Programming in C++

Rex Jaeschke

Programming in C++

© 1998, 2001, 2005, 2007–2008, 2009, 2014, 2020 Rex Jaeschke.

Edition: 3.2

The training materials associated with this book are available for license. Interested parties should contact the author at the address below.

Please address comments, corrections, and questions to the author, Rex Jaeschke, at rex@RexJaeschke.com.

Much of the information in this book has been published previously, as follows:

1. A series of monthly columns called "Let's C Now" in *DEC Professional*, Professional Press, running from 1990–1993.
2. The book "C++: An Introduction for Experienced C Programmers", Rex Jaeschke, CBM Books 1993, ISBN 1-878956-27-2.

For the C language and library subset of C++, the reader may wish to refer to my book, "The Dictionary of Standard C", Prentice Hall 2001, ISBN 0-13-090620-4. This dictionary can be viewed on-line, free of charge, at www.prenhall.com/jaeschke. An earlier version of this dictionary, now out of print, was published by Professional Press Books 1991, ISBN 0-07-707657-5.

Preface

Introduction

Welcome to the world of C++. Throughout this book, we look at the statements and constructs of the C++ programming language. Each statement and construct is introduced by example with corresponding explanations, and, except where errors are intentional, the examples are complete programs or subroutines that are error-free. I encourage you to run and modify these examples, because the only realistic way to master a language is to write programs in it, run them, and debug them.

This book was written with teaching in mind, and evolved from an earlier series on C. It is intended for use both in a classroom environment as well as for self-paced learning.

C++ is a general-purpose, high-level language that supports object-oriented programming (OOP). From a grammatical viewpoint, C++ is a relatively simple language having about 75 keywords. Because of the power of the basic statements and constructs, and the fact that the programmer can effectively extend the language by using callable functions while still maintaining portability, there is little need for language extensions especially considering the fact that C++ supports calls to procedures written in other languages. However, extensions do exist.

Depending on their language backgrounds, programmers new to C++ may initially find programs hard to read because, traditionally, most C++ code is written in lowercase. Lower- and uppercase letters are treated by the compiler as distinct. In fact, C++ language keywords must be written in lowercase. Keywords are also reserved words.

If you have ever wondered what all those special punctuation marks on most keyboards are for, the answer may well be "to write C++ programs!" C++ uses almost all of them for one reason or another and sometimes combines them for yet other purposes.

C++ supports a structured, modular approach to programming using callable subroutines, called *functions*. Source files can be compiled separately, with external references being resolved at runtime.

C++ lends itself to writing terse code. However, there is a fine (and subjective) line between writing code that is terse and code that is cryptic. It is easy to write code that is unreadable and, therefore, unmaintainable. In fact, that's what you will get by default. However, with care and a small dose of discipline, you can produce nicely formatted code that is easy to read and maintain. However, good code doesn't happen automatically—you have to work at it. Throughout the book, I make numerous comments and suggestions regarding style. Perhaps the best advice I can give in that regard is, "Remember that the poor fool who has to read your code in the future just might be you!" Above all, be overt and be consistent.

Numerous "Tips" and "Style Tips" have been added throughout to highlight important suggestions. Considerable attention has been paid to using and teaching a consistent, popular, and overt programming style. Both kinds of tips are highlighted so they stand out. For example:

Tip: Avoid initializing enumerators explicitly or making duplicates unless you have good reason to do so. Assigning a range of values that is non-continuous or that contains duplicates significantly reduces the set of operations that can meaningfully be performed.

Style Tip: Use liberal amounts of white space to improve program readability. The compiler discards all white space, so its presence has no effect on program execution. Apart from separating tokens, white space exists solely for the benefit of the reader. If you can't read the code, you surely won't be able to understand it.

C++ is not all things to all people; nor does it claim to be. For many applications, other languages will suffice. In any event, compared to other high-level languages (including Java and C#), in my opinion, C++ is a relatively expensive language to learn and master.

Reader Assumptions

This is *not* a first course in programming. (Nor do I recommend C++ as a first programming language.)

I assume that you know how to use your particular text editor, C++ compiler, and debugger. Comments on the use of these utility programs will be limited to points of particular interest to the C++ programmer.

To fully understand and exploit the material, you should be conversant with the following topics:

- The basic purpose of a compiler and linker
- Number system theory
- Bit operations such as AND, inclusive-OR, exclusive-OR, complement, and left- and right-shift
- Data representation
- Communication between procedures by passing arguments and/or by returning a value, as well as via global variables
- Use of single- and multi-dimensional arrays
- Creation and use of sequential files and how to do formatted and unformatted I/O
- Basic data structures such as linked lists

Many of C++'s more powerful capabilities, particularly exception handling and inheritance, require advanced programming experience before they can be understood and exploited fully.

Although C++ is essentially a superset of C, you do *not* need to know C to use this book. If you do know C, simply read quickly over those sections that seem familiar to you. I say, "read quickly" rather than "skip" because you may well find that C++ defines things more fully or a little differently than does C. (There are also a few incompatibilities.)

If you know C, you have all the OO stuff to learn. If you know some OO language other than C++, you'll already be familiar with the OO concepts, but not the syntax that is largely common to C, C++, C#, and Java. And if your programming background is in some procedural language, you'll need to read the whole book closely and do all the exercises.

Limitations

This book covers almost all of the C++ language. It also introduces the core class library. However, only a very small percentage of library facilities are mentioned or covered in any detail. The Standard C++ library contains so many classes and functions that whole books have been written about that subject alone.

This book is directed at teaching the C++ language proper, by writing simple stand-alone applications, since without a thorough knowledge of the language, you will not be able to understand and implement other, more advanced solutions.

GUI, calling non-C++ routines, threading, internationalization (I18N), inter-process communications, operating system-specific, and a number of other advanced features are outside the scope of this text.

Presentation Style

The approach used in this book is *different* from that used in many other books and training courses. Having developed and delivered programming-language training courses for quite some years, I have found that the best approach for my students is an incremental one. I avoid introducing things that are unnecessary at any given time, thus making examples small, simple, and well-focused. **Specifically, I introduce the basic elements and constructs of the language using procedural programming examples.** Once those fundamentals have been mastered, I move on to object-oriented concepts and syntax. Then come the more advanced language topics and library usage. Many books on object-oriented languages use objects, inheritance, exception handling, GUI, and numerous non-trivial library facilities in the first few examples, and certainly in the first chapter. **I do not care for this approach**, either as a reader or as an educator. Instead, I prefer the student to have an excellent chance of understanding and absorbing small amounts of new material, and reinforcing it with lab sessions, as they progress. The intent here is to eliminate any chance of their being overwhelmed, provided, of course, they meet the prerequisites.

Different styles work for different teachers and different students. I do not suggest that my approach is better than any other; I simply know that my approach works well, and has formed the basis of my successful seminar business.

Exercises and Solutions

The programs shown in the text are available electronically in a directory tree named Source, where each chapter has its own subdirectory. For example, the source code for the program called ba04 can be found in the following fully qualified directory path: Source, Basics, ba04.cpp. By convention, the names of C++ source files end in `.cpp`.

Each chapter contains exercises, some of which have the character * following their number. For each exercise so marked, a solution is provided electronically in a directory tree named Labs, where each chapter has its own subdirectory.[1] For example, lab solution lbcl01.cpp has the following fully qualified name: Labs, Classes, lbcl01.cpp.

Exercises that are not so marked have no general solution and may require experimentation or research in an implementation's documentation. Numerous exercises contain a statement of the form "(See lab directory *xx*.)" This indicates the corresponding solution in the Labs subdirectory.

[1] The solutions are only available to licensees of these materials when they are used in formal training scenarios.

You are strongly encouraged to solve all exercises in one section before continuing to the next. Also, invent your own exercises as you go and be inquisitive; don't be afraid to experiment. Try to understand why the compiler gives you each error or why a program fails at run time.

Program Behavior

According to the C++ Standard, for correct, well-formed programs, almost all behaviors are well defined and predictable. However, in certain cases, an implementation can choose the behavior it deems best, and this behavior need not be the same as that exhibited by other implementations. Such behavior is called *implementation-defined*, and must be documented by each implementation. For example:

Implementation-Defined Behavior: The range of values that can be stored ….

In other cases, an implementation can choose whatever behavior it wants at that time *without* needing to reproduce or document that choice. Such behavior is called *unspecified*. For example:

Unspecified Behavior: The order of evaluation of the function designator and ….

A third category is *undefined behavior*, which can result from situations for which the standard imposes no requirements or is otherwise silent. For example:

Undefined Behavior: Subscripting a string with an out-of-bounds index.

Clearly, one must be aware of implementation-defined and unspecified behaviors when writing code that is to be ported across different platforms. And one should always avoid relying on undefined behavior.

The Status of C++

The history of the standardization of C++ is as follows:

- C++98 – The first edition of the ISO C++ standard, ISO/IEC 14882:1998, was produced by committee ISO/IEC JTC 1/SC 22/WG 21 in conjunction with US committee X3J16.
- C++03 – The second edition of the ISO C++ standard, ISO/IEC 14882:2003
- C++11 – The third edition of the ISO C++ standard, ISO/IEC 14882:2011
- C++14 – The fourth edition of the ISO C++ standard, ISO/IEC 14882:2014
- C++17 – The fifth edition of the ISO C++ standard, ISO/IEC 14882:2017

Committee, ISO/IEC JTC 1/SC 22/WG 21 continues to work on maintenance issues, Technical Reports (TRs), as well as a revision.

Electronic copies of the latest C++ standard can be purchased from www.ansi.org or www.iso.ch.

Acknowledgments

Many thanks to those people who reviewed all or part of this book. In particular, students in my C++ seminars provided useful feedback and located numerous typographical errors.

Rex Jaeschke, July 2020

1. The Basics

In this chapter, we will learn about a number of fundamental constructs and language elements.

1.1 Basic Program Structure

A correct source program is made up of an acceptable combination of tokens, where a *token* is the smallest possible unit of source. C++ has five different kinds of tokens: keywords, identifiers, literals, operators, and punctuators.

For the most part, C++ is a free-format language, with space between tokens being optional. However, in some cases, some kind of separator is needed between tokens so they can be recognized the way they were intended. White space performs this function. *White space* consists of one or more consecutive characters from the set: space, horizontal tab, vertical tab, form-feed, and *new-line* (entered by pressing the RETURN or ENTER key). In a source file, an arbitrary amount of white space is permitted before the first token, between any two adjacent tokens, and after the last token.

Let's look at the basic structure of a C++ program that writes a welcome message to the console (see directory ba01):

```
/*--
This program is a simple example of C++.
--*/

/*1*/   #include <iostream>

int main()               /* start of the program */
{                        // beginning of function body

/*2*/   std::cout << "Welcome to C++\n";

/*3*/   return 0;        // terminate with success
}                        // end of a function body
```

When run, the output produced is, as follows:

```
Welcome to C++
```

A *delimited comment* is one surrounded by `/*` and `*/`, and can be used on part of a source line, as a whole source line, or it can span any number of source lines. A *line-oriented comment* begins with `//` and continues until the end of its source line. A comment of either kind is treated as a single space. A delimited comment can occur anywhere white space can be used. A line-oriented comment can only appear at the end of a source line. Although comments of the same kind do not nest, each kind of comment can be used to disable a source line containing the other.[1]

[1] Refer to §19.4 to see a novel way to "comment out" a source line containing a comment.

Programming in C++

A C++ program consists of one or more *functions* that can be defined in any order in one or more source files (or *translation units*, as Standard C++ calls them). A program must contain at least one function, called `main`, and this function's name must be spelled using lowercase letters.[1] This specially named function indicates where the program is to begin execution.[2]

The parentheses following the function name `main` surround the function's *parameter list*.[3] The parentheses are required, even if no arguments are expected.

The body of a function is enclosed within a matching pair of braces. All executable code must reside within the body of some function or other. Statements are executed in sequential order unless branching or looping statements dictate otherwise. A program terminates when it returns from `main`, either by dropping into the closing brace of that function—which acts as an implicit `return` statement—or via an explicit `return` statement. According to the C++ standard, the explicit use of `return 0` in case 3[4] above is no longer necessary; however, at the time of writing, many compilers continue to issue a warning if this statement is omitted.

We'll learn about the `return` statement in §4.4. However, why return a value from `main`? Many programs either are invoked at the operating system's command-line level or are spawned by another program. In either case, when a program terminates, it has the ability to return one piece of information to the program that invoked it. Perhaps it returns a status code or a count of the number of transactions processed. We refer to this returned value as the program's *exit status code*. The value used, and its meaning are the business of the programmer. Unless otherwise stated, all `main` functions in this book contain the statement `return 0;`. The value zero has no special significance, although on some systems it is interpreted as some form of success code.

Case 1 provides access to the I/O library, and case 2 outputs the welcome text to the console. These cases are discussed in detail in §1.3.

Style Tip: Use liberal amounts of white space to improve program readability. The compiler discards all white space, so its presence has no effect on program execution. Apart from separating tokens, white space exists solely for the benefit of the reader. If you can't read the code, you surely won't be able to understand it.

We can write any given correct program in an infinite number of ways simply by using different amounts and kinds of white space (including comments). Obviously, there are very good, overt styles and very bad, cryptic styles. Then there is the large and subjective gray area in between.

Style Tip: Be overt; pick a style that isn't too far outside the mainstream, and stick with it. Above all, be consistent. And remember, the style you develop when learning a language is the one with which you will likely stay, so give some thought to programming style from the outset.

[1] While mixed- or uppercase spellings of `main` might be recognized by some systems, this is undefined behavior.
[2] Some environments require a different entry-point name. For example, certain Microsoft Windows programs begin execution at `winmain` instead.
[3] A function uses parameters to declare what it is expecting to be passed, via an *argument list*, at run time.
[4] Throughout this book, many code examples contain source lines with leading delimited comments of the form `/*`*n*`*/`, where *n* is a number. Such lines are referred to as *cases*, with `/*1*/` being case 1, `/*2*/` being case 2, and so on.

The style used in this book is very common. If you choose to adopt a different style, note that no matter how great you think your particular style is, every minute you spend arguing about whose style is superior and why—or worse yet, changing other people's code to your style—is generally time wasted. If no corporate-wide or group style guide exists, and the people on a multi-person project cannot agree on a common style, the project manager should dictate one.

Exercise 1-1*: Compile and link the empty program shown above, and look at the size of the resulting executable file on disk; it may be surprisingly large. Look at the listing produced by the linker and see what goes into a seemingly "empty" program. (See lab directory lbba03.)

Exercise 1-2*: Try compiling, linking, and executing a main program whose name is something like `Main` or `MAIN`. If either of these works, use `xyz` instead and try again. (See lab directory lbba04.)

Exercise 1-3*: Using `/*` and `*/`, comment out a block of code that already contains this kind of comment, and see how your compiler reacts. (Some compilers actually do support nested comments as an extension.) (See lab file lbba05.)

Exercise 1-4*: What happens if you leave off the closing `*/` from a comment? (See lab file lbba06.)

Let's move on to the more common case of a program having multiple functions (see directory ba02):

```
/* C++ program with two functions */

void compute()
{
        // …
/*1*/   return;         // redundant statement
}

int main()
{
/*2*/   compute();      // call function 'compute'

        return 0;
}
```

C++ supports modularization via functions. A source file can contain one or more functions, defined in any order. (That said, it is no accident that function `compute` is defined before function `main` in this example. In §4, we'll learn how to define functions in any order as well as how to put them in separate source files.) Unlike some languages, in C++, function definitions cannot be nested. That is, each function's definition must be outside the braces delimiting the definitions of all other functions. There is no syntactic difference between `main` and any other C++ function; they all have the same basic structure.

We call a function by using its name followed by a possibly empty argument list, as shown in case 2. The parentheses used in the call represent the function call operator.

Each statement must be terminated by a semicolon punctuator. When function `compute` terminates, control is returned to its caller. The explicit `return` statement in case 1 is redundant; we learned earlier that dropping into the closing brace of a function is an implicit `return`. (The presence of `void` indicates that `compute` does not return any value.)

Style Tip: Indent every line inside the body of a function definition by at least one tab. Also, line up the opening and closing braces, one above the other, but not indented with the block they delimit. This way, you can see the shape of the program at a glance by looking at the brace pairs. And if you are interested in the details, you can quite easily see where they are.

Exercise 1-5: See how your compiler reacts to having one function defined inside another. (This can easily happen if you forget the closing brace from the first function. And if you don't line up your brace pairs, you won't easily see which one is missing.)

Exercise 1-6*: What happens when you omit the semicolon from the end of a statement? Add an extra one and see what happens. (See lab directory lbba07.)

1.2 Identifiers

One kind of token is an *identifier*, a name usually invented by the application programmer. Identifiers are used to name variables and functions, among other things. Since keywords are reserved, they cannot be used as identifiers.

An identifier can be spelled using the following characters: Upper- and lowercase Latin letters (which are distinct), the decimal digits 0–9, and the underscore.[1] However, an identifier cannot begin with a digit, and identifiers that begin with an underscore followed by a capital letter are reserved for use by C++ implementers. You should also avoid spelling identifiers with two consecutive underscores, since such names are also reserved for use by implementations. (§4.2 and §19.6 show several ways in which an implementation might provide names having a reserved spelling format.)

