, which is higher than that of the arithmetic operators. The table includes a page reference to each operator's description. We have seen some of these operators already and will cover most of the rest in this chapter. However, there are a few operators that we will not cover until later.

EXERCISES SECTION 4.1.2

Exercise 4.1: What is the value returned by `5 + 10 * 20/2`?

Exercise 4.2: Using Table 4.12 (p. 166), parenthesize the following expressions to indicate the order in which the operands are grouped:

(a) `* vec.begin()` (b) `* vec.begin() + 1`

4.1.3 Order of Evaluation

Precedence specifies how the operands are grouped. It says nothing about the order in which the operands are evaluated. In most cases, the order is largely unspecified. In the following expression

```
int i = f1() * f2();
```

we know that `f1` and `f2` must be called before the multiplication can be done. After all, it is their results that are multiplied. However, we have no way of knowing whether `f1` will be called before `f2` or vice versa.

For operators that do not specify evaluation order, it is an error for an expression to *refer to and change* the same object. Expressions that do so have undefined behavior (§ 2.1.2, p. 36). As a simple example, the `<<` operator makes no guarantees about when or how its operands are evaluated. As a result, the following output expression is undefined:

```
int i = 0;
cout << i << " " << ++i << endl;  // undefined
```

Because this program is undefined, we cannot draw any conclusions about how it might behave. The compiler might evaluate `++i` before evaluating `i`, in which case the output will be `1 1`. Or the compiler might evaluate `i` first, in which case the output will be `0 1`. *Or the compiler might do something else entirely.* Because this expression has undefined behavior, the program is in error, regardless of what code the compiler generates.

There are four operators that do guarantee the order in which operands are evaluated. We saw in § 3.2.3 (p. 94) that the logical AND (`&&`) operator guarantees that its left-hand operand is evaluated first. Moreover, we are also guaranteed that the right-hand operand is evaluated only if the left-hand operand is `true`. The only other operators that guarantee the order in which operands are evaluated are the logical OR (`||`) operator (§ 4.3, p. 141), the conditional (`? :`) operator (§ 4.7, p. 151), and the comma (`,`) operator (§ 4.10, p. 157).

Order of Evaluation, Precedence, and Associativity

Order of operand evaluation is independent of precedence and associativity. In an expression such as `f() + g() * h() + j()`:

- Precedence guarantees that the results of `g()` and `h()` are multiplied.
- Associativity guarantees that the result of `f()` is added to the product of `g()` and `h()` and that the result of that addition is added to the value of `j()`.
- There are no guarantees as to the order in which these functions are called.

If `f`, `g`, `h`, and `j` are independent functions that do not affect the state of the same objects or perform IO, then the order in which the functions are called is irrelevant. If any of these functions do affect the same object, then the expression is in error and has undefined behavior.

EXERCISES SECTION 4.1.3

Exercise 4.3: Order of evaluation for most of the binary operators is left undefined to give the compiler opportunities for optimization. This strategy presents a trade-off between efficient code generation and potential pitfalls in the use of the language by the programmer. Do you consider that an acceptable trade-off? Why or why not?

ADVICE: MANAGING COMPOUND EXPRESSIONS

When you write compound expressions, two rules of thumb can be helpful:

1. **When in doubt, parenthesize expressions to force the grouping that the logic of your program requires.**
2. **If you change the value of an operand, don't use that operand elsewhere in the same expresion.**

An important exception to the second rule occurs when the subexpression that changes the operand is itself the operand of another subexpression. For example, in `*++iter`, the increment changes the value of `iter`. The (now changed) value of `iter` is the operand to the dereference operator. In this (and similar) expressions, order of evaluation isn't an issue. The increment (i.e., the subexpression that changes the operand) must be evaluated before the dereference can be evaluated. Such usage poses no problems and is quite common.

4.2 Arithmetic Operators

Table 4.1: Arithmetic Operators (Left Associative)

Operator	Function	Use
`+`	unary plus	`+ expr`
`-`	unary minus	`- expr`
`*`	multiplication	`expr * expr`
`/`	division	`expr / expr`
`%`	remainder	`expr % expr`
`+`	addition	`expr + expr`
`-`	subtraction	`expr - expr`

Table 4.1 (and the operator tables in subsequent sections) groups the operators by their precedence. The unary arithmetic operators have higher precedence than the multiplication and division operators, which in turn have higher precedence than the binary addition and subtraction operators. Operators of higher precedence group more tightly than do operators with lower precedence. These operators are all left associative, meaning that they group left to right when the precedence levels are the same.

Unless noted otherwise, the arithmetic operators may be applied to any of the arithmetic types (§ 2.1.1, p. 32) or to any type that can be converted to an arithmetic type. The operands and results of these operators are rvalues. As described in § 4.11 (p. 159), operands of small integral types are promoted to a larger integral type, and all operands may be converted to a common type as part of evaluating these operators.

The unary plus operator and the addition and subtraction operators may also be applied to pointers. § 3.5.3 (p. 119) covered the use of binary + and - with

pointer operands. When applied to a pointer or arithmetic value, unary plus returns a (possibly promoted) copy of the value of its operand.

The unary minus operator returns the result of negating a (possibly promoted) copy of the value of its operand:

```
int i = 1024;
int k = -i; // i is -1024
bool b = true;
bool b2 = -b; // b2 is true!
```

In § 2.1.1 (p. 34) we noted that `bool` values should not be used for computation. The result of `-b` is a good example of what we had in mind.

For most operators, operands of type `bool` are promoted to `int`. In this case, the value of `b` is `true`, which promotes to the `int` value `1` (§ 2.1.2, p. 35). That (promoted) value is negated, yielding `-1`. The value `-1` is converted back to `bool` and used to initialize `b2`. This initializer is a nonzero value, which when converted to `bool` is `true`. Thus, the value of `b2` is `true`!

Caution: Overflow and Other Arithmetic Exceptions

Some arithmetic expressions yield undefined results. Some of these undefined expressions are due to the nature of mathematics—for example, division by zero. Others are undefined due to the nature of computers—for example, due to overflow. Overflow happens when a value is computed that is outside the range of values that the type can represent.

Consider a machine on which `short`s are 16 bits. In that case, the maximum `short` is 32767. On such a machine, the following compound assignment overflows:

```
short short_value = 32767; // max value if shorts are 16 bits
short_value += 1; // this calculation overflows
cout << "short_value: " << short_value << endl;
```

The assignment to `short_value` is undefined. Representing a signed value of 32768 requires 17 bits, but only 16 are available. On many systems, there is *no* compile-time or run-time warning when an overflow occurs. As with any undefined behavior, what happens is unpredictable. On our system the program completes and writes

```
short_value: -32768
```

The value "wrapped around": The sign bit, which had been 0, was set to 1, resulting in a negative value. On another system, the result might be different, or the program might behave differently, including crashing entirely.

When applied to objects of arithmetic types, the arithmetic operators, `+`, `-`, `*`, and `/`, have their obvious meanings: addition, subtraction, multiplication, and division. Division between integers returns an integer. If the quotient contains a fractional part, it is truncated toward zero:

```
int ival1 = 21/6;   // ival1 is 3; result is truncated; remainder is discarded
int ival2 = 21/7;   // ival2 is 3; no remainder; result is an integral value
```

The % operator, known as the "remainder" or the "modulus" operator, computes the remainder that results from dividing the left-hand operand by the right-hand operand. The operands to % must have integral type:

```
int ival = 42;
double dval = 3.14;
ival % 12;    // ok: result is 6
ival % dval; // error: floating-point operand
```

In a division, a nonzero quotient is positive if the operands have the same sign and negative otherwise. Earlier versions of the language permitted a negative quotient to be rounded up or down; the new standard requires the quotient to be rounded toward zero (i.e., truncated). C++ 11

The modulus operator is defined so that if m and n are integers and n is nonzero, then (m/n)*n + m%n is equal to m. By implication, if m%n is nonzero, it has the same sign as m. Earlier versions of the language permitted m%n to have the same sign as n on implementations in which negative m/n was rounded away from zero, but such implementations are now prohibited. Moreover, except for the obscure case where -m overflows, (-m)/n and m/(-n) are always equal to -(m/n), m%(-n) is equal to m%n, and (-m)%n is equal to -(m%n). More concretely:

```
 21 % 6;    /* result is 3  */      21 / 6;    /* result is 3  */
 21 % 7;    /* result is 0  */      21 / 7;    /* result is 3  */
-21 % -8;   /* result is -5 */     -21 / -8;   /* result is 2  */
 21 % -5;   /* result is 1  */      21 / -5;   /* result is -4 */
```

Exercises Section 4.2

Exercise 4.4: Parenthesize the following expression to show how it is evaluated. Test your answer by compiling the expression (without parentheses) and printing its result.

```
12 / 3 * 4 + 5 * 15 + 24 % 4 / 2
```

Exercise 4.5: Determine the result of the following expressions.

(a) `-30 * 3 + 21 / 5` (b) `-30 + 3 * 21 / 5`
(c) `30 / 3 * 21 % 5` (d) `-30 / 3 * 21 % 4`

Exercise 4.6: Write an expression to determine whether an int value is even or odd.

Exercise 4.7: What does overflow mean? Show three expressions that will overflow.

4.3 Logical and Relational Operators

The relational operators take operands of arithmetic or pointer type; the logical operators take operands of any type that can be converted to bool. These operators all return values of type bool. Arithmetic and pointer operand(s) with a value of zero are false; all other values are true. The operands to these operators are rvalues and the result is an rvalue.

Table 4.2: Logical and Relational Operators			
Associativity	**Operator**	**Function**	**Use**
Right	`!`	logical NOT	`!expr`
Left	`<`	less than	`expr < expr`
Left	`<=`	less than or equal	`expr <= expr`
Left	`>`	greater than	`expr > expr`
Left	`>=`	greater than or equal	`expr >= expr`
Left	`==`	equality	`expr == expr`
Left	`!=`	inequality	`expr != expr`
Left	`&&`	logical AND	`expr && expr`
Left	`||`	logical OR	`expr || expr`

Logical AND and OR Operators

The overall result of the logical AND operator is `true` if and only if both its operands evaluate to `true`. The logical OR (`||`) operator evaluates as `true` if either of its operands evaluates as `true`.

The logical AND and OR operators always evaluate their left operand before the right. Moreover, the right operand is evaluated *if and only if* the left operand does not determine the result. This strategy is known as **short-circuit evaluation**:

- The right side of an `&&` is evaluated if and only if the left side is `true`.
- The right side of an `||` is evaluated if and only if the left side is `false`.

Several of the programs in Chapter 3 used the logical AND operator. Those programs used the left-hand operand to test whether it was safe to evaluate the right-hand operand. For example, the `for` condition on page 94:

```
index != s.size() && !isspace(s[index])
```

first checks that `index` has not reached the end of its associated `string`. We're guaranteed that the right operand won't be evaluated unless `index` is in range.

As an example that uses the logical OR, imagine we have some text in a `vector` of `strings`. We want to print the `strings`, adding a newline after each empty `string` or after a `string` that ends with a period. We'll use a range-based `for` loop (§ 3.2.3, p. 91) to process each element:

```
// note s as a reference to const; the elements aren't copied and can't be changed
for (const auto &s : text) { // for each element in text
    cout << s;          // print the current element
    // blank lines and those that end with a period get a newline
    if (s.empty() || s[s.size() - 1] == '.')
        cout << endl;
    else
        cout << " ";   // otherwise just separate with a space
}
```

After we print the current element, we check to see if we need to print a newline. The condition in the `if` first checks whether `s` is an empty `string`. If so, we need to print a newline regardless of the value of the right-hand operand. Only if the `string` is not empty do we evaluate the second expression, which checks whether the `string` ends with a period. In this expression, we rely on short-circuit evaluation of `||` to ensure that we subscript `s` only if `s` is not empty.

It is worth noting that we declared `s` as a reference to `const` (§ 2.5.2, p. 69). The elements in `text` are `string`s, and might be large. By making `s` a reference, we avoid copying the elements. Because we don't need to write to the elements, we made `s` a reference to `const`.

Logical NOT Operator

The logical NOT operator (`!`) returns the inverse of the truth value of its operand. We first used this operator in § 3.2.2 (p. 87). As another example, assuming `vec` is a `vector` of `int`s, we might use the logical NOT operator to see whether `vec` has elements by negating the value returned by `empty`:

```
// print the first element in vec if there is one
if (!vec.empty())
    cout << vec[0];
```

The subexpression

```
!vec.empty()
```

evaluates as `true` if the call to `empty` returns `false`.

The Relational Operators

The relational operators (`<`, `<=`, `>`, `<=`) have their ordinary meanings and return `bool` values. These operators are left associative.

Because the relational operators return `bool`s, the result of chaining these operators together is likely to be surprising:

```
// oops! this condition compares k to the bool result of i < j
if (i < j < k) //  true if k is greater than 1!
```

This condition groups `i` and `j` to the first `<` operator. The `bool` result of that expression is the left-hand operand of the second less-than operator. That is, `k` is compared to the `true`/`false` result of the first comparison! To accomplish the test we intended, we can rewrite the expression as follows:

```
// ok: condition is true if i is smaller than j and j is smaller than k
if (i < j && j < k) { /* ... */ }
```

Equality Tests and the `bool` Literals

If we want to test the truth value of an arithmetic or pointer object, the most direct way is to use the value as a condition:

```
if (val)  { /* ... */ } //  true if val is any nonzero value
if (!val) { /* ... */ } //  true if val is zero
```

In both conditions, the compiler converts `val` to `bool`. The first condition succeeds so long as `val` is nonzero; the second succeeds if `val` is zero.

We might think we could rewrite a test of this kind as

```
if (val == true)  { /* ... */ } //  true only if val is equal to 1!
```

There are two problems with this approach. First, it is longer and less direct than the previous code (although admittedly when first learning C++ this kind of abbreviation can be perplexing). Much more importantly, when `val` is not a `bool`, this comparison does not work as expected.

If `val` is not a `bool`, then `true` is converted to the type of `val` before the `==` operator is applied. That is, when `val` is not a `bool`, it is as if we had written

```
if (val == 1) { /* ... */ }
```

As we've seen, when a `bool` is converted to another arithmetic type, `false` converts to `0` and `true` converts to `1` (§ 2.1.2, p. 35). If we really cared whether `val` was the specific value `1`, we should write the condition to test that case directly.

It is usually a bad idea to use the boolean literals `true` and `false` as operands in a comparison. These literals should be used only to compare to an object of type `bool`.

Exercises Section 4.3

Exercise 4.8: Explain when operands are evaluated in the logical AND, logical OR, and equality operators.

Exercise 4.9: Explain the behavior of the condition in the following `if`:

```
const char *cp = "Hello World";
if (cp && *cp)
```

Exercise 4.10: Write the condition for a `while` loop that would read `int`s from the standard input and stop when the value read is equal to `42`.

Exercise 4.11: Write an expression that tests four values, `a`, `b`, `c`, and `d`, and ensures that `a` is greater than `b`, which is greater than `c`, which is greater than `d`.

Exercise 4.12: Assuming `i`, `j`, and `k` are all `int`s, explain what `i != j < k` means.

4.4 Assignment Operators

The left-hand operand of an assignment operator must be a modifiable lvalue. For example, given

```
int i = 0, j = 0, k = 0; //  initializations, not assignment
const int ci = i;        //  initialization, not assignment
```

Each of these assignments is illegal:

```
1024 = k;       // error: literals are rvalues
i + j = k;      // error: arithmetic expressions are rvalues
ci = k;         // error: ci is a const (nonmodifiable) lvalue
```

The result of an assignment is its left-hand operand, which is an lvalue. The type of the result is the type of the left-hand operand. If the types of the left and right operands differ, the right-hand operand is converted to the type of the left:

```
k = 0;          // result: type int, value 0
k = 3.14159;    // result: type int, value 3
```

Under the new standard, we can use a braced initializer list (§ 2.2.1, p. 43) on the right-hand side:

C++ 11

```
k = {3.14};                    // error: narrowing conversion
vector<int> vi;                // initially empty
vi = {0,1,2,3,4,5,6,7,8,9};    // vi now has ten elements, values 0 through 9
```

If the left-hand operand is of a built-in type, the initializer list may contain at most one value, and that value must not require a narrowing conversion (§ 2.2.1, p. 43).

For class types, what happens depends on the details of the class. In the case of `vector`, the `vector` template defines its own version of an assignment operator that can take an initializer list. This operator replaces the elements of the left-hand side with the elements in the list on the right-hand side.

Regardless of the type of the left-hand operand, the initializer list may be empty. In this case, the compiler generates a value-initialized (§ 3.3.1, p. 98) temporary and assigns that value to the left-hand operand.

Assignment Is Right Associative

Unlike the other binary operators, assignment is right associative:

```
int ival, jval;
ival = jval = 0; // ok: each assigned 0
```

Because assignment is right associative, the right-most assignment, `jval = 0`, is the right-hand operand of the left-most assignment operator. Because assignment returns its left-hand operand, the result of the right-most assignment (i.e., `jval`) is assigned to `ival`.

Each object in a multiple assignment must have the same type as its right-hand neighbor or a type to which that neighbor can be converted (§ 4.11, p. 159):

```
int ival, *pval; // ival is an int; pval is a pointer to int
ival = pval = 0; // error: cannot assign the value of a pointer to an int
string s1, s2;
s1 = s2 = "OK";  // string literal "OK" converted to string
```

The first assignment is illegal because `ival` and `pval` have different types and there is no conversion from the type of `pval` (`int*`) to the type of `ival` (`int`). It is illegal even though zero is a value that can be assigned to either object.

On the other hand, the second assignment is fine. The string literal is converted to `string`, and that `string` is assigned to `s2`. The result of that assignment is `s2`, which has the same type as `s1`.

Assignment Has Low Precedence

Assignments often occur in conditions. Because assignment has relatively low precedence, we usually must parenthesize the assignment for the condition to work properly. To see why assignment in a condition is useful, consider the following loop. We want to call a function until it returns a desired value—say, 42:

```
// a verbose and therefore more error-prone way to write this loop
int i = get_value();  // get the first value
while (i != 42) {
    // do something . . .
    i = get_value();  // get remaining values
}
```

Here we start by calling `get_value` followed by a loop whose condition uses the value returned from that call. The last statement in this loop makes another call to `get_value`, and the loop repeats. We can write this code more directly as

```
int i;
// a better way to write our loop---what the condition does is now clearer
while ((i = get_value()) != 42) {
    // do something . . .
}
```

The condition now more clearly expresses our intent: We want to continue until `get_value` returns `42`. The condition executes by assigning the result returned by `get_value` to `i` and then comparing the result of that assignment with `42`.

Without the parentheses, the operands to `!=` would be the value returned from `get_value` and `42`. The `true` or `false` result of that test would be assigned to `i`—clearly not what we intended!

Because assignment has lower precedence than the relational operators, parentheses are usually needed around assignments in conditions.

Beware of Confusing Equality and Assignment Operators

The fact that we can use assignment in a condition can have surprising effects:

```
if (i = j)
```

The condition in this `if` assigns the value of `j` to `i` and then tests the result of the assignment. If `j` is nonzero, the condition will be `true`. The author of this code almost surely intended to test whether `i` and `j` have the same value:

```
if (i == j)
```

Bugs of this sort are notoriously difficult to find. Some, but not all, compilers are kind enough to warn about code such as this example.

Compound Assignment Operators

We often apply an operator to an object and then assign the result to that same object. As an example, consider the sum program from § 1.4.2 (p. 13):

```
int sum = 0;
// sum values from 1 through 10 inclusive
for (int val = 1; val <= 10; ++val)
    sum += val;  // equivalent to sum = sum + val
```

This kind of operation is common not just for addition but for the other arithmetic operators and the bitwise operators, which we cover in § 4.8 (p. 152). There are compound assignments for each of these operators:

```
 +=    -=    *=    /=    %=        // arithmetic operators
<<=   >>=   &=    ^=    |=        // bitwise operators; see § 4.8 (p. 152)
```

Each compound operator is essentially equivalent to

```
a = a op b;
```

with the exception that, when we use the compound assignment, the left-hand operand is evaluated only once. If we use an ordinary assignment, that operand is evaluated twice: once in the expression on the right-hand side and again as the operand on the left hand. In many, perhaps most, contexts this difference is immaterial aside from possible performance consequences.

EXERCISES SECTION 4.4

Exercise 4.13: What are the values of `i` and `d` after each assignment?

```
int i;   double d;
(a) d = i = 3.5;      (b) i = d = 3.5;
```

Exercise 4.14: Explain what happens in each of the `if` tests:

```
if (42 = i)   // ...
if (i = 42)   // ...
```

Exercise 4.15: The following assignment is illegal. Why? How would you correct it?

```
double dval; int ival; int *pi;
dval = ival = pi = 0;
```

Exercise 4.16: Although the following are legal, they probably do not behave as the programmer expects. Why? Rewrite the expressions as you think they should be.

```
(a) if (p = getPtr() != 0)     (b) if (i = 1024)
```

4.5 Increment and Decrement Operators

The increment (++) and decrement (--) operators provide a convenient notational shorthand for adding or subtracting 1 from an object. This notation rises above

mere convenience when we use these operators with iterators, because many iterators do not support arithmetic.

There are two forms of these operators: prefix and postfix. So far, we have used only the prefix form. This form increments (or decrements) its operand and yields the *changed* object as its result. The postfix operators increment (or decrement) the operand but yield a copy of the original, *unchanged* value as its result:

```
int i = 0, j;
j = ++i; //  j = 1, i = 1: prefix yields the incremented value
j = i++; //  j = 1, i = 2: postfix yields the unincremented value
```

These operators require lvalue operands. The prefix operators return the object itself as an lvalue. The postfix operators return a copy of the object's original value as an rvalue.

ADVICE: USE POSTFIX OPERATORS ONLY WHEN NECESSARY

Readers from a C background might be surprised that we use the prefix increment in the programs we've written. The reason is simple: The prefix version avoids unnecessary work. It increments the value and returns the incremented version. The postfix operator must store the original value so that it can return the unincremented value as its result. If we don't need the unincremented value, there's no need for the extra work done by the postfix operator.

For `int`s and pointers, the compiler can optimize away this extra work. For more complicated iterator types, this extra work potentially might be more costly. By habitually using the prefix versions, we do not have to worry about whether the performance difference matters. Moreover—and perhaps more importantly—we can express the intent of our programs more directly.

Combining Dereference and Increment in a Single Expression

The postfix versions of `++` and `--` are used when we want to use the current value of a variable and increment it in a single compound expression.

As one example, we can use postfix increment to write a loop to print the values in a `vector` up to but not including the first negative value:

```
auto pbeg = v.begin();
//  print elements up to the first negative value
while (pbeg != v.end() && *beg >= 0)
    cout << *pbeg++ << endl; //  print the current value and advance pbeg
```

The expression `*pbeg++` is usually confusing to programmers new to both C++ and C. However, because this usage pattern is so common, C++ programmers must understand such expressions.

The precedence of postfix increment is higher than that of the dereference operator, so `*pbeg++` is equivalent to `*(pbeg++)`. The subexpression `pbeg++` increments `pbeg` and yields a copy of the previous value of `pbeg` as its result. Accordingly, the operand of `*` is the unincremented value of `pbeg`. Thus, the statement prints the element to which `pbeg` originally pointed and increments `pbeg`.

This usage relies on the fact that postfix increment returns a copy of its original, unincremented operand. If it returned the incremented value, we'd dereference the incremented value, with disastrous results. We'd skip the first element. Worse, if the sequence had no negative values, we would attempt to dereference one too many elements.

ADVICE: BREVITY CAN BE A VIRTUE

Expressions such as `*pbeg++` can be bewildering—at first. However, it is a useful and widely used idiom. Once the notation is familiar, writing

```
cout << *iter++ << endl;
```

is easier and less error-prone than the more verbose equivalent

```
cout << *iter << endl;
++iter;
```

It is worthwhile to study examples of such code until their meanings are immediately clear. Most C++ programs use succinct expressions rather than more verbose equivalents. Therefore, C++ programmers must be comfortable with such usages. Moreover, once these expressions are familiar, you will find them less error-prone.

Remember That Operands Can Be Evaluated in Any Order

Most operators give no guarantee as to the order in which operands will be evaluated (§ 4.1.3, p. 137). This lack of guaranteed order often doesn't matter. The cases where it does matter are when one subexpression changes the value of an operand that is used in another subexpression. Because the increment and decrement operators change their operands, it is easy to misuse these operators in compound expressions.

To illustrate the problem, we'll rewrite the loop from § 3.4.1 (p. 108) that capitalizes the first word in the input. That example used a `for` loop:

```
for (auto it = s.begin(); it != s.end() && !isspace(*it); ++it)
    *it = toupper(*it); //  capitalize the current character
```

which allowed us to separate the statement that dereferenced `beg` from the one that incremented it. Replacing the `for` with a seemingly equivalent `while`

```
//  the behavior of the following loop is undefined!
while (beg != s.end() && !isspace(*beg))
    *beg = toupper(*beg++);    //  error: this assignment is undefined
```

results in undefined behavior. The problem is that in the revised version, both the left- and right-hand operands to = use `beg` *and* the right-hand operand changes `beg`. The assignment is therefore undefined. The compiler might evaluate this expression as either

```
*beg = toupper(*beg);          //  execution if left-hand side is evaluated first
*(beg + 1) = toupper(*beg);    //  execution if right-hand side is evaluated first
```

or it might evaluate it in yet some other way.

Exercises Section 4.5

Exercise 4.17: Explain the difference between prefix and postfix increment.

Exercise 4.18: What would happen if the `while` loop on page 148 that prints the elements from a `vector` used the prefix increment operator?

Exercise 4.19: Given that `ptr` points to an `int`, that `vec` is a `vector<int>`, and that `ival` is an `int`, explain the behavior of each of these expressions. Which, if any, are likely to be incorrect? Why? How might each be corrected?

(a) `ptr != 0 && *ptr++` (b) `ival++ && ival`
(c) `vec[ival++] <= vec[ival]`

4.6 The Member Access Operators

The dot (§ 1.5.2, p. 23) and arrow (§ 3.4.1, p. 110) operators provide for member access. The dot operator fetches a member from an object of class type; arrow is defined so that *ptr*`->`*mem* is a synonym for `(*`*ptr*`).`*mem*:

```
string s1 = "a string", *p = &s1;
auto n = s1.size(); // run the size member of the string s1
n = (*p).size();    // run size on the object to which p points
n = p->size();      // equivalent to (*p).size()
```

Because dereference has a lower precedence than dot, we must parenthesize the dereference subexpression. If we omit the parentheses, this code means something quite different:

```
// run the size member of p, then dereference the result!
*p.size();      // error: p is a pointer and has no member named size
```

This expression attempts to fetch the `size` member of the object `p`. However, `p` is a pointer, which has no members; this code will not compile.

The arrow operator requires a pointer operand and yields an lvalue. The dot operator yields an lvalue if the object from which the member is fetched is an lvalue; otherwise the result is an rvalue.

Exercises Section 4.6

Exercise 4.20: Assuming that `iter` is a `vector<string>::iterator`, indicate which, if any, of the following expressions are legal. Explain the behavior of the legal expressions and why those that aren't legal are in error.

