# 1

# Introduction to Computers and C++ Programming

## Objectives

- To understand basic computer science concepts.
- To become familiar with different types of programming languages.
- To understand a typical C++ program development environment.
- To be able to write simple computer programs in C++.
- To be able to use simple input and output statements.
- To become familiar with fundamental data types.
- To be able to use arithmetic operators.
- To understand the precedence of arithmetic operators.
- To be able to write simple decision-making statements.


*High thoughts must have high language.*
Aristophanes

*Our life is frittered away by detail … Simplify, simplify.*
Henry Thoreau

*My object all sublime*
*I shall achieve in time.*
W. S. Gilbert

## Outline



## 1.1 Introduction

Welcome to C++! We have worked hard to create what we hope will be an informative, entertaining and challenging learning experience for you. C++ is a difficult language that is normally taught only to experienced programmers, so this book is unique among C++ textbooks:

- It is appropriate for technically oriented people with little or no programming experience.
- It is appropriate for experienced programmers who want a deeper treatment of the language.

How can one book appeal to both groups? The answer is that the common core of the book emphasizes achieving program *clarity* through the proven techniques of *structured programming* and *object-oriented programming*. Non-programmers learn programming the right way from the beginning. We have attempted to write in a clear and straightforward manner. The book is abundantly illustrated. Perhaps most importantly, the book presents hundreds of complete working C++ programs and shows the outputs produced when those programs are run on a computer. We call this the "live-code approach." All of these example programs are provided on the CD-ROM that accompanies this book. You may also download all these examples from our Web site **`www.deitel.com`**. The examples are also available on our interactive CD-ROM product, the *C++ Multimedia Cyber Classroom: Third Edition*. The Cyber Classroom also contains extensive hyperlinking, audio walkthroughs of the program examples in the book and answers to approximately half the exercises in this book (including short answers, small programs and many full projects). The Cyber Classroom's features and ordering information appear at the back of this book.

The first five chapters introduce the fundamentals of computers, computer programming and the C++ computer programming language. Novices who have taken our courses tell us that the material in Chapters 1 through 5 presents a solid foundation for the deeper treatment of C++ in the remaining chapters. Experienced programmers typically read the first five chapters quickly then find the treatment of C++ in the remainder of the book both rigorous and challenging.

Many experienced programmers have told us that they appreciate our treatment of structured programming. Often they have been programming in structured languages like C or Pascal, but because they were never formally introduced to structured programming, they are not writing the best possible code in these languages. As they review structured programming in the early chapters of this book, they are able to improve their C and Pascal programming styles as well. So whether you are a novice or an experienced programmer, there is much here to inform, entertain and challenge you.

Most people are at least somewhat familiar with the exciting things computers do. Using this textbook, you will learn how to command computers to do those things. It is *software* (i.e., the instructions you write to command the computer to perform *actions* and make *decisions*) that controls computers (often referred to as *hardware*). C++ is one of today's most popular software development languages. This text provides an introduction to programming in the version of C++ standardized in the United States through the *American National Standards Institute (ANSI)* and worldwide through the efforts of the *International Standards Organization (ISO).*

The use of computers is increasing in almost every field of endeavor. In an era of steadily rising costs, computing costs have been decreasing dramatically because of the rapid developments in both hardware and software technology. Computers that filled large rooms and cost millions of dollars 25 to 30 years ago are now inscribed on the surfaces of silicon chips smaller than a fingernail and that cost perhaps a few dollars each. Ironically, silicon is one of the most abundant materials on the earth—it is an ingredient in common sand. Silicon-chip technology has made computing so economical that hundreds of millions

of general-purpose computers are in use worldwide helping people in business, industry, government and their personal lives. That number could easily double in a few years.

This book will challenge you for several reasons. Your peers over the last few years probably learned C or Pascal as their first programming language. You will actually learn both C and C++! Why? Simply because C++ includes C and adds much more.

Your peers probably learned the programming methodology called *structured programming*. You will learn both structured programming and the exciting newer methodology, *object-oriented programming*. Why do we teach both? Object-orientation is certain to be the key programming methodology for the next decade. You will create and work with many *objects* in this course. But you will discover that the internal structure of those objects is often best built using structured programming techniques. Also, the logic of manipulating objects is occasionally best expressed with structured programming.

Another reason we present both methodologies is that there currently is a massive migration occurring from C-based systems to C++-based systems. There is a huge amount of so-called "legacy C code" in place. C has been in wide use for about a quarter of a century and its use in recent years has been increasing dramatically. Once people learn C++, they find it more powerful than C and often choose to move to C++. They begin converting their legacy systems to C++. They begin using the various C++ features generally called "C++ enhancements to C" to improve their style of writing C-like programs. Finally, they begin employing the object-oriented programming capabilities of C++ to realize the full benefits of the language.

An interesting phenomenon in programming languages is that most of the vendors simply market a combined C/C++ product rather than offering separate products. This gives users the ability to continue programming in C if they wish then gradually migrate to C++ when appropriate.

C++ has become the implementation language of choice for building high-performance computing systems. But can it be taught in a first programming course, the intended audience for this book? We think so. Nine years ago we took on a similar challenge when Pascal was the entrenched language in first computer science courses. We wrote *C How to Program*. Hundreds of universities worldwide now use the third edition of *C How to Program.* Courses based on that book have proven to be equally effective to their Pascal-based predecessors. No significant differences have been observed, except that students are better motivated because they know they are more likely to use C rather than Pascal in their upper-level courses and in their careers. Students learning C also know that they will be better prepared to learn C++ and the new Internet-ready, C++-based language called *Java*.

In the first five chapters of the book you will learn structured programming in C++, the "C portion" of C++ and the "C++ enhancements to C." In the balance of the book you will learn object-oriented programming in C++. We do not want you to wait until Chapter 6, however, to begin appreciating object-orientation. Therefore, each of the first five chapters concludes with a section entitled "Thinking About Objects." These sections introduce basic concepts and terminology about object-oriented programming. When we reach Chapter 6, Classes and Data Abstraction, you will be prepared to start using C++ to create objects and write object-oriented programs.

This first chapter has three parts. The first part introduces the basics of computers and computer programming. The second part gets you started immediately writing some simple C++ programs. The third part helps you start "thinking about objects."

So there you have it! You are about to start on a challenging and rewarding path. As you proceed, if you would like to communicate with us, please send us email at

**deitel@deitel.com**

or browse our World Wide Web site at

**http://www.deitel.com/**

We will respond immediately. We hope you enjoy learning with *C++ How to Program*. You may want to consider using the interactive CD-ROM version of the book called the *C++ Multimedia Cyber Classroom: Third Edition*. Please see the ordering instructions at the back of this book.

## 1.2 What is a Computer?

A *computer* is a device capable of performing computations and making logical decisions at speeds millions and even billions of times faster than human beings can. For example, many of today's personal computers can perform hundreds of millions of additions per second. A person operating a desk calculator might require decades to complete the same number of calculations a powerful personal computer can perform in one second. (Points to ponder: How would you know whether the person added the numbers correctly? How would you know whether the computer added the numbers correctly?) Today's fastest *supercomputers* can perform hundreds of billions of additions per second—about as many calculations as hundreds of thousands of people could perform in one year! And trillion-instruction-per-second computers are already functioning in research laboratories!

Computers process *data* under the control of sets of instructions called *computer programs.* These computer programs guide the computer through orderly sets of actions specified by people called *computer programmers.*

A computer is comprised of various devices (such as the keyboard, screen, "mouse," disks, memory, CD-ROM and processing units) that are referred to as *hardware.* The computer programs that run on a computer are referred to as *software.* Hardware costs have been declining dramatically in recent years, to the point that personal computers have become a commodity. Unfortunately, software development costs have been rising steadily as programmers develop ever more powerful and complex applications, without significantly improved technology for software development. In this book you will learn proven software development methods that can reduce software development costs—structured programming, top-down stepwise refinement, functionalization, object-based programming, object-oriented programming, object-oriented design and generic programming.

## 1.3 Computer Organization

Regardless of differences in physical appearance, virtually every computer may be envisioned as being divided into six *logical units* or sections. These are:

1. *Input unit.* This is the "receiving" section of the computer. It obtains information (data and computer programs) from various *input devices* and places this information at the disposal of the other units so that the information may be processed. Most information is entered into computers today through keyboards and mouse

devices. Information can also be entered by speaking to your computer and by scanning images.

2. *Output unit.* This is the "shipping" section of the computer. It takes information that has been processed by the computer and places it on various *output devices* to make the information available for use outside the computer. Most information output from computers today is displayed on screens, printed on paper, or used to control other devices.
3. *Memory unit.* This is the rapid access, relatively low-capacity "warehouse" section of the computer. It retains information that has been entered through the input unit so that the information may be made immediately available for processing when it is needed. The memory unit also retains processed information until that information can be placed on output devices by the output unit. The memory unit is often called either *memory* or *primary memory.*
4. *Arithmetic and logic unit (ALU).* This is the "manufacturing" section of the computer. It is responsible for performing calculations such as addition, subtraction, multiplication and division. It contains the decision mechanisms that allow the computer, for example, to compare two items from the memory unit to determine whether or not they are equal.
5. *Central processing unit (CPU).* This is the "administrative" section of the computer. It is the computer's coordinator and is responsible for supervising the operation of the other sections. The CPU tells the input unit when information should be read into the memory unit, tells the ALU when information from the memory unit should be used in calculations and tells the output unit when to send information from the memory unit to certain output devices.
6. *Secondary storage unit.* This is the long-term, high-capacity "warehousing" section of the computer. Programs or data not actively being used by the other units are normally placed on secondary storage devices (such as disks) until they are again needed, possibly hours, days, months, or even years later. Information in secondary storage takes much longer to access than information in primary memory. The cost per unit of secondary storage is much less than the cost per unit of primary memory.

## 1.4 Evolution of Operating Systems

Early computers were capable of performing only one *job* or *task* at a time. This form of computer operation is often called single-user *batch processing.* The computer runs a single program at a time while processing data in groups or *batches.* In these early systems, users generally submitted their jobs to a computer center on decks of punched cards. Users often had to wait hours or even days before printouts were returned to their desks.

Software systems called *operating systems* were developed to help make it more convenient to use computers. Early operating systems managed the smooth transition between jobs. This minimized the time it took for computer operators to switch between jobs and hence increased the amount of work, or *throughput*, computers could process.

As computers became more powerful, it became evident that single-user batch processing rarely utilized the computer's resources efficiently because most of the time was

spent waiting for slow input/output devices to complete their tasks. Instead, it was thought that many jobs or tasks could be made to *share* the resources of the computer to achieve better utilization. This is called *multiprogramming*. Multiprogramming involves the "simultaneous" operation of many jobs on the computer—the computer shares its resources among the jobs competing for its attention. With early multiprogramming operating systems, users still submitted jobs on decks of punched cards and waited hours or days for results.

In the 1960s, several groups in industry and the universities pioneered *timesharing* operating systems. Timesharing is a special case of multiprogramming in which users access the computer through *terminals*, typically devices with keyboards and screens. In a typical timesharing computer system, there may be dozens or even hundreds of users sharing the computer at once. The computer does not actually run all the users simultaneously. Rather, it runs a small portion of one user's job then moves on to service the next user. The computer does this so quickly that it may provide service to each user several times per second. Thus the users' programs *appear* to be running simultaneously. An advantage of timesharing is that the user receives almost immediate responses to requests rather than having to wait long periods for results as with previous modes of computing.

## 1.5 Personal Computing, Distributed Computing and Client/Server Computing

In 1977, Apple Computer popularized the phenomenon of *personal computing*. Initially, it was a hobbyist's dream. Computers became economical enough for people to buy them for their own personal or business use. In 1981, IBM, the world's largest computer vendor, introduced the IBM Personal Computer. Literally overnight, personal computing became legitimate in business, industry and government organizations.

But these computers were "standalone" units—people did their work on their own machines then transported disks back and forth to share information (this is often called "sneakernet"). Although early personal computers were not powerful enough to timeshare several users, these machines could be linked together in computer networks, sometimes over telephone lines and sometimes in *local area networks (LANs)* within an organization. This led to the phenomenon of *distributed computing* in which an organization's computing, instead of being performed strictly at some central computer installation, is distributed over networks to the sites at which the work of the organization is performed. Personal computers were powerful enough to handle the computing requirements of individual users, and to handle the basic communications tasks of passing information back and forth electronically.

Today's most powerful personal computers are as powerful as the million dollar machines of just a decade ago. The most powerful desktop machines—called *workstations*—provide individual users with enormous capabilities. Information is easily shared across computer networks where some computers called *file servers* offer a common store of programs and data that may be used by *client* computers distributed throughout the network, hence the term *client/server computing*. C and C++ have become the programming languages of choice for writing software for operating systems, for computer networking and for distributed client/server applications. Today's popular operating systems such as UNIX, Linux and Microsoft's Windows-based systems provide the kinds of capabilities discussed in this section.

## 1.6 Machine Languages, Assembly Languages, and High-level Languages

Programmers write instructions in various programming languages, some directly understandable by the computer and others that require intermediate *translation* steps. Hundreds of computer languages are in use today. These may be divided into three general types:

1. Machine languages,
2. Assembly languages,
3. High-level languages.

Any computer can directly understand only its own *machine language.* Machine language is the "natural language" of a particular computer. It is defined by the hardware design of that computer. Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time. Machine languages are *machine-dependent,* i.e., a particular machine language can be used on only one type of computer. Machine languages are cumbersome for humans, as can be seen by the following section of a machine language program that adds overtime pay to base pay and stores the result in gross pay.

```
+1300042774
+1400593419
+1200274027
```

As computers became more popular, it became apparent that machine language programming was too slow, tedious and error prone. Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent the elementary operations of the computer. These English-like abbreviations formed the basis of *assembly languages. Translator programs* called *assemblers* were developed to convert assembly language programs to machine language at computer speeds. The following section of an assembly language program also adds overtime pay to base pay and stores the result in gross pay, but more clearly than its machine language equivalent:

```
LOAD    BASEPAY
ADD     OVERPAY
STORE   GROSSPAY
```

Although such code is clearer to humans, it is incomprehensible to computers until translated to machine language.

Computer usage increased rapidly with the advent of assembly languages, but these still required many instructions to accomplish even the simplest tasks. To speed the programming process, *high-level languages* were developed in which single statements accomplish substantial tasks. Translator programs called *compilers* convert high-level language programs into machine language. High-level languages allow programmers to write instructions that look almost like everyday English and contain commonly used mathematical notations. A payroll program written in a high-level language might contain a statement such as:

```
grossPay = basePay + overTimePay
```

Obviously, high-level languages are much more desirable from the programmer's standpoint than either machine languages or assembly languages. C and C++ are among the most powerful and most widely used high-level languages.

The process of compiling a high-level language program into machine language can take a considerable amount of computer time. *Interpreter* programs were developed that can directly execute high-level language programs without the need for compiling those programs into machine language. Although compiled programs execute faster than interpreted programs, interpreters are popular in program development environments in which programs are changed frequently as new features are added and errors are corrected. Once a program is developed, a compiled version can be produced to run most efficiently.

## 1.7 History of C and C++

C++ evolved from C, which evolved from two previous programming languages, BCPL and B. BCPL was developed in 1967 by Martin Richards as a language for writing operating systems software and compilers. Ken Thompson modeled many features in his language B after their counterparts in BCPL and used B to create early versions of the UNIX operating system at Bell Laboratories in 1970 on a DEC PDP-7 computer. Both BCPL and B were "typeless" languages—every data item occupied one "word" in memory and the burden of treating a data item as a whole number or a real number, for example, was the responsibility of the programmer.

The C language was evolved from B by Dennis Ritchie at Bell Laboratories and was originally implemented on a DEC PDP-11 computer in 1972. C uses many important concepts of BCPL and B while adding data typing and other features. C initially became widely known as the development language of the UNIX operating system. Today, most operating systems are written in C and/or C++. C is now available for most computers. C is hardware independent. With careful design, it is possible to write C programs that are *portable* to most computers.