Style Tip: To avoid possible conflict with "private" names used by your implementation, never invent an identifier that begins with an underscore or that contains two consecutive underscores.

An identifier can be arbitrarily long; all characters are significant.

While we can spell identifiers using letters in either case, or combinations of upper- and lowercase, the most common styles of naming variables and functions use either all lowercase or mostly lowercase with uppercase leading characters. Examples are `total`, `maximumNumber`, and `Week_Day`. Identifiers spelled in all uppercase are often used for other purposes, as we shall see in §18.

Keywords are only recognized as such if they are spelled entirely in lowercase.

Style Tip: It is very bad style to invent an identifier such as `If`, `Else`, or `FOR`, since such names are too easily confused by the reader (but never by the compiler) with keywords having the same names but spelled in lowercase.

[1] Some implementations also permit identifiers to contain $ or other characters, as an extension. In recent years, Standard C++ has been extended to allow the use of various non-Latin Unicode characters in tokens, via *universal-character-names*; however, that is outside the scope of this text.

Exercise 1-7*: Which of the following are valid identifiers: name, `_Xyz_`, `_`, `Total.count`, `Today'sdate`, `first-name`, `day_of_week`, `X32BITS`, `TOTAL$COST`, `3WiseMen`, and `LastNameOfMyFathersGrandfatherInLaw`? (See lab directory lbba02.)

1.3 Introduction to Formatted I/O

Unlike most older languages, *C++ has no input or output statements, per se*; instead, I/O is provided by the standard library in the guise of input and output stream objects, as we shall see. Let's examine the following program (see directory ba03a) in detail:

```
/* Fahrenheit-to-Celsius temperature conversion */

/*1*/ #include <iostream>

int main()
{
/*2*/   std::cout << "Enter a temperature";
/*3*/   std::cout << " in degrees"
                     " Fahrenheit: ";

        double fahrValue;
/*4*/   std::cin >> fahrValue;

        double celsValue;
        celsValue = (fahrValue - 32) * 5 / 9;

/*5*/   std::cout << fahrValue << " degrees Fahrenheit is "
                  << celsValue << " degrees Celsius\n";

        return 0;
}
```

Here are some examples of input and their corresponding output:

```
Enter a temperature in degrees Fahrenheit: 212
212 degrees Fahrenheit is 100 degrees Celsius

Enter a temperature in degrees Fahrenheit: 32.0
32 degrees Fahrenheit is 0 degrees Celsius

Enter a temperature in degrees Fahrenheit: 21.5
21.5 degrees Fahrenheit is -5.83333 degrees Celsius
```

Before we can use any I/O constructs, we must tell the compiler to get information about how the library performs I/O. We do this in case 1 by using `#include <iostream>`.[1] This is a directive aimed at the

[1] Prior to the existence of the C++ standard, the name of this header was `iostream.h`, so if your compiler cannot find this header by its new name, try the old one instead. If your implementation provides both versions, use the one without the `.h` suffix.

C++ preprocessor, a program that processes the source before passing it off to the compiler proper.[1] In this directive, the standard *header* called `iostream` is included. (Headers should be included at the beginning of a source file, prior to any function definitions.)

We can think of a header as a source file that contains shareable information. In this case, the header `iostream` contains declarations about the standard I/O library machinery. The compiler uses this information to check that we are using the I/O machinery correctly, and to generate calls to the appropriate underlying library functions. Note that we *must* include `iostream` in *every* source file in which we wish to perform I/O, since the information in this header is needed at compile time.

The library defines an object called `cout`, which is capable of writing formatted output to the *standard output* device. By default, `cout` is directed to the user's terminal. Older languages reserve certain—often many—names for use by their libraries. However, this can be problematic when adding a new name to an old library; we simply don't know how much existing code we might break because it is currently using that new name for its own purposes. For this reason, newer languages support *namespaces*; that is, they allow names to be grouped into sets, each of whose name must be unique. Namespaces are discussed in detail in §17; however, suffice it to say for now that all names in the Standard C++ library belong to a namespace called `std`.[2]

Another output object `cerr` (which is not used in this example), is much like `cout`. However, the latter is buffered while the former is not. A buffered version of standard error, called `clog`, is also provided. The difference between the two is quite important; if a program terminates abnormally, buffered output streams might not be flushed.

In case 2, we send the contents of a string literal to standard output by using the *insertion* operator `<<`. And since the name `cout` belongs to the namespace `std`, we disambiguate using that prefix, followed by `::` Likewise, in case 3 we send another string to standard output. While we can break a long string literal into multiple parts and output each separately, we can simply break up the literal into several smaller, but adjacent, literals, and have the compiler concatenate them for us. In such cases, any intervening white space is ignored.

Case 5 demonstrates that we can output the values of multiple expressions at once, in the specified order left-to-right. And the types of each expression can vary. However, each expression must be preceded by its own insertion operator, as shown.

Reading from standard input via `cin`, and the *extraction* operator `>>` in case 4 is just as simple as writing to standard output. And like standard output, we can read in multiple values at once, provided the destination of each input is preceded by its own extraction operator. If `cin` is used to read in multiple inputs, each such input must be separated by some form of white space.

We'll learn about the type `double` in §1.4.1; for now, suffice it to say that `fahrValue` and `celsValue` are capable of storing fractional values of a value range adequate for this example.

On output, the characters enclosed in double quotes are printed verbatim, except for certain *escape sequences*, which begin with a backslash. `\n` is C++'s notation for a *new-line*, the character that moves the print position to the first character on the next output line.[3] A new-line is *not* automatically appended by `cout`, so `cout` can be

[1] Preprocessor directives are discussed in detail in §18.
[2] Namespaces are an invention of the C++ standard.
[3] While a new-line might actually generate more than one character (such as a carriage-return/line-feed pair) on output on some systems, within a C++ program it is considered a single character.

invoked multiple times to print an output line a piece at a time, as shown in cases 2 and 3. `cout` can also output multiple lines at a time. If we want double spacing, we must use two consecutive new-line escape sequences.

Inside string literals, the backslash is used as an escape character prefix. It indicates that the following one or more characters are to be interpreted with other than their literal meaning. Each escape sequence represents a single character.

Table 1–1: Escape Sequences

Sequence	Meaning
`\a`	Alert
`\b`	Backspace
`\f`	Form feed
`\n`	Line-feed
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote
`\?`	Question mark
`\`*ooo*	Octal bit pattern *ooo*
`\x`*h..h*	Hexadecimal bit pattern *h..h*

Undefined Behavior: If any escape sequence other than those shown in the table is encountered.[1]

For the most part, the escape sequence mechanism allows character-set independence for these special characters. Most are self-explanatory; however, a few need further explanation. Alert sounds the terminal bell, if one exists. Carriage return is a holdover from the early days. It is unlikely it will ever be needed; a new-line will always suffice as a line terminator. Sometimes it is necessary to escape a single quote, as we shall see in §1.5. To enclose a double quote in a string literal, we must use `\"`. The question mark sequence is primarily of interest to

[1] Some implementations define extra escape sequences as extensions.

programmers in countries (such as those in Scandinavia) where terminals using the ISO-646 character set are prevalent.

The octal and hexadecimal bit pattern sequences allow special commands to be sent to output devices. For example, to clear the screen of a terminal that understands ANSI hardware escape sequences, we print the character sequence `\33[2J`, where `\33` represents the ASCII ESCAPE character. Likewise, we can use escape sequences to set a dot-matrix printer to bold, italic, or underline mode, for example.

Variables must be declared explicitly before their first use. When an unknown variable name is used, there is no implicit creation or typing; it is diagnosed as an error. Note that, like statements, each definition must be terminated by a semicolon. And as we can see in the example above, declarations can be interspersed with statements, allowing variables to be declared at the point of their first use.

The terms *definition* and *declaration* are quite often used interchangeably, even though there is a subtle difference. Therefore, we should be careful to use them correctly. A declaration of an identifier declares the attributes of that identifier. In the case of `fahrValue`, it declares that `fahrValue` is an object of type `double`. A definition not only declares an identifier, it also allocates memory for it. Every definition is also a declaration; however, not every declaration is a definition, as we shall see in §5, §7, and §11.

We assign values to variables using the = assignment operator. The value assigned can be a literal, the value of another variable, the result returned from a function, or the value of any expression having *compatible type*.[1] If necessary, the type of the result of the right-hand expression is converted to match that of the left-hand expression.

While the use of namespaces can be important in non-trivial projects, the fact is that many programmers consider the explicit qualification of names like `cin` and `cout` to be distracting. We can eliminate the need for this qualification by bringing all the names from a named namespace such as `std`, into the global namespace; we do this via the `using` directive. Here are the relevant parts of ba03b, which uses this approach:

```
#include <iostream>
using namespace std;

int main()
{
        cout << "Enter a temperature";
        cout << " in degrees"
                " Fahrenheit: ";
// …
        cin >> fahrValue;
// …
        cout << fahrValue << " degrees Fahrenheit is "
             << celsValue << " degrees Celsius\n";
// …
}
```

[1] All arithmetic types are compatible with one another.

1.4 The Data Types

1.4.1 Arithmetic Types

The set of integer types comprises signed and unsigned versions of `char`, `short int`, `int`, `long int`, and `long long int`.[1] In the case of `short int`, `long int`, and `long long int`, the keyword `int` is optional. The set of floating-point types comprises `float`, `double`, and `long double`, all of which are signed.

Implementation-Defined Behavior: The range of values that can be stored in an object of a given arithmetic type

However, Standard C++ specifies a minimum range (and in the case of floating-point types, a minimum precision) that must be supported.

An implementer of a C++ compiler determines how C++'s data types are mapped. And provided they meet the minimum range and precision requirements, different implementations (even those running on the same system) may use different mappings. The following table shows mappings commonly used for machines with word sizes as shown:

Table 1–2: Common Arithmetic Type Mappings

Type/Word Size	8-bit	16-bit	32-bit	64-bit
`[unsigned] char`	8 bits	8 bits	8 bits	8 bits
`[unsigned] short [int]`	16 bits	16 bits	16 bits	32 bits
`[unsigned] [int]`	16 bits	16 bits	32 bits	64 bits
`[unsigned] long [int]`	32 bits	32 bits	32 bits	64 bits
`[unsigned] long long [int]`C++14	64 bits	64 bits	64 bits	64 bits
`float`	32 bits	32 bits	32 bits	32 bits
`double`	64 bits	64 bits	64 bits	64 bits
`long double`	64 bits	64/80 bits	64/80 bits	64/128 bits

In the case of `long double`, entries of the form x/y mean "x or y". Remember that these are *possible* mappings, *not* required mappings.

Standard C++ imposes the following minimal requirements on arithmetic types:

- An object of the integer type `char` must be able to represent every character in the base character set. It must contain at least 8 bits.

[1] C++14 added the type `long long int`.

- An object of the integer type `short` must have at least 16 bits. Its size must be greater than or equal to that of a `char`.
- An object of the integer type `int` must have at least 16 bits. Its size must be greater than or equal to that of a `short`.
- An object of the integer type `long` must have at least 32 bits. Its size must be greater than or equal to that of an `int`.
- An object of the integer type `long long` must have at least 64 bits. Its size must be greater than or equal to that of an `int`.
- An object of the floating type `float` must be able to represent a value with at least 6 significant digits and with an exponent having a range of at least ±37.
- An object of the floating type `double` must be able to represent a value with at least 10 significant digits and with an exponent having a range of at least ±37. An object of type `double` must be able to store a value with at least the same range and precision as an object of type `float`.
- An object of the floating type `long double` must be able to represent a value with at least 10 significant digits and with an exponent having a range of at least ±37. An object of type `long double` must be able to store a value with at least the same range and precision as an object of type `double`.

A series of predefined *symbolic constants* is provided in the standard library headers `climits` and `cfloat`.[1] These constants allow us to find out various attributes of each arithmetic type. Obviously, the range and the precision of these types are important to programmers writing code that is to be portable across different machine architectures.

The type name `char` can be misleading. While a `char` variable can, and usually does, contain a character, characters are really represented in computers as small integers. Therefore, we can use a `char` variable to store integers directly. For example:

```
char c = 10;    // using a char as a small integer
```

The integer types can be made signed or unsigned by prefixing them with the keyword `signed` or `unsigned`, respectively. Signed and unsigned versions of the same type have identical sizes; however, in the case of `signed`, one bit is interpreted as a sign bit.[2] By default, `short`, `int`, `long`, and `long long` are signed, in which case, using `signed` with them is redundant.

Implementation-Defined Behavior: Whether a *plain* `char` (that is, one without an explicit `signed` or `unsigned` keyword) is signed.

We can manipulate the bits in an integer value via a series of operators; for example, we can shift bits left or right using the binary operators <<, <<=, >>, and >>=, we can mask them using the binary operators &, &=, |, |=, ^, and ^=, and we can complement them using the unary operator ~. (See §4.6 for more information.)

[1] Examples of the use of some of these constants are shown in §18.

[2] Standard C++ permits both ones- and twos-complement as well as signed-magnitude representation of integers.

Program ba04 provides some more examples of output formatting:

```
/* demonstrate some output manipulators */

#include <iostream>
/*1*/ #include <iomanip>
using namespace std;

int main()
{
/*2*/   int i = 10, j = 200;    // Use an initializer

/*3*/   cout << i << ", " << j << "\n";
/*4*/   cout << hex << i << ", " << uppercase << i << ", " <<
                nouppercase << i << "\n";
/*5*/   cout << oct << i << "\n";
/*6*/   cout << dec << setw(4) << i << "\n";
/*7*/   cout << setw(4) << setfill('*') << i << endl;

        double d = .125556;
/*8*/   cout << setw(6) << setprecision(3) << d << endl;

/*9*/   cout << 1.23 << ", " << 1.2 << ", " << 1. << ", " << 1.2e4 << '\n';
/*10*/  cout << fixed << 1.23 << ", " << 1.2 << ", " << 1. << ", " << 1.2e4 <<
'\n';

        return 0;
}
```

The output produced is:

```
10, 200
a, A, a
12
  10
**10
*0.126
1.23, 1.2, 1, 1.2e+04
1.230, 1.200, 1.000, 12000.000
```

In case 2, the definitions of `i` and `j` each include an *initializer*, eliminating the need for a separate assignment statement. And as we can see, we can declare multiple names in one declaration simply by separating them with commas.

Style Tip: By placing each variable in its own separate declaration, with one declaration per source line, you leave room for a trailing comment. You can also cut and paste lines more easily in a full-screen text editor.

The output stream designated by `cout` starts out in decimal mode, so when we display the values of the integers `i` and `j` in case 3, they are output as decimal values. We can change this by using one of the *manipulators* hex, `oct`, or `dec` to force integer output to be in hexadecimal, octal, or decimal form, respectively, as shown in cases 4, 5, and 6. As we can see, by default, hexadecimal digits are displayed in lowercase; however, we can control this using the `uppercase` and `noppercase` manipulators. We can set the display width of an output field using the manipulator `setw`, as in cases 6 and 7. By default, the corresponding numeric output is right justified and space-filled. If it needs more display positions it uses them; the specified width is only a minimum. We can specify a fill character other than the default space character using the manipulator `setfill`, as shown in case 7. In case 8, we use the manipulator `setprecision` to specify the number of digits to appear after the decimal point when displaying a floating-point value.

The `setprecision` manipulator in case 8 works well for `d`, because that variable has three or more decimal places. However, as we can see in case 9, when we display values having fewer decimal places, the `setprecision` manipulator is not used. We can rectify this by using the `fixed` manipulator, as shown in case 10. While `fixed` forces a fixed number of decimal places, the manipulator `scientfic` (which is not used in this example) forces scientific—that is, exponent form—notation.

Manipulators without arguments are accessed via `iostream` while those with arguments come from `iomanip`, which we included in case 1. All manipulators except `setw` permanently change the state of the output stream. In the case of `setw`, the specified width stays in force for the next output field only, after which the width is set back to zero, indicating no justification or padding.

Each line of output is terminated with a single new-line character; however, case 8 achieves this a little differently from the others. The manipulator `endl` also causes the output stream to be flushed; that is, it forces the output to be written. This can also be achieved by using the manipulator `flush` directly.

Note that an output statement need not actually cause any output; it could simply contain a manipulator. For example:

```
cout << setfill('*');
```

Other manipulators are provided. For more details, see §10.

Exercise 1-8: Find out the size and precision of all the arithmetic types for your implementation. (Hint: look for information in the standard headers `climits` and `cfloat`.) Is `long double` equivalent to `double`? Are `short`, `int`, `long`, and `long long` all different, or is there some overlap? Is a plain `char` signed or unsigned? Some compilers provide an option to select the signedness of a plain `char`; does yours?

Exercise 1-9*: Write a program that has three `int` variables named `cost`, `markup`, and `quantity`, and initialize them to \$20, \$2, and 123, respectively. Print the result of *retail cost* × *quantity*, where *retail cost* equals *cost* + *markup*. (See lab directory lbba01.)

1.4.2 The Boolean Type

An object of type `bool` can take on one of two possible values: `true` or `false`. For example:

```
bool b = true;
b = false;
```

The standard does not require a `bool` value to have a specific size nor does it require specific values for `false` and `true`. Although 0 and 1, respectively, might be the obvious choices for these values, other values can also be made to work. The only requirement is that when these values are converted to arithmetic types, `false` produces 0 and `true` produces 1; and these operations are permitted to generate code to achieve this.