(a) `*iter++;` (b) `(*iter)++;` (c) `*iter.empty()`
(d) `iter->empty();` (e) `++*iter;` (f) `iter++->empty();`

4.7 The Conditional Operator

The conditional operator (the **`?:` operator**) lets us embed simple if-else logic inside an expression. The conditional operator has the following form:

cond `?` *expr1* `:` *expr2*`;`

where *cond* is an expression that is used as a condition and *expr1* and *expr2* are expressions of the same type (or types that can be converted to a common type). This operator executes by evaluating *cond*. If the condition is `true`, then *expr1* is evaluated; otherwise, *expr2* is evaluated. As one example, we can use a conditional operator to determine whether a grade is pass or fail:

```
string finalgrade = (grade < 60) ?  "fail" : "pass";
```

The condition checks whether `grade` is less than `60`. If so, the result of the expression is `"fail"`; otherwise the result is `"pass"`. Like the logical AND and logical OR (`&&` and `||`) operators, the conditional operator guarantees that only one of *expr1* or *expr2* is evaluated.

That result of the conditional operator is an lvalue if both expressions are lvalues or if they convert to a common lvalue type. Otherwise the result is an rvalue.

Nesting Conditional Operations

We can nest one conditional operator inside another. That is, the conditional operator can be used as the *cond* or as one or both of the *expr*s of another conditional expression. As an example, we'll use a pair of nested conditionals to perform a three-way test to indicate whether a grade is a high pass, an ordinary pass, or fail:

```
finalgrade = (grade > 90) ? "high pass"
                          : (grade < 60) ? "fail" : "pass";
```

The first condition checks whether the grade is above `90`. If so, the expression after the `?` is evaluated, which yields `"high pass"`. If the condition fails, the `:` branch is executed, which is itself another conditional expression. This conditional asks whether the `grade` is less than `60`. If so, the `?` branch is evaluated and yields `"fail"`. If not, the `:` branch returns `"pass"`.

The conditional operator is right associative, meaning (as usual) that the operands group right to left. Associativity accounts for the fact that the right-hand conditional—the one that compares `grade` to `60`—forms the `:` branch of the left-hand conditional expression.

Nested conditionals quickly become unreadable. It's a good idea to nest no more than two or three.

Using a Conditional Operator in an Output Expression

The conditional operator has fairly low precedence. When we embed a conditional expression in a larger expression, we usually must parenthesize the conditional subexpression. For example, we often use the conditional operator to print one or

another value, depending on the result of a condition. An incompletely parenthesized conditional operator in an output expression can have surprising results:

```
cout << ((grade < 60) ?  "fail" : "pass"); // prints pass or fail
cout << (grade < 60) ?  "fail" : "pass";   // prints 1 or 0!
cout << grade < 60 ?  "fail" : "pass"; // error: compares cout to 60
```

The second expression uses the comparison between `grade` and `60` as the operand to the `<<` operator. The value `1` or `0` is printed, depending on whether `grade < 60` is true or false. The `<<` operator returns `cout`, which is tested as the condition for the conditional operator. That is, the second expression is equivalent to

```
cout << (grade < 60);     // prints 1 or 0
cout ?  "fail" : "pass"; // test cout and then yield one of the two literals
                         // depending on whether cout is true or false
```

The last expression is an error because it is equivalent to

```
cout << grade;    // less-than has lower precedence than shift, so print grade first
cout < 60 ? "fail" : "pass"; // then compare cout to 60!
```

Exercises Section 4.7

Exercise 4.21: Write a program to use a conditional operator to find the elements in a `vector<int>` that have odd value and double the value of each such element.

Exercise 4.22: Extend the program that assigned high pass, pass, and fail grades to also assign low pass for grades between 60 and 75 inclusive. Write two versions: One version that uses only conditional operators; the other should use one or more `if` statements. Which version do you think is easier to understand and why?

Exercise 4.23: The following expression fails to compile due to operator precedence. Using Table 4.12 (p. 166), explain why it fails. How would you fix it?

```
string s = "word";
string pl = s + s[s.size() - 1] == 's' ? "" : "s" ;
```

Exercise 4.24: Our program that distinguished between high pass, pass, and fail depended on the fact that the conditional operator is right associative. Describe how that operator would be evaluated if the operator were left associative.

4.8 The Bitwise Operators

The bitwise operators take operands of integral type that they use as a collection of bits. These operators let us test and set individual bits. As we'll see in § 17.2 (p. 723), we can also use these operators on a library type named `bitset` that represents a flexibly sized collection of bits.

As usual, if an operand is a "small integer," its value is first promoted (§ 4.11.1, p. 160) to a larger integral type. The operand(s) can be either signed or unsigned.

Table 4.3: Bitwise Operators (Left Associative)

Operator	Function	Use
~	bitwise NOT	`~expr`
<<	left shift	`expr1 << expr2`
>>	right shift	`expr1 >> expr2`
&	bitwise AND	`expr1 & expr2`
^	bitwise XOR	`expr1 ^ expr2`
|	bitwise OR	`expr1 | expr2`

If the operand is signed and its value is negative, then the way that the "sign bit" is handled in a number of the bitwise operations is machine dependent. Moreover, doing a left shift that changes the value of the sign bit is undefined.

Because there are no guarantees for how the sign bit is handled, we strongly recommend using `unsigned` types with the bitwise operators.

Bitwise Shift Operators

We have already used the overloaded versions of the >> and << operators that the IO library defines to do input and output. The built-in meaning of these operators is that they perform a bitwise shift on their operands. They yield a value that is a copy of the (possibly promoted) left-hand operand with the bits shifted as directed by the right-hand operand. The right-hand operand must not be negative and must be a value that is strictly less than the number of bits in the result. Otherwise, the operation is undefined. The bits are shifted left (<<) or right (>>). Bits that are shifted off the end are discarded:

These illustrations have the low-order bit on the right
These examples assume `char` has 8 bits, and `int` has 32

```
// 0233 is an octal literal (§ 2.1.3, p. 38)
unsigned char bits = 0233;
```

1	0	0	1	1	0	1	1

`bits << 8` *// `bits` promoted to `int` and then shifted left by 8 bits*

0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	1 0 0 1 1 0 1 1	0 0 0 0 0 0 0 0

`bits << 31` *// left shift 31 bits, left-most bits discarded*

1 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0

`bits >> 3` *// right shift 3 bits, 3 right-most bits discarded*

0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 1 0 0 1 1

The left-shift operator (the **<< operator**) inserts 0-valued bits on the right. The behavior of the right-shift operator (the **>> operator**) depends on the type of the left-hand operand: If that operand is `unsigned`, then the operator inserts 0-valued

bits on the left; if it is a signed type, the result is implementation defined—either copies of the sign bit or 0-valued bits are inserted on the left.

Bitwise NOT Operator

The bitwise NOT operator (the **~operator**) generates a new value with the bits of its operand inverted. Each 1 bit is set to 0; each 0 bit is set to 1:

```
unsigned char bits = 0227;   1 0 0 1 0 1 1 1

~bits
1 1 1 1 1 1 1 1 | 1 1 1 1 1 1 1 1 | 1 1 1 1 1 1 1 1 | 0 1 1 0 1 0 0 0
```

Here, our `char` operand is first promoted to `int`. Promoting a `char` to `int` leaves the value unchanged but adds 0 bits to the high order positions. Thus, promoting `bits` to `int` adds 24 high order bits, all of which are 0-valued. The bits in the promoted value are inverted.

Bitwise AND, OR, and XOR Operators

The AND (&), OR (|), and XOR (^) operators generate new values with the bit pattern composed from its two operands:

```
unsigned char b1 = 0145;        0 1 1 0 0 1 0 1

unsigned char b2 = 0257;        1 0 1 0 1 1 1 1

   b1 & b2   24 high-order bits all 0   0 0 1 0 0 1 0 1

   b1 | b2   24 high-order bits all 0   1 1 1 0 1 1 1 1

   b1 ^ b2   24 high-order bits all 0   1 1 0 0 1 0 1 0
```

For each bit position in the result of the bitwise AND operator (the **& operator**) the bit is 1 if both operands contain 1; otherwise, the result is 0. For the OR (inclusive or) operator (the | **operator**), the bit is 1 if either or both operands contain 1; otherwise, the result is 0. For the XOR (exclusive or) operator (the ^ **operator**), the bit is 1 if either but not both operands contain 1; otherwise, the result is 0.

It is a common error to confuse the bitwise and logical operators (§ 4.3, p. 141). For example to confuse the bitwise `&` with the logical `&&`, the bitwise `|` with the logical `||`, and the bitwise `~`and the logical `!`).

Using Bitwise Operators

As an example of using the bitwise operators let's assume a teacher has 30 students in a class. Each week the class is given a pass/fail quiz. We'll track the results of each quiz using one bit per student to represent the pass or fail grade on a given test. We might represent each quiz in an `unsigned` integral value:

```
unsigned long quiz1 = 0;  // we'll use this value as a collection of bits
```

We define `quiz1` as an `unsigned long`. Thus, `quiz1` will have at least 32 bits on any machine. We explicitly initialize `quiz1` to ensure that the bits start out with well-defined values.

The teacher must be able to set and test individual bits. For example, we'd like to be able to set the bit corresponding to student number 27 to indicate that this student passed the quiz. We can indicate that student number 27 passed by creating a value that has only bit 27 turned on. If we then bitwise OR that value with `quiz1`, all the bits except bit 27 will remain unchanged.

For the purpose of this example, we will count the bits of `quiz1` by assigning 0 to the low-order bit, 1 to the next bit, and so on.

We can obtain a value indicating that student 27 passed by using the left-shift operator and an `unsigned long` integer literal 1 (§ 2.1.3, p. 38):

```
1UL << 27   // generate a value with only bit number 27 set
```

`1UL` has a 1 in the low-order bit and (at least) 31 zero bits. We specified `unsigned long` because `int`s are only guaranteed to have 16 bits, and we need at least 27. This expression shifts the 1 bit left 27 positions inserting 0 bits behind it.

Next we OR this value with `quiz1`. Because we want to update the value of `quiz1`, we use a compound assignment (§ 4.4, p. 147):

```
quiz1 |= 1UL << 27;   // indicate student number 27 passed
```

The `|=` operator executes analogously to how `+=` does. It is equivalent to

```
quiz1 = quiz1 | 1UL << 27;   // equivalent to quiz1 |= 1UL << 27;
```

Imagine that the teacher reexamined the quiz and discovered that student 27 actually had failed the test. The teacher must now turn off bit 27. This time we need an integer that has bit 27 turned off and all the other bits turned on. We'll bitwise AND this value with `quiz1` to turn off just that bit:

```
quiz1 &= ~(1UL << 27);   // student number 27 failed
```

We obtain a value with all but bit 27 turned on by inverting our previous value. That value had 0 bits in all but bit 27, which was a 1. Applying the bitwise NOT to that value will turn off bit 27 and turn on all the others. When we bitwise AND this value with `quiz1`, all except bit 27 will remain unchanged.

Finally, we might want to know how the student at position 27 fared:

```
bool status = quiz1 & (1UL << 27);   // how did student number 27 do?
```

Here we AND a value that has bit 27 turned on with `quiz1`. The result is nonzero (i.e., true) if bit 27 of `quiz1` is also on; otherwise, it evaluates to zero.

Shift Operators (aka IO Operators) Are Left Associative

Although many programmers never use the bitwise operators directly, most programmers do use overloaded versions of these operators for IO. An overloaded operator has the same precedence and associativity as the built-in version of that operator. Therefore, programmers need to understand the precedence and associativity of the shift operators even if they never use them with their built-in meaning.

Because the shift operators are left associative, the expression

```
cout << "hi" << " there" << endl;
```

executes as

```
( (cout << "hi") << " there" ) << endl;
```

In this statement, the operand `"hi"` is grouped with the first `<<` symbol. Its result is grouped with the second, and then that result is grouped with the third.

The shift operators have midlevel precedence: lower than the arithmetic operators but higher than the relational, assignment, and conditional operators. These relative precedence levels mean we usually have to use parentheses to force the correct grouping of operators with lower precedence.

```
cout << 42 + 10;    // ok: + has higher precedence, so the sum is printed
cout << (10 < 42); // ok: parentheses force intended grouping; prints 1
cout << 10 < 42;    // error: attempt to compare cout to 42!
```

The last `cout` is interpreted as

```
(cout << 10) < 42;
```

which says to "write `10` onto `cout` and then compare the result of that operation (i.e., `cout`) to `42`."

Exercises Section 4.8

Exercise 4.25: What is the value of `~'q' << 6` on a machine with 32-bit `int`s and 8 bit `char`s, that uses Latin-1 character set in which `'q'` has the bit pattern `01110001`?

Exercise 4.26: In our grading example in this section, what would happen if we used `unsigned int` as the type for `quiz1`?

Exercise 4.27: What is the result of each of these expressions?

```
unsigned long ul1 = 3, ul2 = 7;
(a)  ul1 & ul2            (b)  ul1 | ul2
(c)  ul1 && ul2           (d)  ul1 || ul2
```

4.9 The `sizeof` Operator

The **`sizeof`** operator returns the size, in bytes, of an expression or a type name. The operator is right associative. The result of `sizeof` is a constant expression (§ 2.4.4, p. 65) of type `size_t` (§ 3.5.2, p. 116). The operator takes one of two forms:

```
sizeof (type)
sizeof expr
```

In the second form, `sizeof` returns the size of the type returned by the given expression. The `sizeof` operator is unusual in that it does not evaluate its operand:

```
Sales_data data, *p;
sizeof(Sales_data); //  size required to hold an object of type Sales_data
sizeof data; //  size of data's type, i.e., sizeof(Sales_data)
sizeof p;    //  size of a pointer
sizeof *p;   //  size of the type to which p points, i.e., sizeof(Sales_data)
sizeof data.revenue; //  size of the type of Sales_data's revenue member
sizeof Sales_data::revenue; //  alternative way to get the size of revenue
```

The most interesting of these examples is `sizeof *p`. First, because `sizeof` is right associative and has the same precedence as `*`, this expression groups right to left. That is, it is equivalent to `sizeof (*p)`. Second, because `sizeof` does not evaluate its operand, it doesn't matter that `p` is an invalid (i.e., uninitialized) pointer (§ 2.3.2, p. 52). Dereferencing an invalid pointer as the operand to `sizeof` is safe because the pointer is not actually used. `sizeof` doesn't need dereference the pointer to know what type it will return.

Under the new standard, we can use the scope operator to ask for the size of a member of a class type. Ordinarily we can only access the members of a class through an object of that type. We don't need to supply an object, because `sizeof` does not need to fetch the member to know its size.

C++ 11

The result of applying `sizeof` depends in part on the type involved:

- `sizeof char` or an expression of type `char` is guaranteed to be 1.
- `sizeof` a reference type returns the size of an object of the referenced type.
- `sizeof` a pointer returns the size needed hold a pointer.
- `sizeof` a dereferenced pointer returns the size of an object of the type to which the pointer points; the pointer need not be valid.
- `sizeof` an array is the size of the entire array. It is equivalent to taking the `sizeof` the element type times the number of elements in the array. Note that `sizeof` does not convert the array to a pointer.
- `sizeof` a `string` or a `vector` returns only the size of the fixed part of these types; it does not return the size used by the object's elements.

Because `sizeof` returns the size of the entire array, we can determine the number of elements in an array by dividing the array size by the element size:

```
//  sizeof(ia)/sizeof(*ia) returns the number of elements in ia
constexpr size_t sz = sizeof(ia)/sizeof(*ia);
int arr2[sz];  //  ok sizeof returns a constant expression § 2.4.4 (p. 65)
```

Because `sizeof` returns a constant expression, we can use the result of a `sizeof` expression to specify the dimension of an array.

4.10 Comma Operator

The **comma operator** takes two operands, which it evaluates from left to right. Like the logical AND and logical OR and the conditional operator, the comma operator guarantees the order in which its operands are evaluated.

Exercises Section 4.9

Exercise 4.28: Write a program to print the size of each of the built-in types.

Exercise 4.29: Predict the output of the following code and explain your reasoning. Now run the program. Is the output what you expected? If not, figure out why.

```
int x[10];   int *p = x;
cout << sizeof(x)/sizeof(*x) << endl;
cout << sizeof(p)/sizeof(*p) << endl;
```

Exercise 4.30: Using Table 4.12 (p. 166), parenthesize the following expressions to match the default evaluation:

(a) `sizeof x + y` (b) `sizeof p->mem[i]`
(c) `sizeof a < b` (d) `sizeof f()`

The left-hand expression is evaluated and its result is discarded. The result of a comma expression is the value of its right-hand expression. The result is an lvalue if the right-hand operand is an lvalue.

One common use for the comma operator is in a `for` loop:

```
vector<int>::size_type cnt = ivec.size();
// assign values from size... 1 to the elements in ivec
for(vector<int>::size_type ix = 0;
                ix != ivec.size(); ++ix, --cnt)
    ivec[ix] = cnt;
```

This loop increments `ix` and decrements `cnt` in the expression in the `for` header. Both `ix` and `cnt` are changed on each trip through the loop. As long as the test of `ix` succeeds, we reset the current element to the current value of `cnt`.

Exercises Section 4.10

Exercise 4.31: The program in this section used the prefix increment and decrement operators. Explain why we used prefix and not postfix. What changes would have to be made to use the postfix versions? Rewrite the program using postfix operators.

Exercise 4.32: Explain the following loop.

```
constexpr int size = 5;
int ia[size] = {1,2,3,4,5};
for (int *ptr = ia, ix = 0;
     ix != size && ptr != ia+size;
     ++ix, ++ptr)   { /* ... */ }
```

Exercise 4.33: Using Table 4.12 (p. 166) explain what the following expression does:

```
someValue ? ++x, ++y : --x, --y
```

4.11 Type Conversions

In C++ some types are related to each other. When two types are related, we can use an object or value of one type where an operand of the related type is expected. Two types are related if there is a **conversion** between them.

As an example, consider the following expression, which initializes `ival` to 6:

```
int ival = 3.541 + 3; // the compiler might warn about loss of precision
```

The operands of the addition are values of two different types: `3.541` has type `double`, and `3` is an `int`. Rather than attempt to add values of the two different types, C++ defines a set of conversions to transform the operands to a common type. These conversions are carried out automatically without programmer intervention—and sometimes without programmer knowledge. For that reason, they are referred to as **implicit conversions**.

The implicit conversions among the arithmetic types are defined to preserve precision, if possible. Most often, if an expression has both integral and floating-point operands, the integer is converted to floating-point. In this case, `3` is converted to `double`, floating-point addition is done, and the result is a `double`.

The initialization happens next. In an initialization, the type of the object we are initializing dominates. The initializer is converted to the object's type. In this case, the `double` result of the addition is converted to `int` and used to initialize `ival`. Converting a `double` to an `int` truncates the `double`'s value, discarding the decimal portion. In this expression, the value `6` is assigned to `ival`.

When Implicit Conversions Occur

The compiler automatically converts operands in the following circumstances:

- In most expressions, values of integral types smaller than `int` are first promoted to an appropriate larger integral type.
- In conditions, nonbool expressions are converted to `bool`.
- In initializations, the initializer is converted to the type of the variable; in assignments, the right-hand operand is converted to the type of the left-hand.
- In arithmetic and relational expressions with operands of mixed types, the types are converted to a common type.
- As we'll see in Chapter 6, conversions also happen during function calls.

4.11.1 The Arithmetic Conversions

The **arithmetic conversions**, which we introduced in § 2.1.2 (p. 35), convert one arithmetic type to another. The rules define a hierarchy of type conversions in which operands to an operator are converted to the widest type. For example, if one operand is of type `long double`, then the other operand is converted to type `long double` regardless of what the second type is. More generally, in expressions that mix floating-point and integral values, the integral value is converted to an appropriate floating-point type.

Integral Promotions

The **integral promotions** convert the small integral types to a larger integral type. The types `bool`, `char`, `signed char`, `unsigned char`, `short`, and `unsigned short` are promoted to `int` if all possible values of that type fit in an `int`. Otherwise, the value is promoted to `unsigned int`. As we've seen many times, a `bool` that is `false` promotes to 0 and `true` to 1.

The larger `char` types (`wchar_t`, `char16_t`, and `char32_t`) are promoted to the smallest type of `int`, `unsigned int`, `long`, `unsigned long`, `long long`, or `unsigned long long` in which all possible values of that character type fit.

Operands of Unsigned Type

If the operands of an operator have differing types, those operands are ordinarily converted to a common type. If any operand is an `unsigned` type, the type to which the operands are converted depends on the relative sizes of the integral types on the machine.

As usual, integral promotions happen first. If the resulting type(s) match, no further conversion is needed. If both (possibly promoted) operands have the same signedness, then the operand with the smaller type is converted to the larger type.

When the signedness differs and the type of the unsigned operand is the same as or larger than that of the signed operand, the signed operand is converted to unsigned. For example, given an `unsigned int` and an `int`, the `int` is converted to `unsigned int`. It is worth noting that if the `int` has a negative value, the result will be converted as described in § 2.1.2 (p. 35), with the same results.

The remaining case is when the signed operand has a larger type than the unsigned operand. In this case, the result is machine dependent. If all values in the unsigned type fit in the larger type, then the unsigned operand is converted to the signed type. If the values don't fit, then the signed operand is converted to the unsigned type. For example, if the operands are `long` and `unsigned int`, and `int` and `long` have the same size, the `long` will be converted to `unsigned int`. If the `long` type has more bits, then the `unsigned int` will be converted to `long`.

Understanding the Arithmetic Conversions

One way to understand the arithmetic conversions is to study lots of examples:

```
bool      flag;        char           cval;
short     sval;        unsigned short usval;
int       ival;        unsigned int   uival;
long      lval;        unsigned long  ulval;
float     fval;        double         dval;

3.14159L + 'a'; // 'a' promoted to int, then that int converted to long double
dval + ival;    // ival converted to double
dval + fval;    // fval converted to double
ival = dval;    // dval converted (by truncation) to int
flag = dval;    // if dval is 0, then flag is false, otherwise true
cval + fval;    // cval promoted to int, then that int converted to float
sval + cval;    // sval and cval promoted to int
```

```
cval + lval;    // cval converted to long
ival + ulval;   // ival converted to unsigned long
usval + ival;   // promotion depends on the size of unsigned short and int
uival + lval;   // conversion depends on the size of unsigned int and long
```

In the first addition, the character constant lowercase `'a'` has type `char`, which is a numeric value (§ 2.1.1, p. 32). What that value is depends on the machine's character set. On our machine, `'a'` has the numeric value 97. When we add `'a'` to a `long double`, the `char` value is promoted to `int`, and then that `int` value is converted to a `long double`. The converted value is added to the literal. The other interesting cases are the last two expressions involving unsigned values. The type of the result in these expressions is machine dependent.

Exercises Section 4.11.1

Exercise 4.34: Given the variable definitions in this section, explain what conversions take place in the following expressions:

(a) `if (fval)` (b) `dval = fval + ival;` (c) `dval + ival * cval;`

Remember that you may need to consider the associativity of the operators.

Exercise 4.35: Given the following definitions,

```
char cval;      int ival;     unsigned int ui;
float fval;     double dval;
```

identify the implicit type conversions, if any, taking place:

```
(a) cval = 'a' + 3;       (b) fval = ui - ival * 1.0;
(c) dval = ui * fval;     (d) cval = ival + fval + dval;
```

4.11.2 Other Implicit Conversions

In addition to the arithmetic conversions, there are several additional kinds of implicit conversions. These include:

Array to Pointer Conversions: In most expressions, when we use an array, the array is automatically converted to a pointer to the first element in that array:

```
int ia[10];     // array of ten ints
int* ip = ia;   // convert ia to a pointer to the first element
```

This conversion is not performed when an array is used with `decltype` or as the operand of the address-of (`&`), `sizeof`, or `typeid` (which we'll cover in § 19.2.2 (p. 826)) operators. The conversion is also omitted when we initialize a reference to an array (§ 3.5.1, p. 114). As we'll see in § 6.7 (p. 247), a similar pointer conversion happens when we use a function type in an expression.

Pointer Conversions: There are several other pointer conversions: A constant integral value of 0 and the literal `nullptr` can be converted to any pointer type; a pointer to any nonconst type can be converted to `void*`, and a pointer to any

type can be converted to a `const void*`. We'll see in § 15.2.2 (p. 597) that there is an additional pointer conversion that applies to types related by inheritance.

Conversions to `bool`: There is an automatic conversion from arithmetic or pointer types to `bool`. If the pointer or arithmetic value is zero, the conversion yields `false`; any other value yields `true`:

```
char *cp = get_string();
if (cp) /* ... */      // true if the pointer cp is not zero
while (*cp) /* ... */ // true if *cp is not the null character
```

Conversion to `const`: We can convert a pointer to a nonconst type to a pointer to the corresponding `const` type, and similarly for references. That is, if `T` is a type, we can convert a pointer or a reference to `T` into a pointer or reference to `const T`, respectively (§ 2.4.1, p. 61, and § 2.4.2, p. 62):

```
int i;
const int &j = i;  // convert a nonconst to a reference to const int
const int *p = &i; // convert address of a nonconst to the address of a const
int &r = j, *q = p; // error: conversion from const to nonconst not allowed
```

The reverse conversion—removing a low-level `const`—does not exist.

Conversions Defined by Class Types: Class types can define conversions that the compiler will apply automatically. The compiler will apply only one class-type conversion at a time. In § 7.5.4 (p. 295) we'll see an example of when multiple conversions might be required, and will be rejected.

Our programs have already used class-type conversions: We use a class-type conversion when we use a C-style character string where a library `string` is expected (§ 3.5.5, p. 124) and when we read from an `istream` in a condition:

```
string s, t = "a value";  // character string literal converted to type string
while (cin >> s)          // while condition converts cin to bool
```

The condition (`cin >> s`) reads `cin` and yields `cin` as its result. Conditions expect a value of type `bool`, but this condition tests a value of type `istream`. The IO library defines a conversion from `istream` to `bool`. That conversion is used (automatically) to convert `cin` to `bool`. The resulting `bool` value depends on the state of the stream. If the last read succeeded, then the conversion yields `true`. If the last attempt failed, then the conversion to `bool` yields `false`.

4.11.3 Explicit Conversions

Sometimes we want to explicitly force an object to be converted to a different type. For example, we might want to use floating-point division in the following code:

```
int i, j;
double slope = i/j;
```

To do so, we'd need a way to explicitly convert `i` and/or `j` to `double`. We use a **cast** to request an explicit conversion.

Although necessary at times, casts are inherently dangerous constructs.

Named Casts

A named cast has the following form:

```
cast-name<type>(expression);
```

where *type* is the target type of the conversion, and *expression* is the value to be cast. If *type* is a reference, then the result is an lvalue. The *cast-name* may be one of **`static_cast`**, **`dynamic_cast`**, **`const_cast`**, and **`reinterpret_cast`**. We'll cover `dynamic_cast`, which supports the run-time type identification, in § 19.2 (p. 825). The *cast-name* determines what kind of conversion is performed.