By the late 1970s, C had evolved into what is now referred to as "traditional C," "classic C," or "Kernighan and Ritchie C." The publication by Prentice-Hall in 1978 of Kernighan and Ritchie's book, *The C Programming Language,* brought wide attention to the language.

The widespread use of C with various types of computers (sometimes called *hardware platforms*) unfortunately led to many variations. These were similar, but often incompatible. This was a serious problem for program developers who needed to write portable programs that would run on several platforms. It became clear that a standard version of C was needed. In 1983, the X3J11 technical committee was created under the American National Standards Committee on Computers and Information Processing (X3) to "provide an unambiguous and machine-independent definition of the language." In 1989, the standard was approved. ANSI cooperated with the International Standards Organization (ISO) to standardize C worldwide; the joint standard document was published in 1990 and is referred to as *ANSI/ISO 9899: 1990*. Copies of this document may be ordered from ANSI. The second edition of Kernighan and Ritchie, published in 1988, reflects this version called ANSI C, a version of the language now used worldwide.

**Portability Tip 1.1**

*Because C is a standardized, hardware-independent, widely available language, applications written in C can often be run with little or no modifications on a wide range of different computer systems.*

C++, an extension of C, was developed by Bjarne Stroustrup in the early 1980s at Bell Laboratories. C++ provides a number of features that "spruce up" the C language, but more importantly, it provides capabilities for *object-oriented programming*.

There is a revolution brewing in the software community. Building software quickly, correctly and economically remains an elusive goal, and this at a time when the demand for new and more powerful software is soaring. *Objects* are essentially reusable software *components* that model items in the real world. Software developers are discovering that using a modular, object-oriented design and implementation approach can make software development groups much more productive than is possible with previous popular programming techniques such as structured programming. Object-oriented programs are easier to understand, correct and modify.

Many other object-oriented languages have been developed, including Smalltalk, developed at Xerox's Palo Alto Research Center (PARC). Smalltalk is a pure object-oriented language—literally everything is an object. C++ is a hybrid language—it is possible to program in C++ in either a C-like style, an object-oriented style, or both. In Section 1.9 we discuss the exciting new C and C++-based language, Java.

## 1.8 C++ Standard Library

C++ programs consist of pieces called *classes* and *functions*. You can program each piece you may need to form a C++ program. But most C++ programmers take advantage of the rich collections of existing classes and functions in the C++ standard library. Thus, there are really two parts to learning the C++ "world." The first is learning the C++ language itself and the second is learning how to use the classes and functions in the C++ standard library. Throughout the book, we discuss many of these classes and functions. The book by Plauger is must reading for programmers who need a deep understanding of the ANSI C library functions that are included in C++, how to implement them and how to use them to write portable code. The standard class libraries are generally provided by compiler vendors. Many special-purpose class libraries are supplied by independent software vendors.

### Software Engineering Observation 1.1

*Use a "building block approach" to creating programs. Avoid reinventing the wheel. Use existing pieces where possible—this is called "software reuse" and it is central to object-oriented programming.*

### Software Engineering Observation 1.2

*When programming in C++ you will typically use the following building blocks: classes and functions from the C++ standard library, classes and functions you create yourself, and classes and functions from various popular third-party libraries.*

The advantage of creating your own functions and classes is that you will know exactly how they work. You will be able to examine the C++ code. The disadvantage is the time-consuming and complex effort that goes into designing, developing and maintaining new functions and classes that are correct and that operate efficiently.

### Performance Tip 1.1

*Using standard library functions and classes instead of writing your own comparable versions can improve program performance because this software is carefully written to perform efficiently and correctly.*

**Portability Tip 1.2**

*Using standard library functions and classes instead of writing your own comparable versions can improve program portability because this software is included in virtually all C++ implementations.*

## 1.9 Java and *Java How to Program*

Many people believe that the next major area in which microprocessors will have a profound impact is in intelligent consumer electronic devices. Recognizing this, Sun Microsystems funded an internal corporate research project code-named Green in 1991. The project resulted in the development of a C and C++ based language which its creator, James Gosling, called Oak after an oak tree outside his window at Sun. It was later discovered that there already was a computer language called Oak. When a group of Sun people visited a local coffee place, the name *Java* was suggested and it stuck.

But the Green project ran into some difficulties. The marketplace for intelligent consumer electronic devices was not developing as quickly as Sun had anticipated. Worse yet, a major contract for which Sun competed was awarded to another company. So the project was in danger of being canceled. By sheer good fortune, the World Wide Web exploded in popularity in 1993 and Sun people saw the immediate potential of using Java to create Web pages with so-called *dynamic content*.

Sun formally announced Java at a trade show in May 1995. Ordinarily, an event like this would not have generated much attention. However, Java generated immediate interest in the business community because of the phenomenal interest in the World Wide Web. Java is now used to create Web pages with dynamic and interactive content, to develop large-scale enterprise applications, to enhance the functionality of Web servers (the computers that provide the content we see in our Web browsers), to provide applications for consumer devices (such as cell phones, pagers and personal digital assistants), and more.

In 1995, we were carefully following the development of Java by Sun Microsystems. In November 1995 we attended an Internet conference in Boston. A representative from Sun Microsystems gave a rousing presentation on Java. As the talk proceeded, it became clear to us that Java would play a significant part in the development of interactive, multimedia Web pages. But we immediately saw a much greater potential for the language.

We saw Java as a nice language for teaching first-year programming language students the essentials of graphics, images, animation, audio, video, database, networking, multithreading and collaborative computing. We went to work on the first edition of *Java How to Program* which was published in time for fall 1996 classes. *Java How to Program: Third Edition* was published in 1999.

In addition to its prominence in developing Internet- and intranet-based applications, Java is certain to become the language of choice for implementing software for devices that communicate over a network (such as cellular phones, pagers and personal digital assistants). Do not be surprised when your new stereo and other devices in your home will be networked together using Java technology!

## 1.10 Other High-level Languages

Hundreds of high-level languages have been developed, but only a few have achieved broad acceptance. *FORTRAN* (FORmula TRANslator) was developed by IBM Corporation

between 1954 and 1957 to be used for scientific and engineering applications that require complex mathematical computations. FORTRAN is still widely used, especially in engineering applications.

*COBOL* (COmmon Business Oriented Language) was developed in 1959 by computer manufacturers, government and industrial computer users. COBOL is used primarily for commercial applications that require precise and efficient manipulation of large amounts of data. Today, more than half of all business software is still programmed in COBOL.

*Pascal* was designed at about the same time as C by Professor Niklaus Wirth and was intended for academic use. We will say more about Pascal in the next section.

## 1.11 Structured Programming

During the 1960s, many large software development efforts encountered severe difficulties. Software schedules were typically late, costs greatly exceeded budgets and the finished products were unreliable. People began to realize that software development was a far more complex activity than they had imagined. Research activity in the 1960s resulted in the evolution of *structured programming*—a disciplined approach to writing programs that are clearer than unstructured programs, easier to test and debug and easier to modify. Chapter 2 discusses the principles of structured programming. Chapters 3 through 5 develop many structured programs.

One of the more tangible results of this research was the development of the Pascal programming language by Niklaus Wirth in 1971. Pascal, named after the seventeenth-century mathematician and philosopher Blaise Pascal, was designed for teaching structured programming in academic environments and rapidly became the preferred programming language in most universities. Unfortunately, the language lacks many features needed to make it useful in commercial, industrial and government applications, so it has not been widely accepted outside the universities.

The Ada programming language was developed under the sponsorship of the United States Department of Defense (DOD) during the 1970s and early 1980s. Hundreds of separate languages were being used to produce DOD's massive command-and-control software systems. DOD wanted a single language that would fulfill most of its needs. Pascal was chosen as a base, but the final Ada language is quite different from Pascal. The language was named after Lady Ada Lovelace, daughter of the poet Lord Byron. Lady Lovelace is generally credited with writing the world's first computer program in the early 1800s (for the Analytical Engine mechanical computing device designed by Charles Babbage). One important capability of Ada is called *multitasking;* this allows programmers to specify that many activities are to occur in parallel. The other widely used high-level languages we have discussed—including C and C++—generally allow the programmer to write programs that perform only one activity at a time.

## 1.12 The Key Software Trend: Object Technology

One of the authors, HMD, remembers the great frustration that was felt in the 1960s by software development organizations, especially those developing large-scale projects. During his undergraduate years, HMD had the privilege of working summers at a leading computer vendor on the teams developing time-sharing, virtual memory operating systems. This was a great experience for a college student. But in the summer of 1967 reality set in when the

company "decommitted" from producing as a commercial product the particular system that hundreds of people had been working on for many years. It was difficult to get this software right. Software is "complex stuff."

Hardware costs have been declining dramatically in recent years, to the point that personal computers have become a commodity. Unfortunately, software development costs have been rising steadily as programmers develop ever more powerful and complex applications, without being able to improve significantly the underlying technologies of software development. In this book you will learn many software development methods that can reduce software development costs.

There is a revolution brewing in the software community. Building software quickly, correctly and economically remains an elusive goal, and this at a time when demands for new and more powerful software are soaring. *Objects* are essentially reusable software *components* that model items in the real world. Software developers are discovering that using a modular, object-oriented design and implementation approach can make software development groups much more productive than is possible with previous popular programming techniques such as structured programming. Object-oriented programs are often easier to understand, correct and modify.

Improvements to software technology did start to appear with the benefits of so-called *structured programming* (and the related disciplines of *structured systems analysis and design)* being realized in the 1970s. But it was not until the technology of object-oriented programming became widely used in the 1980s, and especially widely used in the 1990s, that software developers finally felt they had the tools they needed to make major strides in the software development process.

Actually, object technology dates back at least to the mid 1960s. The C++ programming language developed at AT&T by Bjarne Stroustrup in the early 1980s, is based on two languages—C, which was initially developed at AT&T to implement the UNIX operating system in the early 1970s, and Simula 67, a simulation programming language developed in Europe and released in 1967. C++ absorbed the capabilities of C and added Simula's capabilities for creating and manipulating objects. Neither C nor C++ was intended for wide use beyond the AT&T research laboratories. But grass-roots support rapidly developed for each.

What are objects and why are they special. Actually, object technology is a packaging scheme that helps us create meaningful software units. These are large and highly focussed on particular applications areas. There are date objects, time objects, paycheck objects, invoice objects, audio objects, video objects, file objects, record objects and so on. In fact, any noun can be represented as an object.

We live in a world of objects. Just look around you. There are cars, planes, people, animals, buildings, traffic lights, elevators, and so on. Before object-oriented languages appeared, programming languages (such as FORTRAN, Pascal, Basic and C) were focussed on actions (verbs) rather than things or objects (nouns). Programmers living in a world of objects would get to the computer and have to program primarily with verbs. This paradigm shift made it a bit awkward to write programs. Now, with the availability of popular object-oriented languages such as Java and C++ and many others, programmers continue to live in an object-oriented world and when they get to the computer they can program in an object-oriented manner. This means they program in a manner similar to the way in which they perceive the world. This is a more natural process than procedural programming and has resulted in significant productivity enhancements.

One of the key problems with procedural programming is that the program units programmers created do not easily mirror real-world entities effectively. So they are not particularly reusable. It is not unusual for programmers to "start fresh" on each new project and wind up writing very similar software "from scratch." This wastes precious time and money resources as people repeatedly "reinvent the wheel." With object technology, the software entities created (called *objects*), if properly designed, tend to be much more reusable on future projects. Using libraries of reusable componentry such as *MFC (Microsoft Foundation Classes)* and those produced by Rogue Wave and many other software development organizations can greatly reduce the amount of effort it takes to implement certain kinds of systems (compared to the effort that would be required to reinvent these capabilities on new projects).

Some organizations report that software reuse is not, in fact, the key benefit they get from object-oriented programming. Rather, they indicate that object-oriented programming tends to produce software that is more understandable, better organized and easier to maintain, modify and debug. This can be significant because it has been estimated that as much as 80% of software costs are not associated with the original efforts to develop the software, but are associated with the continued evolution and maintenance of that software throughout its lifetime.

Whatever the perceived benefits of object-orientation are, it is clear that object-oriented programming will be the key programming methodology for the next several decades.

[*Note*: We will include many of these *Software Engineering Observations* throughout the text to explain concepts that affect and improve the overall architecture and quality of a software system, and particularly, of large software systems. We will also highlight *Good Programming Practices* (practices that can help you write programs that are clearer, more understandable, more maintainable, and easier to test and debug), *Common Programming Errors* (problems to watch out for so you do not make these same errors in your programs), *Performance Tips* (techniques that will help you write programs that run faster and use less memory), *Portability Tips* (techniques that will help you write programs that can run, with little or no modification, on a variety of computers) and *Testing and Debugging Tips* (techniques that will help you remove bugs from your programs, and more important, techniques that will help you write bug-free programs in the first place). Many of these techniques and practices are only guidelines; you will, no doubt, develop your own preferred programming style.]

The advantage of creating your own code is that you will know exactly how it works. You will be able to examine the code. The disadvantage is the time-consuming and complex effort that goes into designing and developing new code.

### Performance Tip 1.2

*Reusing proven code components instead of writing your own versions can improve program performance because these components are normally written to perform efficiently.*

### Software Engineering Observation 1.3

*Extensive class libraries of reusable software components are available over the Internet and the World Wide Web. Many of these libraries are available at no charge.*

## 1.13 Basics of a Typical C++ Environment

C++ systems generally consist of several parts: a program development environment, the language and the C++ Standard Library. The following discussion explains a typical C++ program development environment shown in Fig. 1.1.

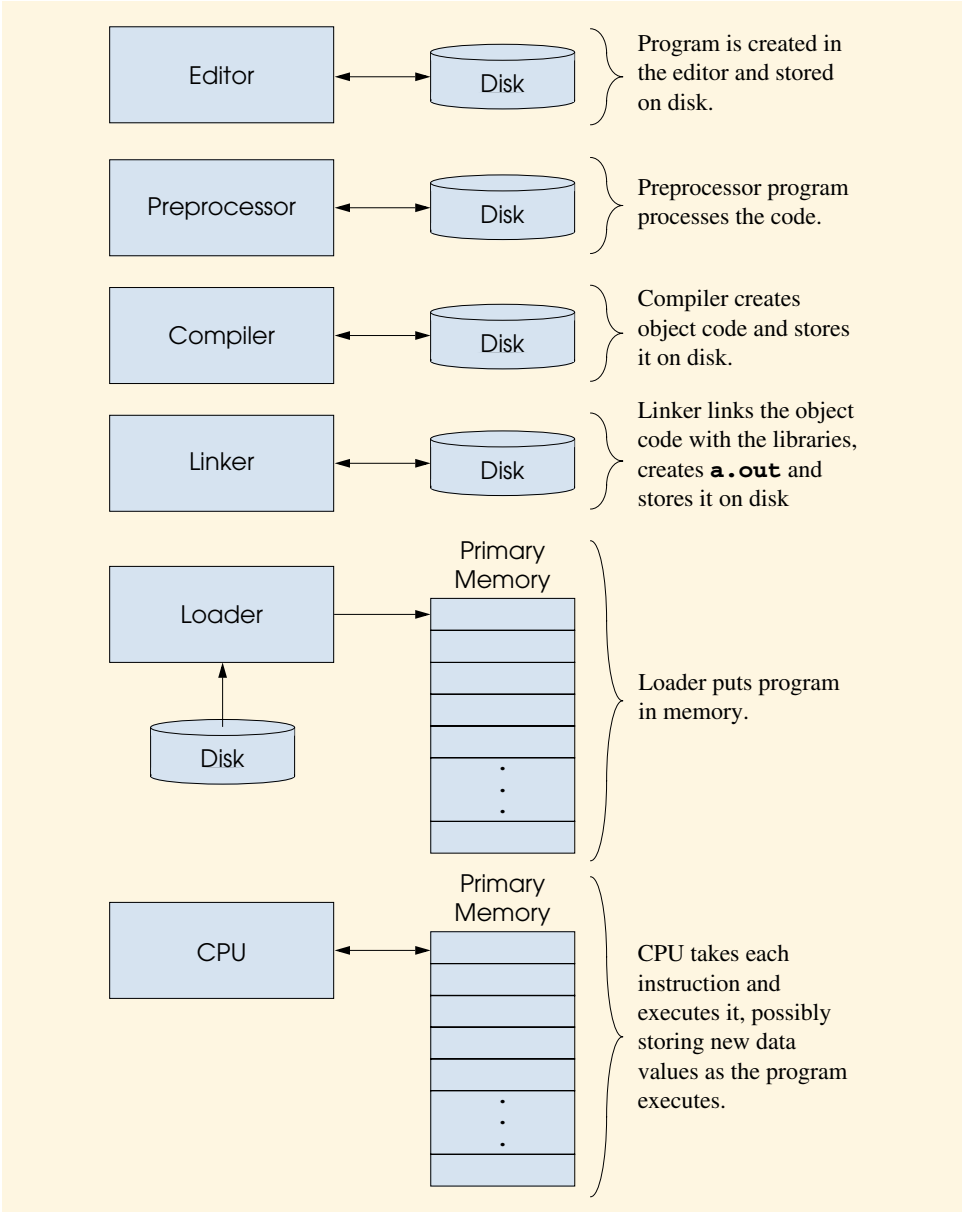



**Fig. 1.1** A typical C++ environment.

C++ programs typically go through six phases to be executed (Fig. 1.1). These are: *edit, preprocess, compile, link, load and execute.* We concentrate on a typical UNIX-based C++ system here (Note: the programs in this book will run with little or no modification on most current C++ systems, including Microsoft Windows-based systems). If you are not using a UNIX system, refer to the manuals for your system, or ask your instructor how to accomplish these tasks in your environment.