While `bool` isn't really an integer type, `bool` values behave a lot like integers, and they participate in arithmetic promotions. One such example of this is with output, where, by default, they are treated as integers (see directory ba05):

```
bool b = true;
cout << "b (true)  = " << b << '\n';
b = false;
cout << "b (false) = " << b << '\n';
```

The output produced is:

```
b (true)  = 1
b (false) = 0
```

By default, `false` is written out as 0 and `true` as 1. However, we can change this by using the manipulator `boolalpha`, as follows:

```
cout << "false = " << boolalpha << false << '\n';
cout << "true  = " << true << noboolalpha << '\n';
```

Now the output produced is:

```
false = false
true  = true
```

We reset back to numeric output mode by using the companion manipulator `noboolalpha`.

Like the standard output stream, the standard input stream also deals with `bool` input as integer by default. And as we might expect, we can use these manipulators to change that. For example:

```
cout << "Enter a bool: ";
cin >> boolalpha >> b;
cout << "Value entered = " << b << '\n';
cout << "Enter another bool: " ;
cin >> noboolalpha >> b;
cout << "Value entered = " << b << '\n';
```

Some input and the corresponding output are:

```
Enter a bool: true
Value entered = 1
Enter another bool: 34
Value entered = 1
```

In `boolalpha` mode, only the lowercase strings "false" and "true" are recognized. As we can see, an input of "true" resulted in an output of 1. Since `boolalpha` mode is disabled before the second input is read in, only integer input is recognized, with 0 meaning false and any nonzero value (positive or negative) meaning true.

1.4.3 Enumerated Types

An *enumeration* is made up of a set of named constant values each of which is called an enumerator. Each distinct enumeration constitutes a different enumerated type. The use of enumerations and their enumerators provides a useful amount of abstraction. Let us look at the following (see directory ba06) for some examples:

```
/*1a*/  enum carColor {black, white};
/*1b*/  carColor c1 = black;

/*2a*/  enum houseColor {red, green = 12, blue, scarlet = red};
/*2b*/  houseColor c2 = green;

/*3*/   c1 = 0;          // error, even though black == 0
/*4*/   c1 = white;      // okay
/*5*/   c1 = blue;       // error, incompatible types
/*6*/   c2 = white;      // error, incompatible types
/*7*/   c2 = blue;       // okay
/*8*/   c1 = c2;         // error, incompatible types
```

In case 1a, we define a named enumeration called `carColor` with two enumerators, `black` and `white`. (Enumerators are identifiers.) By default, the internal value of an enumerator is one more than its predecessor, with the first having value 0.[1] In case 1b, we also define `c1` to be a variable of this type, and give it an initial value of `black`. Similarly, in case 2a, we define an enumeration, `houseColor`, giving it four enumerators; however, `green` is given the value 12, resulting in `blue`'s being 13, while `scarlet` has the same value as `red`. Clearly then, the range of values used by the members in an enumeration can involve gaps and/or duplicates.

Since operations involving enumerations and enumerators are strongly checked with respect to type compatibility, situations such as those occurring in cases 3, 5, 6, and 8, are rejected. One complication of this checking, however, is that different enumerations cannot use the same identifier as an enumerator, even if it is given the same value.[2]

A named enumeration permits compile-time checking to be performed when a variable is permitted to take on any one of an enumerated set of values only. Since this is a common situation in programming, we should use enumerations whenever we can.

Tip: Avoid initializing enumerators explicitly or making duplicates unless you have good reason to do so. Assigning a range of values that is non-continuous or that contains duplicates significantly reduces the set of operations that can meaningfully be performed.

[1] This makes them useful as array subscripts, which, as we will learn in §3, begin at zero.
[2] This is not entirely true; such use is permitted *provided* each enumeration is defined in a different namespace.

Let us consider another example (see directory ba07):

```
/*1*/   enum {veteran, nonVeteran};

/*2*/   cout << "veteran    = " << veteran << "\n";
/*3*/   cout << "nonVeteran = " << nonVeteran << "\n";

/*4*/   int veteranStatus = veteran;
        cout << "veteranStatus = " << veteranStatus << "\n";
```

The output produced is as follows:

```
veteran    = 0
nonVeteran = 1
veteranStatus = 0
```

In case 1, we define an unnamed enumeration having two enumerators; however, we can *never* define variables of this (unnamed) enumeration type. However, as we see in case 4, we can use enumerators in any context permitting an integral constant value. While enumeration types are distinct from integral types, conversion between the two is permitted in the useful cases.

Cases 2 and 3 simply demonstrate that the values of `veteran` and `nonVeteran` are 0 and 1, respectively. The I/O library machinery simply converts them from their unnamed enumeration type to some integer type, as necessary, just as in case 4. (See §13.3 for an example that displays their values as mnemonics.)

In order to share an enumeration between separately compiled source files, we simply create a user-defined header. Let's call this header `Constants.h`, and put it in a source file of the same name:

```
/* source file Constants.h */

enum VeteranStatus {veteran, nonVeteran};
enum CursorDirection {up, down, left, right};
```

To use this header, we simply make it available during compilation much like we do for `iostream`, except that we use "…" delimiters rather than <…>, as follows:

```
#include "Constants.h"

void process()
{
        VeteranStatus code = veteran;
        CursorDirection dir = left;
        // …
}
```

1.4.4 User-Defined Types

We can define an unlimited number of user-defined object types, such as Employee, Publication, Transaction, Message, and Point. We will learn how to do this in §7, and we'll add object-oriented machinery as well in §11.

1.4.5 The string Type

It is common to need variables that can hold strings. The simplest way to do this in C++ is to use the type `string`. However, note that this is *not* a built-in type, but, rather, a user-defined type that is required to be part of the Standard C++ library; hence the `include` directive in case 1 of the following example (see ba08):

```
#include <iostream>
/*1*/   #include <string>
using namespace std;

int main()
{
/*2*/   string firstName, lastName, greeting;

        cout << "Enter your first and last names: ";
/*3*/   cin >> firstName >> lastName;

/*4*/   greeting = "Hello " + firstName + " " + lastName;
/*5*/   cout << greeting << ".\n";

/*6*/   string day = "Monday";
/*7*/   cout << "Character 0 of day is " << day[0] << '\n';
/*8*/   cout << "Character 4 of day is " << day[4] << '\n';
/*9*/   day[3] = '?';
        cout << "day is " << day << '\n';

        return 0;
}
```

An example of some input and the corresponding output, follows:

```
Enter your first and last names: Mary Smith
Hello Mary Smith.
Character 0 of day is M
Character 4 of day is a
day is Mon?ay
```

As we can see, we can define variables of type `string` (see case 2), we can perform I/O on them (cases 3 and 5, respectively), we can concatenate them with each other and with string literals, we can assign them (case 4), we can access individual characters within them using the subscript operator (cases 7 and 8), and characters within them can be modified (see case 9). (Note that character positions in a string begin at zero.)

Type `string` supports many other operations, including substring extraction and replacement, comparison, and searching. This type is described in detail in §9.

Undefined Behavior: Subscripting a `string` with an out-of-bounds index.

1.5 Literals

Arithmetic literals can have integer or floating-point type, depending on the presence or absence of a decimal point, exponent, and type suffix.

C++ does *not* support negative literals. Expressions having the form `-`*literal* involve the unary minus operator and a non-negative literal.

1.5.1 Integer Literals

An integer literal has a base (or radix), as determined by the presence of a prefix. If an integer literal has a prefix of 0x or 0X, it is interpreted as a hexadecimal (base 16) number and is permitted to contain the characters a–f and A–F, as well as the digits 0–9. If the prefix is just 0 (zero), the number is interpreted as octal (base 8) and only the digits 0–7 are permitted. Starting with C++14, if an integer literal has a prefix of 0b or 0B, it is interpreted as a binary (base 2) number and is permitted to contain the digits 0 and 1. All other integer literals are deemed to be decimal (base 10) and can contain only the digits 0–9.

Starting with C++14, an integer literal can contain single quotes to allow visual grouping. For example, 123456789 can be written as 123'456'789, where the quotes are intended as thousands separators. Sometimes it is convenient to think of hexadecimal and binary numbers as groups of 8-bit bytes, as in 0x1F'0A'34'6F and 0b10101010'10000111'11100000'00000111.

Since a literal is an expression, and each expression has a type, a literal has a type as well as a value.

Implementation-Defined Behavior: The type of a given integer literal, since it depends on the mapping of the integer types by the compiler.

The following table shows the steps used by a compiler to find the type of an integer literal:

Table 1–3: Integer Literal Typing Steps

Type/Base	Decimal	Binary/Octal/Hex
`int`	1	1
`unsigned int`	—	2
`long int`	2	3
`unsigned long int`	—	4
`long long int`	3	5
`unsigned long long int`	—	6

If the value of a decimal literal can be represented as an `int`, that is its type. If it cannot, and it fits in a `long int`, that is its type. If it cannot, and it fits in a `long long int`, that is its type.

Undefined Behavior: If the value of a decimal literal cannot be represented as `long long int`.

For binary, octal, and hexadecimal literals, an extra unsigned step is added.[1]

An integer literal can explicitly be given the type `long int` by the addition of a suffix of L or `l`. An integer literal can explicitly be given the type `long long int` by the addition of a suffix of LL or `ll`. A U or u suffix makes an integer literal `unsigned`. This can be combined with `l`, L, `ll`, or LL.

Style Tip: When explicitly typing integer literals with a `long` suffix, use L instead of l since the latter can easily be mistaken for the digit 1.

The following are examples of integer literals:

```
123456          /* decimal, int or possibly long int */
01234           /* octal, int                        */
0x09aB          /* hex, int                          */
0XfFc5          /* hex, int or possibly unsigned int */
5L              /* decimal, long int                 */
0xAb2L          /* hex, long int                     */
065U            /* octal, unsigned int               */
0x234uL         /* hex, unsigned long int            */
1234567890LL    /* decimal, long long int            */
```

1.5.2 Character Literals

Since characters really are represented as integers, variables of type `char` can be initialized with integer expressions. However, initializing a `char` variable with the value 65 on a system using the ASCII character set requires that the reader know that the internal value of A is 65. It also makes the code ASCII-specific. To allow for readability and portability, we can write such values in the form of *character literals* using the form `'`*x*`'`.[2] A character literal has type `char` and its value is that of *x* in the machine's character set.

The following are examples of character literals:

```
'A'  '+'  'ß'  'ñ'  'æ'  '\n'  '\''  '\xff'
```

As we can see, a character literal can contain any of the escape sequences or a printable character. The single quote sequence is needed only when writing a character literal containing a single quote.

Consider the following statements:

```
cout << i << "\n";
cout << i << '\n';
```

[1] This means that a decimal literal with a given value can have a different type than the same-valued literal expressed in binary, octal, or hex!

[2] The initial C++ Standard introduced character literals with a prefix of L. Then C++14 introduced character literals with a prefix of u or U, and C++17 introduced character literals with a prefix of u8. A discussion of these is outside the scope of this text.

Both result in the value of `i` being output followed by a new-line. Since the string literal contains only one character, we can just as easily write it as a character literal; it's simply a matter of personal preference.

1.5.3 Floating-Point Literals

A floating-point literal must contain a decimal point, an exponent, or both. The exponent can be written using either an upper- or lowercase E, and it can contain a leading sign. Both the value and exponent parts are interpreted as decimal.

C++17 added support for floating-point literals written in hexadecimal and having an exponent written using an upper- or lowercase P. It also added support for single-quote separators, as with integer literals.

An unsuffixed floating-point literal has type `double`. A suffix of F or f indicates type `float`, while a suffix of L or l indicates type `long double`. Unlike integer literals, the compiler does not use a table of steps to determine the type of a floating-point literal; the suffix, or absence thereof, says it all.

Style Tip: When typing a `long double` literal, use L instead of l since the latter can easily be mistaken for the digit 1.

The following are examples of floating-point literals:

```
.952            /* double      */
3e34            /* double      */
1.23E5          /* double      */
345.F           /* float       */
3.456e+4f       /* float       */
321e-6L         /* long double */
435.54l         /* long double */
```

1.5.4 Boolean Literals

There are two Boolean literals, the keywords `true` and `false`. Their type is `bool`.

1.5.5 String Literals

As we learned in §1.3, a string literal is a sequence of characters surrounded by double quotes, as in `"hello"`. The string literal `""` represents the empty string. A string literal has type "array of *n* `char`", where *n* is one more than the number of characters shown. (For example, `"hello"` is an array of 6 characters.)[1] **Note carefully that a string literal does not have type `string`.**

The initial C++ Standard introduced string literals with a prefix of L. Then C++14 introduced character literals with a prefix of u, U, u8, and/or R. A discussion of these is outside the scope of this text.

Adjacent string literals are concatenated by the compiler.

Implementation-Defined Behavior: Whether two string literals containing the same characters are stored in distinct arrays.

[1] We'll learn more about arrays of characters and how string literals are stored, in §3.3.

Although programmers tend to think of string literals as being constants, some implementations store them in read/write memory. Whether a string literal can be modified is undefined.

1.6 Type Qualifiers

C++ has two type qualifiers: `const` and `volatile`.[1] As the name implies, a type qualifier somehow qualifies (by restricting or controlling) the way in which an object of that type can be accessed. These qualifiers can be used together.

1.6.1 The const Qualifier

When the `const` qualifier is applied to an object, it prohibits that object from being modified; for example:

```
const int maximum = 100;
const double pi = 3.1415926;

maximum = 200;  // error
```

While a `const`-qualified object cannot be modified, it can be (indeed, it must be) initialized via an initializer, as shown in the definition of `maximum` and `pi` above.

It is important to understand that `const` does not guarantee that an object will be stored in read-only memory at run time. It might or might not be, at the implementation's pleasure.

The obvious application of `const`-qualified "variables" is in defining symbolic constants, those sensible names given to obscure physical constants. In order to share a set of symbolic constants, we simply create a user-defined header, just as we did in §1.4.3. Then if the need arises to change the initial value of a `const`-qualified object, we can simply change its initializer in the header and recompile all source files that include it.

As we shall see throughout this book, `const` can be used in a number of other useful contexts.

1.6.2 The volatile Qualifier

The `volatile` qualifier tells the compiler that the object so qualified might be accessed by a hardware device or by multiple threads of execution at the same time in an asynchronous manner. For example, the object might be the receive buffer of a serial port. Alternatively, the object might reside in shared memory and be used as a synchronizing indicator between cooperating programs.

Essentially, `volatile` is a directive to the compiler to disable certain optimizations in such a way that *every* time there is a logical access to a `volatile`-qualified object, the object is physically accessed as well.

Consider the following example (see directory ba09), which computes $y = 2x^2 + 5x + 6$ in two different ways:

```
volatile int x;

void f1()
{
        int y = (2 * x * x) + (5 * x) + 6;
}
```

[1] The type qualifier, `restrict`, invented by C99, is *not* supported by Standard C++.

```
void f2()
{
        int t = x;      // snapshot copy of x
        int y = (2 * t * t) + (5 * t) + 6;
}
```

The result computed in `f1` is unreliable, because the `volatile` qualifier on `x` requires the compiler to fetch `x`'s value three times, yet its value may have changed between fetches. By accessing `x` once and storing a copy of it in the non-`volatile` object `t`, `f2` produces a value consistent for any given value of `x`.[1]

In this example, `x` is defined outside of any function; we'll learn what this means in §5.8.

1.7 Type Synonyms

The `typedef` keyword provides the ability to create a synonym for another type; for example:

```
typedef unsigned int Counter;
```

Here, we have "invented" the type `Counter` by making it a synonym for the type `unsigned int`. To create a synonym for a type, write a declaration for an identifier of that type and prepend the keyword `typedef`; the identifier now becomes the new type name.

Once created, a type name can be used in any context in which the underlying type may occur; for example:

```
Counter total = 0;
```

Why invent a type synonym if we can always write the underlying "real" type? `typedef` allows us to invent an abstract type name. If we give it a descriptive name and promise certain properties, we can use that type very effectively without knowing exactly its underlying type. For example, we are writing code that is to be ported across a number of platforms and we need various counters. The value of a counter is always in the range 0–1000. Of course, counters are whole numbers. By inventing the type name `Counter`, we allow the underlying type to be changed when the code is ported. We might wish to map the type to a signed or unsigned `short`, `int`, `long`, or `long long` for size or speed reasons, depending on the target machine's architecture.

There are also occasions where we receive a value from some library function, store it, and later pass it back to another library function without ever using it directly. In such cases, we don't need to know what type that value has. By hiding the underlying type information, we allow the author of the library to change the actual type as long as he maps the type synonym to the new type. Type qualifiers can be included in the type underlying a typedef; for example:

```
typedef const unsigned int Constant;

Constant i = 10;
```

Since type synonyms can be shared between separately compiled source files, their definitions should be placed inside a user-defined header.

[1] This approach works only if `x` can be accessed in an atomic operation.

Tip: If you have to know the underlying type of a type synonym in order to use objects of that type effectively, the type synonym is useless.