`static_cast`

Any well-defined type conversion, other than those involving low-level `const`, can be requested using a `static_cast`. For example, we can force our expression to use floating-point division by casting one of the operands to `double`:

```
// cast used to force floating-point division
double slope = static_cast<double>(j) / i;
```

A `static_cast` is often useful when a larger arithmetic type is assigned to a smaller type. The cast informs both the reader of the program and the compiler that we are aware of and are not concerned about the potential loss of precision. Compilers often generate a warning for assignments of a larger arithmetic type to a smaller type. When we do an explicit cast, the warning message is turned off.

A `static_cast` is also useful to perform a conversion that the compiler will not generate automatically. For example, we can use a `static_cast` to retrieve a pointer value that was stored in a `void*` pointer (§ 2.3.2, p. 56):

```
void* p = &d;    // ok: address of any nonconst object can be stored in a void*
// ok: converts void* back to the original pointer type
double *dp = static_cast<double*>(p);
```

When we store a pointer in a `void*` and then use a `static_cast` to cast the pointer back to its original type, we are guaranteed that the pointer value is preserved. That is, the result of the cast will be equal to the original address value. However, we must be certain that the type to which we cast the pointer is the actual type of that pointer; if the types do not match, the result is undefined.

`const_cast`

A `const_cast` changes only a low-level (§ 2.4.3, p. 63) `const` in its operand:

```
const char *pc;
char *p = const_cast<char*>(pc); // ok: but writing through p is undefined
```

Conventionally we say that a cast that converts a `const` object to a nonconst type "casts away the `const`." Once we have cast away the `const` of an object, the compiler will no longer prevent us from writing to that object. If the object was originally not a `const`, using a cast to obtain write access is legal. However, using a `const_cast` in order to write to a `const` object is undefined.

Only a `const_cast` may be used to change the `const`ness of an expression. Trying to change whether an expression is `const` with any of the other forms of named cast is a compile-time error. Similarly, we cannot use a `const_cast` to change the type of an expression:

```
const char *cp;
//  error: static_cast can't cast away const
char *q = static_cast<char*>(cp);
static_cast<string>(cp); //  ok: converts string literal to string
const_cast<string>(cp);  //  error: const_cast only changes constness
```

A `const_cast` is most useful in the context of overloaded functions, which we'll describe in § 6.4 (p. 232).

`reinterpret_cast`

A `reinterpret_cast` generally performs a low-level reinterpretation of the bit pattern of its operands. As an example, given the following cast

```
int *ip;
char *pc = reinterpret_cast<char*>(ip);
```

we must never forget that the actual object addressed by `pc` is an `int`, not a character. Any use of `pc` that assumes it's an ordinary character pointer is likely to fail at run time. For example:

```
string str(pc);
```

is likely to result in bizarre run-time behavior.

The use of `pc` to initialize `str` is a good example of why `reinterpret_cast` is dangerous. The problem is that types are changed, yet there are no warnings or errors from the compiler. When we initialized `pc` with the address of an `int`, there is no error or warning from the compiler because we explicitly said the conversion was okay. Any subsequent use of `pc` will assume that the value it holds is a `char*`. The compiler has no way of knowing that it actually holds a pointer to an `int`. Thus, the initialization of `str` with `pc` is absolutely correct—albeit in this case meaningless or worse! Tracking down the cause of this sort of problem can prove extremely difficult, especially if the cast of `ip` to `pc` occurs in a file separate from the one in which `pc` is used to initialize a `string`.

A `reinterpret_cast` is inherently machine dependent. Safely using `reinterpret_cast` requires completely understanding the types involved as well as the details of how the compiler implements the cast.

Old-Style Casts

In early versions of C++, an explicit cast took one of the following two forms:

```
type  (expr);  // function-style cast notation
(type)  expr;  // C-language-style cast notation
```

ADVICE: AVOID CASTS

Casts interfere with normal type checking (§ 2.2.2, p. 46). As a result, we strongly recommend that programmers avoid casts. This advice is particularly applicable to `reinterpret_cast`s. Such casts are always hazardous. A `const_cast` can be useful in the context of overloaded functions, which we'll cover in § 6.4 (p. 232). Other uses of `const_cast` often indicate a design flaw. The other casts, `static_cast` and `dynamic_cast`, should be needed infrequently. Every time you write a cast, you should think hard about whether you can achieve the same result in a different way. If the cast is unavoidable, errors can be mitigated by limiting the scope in which the cast value is used and by documenting all assumptions about the types involved.

Depending on the types involved, an old-style cast has the same behavior as a `const_cast`, a `static_cast`, or a `reinterpret_cast`. When we use an old-style cast where a `static_cast` or a `const_cast` would be legal, the old-style cast does the same conversion as the respective named cast. If neither cast is legal, then an old-style cast performs a `reinterpret_cast`. For example:

```
char *pc = (char*) ip;  // ip is a pointer to int
```

has the same effect as using a `reinterpret_cast`.

Old-style casts are less visible than are named casts. Because they are easily overlooked, it is more difficult to track down a rogue cast.

EXERCISES SECTION 4.11.3

Exercise 4.36: Assuming `i` is an `int` and `d` is a `double` write the expression `i *= d` so that it does integral, rather than floating-point, multiplication.

Exercise 4.37: Rewrite each of the following old-style casts to use a named cast:

```
int i;  double d;  const string *ps;  char *pc;  void *pv;
(a) pv = (void*)ps;    (b) i = int(*pc);
(c) pv = &d;           (d) pc = (char*) pv;
```

Exercise 4.38: Explain the following expression:

```
double slope = static_cast<double>(j/i);
```

4.12 Operator Precedence Table

Associativity and Operator		Function	Use	See Page
L	`::`	global scope	`::`name	286
L	`::`	class scope	class`::`name	88
L	`::`	namespace scope	namespace`::`name	82
L	`.`	member selectors	object`.`member	23
L	`->`	member selectors	pointer`->`member	110
L	`[]`	subscript	expr`[` expr `]`	116
L	`()`	function call	name`(`expr_list`)`	23
L	`()`	type construction	type`(`expr_list`)`	164
R	`++`	postfix increment	lvalue`++`	147
R	`--`	postfix decrement	lvalue`--`	147
R	`typeid`	type ID	`typeid(`type`)`	826
R	`typeid`	run-time type ID	`typeid(`expr`)`	826
R	explicit cast	type conversion	*cast_name*<type>(expr)	162
R	`++`	prefix increment	`++`lvalue	147
R	`--`	prefix decrement	`--`lvalue	147
R	`~`	bitwise NOT	`~`expr	152
R	`!`	logical NOT	`!`expr	141
R	`-`	unary minus	`-`expr	140
R	`+`	unary plus	`+`expr	140
R	`*`	dereference	`*`expr	53
R	`&`	address-of	`&`lvalue	52
R	`()`	type conversion	(type) expr	164
R	`sizeof`	size of object	`sizeof` expr	156
R	`sizeof`	size of type	`sizeof(` type `)`	156
R	`sizeof...`	size of parameter pack	`sizeof...(` name `)`	700
R	`new`	allocate object	`new` type	458
R	`new[]`	allocate array	`new` type[size]	458
R	`delete`	deallocate object	`delete` expr	460
R	`delete[]`	deallocate array	`delete[]` expr	460
R	`noexcept`	can expr throw	`noexcept (` expr `)`	780
L	`->*`	ptr to member select	ptr`->*`ptr_to_member	837
L	`.*`	ptr to member select	obj`.*`ptr_to_member	837
L	`*`	multiply	expr `*` expr	139
L	`/`	divide	expr `/` expr	139
L	`%`	modulo (remainder)	expr `%` expr	139
L	`+`	add	expr `+` expr	139
L	`-`	subtract	expr `-` expr	139
L	`<<`	bitwise shift left	expr `<<` expr	152
L	`>>`	bitwise shift right	expr `>>` expr	152
L	`<`	less than	expr `<` expr	141
L	`<=`	less than or equal	expr `<=` expr	141
L	`>`	greater than	expr `>` expr	141

Continued on next page

Table 4.4: Operator Precedence (continued)

Associativity and Operator		Function	Use	See Page
L	`>=`	greater than or equal	expr `>=` expr	141
L	`==`	equality	expr `==` expr	141
L	`!=`	inequality	expr `!=` expr	141
L	`&`	bitwise AND	expr `&` expr	152
L	`^`	bitwise XOR	expr `^` expr	152
L	`|`	bitwise OR	expr `|` expr	152
L	`&&`	logical AND	expr `&&` expr	141
L	`||`	logical OR	expr `||` expr	141
R	`?:`	conditional	expr `?` expr `:` expr	151
R	`=`	assignment	lvalue `=` expr	144
R	`*=, /=, %=,`	compound assign	lvalue `+=` expr, etc.	144
R	`+=, -=,`			144
R	`<<=, >>=,`			144
R	`&=, |=, ^=`			144
R	`throw`	throw exception	throw expr	193
L	`,`	comma	expr , expr	157

Chapter Summary

C++ provides a rich set of operators and defines their meaning when applied to values of the built-in types. Additionally, the language supports operator overloading, which allows us to define the meaning of the operators for class types. We'll see in Chapter 14 how to define operators for our own types.

To understand expressions involving more than one operator it is necessary to understand precedence, associativity, and order of operand evaluation. Each operator has a precedence level and associativity. Precedence determines how operators are grouped in a compound expression. Associativity determines how operators at the same precedence level are grouped.

Most operators do not specify the order in which operands are evaluated: The compiler is free to evaluate either the left- or right-hand operand first. Often, the order of operand evaluation has no impact on the result of the expression. However, if both operands refer to the same object and one of the operands *changes* that object, then the program has a serious bug—and a bug that may be hard to find.

Finally, operands are often converted automatically from their initial type to another related type. For example, small integral types are promoted to a larger integral type in every expression. Conversions exist for both built-in and class types. Conversions can also be done explicitly through a cast.

Defined Terms

arithmetic conversion A conversion from one arithmetic type to another. In the context of the binary arithmetic operators, arithmetic conversions usually attempt to preserve precision by converting a smaller type to a larger type (e.g., integral types are converted to floating point).

associativity Determines how operators with the same precedence are grouped. Operators can be either right associative (operators are grouped from right to left) or left associative (operators are grouped from left to right).

binary operators Operators that take two operands.

cast An explicit conversion.

compound expression An expression involving more than one operator.

const_cast A cast that converts a low-level `const` object to the corresponding non`const` type or vice versa.

conversion Process whereby a value of one type is transformed into a value of another type. The language defines conversions among the built-in types. Conversions to and from class types are also possible.

dynamic_cast Used in combination with inheritance and run-time type identification. See § 19.2 (p. 825).

expression The lowest level of computation in a C++ program. Expressions generally apply an operator to one or more operands. Each expression yields a result. Expressions can be used as operands, so we can write compound expressions requiring the evaluation of multiple operators.

implicit conversion A conversion that is automatically generated by the compiler. Given an expression that needs a particular type but has an operand of a differing type, the compiler will automatically convert the operand to the desired type if an appropriate conversion exists.

integral promotions conversions that take a smaller integral type to its most closely related larger integral type. Operands of small integral types (e.g., `short`, `char`, etc.) are always promoted, even in contexts where such conversions might not seem to be required.

lvalue An expression that yields an object or function. A nonconst lvalue that denotes an object may be the left-hand operand of assignment.

operands Values on which an expression operates. Each operator has one or more operands associated with it.

operator Symbol that determines what action an expression performs. The language defines a set of operators and what those operators mean when applied to values of built-in type. The language also defines the precedence and associativity of each operator and specifies how many operands each operator takes. Operators may be overloaded and applied to values of class type.

order of evaluation Order, if any, in which the operands to an operator are evaluated. In most cases, the compiler is free to evaluate operands in any order. However, the operands are always evaluated before the operator itself is evaluated. Only the `&&`, `||`, `?:`, and comma operators specify the order in which their operands are evaluated.

overloaded operator Version of an operator that is defined for use with a class type. We'll see in Chapter 14 how to define overloaded versions of operators.

precedence Defines the order in which different operators in a compound expression are grouped. Operators with higher precedence are grouped more tightly than operators with lower precedence.

promoted See integral promotions.

reinterpret_cast Interprets the contents of the operand as a different type. Inherently machine dependent and dangerous.

result Value or object obtained by evaluating an expression.

rvalue Expression that yields a value but not the associated location, if any, of that value.

short-circuit evaluation Term used to describe how the logical AND and logical OR operators execute. If the first operand to these operators is sufficient to determine the overall result, evaluation stops. We are guaranteed that the second operand is not evaluated.

sizeof Operator that returns the size, in bytes, to store an object of a given type name or of the type of a given expression.

static_cast An explicit request for a well-defined type conversion. Often used to override an implicit conversion that the compiler would otherwise perform.

unary operators Operators that take a single operand.

`,` operator Comma operator. Binary operator that is evaluated left to right. The result of a comma expression is the value of the right-hand operand. The result is an lvalue if and only if that operand is an lvalue.

`?:` operator Conditional operator. Provides an if-then-else expression of the form

```
cond ? expr1 : expr2;
```

If the condition *cond* is true, then *expr1* is evaluated. Otherwise, *expr2* is evaluated. The type *expr1* and *expr2* must be the same type or be convertible to a common type. Only one of *expr1* or *expr2* is evaluated.

`&&` operator Logical AND operator. Result is `true` if both operands are `true`. The right-hand operand is evaluated *only* if the left-hand operand is `true`.

`&` operator Bitwise AND operator. Generates a new integral value in which each bit position is 1 if both operands have a 1 in that position; otherwise the bit is 0.

^ operator Bitwise exclusive or operator. Generates a new integral value in which each bit position is 1 if either but not both operands contain a 1 in that bit position; otherwise, the bit is 0.

|| operator Logical OR operator. Yields `true` if either operand is `true`. The right-hand operand is evaluated *only* if the left-hand operand is `false`.

| operator Bitwise OR operator. Generates a new integral value in which each bit position is 1 if either operand has a 1 in that position; otherwise the bit is 0.

++ operator The increment operator. The increment operator has two forms, prefix and postfix. Prefix increment yields an lvalue. It adds 1 to the operand and returns the changed value of the operand. Postfix increment yields an rvalue. It adds 1 to the operand and returns a copy of the original, unchanged value of the operand. Note: Iterators have ++ even if they do not have the + operator.

-- operator The decrement operator has two forms, prefix and postfix. Prefix decrement yields an lvalue. It subtracts 1 from the operand and returns the changed value of the operand. Postfix decrement yields an rvalue. It subtracts 1 from the operand and returns a copy of the original, unchanged value of the operand. Note: Iterators have `--` even if they do not have the `-`.

<< operator The left-shift operator. Shifts bits in a (possibly promoted) copy of the value of the left-hand operand to the left. Shifts as many bits as indicated by the right-hand operand. The right-hand operand must be zero or positive and strictly less than the number of bits in the result. Left-hand operand should be `unsigned`; if the left-hand operand is `signed`, it is undefined if a shift causes a different bit to shift into the sign bit.

>> operator The right-shift operator. Like the left-shift operator except that bits are shifted to the right. If the left-hand operand is `signed`, it is implementation defined whether bits shifted into the result are 0 or a copy of the sign bit.

~ operator Bitwise NOT operator. Generates a new integral value in which each bit is an inverted copy of the corresponding bit in the (possibly promoted) operand.

! operator Logical NOT operator. Returns the inverse of the `bool` value of its operand. Result is `true` if operand is `false` and vice versa.

CHAPTER 5

STATEMENTS

CONTENTS

Like most languages, C++ provides statements for conditional execution, loops that repeatedly execute the same body of code, and jump statements that interrupt the flow of control. This chapter looks in detail at the statements supported by C++.

Statements are executed sequentially. Except for the simplest programs, sequential execution is inadequate. Therefore, C++ also defines a set of ***flow-of-control*** statements that allow more complicated execution paths.

5.1 Simple Statements

Most statements in C++ end with a semicolon. An expression, such as `ival + 5`, becomes an **expression statement** when it is followed by a semicolon. Expression statements cause the expression to be evaluated and its result discarded:

```
ival + 5;       // rather useless expression statement
cout << ival;   // useful expression statement
```

The first statement is pretty useless: The addition is done but the result is not used. More commonly, an expression statement contains an expression that has a side effect—such as assigning a new value to a variable, or printing a result—when it is evaluated.

Null Statements

The simplest statement is the empty statement, also known as a **null statement**. A null statement is a single semicolon:

```
;  // null statement
```

A null statement is useful where the language requires a statement but the program's logic does not. Such usage is most common when a loop's work can be done within its condition. For example, we might want to read an input stream, ignoring everything we read until we encounter a particular value:

```
// read until we hit end-of-file or find an input equal to sought
while (cin >> s && s != sought)
    ; // null statement
```

This condition reads a value from the standard input and implicitly tests `cin` to see whether the read was successful. Assuming the read succeeded, the second part of the condition tests whether the value we read is equal to the value in `sought`. If we found the value we want, the `while` loop is exited. Otherwise, the condition is evaluated again, which reads another value from `cin`.

Null statements should be commented. That way anyone reading the code can see that the statement was omitted intentionally.

Beware of Missing or Extraneous Semicolons

Because a null statement is a statement, it is legal anywhere a statement is expected. For this reason, semicolons that might appear illegal are often nothing more than null statements. The following fragment contains two statements—the expression statement and the null statement:

```
ival = v1 + v2;; // ok: second semicolon is a superfluous null statement
```

Although an unnecessary null statement is often harmless, an extra semicolon following the condition in a `while` or `if` can drastically alter the programmer's intent. For example, the following code will loop indefinitely:

```
// disaster: extra semicolon: loop body is this null statement
while (iter != svec.end()) ; // the while body is the empty statement
    ++iter;      // increment is not part of the loop
```

Contrary to the indentation, the increment is not part of the loop. The loop body is the null statement formed by the semicolon that follows the condition.

Extraneous null statements are not always harmless.

Compound Statements (Blocks)

A **compound statement**, usually referred to as a **block**, is a (possibly empty) sequence of statements and declarations surrounded by a pair of curly braces. A block is a scope (§ 2.2.4, p. 48). Names introduced inside a block are accessible only in that block and in blocks nested inside that block. Names are visible from where they are defined until the end of the (immediately) enclosing block.

Compound statements are used when the language requires a single statement but the logic of our program needs more than one. For example, the body of a `while` or `for` loop must be a single statement, yet we often need to execute more than one statement in the body of a loop. We do so by enclosing the statements in curly braces, thus turning the sequence of statements into a block.

As one example, recall the `while` loop in the program in § 1.4.1 (p. 11):

```
while (val <= 10) {
    sum += val;  // assigns sum + val to sum
    ++val;       // add 1 to val
}
```

The logic of our program needed two statements but a `while` loop may contain only one statement. By enclosing these statements in curly braces, we made them into a single (compound) statement.

A block is *not* terminated by a semicolon.

We also can define an empty block by writing a pair of curlies with no statements. An empty block is equivalent to a null statement:

```
while (cin >> s && s != sought)
    { } // empty block
```

Exercises Section 5.1

Exercise 5.1: What is a null statement? When might you use a null statement?

Exercise 5.2: What is a block? When might you might use a block?

Exercise 5.3: Use the comma operator (§ 4.10, p. 157) to rewrite the `while` loop from § 1.4.1 (p. 11) so that it no longer requires a block. Explain whether this rewrite improves or diminishes the readability of this code.

5.2 Statement Scope

We can define variables inside the control structure of the `if`, `switch`, `while`, and `for` statements. Variables defined in the control structure are visible only within that statement and are out of scope after the statement ends:

```
while (int i = get_num()) //  i is created and initialized on each iteration
    cout << i << endl;
i = 0;  //  error: i is not accessible outside the loop
```

If we need access to the control variable, then that variable must be defined outside the statement:

```
//  find the first negative element
auto beg = v.begin();
while (beg != v.end() && *beg >= 0)
    ++beg;
if (beg == v.end())
    //  we know that all elements in v are greater than or equal to zero
```

The value of an object defined in a control structure is used by that structure. Therefore, such variables must be initialized.

Exercises Section 5.2

Exercise 5.4: Explain each of the following examples, and correct any problems you detect.

(a)
```
while (string::iterator iter != s.end()) { /* ... */ }
```
(b)
```
while (bool status = find(word)) { /* ... */ }
    if (!status) { /* ... */ }
```

5.3 Conditional Statements

C++ provides two statements that allow for conditional execution. The `if` statement determines the flow of control based on a condition. The `switch` statement evaluates an integral expression and chooses one of several execution paths based on the expression's value.

5.3.1 The `if` Statement

An **`if` statement** conditionally executes another statement based on whether a specified condition is true. There are two forms of the `if`: one with an `else` branch and one without. The syntactic form of the simple `if` is

```
if (condition)
    statement
```

An **`if else` statement** has the form

```
if (condition)
    statement
else
    statement2
```

In both versions, *condition* must be enclosed in parentheses. *condition* can be an expression or an initialized variable declaration (§ 5.2, p. 174). The expression or variable must have a type that is convertible (§ 4.11, p. 159) to `bool`. As usual, either or both *statement* and *statement2* can be a block.

If *condition* is `true`, then *statement* is executed. After *statement* completes, execution continues with the statement following the `if`.

If *condition* is `false`, *statement* is skipped. In a simple `if`, execution continues with the statement following the `if`. In an `if else`, *statement2* is executed.

Using an `if else` Statement

To illustrate an `if` statement, we'll calculate a letter grade from a numeric grade. We'll assume that the numeric grades range from zero to 100 inclusive. A grade of 100 gets an "A++," grades below 60 get an "F," and the others range in clumps of ten: grades from 60 to 69 inclusive get a "D," 70 to 79 a "C," and so on. We'll use a `vector` to hold the possible letter grades:

```
const vector<string> scores = {"F", "D", "C", "B", "A", "A++"};
```

To solve this problem, we can use an `if else` statement to execute different actions for failing and passing grades:

```
// if grade is less than 60 it's an F, otherwise compute a subscript
string lettergrade;
if (grade < 60)
    lettergrade = scores[0];
else
    lettergrade = scores[(grade - 50)/10];
```

Depending on the value of `grade`, we execute the statement after the `if` or the one after the `else`. In the `else`, we compute a subscript from a grade by reducing the grade to account for the larger range of failing grades. Then we use integer division (§ 4.2, p. 141), which truncates the remainder, to calculate the appropriate `scores` index.

Nested `if` Statements

To make our program more interesting, we'll add a plus or minus to passing grades. We'll give a plus to grades ending in 8 or 9, and a minus to those ending in 0, 1, or 2:

```
if (grade % 10 > 7)
    lettergrade += '+';     // grades ending in 8 or 9 get a +
else if (grade % 10 < 3)
    lettergrade += '-';     // those ending in 0, 1, or 2 get a -
```

Here we use the modulus operator (§ 4.2, p. 141) to get the remainder and decide based on the remainder whether to add plus or minus.

We next will incorporate the code that adds a plus or minus to the code that fetches the letter grade from scores:

```
// if failing grade, no need to check for a plus or minus
if (grade < 60)
    lettergrade = scores[0];
else {
    lettergrade = scores[(grade - 50)/10];   // fetch the letter grade
    if (grade != 100)  // add plus or minus only if not already an A++
        if (grade % 10 > 7)
            lettergrade += '+';    // grades ending in 8 or 9 get a +
        else if (grade % 10 < 3)
            lettergrade += '-';    // grades ending in 0, 1, or 2 get a -
}
```

Note that we use a block to enclose the two statements that follow the first `else`. If the `grade` is `60` or more, we have two actions that we need to do: Fetch the letter grade from `scores`, and conditionally set the plus or minus.

Watch Your Braces

It is a common mistake to forget the curly braces when multiple statements must be executed as a block. In the following example, contrary to the indentation, the code to add a plus or minus happens unconditionally:

```
if (grade < 60)
    lettergrade = scores[0];
else  // WRONG: missing curly
    lettergrade = scores[(grade - 50)/10];
    // despite appearances, without the curly brace, this code is always executed
    // failing grades will incorrectly get a - or a +
    if (grade != 100)
        if (grade % 10 > 7)
            lettergrade += '+';    // grades ending in 8 or 9 get a +
        else if (grade % 10 < 3)
            lettergrade += '-';    // grades ending in 0, 1, or 2 get a -
```

Uncovering this error may be very difficult because the program looks correct.

To avoid such problems, some coding styles recommend always using braces after an `if` or an `else` (and also around the bodies of `while` and `for` statements).

Doing so avoids any possible confusion. It also means that the braces are already in place if later modifications of the code require adding statements.

Many editors and development environments have tools to automatically indent source code to match its structure. It is a good idea to use such tools if they are available.

Dangling `else`

When we nest an `if` inside another `if`, it is possible that there will be more `if` branches than `else` branches. Indeed, our grading program has four `if`s and two `else`s. The question arises: How do we know to which `if` a given `else` belongs?

This problem, usually referred to as a **dangling `else`**, is common to many programming languages that have both `if` and `if else` statements. Different languages solve this problem in different ways. In C++ the ambiguity is resolved by specifying that each `else` is matched with the closest preceding unmatched `if`.

Programmers sometimes get into trouble when they write code that contains more `if` than `else` branches. To illustrate the problem, we'll rewrite the innermost `if else` that adds a plus or minus using a different set of conditions:

```
// WRONG: execution does NOT match indentation; the else goes with the inner if
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+';  // grades ending in 8 or 9 get a +
else
    lettergrade += '-'; // grades ending in 3, 4, 5, 6, or 7 get a minus!
```

The indentation in our code indicates that we intend the `else` to go with the outer `if`—we intend for the `else` branch to be executed when the `grade` ends in a digit less than `3`. However, despite our intentions, and contrary to the indentation, the `else` branch is part of the inner `if`. This code adds a `'-'` to grades ending in `3` to `7` inclusive! Properly indented to match the actual execution, what we wrote is:

```
// indentation matches the execution path, not the programmer's intent
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+';  // grades ending in 8 or 9 get a +
    else
        lettergrade += '-';  // grades ending in 3, 4, 5, 6, or 7 get a minus!
```

Controlling the Execution Path with Braces

We can make the `else` part of the outer `if` by enclosing the inner `if` in a block:

```
// add a plus for grades that end in 8 or 9 and a minus for those ending in 0, 1, or 2
if (grade % 10 >= 3) {
    if (grade % 10 > 7)
        lettergrade += '+';  // grades ending in 8 or 9 get a +
} else                   // curlies force the else to go with the outer if
    lettergrade += '-'; // grades ending in 0, 1, or 2 will get a minus
```

Statements do not span block boundaries, so the inner `if` ends at the close curly before the `else`. The `else` cannot be part of the inner `if`. Now, the nearest unmatched `if` is the outer `if`, which is what we intended all along.

EXERCISES SECTION 5.3.1

Exercise 5.5: Using an `if–else` statement, write your own version of the program to generate the letter grade from a numeric grade.

Exercise 5.6: Rewrite your grading program to use the conditional operator (§ 4.7, p. 151) in place of the `if–else` statement.