The first phase consists of editing a file. This is accomplished with an *editor program.* The programmer types a C++ program with the editor and makes corrections if necessary. The program is then stored on a secondary storage device such as a disk. C++ program file names often end with the **.cpp**, **.cxx** or **.C** extensions (note that **C** is in uppercase). See the documentation for your C++ environment for more information on file-name extensions. Two editors widely used on UNIX systems are **vi** and **emacs**. C++ software packages such as Borland C++ and Microsoft Visual C++ for personal computers have built-in editors that are smoothly integrated into the programming environment. We assume the reader knows how to edit a program.

Next, the programmer gives the command to *compile* the program. The compiler translates the C++ program into machine language code (also referred to as *object code*). In a C++ system, a *preprocessor* program executes automatically before the compiler's translation phase begins. The C++ preprocessor obeys special commands called *preprocessor directives* which indicate that certain manipulations are to be performed on the program before compilation. These manipulations usually consist of including other text files in the file to be compiled and performing various text replacements. The most common preprocessor directives are discussed in the early chapters; a detailed discussion of all the preprocessor features appears in the chapter entitled, "The Preprocessor." The preprocessor is invoked by the compiler before the program is converted to machine language.

The next phase is called *linking.* C++ programs typically contain references to functions defined elsewhere, such as in the standard libraries or in the private libraries of groups of programmers working on a particular project. The object code produced by the C++ compiler typically contains "holes" due to these missing parts. A *linker* links the object code with the code for the missing functions to produce an *executable image* (with no missing pieces). On a typical UNIX-based system, the command to compile and link a C++ program is ***CC***. To compile and link a program named **welcome.C** type

```
CC welcome.C
```

at the UNIX prompt and press the *Enter* key (or *Return* key). If the program compiles and links correctly, a file called **a.out** is produced. This is the executable image of our **welcome.C** program.

The next phase is called *loading.* Before a program can be executed, the program must first be placed in memory. This is done by the *loader,* which takes the executable image from disk and transfers it to memory. Additional components from shared libraries that support the program are also loaded.

Finally, the computer, under the control of its CPU, executes the program one instruction at a time. To load and execute the program on a UNIX system, we type **a.out** at the UNIX prompt and press *return*.

Programs do not always work on the first try. Each of the preceding phases can fail because of various errors that we will discuss. For example, an executing program might

attempt to divide by zero (an illegal operation on computers just as it is in arithmetic). This would cause the computer to print an error message. The programmer would then return to the edit phase, make the necessary corrections and proceed through the remaining phases again to determine that the corrections work properly.

**Common Programming Error 1.1**

*Errors like division-by-zero errors occur as a program runs, so these errors are called run-time errors or execution-time errors. Divide-by-zero is generally a fatal error, i.e., an error that causes the program to terminate immediately without having successfully performed its job. Non-fatal errors allow programs to run to completion, often producing incorrect results. (Note: On some systems, divide-by-zero is not a fatal error. Please see your system documentation.)*

Most programs in C++ input and/or output data. Certain C++ functions take their input from ***cin*** (the *standard input stream*; pronounced "see-in") which is normally the keyboard, but **cin** can be connected to another device. Data is often output to ***cout*** (the *standard output stream*; pronounced "see-out") which is normally the computer screen, but **cout** can be connected to another device. When we say that a program prints a result, we normally mean that the result is displayed on a screen. Data may be output to other devices such as disks and hardcopy printers. There is also a *standard error stream* referred to as ***cerr***. The **cerr** stream (normally connected to the screen) is used for displaying error messages. It is common for users to route regular output data, i.e., **cout**, to a device other than the screen while keeping **cerr** assigned to the screen so the user can be immediately informed of errors.

## 1.14 Hardware Trends

The programming community thrives on the continuing stream of dramatic improvements in hardware, software and communications technologies. Every year, people generally expect to pay at least a little more for most products and services. The opposite has been the case in the computer and communications fields, especially with regard to the hardware costs of supporting these technologies. For many decades, and with no change in the foreseeable future, hardware costs have fallen rapidly, if not precipitously. This is a phenomenon of technology, another driving force powering the current economic boom. Every year or two, the capacities of computers, especially the amount of memory they have in which to execute programs, the amount of secondary storage (such as disk storage) they have to hold programs and data over the longer term, and their processor speeds—the speed at which computers execute their programs (i.e., do their work)—each tend to approximately double. The same has been true in the communications field with costs plummeting, especially in recent years with the enormous demand for communications bandwidth attracting tremendous competition. We know of no other fields in which technology moves so quickly and costs fall so rapidly.

When computer use exploded in the sixties and seventies, there was talk of huge improvements in human productivity that computing and communications would bring about. But these improvements did not materialize. Organizations were spending vast sums on computers and certainly employing them effectively, but without realizing the productivity gains that had been expected. It was the invention of microprocessor chip technology and its wide deployment in the late 1970s and 1980s that laid the groundwork for the productivity improvements of the 1990s that have been so crucial to economic prosperity.

## 1.15 History of the Internet

In the late 1960s, one of the authors (HMD) was a graduate student at MIT. His research at MIT's Project Mac (now the Laboratory for Computer Science—the home of the World Wide Web Consortium) was funded by ARPA—the Advanced Research Projects Agency of the Department of Defense. ARPA sponsored a conference at which several dozen ARPA-funded graduate students were brought together at the University of Illinois at Urbana-Champaign to meet and share ideas. During this conference, ARPA rolled out the blueprints for networking the main computer systems of about a dozen ARPA-funded universities and research institutions. They were to be connected with communications lines operating at a then-stunning 56KB (i.e., 56,000 bits per second), this at a time when most people (of the few who could be) were connecting over telephone lines to computers at a rate of 110 bits per second. HMD vividly recalls the excitement at that conference. Researchers at Harvard talked about communication with the Univac 1108 "supercomputer" across the country at the University of Utah to handle calculations related to their computer graphics research. Many other intriguing possibilities were raised. Academic research was about to take a giant leap forward. Shortly after this conference, ARPA proceeded to implement what quickly became called the *ARPAnet*, the grandparent of today's *Internet*.

Things worked out differently from what was originally planned. Rather than the primary benefit being that researchers could share each other's computers, it rapidly became clear that simply enabling the researchers to communicate quickly and easily among themselves via what became known as *electronic mail* (*e-mail*, for short) was to be the key benefit of the ARPAnet. This is true even today on the Internet with e-mail facilitating communications of all kinds among millions of people worldwide.

One of ARPA's primary goals for the network was to allow multiple users to send and receive information at the same time over the same communications paths (such as phone lines). The network operated with a technique called *packet switching* in which digital data was sent in small packages called *packets*. The packets contained data, address information, error-control information and sequencing information. The address information was used to route the packets of data to their destination. The sequencing information was used to help reassemble the packets (which—because of complex routing mechanisms—could actually arrive out of order) into their original order for presentation to the recipient. Packets of many people were intermixed on the same lines. This packet-switching technique greatly reduced transmission costs compared to the cost of dedicated communications lines.

The network was designed to operate without centralized control. This meant that if a portion of the network should fail, the remaining working portions would still be able to route packets from senders to receivers over alternate paths.

The protocols for communicating over the ARPAnet became known as *TCP—the Transmission Control Protocol*. TCP ensured that messages were properly routed from sender to receiver and that those messages arrived intact.

In parallel with the early evolution of the Internet, organizations worldwide were implementing their own networks for both intra-organization (i.e., within the organization) and inter-organization (i.e., between organizations) communication. A huge variety of networking hardware and software appeared. One challenge was to get these to intercommunicate. ARPA accomplished this with the development of *IP*—the *Internetworking Protocol*), truly creating a "network of networks," the current architecture of the Internet. The combined set of protocols is now commonly called *TCP/IP*.

Initially, use of the Internet was limited to universities and research institutions; then the military became a big user. Eventually, the government decided to allow access to the Internet for commercial purposes. Initially there was resentment among the research and military communities—it was felt that response times would become poor as "the net" became saturated with so many users.

In fact, the exact opposite has occurred. Businesses rapidly realized that by making effective use of the Internet they could tune their operations and offer new and better services to their clients, so they started spending vasts amounts of money to develop and enhance the Internet. This generated fierce competition among the communications carriers and hardware and software suppliers to meet this demand. The result is that *bandwidth* (i.e., the information carrying capacity of communications lines) on the Internet has increased tremendously and costs have plummeted. It is widely believed that the Internet has played a significant role in the economic prosperity that the United States and many other industrialized nations have enjoyed over the last decade and are likely to continue enjoying for many years.

## 1.16 History of the World Wide Web

The *World Wide Web* allows computer users to locate and view multimedia-based documents (i.e., documents with text, graphics, animations, audios and/or videos) on almost any subject. Even though the Internet was developed more than three decades ago, the introduction of the *World Wide Web* was a relatively recent event. In 1990, *Tim Berners-Lee* of CERN (the European Laboratory for Particle Physics) developed the World Wide Web and several communication protocols that form its backbone.

The Internet and the World Wide Web will surely be listed among the most important and profound creations of humankind. In the past, most computer applications ran on "stand-alone" computers, i.e., computers that were not connected to one another. Today's applications can be written to communicate among the world's hundreds of millions of computers. The Internet mixes computing and communications technologies. It makes our work easier. It makes information instantly and conveniently accessible worldwide. It makes it possible for individuals and small businesses to get worldwide exposure. It is changing the nature of the way business is done. People can search for the best prices on virtually any product or service. Special-interest communities can stay in touch with one another. Researchers can be made instantly aware of the latest breakthroughs worldwide.

## 1.17 General Notes About C++ and This Book

C++ is a complex language. Experienced C++ programmers sometimes take pride in being able to create some weird, contorted, convoluted usage of the language. This is a poor programming practice. It makes programs more difficult to read, more likely to behave strangely, more difficult to test and debug and more difficult to adapt to changing requirements. This book is geared for novice programmers, so we stress program *clarity*. The following is our first "good programming practice."

**Good Programming Practice 1.1**

*Write your C++ programs in a simple and straightforward manner. This is sometimes referred to as KIS ("keep it simple"). Do not "stretch" the language by trying bizarre usages.*

You have heard that C and C++ are portable languages, and that programs written in C and C++ can run on many different computers. *Portability is an elusive goal.* The ANSI C standard document contains a lengthy list of portability issues and complete books have been written that discuss portability.

### Portability Tip 1.3

*Although it is possible to write portable programs, there are many problems among different C and C++ compilers and different computers that can make portability difficult to achieve. Simply writing programs in C and C++ does not guarantee portability. The programmer will often need to deal directly with compiler and computer variations.*

We have done a careful walkthrough of the ANSI/ISO C++ standard document and audited our presentation against it for completeness and accuracy. However, C++ is a rich language, and there are some subtleties in the language and some advanced subjects we have not covered. If you need additional technical details on C++, we suggest that you read the C++ standard document. You can order the C++ standard document from the ANSI web site

```
http://www.ansi.org/
```

The title of the document is "Information Technology – Programming Languages – C++" and its document number is ISO/IEC 14882-1998. If you prefer not to purchase the document, the older draft version of the standard can be viewed at the World Wide Web site

```
http://www.cygnus.com/misc/wp/
```

We have included an extensive bibliography of books and papers on C++ and object-oriented programming. We also have included a C++ Resources appendix containing many Internet and World Wide Web sites relating to C++ and object-oriented programming.

Many features of the current versions of C++ are not compatible with older C++ implementations, so you may find that some of the programs in this text do not work on older C++ compilers.

### Good Programming Practice 1.2

*Read the manuals for the version of C++ you are using. Refer to these manuals frequently to be sure you are aware of the rich collection of C++ features and that you are using these features correctly.*

### Good Programming Practice 1.3

*Your computer and compiler are good teachers. If after carefully reading your C++ language manual you are not sure how a feature of C++ works, experiment using a small "test program" and see what happens. Set your compiler options for "maximum warnings." Study each message you get when you compile your programs and correct the programs to eliminate the messages.*

## 1.18 Introduction to C++ Programming

The C++ language facilitates a structured and disciplined approach to computer program design. We now introduce C++ programming and present several examples that illustrate many important features of C++. Each example is analyzed one statement at a time. In Chapter 2 we present a detailed treatment of *structured programming* in C++. We then use the structured approach through Chapter 5. Beginning with Chapter 6, we study object-ori-

ented programming in C++. Again, because of the central importance of object-oriented programming in this book, each of the first five chapters concludes with a section entitled "Thinking About Objects." These special sections introduce the concepts of object orientation and present a case study that challenges the reader to design and implement a substantial object-oriented C++ program.

## 1.19 A Simple Program: Printing a Line of Text

C++ uses notations that may appear strange to non-programmers. We begin by considering a simple program that prints a line of text. The program and its screen output are shown in Fig. 1.2.

This program illustrates several important features of the C++ language. We consider each line of the program in detail. Lines 1 and 2

```
// Fig. 1.2: fig01_02.cpp
// A first program in C++
```

each begin with **//** indicating that the remainder of each line is a *comment*. Programmers insert comments to *document* programs and improve program readability. Comments also help other people read and understand your program. Comments do not cause the computer to perform any action when the program is run. Comments are ignored by the C++ compiler and do not cause any machine language object code to be generated. The comment **A first program in C++** simply describes the purpose of the program. A comment that begins with **//** is called a *single-line comment* because the comment terminates at the end of the current line. [Note: C++ programmers may also use C's comment style in which a comment—possibly containing many lines—begins with **/*** and ends with ***/**.]

### Good Programming Practice 1.4

*Every program should begin with a comment describing the purpose of the program.*

Line 3

```
#include <iostream>
```

```
1  // Fig. 1.2: fig01_02.cpp
2  // A first program in C++
3  #include <iostream>
4
5  int main()
6  {
7     std::cout << "Welcome to C++!\n";
8
9     return 0;      // indicate that program ended successfully
10 }
```

```
Welcome to C++!
```

Fig. 1.2 Text printing program.

is a *preprocessor directive,* i.e., a message to the C++ preprocessor. Lines beginning with **#** are processed by the preprocessor before the program is compiled. This specific line tells the preprocessor to include in the program the contents of the *input/output stream header file* **<iostream>**. This file must be included for any program that outputs data to the screen or inputs data from the keyboard using C++-style stream input/output. Figure 1.2 outputs data to the screen, as we will soon see. The contents of **iostream** will be explained in more detail later.