Exercise 1-10: Standard C++ requires numerous type synonyms to be defined in one or more standard headers. Some of these types (and their corresponding headers) are as follows:

Table 1–4: Examples of Abstract Types Defined in Standard Headers

Type	Header
`clock_t`	`ctime`
`div_t`	`cstdlib`
`FILE`	`cstdio`
`fpos_t`	`cstdio`
`ptrdiff_t`	`cstddef`
`size_t`	`cstddef`, `cstdio`, `cstdlib`, `cstring`, among others

Read about these abstract data types in your library documentation.

1.8 References

C++ provides an aliasing mechanism, called a *reference*; for example (see directory ba10):

```
        int i;
/*1*/   int& r1 = i;    // alias for i

/*2*/   i = 5;          // modify i directly
        cout << "i = " << i << '\n';

/*3*/   r1 = 10;        // modify i via reference r1
        cout << "i = " << i << '\n';

/*4*/   int& r2 = i;    // another alias for i
/*5*/   int& r3 = r2;   // alias for r2, which is an alias for i
```

In case 1, we declare `r1` to be an alias for the `int` variable `i`. `r1` is not a variable in its own right, but simply an alias for `i`. By using the & notation, we define `r1` to be a reference to variable `i`. Any context that permits the name `i` to be used also permits `r1`, with exactly the same results. For example, in case 3, we modify the value of `i` through its alias.

When a reference such as `r1` is declared, it must have an initializer, and that reference remains aliased to the object identified in its initializer.

We can have more than one reference to the same object, as demonstrated by case 4. A reference to a reference to a reference, and so on, is the same as a reference to the original object itself. That is, although in case 5, we make `r3` an alias for `r2`, since `r2` is already an alias for `i`, `r3` must also be an alias for `i`.

We can create a read-only alias using `const`, as follows:

```
float f;
const float& rf = f;    // create a read-only alias
```

Since an alias behaves in every way like the object to which it is aliased, we cannot directly make a reference of one type be an alias to an object of another type. Consider the following:

```
int i;
long& rl1 = i;          // error, incompatible
const long& rl2 = i;    // okay, since copy made
```

The declaration of `rl1` is rejected, since an `int` and a `long int` are not the same thing. However, the declaration of `rl2` *is* accepted because of the `const` qualifier. In this case, a temporary object of type `long` is created by the compiler and its value is that of `i`. Then `rl2` is made an alias of that temporary and, since `rl2` is a read-only reference, the temporary object's value can never be changed through that reference.

At this stage, references don't seem to be particularly useful; however, once we learn how to pass arguments to functions in §4.3.2, we'll see a good application. Also, keep in mind that we can create an alias to an element of an array, or to some sub-object buried inside a larger, more complicated object.

1.9 Automatic Variables

With one exception, all of the variables used so far have been defined inside the braces delimiting the body of a function. Variables defined within a function are local to that function and are not accessible by name from within other functions. Ordinarily, they are created each time their parent function is invoked, and they disappear when that function terminates. They are known as *automatic variables* because they automatically come and go.[1]

Typically, the cost of allocating memory for automatic variables is independent of the amount being allocated; that is, it costs the same at run time to allocate one variable, as it will to allocate 100 or 1,000. **By default, the initial value of an automatic variable is undefined; whatever garbage happens to be lying around in the memory allocated becomes the initial value of the newly allocated variable.**[2]

Tip: A common source of error is the failure to initialize an automatic variable.

[1] In §5, we'll see how to define local (and other) variables that live for the whole life of a program.

[2] This undefinedness is true for variables having one of the built-in types. For most user-defined types, the default initial value is determined by that type's designer. For example, a variable of type `string` that is not explicitly initialized contains an empty string.

Consider the following program (see directory ba11):

```
/* define and initialize some automatic variables */

#include <iostream>
using namespace std;

int main()
{
/*1*/   signed int si = 100;    // signed is redundant
/*2*/   unsigned int ui = -20;  // negative to unsigned
/*3*/   int i = 1000000;        // will this number fit?
/*4*/   unsigned long ul;       // undefined initial value
/*5*/   double d = si + ui;     // mixed-mode arithmetic

/* By now, memory has been allocated for all the variables */
/*6*/   cout << "ui = " << ui << '\n';
/*7*/   cout << "i = " << i << '\n';
/*8*/   cout << "si + ui = " << si + ui << '\n';
/*9*/   cout << "d = " << d << '\n';

        return 0;
} /* automatic variables are destroyed here */
```

The output produced on one system (that used 16-bit, twos-complement arithmetic) system was:

```
ui = 65516
i = 16960
si + ui = 80
d = 80
```

When run on another system (that used 32-bit, twos-complement arithmetic), the output was:

```
ui = 4294967276
i = 1000000
si + ui = 80
d = 80
```

When run on another system (that used 64-bit, twos-complement arithmetic), the output was:

```
ui = 4294967276
i = 1000000
si + ui = 80
d = 80
```

For this implementation, a `long` had 32 bits rather than 64.

In cases 1 and 2, the variables are explicitly initialized. Of course, since they are created at run time, on entry to this function, they also are initialized at run time.

In case 2, an unsigned integer is being initialized with a negative value. While this is permitted, the value actually stored is some large unsigned number, as shown by the output produced in case 6.

Tip: Be careful when mixing unsigned integers with signed integers that have negative values.

In case 3, we are attempting to store a large number in a signed `int`. And since an object of this type is not required to have more than 16 bits, truncation can occur, as shown in the output produced by case 7 on the 16-bit system. Whether a compiler warns about this possibility is a *quality of implementation* issue. That is, it is a marketplace issue, not something mandated by the C++ standard.

In case 4, the keyword `int` is implied and the initial value of `ul` is undefined. In case 5, the initializer consists of an expression involving a computation. Such an expression can involve any terms whose value can be computed at run time. The only criterion is that any variables referenced in that expression must already have been defined and initialized.

The initializer in case 5 is worth further mention. Because this expression involves terms having different (but compatible) arithmetic types, type promotion is involved; the value of the `signed int` is promoted to `unsigned int` and the type of the result of the addition is `unsigned int`. However, this type cannot accurately store the result, resulting in "modulo wrap around".[1] The resulting value, 80, is then used to initialize the `double` variable `d`. The important aspect here is that even though `d` can represent the correct result of adding `si` and `ui`, the sum is truncated *before* being used to initialize `d`.

In the four cases involving explicit initialization, we could have achieved the same result by omitting the initializers and using equivalent assignment statements instead; it's a matter of style as to which approach you use.

It is important to note that the order in which we define variables has no effect on where they are stored in memory relative to one another. Their relative locations are determined by the compiler and are unspecified by Standard C++.

1.10 Operator Precedence

When an expression involves more than one operator, the compiler determines the order in which it groups terms, based on the *precedence table*.[2] Consider the following example (see directory ba12):

```
/* Operator precedence and associativity */

#include <iostream>
using namespace std;

int main()
{
        int i = 5;
        long int l = 500;
        float f = 3.58345F;
        double d;
```

[1] It is impossible to get overflow with unsigned integer arithmetic.
[2] This table is shown in the Annex
Operator Precedence along with a discussion of operator precedence and associativity.

```
/*1*/   d = i + l * f;              // d = i + (l * f)
        cout << "d = " << d << '\n';

/*2*/   d = (i + l) * f;
        cout << "d = " << d << '\n';

/*3*/   d = i + l + f;              // d = (i + l) + f
        cout << "d = " << d << '\n';

/*4*/   ((d) = (((i) + (l)) + (f)));
        cout << "d = " << d << '\n';

        return 0;
}
```

The output produced is:

```
d = 1796.73
d = 1809.64
d = 508.583
d = 508.583
```

In case 1, multiplication has higher precedence than addition, resulting in the value of `l` being multiplied by the value of `f`, and the result being added to the value of `i`.

The default precedence and associativity can be overridden via the use of grouping parentheses, as shown in case 2. Here, the values of `i` and `l` are added, and the result is multiplied by the value of `f`.

In case 3, the two addition operators have the same precedence, since they are the same operator. However, these operators have left to right *associativity*. That is, the left one is given higher precedence. Therefore, the values of `i` and `l` are added, and the result is then added to the value of `f`.

In case 4, all the grouping parentheses are redundant.

Style Tip: While the use of redundant grouping parentheses can sometimes help readability, excessive use can hinder it.

In some languages, the order in which terms of an expression are evaluated is tied to the grouping of terms as specified by operator precedence. This is *not* the case in C++. **Unless otherwise stated, the order in which individual terms in an expression are evaluated is unspecified.** Consider the following expression:

```
x = f() + g() * h();
```

The precedence is quite clear; multiplication wins out over addition, as demonstrated by the corresponding parse tree:

```
    +
   / \
  /   \
f()    *
      / \
     /   \
   g()   h()
```

However, in what order are the three terms themselves evaluated? That is, in what order are the three functions called? If the three functions behave differently when called in one order versus another, the order of their evaluation is important. The only way to make the order of evaluation predictable is to break the statement into multiple statements.[1] For example, the steps:

```
x = g();
x = x * h();
x = x + f();
```

result in the same precedence as before, but with the functions called in the guaranteed order `g()`, `h()`, and `f()`.

Tip: A common error is to rely on the order of evaluation of terms across a given operator when no such guarantee is given by the language.

See Annex A for the complete operator precedence table.

1.11 Type Conversion

Like most languages, C++ permits expressions to contain terms of different types, provided the types are compatible. Consider the following mixed-mode expression:

```
d = i + l + f;     // d = (i + l) + f
```

The precedence table determines how pairs of terms are grouped. Because of that grouping, the type of each subexpression is determined. In the example above, since `i` and `l` have different (but compatible) types, implicit conversion of the value of `i` to `long int` occurs, and the result of the addition has type `long int`. Then that value is converted to type `float` so it can be added to `f`, producing a result of type `float`. The value of this result is then converted to type `double`, so it can be assigned to `d`.

The order in which we write the terms in this assignment expression can be important. For example, the following are subtly different:

```
/*1*/   d = i + l + f;          // d = (i + l) + f
/*2*/   d = f + i + l;          // d = (f + i) + l
```

[1] For information on when the compiler is required to perform certain operations, see §20.4.

In case 2, the value of `i` is converted to type `float` and then added to `f`, producing a result of type `float`. Then, the value of `l` is converted to type `float` and added to the result of the previous addition. For many values of `i`, `l`, and `f`, the two versions will produce the same result. However, consider case 1 when `l` contains the largest value that can be stored in an object of type `long int`, and the value of `i` is 1 or more. When they are added, overflow occurs, since the result cannot be represented in the type `long int`. If, however, the expression is rearranged as in case 2, all intermediate results are performed in type `float`, so no integer overflow can occur.[1]

Controlling the type of intermediate results by rearranging the expressions is poor style. Instead, it is better to use explicit type conversion. This is done via the *cast* operator, which has the following equivalent forms:

type-name (*operand*)

(*type-name*)*operand*

As we can see, this operator can be written in two ways: as a type name followed by its operand inside parentheses in a function call-like manner or, as a type name inside parentheses followed by its operand. For example, in the case of `int(x)` and `(int)x`, an unnamed object of type `int` is created, and its value is that of `x` converted to type `int`. The type and value of `x` are unchanged. We can use a cast in our previous example to avoid the possibility of integer overflow, as follows:

```
/*1*/   d = float(i) + l + f;
/*2*/   d = i + float(l) + f;
/*3*/   d = float(i) + float(l) + f;
```

In case 1, the value of `i` is explicitly converted to type `float`, resulting in the value of `l`'s being implicitly converted to type `float` as well. In case 2, the value of `l` is explicitly converted to type `float`, resulting in the value of `i`'s also being implicitly converted to type `float`. In case 3, both `i` and `l` are explicitly cast. All three cases are equivalent.

The value of an expression having any arithmetic type can be cast explicitly to any arithmetic type; however, it may result in a loss of precision, when a `double` is cast to a `float` or an `int`, for example. An expression can be cast to its own type, although such a cast has no effect.

Any arithmetic value, enumeration, or enumerator can be converted to type `bool` with values of zero producing false and all other values producing true. When a value of type `bool` is converted to an arithmetic type or enumeration, `true` results in 1 while `false` results in 0.

Style Tip: Avoid unnecessary casts because they detract from readability.

The cast operator can be used for a number of quite different kinds of conversions, and cast operators are hard to find when reading source as well as when using text search tools. In order to make casting more overt, Standard C++ introduced a number of alternate cast operators. While they provide no new functionality, they certainly make the conversion more obvious. These new operators (which are all keywords) are: `const_cast` (see §6.7), `dynamic_cast` (§20.8), `reinterpret_cast` (see §6.5), and `static_cast` (see §6.12).

[1] In mathematics, addition is both associative and commutative, and overflow cannot occur. However, when numbers are represented in a machine, a finite range is used, so overflow becomes a possibility.

The syntax of one of the new alternate forms of the cast operator is as follows:

static_cast< *type-name* > (*operand*)

which allows

```
d = float(i) + l + f;
```

to be written instead as

```
d = static_cast<float>(i) + l + f;
```

11. Classes and Objects

Classes are the main concept underlying the object-oriented nature of C++. Simply stated, a *class* is a special kind of structure.

In this chapter, we will learn about data hiding and encapsulation. *Data hiding* is the process whereby implementation details of a class can be hidden from general access. *Encapsulation* involves the syntactic association of data and the functions that have permission to operate on it.

11.1 Introduction

Given what we learned about structures in §7, one problem is that any code that can access a structure object as a whole can also access all its members. The problem then, becomes one of a lack of discipline. If we can get at any member of a structure in scope, we do so, generally in a rather haphazard way. That is, the way in which we interface with such objects is not controlled, making debugging and maintenance more difficult. If we could limit the ways in which objects could be accessed, it would be much easier to debug cases in which these objects are inadvertently overwritten. It would also be easier to maintain the code, as we would need to look at only those functions having access in order to learn about the underlying object.

A class permits us to partition the members in an object of that class into two main groups:[1] private and public. We make members public by using the `public` *access specifier* keyword, as shown in the class definition for `CircleB` in the following example (see directory cl01):

```
// CircleA.h

struct CircleA
{
        int xorigin;
        int yorigin;
        float radius;
};

// CircleB.h

class CircleB
{
public:
        int xorigin;
        int yorigin;
        float radius;
};
```

All members in class `CircleB` are below the `public` access specifier, so they are public, and can be accessed in the same way as the members in the structure `CircleA`. Without the explicit access specifier, the members

[1] A third category, protected, is discussed in §14.

would be private; however, one of the main points of object-oriented design is to make class data members private, so access to them is restricted.

In the example above, and in almost all other examples throughout this book, the class definition is placed in its own header having the same name. For example, class `Point` would be defined inside the user-defined header `Point.h`.

For completeness, here is an example (see directory cl01) that uses these layouts:

```
#include "CircleA.h"
#include "CircleB.h"

void test()
{
        CircleA ca;

        ca.xorigin = 10;
        ca.yorigin = 20;
        ca.radius = 1.5f;

        CircleB cb;

        cb.xorigin = 200;
        cb.yorigin = 50;
        cb.radius = 3.67f;
}
```

11.2 Data Hiding

Using the following example (see directory cl02), let's see the implications of making a member private:

```
// Circle.h

class Circle
{
        int xorigin;
public:
        int yorigin;
private:
        float radius;
};
```

```
#include "Circle.h"

void f(Circle *p);

int main()
{
        Circle c;

/*1*/   c.xorigin = 4;          // error, member is private
/*2*/   c.yorigin = 5;          // okay, member is public
/*3*/   c.radius = 6;           // error, member is private

        f(&c);

        return 0;
}

void f(Circle *p)
{
/*4*/   p->xorigin = 4;         // error, member is private
/*5*/   p->yorigin = 5;         // okay, member is public
/*6*/   p->radius = 6;          // error, member is private
}
```

By default, all members of a class are private; it is as if there is an occurrence of the `private` access specifier keyword before the first member. In this case, we have one implicitly private member, one explicitly public member, and one explicitly private member, in that order. The simplest approach is to declare all the private members first, followed by the `public` access specifier and then the public members.[1]

C++ programmers often say that `c` is an *instance* of the object type `Circle`, or that `c` is an *instantiation* of that type.

In `main`, even though object `c` is in scope, we cannot access the private members `xorigin` and `radius` directly in cases 1 and 3. And when we pass the address of `c` to function `f`, we are likewise prohibited from accessing those members in cases 4 and 6. On the other hand, the public member `yorigin` is directly and indirectly accessible, as shown in cases 2 and 5.

[1] Some programmers prefer to do the exact opposite; public first then private.

As we learned in §7.1, a structure can contain more than data, and since a class is a structure, a class can do likewise; for example (see directory cl03), it can contain enumerations and type synonyms:

```
// Auto.h

class Auto
{
public:
        enum Color {red, white, blue};
        // …
private:
        Color auto_color;
        typedef float coefficient;
        // …
};

#include "Auto.h"

void test()
{
/*1*/   Auto::Color paint = Auto::red;  // can access public enumeration
/*2*/   Auto::coefficient exhaust;      // can't access private type name
}
```

Since the names `Color` and `coefficient` belong to class `Auto`, references to these names must be prefixed with that class name, as shown in cases 1 and 2. However, as the type synonym `coefficient` is private, it cannot be accessed by application code.

In this example, the enumeration is public yet the member that uses it, `auto_color`, is private. Since the declaration of the type must precede its first use, we must declare the public part first.