Exercise 5.7: Correct the errors in each of the following code fragments:

```
(a) if (ival1 != ival2)
        ival1 = ival2
    else ival1 = ival2 = 0;
(b) if (ival < minval)
        minval = ival;
        occurs = 1;
(c) if (int ival = get_value())
        cout << "ival = " << ival << endl;
    if (!ival)
        cout << "ival = 0\n";
(d) if (ival = 0)
        ival = get_value();
```

Exercise 5.8: What is a "dangling `else`"? How are `else` clauses resolved in C++?

5.3.2 The `switch` Statement

A **`switch` statement** provides a convenient way of selecting among a (possibly large) number of fixed alternatives. As one example, suppose that we want to count how often each of the five vowels appears in some segment of text. Our program logic is as follows:

- Read every character in the input.
- Compare each character to the set of vowels.
- If the character matches one of the vowels, add 1 to that vowel's count.
- Display the results.

For example, when we run the program on the text of this chapter, the output is

```
Number of vowel a:  3195
Number of vowel e:  6230
Number of vowel i:  3102
Number of vowel o:  3289
Number of vowel u:  1033
```

We can solve our problem most directly using a `switch` statement:

```
// initialize counters for each vowel
unsigned aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0;
char ch;
while (cin >> ch) {
    // if ch is a vowel, increment the appropriate counter
    switch (ch) {
        case 'a':
            ++aCnt;
            break;
        case 'e':
            ++eCnt;
            break;
        case 'i':
            ++iCnt;
            break;
        case 'o':
            ++oCnt;
            break;
        case 'u':
            ++uCnt;
            break;
    }
}
// print results
cout << "Number of vowel a: \t" << aCnt << '\n'
     << "Number of vowel e: \t" << eCnt << '\n'
     << "Number of vowel i: \t" << iCnt << '\n'
     << "Number of vowel o: \t" << oCnt << '\n'
     << "Number of vowel u: \t" << uCnt << endl;
```

A `switch` statement executes by evaluating the parenthesized expression that follows the keyword `switch`. That expression may be an initialized variable declaration (§ 5.2, p. 174). The expression is converted to integral type. The result of the expression is compared with the value associated with each `case`.

If the expression matches the value of a `case` label, execution begins with the first statement following that label. Execution continues normally from that statement through the end of the `switch` or until a `break` statement.

We'll look at `break` statements in detail in § 5.5.1 (p. 190), but, briefly, a `break` interrupts the current control flow. In this case, the `break` transfers control out of the `switch`. In this program, the `switch` is the only statement in the body of a `while`. Breaking out of this `switch` returns control to the enclosing `while`. Because there are no other statements in that `while`, execution continues at the condition in the `while`.

If no match is found, execution falls through to the first statement following the `switch`. As we already know, in this example, exiting the `switch` returns control to the condition in the `while`.

The `case` keyword and its associated value together are known as the **`case` label**. `case` labels must be integral constant expressions (§ 2.4.4, p. 65):

```
char ch = getVal();
int ival = 42;
switch(ch) {
    case 3.14: //  error: noninteger as case label
    case ival: //  error: nonconstant as case label
    // ...
```

It is an error for any two `case` labels to have the same value. There is also a special-case label, `default`, which we cover on page 181.

Control Flow within a `switch`

It is important to understand that execution flows across `case` labels. After a `case` label is matched, execution starts at that label and continues across all the remaining `case`s or until the program explicitly interrupts it. To avoid executing code for subsequent `case`s, we must explicitly tell the compiler to stop execution. Under most conditions, the last statement before the next `case` label is `break`.

However, there are situations where the default `switch` behavior is exactly what is needed. Each `case` label can have only a single value, but sometimes we have two or more values that share a common set of actions. In such instances, we omit a `break` statement, allowing the program to *fall through* multiple `case` labels.

For example, we might want to count only the total number of vowels:

```
unsigned vowelCnt = 0;
// ...
switch (ch)
{
    //  any occurrence of a, e, i, o, or u increments vowelCnt
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
       ++vowelCnt;
       break;
}
```

Here we stacked several `case` labels together with no intervening `break`. The same code will be executed whenever `ch` is a vowel.

Because C++ programs are free-form, `case` labels need not appear on a new line. We can emphasize that the `case`s represent a range of values by listing them all on a single line:

```
switch (ch)
{
    //  alternative legal syntax
    case 'a': case 'e': case 'i': case 'o': case 'u':
       ++vowelCnt;
       break;
}
```

Omitting a `break` at the end of a `case` happens rarely. If you do omit a `break`, include a comment explaining the logic.

Forgetting a `break` Is a Common Source of Bugs

It is a common misconception to think that only the statements associated with the matched `case` label are executed. For example, here is an *incorrect* implementation of our vowel-counting `switch` statement:

```
// warning: deliberately incorrect!
switch (ch) {
    case 'a':
        ++aCnt;    // oops: should have a break statement
    case 'e':
        ++eCnt;    // oops: should have a break statement
    case 'i':
        ++iCnt;    // oops: should have a break statement
    case 'o':
        ++oCnt;    // oops: should have a break statement
    case 'u':
        ++uCnt;
}
```

To understand what happens, assume that the value of `ch` is `'e'`. Execution jumps to the code following the `case 'e'` label, which increments `eCnt`. Execution *continues* across the `case` labels, incrementing `iCnt`, `oCnt`, and `uCnt` as well.

Although it is not necessary to include a `break` after the last label of a `switch`, the safest course is to provide one. That way, if an additional `case` is added later, the `break` is already in place.

The `default` Label

The statements following the **`default` label** are executed when no `case` label matches the value of the `switch` expression. For example, we might add a counter to track how many nonvowels we read. We'll increment this counter, which we'll name `otherCnt`, in the `default` case:

```
// if ch is a vowel, increment the appropriate counter
switch (ch) {
    case 'a': case 'e': case 'i': case 'o': case 'u':
        ++vowelCnt;
        break;
    default:
        ++otherCnt;
        break;
}
```

In this version, if `ch` is not a vowel, execution will start at the `default` label and we'll increment `otherCnt`.

It can be useful to define a `default` label even if there is no work for the `default` case. Defining an empty `default` section indicates to subsequent readers that the case was considered.

A label may not stand alone; it must precede a statement or another `case` label. If a `switch` ends with a `default` case that has no work to do, then the `default` label must be followed by a null statement or an empty block.

Variable Definitions inside the Body of a `switch`

As we've seen, execution in a `switch` can jump across `case` labels. When execution jumps to a particular `case`, any code that occurred inside the `switch` before that label is ignored. The fact that code is bypassed raises an interesting question: What happens if the code that is skipped includes a variable definition?

The answer is that it is illegal to jump from a place where a variable with an initializer is out of scope to a place where that variable is in scope:

```
case true:
    // this switch statement is illegal because these initializations might be bypassed
    string file_name; // error: control bypasses an implicitly initialized variable
    int ival = 0;     // error: control bypasses an explicitly initialized variable
    int jval;         // ok: because jval is not initialized
    break;
case false:
    // ok: jval is in scope but is uninitialized
    jval = next_num(); // ok: assign a value to jval
    if (file_name.empty()) // file_name is in scope but wasn't initialized
        // ...
```

If this code were legal, then any time control jumped to the `false` case, it would bypass the initialization of `file_name` and `ival`. Those variables would be in scope. Code following `false` could use those variables. However, these variables would not have been initialized. As a result, the language does not allow us to jump over an initialization if the initialized variable is in scope at the point to which control transfers.

If we need to define and initialize a variable for a particular `case`, we can do so by defining the variable inside a block, thereby ensuring that the variable is out of scope at the point of any subsequent label.

```
case true:
    {
       // ok: declaration statement within a statement block
       string file_name = get_file_name();
       // ...
    }
    break;
case false:
       if (file_name.empty())  // error: file_name is not in scope
```

EXERCISES SECTION 5.3.2

Exercise 5.9: Write a program using a series of `if` statements to count the number of vowels in text read from `cin`.

Exercise 5.10: There is one problem with our vowel-counting program as we've implemented it: It doesn't count capital letters as vowels. Write a program that counts both lower- and uppercase letters as the appropriate vowel—that is, your program should count both `'a'` and `'A'` as part of `aCnt`, and so forth.

Exercise 5.11: Modify our vowel-counting program so that it also counts the number of blank spaces, tabs, and newlines read.

Exercise 5.12: Modify our vowel-counting program so that it counts the number of occurrences of the following two-character sequences: `ff`, `fl`, and `fi`.

Exercise 5.13: Each of the programs in the highlighted text on page 184 contains a common programming error. Identify and correct each error.

5.4 Iterative Statements

Iterative statements, commonly called loops, provide for repeated execution until a condition is true. The `while` and `for` statements test the condition before executing the body. The `do while` executes the body and then tests its condition.

5.4.1 The `while` Statement

A **`while` statement** repeatedly executes a target statement as long as a condition is true. Its syntactic form is

```
while (condition)
      statement
```

In a `while`, *statement* (which is often a block) is executed as long as *condition* evaluates as `true`. *condition* may not be empty. If the first evaluation of *condition* yields `false`, *statement* is not executed.

The condition can be an expression or an initialized variable declaration (§ 5.2, p. 174). Ordinarily, the condition itself or the loop body must do something to change the value of the expression. Otherwise, the loop might never terminate.

Variables defined in a `while` condition or `while` body are created and destroyed on each iteration.

Using a `while` Loop

A `while` loop is generally used when we want to iterate indefinitely, such as when we read input. A `while` is also useful when we want access to the value of the loop control variable after the loop finishes. For example:

CODE FOR EXERCISE 5.13

(a)
```
unsigned aCnt = 0, eCnt = 0, iouCnt = 0;
char ch = next_text();
switch (ch) {
    case 'a': aCnt++;
    case 'e': eCnt++;
    default: iouCnt++;
}
```

(b)
```
unsigned index = some_value();
switch (index) {
    case 1:
        int ix = get_value();
        ivec[ ix ] = index;
        break;
    default:
        ix = ivec.size()-1;
        ivec[ ix ] = index;
}
```

(c)
```
unsigned evenCnt = 0, oddCnt = 0;
int digit = get_num() % 10;
switch (digit) {
    case 1, 3, 5, 7, 9:
        oddcnt++;
        break;
    case 2, 4, 6, 8, 10:
        evencnt++;
        break;
}
```

(d)
```
unsigned ival=512, jval=1024, kval=4096;
unsigned bufsize;
unsigned swt = get_bufCnt();
switch(swt) {
    case ival:
        bufsize = ival * sizeof(int);
        break;
    case jval:
        bufsize = jval * sizeof(int);
        break;
    case kval:
        bufsize = kval * sizeof(int);
        break;
}
```

```
vector<int> v;
int i;
// read until end-of-file or other input failure
while (cin >> i)
    v.push_back(i);
// find the first negative element
auto beg = v.begin();
while (beg != v.end() && *beg >= 0)
    ++beg;
if (beg == v.end())
    // we know that all elements in v are greater than or equal to zero
```

The first loop reads data from the standard input. We have no idea how many times this loop will execute. The condition fails when `cin` reads invalid data, encounters some other input failure, or hits end-of-file. The second loop continues until we find a negative value. When the loop terminates, `beg` is either equal to `v.end()`, or it denotes an element in `v` whose value is less than zero. We can use the state of `beg` outside the `while` to determine further processing.

EXERCISES SECTION 5.4.1

Exercise 5.14: Write a program to read `strings` from standard input looking for duplicated words. The program should find places in the input where one word is followed immediately by itself. Keep track of the largest number of times a single repetition occurs and which word is repeated. Print the maximum number of duplicates, or else print a message saying that no word was repeated. For example, if the input is

```
how now now now brown cow cow
```

the output should indicate that the word `now` occurred three times.

5.4.2 Traditional `for` Statement

The syntactic form of the **`for` statement** is:

```
for (init-statement condition; expression)
    statement
```

The `for` and the part inside the parentheses is often referred to as the `for` header.

init-statement must be a declaration statement, an expression statement, or a null statement. Each of these statements ends with a semicolon, so the syntactic form can also be thought of as

```
for (initializer; condition; expression)
    statement
```

In general, *init-statement* is used to initialize or assign a starting value that is modified over the course of the loop. *condition* serves as the loop control. As long as *condition* evaluates as `true`, *statement* is executed. If the first evaluation

of *condition* yields `false`, *statement* is not executed. *expression* usually modifies the variable(s) initialized in *init-statement* and tested in *condition*. *expression* is evaluated after each iteration of the loop. As usual, *statement* can be either a single or a compound statement.

Execution Flow in a Traditional `for` Loop

Given the following `for` loop from § 3.2.3 (p. 94):

```
// process characters in s until we run out of characters or we hit a whitespace
for (decltype(s.size()) index = 0;
     index != s.size() && !isspace(s[index]); ++index)
        s[index] = toupper(s[index]); // capitalize the current character
```

the order of evaluation is as follows:

1. *init-statement* is executed once at the start of the loop. In this example, `index` is defined and initialized to zero.
2. Next, *condition* is evaluated. If `index` is not equal to `s.size()` and the character at `s[index]` is not whitespace, the `for` body is executed. Otherwise, the loop terminates. If the condition is `false` on the first iteration, then the `for` body is not executed at all.
3. If the condition is `true`, the `for` body executes. In this case, the `for` body makes the character at `s[index]` uppercase.
4. Finally, *expression* is evaluated. In this example, `index` is incremented by 1.

These four steps represent the first iteration of the `for` loop. Step 1 is executed only once on entry to the loop. Steps 2, 3, and 4 are repeated until the condition evaluates as `false`—that is, when we encounter a whitespace character in `s`, or `index` is greater than `s.size()`.

It is worth remembering that the visibility of any object defined within the `for` header is limited to the body of the `for` loop. Thus, in this example, `index` is inaccessible after the `for` completes.

Multiple Definitions in the `for` Header

As in any other declaration, *init-statement* can define several objects. However, *init-statement* may be only a single declaration statement. Therefore, all the variables must have the same base type (§ 2.3, p. 50). As one example, we might write a loop to duplicate the elements of a `vector` on the end as follows:

```
// remember the size of v and stop when we get to the original last element
for (decltype(v.size()) i = 0, sz = v.size(); i != sz; ++i)
    v.push_back(v[i]);
```

In this loop we define both the index, `i`, and the loop control, `sz`, in *init-statement*.

Omitting Parts of the `for` Header

A `for` header can omit any (or all) of *init-statement*, *condition*, or *expression*.

We can use a null statement for *init-statement* when an initialization is unnecessary. For example, we might rewrite the loop that looked for the first negative number in a `vector` so that it uses a `for`:

```
auto beg = v.begin();
for ( /* null */; beg != v.end() && *beg >= 0; ++beg)
    ; // no work to do
```

Note that the semicolon is necessary to indicate the absence of *init-statement*—more precisely, the semicolon represents a null *init-statement*. In this loop, the `for` body is also empty because all the work of the loop is done inside the `for` condition and expression. The condition decides when it's time to stop looking and the expression increments the iterator.

Omitting *condition* is equivalent to writing `true` as the condition. Because the condition always evaluates as `true`, the `for` body must contain a statement that exits the loop. Otherwise the loop will execute indefinitely:

```
for (int i = 0; /* no condition */ ; ++i) {
    // process i; code inside the loop must stop the iteration!
}
```

We can also omit *expression* from the `for` header. In such loops, either the condition or the body must do something to advance the iteration. As an example, we'll rewrite the `while` loop that read input into a `vector` of `int`s:

```
vector<int> v;
for (int i; cin >> i; /* no expression */ )
   v.push_back(i);
```

In this loop there is no need for an expression because the condition changes the value of `i`. The condition tests the input stream so that the loop ends when we've read all the input or encounter an input error.

5.4.3 Range `for` Statement

The new standard introduced a simpler `for` statement that can be used to iterate through the elements of a container or other sequence. The syntactic form of the **range `for` statement** is:

C++ 11

```
for (declaration : expression)
    statement
```

expression must represent a sequence, such as a braced initializer list (§ 3.3.1, p. 98), an array (§ 3.5, p. 113), or an object of a type such as `vector` or `string` that has `begin` and `end` members that return iterators (§ 3.4, p. 106).

declaration defines a variable. It must be possible to convert each element of the sequence to the variable's type (§ 4.11, p. 159). The easiest way to ensure that the

Exercises Section 5.4.2

Exercise 5.15: Explain each of the following loops. Correct any problems you detect.

```
(a) for (int ix = 0; ix != sz; ++ix)  { /* ... */ }
    if (ix != sz)
        // ...
(b) int ix;
    for (ix != sz; ++ix) { /* ... */ }
(c) for (int ix = 0; ix != sz; ++ix, ++ sz)  { /* ... */ }
```

Exercise 5.16: The `while` loop is particularly good at executing while some condition holds; for example, when we need to read values until end-of-file. The `for` loop is generally thought of as a step loop: An index steps through a range of values in a collection. Write an idiomatic use of each loop and then rewrite each using the other loop construct. If you could use only one loop, which would you choose? Why?

Exercise 5.17: Given two `vectors` of `ints`, write a program to determine whether one `vector` is a prefix of the other. For `vectors` of unequal length, compare the number of elements of the smaller `vector`. For example, given the `vectors` containing `0, 1, 1,` and `2` and `0, 1, 1, 2, 3, 5, 8`, respectively your program should return `true`.

types match is to use the `auto` type specifier (§ 2.5.2, p. 68). That way the compiler will deduce the type for us. If we want to write to the elements in the sequence, the loop variable must be a reference type.

On each iteration, the control variable is defined and initialized by the next value in the sequence, after which *statement* is executed. As usual, *statement* can be a single statement or a block. Execution ends once all the elements have been processed.

We have already seen several such loops, but for completeness, here is one that doubles the value of each element in a `vector`:

```
vector<int> v = {0,1,2,3,4,5,6,7,8,9};
// range variable must be a reference so we can write to the elements
for (auto &r : v)    // for each element in v
    r *= 2;          // double the value of each element in v
```

The `for` header declares the loop control variable, `r`, and associates it with `v`. We use `auto` to let the compiler infer the correct type for `r`. Because we want to change the value of the elements in `v`, we declare `r` as a reference. When we assign to `r` inside the loop, that assignment changes the element to which `r` is bound.

A range `for` is defined in terms of the equivalent traditional `for`:

```
for (auto beg = v.begin(), end = v.end(); beg != end; ++beg) {
    auto &r = *beg; // r must be a reference so we can change the element
    r *= 2;         // double the value of each element in v
}
```

Now that we know how a range `for` works, we can understand why we said in § 3.3.2 (p. 101) that we cannot use a range `for` to add elements to a `vector` (or

other container). In a range `for`, the value of `end()` is cached. If we add elements to (or remove them from) the sequence, the value of `end` might be invalidated (§ 3.4.1, p. 110). We'll have more to say about these matters in § 9.3.6 (p. 353).

5.4.4 The `do while` Statement

A **`do while` statement** is like a `while` but the condition is tested after the statement body completes. Regardless of the value of the condition, we execute the loop at least once. The syntactic form is as follows:

```
do
        statement
while (condition);
```

A `do while` ends with a semicolon after the parenthesized condition.

In a `do`, *statement* is executed before *condition* is evaluated. *condition* cannot be empty. If *condition* evaluates as `false`, then the loop terminates; otherwise, the loop is repeated. Variables used in *condition* must be defined outside the body of the `do while` statement.

We can write a program that (indefinitely) does sums using a `do while`:

```
// repeatedly ask the user for a pair of numbers to sum
string rsp;  // used in the condition; can't be defined inside the do
do {
    cout << "please enter two values: ";
    int val1 = 0, val2 = 0;
    cin  >> val1 >> val2;
    cout << "The sum of " << val1 << " and " << val2
         << " = " << val1 + val2 << "\n\n"
         << "More? Enter yes or no: ";
    cin  >> rsp;
} while (!rsp.empty() && rsp[0] != 'n');
```

The loop starts by prompting the user for two numbers. It then prints their sum and asks whether the user wishes to do another sum. The condition checks that the user gave a response. If not, or if the input starts with an `n`, the loop is exited. Otherwise the loop is repeated.

Because the condition is not evaluated until after the statement or block is executed, the `do while` loop does not allow variable definitions inside the condition:

```
do {
    // ...
    mumble(foo);
} while (int foo = get_foo()); // error: declaration in a do condition
```

If we could define variables in the condition, then any use of the variable would happen *before* the variable was defined!

EXERCISES SECTION 5.4.4

Exercise 5.18: Explain each of the following loops. Correct any problems you detect.

(a)
```
do
        int v1, v2;
        cout << "Please enter two numbers to sum:" ;
        if (cin >> v1 >> v2)
            cout << "Sum is: " << v1 + v2 << endl;
    while (cin);
```
(b)
```
do {
        // ...
    } while (int ival = get_response());
```
(c)
```
do {
        int ival = get_response();
     } while (ival);
```

Exercise 5.19: Write a program that uses a `do while` loop to repetitively request two `string`s from the user and report which `string` is less than the other.

5.5 Jump Statements

Jump statements interrupt the flow of execution. C++ offers four jumps: `break`, `continue`, and `goto`, which we cover in this chapter, and the `return` statement, which we'll describe in § 6.3 (p. 222).

5.5.1 The `break` Statement

A **`break` statement** terminates the nearest enclosing `while`, `do while`, `for`, or `switch` statement. Execution resumes at the statement immediately following the terminated statement.

A `break` can appear only within an iteration statement or `switch` statement (including inside statements or blocks nested inside such loops). A `break` affects only the nearest enclosing loop or `switch`:

```
string buf;
while (cin >> buf && !buf.empty()) {
    switch(buf[0]) {
    case '-':
        //  process up to the first blank
        for (auto it = buf.begin()+1; it != buf.end(); ++it) {
              if (*it == ' ')
                   break; //  #1, leaves the for loop
              // ...
        }
        //  break #1 transfers control here
        //  remaining '-' processing:
        break; //  #2, leaves the switch statement
```

```
        case '+':
            // ...
        } // end switch
      // end of switch: break #2 transfers control here
  } // end while
```

The `break` labeled #1 terminates the `for` loop that follows the hyphen `case` label. It does not terminate the enclosing `switch` statement and in fact does not even terminate the processing for the current case. Processing continues with the first statement following the `for`, which might be additional code to handle a hyphen or the `break` that completes that section.

The `break` labeled #2 terminates the `switch` but does not terminate the enclosing `while` loop. Processing continues after that `break` by executing the condition in the `while`.

EXERCISES SECTION 5.5.1

Exercise 5.20: Write a program to read a sequence of `string`s from the standard input until either the same word occurs twice in succession or all the words have been read. Use a `while` loop to read the text one word at a time. Use the `break` statement to terminate the loop if a word occurs twice in succession. Print the word if it occurs twice in succession, or else print a message saying that no word was repeated.

5.5.2 The `continue` Statement

A **`continue` statement** terminates the current iteration of the nearest enclosing loop and immediately begins the next iteration. A `continue` can appear only inside a `for`, `while`, or `do while` loop, including inside statements or blocks nested inside such loops. Like the `break` statement, a `continue` inside a nested loop affects only the nearest enclosing loop. Unlike a `break`, a `continue` may appear inside a `switch` only if that `switch` is embedded inside an iterative statement.

A `continue` interrupts the current iteration; execution stays inside the loop. In the case of a `while` or a `do while`, execution continues by evaluating the condition. In a traditional `for` loop, execution continues at the *expression* inside the `for` header. In a range `for`, execution continues by initializing the control variable from the next element in the sequence.

For example, the following loop reads the standard input one word at a time. Only words that begin with an underscore will be processed. For any other value, we terminate the current iteration and get the next input:

```
string buf;
while (cin >> buf && !buf.empty()) {
    if (buf[0] != '_')
        continue; // get another input
    // still here? the input starts with an underscore; process buf . . .
}
```

EXERCISES SECTION 5.5.2

Exercise 5.21: Revise the program from the exercise in § 5.5.1 (p. 191) so that it looks only for duplicated words that start with an uppercase letter.

5.5.3 The goto Statement

A **goto statement** provides an unconditional jump from the `goto` to a another statement in the same function.

Programs should not use `goto`s. `goto`s make programs hard to understand and hard to modify.

The syntactic form of a `goto` statement is

```
goto label;
```

where *label* is an identifier that identifies a statement. A **labeled statement** is any statement that is preceded by an identifier followed by a colon:

```
end: return;   // labeled statement; may be the target of a goto
```

Label identifiers are independent of names used for variables and other identifiers. Hence, a label may have the same identifier as another entity in the program without interfering with the other uses of that identifier. The `goto` and the labeled statement to which it transfers control must be in the same function.

As with a `switch` statement, a `goto` cannot transfer control from a point where an initialized variable is out of scope to a point where that variable is in scope:

```
    // ...
    goto end;
    int ix = 10; // error: goto bypasses an initialized variable definition
end:
    // error: code here could use ix but the goto bypassed its declaration
    ix = 42;
```

A jump backward over an already executed definition is okay. Jumping back to a point before a variable is defined destroys the variable and constructs it again:

```
// backward jump over an initialized variable definition is okay
  begin:
    int sz = get_size();
    if (sz <= 0) {
          goto begin;
    }
```

Here `sz` is destroyed when the `goto` executes. It is defined and initialized anew when control passes back through its definition after the jump back to `begin`.

EXERCISES SECTION 5.5.3

Exercise 5.22: The last example in this section that jumped back to `begin` could be better written using a loop. Rewrite the code to eliminate the `goto`.

5.6 `try` Blocks and Exception Handling

Exceptions are run-time anomalies—such as losing a database connection or encountering unexpected input—that exist outside the normal functioning of a program. Dealing with anomalous behavior can be one of the most difficult parts of designing any system.

Exception handling is generally used when one part of a program detects a problem that it cannot resolve and the problem is such that the detecting part of the program cannot continue. In such cases, the detecting part needs a way to signal that something happened and that it cannot continue. Moreover, the detecting part needs a way to signal the problem without knowing what part of the program will deal with the exceptional condition. Having signaled what happened, the detecting part stops processing.

A program that contains code that might raise an exception (usually) has another part to handle whatever happened. For example, if the problem is invalid input, the handling part might ask the user to provide correct input. If the database was lost, the handling part might alert an operator.

Exception handling supports this cooperation between the detecting and handling parts of a program. In C++, exception handling involves

- **`throw` expressions**, which the detecting part uses to indicate that it encountered something it can't handle. We say that a `throw` **raises** an exception.
- **`try` blocks**, which the handling part uses to deal with an exception. A `try` block starts with the keyword `try` and ends with one or more **`catch` clauses**. Exceptions thrown from code executed inside a `try` block are usually handled by one of the `catch` clauses. Because they "handle" the exception, `catch` clauses are also known as **exception handlers**.
- A set of **`exception` classes** that are used to pass information about what happened between a `throw` and an associated `catch`.

In the remainder of this section, we'll introduce these three components of exception handling. We'll also have more to say about exceptions in § 18.1 (p. 772).

5.6.1 A `throw` Expression

The detecting part of a program uses a `throw` expression to raise an exception. A `throw` consists of the keyword `throw` followed by an expression. The type of the expression determines what kind of exception is thrown. A `throw` expression is usually followed by a semicolon, making it into an expression statement.