**Common Programming Error 1.2**

*Forgetting to include the* ***iostream*** *file in a program that inputs data from the keyboard or outputs data to the screen causes the compiler to issue an error message.*

Line 5

```
int main()
```

is a part of every C++ program. The parentheses after **main** indicate that **main** is a program building block called a *function.* C++ programs contain one or more functions, exactly one of which must be **main**. Figure 1.2 contains only one function. C++ programs begin executing at function **main**, even if **main** is not the first function in the program. The keyword **int** to the left of **main** indicates that **main** "returns" an integer (whole number) value. We will explain what it means for a function to "return a value" when we study functions in depth in Chapter 3. For now, simply include the keyword **int** to the left of **main** in each of your programs.

The *left brace,* **{**, (line 6) must begin the *body* of every function. A corresponding *right brace,* **}**, (line 10) must end the body of each function. Line 7

```
std::cout << "Welcome to C++!\n";
```

instructs the computer to print on the screen the *string* of characters contained between the quotation marks. The entire line, including **std::cout**, the **<<** *operator,* the *string* **"Welcome to C++!\n"** and the *semicolon* (**;**), is called a *statement.* Every statement must end with a semicolon (also known as the *statement terminator*). Output and input in C++ is accomplished with *streams* of characters. Thus, when the preceding statement is executed, it sends the stream of characters **Welcome to C++!** to the *standard output stream object*—***std::cout***—which is normally "connected" to the screen. We discuss **std::cout**'s many features in detail in Chapter 11, *Stream Input/Output*.

Notice that we placed **std::** before **cout**. This is required when we use the preprocessor directive **#include <iostream>**. The notation **std::cout** specifies that we are using a name, in this case **cout**, that belongs to "namespace" **std**. Namespaces are an advanced C++ feature. We discuss namespaces in depth in Chapter 21. For now, you should simply remember to include **std::** before each mention of **cout**, **cin** and **cerr** in a program. This can be cumbersome—in Fig. 1.14, we introduce the **using** statement, which will enable us to avoid having to place **std::** before each use of a namespace **std** name.

The operator **<<** is referred to as the *stream insertion operator.* When this program executes, the value to the right of the operator, the right *operand,* is inserted in the output stream. The characters of the right operand normally print exactly as they appear between the double quotes. Notice, however, that the characters **\n** are not printed on the screen. The backslash (****) is called an *escape character.* It indicates that a "special" character is to

be output. When a backslash is encountered in a string of characters, the next character is combined with the backslash to form an *escape sequence*. The escape sequence **\n** means *newline*. It causes the *cursor* (i.e., the current screen position indicator) to move to the beginning of the next line on the screen. Some other common escape sequences are listed in Fig. 1.3.

### Common Programming Error 1.3

*Omitting the semicolon at the end of a statement is a syntax error. A syntax error is caused when the compiler cannot recognize a statement. The compiler normally issues an error message to help the programmer locate and fix the incorrect statement. Syntax errors are violations of the language. Syntax errors are also called compile errors, compile-time errors, or compilation errors because they appear during the compilation phase.*

Line 9

```
return 0;   // indicate that program ended successfully
```

is included at the end of every **main** function. C++ keyword **return** is one of several means we will use to *exit a function*. When the **return** statement is used at the end of **main** as shown here, the value **0** indicates that the program has terminated successfully. In Chapter 3, we discuss functions in detail and the reasons for including this statement will become clear. For now, simply include this statement in each program, or the compiler may produce a warning on some systems.

The right brace, **}**, (line 10) indicates the end of **main**.

### Good Programming Practice 1.5

*Many programmers make the last character printed by a function a newline (**\n**). This ensures that the function will leave the screen cursor positioned at the beginning of a new line. Conventions of this nature encourage software reusability—a key goal in software development environments.*

### Good Programming Practice 1.6

*Indent the entire body of each function one level of indentation within the braces that define the body of the function. This makes the functional structure of a program stand out and helps make programs easier to read.*

| Escape Sequence | Description |
|---|---|
| **\n** | Newline. Position the screen cursor to the beginning of the next line. |
| **\t** | Horizontal tab. Move the screen cursor to the next tab stop. |
| **\r** | Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line. |
| **\a** | Alert. Sound the system bell. |
| **** | Backslash. Used to print a backslash character. |
| **\"** | Double quote. Used to print a double quote character. |

**Fig. 1.3** Some common escape sequences.

### Good Programming Practice 1.7

*Set a convention for the size of indent you prefer then uniformly apply that convention. The tab key may be used to create indents, but tab stops may vary. We recommend using either 1/4-inch tab stops or (preferably) three spaces to form a level of indent.*

**`Welcome to C++!`** can be printed several ways. For example, Fig. 1.4 uses multiple stream insertion statements (lines 7 and 8), yet produces identical output to the program of Fig. 1.2. This works because each stream insertion statement resumes printing where the previous statement stopped printing. The first stream insertion prints **`Welcome`** followed by a space and the second stream insertion begins printing on the same line immediately following the space. In general, C++ allows the programmer to express statements in a variety of ways.

A single statement can print multiple lines by using newline characters as in Fig. 1.5. Each time the **`\n`** (newline) escape sequence is encountered in the output stream, the screen cursor is positioned to the beginning of the next line. To get a blank line in your output, simply place two newline characters back to back as in Fig. 1.5.

```
1  // Fig. 1.4: fig01_04.cpp
2  // Printing a line with multiple statements
3  #include <iostream>
4
5  int main()
6  {
7     std::cout << "Welcome ";
8     std::cout << "to C++!\n";
9
10    return 0;   // indicate that program ended successfully
11 }
```

```
Welcome to C++!
```

**Fig. 1.4** Printing on one line with separate statements using **`cout`**.

```
1  // Fig. 1.5: fig01_05.cpp
2  // Printing multiple lines with a single statement
3  #include <iostream>
4
5  int main()
6  {
7     std::cout << "Welcome\nto\n\nC++!\n";
8
9     return 0;   // indicate that program ended successfully
10 }
```

```
Welcome
to

C++!
```

**Fig. 1.5** Printing on multiple lines with a single statement using **`cout`**.

## 1.20 Another Simple Program: Adding Two Integers

Our next program uses the input stream object **`std::cin`** and the *stream extraction operator*, **`>>`**, to obtain two integers typed by a user at the keyboard, computes the sum of these values and outputs the result using **`std::cout`**. The program and sample output are shown in Fig. 1.6.

The comments in lines 1 and 2

```
// Fig. 1.6: fig01_06.cpp
// Addition program
```

state the name of the file and the purpose of the program. The C++ preprocessor directive

```
#include <iostream>
```

in line 3 includes the contents of the **`iostream`** header file in the program.

As stated earlier, every program begins execution with function **`main`**. The left brace marks the beginning of **`main`**'s body and the corresponding right brace marks the end of **`main`**. Line 7

```
int integer1, integer2, sum;      // declaration
```

is a *declaration*. The words **`integer1`**, **`integer2`** and **`sum`** are the names of *variables*. A variable is a location in the computer's memory where a value can be stored for use by a program. This declaration specifies that the variables **`integer1`**, **`integer2`** and **`sum`** are data of type **`int`** which means that these variables will hold *integer* values, i.e., whole numbers such as 7, -11, 0, 31914. All variables must be declared with a name and a data type before they can be used in a program. Several variables of the same type may be declared in one declaration or in multiple declarations. We could have written three declarations, one for each variable, but the preceding declaration is more concise.

```
1  // Fig. 1.6: fig01_06.cpp
2  // Addition program
3  #include <iostream>
4
5  int main()
6  {
7     int integer1, integer2, sum;           // declaration
8
9     std::cout << "Enter first integer\n";  // prompt
10    std::cin >> integer1;                  // read an integer
11    std::cout << "Enter second integer\n"; // prompt
12    std::cin >> integer2;                  // read an integer
13    sum = integer1 + integer2;             // assignment of sum
14    std::cout << "Sum is " << sum << std::endl; // print sum
15
16    return 0;   // indicate that program ended successfully
17 }
```

**Fig. 1.6** An addition program (part 1 of 2).

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

Fig. 1.6 An addition program (part 2 of 2).

### Good Programming Practice 1.8

*Some programmers prefer to declare each variable on a separate line. This format allows for easy insertion of a descriptive comment next to each declaration.*

We will soon discuss the data types **`double`** (for specifying real numbers, i.e., numbers with decimal points like 3.4, 0.0, -11.19) and **`char`** (for specifying character data; a **`char`** variable may hold only a single lowercase letter, a single uppercase letter, a single digit, or a single special character like a **`x`**, **`$`**, **`7`**, **`*`**, etc.).

### Good Programming Practice 1.9

*Place a space after each comma (`,`) to make programs more readable.*

A variable name is any valid *identifier*. An identifier is a series of characters consisting of letters, digits and underscores ( _ ) that does not begin with a digit. C++ is *case sensitive*—uppercase and lowercase letters are different, so **`a1`** and **`A1`** are different identifiers.

### Portability Tip 1.4

*C++ allows identifiers of any length, but your system and/or C++ implementation may impose some restrictions on the length of identifiers. Use identifiers of 31 characters or fewer to ensure portability.*

### Good Programming Practice 1.10

*Choosing meaningful variable names helps a program to be "self-documenting," i.e., it becomes easier to understand the program simply by reading it rather than having to read manuals or use excessive comments.*

### Good Programming Practice 1.11

*Avoid identifiers that begin with underscores and double underscores because C++ compilers may use names like that for their own purposes internally. This will prevent names you choose from being confused with names the compilers choose.*

Declarations of variables can be placed almost anywhere in a function. However, the declaration of a variable must appear before the variable is used in the program. For example, in the program of Fig. 1.6, instead of using a single declaration for all three variables, three separate declarations could have been used. The declaration

```
int integer1;
```

could have been placed immediately before the line

```
std::cin >> integer1;
```

the declaration

```
int integer2;
```

could have been placed immediately before the line

```
std::cin >> integer2;
```

and the declaration

```
int sum;
```

could have been placed immediately before the line

```
sum = integer1 + integer2;
```

### Good Programming Practice 1.12

*Always place a blank line between a declaration abd adjacent executable statements. This makes the declarations stand out in the program and contributes to program clarity.*

### Good Programming Practice 1.13

*If you prefer to place declarations at the beginning of a function, separate those declarations from the executable statements in that function with one blank line to highlight where the declarations end and the executable statements begin.*

Line 9

```
std::cout << "Enter first integer\n";  // prompt
```

prints the string **`Enter first integer`** (also known as a *string literal* or a *literal*) on the screen and positions to the beginning of the next line. This message is called a *prompt* because it tells the user to take a specific action. We like to pronounce the preceding statement as "**`cout`** *gets* the character string **`"Enter first integer\n"`**."

Line 10

```
std::cin >> integer1;                    // read an integer
```

uses the *input stream object* ***`cin`*** (of namespace **`std`**) and the *stream extraction operator,* **`>>`**, to obtain a value from the keyboard. Using the stream extraction operator with **`std::cin`** takes character input from the standard input stream which is usually the keyboard. We like to pronounce the preceding statement as, "**`std::cin`** *gives* a value to **`integer1`**" or simply "**`std::cin`** *gives* **`integer1`**."

When the computer executes the preceding statement, it waits for the user to enter a value for variable **`integer1`**. The user responds by typing an integer (as characters) then pressing the *Enter* key (sometimes called the *Return* key) to send the number to the computer. The computer then converts the character representation of the number to an integer and assigns this number (or *value*) to the variable **`integer1`**. Any subsequent references to **`integer1`** in the program will use this same value.

The **`std::cout`** and **`std::cin`** stream objects facilitate interaction between the user and the computer. Because this interaction resembles a dialogue, it is often called *conversational computing* or *interactive computing.*

Line 11

```
std::cout << "Enter second integer\n"; // prompt
```

prints the words **Enter second integer** on the screen, then positions to the beginning of the next line. This statement prompts the user to take action. Line 12

```
std::cin >> integer2;                    // read an integer
```

obtains a value for variable **integer2** from the user.

The assignment statement in line 13

```
sum = integer1 + integer2;          // assignment of sum
```

calculates the sum of the variables **integer1** and **integer2** and assigns the result to variable **sum** using the *assignment operator* **=**. The statement is read as, "**sum** *gets* the value of **integer1 + integer2**." Most calculations are performed in assignment statements. The **=** operator and the **+** operator are called *binary operators* because they each have two *operands.* In the case of the **+** operator, the two operands are **integer1** and **integer2**. In the case of the preceding **=** operator, the two operands are **sum** and the value of the expression **integer1 + integer2**.

### Good Programming Practice 1.14

*Place spaces on either side of a binary operator. This makes the operator stand out and makes the program more readable.*

Line 14

```
std::cout << "Sum is " << sum << std::endl; // print sum
```

displays the character string **Sum is** followed by the numerical value of variable **sum** followed by **std::endl** (**endl** is an abbreviation for "end line;" **endl**, also, is a name in namespace **std**)—a so-called *stream manipulator*. The **std::endl** manipulator outputs a newline then "flushes the output buffer." This simply means that on some systems where outputs accumulate in the machine until there are enough to "make it worthwhile to display on the screen," **std::endl** forces any accumulated outputs to be displayed at that moment.

Note that the preceding statement outputs multiple values of different types. The stream insertion operator "knows" how to output each piece of data. Using multiple stream insertion operators (**<<**) in a single statement is referred to as *concatenating, chaining* or *cascading stream insertion operations.* Thus, it is unnecessary to have multiple output statements to output multiple pieces of data.

Calculations can also be performed in output statements. We could have combined the statements at lines 13 and 14 into the statement

```
std::cout << "Sum is " << integer1 + integer2 << std::endl;
```

thus eliminating the need for the variable **sum**.

The right brace, **}**, informs the computer that the end of function **main** has been reached.

A powerful feature of C++ is that users can create their own data types (we will explore this capability in Chapter 6). Users can then "teach" C++ how to input and output values of these new data types using the **>>** and **<<** operators (this is called *operator overloading*—a topic we will explore in Chapter 8).

## 1.21 Memory Concepts

Variable names such as **integer1**, **integer2** and **sum** actually correspond to *locations* in the computer's memory. Every variable has a *name,* a *type,* a *size* and a *value.*

In the addition program of Fig. 1.6, when the statement

```
std::cin >> integer1;
```

is executed, the characters typed by the user are converted to an integer that is placed into a memory location to which the name **integer1** has been assigned by the C++ compiler. Suppose the user enters the number **45** as the value for **integer1**. The computer will place **45** into location **integer1** as shown in Fig. 1.7.

Whenever a value is placed in a memory location, the value replaces the previous value in that location. The previous value is lost.

Returning to our addition program, when the statement

```
std::cin >> integer2;
```

is executed, suppose the user enters the value **72**. This value is placed into location **integer2** and memory appears as in Fig. 1.8. Note that these locations are not necessarily adjacent locations in memory.

Once the program has obtained values for **integer1** and **integer2**, it adds these values and places the sum into variable **sum**. The statement

```
sum = integer1 + integer2;
```

that performs the addition also replaces whatever value was stored in **sum**. This occurs when the calculated sum of **integer1** and **integer2** is placed into location **sum** (without regard to what value may already be in **sum**; that value is lost). After **sum** is calculated, memory appears as in Fig. 1.9. Note that the values of **integer1** and **integer2** appear exactly as they did before they were used in the calculation of **sum**. These values were used, but not destroyed, as the computer performed the calculation. Thus, when a value is read out of a memory location, the process is nondestructive.