If the user program cannot access private members, what use are they? To answer that we need to learn about member functions, the topic of the next section. For now, the important lesson is that **application program code cannot access the private members of any class objects to which it has access**.

11.3 Encapsulation

Until now, we have seen only two kinds of function linkage: internal and external. These are achieved via the `static` and `extern` keywords, respectively. A `static` function is callable from anywhere inside the source file in which it is defined, while an `extern` function is globally callable. C++ also provides a much finer granularity of access to functions, as we will see. For example, the designer of a class can limit access to its private members to a specified set of functions, as the following program demonstrates (see directory cl04):

```
// Circle.h

class Circle
{
        int xorigin;
        int yorigin;
        float radius;
public:
        void init(int xo, int yo, double rad);
        void print() const;
};
```

In this case, all the data members are private. However, there are two function prototypes in the public section. This may look rather unusual, because it appears that an object of that class contains those functions, however, this is not the case. These functions, called *member functions*, are the *only* functions that are authorized to access the private members.

Why are all the data members private? Why limit access to them? By providing a family of public member functions that can access the private data, we give users of this class a way to perform the operations they need, yet without their needing to access that data directly themselves. This means that the designer of the class can change the internal representation of that class without affecting end-user code, since these changes can be accommodated in the member functions. So, applications programs interact with class objects via a public functional interface, and as long as that interface stays the same, the layout and inner workings of those objects is of no concern to those programs.

Here's how we call these functions (see directory cl04):

```
#include <iostream>
using namespace std;
#include "Circle.h"

int main()
{
        Circle c1;
        c1.init(5, 4, 10.5);    // call init for c1
        c1.print();             // call print for c1

        Circle c2;
        c2.init(2, 9, 5.3);     // call init for c2
        c2.print();             // call print for c2

        return 0;
}
```

The key lies in the expression `c1.init(5,4,10)`. This is a call to the member function `init`. To select a member inside a structure or class, we use the dot operator, and if that member is a function, we then call it using the `()` function call operator. We cannot call a member function unless we are operating on an object of its parent class. Specifically, `c1.init(...)` calls the `init` function to operate directly on the object `c1`, while `c2.init(...)` invokes `init` to operate on object `c2`.

Let's look at the definitions of these functions (see directory cl04):

```
#include "Circle.h"

void Circle::init(int xo, int yo, double rad)
{
        xorigin = xo;
        yorigin = yo;
        radius = static_cast<float>(rad);
}
```

Note the `Circle::` prefix on the function's name. This declares that `init` is a member function of class `Circle`.

The type `float` was chosen to represent the radius, since that type provides sufficient range and precision and, on most systems, it takes up less memory than objects of type `double` or `long double`. Why then is the type of the third argument of `init`, `double`? In §4.3.6, we briefly discussed the idea of narrow and wide types. Suffice it to say that for most C++ programmers, `double` tends to be the more natural choice for floating-point values; it certainly is the type of a floating-point constant unless a suffix is added.

The important thing here is that the type used in the public interface must be able to handle at least the range and precision of that needed internally, and the range of values supported by `double` is certainly a superset of those for `float`. Since the assignment in `init` can theoretically result in a loss of range and/or precision, an explicit cast from `double` to `float` has been used. From this we learn two things: first, the type we use for internal representation need not be the same as that used in the public interface, and second, when implementing a class, we should do all we can to eliminate compiler warnings such as those issued by better compilers during implicit conversions as mentioned above. By disassociating a member's internal type from its public interface, we are free to change that representation without affecting that interface, and that's most important.

```
#include <iostream>
using namespace std;
#include "Circle.h"

void Circle::print() const
{
        cout << "Circle := [(" << xorigin << ',' << yorigin << "),"
             << radius << "]\n";
}
```

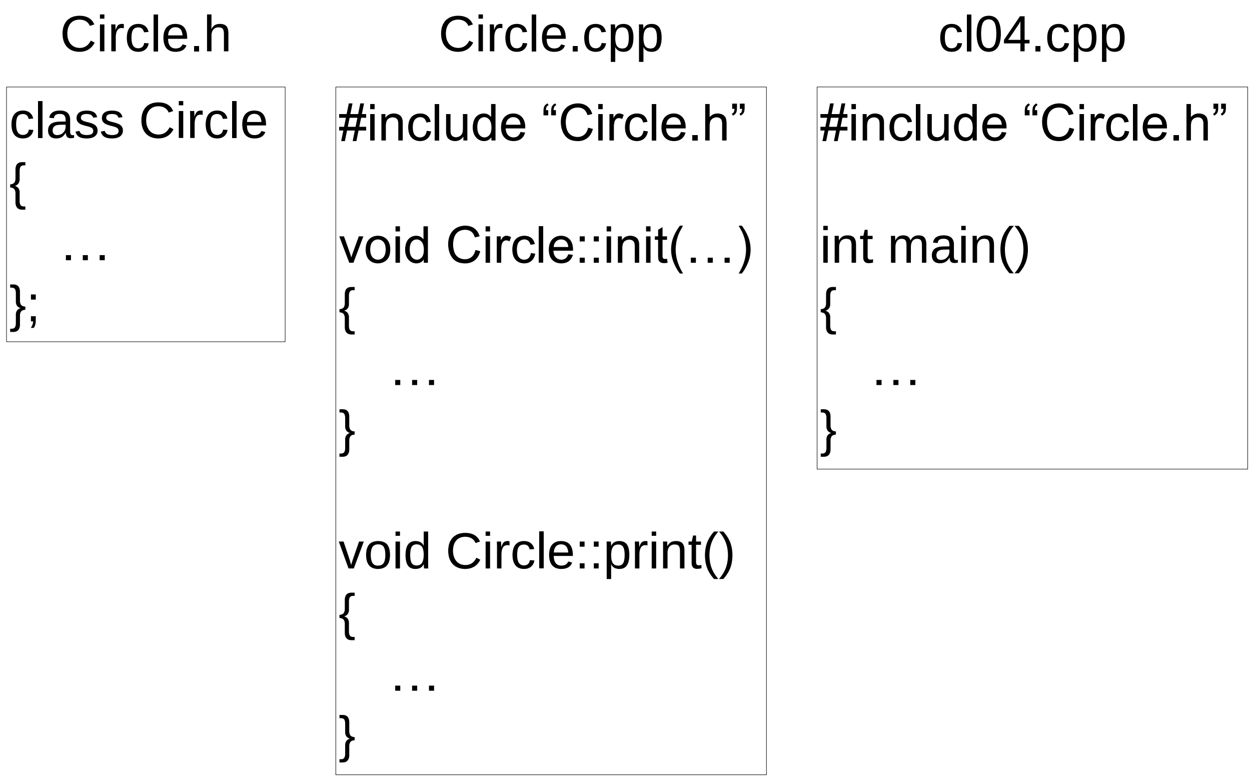
Note that nowhere in these functions do we indicate the object on which we are operating; we simply access the members directly by name without preceding them with a dot or arrow operator. The reason no qualification is needed is that when the functions are called, the calls are implicitly qualified by the object being operated on. That is, when these functions execute, they intuitively know which object on which to operate. (We will see how this works and how to access the "current object" in §11.4.1.)

The output produced is:

```
Circle := [(5,4),10.5]
Circle := [(2,9),5.3]
```

What's the significance of the `const` on `print`? By declaring `print` as being `const`-qualified, we are telling the compiler that `print` does not need write access to the Circle on which it is called to operate; that is, `print` will not modify any of a Circle's data members. Any attempt to modify these members from within `print` will be rejected. We've promised it's a read-only operation and we must make it so; otherwise, we shouldn't have said `const` in the first place.[1] Since this qualifier is included in the name mangling process, if it is present in the function's declaration, it must also be present on its definition, and vice versa. Of course, we cannot make `init` `const`-qualified, since that function needs write access to the Circle's data members.

Let's review the source file organization for this program, which is shown in the following diagram:



```
class Circle
{
    …
};
```

```
#include “Circle.h”

void Circle::init(…)
{
    …
}

void Circle::print()
{
    …
}
```

```
#include “Circle.h”

int main()
{
    …
}
```

The user-defined type `Circle` is defined in its own header, `Circle.h`. The definitions of its member functions are contained in an implementation file for that class, called Circle.cpp. Then the application that uses this type resides in cl04.cpp. Of course, in non-trivial applications, the application code may be distributed among numerous application source files. In addition, for non-trivial classes, the member functions may well be distributed among a number of files as well. However, the model presented above is commonly used, and, for the most part, we will use it in examples in this and subsequent chapters.

[1] In some very special cases, it can be useful to give a `const`-qualified member function write access to one or more private data members. For information on how to do this, see §20.7.

What would happen if we omitted `const` from both the definition and declaration of `print`? In this example, nothing; however, consider the following scenario:

```
class Circle
{
        // …
public:
        void print();           // no const
};

void process(const Circle& c, const Circle *pc)
{
        // …
        c.print();              // won't compile
        pc->print();            // won't compile
}
```

Function `process` has been given two Circle arguments, one by reference, and the other by address and, since both have the `const` qualifier, `process` does not have write access to either Circle. However, when `print` is called, the compiler believes that function needs write access; after all, if `print` did not, it would have been `const`-qualified! **The lesson then is to `const`-qualify every member function for which it makes sense**, for quality assurance purposes. Remember, a `const` member function can *never* call a non-`const` member function.

Consider the case in which not only is the same name used for member functions in different classes, but it is also a nonmember function name as well (see directory cl05):

```
// Circle.h

class Circle
{
        int xorigin;
        // …
public:
        void init(int xo, int yo, double rad);
        void print() const;
};

// Rectangle.h

class Rectangle
{
        // …
public:
        void init(int xo, int yo, int xl, int yl);
        void print() const;
};
```

```
// Misc.h

void print();           // non-member function

#include <iostream>
using namespace std;
#include "Circle.h"
#include "Rectangle.h"
#include "Misc.h"

int main()
{
        Circle c;
/*1*/   c.init(0, 0, 0.0);
/*2*/   c.print();      // call Circle::print

        Rectangle r;
/*3*/   r.init(0, 0, 1, 2);
/*4*/   r.print();      // call Rectangle::print

/*5*/   print();        // call ::print

        return 0;
}
```

We now have three functions called `print`: one member function each for classes `Circle` and `Rectangle`, and one nonmember function. From this, we learn that the parent class name of a member function is also factored into name mangling of function names.

Consider the following definition of member function `init` in class `Circle`:

```
#include "Circle.h"
#include "Misc.h"

void Circle::init(int xo, int yo, double rad)
{
/*6*/   xorigin = xo;
        // …
/*7*/   print();        // call Circle::print
/*8*/   ::print();      // call ::print
}
```

When the compiler comes across an identifier in the body of a member function, it first resolves that name to local variables and then parameters, such as `xo`. If it cannot find a match, it looks at the names of the members in the parent class—that's where it finds `xorigin` and `print`. However, if we want to refer to a name outside of the class when one already exists (or might exist) in that class in the future, we must force the compiler to look outside the class by using the global scope operator `::`, as shown in case 8.

In many classes, all data members are made private and all member functions are made public. We really don't want to make data members public, since that exposes the internal representation; besides, we can always give public access to them via a public member function. However, it is not uncommon to have private member functions. For example:

```
class Employee
{
        // …
        void changeBirthMonth();        // private member function
public:
        // …
};
```

Like private data member access, private member functions can be called only by another public or private member function of the same class. In this case, we don't want an application program changing the birth month, since doing so could invalidate that date; for example, changing from month 1 to month 2 when the day was 30. So, private member functions are tools for other member functions. Since they are private, their existence need not be made known to users of their class; they are not part of the public interface.

Exercise 11-1*: Define a class called `Date` that has three private integer members: `day`, `month`, and `year`. Dates for other than the 20th century must be supported. Define a member function called `init` that can be used to initialize an existing `Date` object. `init` is passed three arguments: year, month, and day, in that order. (If a year is less than 100, assume it is a year in the 20th century.) Assume the arguments to `init` are in-range and their combination represents a valid date.

Also, define a member function called `print` that can be used to display the current contents of a `Date` object once it has been initialized. The display format is dd-Mmm-yyyy, where dd has a leading zero for days less than 10. Months are displayed as Mmm, such that January is Jan, February is Feb, and so on. Define and initialize some `Date` objects, and display their values. The `print` function must ensure that the numbers it outputs are displayed in decimal. (See §10.1 for an example of saving and restoring an output stream's context.)

There are a number of integer types that can be used to represent the `year`, `month`, and `day` members. Chose types that make a Date object as compact as possible. (See labs directory lbcl01.)

11.4 More on Member Functions

11.4.1 Argument Passing Machinery

How does a member function "know" which object it is operating on and can we get hold of that object's name? Whenever we call a member function, a pointer to the object for which the member function is invoked, is passed as a hidden argument. Consider the following:

```
void test()
{
        Circle c1;
/*1*/   c1.init(5, 4, 10.5);    // call init for c1
}

void Circle::init(int xo, int yo, double rad)
{
        xorigin = xo;
        yorigin = yo;
        radius = static_cast<float>(rad);
}
```

When `init` is called in case 1, the address of `c1` is secretly passed in. Inside `init` this hidden argument is available by name, even though it is never declared explicitly. That name is `this`, a keyword that denotes a local variable of automatic storage class that exists in every member function.[1] Its type is "`const` pointer to object for which this member function was invoked". In this case, `this` has type `Circle * const`, and its value is the address of `c1`. Note that `this` is `const`-qualified, so we cannot make it point anywhere but at the object on which `init` was called to operate.

The type of `this` is slightly different inside `print`, since that function is `const`-qualified. Specifically, given that `print` has read access only, the type of `this` is `const Circle * const`. Since the compiler sees that `this` points to a read-only Circle, it won't let us modify that object.

Knowing about `this`, we can implement `init` as follows, instead:

```
void Circle::init(int xo, int yo, double rad)
{
        this->xorigin = xo;
        this->yorigin = yo;
        this->radius = static_cast<float>(rad);
}
```

However, all of the uses of `this->` are redundant, since that's what the compiler has been doing for us automatically.

`this` is defined and initialized before any local, automatic variables declared in our function, so it is available for use immediately that function is entered.

If the use of `this` is always implied inside member functions, why then would we want to use it? Consider the following, not uncommon, programming style:

```
void Circle::init(int xorigin, int yorigin, double radius)
{
        this->xorigin = xorigin;
        this->yorigin = yorigin;
        this->radius = static_cast<float>(radius);
}
```

[1] Actually, it only exists in non-static member functions, as we will learn later.

Rather than inventing different names, why not give the parameters the same names as their corresponding data members? If this approach is used, the explicit qualification of `this->` is needed on the left-hand side of the assignments; without it, we'd be assigning each parameter's value to itself! Remember, the compiler looks to local variable and parameter names first when resolving the meaning of identifiers. Of course, we can avoid this use of `this` simply by using different names.

The name `this` also is needed in member functions that return their hidden argument by value, address, or reference. For example (see directory cl06):

```
// Circle.h

class Circle
{
        // …
        float radius;
public:
        // …
        Circle larger(const Circle& c) const;
};

#include "Circle.h"

void test()
{
        Circle c1, c2, c3;
        // …

        c3 = c1.larger(c2);
}

#include "Circle.h"

Circle Circle::larger(const Circle& c) const
{
        return radius <= c.radius ? c : *this;
}
```

Member function `larger` returns by value the Circle that has the larger radius. If the hidden argument points to the larger object, that object must be returned by value by dereferencing `this`. Since `larger` does not need write access to the Circle it is called to operate on, that function is `const`-qualified. Similarly, the Circle passed in explicitly should not be modified by `larger`, so it is passed using a reference to `const`.

Another example of needing to use `this` involves a member function that is building a linked data structure, in which case it can use `this` when it needs to refer to the address of the current object, which it is adding to that list.

11.4.2 Inline Member Functions

Member functions can also be inline. For example:

```
class Circle
{
        // …
public:
        void init(int xo, int yo, double rad)
        {
                /* function body */
        }
};
```

Although the `inline` keyword can be used in front of the function's return type, its use in this example is redundant. While this approach is common, for classes containing more than a few member functions, defining them inside the class can make it harder to identify the members of that class by simple inspection. We can move the member function definitions outside the class body while still keeping them as inline functions, as follows:

```
class Circle
{
        // …
public:
        void init(int xo, int yo, double rad);
};

inline void Circle::init(int xo, int yo, double rad)
{
        /* function body */
}
```

Assuming the class definition and all inline functions defined outside the class are placed in a header, we can include this header wherever this class is used, allowing the compiler to inline the member functions.

A member function defined inside a class can refer to member functions following it in the class definition. This is because such function definitions are actually treated by the compiler as if they had followed the class definition in the source.

In the member functions we have seen thus far, we have accessed data members directly by name. As such, any changes to the way in which data is represented can affect some, if not all, member functions. Consider the case in which a list is stored as a private array of pointers. This was known when the member functions were written and was exploited by using array subscripting to access strings. However, later on, the class designer decided to store the strings in a linked list or perhaps even in a disk file. Now, all direct accesses to that array must be changed.