As a simple example, recall the program in § 1.5.2 (p. 23) that added two objects of type `Sales_item`. That program checked whether the records it read referred to the same book. If not, it printed a message and exited.

```
Sales_item item1, item2;
cin >> item1 >> item2;
// first check that item1 and item2 represent the same book
if (item1.isbn() == item2.isbn()) {
    cout << item1 + item2 << endl;
    return 0;   // indicate success
} else {
    cerr << "Data must refer to same ISBN"
              << endl;
    return -1;  // indicate failure
}
```

In a more realistic program, the part that adds the objects might be separated from the part that manages the interaction with a user. In this case, we might rewrite the test to throw an exception rather than returning an error indicator:

```
// first check that the data are for the same item
if (item1.isbn() != item2.isbn())
    throw runtime_error("Data must refer to same ISBN");
// if we're still here, the ISBNs are the same
cout << item1 + item2 << endl;
```

In this code, if the ISBNs differ, we throw an expression that is an object of type `runtime_error`. Throwing an exception terminates the current function and transfers control to a handler that will know how to handle this error.

The type `runtime_error` is one of the standard library exception types and is defined in the `stdexcept` header. We'll have more to say about these types in § 5.6.3 (p. 197). We must initialize a `runtime_error` by giving it a `string` or a C-style character string (§ 3.5.4, p. 122). That string provides additional information about the problem.

5.6.2 The `try` Block

The general form of a `try` block is

```
try {
    program-statements
} catch (exception-declaration) {
    handler-statements
} catch (exception-declaration) {
    handler-statements
} // ...
```

A `try` block begins with the keyword `try` followed by a block, which, as usual, is a sequence of statements enclosed in curly braces.

Following the try block is a list of one or more catch clauses. A catch consists of three parts: the keyword catch, the declaration of a (possibly unnamed) object within parentheses (referred to as an **exception declaration**), and a block. When a catch is selected to handle an exception, the associated block is executed. Once the catch finishes, execution continues with the statement immediately following the last catch clause of the try block.

The *program-statements* inside the try constitute the normal logic of the program. Like any other blocks, they can contain any C++ statement, including declarations. As with any block, variables declared inside a try block are inaccessible outside the block—in particular, they are not accessible to the catch clauses.

Writing a Handler

In the preceding example, we used a throw to avoid adding two Sales_items that represented different books. We imagined that the part of the program that added two Sales_items was separate from the part that communicated with the user. The part that interacts with the user might contain code something like the following to handle the exception that was thrown:

```
while (cin >> item1 >> item2) {
    try {
        // execute code that will add the two Sales_items
        // if the addition fails, the code throws a runtime_error exception
    } catch (runtime_error err) {
        // remind the user that the ISBNs must match and prompt for another pair
        cout << err.what()
             << "\nTry Again?  Enter y or n" << endl;
        char c;
        cin >> c;
        if (!cin || c == 'n')
            break;       // break out of the while loop
    }
}
```

The ordinary logic of the program that manages the interaction with the user appears inside the try block. This part of the program is wrapped inside a try because it might throw an exception of type runtime_error.

This try block has a single catch clause, which handles exceptions of type runtime_error. The statements in the block following the catch are executed if code inside the try block throws a runtime_error. Our catch handles the error by printing a message and asking the user to indicate whether to continue. If the user enters 'n', then the break is executed and we exit the while. Otherwise, execution falls through to the closing brace of the while, which transfers control back to the while condition for the next iteration.

The prompt to the user prints the return from err.what(). We know that err has type runtime_error, so we can infer that what is a member function (§ 1.5.2, p. 23) of the runtime_error class. Each of the library exception classes defines a member function named what. These functions take no arguments and return a C-style character string (i.e., a const char*). The what member of

`runtime_error` returns a copy of the `string` used to initialize the particular object. If the code described in the previous section threw an exception, then this `catch` would print

```
Data must refer to same ISBN
Try Again?  Enter y or n
```

Functions Are Exited during the Search for a Handler

In complicated systems, the execution path of a program may pass through multiple `try` blocks before encountering code that throws an exception. For example, a `try` block might call a function that contains a `try`, which calls another function with its own `try`, and so on.

The search for a handler reverses the call chain. When an exception is thrown, the function that threw the exception is searched first. If no matching `catch` is found, that function terminates. The function that called the one that threw is searched next. If no handler is found, that function also exits. That function's caller is searched next, and so on back up the execution path until a `catch` of an appropriate type is found.

If no appropriate `catch` is found, execution is transferred to a library function named **`terminate`**. The behavior of that function is system dependent but is guaranteed to stop further execution of the program.

Exceptions that occur in programs that do not define any `try` blocks are handled in the same manner: After all, if there are no `try` blocks, there can be no handlers. If a program has no `try` blocks and an exception occurs, then `terminate` is called and the program is exited.

CAUTION: WRITING EXCEPTION SAFE CODE IS *Hard*

It is important to realize that exceptions interrupt the normal flow of a program. At the point where the exception occurs, some of the computations that the caller requested may have been done, while others remain undone. In general, bypassing part of the program might mean that an object is left in an invalid or incomplete state, or that a resource is not freed, and so on. Programs that properly "clean up" during exception handling are said to be *exception safe*. Writing exception safe code is surprisingly hard, and (largely) beyond the scope of this language Primer.

Some programs use exceptions simply to terminate the program when an exceptional condition occurs. Such programs generally don't worry about exception safety.

Programs that do handle exceptions and continue processing generally must be constantly aware of whether an exception might occur and what the program must do to ensure that objects are valid, that resources don't leak, and that the program is restored to an appropriate state.

We will occasionally point out particularly common techniques used to promote exception safety. However, readers whose programs require robust exception handling should be aware that the techniques we cover are insufficient by themselves to achieve exception safety.

5.6.3 Standard Exceptions

The C++ library defines several classes that it uses to report problems encountered in the functions in the standard library. These exception classes are also intended to be used in the programs we write. These classes are defined in four headers:

- The `exception` header defines the most general kind of exception class named `exception`. It communicates only that an exception occurred but provides no additional information.
- The `stdexcept` header defines several general-purpose exception classes, which are listed in Table 5.1.
- The `new` header defines the `bad_alloc` exception type, which we cover in § 12.1.2 (p. 458).
- The `type_info` header defines the `bad_cast` exception type, which we cover in § 19.2 (p. 825).

Table 5.1: Standard Exception Classes Defined in `<stdexcept>`

`exception`	The most general kind of problem.
`runtime_error`	Problem that can be detected only at run time.
`range_error`	Run-time error: result generated outside the range of values that are meaningful.
`overflow_error`	Run-time error: computation that overflowed.
`underflow_error`	Run-time error: computation that underflowed.
`logic_error`	Error in the logic of the program.
`domain_error`	Logic error: argument for which no result exists.
`invalid_argument`	Logic error: inappropriate argument.
`length_error`	Logic error: attempt to create an object larger than the maximum size for that type.
`out_of_range`	Logic error: used a value outside the valid range.

The library exception classes have only a few operations. We can create, copy, and assign objects of any of the exception types.

We can only default initialize (§ 2.2.1, p. 43) `exception`, `bad_alloc`, and `bad_cast` objects; it is not possible to provide an initializer for objects of these exception types.

The other exception types have the opposite behavior: We can initialize those objects from either a `string` or a C-style string, but we *cannot* default initialize them. When we create objects of any of these other exception types, we must supply an initializer. That initializer is used to provide additional information about the error that occurred.

The exception types define only a single operation named `what`. That function takes no arguments and returns a `const char*` that points to a C-style character string (§ 3.5.4, p. 122). The purpose of this C-style character string is to provide some sort of textual description of the exception thrown.

The contents of the C-style string that `what` returns depends on the type of the exception object. For the types that take a string initializer, the `what` function returns that string. For the other types, the value of the string that `what` returns varies by compiler.

EXERCISES SECTION 5.6.3

Exercise 5.23: Write a program that reads two integers from the standard input and prints the result of dividing the first number by the second.

Exercise 5.24: Revise your program to throw an exception if the second number is zero. Test your program with a zero input to see what happens on your system if you don't `catch` an exception.

Exercise 5.25: Revise your program from the previous exercise to use a `try` block to `catch` the exception. The `catch` clause should print a message to the user and ask them to supply a new number and repeat the code inside the `try`.

Chapter Summary

C++ provides a limited number of statements. Most of these affect the flow of control within a program:

- `while`, `for`, and `do while` statements, which provide iterative execution.
- `if` and `switch`, which provide conditional execution.
- `continue`, which stops the current iteration of a loop.
- `break`, which exits a loop or `switch` statement.
- `goto`, which transfers control to a labeled statement.
- `try` and `catch`, which define a `try` block enclosing a sequence of statements that might throw an exception. The `catch` clause(s) are intended to handle the exception(s) that the enclosed code might throw.
- `throw` expression statements, which exit a block of code, transferring control to an associated `catch` clause.
- `return`, which stops execution of a function. (We'll cover `return` statements in Chapter 6.)

In addition, there are expression statements and declaration statements. An expression statement causes the subject expression to be evaluated. Declarations and definitions of variables were described in Chapter 2.

Defined Terms

block Sequence of zero or more statements enclosed in curly braces. A block is a statement, so it can appear anywhere a statement is expected.

break statement Terminates the nearest enclosing loop or `switch` statement. Execution transfers to the first statement following the terminated loop or `switch`.

case label Constant expression (§ 2.4.4, p. 65) that follows the keyword `case` in a `switch` statement. No two `case` labels in the same `switch` statement may have the same value.

catch clause The `catch` keyword, an exception declaration in parentheses, and a block of statements. The code inside a `catch` clause does whatever is necessary to handle an exception of the type defined in its exception declaration.

compound statement Synonym for block.

continue statement Terminates the current iteration of the nearest enclosing loop. Execution transfers to the loop condition in a `while` or `do`, to the next iteration in a range `for`, or to the expression in the header of a traditional `for` loop.

dangling else Colloquial term used to refer to the problem of how to process nested `if` statements in which there are more `if`s than `else`s. In C++, an `else` is always paired with the closest preceding unmatched `if`. Note that curly braces can be used to effectively hide an inner `if` so that the programmer can control which `if` a given `else` should match.

default label `case` label that matches any otherwise unmatched value computed in the `switch` expression.

do while statement Like a `while`, except that the condition is tested at the end of the loop, not the beginning. The statement inside the `do` is executed at least once.

exception classes Set of classes defined by the standard library to be used to represent errors. Table 5.1 (p. 197) lists the general-purpose exception classes.

exception declaration The declaration in a `catch` clause. This declaration specifies the type of exceptions the `catch` can handle.

exception handler Code that deals with an exception raised in another part of the program. Synonym for `catch` clause.

exception safe Term used to describe programs that behave correctly when exceptions are thrown.

expression statement An expression followed by a semicolon. An expression statement causes the expression to be evaluated.

flow of control Execution path through a program.

for statement Iteration statement that provides iterative execution. Ordinarily used to step through a container or to repeat a calculation a given number of times.

goto statement Statement that causes an unconditional transfer of control to a specified labeled statement elsewhere in the same function. `goto`s obfuscate the flow of control within a program and should be avoided.

if else statement Conditional execution of code following the `if` or the `else`, depending on the truth value of the condition.

if statement Conditional execution based on the value of the specified condition. If the condition is `true`, then the `if` body is executed. If not, control flows to the statement following the `if`.

labeled statement Statement preceded by a label. A label is an identifier followed by a colon. Label identifiers are independent of other uses of the same identifier.

null statement An empty statement. Indicated by a single semicolon.

raise Often used as a synonym for throw. C++ programmers speak of "throwing" or "raising" an exception interchangeably.

range for statement Statement that iterates through a sequence.

switch statement A conditional statement that starts by evaluating the expression that follows the `switch` keyword. Control passes to the labeled statement with a `case` label that matches the value of the expression. If there is no matching label, execution either continues at the `default` label, if there is one, or falls out of the `switch` if there is no `default` label.

terminate Library function that is called if an exception is not caught. `terminate` aborts the program.

throw expression Expression that interrupts the current execution path. Each `throw` throws an object and transfers control to the nearest enclosing `catch` clause that can handle the type of exception that is thrown.

try block Block enclosed by the keyword `try` and one or more `catch` clauses. If the code inside a `try` block raises an exception and one of the `catch` clauses matches the type of the exception, then the exception is handled by that `catch`. Otherwise, the exception is handled by an enclosing `try` block or the program terminates.

while statement Iteration statement that executes its target statement as long as a specified condition is `true`. The statement is executed zero or more times, depending on the truth value of the condition.

CHAPTER 6

FUNCTIONS

CONTENTS

This chapter describes how to define and declare functions. We'll cover how arguments are passed to and values returned from functions. In C++, functions can be overloaded, which means that we can use the same name for several different functions. We'll cover both how to overload functions and how the compiler selects the matching version for a particular call from several overloaded functions. The chapter closes by describing pointers to functions.

A function is a block of code with a name. We execute the code by calling the function. A function may take zero or more arguments and (usually) yields a result. Functions can be overloaded, meaning that the same name may refer to several different functions.

6.1 Function Basics

A ***function*** definition typically consists of a ***return type***, a name, a list of zero or more ***parameters***, and a body. The parameters are specified in a comma-separated list enclosed in parentheses. The actions that the function performs are specified in a statement block (§ 5.1, p. 173), referred to as the ***function body***.

We execute a function through the **call operator**, which is a pair of parentheses. The call operator takes an expression that is a function or points to a function. Inside the parentheses is a comma-separated list of ***arguments***. The arguments are used to initialize the function's parameters. The type of a call expression is the return type of the function.

Writing a Function

As an example, we'll write a function to determine the factorial of a given number. The factorial of a number `n` is the product of the numbers from 1 through `n`. The factorial of 5, for example, is 120.

```
1 * 2 * 3 * 4 * 5 = 120
```

We might define this function as follows:

```
// factorial of val is val * (val - 1) * (val - 2) ... * ((val - (val - 1)) * 1)
int fact(int val)
{
    int ret = 1; // local variable to hold the result as we calculate it
    while (val > 1)
        ret *= val--;  // assign ret * val to ret and decrement val
    return ret;        // return the result
}
```

Our function is named `fact`. It takes one `int` parameter and returns an `int` value. Inside the `while` loop, we compute the factorial using the postfix decrement operator (§ 4.5, p. 147) to reduce the value of `val` by 1 on each iteration. The `return` statement ends execution of `fact` and returns the value of `ret`.

Calling a Function

To call `fact`, we must supply an `int` value. The result of the call is also an `int`:

```
int main()
{
    int j = fact(5);  // j equals 120, i.e., the result of fact(5)
    cout << "5! is " << j << endl;
    return 0;
}
```

A function call does two things: It initializes the function's parameters from the corresponding arguments, and it transfers control to that function. Execution of the *calling* function is suspended and execution of the *called* function begins.

Execution of a function begins with the (implicit) definition and initialization of its parameters. Thus, when we call `fact`, the first thing that happens is that an `int` variable named `val` is created. This variable is initialized by the argument in the call to `fact`, which in this case is `5`.

Execution of a function ends when a `return` statement is encountered. Like a function call, the `return` statement does two things: It returns the value (if any) in the `return`, and it transfers control out of the *called* function back to the *calling* function. The value returned by the function is used to initialize the result of the call expression. Execution continues with whatever remains of the expression in which the call appeared. Thus, our call to `fact` is equivalent to the following:

```
int val = 5;        // initialize val from the literal 5
int ret = 1;        // code from the body of fact
while (val > 1)
    ret *= val--;
int j = ret;        // initialize j as a copy of ret
```

Parameters and Arguments

Arguments are the initializers for a function's parameters. The first argument initializes the first parameter, the second argument initializes the second parameter, and so on. Although we know which argument initializes which parameter, we have no guarantees about the order in which arguments are evaluated (§ 4.1.3, p. 137). The compiler is free to evaluate the arguments in whatever order it prefers.

The type of each argument must match the corresponding parameter in the same way that the type of any initializer must match the type of the object it initializes. We must pass exactly the same number of arguments as the function has parameters. Because every call is guaranteed to pass as many arguments as the function has parameters, parameters are always initialized.

Because `fact` has a single parameter of type `int`, every time we call it we must supply a single argument that can be converted (§ 4.11, p. 159) to `int`:

```
fact("hello");        // error: wrong argument type
fact();               // error: too few arguments
fact(42, 10, 0);      // error: too many arguments
fact(3.14);           // ok: argument is converted to int
```

The first call fails because there is no conversion from `const char*` to `int`. The second and third calls pass the wrong number of arguments. The `fact` function must be called with one argument; it is an error to call it with any other number. The last call is legal because there is a conversion from `double` to `int`. In this call, the argument is implicitly converted to `int` (through truncation). After the conversion, this call is equivalent to

```
fact(3);
```

Function Parameter List

A function's parameter list can be empty but cannot be omitted. Typically we define a function with no parameters by writing an empty parameter list. For compatibility with C, we also can use the keyword `void` to indicate that there are no parameters:

```
void f1(){ /* ... */ }      // implicit void parameter list
void f2(void){ /* ... */ } // explicit void parameter list
```

A parameter list typically consists of a comma-separated list of parameters, each of which looks like a declaration with a single declarator. Even when the types of two parameters are the same, the type must be repeated:

```
int f3(int v1, v2) { /* ... */ }      // error
int f4(int v1, int v2) { /* ... */ } // ok
```

No two parameters can have the same name. Moreover, local variables at the outermost scope of the function may not use the same name as any parameter.

Parameter names are optional. However, there is no way to use an unnamed parameter. Therefore, parameters ordinarily have names. Occasionally a function has a parameter that is not used. Such parameters are often left unnamed, to indicate that they aren't used. Leaving a parameter unnamed doesn't change the number of arguments that a call must supply. A call must supply an argument for every parameter, even if that parameter isn't used.

Function Return Type

Most types can be used as the return type of a function. In particular, the return type can be `void`, which means that the function does not return a value. However, the return type may not be an array type (§ 3.5, p. 113) or a function type. However, a function may return a pointer to an array or a function. We'll see how to define functions that return pointers (or references) to arrays in § 6.3.3 (p. 228) and how to return pointers to functions in § 6.7 (p. 247).

6.1.1 Local Objects

In C++, names have scope (§ 2.2.4, p. 48), and objects have **lifetimes**. It is important to understand both of these concepts.

- The scope of a name is *the part of the program's text* in which that name is visible.
- The lifetime of an object is *the time during the program's execution* that the object exists.

As we've seen, the body of a function is a statement block. As usual, the block forms a new scope in which we can define variables. Parameters and variables defined inside a function body are referred to as **local variables**. They are "local" to that function and **hide** declarations of the same name made in an outer scope.

Exercises Section 6.1

Exercise 6.1: What is the difference between a parameter and an argument?

Exercise 6.2: Indicate which of the following functions are in error and why. Suggest how you might correct the problems.

```
(a) int f() {
        string s;
        // ...
        return s;
    }
(b) f2(int i) { /* ... */ }
(c) int calc(int v1, int v1) /* ... */ }
(d) double square(double x) return x * x;
```

Exercise 6.3: Write and test your own version of `fact`.

Exercise 6.4: Write a function that interacts with the user, asking for a number and generating the factorial of that number. Call this function from `main`.

Exercise 6.5: Write a function to return the absolute value of its argument.

Objects defined outside any function exist throughout the program's execution. Such objects are created when the program starts and are not destroyed until the program ends. The lifetime of a local variable depends on how it is defined.

Automatic Objects

The objects that correspond to ordinary local variables are created when the function's control path passes through the variable's definition. They are destroyed when control passes through the end of the block in which the variable is defined. Objects that exist only while a block is executing are known as **automatic objects**. After execution exits a block, the values of the automatic objects created in that block are undefined.

Parameters are automatic objects. Storage for the parameters is allocated when the function begins. Parameters are defined in the scope of the function body. Hence they are destroyed when the function terminates.

Automatic objects corresponding to the function's parameters are initialized by the arguments passed to the function. Automatic objects corresponding to local variables are initialized if their definition contains an initializer. Otherwise, they are default initialized (§ 2.2.1, p. 43), which means that uninitialized local variables of built-in type have undefined values.

Local `static` Objects

It can be useful to have a local variable whose lifetime continues across calls to the function. We obtain such objects by defining a local variable as `static`. Each **local `static` object** is initialized before the *first* time execution passes through the

object's definition. Local `static`s are not destroyed when a function ends; they are destroyed when the program terminates.

As a trivial example, here is a function that counts how many times it is called:

```
size_t count_calls()
{
    static size_t ctr = 0;   // value will persist across calls
    return ++ctr;
}
int main()
{
    for (size_t i = 0; i != 10; ++i)
        cout << count_calls() << endl;
    return 0;
}
```

This program will print the numbers from 1 through 10 inclusive.

Before control flows through the definition of `ctr` for the first time, `ctr` is created and given an initial value of `0`. Each call increments `ctr` and returns its new value. Whenever `count_calls` is executed, the variable `ctr` already exists and has whatever value was in that variable the last time the function exited. Thus, on the second invocation, the value of `ctr` is `1`, on the third it is `2`, and so on.

If a local `static` has no explicit initializer, it is value initialized (§ 3.3.1, p. 98), meaning that local `static`s of built-in type are initialized to zero.

Exercises Section 6.1.1

Exercise 6.6: Explain the differences between a parameter, a local variable, and a local `static` variable. Give an example of a function in which each might be useful.

Exercise 6.7: Write a function that returns 0 when it is first called and then generates numbers in sequence each time it is called again.

6.1.2 Function Declarations

Like any other name, the name of a function must be declared before we can use it. As with variables (§ 2.2.2, p. 45), a function may be defined only once but may be declared multiple times. With one exception that we'll cover in § 15.3 (p. 603), we can declare a function that is not defined so long as we never use that function.

A function declaration is just like a function definition except that a declaration has no function body. In a declaration, a semicolon replaces the function body.

Because a function declaration has no body, there is no need for parameter names. Hence, parameter names are often omitted in a declaration. Although parameter names are not required, they can be used to help users of the function understand what the function does:

```
// parameter names chosen to indicate that the iterators denote a range of values to print
void print(vector<int>::const_iterator beg,
           vector<int>::const_iterator end);
```

These three elements—the return type, function name, and parameter types—describe the function's interface. They specify all the information we need to call the function. Function declarations are also known as the **function prototype**.

Function Declarations Go in Header Files

Recall that variables are declared in header files (§ 2.6.3, p. 76) and defined in source files. For the same reasons, functions should be declared in header files and defined in source files.

It may be tempting—and would be legal—to put a function declaration directly in each source file that uses the function. However, doing so is tedious and error-prone. When we use header files for our function declarations, we can ensure that all the declarations for a given function agree. Moreover, if the interface to the function changes, only one declaration has to be changed.

The source file that defines a function should include the header that contains that function's declaration. That way the compiler will verify that the definition and declaration are consistent.

The header that *declares* a function should be included in the source file that *defines* that function.

EXERCISES SECTION 6.1.2

Exercise 6.8: Write a header file named `Chapter6.h` that contains declarations for the functions you wrote for the exercises in § 6.1 (p. 205).

6.1.3 Separate Compilation

As our programs get more complicated, we'll want to store the various parts of the program in separate files. For example, we might store the functions we wrote for the exercises in § 6.1 (p. 205) in one file and store code that uses these functions in other source files. To allow programs to be written in logical parts, C++ supports what is commonly known as ***separate compilation***. Separate compilation lets us split our programs into several files, each of which can be compiled independently.

Compiling and Linking Multiple Source Files

As an example, assume that the definition of our `fact` function is in a file named `fact.cc` and its declaration is in a header file named `Chapter6.h`. Our `fact.cc` file, like any file that uses these functions, will include the `Chapter6.h` header. We'll store a `main` function that calls `fact` in a second file named `factMain.cc`.

To produce an ***executable file***, we must tell the compiler where to find all of the code we use. We might compile these files as follows:

```
$ CC factMain.cc fact.cc   # generates factMain.exe or a.out
$ CC factMain.cc fact.cc -o main # generates main or main.exe
```

Here `CC` is the name of our compiler, `$` is our system prompt, and `#` begins a command-line comment. We can now run the executable file, which will run our `main` function.

If we have changed only one of our source files, we'd like to recompile only the file that actually changed. Most compilers provide a way to separately compile each file. This process usually yields a file with the `.obj` (Windows) or `.o` (UNIX) file extension, indicating that the file contains ***object code***.

The compiler lets us ***link*** object files together to form an executable. On the system we use, we would separately compile our program as follows:

```
$ CC -c factMain.cc     # generates factMain.o
$ CC -c fact.cc         # generates fact.o
$ CC factMain.o fact.o  # generates factMain.exe or a.out
$ CC factMain.o fact.o -o main # generates main or main.exe
```

You'll need to check with your compiler's user's guide to understand how to compile and execute programs made up of multiple source files.

Exercises Section 6.1.3

Exercise 6.9: Write your own versions of the `fact.cc` and `factMain.cc` files. These files should include your `Chapter6.h` from the exercises in the previous section. Use these files to understand how your compiler supports separate compilation.

6.2 Argument Passing

As we've seen, each time we call a function, its parameters are created and initialized by the arguments passed in the call.

Parameter initialization works the same way as variable initialization.

As with any other variable, the type of a parameter determines the interaction between the parameter and its argument. If the parameter is a reference (§ 2.3.1, p. 50), then the parameter is bound to its argument. Otherwise, the argument's value is copied.

When a parameter is a reference, we say that its corresponding argument is **"passed by reference"** or that the function is **"called by reference."** As with any other reference, a reference parameter is an alias for the object to which it is bound; that is, the parameter is an alias for its corresponding argument.

When the argument value is copied, the parameter and argument are independent objects. We say such arguments are **"passed by value"** or alternatively that the function is **"called by value."**

6.2.1 Passing Arguments by Value

When we initialize a nonreference type variable, the value of the initializer is copied. Changes made to the variable have no effect on the initializer:

```
int n = 0;              // ordinary variable of type int
int i = n;              // i is a copy of the value in n
i = 42;                 // value in i is changed; n is unchanged
```

Passing an argument by value works exactly the same way; nothing the function does to the parameter can affect the argument. For example, inside `fact` (§ 6.1, p. 202) the parameter `val` is decremented:

```
ret *= val--;  // decrements the value of val
```

Although `fact` changes the value of `val`, that change has no effect on the argument passed to `fact`. Calling `fact(i)` does not change the value of `i`.

Pointer Parameters

Pointers (§ 2.3.2, p. 52) behave like any other nonreference type. When we copy a pointer, the value of the pointer is copied. After the copy, the two pointers are distinct. However, a pointer also gives us indirect access to the object to which that pointer points. We can change the value of that object by assigning through the pointer (§ 2.3.2, p. 55):

```
int n = 0, i = 42;
int *p = &n, *q = &i; // p points to n; q points to i
*p = 42;              // value in n is changed; p is unchanged
p = q;                // p now points to i; values in i and n are unchanged
```

The same behavior applies to pointer parameters:

```
// function that takes a pointer and sets the pointed-to value to zero
void reset(int *ip)
{
    *ip = 0;  // changes the value of the object to which ip points
    ip = 0;   // changes only the local copy of ip; the argument is unchanged
}
```

After a call to `reset`, the object to which the argument points will be `0`, but the pointer argument itself is unchanged:

```
int i = 42;
reset(&i);                      // changes i but not the address of i
cout << "i = "  << i << endl;   // prints i = 0
```

Programmers accustomed to programming in C often use pointer parameters to access objects outside a function. In C++, programmers generally use reference parameters instead.