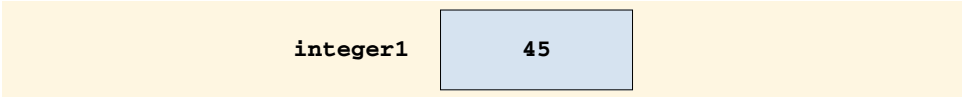



**Fig. 1.7** A memory location showing the name and value of a variable.

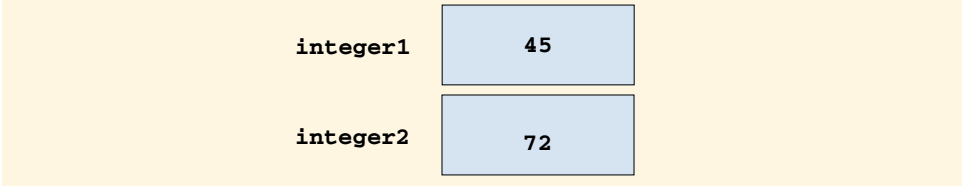



**Fig. 1.8** Memory locations after values for two variables have been input.

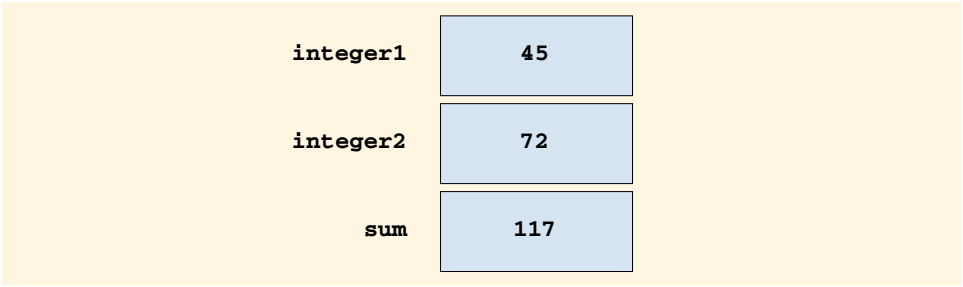



**Fig. 1.9** Memory locations after a calculation.

## 1.22 Arithmetic

Most programs perform arithmetic calculations. The *arithmetic operators* are summarized in Fig. 1.10. Note the use of various special symbols not used in algebra. The *asterisk (`*`)* indicates multiplication and the *percent sign (`%`)* is the *modulus* operator that will be discussed shortly. The arithmetic operators in Fig. 1.10 are all binary operators, i.e., operators that take two operands. For example, the expression **`integer1 + integer2`** contains the binary operator **`+`** and the two operands **`integer1`** and **`integer2`**.

*Integer division* (i.e., both the numerator and the denominator are integers) yields an integer result; for example, the expression **`7 / 4`** evaluates to **`1`** and the expression **`17 / 5`** evaluates to **`3`**. Note that any fractional part in integer division is simply discarded (i.e., *truncated*)—no rounding occurs.

C++ provides the *modulus operator*, **`%`**, that yields the remainder after integer division. The modulus operator can be used only with integer operands. The expression **`x % y`** yields the remainder after **`x`** is divided by **`y`**. Thus, **`7 % 4`** yields **`3`** and **`17 % 5`** yields **`2`**. In later chapters, we will discuss many interesting applications of the modulus operator such as determining if one number is a multiple of another (a special case of this is determining if a number is odd or even).

### Common Programming Error 1.4

*Attempting to use the modulus operator, **`%`**, with non-integer operands is a syntax error.*

| C++ operation | Arithmetic operator | Algebraic expression | C++ expression |
|---|---|---|---|
| Addition | `+` | $f + 7$ | `f + 7` |
| Subtraction | `-` | $p - c$ | `p - c` |
| Multiplication | `*` | $bm$ | `b * m` |
| Division | `/` | $x / y$ or $\frac{x}{y}$ or $x \div y$ | `x / y` |
| Modulus | `%` | $r \bmod s$ | `r % s` |

**Fig. 1.10** Arithmetic operators.

Arithmetic expressions in C++ must be entered into the computer in *straight-line form*. Thus, expressions such as "**a** divided by **b**" must be written as **a / b** so that all constants, variables and operators appear in a straight line. The algebraic notation

$$\frac{a}{b}$$

is generally not acceptable to compilers, although some special-purpose software packages do exist that support more natural notation for complex mathematical expressions.

Parentheses are used in C++ expressions in much the same manner as in algebraic expressions. For example, to multiply **a** times the quantity **b + c** we write:

```
a * (b + c)
```

C++ applies the operators in arithmetic expressions in a precise sequence determined by the following *rules of operator precedence,* which are generally the same as those followed in algebra:

1. Operators in expressions contained within pairs of parentheses are evaluated first. Thus, *parentheses may be used to force the order of evaluation to occur in any sequence desired by the programmer.* Parentheses are said to be at the "highest level of precedence." In cases of *nested,* or *embedded,* parentheses, the operators in the innermost pair of parentheses are applied first.
2. Multiplication, division and modulus operations are applied next. If an expression contains several multiplication, division and modulus operations, operators are applied from left to right. Multiplication, division and modulus are said to be on the same level of precedence.
3. Addition and subtraction operations are applied last. If an expression contains several addition and subtraction operations, operators are applied from left to right. Addition and subtraction also have the same level of precedence.

The rules of operator precedence enable C++ to apply operators in the correct order. When we say that certain operators are applied from left to right, we are referring to the *associativity* of the operators. For example, in the expression

```
a + b + c
```

the addition operators (**+**) associate from left to right. We will see that some operators associate from right to left. Fig. 1.11 summarizes these rules of operator precedence. This table will be expanded as additional C++ operators are introduced. A complete precedence chart is included in the appendices.

Now let us consider several expressions in light of the rules of operator precedence. Each example lists an algebraic expression and its C++ equivalent.

The following is an example of an arithmetic mean (average) of five terms:

Algebra: $m = \frac{a + b + c + d + e}{5}$

C++: `m = ( a + b + c + d + e ) / 5;`

| Operator(s) | Operation(s) | Order of evaluation (precedence) |
|---|---|---|
| `( )` | Parentheses | Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right. |
| `*`, `/`, or `%` | Multiplication<br>Division<br>Modulus | Evaluated second. If there are several, they are evaluated left to right. |
| `+` or `-` | Addition<br>Subtraction | Evaluated last. If there are several, they are evaluated left to right. |

**Fig. 1.11** Precedence of arithmetic operators.

The parentheses are required because division has higher precedence than addition. The entire quantity `( a + b + c + d + e )` is to be divided by `5`. If the parentheses are erroneously omitted, we obtain `a + b + c + d + e / 5` which evaluates incorrectly as

$$a + b + c + d + \frac{e}{5}$$

The following is an example of the equation of a straight line:

Algebra: $y = mx + b$

C++: `y = m * x + b;`

No parentheses are required. The multiplication is applied first because multiplication has a higher precedence than addition.

The following example contains modulus (`%`), multiplication, division, addition and subtraction operations:

Algebra: $z = pr\%q + w/x - y$

C++:
```
z = p * r % q + w / x - y;
    6   1   2   4   3   5
```

The circled numbers under the statement indicate the order in which C++ applies the operators. The multiplication, modulus and division are evaluated first in left-to-right order (i.e., they associate from left to right) since they have higher precedence than addition and subtraction. The addition and subtraction are applied next. These are also applied left to right.

Not all expressions with several pairs of parentheses contain nested parentheses. For example, the expression

```
a * (b + c) + c * (d + e)
```

does not contain nested parentheses. Rather, the parentheses are said to be "on the same level."

To develop a better understanding of the rules of operator precedence, consider how a second-degree polynomial is evaluated.

```
y = a * x * x + b * x + c;
    6   1   2   4   3   5
```

The circled numbers under the statement indicate the order in which C++ applies the operators. There is no arithmetic operator for exponentiation in C++, so we have represented $x^2$ as **x * x**. We will soon discuss the standard library function **pow** ("power") that performs exponentiation. Because of some subtle issues related to the data types required by **pow**, we defer a detailed explanation of **pow** until Chapter 3.

Suppose variables **a**, **b**, **c** and **x** are initialized as follows: **a = 2**, **b = 3**, **c = 7** and **x = 5**. Figure 1.12 illustrates the order in which the operators are applied in the preceding second degree polynomial.

The preceding assignment statement can be parenthesized with unnecessary parentheses for clarity as

```
y = ( a * x * x ) + ( b * x ) + c;
```

*Step 1.* **y = 2 * 5 * 5 + 3 * 5 + 7;**

**2 * 5 is 10** (Leftmost multiplication)

*Step 2.* **y = 10 * 5 + 3 * 5 + 7;**

**10 * 5 is 50** (Leftmost multiplication)

*Step 3.* **y = 50 + 3 * 5 + 7;**

**3 * 5 is 15** (Multiplication before addition)

*Step 4.* **y = 50 + 15 + 7;**

**50 + 15 is 65** (Leftmost addition)

*Step 5.* **y = 65 + 7;**

**65 + 7 is 72** (Last addition)

*Step 6.* **y = 72;** (Last operation—place **72** into **y**)

**Fig. 1.12** Order in which a second-degree polynomial is evaluated.

### Good Programming Practice 1.15

*As in algebra, it is acceptable to place unnecessary parentheses in an expression to make the expression clearer. These parentheses are called redundant parentheses. Redundant parentheses are commonly used to group subexpressions in a large expression to make that expression clearer. Breaking a large statement into a sequence of shorter, simpler statements also promotes clarity.*

## 1.23 Decision Making: Equality and Relational Operators

This section introduces a simple version of C++'s ***`if`*** *structure* that allows a program to make a decision based on the truth or falsity of some *condition*. If the condition is met, i.e., the condition is **`true`**, the statement in the body of the **`if`** structure is executed. If the condition is not met, i.e., the condition is **`false`**, the body statement is not executed. We will see an example shortly.

Conditions in **`if`** structures can be formed by using the *equality operators* and *relational operators* summarized in Fig. 1.13. The relational operators all have the same level of precedence and associate left to right. The equality operators both have the same level of precedence, which is lower than the precedence of the relational operators. The equality operators also associate left to right.

### Common Programming Error 1.5

*A syntax error will occur if any of the operators* **`==`**, **`!=`**, **`>=`** *and* **`<=`** *appears with spaces between its pair of symbols.*

### Common Programming Error 1.6

*Reversing the order of the pair of operators in any of the operators* **`!=`**, **`>=`** *and* **`<=`** *(by writing them as* **`=!`**, **`=>`** *and* **`=<`**, *respectively) is normally a syntax error. In some cases, writing* **`!=`** *as* **`=!`** *will not be a syntax error, but will almost certainly be a logic error.*

| Standard algebraic equality operator or relational operator | C++ equality or relational operator | Example of C++ condition | Meaning of C++ condition |
|---|---|---|---|
| *Relational operators* | | | |
| > | `>` | `x > y` | `x` is greater than `y` |
| < | `<` | `x < y` | `x` is less than `y` |
| ≥ | `>=` | `x >= y` | `x` is greater than or equal to `y` |
| ≤ | `<=` | `x <= y` | `x` is less than or equal to `y` |
| *Equality operators* | | | |
| = | `==` | `x == y` | `x` is equal to `y` |
| ≠ | `!=` | `x != y` | `x` is not equal to `y` |

**Fig. 1.13** Equality and relational operators.

### Common Programming Error 1.7

*Confusing the equality operator* **==** *with the assignment operator* **=**. *The equality operator should be read "is equal to" and the assignment operator should be read "gets" or "gets the value of" or "is assigned the value of." Some people prefer to read the equality operator as "double equals." As we will soon see, confusing these operators may not necessarily cause an easy-to-recognize syntax error, but may cause extremely subtle logic errors.*

The following example uses six **if** statements to compare two numbers input by the user. If the condition in any of these **if** statements is satisfied, the output statement associated with that **if** is executed. The program and the input/output dialogs of three sample executions are shown in Fig. 1.14.

```
1  // Fig. 1.14: fig01_14.cpp
2  // Using if statements, relational
3  // operators, and equality operators
4  #include <iostream>
5
6  using std::cout;  // program uses cout
7  using std::cin;   // program uses cin
8  using std::endl;  // program uses endl
9
10 int main()
11 {
12    int num1, num2;
13
14    cout << "Enter two integers, and I will tell you\n"
15         << "the relationships they satisfy: ";
16    cin >> num1 >> num2;   // read two integers
17
18    if ( num1 == num2 )
19       cout << num1 << " is equal to " << num2 << endl;
20
21    if ( num1 != num2 )
22       cout << num1 << " is not equal to " << num2 << endl;
23
24    if ( num1 < num2 )
25       cout << num1 << " is less than " << num2 << endl;
26
27    if ( num1 > num2 )
28       cout << num1 << " is greater than " << num2 << endl;
29
30    if ( num1 <= num2 )
31       cout << num1 << " is less than or equal to "
32            << num2 << endl;
33
34    if ( num1 >= num2 )
35       cout << num1 << " is greater than or equal to "
36            << num2 << endl;
37
38    return 0;   // indicate that program ended successfully
39 }
```

**Fig. 1.14** Using equality and relational operators (part 1 of 2).

```
Enter two integers, and I will tell you
the relationships they satisfy: 3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 22 12
22 is not equal to 12
22 is greater than 12
22 is greater than or equal to 12
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 7 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

**Fig. 1.14** Using equality and relational operators (part 2 of 2).

Lines 6 through 8

```
using std::cout;   // program uses cout
using std::cin;    // program uses cin
using std::endl;   // program uses endl
```

are ***using*** *statements* that help us eliminate the need to repeat the **std::** prefix. Once we include these **using** statements, we can write **cout** instead of **std::cout**, **cin** instead of **std::cin** and **endl** instead of **std::endl**, respectively, in the remainder of the program. [*Note:* From this point forward in the book, each example contains one or more **using** statements.]

Line 12

```
int num1, num2;
```

declares the variables used in the program. Remember that variables may be declared in one declaration or in multiple declarations. If more than one name is declared in a declaration (as in this example), the names are separated by commas (**,**). This is referred to as a *comma-separated list*.

The program uses cascaded stream extraction operations (line 16) to input two integers. Remember, that we are allowed to write **cin** (instead of **std::cin**) because of line 7. First a value is read into variable **num1**, then a value is read into variable **num2**.

The **if** structure at lines 18 and 19