If we access private members only indirectly, for example, through `set*`and `get*`functions—called *accessors* and *mutators*, respectively—we can make the program immune from changes in the way in which the private data members are represented. Of course, such changes would require the member-access function implementations to be changed accordingly. However, the user interface to those functions would remain the same. The following example (see directory cl07) contains such a set of functions for the `Circle` class:

```
// Circle.h

class Circle
{
        int xorigin;
        int yorigin;
        float radius;
public:
        int getXorigin() const     { return xorigin; }
        int getYorigin() const     { return yorigin; }
        double getRadius() const   { return radius;  }

        void setXorigin(int xo)    { xorigin = xo; }
        void setYorigin(int yo)    { yorigin = yo; }
        void setRadius(double rad) { radius = static_cast<float>(rad); }

        void init(int xo, int yo, double rad);
        void print() const;
};
```

By defining these six functions inline, we reduce them to code that is just as efficient as if we had accessed the members directly. We also hide the explicit conversion from `double` to `float` in `setRadius`, so no other member functions need worry about this. Here are the revised versions of `init` and `print`:

```
// Circle.cpp

void Circle::init(int xo, int yo, double rad)
{
        setXorigin(xo);
        setYorigin(yo);
        setRadius(rad);
}

void Circle::print() const
{
        cout << "Circle := [(" << getXorigin() << ','
             << getYorigin() << ")," << getRadius()  << "]\n";
}
```

There is no direct way to prohibit member functions from accessing private members directly. Therefore, if we are converting direct accesses to indirect accesses in an existing program, how could we absolutely be sure we changed all the places necessary? It's really quite simple: change the private data members' names and recompile; the only errors you should get are from the member access functions.

We have mentioned the idea of minimizing the changes required when the internal representation of a class is changed. Let's consider a concrete example, in which we have been using the Circle class, and then later on, we define a class called `Point` that can represent a two-dimensional point. It might be useful to change the separate x- and y-coordinates in `Circle` into a Point. For example (see directory cl08):

```
// Point.h

class Point
{
        int xcoord;
        int ycoord;
public:
        int getX() const  { return xcoord; }
        int getY() const  { return ycoord; }
        void setX(int xo) { xcoord = xo; }
        void setY(int yo) { ycoord = yo; }
};

// Circle.h

#include "Point.h"

class Circle
{
        Point origin;
        float radius;
public:
        int getXorigin() const     { return origin.getX(); }
        int getYorigin() const     { return origin.getY(); }
        double getRadius() const   { return radius;  }

        void setXorigin(int xo)    { origin.setX(xo); }
        void setYorigin(int yo)    { origin.setY(yo); }
        void setRadius(double rad) { radius = static_cast<float>(rad); }
};
```

Only four functions are impacted by the change in internal representation: `getXorigin`, `getYorigin`, `setXorigin`, and `setYorigin`. Any other of Circle's member functions will be immune from this change, since they were already written in terms of these four.

When one class type contains a member of another class type, this is referred to as *composition* or *containment*.

Exercise 11-2*: Using the Date class defined in the previous exercise, remove all direct access to the private members from within the member functions, by defining and using a number of accessors and mutators, which are trivial enough that they should be inlined. For example:

```
int getDay();
int getMonth();
int getYear();

void setDay(int dd);
void setMonth(int mm);
void setYear(int yy);
```

Now the program is immune from changes in the way in which the private data members are represented. For example, instead of storing the date as year, month, and day, we could store it using a year and Julian day. Of course, such changes would require the new member access functions to be changed accordingly. (See labs directory lbcl02.)

11.4.3 Overloading Member Functions

Since functions can be overloaded, so too can member functions. For example (see directory cl09), we might want multiple versions of `init`:

```
// Circle.h

class Circle
{
        // …
public:
        // …
/*1*/   void init(int xo, int yo, double rad);
/*2*/   void init(int xo, int yo)       { init(xo, yo, 1.0); }
/*3*/   void init(double rad)           { init(0, 0, rad); }
/*4*/   void init()                     { init(0, 0, 1.0); }
        // …
};
```

Note how the three new versions of `init` are defined in terms of the original version, thereby making them immune to changes in private implementation. And since they are trivial functions, they are defined inline.

```
#include "Circle.h"

void test()
{
        Circle c;

        c.init(5, 4, 10.2);     // call version taking 3 arguments
        c.init(3, 6);           // call version taking 2 arguments
        c.init(1.3);            // call version taking 1 argument
        c.init();               // call version taking 0 arguments
}
```

Function overloading works by having different signatures for the same-named function. We also can have two versions of the same function in which both have the same number *and* type of arguments. This is possible if one function is qualified and the other is not, or if they have different qualifiers. For example (see directory cl10):

```
// Manager.h

class Manager
{
        // …
public:
        // …
/*1*/   void process();         // used for writeable objects
/*2*/   void process() const;   // used for read-only objects
};

#include "Manager.h"

void test()
{
/*3*/   Manager c1;
/*4*/   const Manager c2 = c1;

/*5*/   c1.process();   // call non-const version
/*6*/   c2.process();   // call const version
}
```

Since the non-`const` version of `process` has write access to the object on which it is called to operate, it will only be called to operate on objects that are writeable.

Exercise 11-3*: Using the Date class defined in the previous exercise, overload the `init` function to accept the following sets of arguments:

```
init(int year, int month, int day)
init()
init(const string& dateString)
```

If no argument is provided, the date to use is Jan 1, 1800; however, do not hard-code the default day, month, and year values in any function. Instead, use an unnamed enum type to define them as named constants, and make them available to all member functions of the class.

In the third version of `init`, assume the string has a valid format and represents a real date of the form as output by `print`.

The month name table previously needed inside function `print` is now also needed inside the third version of `init`. One way to deal with this is to hide this table inside a nonmember function.

Hint: For some ideas on how to decode the string into its constituent parts, see function `unpack_date` in §10.4.

(See labs directory lbcl03a. The files in directory lbcl03b contain an alternate solution to the month name table-sharing problem.)

11.4.4 Default Argument Values

We can define default argument values for member functions. For example (see directory cl11), we can re-implement the earlier version of `Circle` as follows:

```
// Circle.h

class Circle
{
        // …
public:
        // …
        void init(int xo = 0, int yo = 0, double rad = 1.0);
        void init(double rad)           { init(0, 0, rad); }
        // …
};

#include "Circle.h"

void test()
{
        Circle c;

        c.init(5, 7, 2.3);      //    init(int, int, double)
        c.init(5, 7);           // => init(5, 7, 1.0)
        c.init(5);              // => init(5, 0, 1.0)
        c.init();               // => init(0, 0, 1.0)
        c.init(5.0);            //    init(double);
}
```

In doing so we eliminate two of the versions of `init`, since they are subsumed by the one having default arguments. However, that new version provides more capability than we had previously, by allowing us to call `init` with one `int` argument only. On further study, we might conclude that this is not a good idea. Why allow `init` to be called with an x-, but no y-coordinate? Is this useful? And even if we convince ourselves it is, it is important to note that we cannot do the opposite. For some classes it is better to have overloaded functions with specific signatures than to allow more combinations than we really need with default argument values. And, of course, a class can have overloaded functions some of which have default argument values.

Exercise 11-4*: Extend the `print` function of the `Date` class so that you can optionally pass it a print format, which has some enum type. The formats to implement are: dd-Mmm-yyyy, mm/dd/yyyy, and yyyy/mm/dd. If the print format argument is omitted, dd-Mmm-yyyy should be used as the default format. Hint: The example in §2.7 shows how you might define and use the enum type. (See labs directory lbcl04.)

11.5 Class Members versus Instance Members

In all of the classes we have seen thus far, each object contained its own set of data members, since it's these values that distinguish one object from another. This kind of data is referred to as *instance data*, since each instance of some object type gets its own set of descriptive data.

The alternative to this is to have data that belongs to the class itself; that is, data that is shared by all instances. This kind of data is referred to as *class data*. In the following example (see directory cl12), class `Message` contains a number of instance data members that describe the properties of each particular Message. Since each new

Message must have a unique ID, the class needs some way of keeping track of the last message ID used, so it knows which ID to issue next:

```
// Message.h

#include <string>
using namespace std;

class Message
{
        string messageText;
/*1*/   unsigned long int messageID;

/*2*/   static unsigned long int lastIssuedID;
/*3*/   static unsigned long int getNextID() { return ++lastIssuedID; }
public:
        const string& getMessageText() const { return messageText; }
/*4*/   unsigned long int getMessageID() const { return messageID; }

        void init(const string& text);
        void print() const;
};
```

In case 2, we define a data member called `lastIssuedID` to keep track of the most recently issued ID. Because we only want one of these for the whole class, we have declared it `static`. In case 3, we define the function `getNextID`, which issues the next ID in increasing sequence. It is private, since it will only be called by other member functions. It too is declared `static`. A static member function is restricted to accessing other static data and functions only; specifically, it *cannot* access instance data nor can it call non-static member functions.

```
// Message.cpp

#include "Message.h"

/*5*/
unsigned long int Message::lastIssuedID = 0;

void Message::init(const string& text)
{
        messageText = text;
/*6*/   messageID = getNextID();
}
```

The static data member `lastIssuedID` does not physically occupy space in any Message; that is, `sizeof(Message)` reports the sum of the sizes of the instance data members only. How then do static data members get memory allocated and initialized? To do this, we must define and initialize them separately from the class definition, and this is usually done in the same source file in which the non-inline member functions are defined. So, in case 5 above, we define `lastIssuedID` to be an unsigned integer belonging to the class `Message`. Most importantly, we must *not* use the keyword `static` in the definition. This variable really is a global variable whose access happens to be severely restricted by the compiler; adding `static` would make it local to its parent source file instead.

```
void Message::print() const
{
        cout << "Message " << getMessageID() << ": " << getMessageText() << '\n';
}

#include "Message.h"

int main()
{
        Message msg1;
        msg1.init("Welcome");
        msg1.print();

        Message msg2;
        msg2.init("Hello");
        msg2.print();

        Message msg3;
        msg3.init("Hi");
        msg3.print();

        return 0;
}
```

The main program simply defines three Messages, initializing each with a string. The output produced is:

```
Message 1: Welcome
Message 2: Hello
Message 3: Hi
```

As we can see, each message was assigned a unique ID, in sequence.

In the previous example, the static data member was only used "behind the scenes". Here's another example (see directory cl13) in which a static data member is used to keep track of the foreground color to be used when some graphical object is drawn. The application program can set this color and it can find out the current foreground color:

```
// Diagram.h

class Diagram
{
public:
        enum Color {red, blue, green, black};
private:
        // other instance data goes here

/*1*/   static Color foreground;
public:
        void draw() const;
/*2*/   static Color getForeground() { return foreground; }
/*3*/   static Color setForeground(Color col);
};
```

In this example, both of the access functions are public; they are also static, since they only access static members.

```
// Diagram.cpp

/*4*/ Diagram::Color Diagram::foreground = Diagram::black;

Diagram::Color Diagram::setForeground(Color col)
{
        Color oldColor = getForeground();
        foreground = col;
        return oldColor;
}
```

As we can see, `setForeground` not only sets the foreground color as specified, as a convenience, it returns the previous foreground color. The definition for this function is quite simple and it is a subjective call as to whether or not it should have been declared inline.

```
#include "Diagram.h"

int main()
{
        Diagram d1, d2;

        // init d1 and d2 somehow

/*5*/   d1.draw();
/*6*/   Diagram::setForeground(Diagram::red);
/*7*/   d1.setForeground(d2.red);               // bad style
/*8*/   d2.setForeground(d1.red);               // bad style
/*9*/   d2.draw();

        return 0;
}
```

Note that in case 6, we call `setForeground` without saying which Diagram object it is to operate on; instead, we prefix it with `Diagram::`, its parent class name. This is permitted, because a static member function can only access static members, which, in turn, belong to the whole class, not to any one instance of that class. Since there can only be one member by this name in this class, the call is unambiguous. Cases 7 and 8 do *exactly* the same thing as case 6; however, they are covert. The fact that d1 and d2 are used implies that they somehow play a role, however, they don't. In case 7, once the compiler sees that `setForeground` is a static member of d1 it loses all interest in d1, since it doesn't need that object's data. Likewise, for d2 in the argument `d2.red`.

Since a static member function needs no object to operate on, no hidden argument is passed to that function, so the keyword `this` cannot be used inside a static member function. And if there is no hidden argument, a static member function cannot be declared `const`, since that keyword indicates that the object whose address is secretly passed must be treated as read-only. Finally, we cannot have a static member function and a non-static member function in the same class if they have the same name and argument type list. If this were allowed, cases 7 and 8 above would be ambiguous.

In the following example (see directory cl14), we have a number of attributes that are used in laying out a page in some word processing-like application:

```
class PageAttributes
{
        static int pageWidth;
        static int pageHeight;
public:
        static int getPageWidth() { return pageWidth; }
        static int getPageHeight() { return pageHeight; }
        static int setPageWidth(int width);
        static int setPageHeight(int height);
};
```

The interesting issue here is that *all* the data members are static; that is, there is no instance data in the class. What is the point of defining a type such that when we define objects of that type, they are empty? Certainly, this is a non-intuitive use of a class, but nonetheless, it is very useful. It is not intended that we ever define objects of this type; all we are doing is encapsulating a number of global variables inside a class and providing access functions, so we can change the representation of these variables at any time. We can also add debugging and/or trace statements inside the access functions to monitor how the values are being changed. Since all users must go through these member functions, we have a place at which we can intercept all set/get actions on these variables. If we had used regular global variables instead, we would have none of this flexibility. Instead, to trace their values, we'd have to put trace statements in *every* place in which these variables' values were changed, rather than just inside these access functions.

This example shows very clearly that the memory allocated for static data members is *not* physically part of any object of that class. Whether or not objects of that class exist has no bearing on the static data members; they *always* exist.

We'll see in §12.1 how to prohibit anyone from actually defining an object of type `PageAttributes`, since doing so makes no sense.[1]

Since a class's static data members are not stored inside an instance of that class, the constness of instances has no bearing on the static data members. For example, the values of instance members of a `const`-qualified object cannot be changed; however, that does not mean that the values of the class members can't be changed. Whether or not a static data member is `const`-qualified is quite separate from whether or not any instance object is `const`-qualified. We can apply `const` to either, both, or neither.

[1] If we were to define an object of this type, the compiler would be obliged to allocate at least 1 byte for it, so `sizeof` could be applied to that type; `sizeof` must produce a result greater than zero. Also, the object must have a unique address, since its address could be taken.

A static member can have any data type including array; for example:

```
class X
{
        static int values[];
        static int coefficients[][3];
        // …
};

int X::values[] = {10, 20, 30, 40};
int X::coefficients[12][3];
```

Since memory for static members is not allocated inside the class type being defined, the first dimension size of any static array member can be omitted. However, subsequent dimension sizes are required.

The C++ standard added a new way to initialize certain static data members; for example (see directory cl18):

```
class Widget
{
public:
        enum Color {red, white, blue};
        static const int maximum = 10;          // initializer in class
private:
        static const Color defaultColor = red;  // initializer in class
        static int values[maximum];
public:
        void process(int count = maximum);
};

const int Widget::maximum;                      // no initializer
const Widget::Color Widget::defaultColor;       //   "         "
```

Static members that are `const`-qualified and have integer or enumeration type can have their initializer inside the class definition. This allows their constant value to be used later in the class definition in things like array dimensions and bit-field sizes.

11.6 Relaxing Member Access Restrictions

Until now, access to private members of a class has been restricted to member functions of that class. However, sometimes it is useful to give functions from outside that class the same access. We do this by making them *friend functions*, as follows (see directory cl15):

```
// IntMatrix.h

class IntVector;                // tentative definition

class IntMatrix
{
        int numRows;
        int numColumns;
        // …
public:
/*1*/   friend IntMatrix add(const IntMatrix& im, const IntVector& iv);
};
```

We have two classes that refer to each other. Class `IntVector` is tentatively defined, so that it can be referenced before its actual definition is seen. Function add is not a member function of class `IntMatrix` yet it has been given direct access to that class's private members by virtue of the keyword `friend`. Here is the definition of the class `IntVector`:

```
// IntVector.h

class IntVector
{
        int numColumns;
        // …
public:
/*2*/   friend IntMatrix add(const IntMatrix& im, const IntVector& iv);
};
```

It too has given to nonmember function add, direct access to its private members. add's definition follows:

```
#include "IntMatrix.h"
#include "IntVector.h"

IntMatrix add(const IntMatrix& im, const IntVector& iv)
{
        IntMatrix result;

/*3*/   /* access im.numRows, im.numColumns, and iv.numColumns, as well as
           any other private members, in order to create a new IntMatrix
           that contains the sum of these two arguments.
        */

        return result;
}
```

Why does add need access to the private members of both classes? If we want to add a matrix and a vector, we'll need access to their respective data, and that is going to be private. If we make add a member function of one class, it won't be able to access the private data from the other class, and vice-versa. To give it access to the private data of multiple classes, we have two choices: we can make it a member of any one of those classes and a friend of all the others or, we can make it a nonmember function (as in the example above) and make it a friend of all the classes. The later approach is preferred, since add belongs to any one class just as much as it does to any other, so making it a member of one particular class makes no sense; the choice of which would be arbitrary.

In cases in which multiple member functions in one class need to access private members of another class, the whole second class can be made a friend of the first, as follows:

```
class X
{
        // …
};

class Y
{
        friend class X;
        // …
};
```

The important aspect of friendship is that a function *cannot* declare itself a friend of any class; it is the class that identifies its friends.