EXERCISES SECTION 6.2.1

Exercise 6.10: Using pointers, write a function to swap the values of two `int`s. Test the function by calling it and printing the swapped values.

6.2.2 Passing Arguments by Reference

Recall that operations on a reference are actually operations on the object to which the reference refers (§ 2.3.1, p. 50):

```
int n = 0, i = 42;
int &r = n;              //  r is bound to n (i.e., r is another name for n)
r = 42;                  //  n is now 42
r = i;                   //  n now has the same value as i
i = r;                   //  i has the same value as n
```

Reference parameters exploit this behavior. They are often used to allow a function to change the value of one or more of its arguments.

As one example, we can rewrite our `reset` program from the previous section to take a reference instead of a pointer:

```
//  function that takes a reference to an int and sets the given object to zero
void reset(int &i)   //  i is just another name for the object passed to reset
{
    i = 0;   //  changes the value of the object to which i refers
}
```

As with any other reference, a reference parameter is bound directly to the object from which it is initialized. When we call this version of `reset`, `i` will be bound to whatever `int` object we pass. As with any reference, changes made to `i` are made to the object to which `i` refers. In this case, that object is the argument to `reset`.

When we call this version of `reset`, we pass an object directly; there is no need to pass its address:

```
int j = 42;
reset(j);  //  j is passed by reference; the value in j is changed
cout << "j = " << j  << endl;  //  prints j = 0
```

In this call, the parameter `i` is just another name for `j`. Any use of `i` inside `reset` is a use of `j`.

Using References to Avoid Copies

It can be inefficient to copy objects of large class types or large containers. Moreover, some class types (including the IO types) cannot be copied. Functions must use reference parameters to operate on objects of a type that cannot be copied.

As an example, we'll write a function to compare the length of two `string`s. Because `string`s can be long, we'd like to avoid copying them, so we'll make our parameters references. Because comparing two `string`s does not involve changing the `string`s, we'll make the parameters references to `const` (§ 2.4.1, p. 61):

```
// compare the length of two strings
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
```

As we'll see in § 6.2.3 (p. 213), functions should use references to `const` for reference parameters they do not need to change.

Reference parameters that are not changed inside a function should be references to `const`.

Using Reference Parameters to Return Additional Information

A function can return only a single value. However, sometimes a function has more than one value to return. Reference parameters let us effectively return multiple results. As an example, we'll define a function named `find_char` that will return the position of the first occurrence of a given character in a `string`. We'd also like the function to return a count of how many times that character occurs.

How can we define a function that returns a position and an occurrence count? We could define a new type that contains the position and the count. An easier solution is to pass an additional reference argument to hold the occurrence count:

```
// returns the index of the first occurrence of c in s
// the reference parameter occurs counts how often c occurs
string::size_type find_char(const string &s, char c,
                            string::size_type &occurs)
{
    auto ret = s.size();    // position of the first occurrence, if any
    occurs = 0;             // set the occurrence count parameter
    for (decltype(ret) i = 0; i != s.size(); ++i) {
        if (s[i] == c) {
            if (ret == s.size())
                ret = i;    // remember the first occurrence of c
            ++occurs;       // increment the occurrence count
        }
    }
    return ret;             // count is returned implicitly in occurs
}
```

When we call `find_char`, we have to pass three arguments: a `string` in which to look, the character to look for, and a `size_type` (§ 3.2.2, p. 88) object to hold the occurrence count. Assuming `s` is a `string`, and `ctr` is a `size_type` object, we can call `find_char` as follows:

```
auto index = find_char(s, 'o', ctr);
```

After the call, the value of `ctr` will be the number of times `o` occurs, and `index` will refer to the first occurrence if there is one. Otherwise, `index` will be equal to `s.size()` and `ctr` will be zero.

EXERCISES SECTION 6.2.2

Exercise 6.11: Write and test your own version of `reset` that takes a reference.

Exercise 6.12: Rewrite the program from exercise 6.10 in § 6.2.1 (p. 210) to use references instead of pointers to swap the value of two `int`s. Which version do you think would be easier to use and why?

Exercise 6.13: Assuming `T` is the name of a type, explain the difference between a function declared as `void f(T)` and `void f(T&)`.

Exercise 6.14: Give an example of when a parameter should be a reference type. Give an example of when a parameter should not be a reference.

Exercise 6.15: Explain the rationale for the type of each of `find_char`'s parameters In particular, why is `s` a reference to `const` but `occurs` is a plain reference? Why are these parameters references, but the `char` parameter `c` is not? What would happen if we made `s` a plain reference? What if we made `occurs` a reference to `const`?

6.2.3 `const` Parameters and Arguments

When we use parameters that are `const`, it is important to remember the discussion of top-level `const` from § 2.4.3 (p. 63). As we saw in that section, a top-level `const` is one that applies to the object itself:

```
const int ci = 42;      // we cannot change ci; const is top-level
int i = ci;             // ok: when we copy ci, its top-level const is ignored
int * const p = &i;     // const is top-level; we can't assign to p
*p = 0;                 // ok: changes through p are allowed; i is now 0
```

Just as in any other initialization, when we copy an argument to initialize a parameter, top-level `const`s are ignored. As a result, top-level `const` on parameters are ignored. We can pass either a `const` or a nonconst object to a parameter that has a top-level `const`:

```
void fcn(const int i) { /* fcn can read but not write to i */ }
```

We can call `fcn` passing it either a `const int` or a plain `int`. The fact that top-level `const`s are ignored on a parameter has one possibly surprising implication:

```
void fcn(const int i) { /* fcn can read but not write to i */ }
void fcn(int i) { /* ... */ }  // error: redefines fcn(int)
```

In C++, we can define several different functions that have the same name. However, we can do so only if their parameter lists are sufficiently different. Because top-level consts are ignored, we can pass exactly the same types to either version of fcn. The second version of fcn is an error. Despite appearances, its parameter list doesn't differ from the list in the first version of fcn.

Pointer or Reference Parameters and const

Because parameters are initialized in the same way that variables are initialized, it can be helpful to remember the general initialization rules. We can initialize an object with a low-level const from a nonconst object but not vice versa, and a plain reference must be initialized from an object of the same type.

```
int i = 42;
const int *cp = &i; // ok: but cp can't change i (§ 2.4.2 (p. 62))
const int &r = i;   // ok: but r can't change i (§ 2.4.1 (p. 61))
const int &r2 = 42; // ok: (§ 2.4.1 (p. 61))
int *p = cp;   // error: types of p and cp don't match (§ 2.4.2 (p. 62))
int &r3 = r;   // error: types of r3 and r don't match (§ 2.4.1 (p. 61))
int &r4 = 42; // error: can't initialize a plain reference from a literal (§ 2.3.1 (p. 50))
```

Exactly the same initialization rules apply to parameter passing:

```
int i = 0;
const int ci = i;
string::size_type ctr = 0;
reset(&i);    // calls the version of reset that has an int* parameter
reset(&ci);   // error: can't initialize an int* from a pointer to a const int object
reset(i);     // calls the version of reset that has an int& parameter
reset(ci);    // error: can't bind a plain reference to the const object ci
reset(42);    // error: can't bind a plain reference to a literal
reset(ctr);   // error: types don't match; ctr has an unsigned type
// ok: find_char's first parameter is a reference to const
find_char("Hello World!", 'o', ctr);
```

We can call the reference version of reset (§ 6.2.2, p. 210) only on int objects. We cannot pass a literal, an expression that evaluates to an int, an object that requires conversion, or a const int object. Similarly, we may pass only an int* to the pointer version of reset (§ 6.2.1, p. 209). On the other hand, we can pass a string literal as the first argument to find_char (§ 6.2.2, p. 211). That function's reference parameter is a reference to const, and we can initialize references to const from literals.

Use Reference to const When Possible

It is a somewhat common mistake to define parameters that a function does not change as (plain) references. Doing so gives the function's caller the misleading impression that the function might change its argument's value. Moreover, using a

reference instead of a reference to `const` unduly limits the type of arguments that can be used with the function. As we've just seen, we cannot pass a `const` object, or a literal, or an object that requires conversion to a plain reference parameter.

The effect of this mistake can be surprisingly pervasive. As an example, consider our `find_char` function from § 6.2.2 (p. 211). That function (correctly) made its `string` parameter a reference to `const`. Had we defined that parameter as a plain `string&`:

```
// bad design: the first parameter should be a const string&
string::size_type find_char(string &s, char c,
                            string::size_type &occurs);
```

we could call `find_char` only on a `string` object. A call such as

```
find_char("Hello World", 'o', ctr);
```

would fail at compile time.

More subtly, we could not use this version of `find_char` from other functions that (correctly) define their parameters as references to `const`. For example, we might want to use `find_char` inside a function that determines whether a `string` represents a sentence:

```
bool is_sentence(const string &s)
{
    // if there's a single period at the end of s, then s is a sentence
    string::size_type ctr = 0;
    return find_char(s, '.', ctr) == s.size() - 1 && ctr == 1;
}
```

If `find_char` took a plain `string&`, then this call to `find_char` would be a compile-time error. The problem is that `s` is a reference to a `const string`, but `find_char` was (incorrectly) defined to take a plain reference.

It might be tempting to try to fix this problem by changing the type of the parameter in `is_sentence`. But that fix only propagates the error—callers of `is_sentence` could pass only nonconst `string`s.

The right way to fix this problem is to fix the parameter in `find_char`. If it's not possible to change `find_char`, then define a local `string` copy of `s` inside `is_sentence` and pass that `string` to `find_char`.

6.2.4 Array Parameters

Arrays have two special properties that affect how we define and use functions that operate on arrays: We cannot copy an array (§ 3.5.1, p. 114), and when we use an array it is (usually) converted to a pointer (§ 3.5.3, p. 117). Because we cannot copy an array, we cannot pass an array by value. Because arrays are converted to pointers, when we pass an array to a function, we are actually passing a pointer to the array's first element.

Even though we cannot pass an array by value, we can write a parameter that looks like an array:

EXERCISES SECTION 6.2.3

Exercise 6.16: The following function, although legal, is less useful than it might be. Identify and correct the limitation on this function:

```
bool is_empty(string& s) { return s.empty(); }
```

Exercise 6.17: Write a function to determine whether a `string` contains any capital letters. Write a function to change a `string` to all lowercase. Do the parameters you used in these functions have the same type? If so, why? If not, why not?

Exercise 6.18: Write declarations for each of the following functions. When you write these declarations, use the name of the function to indicate what the function does.

(a) A function named `compare` that returns a `bool` and has two parameters that are references to a class named `matrix`.

(b) A function named `change_val` that returns a `vector<int>` iterator and takes two parameters: One is an `int` and the other is an iterator for a `vector<int>`.

Exercise 6.19: Given the following declarations, determine which calls are legal and which are illegal. For those that are illegal, explain why.

```
double calc(double);
int count(const string &, char);
int sum(vector<int>::iterator, vector<int>::iterator, int);
vector<int> vec(10);
(a) calc(23.4, 55.1); (b) count("abcda", 'a');
(c) calc(66);         (d) sum(vec.begin(), vec.end(), 3.8);
```

Exercise 6.20: When should reference parameters be references to `const`? What happens if we make a parameter a plain reference when it could be a reference to `const`?

```
// despite appearances, these three declarations of print are equivalent
// each function has a single parameter of type const int*
void print(const int*);
void print(const int[]);    // shows the intent that the function takes an array
void print(const int[10]); // dimension for documentation purposes (at best)
```

Regardless of appearances, these declarations are equivalent: Each declares a function with a single parameter of type `const int*`. When the compiler checks a call to `print`, it checks only that the argument has type `const int*`:

```
int i = 0, j[2] = {0, 1};
print(&i); // ok: &i is int*
print(j);  // ok: j is converted to an int* that points to j[0]
```

If we pass an array to `print`, that argument is automatically converted to a pointer to the first element in the array; the size of the array is irrelevant.

As with any code that uses arrays, functions that take array parameters must ensure that all uses of the array stay within the array bounds.

Because arrays are passed as pointers, functions ordinarily don't know the size of the array they are given. They must rely on additional information provided by the caller. There are three common techniques used to manage pointer parameters.

Using a Marker to Specify the Extent of an Array

The first approach to managing array arguments requires the array itself to contain an end marker. C-style character strings (§ 3.5.4, p. 122) are an example of this approach. C-style strings are stored in character arrays in which the last character of the string is followed by a null character. Functions that deal with C-style strings stop processing the array when they see a null character:

```
void print(const char *cp)
{
    if (cp)           //  if cp is not a null pointer
        while (*cp)   //  so long as the character it points to is not a null character
            cout << *cp++;   //  print the character and advance the pointer
}
```

This convention works well for data where there is an obvious end-marker value (like the null character) that does not appear in ordinary data. It works less well with data, such as `int`s, where every value in the range is a legitimate value.

Using the Standard Library Conventions

A second technique used to manage array arguments is to pass pointers to the first and one past the last element in the array. This approach is inspired by techniques used in the standard library. We'll learn more about this style of programming in Part II. Using this approach, we'll print the elements in an array as follows:

```
void print(const int *beg, const int *end)
{
    //  print every element starting at beg up to but not including end
    while (beg != end)
        cout << *beg++ << endl; //  print the current element
                                //  and advance the pointer
}
```

The `while` uses the dereference and postfix increment operators (§ 4.5, p. 148) to print the current element and advance `beg` one element at a time through the array. The loop stops when `beg` is equal to `end`.

To call this function, we pass two pointers—one to the first element we want to print and one just past the last element:

```
int j[2] = {0, 1};
//  j is converted to a pointer to the first element in j
//  the second argument is a pointer to one past the end of j
print(begin(j), end(j));   //  begin and end functions, see § 3.5.3 (p. 118)
```

This function is safe, as long as the caller correctly calculates the pointers. Here we let the library `begin` and `end` functions (§ 3.5.3, p. 118) provide those pointers.

Explicitly Passing a Size Parameter

A third approach for array arguments, which is common in C programs and older C++ programs, is to define a second parameter that indicates the size of the array. Using this approach, we'll rewrite `print` as follows:

```
// const int ia[] is equivalent to const int* ia
// size is passed explicitly and used to control access to elements of ia
void print(const int ia[], size_t size)
{
    for (size_t i = 0; i != size; ++i) {
        cout << ia[i] << endl;
    }
}
```

This version uses the `size` parameter to determine how many elements there are to print. When we call `print`, we must pass this additional parameter:

```
int j[] = { 0, 1 };  // int array of size 2
print(j, end(j) - begin(j));
```

The function executes safely as long as the size passed is no greater than the actual size of the array.

Array Parameters and `const`

Note that all three versions of our `print` function defined their array parameters as pointers to `const`. The discussion in § 6.2.3 (p. 213) applies equally to pointers as to references. When a function does not need write access to the array elements, the array parameter should be a pointer to `const` (§ 2.4.2, p. 62). A parameter should be a plain pointer to a nonconst type only if the function needs to change element values.

Array Reference Parameters

Just as we can define a variable that is a reference to an array (§ 3.5.1, p. 114), we can define a parameter that is a reference to an array. As usual, the reference parameter is bound to the corresponding argument, which in this case is an array:

```
// ok: parameter is a reference to an array; the dimension is part of the type
void print(int (&arr)[10])
{
    for (auto elem : arr)
        cout << elem << endl;
}
```

The parentheses around `&arr` are necessary (§ 3.5.1, p. 114):

```
f(int &arr[10])   // error: declares arr as an array of references
f(int (&arr)[10]) // ok: arr is a reference to an array of ten ints
```

Because the size of an array is part of its type, it is safe to rely on the dimension in the body of the function. However, the fact that the size is part of the type limits the usefulness of this version of `print`. We may call this function only for an array of exactly ten `int`s:

```
int i = 0, j[2] = {0, 1};
int k[10] = {0,1,2,3,4,5,6,7,8,9};
print(&i);    // error: argument is not an array of ten int
print(j);     // error: argument is not an array of ten int
print(k);     // ok: argument is an array of ten int
```

We'll see in § 16.1.1 (p. 654) how we might write this function in a way that would allow us to pass a reference parameter to an array of any size.

Passing a Multidimensional Array

Recall that there are no multidimensional arrays in C++ (§ 3.6, p. 125). Instead, what appears to be a multidimensional array is an array of arrays.

As with any array, a multidimensional array is passed as a pointer to its first element (§ 3.6, p. 128). Because we are dealing with an array of arrays, that element is an array, so the pointer is a pointer to an array. The size of the second (and any subsequent) dimension is part of the element type and must be specified:

```
// matrix points to the first element in an array whose elements are arrays of ten int
void print(int (*matrix)[10], int rowSize) { /* ... */ }
```

declares `matrix` as a pointer to an array of ten `int`s.

Again, the parentheses around `*matrix` are necessary:

```
int *matrix[10];    // array of ten pointers
int (*matrix)[10]; // pointer to an array of ten int
```

We can also define our function using array syntax. As usual, the compiler ignores the first dimension, so it is best not to include it:

```
// equivalent definition
void print(int matrix[][10], int rowSize) { /* ... */ }
```

declares `matrix` to be what looks like a two-dimensional array. In fact, the parameter is a pointer to an array of ten `int`s.

6.2.5 `main`: Handling Command-Line Options

It turns out that `main` is a good example of how C++ programs pass arrays to functions. Up to now, we have defined `main` with an empty parameter list:

```
int main() { ... }
```

However, we sometimes need to pass arguments to `main`. The most common use of arguments to `main` is to let the user specify a set of options to guide the operation of the program. For example, assuming our `main` program is in an executable file named `prog`, we might pass options to the program as follows:

Exercises Section 6.2.4

Exercise 6.21: Write a function that takes an `int` and a pointer to an `int` and returns the larger of the `int` value or the value to which the pointer points. What type should you use for the pointer?

Exercise 6.22: Write a function to swap two `int` pointers.

Exercise 6.23: Write your own versions of each of the `print` functions presented in this section. Call each of these functions to print `i` and `j` defined as follows:

```
int i = 0, j[2] = {0, 1};
```

Exercise 6.24: Explain the behavior of the following function. If there are problems in the code, explain what they are and how you might fix them.

```
void print(const int ia[10])
{
    for (size_t i = 0; i != 10; ++i)
        cout << ia[i] << endl;
}
```

```
prog -d -o ofile data0
```

Such command-line options are passed to `main` in two (optional) parameters:

```
int main(int argc, char *argv[]) { ... }
```

The second parameter, `argv`, is an array of pointers to C-style character strings. The first parameter, `argc`, passes the number of strings in that array. Because the second parameter is an array, we might alternatively define `main` as

```
int main(int argc, char **argv) { ... }
```

indicating that `argv` points to a `char*`.

When arguments are passed to `main`, the first element in `argv` points either to the name of the program or to the empty string. Subsequent elements pass the arguments provided on the command line. The element just past the last pointer is guaranteed to be 0.

Given the previous command line, `argc` would be 5, and `argv` would hold the following C-style character strings:

```
argv[0] = "prog";   // or argv[0] might point to an empty string
argv[1] = "-d";
argv[2] = "-o";
argv[3] = "ofile";
argv[4] = "data0";
argv[5] = 0;
```

When you use the arguments in `argv`, remember that the optional arguments begin in `argv[1]`; `argv[0]` contains the program's name, not user input.

Exercises Section 6.2.5

Exercise 6.25: Write a `main` function that takes two arguments. Concatenate the supplied arguments and print the resulting `string`.

Exercise 6.26: Write a program that accepts the options presented in this section. Print the values of the arguments passed to `main`.

6.2.6 Functions with Varying Parameters

Sometimes we do not know in advance how many arguments we need to pass to a function. For example, we might want to write a routine to print error messages generated from our program. We'd like to use a single function to print these error messages in order to handle them in a uniform way. However, different calls to our error-printing function might pass different arguments, corresponding to different kinds of error messages.

The new standard provides two primary ways to write a function that takes a varying number of arguments: If all the arguments have the same type, we can pass a library type named `initializer_list`. If the argument types vary, we can write a special kind of function, known as a variadic template, which we'll cover in § 16.4 (p. 699).

C++ also has a special parameter type, ellipsis, that can be used to pass a varying number of arguments. We'll look briefly at ellipsis parameters in this section. However, it is worth noting that this facility ordinarily should be used only in programs that need to interface to C functions.

`initializer_list` Parameters

C++ 11

We can write a function that takes an unknown number of arguments of a single type by using an **`initializer_list`** parameter. An `initializer_list` is a library type that represents an array (§ 3.5, p. 113) of values of the specified type. This type is defined in the `initializer_list` header. The operations that `initializer_list` provides are listed in Table 6.1.

Table 6.1: Operations on `initializer_lists`

`initializer_list<T> lst;`	Default initialization; an empty list of elements of type `T`.
`initializer_list<T> lst{a,b,c...};`	`lst` has as many elements as there are initializers; elements are copies of the corresponding initializers. Elements in the list are `const`.
`lst2(lst)` `lst2 = lst`	Copying or assigning an `initializer_list` does not copy the elements in the list. After the copy, the original and the copy share the elements.
`lst.size()`	Number of elements in the list.
`lst.begin()` `lst.end()`	Returns a pointer to the first and one past the last element in `lst`.

Like a vector, initializer_list is a template type (§ 3.3, p. 96). When we define an initializer_list, we must specify the type of the elements that the list will contain:

```
initializer_list<string> ls; // initializer_list of strings
initializer_list<int> li;    // initializer_list of ints
```

Unlike vector, the elements in an initializer_list are always const values; there is no way to change the value of an element in an initializer_list.

We can write our function to produce error messages from a varying number of arguments as follows:

```
void error_msg(initializer_list<string> il)
{
    for (auto beg = il.begin(); beg != il.end(); ++beg)
        cout << *beg << " " ;
    cout << endl;
}
```

The begin and end operations on initializer_list objects are analogous to the corresponding vector members (§ 3.4.1, p. 106). The begin() member gives us a pointer to the first element in the list, and end() is an off-the-end pointer one past the last element. Our function initializes beg to denote the first element and iterates through each element in the initializer_list. In the body of the loop we dereference beg in order to access the current element and print its value.

When we pass a sequence of values to an initializer_list parameter, we must enclose the sequence in curly braces:

```
// expected, actual are strings
if (expected != actual)
    error_msg({"functionX", expected, actual});
else
    error_msg({"functionX", "okay"});
```

Here we're calling the same function, error_msg, passing three values in the first call and two values in the second.

A function with an initializer_list parameter can have other parameters as well. For example, our debugging system might have a class, named ErrCode, that represents various kinds of errors. We can revise our program to take an ErrCode in addition to an initializer_list as follows:

```
void error_msg(ErrCode e, initializer_list<string> il)
{
    cout << e.msg() << ": ";
    for (const auto &elem : il)
        cout << elem << " " ;
    cout << endl;
}
```

Because initializer_list has begin and end members, we can use a range for (§ 5.4.3, p. 187) to process the elements. This program, like our previous version, iterates an element at a time through the braced list of values passed to the il parameter.

To call this version, we need to revise our calls to pass an `ErrCode` argument:

```
if (expected != actual)
    error_msg(ErrCode(42), {"functionX", expected, actual});
else
    error_msg(ErrCode(0), {"functionX", "okay"});
```

Ellipsis Parameters

Ellipsis parameters are in C++ to allow programs to interface to C code that uses a C library facility named `varargs`. Generally an ellipsis parameter should not be used for other purposes. Your C compiler documentation will describe how to use `varargs`.

Ellipsis parameters should be used only for types that are common to both C and C++. In particular, objects of most class types are not copied properly when passed to an ellipsis parameter.

An ellipsis parameter may appear only as the last element in a parameter list and may take either of two forms:

```
void foo(parm_list, ...);
void foo(...);
```

The first form specifies the type(s) for some of `foo`'s parameters. Arguments that correspond to the specified parameters are type checked as usual. No type checking is done for the arguments that correspond to the ellipsis parameter. In this first form, the comma following the parameter declarations is optional.

EXERCISES SECTION 6.2.6

Exercise 6.27: Write a function that takes an `initializer_list<int>` and produces the sum of the elements in the list.

Exercise 6.28: In the second version of `error_msg` that has an `ErrCode` parameter, what is the type of `elem` in the `for` loop?

Exercise 6.29: When you use an `initializer_list` in a range `for` would you ever use a reference as the loop control variable? If so, why? If not, why not?

6.3 Return Types and the `return` Statement

A `return` statement terminates the function that is currently executing and returns control to the point from which the function was called. There are two forms of `return` statements:

```
return;
return expression;
```

6.3.1 Functions with No Return Value

A `return` with no value may be used only in a function that has a return type of `void`. Functions that return `void` are not required to contain a `return`. In a `void` function, an implicit `return` takes place after the function's last statement.

Typically, `void` functions use a `return` to exit the function at an intermediate point. This use of `return` is analogous to the use of a `break` statement (§ 5.5.1, p. 190) to exit a loop. For example, we can write a `swap` function that does no work if the values are identical:

```
void swap(int &v1, int &v2)
{
    // if the values are already the same, no need to swap, just return
    if (v1 == v2)
        return;
    // if we're here, there's work to do
    int tmp = v2;
    v2 = v1;
    v1 = tmp;
    // no explicit return necessary
}
```

This function first checks if the values are equal and, if so, exits the function. If the values are unequal, the function swaps them. An implicit return occurs after the last assignment statement.

A function with a `void` return type may use the second form of the `return` statement only to return the result of calling another function that returns `void`. Returning any other expression from a `void` function is a compile-time error.

6.3.2 Functions That Return a Value

The second form of the `return` statement provides the function's result. Every return in a function with a return type other than `void` must return a value. The value returned must have the same type as the function return type, or it must have a type that can be implicitly converted (§ 4.11, p. 159) to that type.

Although C++ cannot guarantee the correctness of a result, it can guarantee that every `return` includes a result of the appropriate type. Although it cannot do so in all cases, the compiler attempts to ensure that functions that return a value are exited only through a valid `return` statement. For example:

```
// incorrect return values, this code will not compile
bool str_subrange(const string &str1, const string &str2)
{
    // same sizes: return normal equality test
    if (str1.size() == str2.size())
        return str1 == str2;    // ok: == returns bool
    // find the size of the smaller string; conditional operator, see § 4.7 (p. 151)
    auto size = (str1.size() < str2.size())
                ? str1.size() : str2.size();
```

```
    // look at each element up to the size of the smaller string
    for (decltype(size) i = 0; i != size; ++i) {
        if (str1[i] != str2[i])
            return; // error #1: no return value; compiler should detect this error
    }
    // error #2: control might flow off the end of the function without a return
    // the compiler might not detect this error
}
```

The `return` from within the `for` loop is an error because it fails to return a value. The compiler should detect this error.

The second error occurs because the function fails to provide a `return` after the loop. If we call this function with one `string` that is a subset of the other, execution would fall out of the `for`. There should be a return to handle this case. The compiler may or may not detect this error. If it does not detect the error, what happens at run time is undefined.