```
if ( num1 == num2 )
   cout << num1 << " is equal to " << num2 << endl;
```

compares the values of variables **num1** and **num2** to test for equality. If the values are equal, the statement at line 19 displays a line of text indicating that the numbers are equal. If the conditions are **true** in one or more of the **if** structures starting at lines 21, 24, 27, 30 and 34, the corresponding **cout** statement displays a line of text.

Notice that each **if** structure in Fig. 1.14 has a single statement in its body and that each body is indented. Indenting the body of an **if** structure enhances program readability. In Chapter 2 we show how to specify **if** structures with multiple-statement bodies (by enclosing the body statements in a pair of braces, **{ }**).

### Good Programming Practice 1.16

*Indent the statement in the body of an **if** structure to make the body of the structure stand out and to enhance program readability.*

### Good Programming Practice 1.17

*There should be no more than one statement per line in a program.*

### Common Programming Error 1.8

*Placing a semicolon immediately after the right parenthesis after the condition in an **if** structure is often a logic error (although not a syntax error). The semicolon would cause the body of the **if** structure to be empty, so the **if** structure itself would perform no action regardless of whether or not its condition is **true**. Worse yet, the original body statement of the **if** structure would now become a statement in sequence with the **if** structure and would always be executed, often causing the program to produce incorrect results.*

Notice the use of spacing in Fig. 1.14. In C++ statements, *white space* characters such as tabs, newlines and spaces are normally ignored by the compiler. So, statements may be split over several lines and may be spaced according to the programmer's preferences. It is incorrect to split identifiers, strings (such as **"hello"**) and constants (such as the number **1000**) over several lines.

### Common Programming Error 1.9

*It is a syntax error to split an identifier by inserting white space characters (e.g., writing **main** as **ma in**).*

### Good Programming Practice 1.18

*A lengthy statement may be spread over several lines. If a single statement must be split across lines, choose breaking points that make sense such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines.*

Figure 1.15 shows the precedence of the operators introduced in this chapter. The operators are shown top to bottom in decreasing order of precedence. Notice that all these operators, with the exception of the assignment operator **=**, associate from left to right. Addition is left associative, so an expression like **x + y + z** is evaluated as if it had been written **(x + y) + z**. The assignment operator **=** associates from right to left, so an expression like **x = y = 0** is evaluated as if it had been written **x = (y = 0)** which, as we will soon see, first assigns **0** to **y** then assigns the result of that assignment—**0**—to **x**.

| Operators | Associativity | Type |
|---|---|---|
| `( )` | left to right | parentheses |
| `*` `/` `%` | left to right | multiplicative |
| `+` `-` | left to right | additive |
| `<<` `>>` | left to right | stream insertion/extraction |
| `<` `<=` `>` `>=` | left to right | relational |
| `==` `!=` | left to right | equality |
| `=` | right to left | assignment |

**Fig. 1.15** Precedence and associativity of the operators discussed so far.

### Good Programming Practice 1.19

*Refer to the operator precedence chart when writing expressions containing many operators. Confirm that the operators in the expression are performed in the order you expect. If you are uncertain about the order of evaluation in a complex expression, break the expression into smaller statements or use parentheses to force the order, exactly as you would do in an algebraic expression. Be sure to observe that some operators such as assignment (=) associate right to left rather than left to right.*

We have introduced many important features of C++ including printing data on the screen, inputting data from the keyboard, performing calculations and making decisions. In Chapter 2, we build on these techniques as we introduce *structured programming*. You will become more familiar with indentation techniques. We will study how to specify and vary the order in which statements are executed—this order is called *flow of control*.

## 1.24 Thinking About Objects: Introduction to Object Technology and the Unified Modeling Language™

Now we begin our early introduction to object orientation. We will see that object orientation is a natural way of thinking about the world and of writing computer programs.

In each of the first five chapters we concentrate on the "conventional" methodology of structured programming, because the objects we will build will be composed in part of structured-program pieces. We then end each chapter with a "Thinking About Objects" section in which we present a carefully paced introduction to object orientation. Our goal in these "Thinking About Objects" sections is to help you develop an object-oriented way of thinking, so that you can immediately put to use the knowledge of object-oriented programming that you begin to receive in Chapter 6. We will also introduce you to the *Unified Modeling Language (UML)*. The UML is a graphical language that allows people who build systems (e.g., software architects, systems engineers, programmers, etc.) to represent their object-oriented designs using a common notation.

In this required section (1.24), we introduce basic concepts (i.e., "object think") and terminology (i.e., "object speak"). In the optional "Thinking About Objects" sections at the ends of Chapters 2 through 5 we consider more substantial issues as we attack a challenging problem with the techniques of *object-oriented design (OOD)*. We will analyze a typical

problem statement that requires a system to be built, determine the objects needed to implement the system, determine the attributes the objects will need to have, determine the behaviors these objects will need to exhibit and specify how the objects will need to interact with one another to meet the system requirements. We will do all this even before we have learned how to write object-oriented C++ programs. In the optional "Thinking About Objects" sections at the ends of Chapters 6, 7 and 9, we discuss a C++ implementation of the object-oriented system we will design in the earlier chapters.

This case study will help prepare you for the kinds of substantial projects encountered in industry. If you are a student, and your instructor does not plan on including this case study in your course, please consider covering the case study on your own time. We believe it will be well worth your while to walk through this large and challenging project. You will experience a solid introduction to object-oriented design with the UML, and you will sharpen your code-reading skills by touring a carefully written and well-documented 1000-line C++ program that solves the problem presented in the case study.

We begin our introduction to object orientation with some of the key terminology of object orientation. Look around you in the real world. Everywhere you look you see them—*objects*! People, animals, plants, cars, planes, buildings, computers, etc. Humans think in terms of objects. We have the marvelous ability of *abstraction* that enables us to view screen images as objects such as people, planes, trees and mountains, rather than as individual dots of color. We can, if we wish, think in terms of beaches rather than grains of sand, forests rather than trees and houses rather than bricks.

We might be inclined to divide objects into two categories—animate objects and inanimate objects. Animate objects are "alive" in some sense. They move around and do things. Inanimate objects, like towels, seem not to do much at all. They just kind of "sit around." All these objects, however, do have some things in common. They all have *attributes* like size, shape, color, weight, etc. And they all exhibit *behaviors* (e.g., a ball rolls, bounces, inflates and deflates; a baby cries, sleeps, crawls, walks and blinks; a car accelerates, brakes and turns; a towel absorbs water; etc.)

Humans learn about objects by studying their attributes and observing their behaviors. Different objects can have similar attributes and can exhibit similar behaviors. Comparisons can be made, for example, between babies and adults and between humans and chimpanzees. Cars, trucks, little red wagons and roller skates have much in common.

*Object-oriented programming (OOP)* models real-world objects with software counterparts. It takes advantage of *class* relationships where objects of a certain class—such as a class of vehicles—have the same characteristics. It takes advantage of *inheritance* relationships, and even *multiple inheritance* relationships where newly created classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own. An object of class "convertible" certainly has the characteristics of the more general class "automobile," but a convertible's roof goes up and down.

Object-oriented programming gives us a more natural and intuitive way to view the programming process, namely, by *modeling* real-world objects, their attributes and their behaviors. OOP also models communication between objects. Just as people send *messages* to one another (e.g., a sergeant commanding a soldier to stand at attention), objects also communicate via messages.

OOP *encapsulates* data (attributes) and functions (behavior) into packages called *objects;* the data and functions of an object are intimately tied together. Objects have the

property of *information hiding*. This means that although objects may know how to communicate with one another across well-defined *interfaces,* objects normally are not allowed to know how other objects are implemented—implementation details are hidden within the objects themselves. Surely it is possible to drive a car effectively without knowing the details of how engines, transmissions and exhaust systems work internally. We will see why information hiding is so crucial to good software engineering.

In C and other *procedural programming languages,* programming tends to be *action-oriented*, whereas in C++ programming tends to be *object-oriented.* In C, the unit of programming is the *function.* In C++, the unit of programming is the *class* from which objects are eventually *instantiated* (a fancy term for "created"). C++ classes contain functions (that implement class behaviors) and data (that implement class attributes).

C programmers concentrate on writing functions. Groups of actions that perform some common task are formed into functions, and functions are grouped to form programs. Data is certainly important in C, but the view is that data exists primarily in support of the actions that functions perform. The *verbs* in a system specification help the C programmer determine the set of functions that work together to implement the system.

C++ programmers concentrate on creating their own *user-defined types* called *classes* and *components.* Each class contains data as well as the set of functions that manipulate that data. The data components of a class are called *data members.* The function components of a class are called *member functions* (typically called *methods* in other object-oriented programming languages like Java). Just as an instance of a built-in type such as **`int`** is called a *variable,* an instance of a user-defined type (i.e., a class) is called an *object.* The programmer uses built-in types as the "building blocks" for constructing user-defined types. The focus of attention in C++ is on classes (out of which we make objects) rather than on functions. The *nouns* in a system specification help the C++ programmer determine the set of classes from which objects will be created that will work together to implement the system.

Classes are to objects as blueprints are to houses. We can build many houses from one blueprint, and we can instantiate many objects from one class. Classes can also have relationships with other classes. For example, in an object-oriented design of a bank, the **`BankTeller`** class needs to relate to the **`Customer`** class. These relationships are called *associations*.

We will see that when software is packaged as classes, these classes can be *reused* in future software systems. Groups of related classes are often packaged as reusable *components*. Just as real-estate brokers tell their clients that the three most important factors affecting the price of real estate are "location, location and location," we believe the three most important factors affecting the future of software development are "reuse, reuse and reuse."

Indeed, with object technology, we will build most future software by combining "standardized, interchangeable parts" called classes. This book will teach you how to "craft valuable classes" for reuse, reuse and reuse. Each new class you create will have the potential to become a valuable software asset that you and other programmers can use to speed and enhance the quality of future software development efforts. This is an exciting possibility.

***Introduction to Object-Oriented Analysis and Design (OOAD)***

By now, you have probably written a few small programs in C++. How did you create the code for your programs? If you are like many beginning programmers, you may have

turned on your computer and simply started typing. This approach may work for small projects, but what would you do if you were asked to create a software system to control the automated teller machines for a major bank? Such a project is too large and complex to sit down and simply start typing.

For creating the best solutions, you should follow a detailed process for obtaining an *analysis* of your project's *requirements* and developing a *design* for satisfying those requirements. You would go through this process and have its results reviewed and approved by your superiors before writing any code for your project. If this process involves analyzing and designing your system from an object-oriented point of view, we call it an *object-oriented analysis and design (OOAD) process.* Experienced programmers know that no matter how simple a problem appears, time spent on analysis and design can save innumerable hours that might be lost from abandoning an ill-planned system development approach part of the way through its implementation.

OOAD is the generic term for the ideas behind the process we employ to analyze a problem and develop an approach for solving it. Small problems like the ones in these first few chapters do not require an exhaustive process. It may be sufficient to write *pseudocode* before we begin writing code. (Pseudocode is an informal means of expressing program code. It is not actually a programming language, but we can use it as a kind of "outline" to guide us as we write our code. We introduce pseudocode in Chapter 2.)

Pseudocode may suffice for small problems, but as problems and the groups of people solving these problems increase in size, the methods of OOAD become more involved. Ideally, a group should agree on a strictly defined process for solving the problem and on a uniform way of communicating the results of that process with one another. Many different OOAD processes exist; however, a graphical language for communicating the results of any OOAD process has become widely used. This language is known as the *Unified Modeling Language (UML)*. The UML was developed in the mid-1990s, under the initial direction of a trio of software methodologists: Grady Booch, James Rumbaugh and Ivar Jacobson.

***History of the UML***

In the 1980s, increasing numbers of organizations began using OOP to program their applications, and a need developed for an established process with which to approach OOAD. Many methodologists—including Booch, Rumbaugh and Jacobson—individually produced and promoted separate processes to satisfy this need. Each of these processes had their own notation, or "language" (in the form of graphical diagrams), to convey the results of analysis and design.

By the early 1990s, different companies, and even different divisions within the same company, were using different processes and notations. Additionally, these companies wanted to use software tools that would support their particular processes. With so many processes, software vendors found it difficult to provide such tools. Clearly, standard processes and notation were needed.

In 1994, James Rumbaugh joined Grady Booch at Rational Software Corporation, and the two began working to unify their popular processes. They were soon joined by Ivar Jacobson. In 1996, the group released early versions of the UML to the software engineering community and requested feedback. Around the same time, an organization known as the *Object Management Group™ (OMG™)* invited submissions for a common mod-

eling language. The OMG is a not-for-profit organization that promotes the use of object-oriented technology by issuing guidelines and specifications for object-oriented technologies. Several corporations—among them HP, IBM, Microsoft, Oracle and Rational Software—had already recognized the need for a common modeling language. These companies formed the *UML Partners* in response to the OMG's request for proposals. This consortium developed and submitted the UML version 1.1 to the OMG. The OMG accepted the proposal and, in 1997, assumed responsibility for the continuing maintenance and revision of the UML. In 1999, the OMG released the UML version 1.3 (the current version at the time this book was published).

***What is the UML?***

The Unified Modeling Language is now the most widely used graphical representation scheme for modeling object-oriented systems. It has indeed unified the various notational schemes that existed in the late 1980s. Those who design systems use the language (in the form of graphical diagrams) to model their systems.

One of the most attractive features of the UML is its flexibility. The UML is extendable and is independent of the many OOAD processes. UML modelers are free to develop systems using various processes, but all developers can now express those systems with one standard set of notations.

The UML is a complex, feature-rich graphical language. In our "Thinking About Objects" sections, we present a concise, simplified subset of these features. We then use this subset to guide the reader through a first design experience with the UML intended for the novice object-oriented designer/programmer. For a more complete discussion of the UML, refer to the Object Management Group's Web site (**`www.omg.org`**) and to the official UML 1.3 specifications document (**`www.omg.org/uml/`**). Many UML books have been published. *UML Distilled: Second Edition*, by Martin Fowler (with Kendall Scott) provides a detailed introduction to the UML version 1.3, with many examples. *The Unified Modeling Language User Guide*, written by Booch, Rumbaugh and Jacobson, is the definitive tutorial to the UML.

Object-oriented technology is ubiquitous in the software industry, and the UML is rapidly becoming so. Our goal in these "Thinking About Objects" sections is to encourage you to think in an object-oriented manner as early, and as often, as possible. Beginning in the "Thinking About Objects" section at the end of Chapter 2, you will apply object technology to implement a solution to a substantial problem. We hope that you will find this optional project to be an enjoyable and challenging introduction to object-oriented design with the UML and to object-oriented programming.

## SUMMARY

- A computer is a device capable of performing computations and making logical decisions at speeds millions and even billions of times faster than human beings can.
- Computers process data under the control of computer programs.
- The various devices (such as the keyboard, screen, disks, memory and processing units) that comprise a computer system are referred to as hardware.
- The computer programs that run on a computer are referred to as software.
- The input unit is the "receiving" section of the computer. Most information is entered into computers today through typewriter-like keyboards.

- The output unit is the "shipping" section of the computer. Most information is output from computers today by displaying it on screens or by printing it on paper.
- The memory unit is the "warehouse" section of the computer and is often called either memory or primary memory.
- The arithmetic and logic unit (ALU) performs calculations and makes decisions.
- Programs or data not actively being used by the other units are normally placed on secondary storage devices (such as disks) until they are again needed.
- In single-user batch processing, the computer runs a single program at a time while processing data in groups or batches.
- Operating systems are software systems that make it more convenient to use computers and to get the best performance from computers.
- Multiprogramming operating systems enable the "simultaneous" operation of many jobs on the computer—the computer shares its resources among the jobs.
- Timesharing is a special case of multiprogramming in which users access the computer through terminals. The users' programs appear to be running simultaneously.
- With distributed computing, an organization's computing is distributed via networking to the sites where the work of the organization is performed.
- Servers store programs and data that may be shared by client computers distributed throughout a network, hence the term client/server computing.
- Any computer can directly understand only its own machine language. Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time. Machine languages are machine-dependent.
- English-like abbreviations form the basis of assembly languages. Assemblers translate assembly language programs into machine language.
- Compilers translate high-level language programs into machine language. High-level languages contain English words and conventional mathematical notations.
- Interpreter programs directly execute high-level language programs without the need for compiling those programs into machine language.
- Although compiled programs execute faster than interpreted programs, interpreters are popular in program development environments in which programs are recompiled frequently as new features are added and errors are corrected. Once a program is developed, a compiled version can then be produced to run more efficiently.
- It is possible to write programs in C and C++ that are portable to most computers.
- FORTRAN (FORmula TRANslator) is used for mathematical applications. COBOL (COmmon Business Oriented Language) is used primarily for commercial applications that require precise and efficient manipulation of large amounts of data.
- Structured programming is a disciplined approach to writing programs that are clearer than unstructured programs, easier to test and debug and easier to modify.
- Pascal was designed for teaching structured programming in academic environments.
- Ada was developed under the sponsorship of the United States Department of Defense (DOD) using Pascal as a base.
- Multitasking allows programmers to specify parallel activities.
- All C++ systems consist of three parts: the environment, the language and the standard libraries. Library functions are not part of the C++ language itself; these functions perform operations such as popular mathematical calculations.

- C++ programs typically go through six phases to be executed: edit, preprocess, compile, link, load and execute.
- The programmer types a program with an editor and makes corrections if necessary. C++ file names on a typical UNIX-based system end with the **`.C`** extension.
- A compiler translates a C++ program into machine language code (or object code).
- The preprocessor obeys preprocessor directives which typically indicate files to be included in the file being compiled and special symbols to be replaced with program text.
- A linker links the object code with the code for missing functions to produce an executable image (with no missing pieces). On a typical UNIX-based system, the command to compile and link a C++ program is **`CC`**. If the program compiles and links correctly, a file called **`a.out`** is produced. This is the executable image of the program.
- A loader takes an executable image from disk and transfers it to memory.
- A computer, under the control of its CPU, executes a program one instruction at a time.
- Errors like division-by-zero errors occur as a program runs, so these errors are called run-time errors or execution-time errors.
- Divide-by-zero is generally a fatal error, i.e., an error that causes the program to terminate immediately without having successfully performed its job. Non-fatal errors allow programs to run to completion, often producing incorrect results.
- Certain C++ functions take their input from **`cin`** (the standard input stream) which is normally the keyboard, but **`cin`** can be connected to another device. Data is output to **`cout`** (the standard output stream) which is normally the computer screen, but **`cout`** can be connected to another device.
- The standard error stream is referred to as **`cerr`**. The **`cerr`** stream (normally connected to the screen) is used for displaying error messages.
- There are many variations between different C++ implementations and different computers that make portability an elusive goal.
- C++ provides capabilities to do object-oriented programming.
- Objects are essentially reusable software components that model items in the real world. Objects are made from "blueprints" called classes.
- Single-line comments begin with **`//`**. Programmers insert comments to document programs and improve their readability. Comments do not cause the computer to perform any action when the program is run.
- The line **`#include <iostream>`** tells the C++ preprocessor to include the contents of the input/output stream header file in the program. This file contains information necessary to compile programs that use **`std::cin`** and **`std::cout`** and operators **`<<`** and **`>>`**.
- C++ programs begin executing at the function **`main`**.
- The output stream object **`std::cout`**—normally connected to the screen—is used to output data. Multiple data items can be output by concatenating stream insertion (**`<<`**) operators.
- The input stream object **`std::cin`**—normally connected to the keyboard—is used to input data. Multiple data items can be input by concatenating stream extraction (**`>>`**) operators.
- All variables in a C++ program must be declared before they can be used.
- A variable name in C++ is any valid identifier. An identifier is a series of characters consisting of letters, digits and underscores ( _ ). Identifiers cannot start with a digit. C++ identifiers can be any length; however, some systems and/or C++ implementations may impose some restrictions on the length of identifiers.
- C++ is case sensitive.

- Most calculations are performed in assignment statements.
- Every variable stored in the computer's memory has a name, a value, a type and a size.
- Whenever a new value is placed in a memory location, it replaces the previous value in that location. The previous value is destroyed.
- When a value is read from memory, the process is nondestructive, i.e., a copy of the value is read leaving the original value undisturbed in the memory location.
- C++ evaluates arithmetic expressions in a precise sequence determined by the rules of operator precedence and associativity.
- The **`if`** statement allows a program to make a decision when a certain condition is met. The format for an **`if`** statement is

  **`if`** **`(`** *condition* **`)`**
  *statement***`;`**

  If the condition is **`true`**, the statement in the body of the **`if`** is executed. If the condition is not met, i.e., the condition is **`false`**, the body statement is skipped.
- Conditions in **`if`** statements are commonly formed by using equality operators and relational operators. The result of using these operators is always simply the observation of **`true`** or **`false`**.
- The statements