Friend functions can easily be overused. In the example above, does add *really* need access to the private members of both classes? If we provide public accessors to those private data members that add really needed to use, we can avoid making add a friend of either class, since any function can call those public functions.

A nonmember function can deal with a class object without having to access its private members. For example:

```
class Point
{
        // …
public:
        void draw() const;
};

void display(const Point& p)
{
        // …
        p.draw();
}
```

Since display needs no access to p's private members, it need not be a friend of class Point.

Another situation in which friends are often used unnecessarily has to do with operator overloading functions; we'll learn about them in §13.

Most member functions can be declared instead as non-member function friends. For example (see directory cl16):

```
// Point.h

class Point
{
        int xcoord;
        int ycoord;
public:
        friend int getX(const Point& p);
        friend int setX(Point& p, int xo);
        // …
};

inline int getX(const Point& p)
{
        return p.xcoord;
}

inline int setX(Point& p, int xo)
{
        return p.xcoord = xo;
}

#include "Point.h"

void test()
{
        Point p1;
        int x;

/*1*/   setX(p1, 10);
/*2*/   x = getX(p1);
}
```

However, there's really no advantage over the member function approach in which cases 1 and 2 would be written instead as follows:

```
/*1*/   p1.setX(10);
/*2*/   x = p1.getX();
```

There is no doubt that programmers new to C++ and used to procedural programming initially prefer the first approach, in which every argument is passed explicitly, and no object or dot operator prefix is needed. And while they can design their own classes in this manner, that is not how the standard ones are designed, nor are the vast majority of those they will ever have to use. They should "get with the program!"

Certain special functions must be members; they cannot be friends: These include constructors, destructors, conversion functions, and various operator-overloading functions. We'll learn about these in §12 and §13.

Having shown above that there is no technical advantage to making a member function a friend instead, let us now look at an example (see directory cl17) in which it makes a great deal of sense to move what would normally be a member function, out of the class without necessarily needing to make it a friend either. This example involves a class called `Complex`, which represents a complex number, which, by definition, has separate real and imaginary parts:

```
// Complex.h

class Complex
{
        double real;
        double imag;
public:
/*1*/   Complex sin() const;
/*2*/   friend Complex cos(const Complex& c);
};

/*3*/ Complex tan(const Complex& c);

#include "Complex.h"

void test()
{
        Complex a, b;

        // b gets initialized

/*4*/   a = b.sin();
/*5*/   a = cos(b);
/*6*/   a = tan(b);
}
```

In case 1, we declare `sin` to be a member function while in case 2, we declare `cos` to be a friend and, in case 3, we make `tan` a nonmember function. As we can see in cases 4, 5, and 6, the issue is purely notational. Since it is more natural to the mathematical community to pass the argument explicitly, cases 5 and 6 are preferred over case 4. The only reason to make `cos` a friend is if it needs direct access to the complex number's private members. Since we can avoid that by providing public access member functions, we do not need this friendship—we can simply make it a nonmember/non-friend function like `tan`.

We have seen a number of examples in which a function can be made a friend of some class. However, technically, none of these is a requirement. The only good reason to make a function a friend is to give it access to private members. Once we learn about constructors (§12) and operator overloading functions (§13), we might see a reason to make a function a friend.

If in doubt, be conservative; don't make a function a friend if there is an alternative.

11.7 Classes versus Structures and Unions

The term *class* includes types defined using any of the keywords `class`, `struct`, and `union`. Unless otherwise stated, objects of these kinds have the same general properties and capabilities. For example, all three can have public and private members. However, because of their nature, unions are more restricted than class and structure types. For example, a union cannot contain a `static` or reference data member. Also, an anonymous union cannot have members that are `private`, nor can it have member functions. All class types can contain bit-fields.

Simply stated then, the members of a class are private by default while those of a structure (or union) are public by default. Therefore, we most often use `class` instead of `struct` to avoid having to override the default public access specifier. **Every structure can be implemented as a class and every class can be implemented as a structure.**

12. Object Creation and Destruction

It can be very useful to have a function called automatically each time an object of some class is created. C++ provides such a capability via a special member function called a *constructor*. The main purpose of such a function is to ensure that the object being created is initialized with a predictable and valid value. It can also be useful to have another function be called when that object's memory is released. Such a function is called a *destructor*.

In this chapter, we will see how to define and use constructors and destructors for our own classes. We will also see how they help make dynamic memory allocation fit more smoothly into the language.

12.1 Introduction

A constructor is a member function whose name is the same as its parent class. For example, any constructor for the class `Circle` must also be called `Circle`. A constructor is called automatically every time an object of its type is created. Consider the following program (see directory cd01) that creates four Circles, initializes them, and displays their contents:

```
// Circle.h

class Circle
{
        int xorigin;
        int yorigin;
        float radius;
public:
        Circle(int xo, int yo, double rad = 1.0) { init(xo, yo, rad); }
        Circle(double rad = 1.0)                 { init(0, 0, rad); }

        void init(int xo, int yo, double rad);
        void print() const;

        int getXorigin() const     { return xorigin; }
        int getYorigin() const     { return yorigin; }
        double getRadius() const   { return radius;  }

        void setXorigin(int xo)    { xorigin = xo; }
        void setYorigin(int yo)    { yorigin = yo; }
        void setRadius(double rad) { radius = static_cast<float>(rad); }
};
```

There are two constructors. Like other member functions, constructors can be overloaded; they can also have default arguments and be defined inline. A constructor is needed to initialize an object as it is being created. However, the `init` function reinitializes the value of an existing object. As such, we really need both for most classes, and, since they often do the same thing, it is common that the constructor simply be an inline function that calls `init`, as shown.

Although a constructor looks like and behaves like a `void` function, we cannot explicitly declare it as such; `void` is simply implied.

Here is the main program (see directory cd01):

```
#include "Circle.h"

int main()
{
/*1*/   Circle c1;              // call 2nd constructor with 0 args
/*2*/   Circle c2(3, 5, 2.1);   // call 1st constructor with 3 args
/*3*/   Circle c3(4, 6);        // call 1st constructor with 2 args
/*4*/   const Circle c4(2.5);   // call 2nd constructor with 1 arg

        c1.print();
        c2.print();
        c3.print();
        c4.print();

        return 0;
}
```

The declarations in cases 1–4 cause each object to be initialized using the constructor with the matching argument list. As `c1` has no initializer, the *default constructor*—that having no arguments[1]—is called. If constructors are defined, but none of them is a default constructor, situations requiring a default constructor will cause a compile-time error to be produced.

Case 4 is worth a special mention given that `c4` is `const`-qualified. As we can see, a constructor has write access to an object ordinarily only available for read; after all, even a read-only object has to get created and given an initial value. Note the call to `init` from within each constructor. Since we cannot call `init` on `c4` directly, how does this work? In the case of a constructor only, the type of `this` is *always* a pointer to a non-`const` object, thereby allowing the constructor to modify that object. And since the constructor passes on its `this` to all other member functions it calls, they too have write access (unless, of course, like `print`, they are themselves `const`-qualified).

There is a subtle difference between `Circle c1;` and `Circle c1();`. The first declares `c1` to be an object, whereas the second declares it a function having no arguments, not a call to the default constructor. Beware!

The output produced is:

```
Circle := [(0,0),1]
Circle := [(3,5),2.1]
Circle := [(4,6),1]
Circle := [(0,0),2.5]
```

[1] If all arguments in a constructor have default values, that constructor is also considered to be a default constructor. There can be only one default constructor for a class, however.

The syntax used to initialize a structure or union can also be used to initialize a class object, provided the class has no constructors and all its data members are public. The class may contain static data members, provided they are public.

The name of a destructor must be the same as its class preceded by a tilde (~). For example, the destructor for the class `Circle` must be called ~Circle. Since a destructor is called automatically every time an object of its type has its memory released, a class can have at most, only one destructor, and it must not have any parameters. Consider the following program (see directory cd03):

```
// IntVector.h

class IntVector
{
        int size;
        int *plist;
public:
/*1*/   IntVector(int vectorSize)       // constructor
        {
                size = vectorSize;
                plist = new int[size];
        }

/*2*/   ~IntVector()                    // destructor
        {
                delete [] plist;
        }
};
```

An object of this type does not contain all the information needed to describe fully its value; specifically, since an IntVector object contains a pointer, it is a descriptor for the real data that is stored on the heap. By default, defining an object of type `IntVector` does not allocate memory for the vector of integers; for that to happen we need a constructor to allocate memory. We shall refer to classes that physically contain all their data to be *self-contained classes* while those that do not, shall be called *non-self-contained classes*. `IntVector` therefore, is a non-self-contained class, so it requires a destructor to free up the memory allocated by its constructor. Clearly then, not every class having a constructor needs a destructor. And while it is common for classes to have constructors, but no destructor, the opposite is most unlikely.

Like constructors, destructors cannot be declared with an explicit return type.

The following program (see directory cd04) demonstrates just when constructors and destructors are called for automatic objects:[1]

[1] We'll see what happens with static objects in §12.5, and with dynamic objects in §12.6.

```
#include "IntVector.h"

void test()
{
        IntVector iv1(10);              // iv1 constructor called
        {
                IntVector iv2(15);      // iv2 constructor called
                {
                        IntVector iv3(5);// iv3 constructor called
                }                       // iv3 destructor called
        }                               // iv2 destructor called

        {
                IntVector iv4(30);      // iv4 constructor called
        }                               // iv4 destructor called
}                                       // iv1 destructor called
```

Since all objects have automatic storage duration, they come and go at the start and end, respectively, of their parent block.

A constructor or destructor cannot be `static`, or `const`- or `volatile`-qualified. Despite this, a constructor or destructor can be called for `const`- and `volatile`-qualified objects. They, in turn can also call `const`- and `volatile`-qualified member functions. An object whose class has a constructor or destructor cannot be a member of a union.

We cannot take the address of a constructor or a destructor.

Exercise 12-1*: In §11.5, we saw an example in which unique object IDs were allocated to each Message object as it was initialized via a member function called `init`. The flaw with using `init` in this manner is that this function can be called repeatedly on the same object, but we probably don't want that object's ID to change every time its value does. Clearly, now that we know about constructors, we can set an object's ID once only, when it is created. Modify that program accordingly. (See labs directory lbcd03.)

12.2 Temporary Objects

A constructor can be called directly to create a temporary, unnamed object. For example, given a Point class, we could use something like the following:

```
void drawLine(const Point& p1, const Point& p2);

drawLine(Point(-10, 20), Point(70, 89));
```

Here, the constructor `Point::Point` is called directly to create two Points that are initialized as shown. The temporary objects are then passed to `drawLine` and then are destroyed automatically once `drawLine` returns, thus avoiding the need for the programmer to create named Points.

The lifetime of a temporary object is typically quite short—no longer than the life of its parent expression. An expression that causes more than one temporary object to be created will destroy them in the reverse order of their creation, whatever that may have been, as is the case in the call to `drawLine` above.

Temporary objects are created and destroyed—and therefore, require corresponding constructors and destructors to be called—under other circumstances. Examples include when objects are passed to, and returned from, functions by value, and when an object is created as the result of an explicit or implicit cast.

12.3 Arrays and Nested Objects

As we should expect, we can have an array of Circles. Also, objects can be nested inside structures, unions, and other classes. For example (see directory cd05):

```
// Line.h

#include "Point.h"

class Line
{
        Point point1;
        Point point2;
public:
        Line(int x1 = 0, int y1 = 0, int x2 = 0, int y2 = 0)
        {
                // …
        }
};

#include "Line.h"
#include "Point.h"

int main()
{
/*1*/   Point pts[3] = {Point(2,3), Point(4,5) /* no [2] */ };

/*2*/   Line l(10, 12, 75, 87);

/*3*/   Line c[2][2] = {
                {Line(3, 4, 9, 7), Line(4, 5, 7, 8)},
                {Line(1, 2, 3, 5) /* no [1][1] */  }
        };

        // …

        return 0;
}
```

The elements of a one-dimensional array are constructed in increasing element order, and their destructors are called in the reverse order. Therefore, in case 1, the elements of `pts` are constructed in the order [0], [1], and [2], and then destructed in the order [2], [1], and [0]. Elements without explicit initializers (such as `pts[2]`) are initialized using the default constructor. The initializer expressions are calls to the corresponding constructor.

In case 2, we define a Line object. First, the default constructor for `l`'s `point1` is called, then the default constructor for its `point2` is called, and, finally, `l`'s constructor is called.[1] Then when `l` is destructed, the three destructors are called in the reverse order.

In case 3, we combine the previous cases, by having an array of Lines, each element of which contains a pair of Points. The elements of a two-dimensional array are constructed in increasing by column within row, and their destructors are called in the reverse order. As in case 1, elements without explicit initializers (such as `c[1][1]`) are initialized using the default constructor.

Exercise 12-2*: To the Date class, add a family of constructors that parallel the `init` functions. Now we can initialize a Date when we first create it as well as being able to re-initialize an existing Date. Define a new constructor (and matching version of `init`) as follows:

```
Date(relday ytt);
```

where `ytt` can be any one of the three enumerators `yesterday`, `today`, or `tomorrow`, from the relative day enumerated type `relday`.

In the main program, define a 3×2 array of Dates and initialize the first column only, with the dates for yesterday, today, and tomorrow, using the relative day enumerators. Print the contents of all 6 elements.

Use the `ctime` functions `time` and `localtime` to get today's date. (See labs directory lbcd01.)

Exercise 12-3*: For the Date class, define two public member functions called `earlierDate` and `laterDate` that compare two dates and return the earlier or later one, respectively, by value. Since both do very similar things, you should find it useful to write a third function (say `compareDate`) that does the real work, and is called by both. Have `compareDate` return an `int` whose value is negative if the first date is earlier than the second, zero if the dates are the same, or positive if the first date is later than the second. Here are these functions' prototypes:

```
Date earlierDate(const Date& date2) const;
Date laterDate(const Date& date2) const;
int compareDate(const Date& date2) const;
```

(See labs directory lbcd02.)

12.4 The Effects of Changing Control Flow

There are a number of ways of bypassing the normal flow of control in a program. We can use the statements `break`, `continue`, `goto`, and `switch`, or the library functions `exit`[2] and `abort`. These statements and functions unconditionally transfer control to some other place in the program and, in the process, bypass the usual block termination. Will destructors be called for objects in existence at the time of such a statement or function call? Consider the following program (see directory cd06):

[1] Clearly, it is inefficient to have the default constructors be called as well as having the actual constructor get called immediately after. We'll see how to avoid this redundancy in §12.10.

[2] `exit` and `abort` are holdovers from the standard C library.

```
// Circle.h

class Circle
{
        int xorigin;
        int yorigin;
        float radius;
public:
        Circle(double rad = 1.0);
        ~Circle();
};

#include <iostream>
#include <cstdlib>
using namespace std;
#include "Circle.h"

int main()
{
        Circle c1(1);
        int i;

        for (i = 1; i < 4; ++i)
        {
/*1*/           Circle c2(2);

                if (i == 2)
                {
/*2*/                   break;
                }
        }

/*3*/   exit(0);
        return 0;
}
```

In case 1, a new c2 object is created and destroyed for each iteration of the loop, as we would expect. The break in case 2 forces destructors to be called for automatic objects in the block(s) from which it breaks out. (The same is true for goto and continue, although they are not present in this example.)

Calling exit directly in case 3, bypasses the calling of destructors for any automatic object still in existence. Hence, no destructor is called for c1. Therefore, we should be careful when calling exit from a C++ program. The sibling C function, abort, does not cause destructors to be called at all, which is in keeping with abort's purpose, which is to terminate the program immediately.

12.5 Static Data and Execution Order

Thus far, all of the objects used with constructors have had been automatic, and it is clear when such objects are created and destroyed, but what about static objects? If they really are created at compile or link time, in what order are their constructors and destructors called?

The following example (see directory cd07) contains global Circles, `static` Circles with file scope, and automatic and `static` Circles with block scope. The output shows the order in which the constructors and destructors are called:

```
// Circle.h

class Circle
{
        int xorigin;
        int yorigin;
        float radius;
public:
        Circle(double rad = 1.0)
        {
                radius = rad;
                cout << "++ " << radius << '\n';
        }

        ~Circle()
        {
                cout << "-- " << radius << '\n';
        }
};

#include <iostream>
using namespace std;
#include "Circle.h"

void test1();
void test2();

static Circle c1(1), c2(2);
Circle c3(3);
static Circle c4(4);

int main()
{
        Circle c5(5);
        static Circle c6(6), c7(7);
        Circle c8(8), c9(9);
        static Circle c10(10);
        {
                static Circle c11(11);
                {
                        Circle c12(12);

                        test1();
                }
        }

        {
                Circle c13(13);
        }

        return 0;
}

Circle c14(14), c15(15);

void test1()
{
        Circle c16(16);
        static Circle c17(17);

        {
                Circle c18(18);
        }
}

void test2()
{
        Circle c19(19);
        static Circle c20(20);
}
```

The output produced is as follows; it is arranged in tabular form simply to save space. Read down each column, left to right.