Failing to provide a `return` after a loop that contains a `return` is an error. However, many compilers will not detect such errors.

How Values Are Returned

Values are returned in exactly the same way as variables and parameters are initialized: The return value is used to initialize a temporary at the call site, and that temporary is the result of the function call.

It is important to keep in mind the initialization rules in functions that return local variables. As an example, we might write a function that, given a counter, a word, and an ending, gives us back the plural version of the word if the counter is greater than 1:

```
// return the plural version of word if ctr is greater than 1
string make_plural(size_t ctr, const string &word,
                                const string &ending)
{
    return (ctr > 1) ? word + ending : word;
}
```

The return type of this function is `string`, which means the return value is copied to the call site. This function returns a copy of `word`, or it returns an unnamed temporary `string` that results from adding `word` and `ending`.

As with any other reference, when a function returns a reference, that reference is just another name for the object to which it refers. As an example, consider a function that returns a reference to the shorter of its two `string` parameters:

```
// return a reference to the shorter of two strings
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() <= s2.size() ? s1 : s2;
}
```

The parameters and return type are references to `const string`. The `strings` are not copied when the function is called or when the result is returned.

Never Return a Reference or Pointer to a Local Object

When a function completes, its storage is freed (§ 6.1.1, p. 204). After a function terminates, references to local objects refer to memory that is no longer valid:

```
// disaster: this function returns a reference to a local object
const string &manip()
{
    string ret;
    // transform ret in some way
    if (!ret.empty())
        return ret;      // WRONG: returning a reference to a local object!
    else
        return "Empty"; // WRONG: "Empty" is a local temporary string
}
```

Both of these `return` statements return an undefined value—what happens if we try to use the value returned from `manip` is undefined. In the first `return`, it should be obvious that the function returns a reference to a local object. In the second case, the string literal is converted to a local temporary `string` object. That object, like the `string` named `ret`, is local to `manip`. The storage in which the temporary resides is freed when the function ends. Both `returns` refer to memory that is no longer available.

One good way to ensure that the return is safe is to ask: To what *preexisting* object is the reference referring?

For the same reasons that it is wrong to return a reference to a local object, it is also wrong to return a pointer to a local object. Once the function completes, the local objects are freed. The pointer would point to a nonexistent object.

Functions That Return Class Types and the Call Operator

Like any operator the call operator has associativity and precedence (§ 4.1.2, p. 136). The call operator has the same precedence as the dot and arrow operators (§ 4.6, p. 150). Like those operators, the call operator is left associative. As a result, if a function returns a pointer, reference or object of class type, we can use the result of a call to call a member of the resulting object.

For example, we can determine the size of the shorter `string` as follows:

```
// call the size member of the string returned by shorterString
auto sz = shorterString(s1, s2).size();
```

Because these operators are left associative, the result of `shorterString` is the left-hand operand of the dot operator. That operator fetches the `size` member of that `string`. That member is the left-hand operand of the second call operator.

Reference Returns Are Lvalues

Whether a function call is an lvalue (§ 4.1.1, p. 135) depends on the return type of the function. Calls to functions that return references are lvalues; other return types yield rvalues. A call to a function that returns a reference can be used in the same ways as any other lvalue. In particular, we can assign to the result of a function that returns a reference to nonconst:

```
char &get_val(string &str, string::size_type ix)
{
    return str[ix]; // get_val assumes the given index is valid
}
int main()
{
    string s("a value");
    cout << s << endl;    // prints a value
    get_val(s, 0) = 'A'; // changes s[0] to A
    cout << s << endl;    // prints A value
    return 0;
}
```

It may be surprising to see a function call on the left-hand side of an assignment. However, nothing special is involved. The return value is a reference, so the call is an lvalue. Like any other lvalue, it may appear as the left-hand operand of the assignment operator.

If the return type is a reference to const, then (as usual) we may not assign to the result of the call:

```
shorterString("hi", "bye") = "X"; // error: return value is const
```

List Initializing the Return Value

C++ 11

Under the new standard, functions can return a braced list of values. As in any other return, the list is used to initialize the temporary that represents the function's return. If the list is empty, that temporary is value initialized (§ 3.3.1, p. 98). Otherwise, the value of the return depends on the function's return type.

As an example, recall the error_msg function from § 6.2.6 (p. 220). That function took a varying number of string arguments and printed an error message composed from the given strings. Rather than calling error_msg, in this function we'll return a vector that holds the error-message strings:

```
vector<string> process()
{
    // ...
    // expected and actual are strings
    if (expected.empty())
        return {};  // return an empty vector
    else if (expected == actual)
        return {"functionX", "okay"}; // return list-initialized vector
    else
        return {"functionX", expected, actual};
}
```

In the first return statement, we return an empty list. In this case, the vector that process returns will be empty. Otherwise, we return a vector initialized with two or three elements depending on whether expected and actual are equal.

In a function that returns a built-in type, a braced list may contain at most one value, and that value must not require a narrowing conversion (§ 2.2.1, p. 43). If the function returns a class type, then the class itself defines how the intiailizers are used (§ 3.3.1, p. 99).

Return from main

There is one exception to the rule that a function with a return type other than void must return a value: The main function is allowed to terminate without a return. If control reaches the end of main and there is no return, then the compiler implicitly inserts a return of 0.

As we saw in § 1.1 (p. 2), the value returned from main is treated as a status indicator. A zero return indicates success; most other values indicate failure. A nonzero value has a machine-dependent meaning. To make return values machine independent, the cstdlib header defines two preprocessor variables (§ 2.3.2, p. 54) that we can use to indicate success or failure:

```
int main()
{
    if (some_failure)
        return EXIT_FAILURE;  // defined in cstdlib
    else
        return EXIT_SUCCESS;  // defined in cstdlib
}
```

Because these are preprocessor variables, we must not precede them with std::, nor may we mention them in using declarations.

Recursion

A function that calls itself, either directly or indirectly, is a ***recursive function***. As an example, we can rewrite our factorial function to use recursion:

```
// calculate val!, which is 1 * 2 * 3 ... * val
int factorial(int val)
{
    if (val > 1)
        return factorial(val-1) * val;
    return 1;
}
```

In this implementation, we recursively call factorial to compute the factorial of the numbers counting down from the original value in val. Once we have reduced val to 1, we stop the recursion by returning 1.

There must always be a path through a recursive function that does not involve a recursive call; otherwise, the function will recurse "forever," meaning that the function will continue to call itself until the program stack is exhausted. Such

functions are sometimes described as containing a **recursion loop**. In the case of `factorial`, the stopping condition occurs when `val` is `1`.

The following table traces the execution of `factorial` when passed the value 5.

Trace of `factorial(5)`

Call	Returns	Value
`factorial(5)`	`factorial(4) * 5`	`120`
`factorial(4)`	`factorial(3) * 4`	`24`
`factorial(3)`	`factorial(2) * 3`	`6`
`factorial(2)`	`factorial(1) * 2`	`2`
`factorial(1)`	`1`	`1`

The `main` function may *not* call itself.

EXERCISES SECTION 6.3.2

Exercise 6.30: Compile the version of `str_subrange` as presented on page 223 to see what your compiler does with the indicated errors.

Exercise 6.31: When is it valid to return a reference? A reference to `const`?

Exercise 6.32: Indicate whether the following function is legal. If so, explain what it does; if not, correct any errors and then explain it.

```
int &get(int *arry, int index) { return arry[index]; }
int main() {
    int ia[10];
    for (int i = 0; i != 10; ++i)
        get(ia, i) = i;
}
```

Exercise 6.33: Write a recursive function to print the contents of a `vector`.

Exercise 6.34: What would happen if the stopping condition in `factorial` were

```
if (val != 0)
```

Exercise 6.35: In the call to `factorial`, why did we pass `val - 1` rather than `val--`?

6.3.3 Returning a Pointer to an Array

Because we cannot copy an array, a function cannot return an array. However, a function can return a pointer or a reference to an array (§ 3.5.1, p. 114). Unfortunately, the syntax used to define functions that return pointers or references to

arrays can be intimidating. Fortunately, there are ways to simplify such declarations. The most straightforward way is to use a type alias (§ 2.5.1, p. 67):

```
typedef int arrT[10]; // arrT is a synonym for the type array of ten ints
using arrT = int[10]; // equivalent declaration of arrT; see § 2.5.1 (p. 68)
arrT* func(int i);    // func returns a pointer to an array of ten ints
```

Here `arrT` is a synonym for an array of ten `int`s. Because we cannot return an array, we define the return type as a pointer to this type. Thus, `func` is a function that takes a single `int` argument and returns a pointer to an array of ten `int`s.

Declaring a Function That Returns a Pointer to an Array

To declare `func` without using a type alias, we must remember that the dimension of an array follows the name being defined:

```
int arr[10];           // arr is an array of ten ints
int *p1[10];           // p1 is an array of ten pointers
int (*p2)[10] = &arr; // p2 points to an array of ten ints
```

As with these declarations, if we want to define a function that returns a pointer to an array, the dimension must follow the function's name. However, a function includes a parameter list, which also follows the name. The parameter list precedes the dimension. Hence, the form of a function that returns a pointer to an array is:

Type `(*`*function*`(`*parameter_list*`))[`*dimension*`]`

As in any other array declaration, *Type* is the type of the elements and *dimension* is the size of the array. The parentheses around `(*`*function*`(`*parameter_list*`))` are necessary for the same reason that they were required when we defined `p2`. Without them, we would be defining a function that returns an array of pointers.

As a concrete example, the following declares `func` without using a type alias:

```
int (*func(int i))[10];
```

To understand this declaration, it can be helpful to think about it as follows:

- `func(int)` says that we can call `func` with an `int` argument.
- `(*func(int))` says we can dereference the result of that call.
- `(*func(int))[10]` says that dereferencing the result of a call to `func` yields an array of size ten.
- `int (*func(int))[10]` says the element type in that array is `int`.

Using a Trailing Return Type

C++ 11

Under the new standard, another way to simplify the declaration of `func` is by using a **trailing return type**. Trailing returns can be defined for any function, but are most useful for functions with complicated return types, such as pointers (or references) to arrays. A trailing return type follows the parameter list and is preceded by `->`. To signal that the return follows the parameter list, we use `auto` where the return type ordinarily appears:

```
// fcn takes an int argument and returns a pointer to an array of ten ints
auto func(int i) -> int(*)[10];
```

Because the return type comes after the parameter list, it is easier to see that `func` returns a pointer and that that pointer points to an array of ten `int`s.

Using `decltype`

As another alternative, if we know the array(s) to which our function can return a pointer, we can use `decltype` to declare the return type. For example, the following function returns a pointer to one of two arrays, depending on the value of its parameter:

```
int odd[] = {1,3,5,7,9};
int even[] = {0,2,4,6,8};
// returns a pointer to an array of five int elements
decltype(odd) *arrPtr(int i)
{
    return (i % 2) ? &odd : &even; // returns a pointer to the array
}
```

The return type for `arrPtr` uses `decltype` to say that the function returns a pointer to whatever type `odd` has. That object is an array, so `arrPtr` returns a pointer to an array of five `int`s. The only tricky part is that we must remember that `decltype` does not automatically convert an array to its corresponding pointer type. The type returned by `decltype` is an array type, to which we must add a `*` to indicate that `arrPtr` returns a pointer.

Exercises Section 6.3.3

Exercise 6.36: Write the declaration for a function that returns a reference to an array of ten `string`s, without using either a trailing return, `decltype`, or a type alias.

Exercise 6.37: Write three additional declarations for the function in the previous exercise. One should use a type alias, one should use a trailing return, and the third should use `decltype`. Which form do you prefer and why?

Exercise 6.38: Revise the `arrPtr` function on to return a reference to the array.

6.4 Overloaded Functions

Functions that have the same name but different parameter lists and that appear in the same scope are **overloaded**. For example, in § 6.2.4 (p. 214) we defined several functions named `print`:

```
void print(const char *cp);
void print(const int *beg, const int *end);
void print(const int ia[], size_t size);
```

These functions perform the same general action but apply to different parameter types. When we call these functions, the compiler can deduce which function we want based on the argument type we pass:

```
int j[2] = {0,1};
print("Hello World");         //  calls print(const char*)
print(j, end(j) - begin(j)); //  calls print(const int*, size_t)
print(begin(j), end(j));      //  calls print(const int*, const int*)
```

Function overloading eliminates the need to invent—and remember—names that exist only to help the compiler figure out which function to call.

The `main` function may *not* be overloaded.

Defining Overloaded Functions

Consider a database application with several functions to find a record based on name, phone number, account number, and so on. Function overloading lets us define a collection of functions, each named `lookup`, that differ in terms of how they do the search. We can call `lookup` passing a value of any of several types:

```
Record lookup(const Account&);  //  find by Account
Record lookup(const Phone&);    //  find by Phone
Record lookup(const Name&);     //  find by Name

Account acct;
Phone phone;
Record r1 = lookup(acct);  //  call version that takes an Account
Record r2 = lookup(phone); //  call version that takes a Phone
```

Here, all three functions share the same name, yet they are three distinct functions. The compiler uses the argument type(s) to figure out which function to call.

Overloaded functions must differ in the number or the type(s) of their parameters. Each of the functions above takes a single parameter, but the parameters have different types.

It is an error for two functions to differ only in terms of their return types. If the parameter lists of two functions match but the return types differ, then the second declaration is an error:

```
Record lookup(const Account&);
bool lookup(const Account&);    //  error: only the return type is different
```

Determining Whether Two Parameter Types Differ

Two parameter lists can be identical, even if they don't look the same:

```
//  each pair declares the same function
Record lookup(const Account &acct);
Record lookup(const Account&); //  parameter names are ignored

typedef Phone Telno;
Record lookup(const Phone&);
Record lookup(const Telno&); //  Telno and Phone are the same type
```

In the first pair, the first declaration names its parameter. Parameter names are only a documentation aid. They do not change the parameter list.

In the second pair, it looks like the types are different, but `Telno` is not a new type; it is a synonym for `Phone`. A type alias (§ 2.5.1, p. 67) provides an alternative name for an existing type; it does not create a new type. Therefore, two parameters that differ only in that one uses an alias and the other uses the type to which the alias corresponds are not different.

Overloading and `const` Parameters

As we saw in § 6.2.3 (p. 212), top-level `const` (§ 2.4.3, p. 63) has no effect on the objects that can be passed to the function. A parameter that has a top-level `const` is indistinguishable from one without a top-level `const`:

```
Record lookup(Phone);
Record lookup(const Phone);   // redeclares Record lookup(Phone)
Record lookup(Phone*);
Record lookup(Phone* const); // redeclares Record lookup(Phone*)
```

In these declarations, the second declaration declares the same function as the first.

On the other hand, we can overload based on whether the parameter is a reference (or pointer) to the `const` or nonconst version of a given type; such `const`s are low-level:

```
// functions taking const and nonconst references or pointers have different parameters
// declarations for four independent, overloaded functions
Record lookup(Account&);        // function that takes a reference to Account
Record lookup(const Account&); // new function that takes a const reference
Record lookup(Account*);        // new function, takes a pointer to Account
Record lookup(const Account*); // new function, takes a pointer to const
```

In these cases, the compiler can use the `const`ness of the argument to distinguish which function to call. Because there is no conversion (§ 4.11.2, p. 162) *from* `const`, we can pass a `const` object (or a pointer to `const`) only to the version with a `const` parameter. Because there is a conversion *to* `const`, we can call either function on a nonconst object or a pointer to nonconst. However, as we'll see in § 6.6.1 (p. 246), the compiler will prefer the nonconst versions when we pass a nonconst object or pointer to nonconst.

`const_cast` and Overloading

In § 4.11.3 (p. 163) we noted that `const_cast`s are most useful in the context of overloaded functions. As one example, recall our `shorterString` function from § 6.3.2 (p. 224):

```
// return a reference to the shorter of two strings
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() <= s2.size() ? s1 : s2;
}
```

ADVICE: WHEN NOT TO OVERLOAD A FUNCTION NAME

Although overloading lets us avoid having to invent (and remember) names for common operations, we should only overload operations that actually do similar things. There are some cases where providing different function names adds information that makes the program easier to understand. Consider a set of functions that move the cursor on a `Screen`.

```
Screen& moveHome();
Screen& moveAbs(int, int);
Screen& moveRel(int, int, string direction);
```

It might at first seem better to overload this set of functions under the name `move`:

```
Screen& move();
Screen& move(int, int);
Screen& move(int, int, string direction);
```

However, by overloading these functions, we've lost information that was inherent in the function names. Although cursor movement is a general operation shared by all these functions, the specific nature of that movement is unique to each of these functions. `moveHome`, for example, represents a special instance of cursor movement. Whether to overload these functions depends on which of these two calls is easier to understand:

```
// which is easier to understand?
myScreen.moveHome(); // we think this one!
myScreen.move();
```

This function takes and returns references to `const string`. We can call the function on a pair of nonconst `string` arguments, but we'll get a reference to a `const string` as the result. We might want to have a version of `shorterString` that, when given nonconst arguments, would yield a plain reference. We can write this version of our function using a `const_cast`:

```
string &shorterString(string &s1, string &s2)
{
    auto &r = shorterString(const_cast<const string&>(s1),
                            const_cast<const string&>(s2));
    return const_cast<string&>(r);
}
```

This version calls the `const` version of `shorterString` by casting its arguments to references to `const`. That function returns a reference to a `const string`, which we know is bound to one of our original, nonconst arguments. Therefore, we know it is safe to cast that `string` back to a plain `string&` in the return.

Calling an Overloaded Function

Once we have defined a set of overloaded functions, we need to be able to call them with appropriate arguments. **Function matching** (also known as **overload resolution**) is the process by which a particular function call is associated with

a specific function from a set of overloaded functions. The compiler determines which function to call by comparing the arguments in the call with the parameters offered by each function in the overload set.

In many—probably most—cases, it is straightforward for a programmer to determine whether a particular call is legal and, if so, which function will be called. Often the functions in the overload set differ in terms of the number of arguments, or the types of the arguments are unrelated. In such cases, it is easy to determine which function is called. Determining which function is called when the overloaded functions have the same number of parameters and those parameters are related by conversions (§ 4.11, p. 159) can be less obvious. We'll look at how the compiler resolves calls involving conversions in § 6.6 (p. 242).

For now, what's important to realize is that for any given call to an overloaded function, there are three possible outcomes:

- The compiler finds exactly one function that is a **best match** for the actual arguments and generates code to call that function.
- There is no function with parameters that match the arguments in the call, in which case the compiler issues an error message that there was **no match**.
- There is more than one function that matches and none of the matches is clearly best. This case is also an error; it is an **ambiguous call**.

EXERCISES SECTION 6.4

Exercise 6.39: Explain the effect of the second declaration in each one of the following sets of declarations. Indicate which, if any, are illegal.

```
(a) int calc(int, int);
    int calc(const int, const int);
(b) int get();
    double get();
(c) int *reset(int *);
    double *reset(double *);
```

6.4.1 Overloading and Scope

Ordinarily, it is a bad idea to declare a function locally. However, to explain how scope interacts with overloading, we will violate this practice and use local function declarations.

Programmers new to C++ are often confused about the interaction between scope and overloading. However, overloading has no special properties with respect to scope: As usual, if we declare a name in an inner scope, that name *hides* uses of that name declared in an outer scope. Names do not overload across scopes:

```
string read();
void print(const string &);
void print(double);    // overloads the print function
void fooBar(int ival)
{
    bool read = false; // new scope: hides the outer declaration of read
    string s = read(); // error: read is a bool variable, not a function
    // bad practice: usually it's a bad idea to declare functions at local scope
    void print(int);   // new scope: hides previous instances of print
    print("Value: "); // error: print(const string &) is hidden
    print(ival);       // ok: print(int) is visible
    print(3.14);       // ok: calls print(int); print(double) is hidden
}
```

Most readers will not be surprised that the call to `read` is in error. When the compiler processes the call to `read`, it finds the local definition of `read`. That name is a `bool` variable, and we cannot call a `bool`. Hence, the call is illegal.

Exactly the same process is used to resolve the calls to `print`. The declaration of `print(int)` in `fooBar` hides the earlier declarations of `print`. It is as if there is only one `print` function available: the one that takes a single `int` parameter.

When we call `print`, the compiler first looks for a declaration of that name. It finds the local declaration for `print` that takes an `int`. Once a name is found, the compiler ignores uses of that name in any outer scope. Instead, the compiler assumes that the declaration it found is the one for the name we are using. What remains is to see if the use of the name is valid.

In C++, name lookup happens before type checking.

The first call passes a string literal, but the only declaration for `print` that is in scope has a parameter that is an `int`. A string literal cannot be converted to an `int`, so this call is an error. The `print(const string&)` function, which would have matched this call, is hidden and is not considered.

When we call `print` passing a `double`, the process is repeated. The compiler finds the local definition of `print(int)`. The `double` argument can be converted to an `int`, so the call is legal.

Had we declared `print(int)` in the same scope as the other `print` functions, then it would be another overloaded version of `print`. In that case, these calls would be resolved differently, because the compiler will see all three functions:

```
void print(const string &);
void print(double); // overloads the print function
void print(int);    // another overloaded instance
void fooBar2(int ival)
{
    print("Value: "); // calls print(const string &)
    print(ival);      // calls print(int)
    print(3.14);      // calls print(double)
}
```

6.5 Features for Specialized Uses

In this section we'll cover three function-related features that are useful in many, but not all, programs: default arguments, inline and `constexpr` functions, and some facilities that are often used during debugging.

6.5.1 Default Arguments

Some functions have parameters that are given a particular value in most, but not all, calls. In such cases, we can declare that common value as a **default argument** for the function. Functions with default arguments can be called with or without that argument.

For example, we might use a `string` to represent the contents of a window. By default, we might want the window to have a particular height, width, and background character. However, we might also want to allow users to pass values other than the defaults. To accommodate both default and specified values we would declare our function to define the window as follows:

```
typedef string::size_type sz;  // typedef see § 2.5.1 (p. 67)
string screen(sz ht = 24, sz wid = 80, char backgrnd = ' ');
```

Here we've provided a default for each parameter. A default argument is specified as an initializer for a parameter in the parameter list. We may define defaults for one or more parameters. However, if a parameter has a default argument, all the parameters that follow it must also have default arguments.

Calling Functions with Default Arguments

If we want to use the default argument, we omit that argument when we call the function. Because `screen` provides defaults for all of its parameters, we can call `screen` with zero, one, two, or three arguments:

```
string window;
window = screen();  // equivalent to screen(24,80,' ')
window = screen(66);// equivalent to screen(66,80,' ')
window = screen(66, 256);       // screen(66,256,' ')
window = screen(66, 256, '#'); // screen(66,256,'#')
```

Arguments in the call are resolved by position. The default arguments are used for the trailing (right-most) arguments of a call. For example, to override the default for `backgrnd`, we must also supply arguments for `ht` and `wid`:

```
window = screen(, , '?'); // error: can omit only trailing arguments
window = screen('?');     // calls screen('?',80,' ')
```

Note that the second call, which passes a single character value, is legal. Although legal, it is unlikely to be what was intended. The call is legal because `'?'` is a `char`, and a `char` can be converted (§ 4.11.1, p. 160) to the type of the left-most parameter. That parameter is `string::size_type`, which is an `unsigned` integral type. In this call, the `char` argument is implicitly converted to `string::size_type`,

and is passed as the argument to `height`. On our machine, `'?'` has the hexadecimal value `0x3F`, which is decimal `63`. Thus, this call passes `63` to the `height` parameter.

Part of the work of designing a function with default arguments is ordering the parameters so that those least likely to use a default value appear first and those most likely to use a default appear last.

Default Argument Declarations

Although it is normal practice to declare a function once inside a header, it is legal to redeclare a function multiple times. However, each parameter can have its default specified only once in a given scope. Thus, any subsequent declaration can add a default only for a parameter that has not previously had a default specified. As usual, defaults can be specified only if all parameters to the right already have defaults. For example, given

```
// no default for the height or width parameters
string screen(sz, sz, char = ' ');
```

we cannot change an already declared default value:

```
string screen(sz, sz, char = '*'); // error: redeclaration
```

but we can add a default argument as follows:

```
string screen(sz = 24, sz = 80, char);  // ok: adds default arguments
```

Default arguments ordinarily should be specified with the function declaration in an appropriate header.

Default Argument Initializers

Local variables may not be used as a default argument. Excepting that restriction, a default argument can be any expression that has a type that is convertible to the type of the parameter:

```
// the declarations of wd, def, and ht must appear outside a function
sz wd = 80;
char def = ' ';
sz ht();
string screen(sz = ht(), sz = wd, char = def);
string window = screen(); // calls screen(ht(), 80, ' ')
```

Names used as default arguments are resolved in the scope of the function declaration. The value that those names represent is evaluated at the time of the call:

```
void f2()
{
    def = '*';    // changes the value of a default argument
    sz wd = 100; // hides the outer definition of wd but does not change the default
    window = screen(); // calls screen(ht(), 80, '*')
}
```

Inside `f2`, we changed the value of `def`. The call to `screen` passes this updated value. Our function also declared a local variable that hides the outer `wd`. However, the local named `wd` is unrelated to the default argument passed to `screen`.

EXERCISES SECTION 6.5.1

Exercise 6.40: Which, if either, of the following declarations are errors? Why?

(a)
```
int ff(int a, int b = 0, int c = 0);
```
(b)
```
char *init(int ht = 24, int wd, char bckgrnd);
```

Exercise 6.41: Which, if any, of the following calls are illegal? Why? Which, if any, are legal but unlikely to match the programmer's intent? Why?

```
char *init(int ht, int wd = 80, char bckgrnd = ' ');
```

(a) `init();` (b) `init(24,10);` (c) `init(14, '*');`

Exercise 6.42: Give the second parameter of `make_plural` (§ 6.3.2, p. 224) a default argument of `'s'`. Test your program by printing singular and plural versions of the words `success` and `failure`.

6.5.2 Inline and `constexpr` Functions

In § 6.3.2 (p. 224) we wrote a small function that returned a reference to the shorter of its two `string` parameters. The benefits of defining a function for such a small operation include the following:

- It is easier to read and understand a call to `shorterString` than it would be to read and understand the equivalent conditional expression.
- Using a function ensures uniform behavior. Each test is guaranteed to be done the same way.
- If we need to change the computation, it is easier to change the function than to find and change every occurrence of the equivalent expression.
- The function can be reused rather than rewritten for other applications.

There is, however, one potential drawback to making `shorterString` a function: Calling a function is apt to be slower than evaluating the equivalent expression. On most machines, a function call does a lot of work: Registers are saved before the call and restored after the return; arguments may be copied; and the program branches to a new location.

`inline` Functions Avoid Function Call Overhead

A function specified as **`inline`** (usually) is expanded "in line" at each call. If `shorterString` were defined as `inline`, then this call

```
cout << shorterString(s1, s2) << endl;
```

(probably) would be expanded during compilation into something like

```
cout << (s1.size() < s2.size() ? s1 : s2) << endl;
```

The run-time overhead of making `shorterString` a function is thus removed.