  ```
  using std::cout;
  using std::cin;
  using std::endl;
  ```

  are ***`using`*** *statements* that help us eliminate the need to repeat the **`std::`** prefix. Once we include these **`using`** statements, we can write **`cout`** instead of **`std::cout`**, **`cin`** instead of **`std::cin`** and **`endl`** instead of **`std::endl`**, respectively, in the remainder of a program.
- Object-orientation is a natural way of thinking about the world and of writing computer programs.
- Objects have attributes (like size, shape, color, weight and the like) and they exhibit behaviors.
- Humans learn about objects by studying their attributes and observing their behaviors.
- Different objects can have many of the same attributes and exhibit similar behaviors.
- Object-oriented programming (OOP) models real-world objects with software counterparts. It takes advantage of class relationships where objects of a certain class have the same characteristics. It takes advantage of inheritance relationships and even multiple inheritance relationships where newly created classes are derived by inheriting characteristics of existing classes, yet contain unique characteristics of their own.
- Object-oriented programming provides an intuitive way to view the programming process, namely by modeling real-world objects, their attributes and their behaviors.
- OOP also models communication between objects via messages.
- OOP encapsulates data (attributes) and functions (behavior) into objects.
- Objects have the property of information hiding. Although objects may know how to communicate with one another across well-defined interfaces, objects normally are not allowed to know implementation details of other objects.
- Information hiding is crucial to good software engineering.
- In C and other procedural programming languages, programming tends to be action-oriented. Data is certainly important in C, but the view is that data exists primarily in support of the actions that functions perform.

- C++ programmers concentrate on creating their own user-defined types called classes. Each class contains data as well as the set of functions that manipulate the data. The data components of a class are called data members. The function components of a class are called member functions or methods.

## TERMINOLOGY

abstraction
action
action-oriented
analysis
ANSI/ISO standard C
ANSI/ISO standard C++
arithmetic and logic unit (ALU)
arithmetic operators
assembly language
assignment operator (**=**)
association
associativity of operators
attribute
attributes of an object
behavior
behaviors of an object
binary operator
body of a function
Booch, Grady
C
C++
C++ standard library
case sensitive
central processing unit (CPU)
**cerr** object
**cin** object
clarity
class
client/server computing
comma-separated list
comment (**//**)
compile error
compile-time error
compiler
component
computer
computer program
condition
**cout** object
CPU
"crafting valuable classes"
data
data member
decision
declaration
design
distributed computing
editor
encapsulation
equality operators
  **==** "is equal to"
  **!=** "is not equal to"
escape character (****)
escape sequence
execution-time error
fatal error
file server
flow of control
function
hardware
high-level language
identifier
**if** structure
information hiding
inheritance
input device
input/output (I/O)
instantiate
**int**
integer (**int**)
integer division
interface
interpreter
**iostream**
Jacobson, Ivar
left-to-right associativity
linking
loading
logic error
machine dependent
machine independent
machine language
**main**
member function
memory
memory location
message
method

modeling
multiple inheritance
modulus operator (`%`)
multiple inheritance
multiplication operator (`*`)
multiprocessor
multiprogramming
multitasking
nested parentheses
newline character (`\n`)
non-fatal error
nouns in a system specification
object
Object Management Group (OMG)
object-oriented analysis and design (OOAD)
object-oriented design (OOD)
object-oriented programming (OOP)
operand
operator
operator associativity
output device
parentheses `()`
precedence
preprocessor
primary memory
procedural programming
procedural programming language
programming language
prompt
pseudocode
Rational Software Corporation
relational operators
- `<` "is less than"
- `<=` "is less than or equal to"
- `>` "is greater than"
- `>=` "is greater than or equal to"

requirements
reserved words
"reuse, reuse and reuse"
right-to-left associativity
rules of operator precedence
Rumbaugh, James
run-time error
semicolon (`;`) statement terminator
software
software asset
software reusability
standard error object (`cerr`)
standard input object (`cin`)
standard output object (`cout`)
statement
statement terminator (`;`)
`std::cerr`
`std::cin`
`std::cout`
`std::endl`
string
structured programming
syntax error
translator program
Unified Modeling Language (UML)
user-defined type
`using`
using std::cerr;
`using std::cin;`
`using std::cout;`
`using std::endl;`
variable
variable name
variable value
verbs in a system specification
white-space characters

## COMMON PROGRAMMING ERRORS

**1.1** Errors like division-by-zero errors occur as a program runs, so these errors are called run-time errors or execution-time errors. Divide-by-zero is generally a fatal error, i.e., an error that causes the program to terminate immediately without having successfully performed its job. Non-fatal errors allow programs to run to completion, often producing incorrect results. (Note: On some systems, divide-by-zero is not a fatal error. Please see your system documentation.)

**1.2** Forgetting to include the `iostream` file in a program that inputs data from the keyboard or outputs data to the screen causes the compiler to issue an error message.

**1.3** Omitting the semicolon at the end of a statement is a syntax error. A syntax error is caused when the compiler can not recognize a statement. The compiler normally issues an error message to help the programmer locate and fix the incorrect statement. Syntax errors are violations of the language. Syntax errors are also called compile errors, compile-time errors, or compilation errors because they appear during the compilation phase.

**1.4** Attempting to use the modulus operator, **%**, with non-integer operands is a syntax error.

**1.5** A syntax error will occur if any of the operators **==**, **!=**, **>=** and **<=** appears with spaces between its pair of symbols.

**1.6** Reversing the order of the pair of operators in any of the operators **!=**, **>=** and **<=** (by writing them as **=!**, **=>** and **=<**, respectively) is normally a syntax error. In some cases, writing **!=** as **=!** will not be a syntax error, but will almost certainly be a logic error.

**1.7** Confusing the equality operator **==** with the assignment operator **=**. The equality operator should be read "is equal to" and the assignment operator should be read "gets" or "gets the value of" or "is assigned the value of." Some people prefer to read the equality operator as "double equals." As we will soon see, confusing these operators may not necessarily cause an easy-to-recognize syntax error, but may cause extremely subtle logic errors.

**1.8** Placing a semicolon immediately after the right parenthesis after the condition in an **if** structure is often a logic error (although not a syntax error). The semicolon would cause the body of the **if** structure to be empty, so the **if** structure itself would perform no action regardless of whether or not its condition is **true**. Worse yet, the original body statement of the **if** structure would now become a statement in sequence with the **if** structure and would always be executed, often causing the program to produce incorrect results.

**1.9** It is a syntax error to split an identifier by inserting white space characters (e.g., writing **main** as **ma in**).

## GOOD PROGRAMMING PRACTICES

**1.1** Write your C++ programs in a simple and straightforward manner. This is sometimes referred to as KIS ("keep it simple"). Do not "stretch" the language by trying bizarre usages.

**1.2** Read the manuals for the version of C++ you are using. Refer to these manuals frequently to be sure you are aware of the rich collection of C++ features and that you are using these features correctly.

**1.3** Your computer and compiler are good teachers. If after carefully reading your C++ language manual you are not sure how a feature of C++ works, experiment using a small "test program" and see what happens. Set your compiler options for "maximum warnings." Study each message you get when you compile your programs and correct the programs to eliminate the messages.

**1.4** Every program should begin with a comment describing the purpose of the program.

**1.5** Many programmers make the last character printed by a function a newline (**\n**). This ensures that the function will leave the screen cursor positioned at the beginning of a new line. Conventions of this nature encourage software reusability—a key goal in software development environments.

**1.6** Indent the entire body of each function one level of indentation within the braces that define the body of the function. This makes the functional structure of a program stand out and helps make programs easier to read.

**1.7** Set a convention for the size of indent you prefer then uniformly apply that convention. The tab key may be used to create indents, but tab stops may vary. We recommend using either 1/4-inch tab stops or (preferably) three spaces to form a level of indent.

**1.8** Some programmers prefer to declare each variable on a separate line. This format allows for easy insertion of a descriptive comment next to each declaration.

**1.9** Place a space after each comma (**,**) to make programs more readable.

**1.10** Choosing meaningful variable names helps a program to be "self-documenting," i.e., it becomes easier to understand the program simply by reading it rather than having to read manuals or use excessive comments.

1.11 Avoid identifiers that begin with underscores and double underscores because C++ compilers may use names like that for their own purposes internally. This will prevent names you choose from being confused with names the compilers choose.

1.12 Always place a blank line between a declaration and adjacent executable statements. This makes the declarations stand out in the program and contributes to program clarity.

1.13 If you prefer to place declarations at the beginning of a function, separate those declarations from the executable statements in that function with one blank line to highlight where the declarations end and the executable statements begin.

1.14 Place spaces on either side of a binary operator. This makes the operator stand out and makes the program more readable.

1.15 As in algebra, it is acceptable to place unnecessary parentheses in an expression to make the expression clearer. These parentheses are called redundant parentheses. Redundant parentheses are commonly used to group subexpressions in a large expression to make that expression clearer. Breaking a large statement into a sequence of shorter, simpler statements also promotes clarity.

1.16 Indent the statement in the body of an **`if`** structure to make the body of the structure stand out and to enhance program readability.

1.17 There should be no more than one statement per line in a program.

1.18 A lengthy statement may be spread over several lines. If a single statement must be split across lines, choose breaking points that make sense such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines.

1.19 Refer to the operator precedence chart when writing expressions containing many operators. Confirm that the operators in the expression are performed in the order you expect. If you are uncertain about the order of evaluation in a complex expression, break the expression into smaller statements or use parentheses to force the order, exactly as you would do in an algebraic expression. Be sure to observe that some operators such as assignment (**`=`**) associate right to left rather than left to right.

## *PERFORMANCE TIP*

1.1 Using standard library functions and classes instead of writing your own comparable versions can improve program performance because this software is carefully written to perform efficiently and correctly.

1.2 Reusing proven code components instead of writing your versions can improve program performance because these components are normally written to perform efficiently.

## *PORTABILITY TIPS*

1.1 Because C is a standardized, hardware-independent, widely available language, applications written in C can often be run with little or no modifications on a wide range of different computer systems.

1.2 Using standard library functions and classes instead of writing your own comparable versions can improve program portability because this software is included in virtually all C++ implementations.

1.3 Although it is possible to write portable programs, there are many problems among different C and C++ compilers and different computers that can make portability difficult to achieve. Simply writing programs in C and C++ does not guarantee portability. The programmer will often need to deal directly with compiler and computer variations.

**1.4** C++ allows identifiers of any length, but your system and/or C++ implementation may impose some restrictions on the length of identifiers. Use identifiers of 31 characters or fewer to ensure portability.

## SOFTWARE ENGINEERING OBSERVATIONS

**1.1** Use a "building block approach" to creating programs. Avoid reinventing the wheel. Use existing pieces where possible—this is called "software reuse" and it is central to object-oriented programming.

**1.2** When programming in C++ you will typically use the following building blocks: classes and functions from the C++ standard library, classes and functions you create yourself, and classes and functions from various popular libraries provided by third-party vendors.

**1.3** Extensive class libraries of reusable software components are available over the Internet and the World Wide Web. Many of these libraries are available at no charge.

## SELF-REVIEW EXERCISES

**1.1** Fill in the blanks in each of the following:

a) The company that popularized personal computing was _____.
b) The computer that made personal computing legitimate in business and industry was the _____.
c) Computers process data under the control of sets of instructions called computer _____.
d) The six key logical units of the computer are the _____, _____, _____, _____, _____ and the _____.
e) The three classes of languages discussed in the chapter are _____, _____ and _____.
f) The programs that translate high-level language programs into machine language are called _____.
g) C is widely known as the development language of the _____ operating system.
h) The _____ language was developed by Wirth for teaching structured programming in universities.
i) The Department of Defense developed the Ada language with a capability called _____ which allows programmers to specify that many activities can proceed in parallel.

**1.2** Fill in the blanks in each of the following sentences about the C++ environment.

a) C++ programs are normally typed into a computer using an _____ program.
b) In a C++ system, a _____ program executes before the compiler's translation phase begins.
c) The _____ program combines the output of the compiler with various library functions to produce an executable image.
d) The _____ program transfers the executable image of a C++ program from disk to memory.

**1.3** Fill in the blanks in each of the following.

a) Every C++ program begins execution at the function _____.
b) The _____ begins the body of every function and the _____ ends the body of every function.
c) Every statement ends with a _____.

d) The escape sequence **\n** represents the _____ character which causes the cursor to position to the beginning of the next line on the screen.
e) The _____ statement is used to make decisions.

**1.4** State whether each of the following is *true* or *false*. If *false*, explain why. Assume the statement **using std::cout;** is used.

a) Comments cause the computer to print the text after the **//** on the screen when the program is executed.
b) The escape sequence **\n** when output with **cout** causes the cursor to position to the beginning of the next line on the screen.
c) All variables must be declared before they are used.
d) All variables must be given a type when they are declared.
e) C++ considers the variables **number** and **NuMbEr** to be identical.
f) Declarations can appear almost anywhere in the body of a C++ function.
g) The modulus operator (**%**) can be used only with integer operands.
h) The arithmetic operators *****, **/**, **%**, **+** and **-** all have the same level of precedence.
i) A C++ program that prints three lines of output must contain three output statements using **cout**.

**1.5** Write a single C++ statement to accomplish each of the following: (Assume that **using** statements have not been used)

a) Declare the variables **c**, **thisIsAVariable**, **q76354** and **number** to be of type **int**.
b) Prompt the user to enter an integer. End your prompting message with a colon (**:**) followed by a space and leave the cursor positioned after the space.
c) Read an integer from the user at the keyboard and store the value entered in integer variable **age**.
d) If the variable **number** is not equal to **7**, print **"The variable number is not equal to 7"**.
e) Print the message **"This is a C++ program"** on one line.
f) Print the message **"This is a C++ program"** on two lines where the first line ends with **C++**.
g) Print the message **"This is a C++ program"** with each word of the message on a separate line.
h) Print the message **"This is a C++ program"** with each word separated from the next by a tab.

**1.6** Write a statement (or comment) to accomplish each of the following: (Assume that **using** statements have been used)

a) State that a program will calculate the product of three integers.
b) Declare the variables **x**, **y**, **z** and **result** to be of type **int**.
c) Prompt the user to enter three integers.
d) Read three integers from the keyboard and store them in the variables **x**, **y** and **z**.
e) Compute the product of the three integers contained in variables **x**, **y** and **z**, and assign the result to the variable **result**.
f) Print **"The product is "** followed by the value of the variable **result**.
g) Return a value from **main** indicating that the program terminated successfully.

**1.7** Using the statements you wrote in Exercise 1.6, write a complete program that calculates and displays the product of three integers. Note: you will need to write the necessary **using** statements.

**1.8** Identify and correct the errors in each of the following statements (assume that the statement **using std::cout;** is used):

a)
```
if ( c < 7 );
   cout << "c is less than 7\n";
```
b)
```
if ( c => 7 )
   cout << "c is equal to or greater than 7\n";
```

**1.9** Fill the correct "object speak" term into the blanks in each of the following:

a) Humans can look at a TV screen and see dots of color, or they can step back and see three people sitting at a conference table; this is an example of a capability called _____.
b) If we view a car as an object, the fact that the car is a convertible is a(n) attribute/behavior (pick one) _____ of the car.
c) The fact that a car can accelerate or decelerate, turn left or turn right, or go forward or backward are all examples of _____ of a car object.
d) When a new class inherits characteristics from several different existing classes, this is called _____ inheritance .
e) Objects communicate by sending each other _____.
f) Objects communicate with one another across well-defined _____.
g) Each object is ordinarily not allowed to know how other objects are implemented; this property is called _____.
h) The _____ in a system specification help the C++ programmer determine the classes that will be needed to implement the system.
i) The data components of a class are called _____ and the function components of a class are called _____.
j) An instance of a user-defined type is called a(n) _____.

## ANSWERS TO SELF-REVIEW EXERCISES

**1.1** a) Apple. b) IBM Personal Computer. c) programs. d) input unit, output unit, memory unit, arithmetic and logic unit, central processing unit, secondary storage unit. e) machine languages, assembly languages, high-level languages. f) compilers. g) UNIX. h) Pascal. i) multitasking.

**1.2** a) editor. b) preprocessor. c) linker. d) loader.

**1.3** a) **`main`**. b) Left brace (**`{`**), right brace (**`}`**). c) Semicolon. d) newline. e) **`if`**.

**1.4**

a) False. Comments do not cause any action to be performed when the program is executed. They are used to document programs and improve their readability.
b) True.
c) True.
d) True.
e) False. C++ is case sensitive, so these variables are unique.
f) True.
g) True.
h) False. The operators **`*`**, **`/`** and **`%`** have the same precedence, and the operators **`+`** and **`-`** have a lower precedence.
i) False. A single output statement using cout containing multiple **`\n`** escape sequences can print several lines.

**1.5** a)
```
int c, thisIsAVariable, q76354, number;
```
b)
```
std::cout << "Enter an integer: ";
```
c)
```
std::cin >> age;
```
d)
```
if ( number != 7 )
   std::cout << "The variable number is not equal to 7\n";
```
e)
```
std::cout << "This is a C++ program\n";
```

f) `std::cout << "This is a C++\nprogram\n";`
g) `std::cout << "This\nis\na\nC++\nprogram\n";`
h) `std::cout << "This\tis\ta\tC++\tprogram\n";`

**1.6** a) `// Calculate the product of three integers`
b) `int x, y, z, result;`
c) `cout << "Enter three integers: ";`
d) `cin >> x >> y >> z;`
e) `result = x * y * z;`
f) `cout << "The product is " << result << endl;`
g) `return 0;`

**1.7**
```
// Calculate the product of three integers
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   int x, y, z, result;