Table 12–1: Constructor and Destructor Call Order

Part 1	Part 2	Part 3	Part 4
++ 1	++ 8	-- 16	-- 10
++ 2	++ 9	-- 12	-- 7
++ 3	++ 10	++ 13	-- 6
++ 4	++ 11	-- 13	-- 15

Part 1	Part 2	Part 3	Part 4
++ 14	++ 12	-- 9	-- 14
++ 15	++ 16	-- 8	-- 4
++ 5	++ 17	-- 5	-- 3
++ 6	++ 18	-- 17	-- 2
++ 7	-- 18	-- 11	-- 1

To make the output easier to follow, the radius of each Circle created is the same number used in that variable's name. For example, `c6` is initialized with 6, `c10` with 10, and so on.

The rules for calling constructors and destructors for static objects are as follows:

- Constructors for nonlocal statics (`c1`–`c4` and `c14`–`c15`) are called in their lexical order within the source module. The initialization of any nonlocal statics in a source module is performed before the first use of any function or object defined in that module. The corresponding destructors are called in the reverse order when `main` returns or when `exit` is called.
- Constructors for local statics (`c6`, `c7`, `c10`, `c11`, and `c17`) are called the first time execution passes through their declarations. The corresponding destructors are called in the reverse order just before the program terminates. Note that function `test2` is not called, so execution never drops through the local static `20`. Consequently, its constructor and destructor are never called. Likewise for the automatic variable 19.

We have learned about the ordering within a program built from a single source file. However, real programs are made up from multiple source modules, each of which may contain static data declarations. How does that affect the order in which constructors and destructors are called? Just how does the executable program know which static constructors and destructors to call? The compiler sees only one source module at a time. The answer is that no ordering is imposed *across* source modules.

12.6 Dynamically Allocated Objects

A big advantage of new and `delete` comes with classes having constructors and/or destructors. If an object created by new is to behave just like an object created using an automatic or static definition, new and `delete` must involve implicit calls to constructors and destructors, respectively, and this is, in fact, the case. For example (see directory cd08):

```
#include "Circle.h"

int main()
{
        Circle *pcir1;
/*1*/   pcir1 = new Circle;             // default constructor is called
        if (pcir1 == 0)
        {
                // allocation failure
        }

/*2*/   pcir1->print();
```

In case 1, we allocate a Circle and since no explicit initializer was provided, the default constructor is called, resulting in the object's being initialized with an origin of (0,0) and radius 1. The expression `pcir1->print()` in case 2 is new but not surprising. Thus far, we have called member functions using a class object's name. However, in this case we do not know its name; we only know its address, so we use the `->` operator to select the `print` member function for class `Circle` to operate on the Circle pointed to by `pcir1`.

```
        Circle *pcir2;
/*3*/   pcir2 = new Circle(1, 3, 3.5);  // constructor taking 3 args is called
        pcir2->print();
```

Case 3 results in the Circle's being constructed using the three arguments provided.

```
        Circle *pcir3;
/*4*/   pcir3 = new Circle [3];  // default constructor is called for each element
        pcir3[1].print();
```

Case 4 results in the default constructor's being used with the elements being constructed in ascending element number order. Arrays cannot be initialized explicitly when using new. However, each element of an array can be initialized to the same value using the default constructor.

```
        delete pcir2;                   // destructor is called
        delete pcir1;                   // destructor is called
        delete [] pcir3;                // destructor is called for each element

        return 0;
}
```

12.7 Copy Constructors

As stated in §12.2, there are a number of contexts in which an object can be copied. They include passing an object by value to a function, returning an object by value from a function, and initializing a new object during its definition. The following code fragment contains an instance of each:

```
C c1;
C c2 = c1;      // initialize

C f(C);
c2 = f(c1);     // pass and return by value
```

Everything works just fine when C is a self-contained class; however, for non-self-contained classes like `IntVector`, simply copying the descriptor leaves two IntVectors pointing to the same data. What we really need is for the descriptor's dependent data to be copied as well. For this, we need to use a *copy constructor*, which has the following general form:

```
C::C(const C&)
```

where C is the class name. The `const` qualifier need not be present and the constructor can have more arguments *provided each has a default value*. So, a copy constructor is one that can be called with only one argument, which is of the class type, and is passed by (possibly `const`) reference.

Here's class `IntVector` with a copy constructor (see directory cd09):

```
// IntVector.h

class IntVector
{
        int size;
        int *plist;
public:
        IntVector(int vectorSize)       // constructor
        {
                size = vectorSize;
                plist = new int[size];
        }

        IntVector(const IntVector& iv);// copy constructor

        ~IntVector()                    // destructor
        {
                delete [] plist;
        }
};

// IntVector.cpp

IntVector::IntVector(const IntVector& iv)       // copy constructor
{
        size = iv.size;
        plist = new int [size];
        for (int i = 0; i < size; ++i)
        {
                plist[i] = iv.plist[i];
        }
}
```

```
#include "IntVector.h"

IntVector f(IntVector iv);

int main()
{
        IntVector iv1(3);
/*1*/   IntVector iv2 = iv1;

/*2*/   /*iv2 =*/ f(iv1);

        return 0;
}
```

Now complete copies are made rather than just copies of the descriptors. However, this does not properly handle the case of assignment between two existing objects. For that we need to overload the assignment operator, which we'll see how to do in §13.

To be maximally robust, a class like `IntVector` must have a copy constructor defined; however, we should avoid using it as much as possible, since it can get quite expensive. For example, if we have a vector of 10,000 integers and we pass or return it by value, the copy constructor must allocate memory for a complete copy and then copy the elements over. So, whenever possible, we should pass and return by address or reference, lest the called functions spend more time copying argument lists and return values than they do with real work.

12.8 Conversion by Constructor

When a constructor is called, its job is to create an object of its parent type using the argument list provided. When the argument list consists of only one argument, the constructor call can be viewed as performing a conversion. For example, given the constructor `Circle(double rad)`, `Circle(2.5)` converts the floating-point value 2.5 into a Circle having that radius and some default origin. Therefore, the following

```
Circle c3(2.5);
```

can be rewritten instead as

```
Circle c3 = 2.5;
```

because the class has a constructor that takes a floating-point argument. Similarly, we can pass any floating-point value to a function that expects a Circle, and that value will be converted to type `Circle` by the constructor. Similar behavior occurs when we return a floating-point value, yet the function's return type is `Circle`. Conversion constructors are automatically called as necessary.

Consider the following example (see directory cd10a):

```
// Circle.h

#include "Point.h"

class Circle
{
        Point origin;
        float radius;
public:
        Circle(const Point& p, double rad = 1.0);
        // …
};

#include "Point.h"
#include "Circle.h"

void f(const Circle& c);

int main()
{
        Point p;
/*1*/   Circle c(p);

/*2*/   Circle c2 = p;          // implicit Point=>Circle conversion
/*3*/   Circle c3 = Circle(p);  // explicit Point=>Circle conversion

/*4*/   f(p);                   // implicit Point=>Circle conversion
/*5*/   f(Circle(p));           // explicit Point=>Circle conversion

        return 0;
}
```

In cases 3 and 5, we call the conversion constructor explicitly to create a Circle from a Point. And in cases 2 and 4, that constructor is called implicitly. Most of the time such implicit conversion is fine; however, occasionally we might find a situation in which an expression of one type is converted implicitly by a constructor to some class type when that conversion was neither intended nor desired. In order to prohibit such conversions, the keyword `explicit` has been added to the language. We can use it as follows (see directory cd10b):

```
explicit Circle(const Point& p, double rad = 1.0);
```

When we do so, cases 2 and 4 above, are rejected. By declaring this constructor as such, we restrict its use to initializing objects and explicit casts.

The keyword `explicit` is permitted only within a class; it cannot be used on any constructor definition outside a class:

```
class X
{
public:
        explicit X();   // okay here
};

explicit X::X()         // error
{
        // …
}
```

12.9 Private Constructors

In §11.5, we saw the definition of a class called PageAttributes whose data members were all static. The question raised at that time was "How do we prohibit anyone from actually creating an object of that type?" The simple answer is "Give it an empty, but private, constructor that does nothing". Since the compiler can't get at the private constructor, we have achieved the desired result. For example (see directory cd12):

```
// PageAttributes.h

class PageAttributes
{
        static int pageWidth;
        static int pageHeight;

/*1*/   PageAttributes() {}     // the only constructor is private
public:
        static int getPageWidth() { return pageWidth; }
        static int getPageHeight() { return pageHeight; }
        static int setPageWidth(int width);
        static int setPageHeight(int height);
};

#include "PageAttributes.h"

void test()
{
/*2*/   PageAttributes p;       // error; default constructor not accessible
}
```

In the following example (see Month.h in directory cd11), we see a private constructor that does useful work, as well as inhibiting application programs from creating any objects of its parent class:

```
// Month.h

#include <string>
using namespace std;

class Month
{
        string fullName;
        string abbrevName;
        char daysLeap;
        char daysNonLeap;

        Month(const string& fullName, const string& abbrevName,
                int daysLeap, int daysNonLeap)
        {
                this->fullName = fullName;
                this->abbrevName = abbrevName;
                this->daysLeap = daysLeap;
                this->daysNonLeap = daysNonLeap;
        }

public:
        static const Month JANUARY;
        static const Month FEBRUARY;
        static const Month MARCH;
        static const Month APRIL;
        static const Month MAY;
        static const Month JUNE;
        static const Month JULY;
        static const Month AUGUST;
        static const Month SEPTEMBER;
        static const Month OCTOBER;
        static const Month NOVEMBER;
        static const Month DECEMBER;

        const string& getFullName() const   { return fullName; }
        const string& getAbbrevName() const { return abbrevName; }
        int getDaysLeap() const             { return daysLeap; }
        int getDaysNonLeap() const          { return daysNonLeap; }

        void display() const;
};
```

A Month object contains four instance data members: full month name, abbreviated month name, number of days in a leap year, and number of days in a non-leap year. The constructor initializes all four fields of a Month, but being private, it can only be called from within the class itself. The class contains 12 public static Month objects, whose values cannot be changed. Let's look in the class implementation file to see how these objects are initialized:

```
// Month.cpp

#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

#include "Month.h"

const Month Month::JANUARY("January", "Jan", 31, 31);
const Month Month::FEBRUARY("February", "Feb", 29, 28);
const Month Month::MARCH("March", "Mar", 31, 31);
const Month Month::APRIL("April", "Apr", 30, 30);
const Month Month::MAY("May", "May", 31, 31);
const Month Month::JUNE("June", "Jun", 30, 30);
const Month Month::JULY("July", "Jul", 31, 31);
const Month Month::AUGUST("August", "Aug", 31, 31);
const Month Month::SEPTEMBER("September", "Sep", 30, 30);
const Month Month::OCTOBER("October", "Oct", 31, 31);
const Month Month::NOVEMBER("November", "Nov", 30, 30);
const Month Month::DECEMBER("December", "Dec", 31, 31);

void Month::display() const
{
        cout << setw(10) << getFullName() << " (" << getAbbrevName() << ") "
                << getDaysLeap() << ", " << getDaysNonLeap() << '\n';
}
```

Each of the 12 Months is constructed via a call to the private constructor. Since these objects are inside the class, they do have access to private members. So, when the application program is executed, before `main` gets control, all static objects are allocated memory and initialized which, in this case, involves calling the private constructor.

Here then is a program (see directory cd11):

```
#include <iostream>
using namespace std;

#include "Month.h"

int main()
{
/*1*/   Month newMonth = Month::APRIL;
        newMonth.display();

/*2*/   Month months[] = {
                Month::MARCH, Month::JANUARY, Month::FEBRUARY
        };

        for (int i = 0; i < sizeof(months)/sizeof(months[0]); ++i)
        {
                months[i].display();
        }

/*3*/   Month::JUNE.display();

/*4*/   const Month *x = &newMonth;
        x->display();

        return 0;
}
```

and here is the output it produces:

```
     April (Apr) 30, 30
     March (Mar) 31, 31
   January (Jan) 31, 31
  February (Feb) 29, 28
      June (Jun) 30, 30
     April (Apr) 30, 30
```

12.10 ctor Initializers

Consider a Circle class defined as follows:

```
// Circle.h

class Circle
{
        int xorigin;
        int yorigin;
        float radius;
public:
        Circle(int xo, int yo, double rad) { init(xo, yo, rad); }

        // …
}
```

It is a relatively common practice to implement the constructor as follows, instead (see directory cd02a):

```
Circle::Circle(int xo, int yo, double rad)
        : xorigin(xo), yorigin(yo), radius(rad)
{
}
```

Here we have what is called a *ctor-initializer*; it follows the function's parameter list and a colon, but precedes the function's body, as shown. And as often happens, all the work is done in that initializer leaving the body empty, making for a strange looking constructor indeed. A ctor-initializer is a comma-separated list of expressions each having the form

member-name (*initial-value*)

While this approach is common, several issues are worth noting. First, it requires us to refer to the private data members by name thereby violating the approach of keeping as many member functions as possible ignorant of the object's physical representation. However, since the number of constructors for a class is usually far less than the number of member functions, we might justify using direct name access in the constructors, but not in any other member functions. Second, it doesn't help with situations in which the constructor is implemented simply as a call to another member function such as `init`. Third, the ctor-initializer approach has a surprising and potentially nasty property if we try to get too cute; for example (see directory cd02b):

```
// Circle.h

class Circle
{
        int xorigin;
        int yorigin;
        float radius;
        // …
};
// Circle.cpp

Circle::Circle(double rad)
        : yorigin(0), xorigin(yorigin), radius(rad)
{
}
```

The order in which the comma-separated expressions are evaluated is *not* left-to-right as written; instead, they are evaluated in the order in which their destination objects were declared inside the class. Specifically, `xorigin(yorigin)` is evaluated first, and since `yorigin` has not yet been initialized and the Circle is automatic (as we'll see below), `yorigin` (and therefore `xorigin`) takes on some unspecified value. Then `yorigin(0)` is evaluated, resulting in that member taking on the value 0. Finally, `radius(rad)` is evaluated. Here's the application code:

```
#include "Circle.h"

int main()
{
/*1*/   Circle c1(2.5);
        c1.print();

/*2*/   static Circle c2(2.5);
        c2.print();

        return 0;
}
```

and here's the output produced on one 32-bit system:

```
Circle := [(2012894640,0),2.5]
Circle := [(0,0),2.5]
```

In case 1, we see that the unspecified value used was 2012894640; of course, it could have been anything, including the value 0. In case 2, the initial value of `yorigin` is *not* unspecified, since `c2` is static instead of automatic; and as we know, static variables take on the default initial value of zero.

Writing cute ctor-initializers like the one above relies on a particular ordering of the data members; if that is violated, such bugs can be hard to locate. After all, it is "obvious" to most readers of the constructor source that these expressions should be evaluated left-to-right.

The uses of a ctor-initializer above are unnecessary; however, there are situations in which they *must* be used. A data member can be a reference to another object, so that all operations on that member actually take place on the object to which it is referenced. A data member can also be `const`-qualified. Reference and `const`-qualified data members can *only* be initialized using a ctor-initializer.

Here's another situation in which a ctor-initializer can help. Consider the following (see directory cd13a):

```
// Point.h

#include <iostream>
using namespace std;

class Point
{
        int xcoord;
        int ycoord;
public:
        int getX() const  { return xcoord; }
        int getY() const  { return ycoord; }
        void setX(int xo) { xcoord = xo; }
        void setY(int yo) { ycoord = yo; }

        Point(int xo = 0, int yo = 0)
        {
                setX(xo);
                setY(yo);
                cout << "Point::Point (" << getX() << "," << getY() << ")\n";
        }
};

// Line.h

#include <iostream>
using namespace std;

#include "Point.h"
```

```
class Line
{
        Point point1;
        Point point2;
public:
        Line(const Point& p1, const Point& p2)
        {
                point1 = p1;
                point2 = p2;
                cout << "Line::Line, set Points to ("
                     << p1.getX() << "," << p1.getY() << ") and ("
                     << p2.getX() << "," << p2.getY() << ")\n";
        }
};

#include "Line.h"

int main()
{
        Point p1(10,12), p2(75,87);

        Line l(p1, p2);

        return 0;
}
```

The output produced is as follows:

```
Point::Point (10,12)
Point::Point (75,87)
Point::Point (0,0)
Point::Point (0,0)
Line::Line, set Points to (10,12) and (75,87)
```

When p1 and p2 are created, their constructors are called, and they issue the appropriate message. However, why do we get two messages from the Point constructor saying it constructed two Points at the origin? The reason for this is that when the Line constructor is called, before its body is executed, the default constructor for each class object contained within that Line is called first. So, `Point::Point` is called with no arguments, resulting in the default values of zero being used. Then the Line constructor executes, which sets the Point members to their intended value. So this approach results in the contained Points being initialized twice, once implicitly and once explicitly. The only way to avoid this is to define the Line constructor with a ctor-initializer, as follows (see Line.h in directory cd13b):

```
Line(const Point& p1, const Point& p2) : point1(p1), point2(p2)
{
        cout << "Line::Line, set Points to ("
             << p1.getX() << "," << p1.getY() << ") and ("
             << p2.getX() << "," << p2.getY() << ")\n";
}
```

Now the output produced is:

```
Point::Point (10,12)
Point::Point (75,87)
Line::Line, set Points to (10,12) and (75,87)
```

As we can see, the underlying Points are only initialized once, however, we have had to access the private members directly by name.