We can define `shorterString` as an inline function by putting the keyword `inline` before the function's return type:

```
// inline version: find the shorter of two strings
inline const string &
shorterString(const string &s1, const string &s2)
{
        return s1.size() <= s2.size() ? s1 : s2;
}
```

The `inline` specification is only a *request* to the compiler. The compiler may choose to ignore this request.

In general, the `inline` mechanism is meant to optimize small, straight-line functions that are called frequently. Many compilers will not inline a recursive function. A 75-line function will almost surely not be expanded inline.

`constexpr` Functions

A **`constexpr` function** is a function that can be used in a constant expression (§ 2.4.4, p. 65). A `constexpr` function is defined like any other function but must meet certain restrictions: The `return` type and the type of each parameter in a must be a literal type (§ 2.4.4, p. 66), and the function body must contain exactly one `return` statement:

C++ 11

```
constexpr int new_sz() { return 42; }
constexpr int foo = new_sz();  // ok: foo is a constant expression
```

Here we defined `new_sz` as a `constexpr` that takes no arguments. The compiler can verify—at compile time—that a call to `new_sz` returns a constant expression, so we can use `new_sz` to initialize our `constexpr` variable, `foo`.

When it can do so, the compiler will replace a call to a `constexpr` function with its resulting value. In order to be able to expand the function immediately, `constexpr` functions are implicitly `inline`.

A `constexpr` function body may contain other statements so long as those statements generate no actions at run time. For example, a `constexpr` function may contain null statements, type aliases (§ 2.5.1, p. 67), and `using` declarations.

A `constexpr` function is permitted to return a value that is not a constant:

```
// scale(arg) is a constant expression if arg is a constant expression
constexpr size_t scale(size_t cnt) { return new_sz() * cnt; }
```

The `scale` function will return a constant expression if its argument is a constant expression but not otherwise:

```
int arr[scale(2)]; //  ok: scale(2) is a constant expression
int i = 2;         //  i is not a constant expression
int a2[scale(i)];  //  error: scale(i) is not a constant expression
```

When we pass a constant expression—such as the literal `2`—then the return is a constant expression. In this case, the compiler will replace the call to `scale` with the resulting value.

If we call `scale` with an expression that is not a constant expression—such as on the `int` object `i`—then the return is not a constant expression. If we use `scale` in a context that requires a constant expression, the compiler checks that the result is a constant expression. If it is not, the compiler will produce an error message.

A `constexpr` function is not required to return a constant expression.

Put `inline` and `constexpr` Functions in Header Files

Unlike other functions, `inline` and `constexpr` functions may be defined multiple times in the program. After all, the compiler needs the definition, not just the declaration, in order to expand the code. However, all of the definitions of a given `inline` or `constexpr` must match exactly. As a result, `inline` and `constexpr` functions normally are defined in headers.

Exercises Section 6.5.2

Exercise 6.43: Which one of the following declarations and definitions would you put in a header? In a source file? Explain why.

(a) `inline bool eq(const BigInt&, const BigInt&) {...}`
(b) `void putValues(int *arr, int size);`

Exercise 6.44: Rewrite the `isShorter` function from § 6.2.2 (p. 211) to be `inline`.

Exercise 6.45: Review the programs you've written for the earlier exercises and decide whether they should be defined as `inline`. If so, do so. If not, explain why they should not be `inline`.

Exercise 6.46: Would it be possible to define `isShorter` as a `constexpr`? If so, do so. If not, explain why not.

6.5.3 Aids for Debugging

C++ programmers sometimes use a technique similar to header guards (§ 2.6.3, p. 77) to conditionally execute debugging code. The idea is that the program will contain debugging code that is executed only while the program is being developed. When the application is completed and ready to ship, the debugging code is turned off. This approach uses two preprocessor facilities: `assert` and `NDEBUG`.

The `assert` Preprocessor Macro

`assert` is a **preprocessor macro**. A preprocessor macro is a preprocessor variable that acts somewhat like an inline function. The `assert` macro takes a single expression, which it uses as a condition:

```
assert(expr);
```

evaluates *expr* and if the expression is false (i.e., zero), then `assert` writes a message and terminates the program. If the expression is true (i.e., is nonzero), then `assert` does nothing.

The `assert` macro is defined in the `cassert` header. As we've seen, preprocessor names are managed by the preprocessor not the compiler (§ 2.3.2, p. 54). As a result, we use preprocessor names directly and do not provide a `using` declaration for them. That is, we refer to `assert`, not `std::assert`, and provide no `using` declaration for `assert`.

As with preprocessor variables, macro names must be unique within the program. Programs that include the `cassert` header may not define a variable, function, or other entity named `assert`. In practice, it is a good idea to avoid using the name `assert` for our own purposes even if we don't include `cassert`. Many headers include the `cassert` header, which means that even if you don't directly include that file, your programs are likely to have it included anyway.

The `assert` macro is often used to check for conditions that "cannot happen." For example, a program that does some manipulation of input text might know that all words it is given are always longer than a threshold. That program might contain a statement such as

```
assert(word.size() > threshold);
```

The `NDEBUG` Preprocessor Variable

The behavior of `assert` depends on the status of a preprocessor variable named `NDEBUG`. If `NDEBUG` is defined, `assert` does nothing. By default, `NDEBUG` is *not* defined, so, by default, `assert` performs a run-time check.

We can "turn off" debugging by providing a `#define` to define `NDEBUG`. Alternatively, most compilers provide a command-line option that lets us define preprocessor variables:

```
$ CC -D NDEBUG main.C  # use /D with the Microsoft compiler
```

has the same effect as writing `#define NDEBUG` at the beginning of `main.C`.

If `NDEBUG` is defined, we avoid the potential run-time overhead involved in checking various conditions. Of course, there is also no run-time check. Therefore, `assert` should be used only to verify things that truly should not be possible. It can be useful as an aid in getting a program debugged but should not be used to substitute for run-time logic checks or error checking that the program should do.

In addition to using `assert`, we can write our own conditional debugging code using `NDEBUG`. If `NDEBUG` is *not* defined, the code between the `#ifndef` and the `#endif` is executed. If `NDEBUG` *is* defined, that code is ignored:

```
void print(const int ia[], size_t size)
{
#ifndef NDEBUG
// __func__ is a local static defined by the compiler that holds the function's name
cerr << __func__ << ": array size is " << size << endl;
#endif
// ...
```

Here we use a variable named `__func__` to print the name of the function we are debugging. The compiler defines `__func__` in every function. It is a local `static` array of `const char` that holds the name of the function.

In addition to `__func__`, which the C++ compiler defines, the preprocessor defines four other names that can be useful in debugging:

`__FILE__` string literal containing the name of the file

`__LINE__` integer literal containing the current line number

`__TIME__` string literal containing the time the file was compiled

`__DATE__` string literal containing the date the file was compiled

We might use these constants to report additional information in error messages:

```
if (word.size() < threshold)
    cerr << "Error: " << __FILE__
         << " : in function " << __func__
         << " at line " << __LINE__ << endl
         << "       Compiled on " << __DATE__
         << " at " << __TIME__ << endl
         << "       Word read was \"" << word
         << "\":  Length too short" << endl;
```

If we give this program a `string` that is shorter than the `threshold`, then the following error message will be generated:

```
Error: wdebug.cc : in function main at line 27
       Compiled on Jul 11 2012 at 20:50:03
       Word read was "foo":  Length too short
```

6.6 Function Matching

In many (if not most) cases, it is easy to figure out which overloaded function matches a given call. However, it is not so simple when the overloaded functions have the same number of parameters and when one or more of the parameters have types that are related by conversions. As an example, consider the following set of functions and function call:

```
void f();
void f(int);
void f(int, int);
void f(double, double = 3.14);
f(5.6);  // calls void f(double, double)
```

EXERCISES SECTION 6.5.3

Exercise 6.47: Revise the program you wrote in the exercises in § 6.3.2 (p. 228) that used recursion to print the contents of a `vector` to conditionally print information about its execution. For example, you might print the size of the `vector` on each call. Compile and run the program with debugging turned on and again with it turned off.

Exercise 6.48: Explain what this loop does and whether it is a good use of `assert`:

```
string s;
while (cin >> s && s != sought) { }  // empty body
assert(cin);
```

Determining the Candidate and Viable Functions

The first step of function matching identifies the set of overloaded functions considered for the call. The functions in this set are the **candidate functions**. A candidate function is a function with the same name as the called function and for which a declaration is visible at the point of the call. In this example, there are four candidate functions named `f`.

The second step selects from the set of candidate functions those functions that can be called with the arguments in the given call. The selected functions are the **viable functions**. To be viable, a function must have the same number of parameters as there are arguments in the call, and the type of each argument must match—or be convertible to—the type of its corresponding parameter.

We can eliminate two of our candidate functions based on the number of arguments. The function that has no parameters and the one that has two `int` parameters are not viable for this call. Our call has only one argument, and these functions have zero and two parameters, respectively.

The function that takes a single `int` and the function that takes two `doubles` might be viable. Either of these functions can be called with a single argument. The function taking two `doubles` has a default argument, which means it can be called with a single argument.

When a function has default arguments (§ 6.5.1, p. 236), a call may appear to have fewer arguments than it actually does.

Having used the number of arguments to winnow the candidate functions, we next look at whether the argument types match those of the parameters. As with any call, an argument might match its parameter either because the types match exactly or because there is a conversion from the argument type to the type of the parameter. In this example, both of our remaining functions are viable:

- `f(int)` is viable because a conversion exists that can convert the argument of type `double` to the parameter of type `int`.
- `f(double, double)` is viable because a default argument is provided for the function's second parameter and its first parameter is of type `double`, which exactly matches the type of the argument in the call.

If there are no viable functions, the compiler will complain that there is no matching function.

Finding the Best Match, If Any

The third step of function matching determines which viable function provides the best match for the call. This process looks at each argument in the call and selects the viable function (or functions) for which the corresponding parameter best matches the argument. We'll explain the details of "best" in the next section, but the idea is that the closer the types of the argument and parameter are to each other, the better the match.

In our case, there is only one (explicit) argument in the call. That argument has type `double`. To call `f(int)`, the argument would have to be converted from `double` to `int`. The other viable function, `f(double, double)`, is an exact match for this argument. An exact match is better than a match that requires a conversion. Therefore, the compiler will resolve the call `f(5.6)` as a call to the function that has two `double` parameters. The compiler will add the default argument for the second, missing argument.

Function Matching with Multiple Parameters

Function matching is more complicated if there are two or more arguments. Given the same functions named `f`, let's analyze the following call:

```
f(42, 2.56);
```

The set of viable functions is selected in the same way as when there is only one parameter. The compiler selects those functions that have the required number of parameters and for which the argument types match the parameter types. In this case, the viable functions are `f(int, int)` and `f(double, double)`. The compiler then determines, argument by argument, which function is (or functions are) the best match. There is an overall best match if there is one and only one function for which

- The match for each argument is no worse than the match required by any other viable function
- There is at least one argument for which the match is better than the match provided by any other viable function

If after looking at each argument there is no single function that is preferable, then the call is in error. The compiler will complain that the call is ambiguous.

In this call, when we look only at the first argument, we find that the function `f(int, int)` is an exact match. To match the second function, the `int` argument `42` must be converted to `double`. A match through a built-in conversion is "less good" than one that is exact. Considering only the first argument, `f(int, int)` is a better match than `f(double, double)`.

When we look at the second argument, `f(double, double)` is an exact match to the argument `2.56`. Calling `f(int, int)` would require that `2.56` be converted from `double` to `int`. When we consider only the second parameter, the function `f(double, double)` is a better match.

The compiler will reject this call because it is ambiguous: Each viable function is a better match than the other on one of the arguments to the call. It might be tempting to force a match by explicitly casting (§ 4.11.3, p. 162) one of our arguments. However, in well-designed systems, argument casts should not be necessary.

Casts should not be needed to call an overloaded function. The need for a cast suggests that the parameter sets are designed poorly.

Exercises Section 6.6

Exercise 6.49: What is a candidate function? What is a viable function?

Exercise 6.50: Given the declarations for `f` from page 242, list the viable functions, if any for each of the following calls. Indicate which function is the best match, or if the call is illegal whether there is no match or why the call is ambiguous.

(a) `f(2.56, 42)` (b) `f(42)` (c) `f(42, 0)` (d) `f(2.56, 3.14)`

Exercise 6.51: Write all four versions of `f`. Each function should print a distinguishing message. Check your answers for the previous exercise. If your answers were incorrect, study this section until you understand why your answers were wrong.

6.6.1 Argument Type Conversions

In order to determine the best match, the compiler ranks the conversions that could be used to convert each argument to the type of its corresponding parameter. Conversions are ranked as follows:

1. An exact match. An exact match happens when:
   - The argument and parameter types are identical.
   - The argument is converted from an array or function type to the corresponding pointer type. (§ 6.7 (p. 247) covers function pointers.)
   - A top-level `const` is added to or discarded from the argument.
2. Match through a `const` conversion (§ 4.11.2, p. 162).
3. Match through a promotion (§ 4.11.1, p. 160).
4. Match through an arithmetic (§ 4.11.1, p. 159) or pointer conversion (§ 4.11.2, p. 161).
5. Match through a class-type conversion. (§ 14.9 (p. 579) covers these conversions.)

Matches Requiring Promotion or Arithmetic Conversion

Promotions and conversions among the built-in types can yield surprising results in the context of function matching. Fortunately, well-designed systems rarely include functions with parameters as closely related as those in the following examples.

In order to analyze a call, it is important to remember that the small integral types always promote to `int` or to a larger integral type. Given two functions, one of which takes an `int` and the other a `short`, the `short` version will be called only on values of type `short`. Even though the smaller integral values might appear to be a closer match, those values are promoted to `int`, whereas calling the `short` version would require a conversion:

```
void ff(int);
void ff(short);
ff('a');    // char promotes to int; calls f(int)
```

All the arithmetic conversions are treated as equivalent to each other. The conversion from `int` to `unsigned int`, for example, does not take precedence over the conversion from `int` to `double`. As a concrete example, consider

```
void manip(long);
void manip(float);
manip(3.14); // error: ambiguous call
```

The literal `3.14` is a `double`. That type can be converted to either `long` or `float`. Because there are two possible arithmetic conversions, the call is ambiguous.

Function Matching and `const` Arguments

When we call an overloaded function that differs on whether a reference or pointer parameter refers or points to `const`, the compiler uses the `const`ness of the argument to decide which function to call:

```
Record lookup(Account&);        // function that takes a reference to Account
Record lookup(const Account&); // new function that takes a const reference
const Account a;
Account b;

lookup(a);    // calls lookup(const Account&)
lookup(b);    // calls lookup(Account&)
```

In the first call, we pass the `const` object `a`. We cannot bind a plain reference to a `const` object. In this case the only viable function is the version that takes a reference to `const`. Moreover, that call is an exact match to the argument `a`.

In the second call, we pass the nonconst object `b`. For this call, both functions are viable. We can use `b` to initialize a reference to either `const` or nonconst type. However, initializing a reference to `const` from a nonconst object requires a conversion. The version that takes a nonconst parameter is an exact match for `b`. Hence, the nonconst version is preferred.

Pointer parameters work in a similar way. If two functions differ only as to whether a pointer parameter points to `const` or nonconst, the compiler can distinguish which function to call based on the `const`ness of the argument: If the argument is a pointer to `const`, the call will match the function that takes a `const*`; otherwise, if the argument is a pointer to nonconst, the function taking a plain pointer is called.

EXERCISES SECTION 6.6.1

Exercise 6.52: Given the following declarations,

```
void manip(int, int);
double dobj;
```

what is the rank (§ 6.6.1, p. 245) of each conversion in the following calls?

(a) `manip('a', 'z');` (b) `manip(55.4, dobj);`

Exercise 6.53: Explain the effect of the second declaration in each one of the following sets of declarations. Indicate which, if any, are illegal.

```
(a) int calc(int&, int&);
    int calc(const int&, const int&);
(b) int calc(char*, char*);
    int calc(const char*, const char*);
(c) int calc(char*, char*);
    int calc(char* const, char* const);
```

6.7 Pointers to Functions

A function pointer is just that—a pointer that denotes a function rather than an object. Like any other pointer, a function pointer points to a particular type. A function's type is determined by its return type and the types of its parameters. The function's name is not part of its type. For example:

```
// compares lengths of two strings
bool lengthCompare(const string &, const string &);
```

has type `bool(const string&, const string&)`. To declare a pointer that can point at this function, we declare a pointer in place of the function name:

```
// pf points to a function returning bool that takes two const string references
bool (*pf)(const string &, const string &);  // uninitialized
```

Starting from the name we are declaring, we see that `pf` is preceded by a `*`, so `pf` is a pointer. To the right is a parameter list, which means that `pf` points to a function. Looking left, we find that the type the function returns is `bool`. Thus, `pf` points to a function that has two `const string&` parameters and returns `bool`.

The parentheses around `*pf` are necessary. If we omit the parentheses, then we declare `pf` as a function that returns a pointer to `bool`:

```
// declares a function named pf that returns a bool*
bool *pf(const string &, const string &);
```

Using Function Pointers

When we use the name of a function as a value, the function is automatically converted to a pointer. For example, we can assign the address of `lengthCompare` to `pf` as follows:

```
pf = lengthCompare;  // pf now points to the function named lengthCompare
pf = &lengthCompare; // equivalent assignment: address-of operator is optional
```

Moreover, we can use a pointer to a function to call the function to which the pointer points. We can do so directly—there is no need to dereference the pointer:

```
bool b1 = pf("hello", "goodbye");     // calls lengthCompare
bool b2 = (*pf)("hello", "goodbye"); // equivalent call
bool b3 = lengthCompare("hello", "goodbye");  // equivalent call
```

There is no conversion between pointers to one function type and pointers to another function type. However, as usual, we can assign `nullptr` (§ 2.3.2, p. 53) or a zero-valued integer constant expression to a function pointer to indicate that the pointer does not point to any function:

```
string::size_type sumLength(const string&, const string&);
bool cstringCompare(const char*, const char*);

pf = 0;               // ok: pf points to no function
pf = sumLength;       // error: return type differs
pf = cstringCompare; // error: parameter types differ
pf = lengthCompare;  // ok: function and pointer types match exactly
```

Pointers to Overloaded Functions

As usual, when we use an overloaded function, the context must make it clear which version is being used. When we declare a pointer to an overloaded function

```
void ff(int*);
void ff(unsigned int);

void (*pf1)(unsigned int) = ff;  // pf1 points to ff(unsigned)
```

the compiler uses the type of the pointer to determine which overloaded function to use. The type of the pointer must match one of the overloaded functions exactly:

```
void (*pf2)(int) = ff;     // error: no ff with a matching parameter list
double (*pf3)(int*) = ff; // error: return type of ff and pf3 don't match
```

Function Pointer Parameters

Just as with arrays (§ 6.2.4, p. 214), we cannot define parameters of function type but can have a parameter that is a pointer to function. As with arrays, we can write a parameter that looks like a function type, but it will be treated as a pointer:

```
// third parameter is a function type and is automatically treated as a pointer to function
void useBigger(const string &s1, const string &s2,
               bool pf(const string &, const string &));
// equivalent declaration: explicitly define the parameter as a pointer to function
void useBigger(const string &s1, const string &s2,
               bool (*pf)(const string &, const string &));
```

When we pass a function as an argument, we can do so directly. It will be automatically converted to a pointer:

```
// automatically converts the function lengthCompare to a pointer to function
useBigger(s1, s2, lengthCompare);
```

As we've just seen in the declaration of `useBigger`, writing function pointer types quickly gets tedious. Type aliases (§ 2.5.1, p. 67), along with `decltype` (§ 2.5.3, p. 70), let us simplify code that uses function pointers:

```
// Func and Func2 have function type
typedef bool Func(const string&, const string&);
typedef decltype(lengthCompare) Func2; // equivalent type
// FuncP and FuncP2 have pointer to function type
typedef bool(*FuncP)(const string&, const string&);
typedef decltype(lengthCompare) *FuncP2;  // equivalent type
```

Here we've used `typedef` to define our types. Both `Func` and `Func2` are function types, whereas `FuncP` and `FuncP2` are pointer types. It is important to note that `decltype` returns the function type; the automatic conversion to pointer is not done. Because `decltype` returns a function type, if we want a pointer we must add the `*` ourselves. We can redeclare `useBigger` using any of these types:

```
// equivalent declarations of useBigger using type aliases
void useBigger(const string&, const string&, Func);
void useBigger(const string&, const string&, FuncP2);
```

Both declarations declare the same function. In the first case, the compiler will automatically convert the function type represented by `Func` to a pointer.

Returning a Pointer to Function

As with arrays (§ 6.3.3, p. 228), we can't return a function type but can return a pointer to a function type. Similarly, we must write the return type as a pointer type; the compiler will not automatically treat a function return type as the corresponding pointer type. Also as with array returns, by far the easiest way to declare a function that returns a pointer to function is by using a type alias:

```
using F = int(int*, int);      // F is a function type, not a pointer
using PF = int(*)(int*, int); // PF is a pointer type
```

Here we used type alias declarations (§ 2.5.1, p. 68) to define `F` as a function type and `PF` as a pointer to function type. The thing to keep in mind is that, unlike what happens to parameters that have function type, the return type is not automatically converted to a pointer type. We must explicitly specify that the return type is a pointer type:

```
PF f1(int);   // ok: PF is a pointer to function; f1 returns a pointer to function
F f1(int);    // error: F is a function type; f1 can't return a function
F *f1(int);   // ok: explicitly specify that the return type is a pointer to function
```

Of course, we can also declare `f1` directly, which we'd do as

```
int (*f1(int))(int*, int);
```

Reading this declaration from the inside out, we see that `f1` has a parameter list, so `f1` is a function. `f1` is preceded by a `*` so `f1` returns a pointer. The type of that pointer itself has a parameter list, so the pointer points to a function. That function returns an `int`.

For completeness, it's worth noting that we can simplify declarations of functions that return pointers to function by using a trailing return (§ 6.3.3, p. 229):

```
auto f1(int) -> int (*)(int*, int);
```

Using `auto` or `decltype` for Function Pointer Types

If we know which function(s) we want to return, we can use `decltype` to simplify writing a function pointer return type. For example, assume we have two functions, both of which return a `string::size_type` and have two `const string&` parameters. We can write a third function that takes a `string` parameter and returns a pointer to one of these two functions as follows:

```
string::size_type sumLength(const string&, const string&);
string::size_type largerLength(const string&, const string&);
// depending on the value of its string parameter,
// getFcn returns a pointer to sumLength or to largerLength
decltype(sumLength) *getFcn(const string &);
```

The only tricky part in declaring `getFcn` is to remember that when we apply `decltype` to a function, it returns a function type, not a pointer to function type. We must add a `*` to indicate that we are returning a pointer, not a function.

Exercises Section 6.7

Exercise 6.54: Write a declaration for a function that takes two `int` parameters and returns an `int`, and declare a `vector` whose elements have this function pointer type.

Exercise 6.55: Write four functions that add, subtract, multiply, and divide two `int` values. Store pointers to these functions in your `vector` from the previous exercise.

Exercise 6.56: Call each element in the `vector` and print their result.

Chapter Summary

Functions are named units of computation and are essential to structuring even modest programs. Every function has a return type, a name, a (possibly empty) list of parameters, and a function body. The function body is a block that is executed when the function is called. When a function is called, the arguments passed to the function must be compatible with the types of the corresponding parameters.

In C++, functions may be overloaded: The same name may be used to define different functions as long as the number or types of the parameters in the functions differ. The compiler automatically figures out which function to call based on the arguments in a call. The process of selecting the right function from a set of overloaded functions is referred to as function matching.

Defined Terms

ambiguous call Compile-time error that results during function matching when two or more functions provide an equally good match for a call.

arguments Values supplied in a function call that are used to initialize the function's parameters.

assert Preprocessor macro that takes a single expression, which it uses as a condition. When the preprocessor variable `NDEBUG` is not defined, `assert` evaluates the condition and, if the condition is false, writes a message and terminates the program.

automatic objects Objects that exist only during the execution of a function. They are created when control passes through their definition and are destroyed at the end of the block in which they are defined.

best match Function selected from a set of overloaded functions for a call. If a best match exists, the selected function is a better match than all the other viable candidates for at least one argument in the call and is no worse on the rest of the arguments.

call by reference See pass by reference.

call by value See pass by value.

candidate functions Set of functions that are considered when resolving a function call. The candidate functions are all the functions with the name used in the call for which a declaration is in scope at the time of the call.

constexpr Function that may return a constant expression. A `constexpr` function is implicitly `inline`.

default argument Value specified to be used when an argument is omitted in a call to the function.

executable file File, which the operating system executes, that contains code corresponding to our program.

function Callable unit of computation.

function body Block that defines the actions of a function.

function matching Compiler process by which a call to an overloaded function is resolved. Arguments used in the call are compared to the parameter list of each overloaded function.

function prototype Function declaration, consisting of the name, return type, and parameter types of a function. To call a function, its prototype must have been declared before the point of call.

hidden names Names declared inside a scope hide previously declared entities with the same names declared outside that scope.

initializer_list Library class that represents a comma-separated list of objects of a single type enclosed inside curly braces.

inline function Request to the compiler to expand a function at the point of call, if possible. Inline functions avoid the normal function-calling overhead.

link Compilation step in which multiple object files are put together to form an executable program.

local static objects Local objects whose value persists across calls to the function. Local `static` objects that are created and initialized before control reaches their use and are destroyed when the program ends.

local variables Variables defined inside a block.

no match Compile-time error that results during function matching when there is no function with parameters that match the arguments in a given call.

object code Format into which the compiler transforms our source code.

object file File holding object code generated by the compiler from a given source file. An executable file is generated from one or more object files after the files are linked together.

object lifetime Every object has an associated lifetime. Nonstatic objects that are defined inside a block exist from when their definition is encountered until the end of the block in which they are defined. Global objects are created during program startup. Local `static` objects are created before the first time execution passes through the object's definition. Global objects and local `static` objects are destroyed when the `main` function ends.

overload resolution See function matching.

overloaded function Function that has the same name as at least one other function. Overloaded functions must differ in the number or type of their parameters.

parameters Local variables declared inside the function parameter list. Parameters are initialized by the arguments provided in each function call.

pass by reference Description of how arguments are passed to parameters of reference type. Reference parameters work the same way as any other use of references; the parameter is bound to its corresponding argument.

pass by value How arguments are passed to parameters of a nonreference type. A nonreference parameter is a copy of the value of its corresponding argument.

preprocessor macro Preprocessor facility that behaves like an inline function. Aside from `assert`, modern C++ programs make very little use of preprocessor macros.

recursion loop Description of a recursive function that omits a stopping condition and which calls itself until exhasuting the program stack.

recursive function Function that calls itself directly or indirectly.

return type Part of a function declaration that specifies the type of the value that the function returns.

separate compilation Ability to split a program into multiple separate source files.

trailing return type Return type specified after the parameter list.

viable functions Subset of the candidate functions that could match a given call. Viable functions have the same number of parameters as arguments to the call, and each argument type can be converted to the corresponding parameter type.

`()` operator Call operator. Executes a function. The name of a function or a function pointer precedes the parentheses, which enclose a (possibly empty) comma-separated list of arguments.