   cout << "Enter three integers: ";
   cin >> x >> y >> z;
   result = x * y * z;
   cout << "The product is " << result << endl;

   return 0;
}
```

**1.8** a) Error: Semicolon after the right parenthesis of the condition in the if statement. Correction: Remove the semicolon after the right parenthesis. Note: The result of this error is that the output statement will be executed whether or not the condition in the if statement is true. The semicolon after the right parenthesis is considered an empty statement—a statement that does nothing. We will learn more about the empty statement in the next chapter.
b) Error: The relational operator `=>`. Correction: Change `=>` to `>=`.

**1.9** a) abstraction. b) attribute. c) behaviors. d) multiple. e) messages. f) interfaces. g) information hiding. h) nouns. i) data members; member functions or methods. j) object.

## EXERCISES

**1.10** Categorize each of the following items as either hardware or software:
a) CPU
b) C++ compiler
c) ALU
d) C++ preprocessor
e) input unit
f) an editor program

**1.11** Why might you want to write a program in a machine-independent language instead of a machine-dependent language? Why might a machine-dependent language be more appropriate for writing certain types of programs?

**1.12** Fill in the blanks in each of the following statements:
a) Which logical unit of the computer receives information from outside the computer for use by the computer? _____.
b) The process of instructing the computer to solve specific problems is called _____.
c) What type of computer language uses English-like abbreviations for machine language instructions? _____.
d) Which logical unit of the computer sends information that has already been processed by the computer to various devices so that the information may be used outside the computer? _____.
e) Which logical unit of the computer retains information? _____.
f) Which logical unit of the computer performs calculations? _____.
g) Which logical unit of the computer makes logical decisions? _____.
h) The level of computer language most convenient to the programmer for writing programs quickly and easily is _____.
i) The only language a computer directly understands is called that computer's _____.
j) Which logical unit of the computer coordinates the activities of all the other logical units? _____.

**1.13** Discuss the meaning of each of the following objects:
a) **`std::cin`**
b) **`std::cout`**
c) **`std::cerr`**

**1.14** Why is so much attention today focused on object-oriented programming in general and C++ in particular?

**1.15** Fill in the blanks in each of the following:
a) _____ are used to document a program and improve its readability.
b) The object used to print information on the screen is _____.
c) A C++ statement that makes a decision is _____.
d) Calculations are normally performed by _____ statements.
e) The _____ object inputs values from the keyboard.

**1.16** Write a single C++ statement or line that accomplishes each of the following:
a) Print the message **`"Enter two numbers"`**.
b) Assign the product of variables **`b`** and **`c`** to variable **`a`**.
c) State that a program performs a sample payroll calculation (i.e., use text that helps to document a program).
d) Input three integer values from the keyboard and into integer variables **`a`**, **`b`** and **`c`**.

**1.17** State which of the following are *true* and which are *false*. If *false*, explain your answers.
a) C++ operators are evaluated from left to right.
b) The following are all valid variable names: **`_under_bar_`**, **`m928134`**, **`t5`**, **`j7`**, **`her_sales`**, **`his_account_total`**, **`a`**, **`b`**, **`c`**, **`z`**, **`z2`**.
c) The statement **`cout << "a = 5;"`**; is a typical example of an assignment statement.
d) A valid C++ arithmetic expression with no parentheses is evaluated from left to right.
e) The following are all invalid variable names: **`3g`**, **`87`**, **`67h2`**, **`h22`**, **`2h`**.

**1.18** Fill in the blanks in each of the following:
a) What arithmetic operations are on the same level of precedence as multiplication? _____.
b) When parentheses are nested, which set of parentheses is evaluated first in an arithmetic expression? _____.
c) A location in the computer's memory that may contain different values at various times throughout the execution of a program is called a _____.

**1.19** What, if anything, prints when each of the following C++ statements is performed? If nothing prints, then answer "nothing." Assume `x = 2` and `y = 3`.

a) `cout << x;`
b) `cout << x + x;`
c) `cout << "x=";`
d) `cout << "x = " << x;`
e) `cout << x + y << " = " << y + x;`
f) `z = x + y;`
g) `cin >> x >> y;`
h) `// cout << "x + y = " << x + y;`
i) `cout << "\n";`

**1.20** Which of the following C++ statements contain variables whose values are replaced?

a) `cin >> b >> c >> d >> e >> f;`
b) `p = i + j + k + 7;`
c) `cout << "variables whose values are destroyed";`
d) `cout << "a = 5";`

**1.21** Given the algebraic equation $y = ax^3 + 7$, which of the following, if any, are correct C++ statements for this equation?

a) `y = a * x * x * x + 7;`
b) `y = a * x * x * ( x + 7 );`
c) `y = ( a * x ) * x * ( x + 7 );`
d) `y = (a * x) * x * x + 7;`
e) `y = a * ( x * x * x ) + 7;`
f) `y = a * x * ( x * x + 7 );`

**1.22** State the order of evaluation of the operators in each of the following C++ statements and show the value of `x` after each statement is performed.

a) `x = 7 + 3 * 6 / 2 - 1;`
b) `x = 2 % 2 + 2 * 2 - 2 / 2;`
c) `x = ( 3 * 9 * ( 3 + ( 9 * 3 / ( 3 ) ) ) );`

**1.23** Write a program that asks the user to enter two numbers, obtains the two numbers from the user and prints the sum, product, difference, and quotient of the two numbers.

**1.24** Write a program that prints the numbers 1 to 4 on the same line with each pair of adjacent numbers separated by one space. Write the program using the following methods:

a) Using one output statement with one stream insertion operator.
b) Using one output statement with four stream insertion operators.
c) Using four output statements.

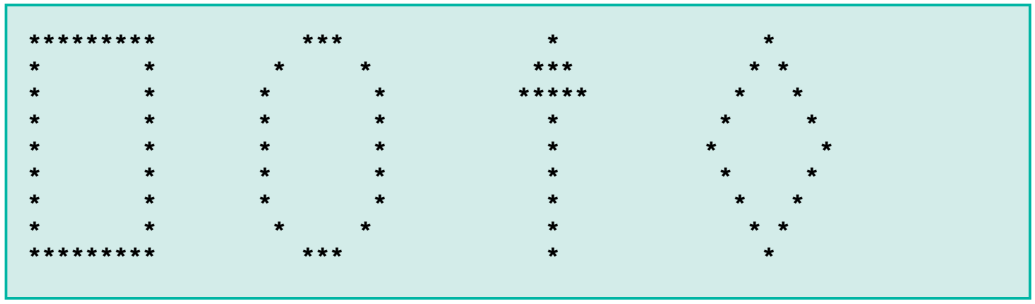
**1.25** Write a program that asks the user to enter two integers, obtains the numbers from the user, then prints the larger number followed by the words "`is larger`." If the numbers are equal, print the message "`These numbers are equal`."

**1.26** Write a program that inputs three integers from the keyboard and prints the sum, average, product, smallest and largest of these numbers. The screen dialogue should appear as follows:

```
Input three different integers: 13 27 14
Sum is 54
Average is 18
Product is 4914
Smallest is 13
Largest is 27
```

**1.27** Write a program that reads in the radius of a circle and prints the circle's diameter, circumference and area. Use the constant value 3.14159 for $\pi$. Do these calculations in output statements. (Note: In this chapter, we have discussed only integer constants and variables. In Chapter 3 we will discuss floating-point numbers, i.e., values that can have decimal points.)

**1.28** Write a program that prints a box, an oval, an arrow and a diamond as follows:



**1.29** What does the following code print?

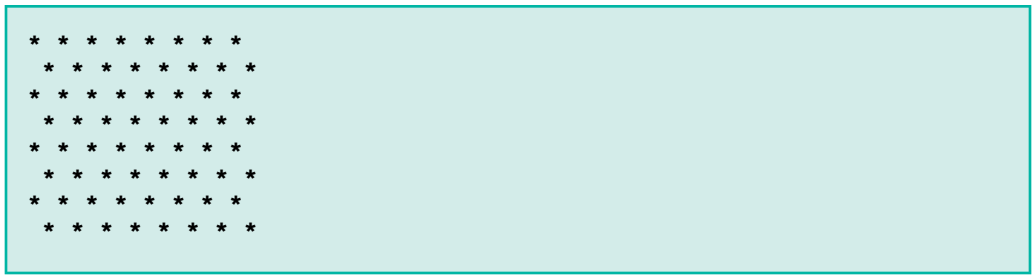
```
cout << "*\n**\n***\n****\n*****\n";
```

**1.30** Write a program that reads in five integers and determines and prints the largest and the smallest integers in the group. Use only the programming techniques you learned in this chapter.

**1.31** Write a program that reads an integer and determines and prints whether it is odd or even. (Hint: Use the modulus operator. An even number is a multiple of two. Any multiple of two leaves a remainder of zero when divided by 2.)

**1.32** Write a program that reads in two integers and determines and prints if the first is a multiple of the second. (Hint: Use the modulus operator.)

**1.33** Display a checkerboard pattern with eight output statements, then display the same pattern with as few output statements as possible.



**1.34** Distinguish between the terms fatal error and non-fatal error. Why might you prefer to experience a fatal error rather than a non-fatal error?

**1.35** Here is a peek ahead. In this chapter you learned about integers and the type **`int`**. C++ can also represent uppercase letters, lowercase letters and a considerable variety of special symbols. C++ uses small integers internally to represent each different character. The set of characters a computer uses and the corresponding integer representations for those characters is called that computer's *character set*. You can print a character by simply enclosing that character in single quotes as with

```
cout << 'A';
```

You can print the integer equivalent of a character using **static_cast** as follows:

```
cout << static_cast< int >( 'A' );
```

This is called a *cast* operation (we formally introduce casts in Chapter 2). When the preceding statement executes, it prints the value 65 (on systems that use the *ASCII character set*). Write a program that prints the integer equivalents of some uppercase letters, lowercase letters, digits and special symbols. At a minimum, determine the integer equivalents of the following: **A B C a b c 0 1 2 $ * + /** and the blank character.

**1.36** Write a program that inputs a five-digit number, separates the number into its individual digits and prints the digits separated from one another by three spaces each. (Hint: Use the integer division and modulus operators.) For example, if the user types in **42339** the program should print:

```
4   2   3   3   9
```

**1.37** Using only the techniques you learned in this chapter, write a program that calculates the squares and cubes of the numbers from 0 to 10 and uses tabs to print the following table of values:

```
number  square  cube
0       0       0
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
10      100     1000
```

**1.38** Give a brief answer to each of the following "object think" questions:

a) Why does this text choose to discuss structured programming in detail before proceeding with an in-depth treatment of object-oriented programming?
b) What are the typical steps (mentioned in the text) of an object-oriented design process?
c) How is multiple inheritance exhibited by human beings?
d) What kinds of messages do people send to one another?
e) Objects send messages to one another across well-defined interfaces. What interfaces does a car radio (object) present to its user (a person object)?

**1.39** You are probably wearing on your wrist one of the world's most common types of objects—a watch. Discuss how each of the following terms and concepts applies to the notion of a watch: object, attributes, behaviors, class, inheritance (consider, for example, an alarm clock), abstraction, modeling, messages, encapsulation, interface, information hiding, data members and member functions.