

Chapter 1

Introduction to Computers and C++

Man is still the most extraordinary computer of all.

—John F. Kennedy

Good design is good business.

—Thomas J. Watson, Founder of IBM

How wonderful it is that nobody need wait a single moment before starting to improve the world.

—Anne Frank

Objectives

In this chapter you'll learn:

- Exciting recent developments in the computer field.
- Computer hardware, software and networking basics.
- The data hierarchy.
- The different types of programming languages.
- Basic object-technology concepts.
- Some basics of the Internet and the World Wide Web.
- A typical C++ program-development environment.
- To test-drive a C++ application.
- Some key recent software technologies.
- How computers can help you make a difference.

Outline

- 1.1 Introduction
- 1.2 Computers and the Internet in Industry and Research
- 1.3 Hardware and Software
 - 1.3.1 Moore's Law
 - 1.3.2 Computer Organization
- 1.4 Data Hierarchy
- 1.5 Machine Languages, Assembly Languages and High-Level Languages
- 1.6 C++
- 1.7 Programming Languages
- 1.8 Introduction to Object Technology
- 1.9 Typical C++ Development Environment
- 1.10 Test-Driving a C++ Application
- 1.11 Operating Systems
 - 1.11.1 Windows—A Proprietary Operating System
 - 1.11.2 Linux—An Open-Source Operating System
 - 1.11.3 Apple's OS X; Apple's iOS for iPhone[®], iPad[®] and iPod Touch[®] Devices

- 1.11.4 Google's Android
- 1.12 The Internet and World Wide Web
- 1.13 Some Key Software Development Terminology
- 1.14 C++11 and the Open Source Boost Libraries
- 1.15 Keeping Up to Date with Information Technologies
- 1.16 Web Resources

Self-Review Exercises | *Answers to Self-Review Exercises* | *Exercises* | *Making a Difference* | *Making a Difference Resources*

1.1 Introduction

Welcome to C++—a powerful computer programming language that's appropriate for technically oriented people with little or no programming experience, and for experienced programmers to use in building substantial information systems. You're already familiar with the powerful tasks computers perform. Using this textbook, you'll write instructions commanding computers to perform those kinds of tasks. *Software* (i.e., the instructions you write) controls *hardware* (i.e., computers).

You'll learn *object-oriented programming*—today's key programming methodology. You'll create many *software objects* that model *things* in the real-world.

C++ is one of today's most popular software development languages. This text provides an introduction to programming in C++11—the latest version standardized through the **International Organization for Standardization (ISO)** and the **International Electrotechnical Commission (IEC)**.

In use today are more than a billion general-purpose computers and billions more cell phones, smartphones and handheld devices (such as tablet computers). According to a study by eMarketer, the number of mobile Internet users will reach approximately 134 million by 2013.^① Smartphone sales surpassed personal computer sales in 2011.^② Tablet sales are expected to account for over 20% of all personal computer sales by 2015.^③ By 2014, the smartphone applications market is expected to exceed \$40 billion.^④ This explosive growth is creating significant opportunities for programming mobile applications.

1.2 Computers and the Internet in Industry and Research

These are exciting times in the computer field. Many of the most influential and successful businesses of the last two decades are technology companies, including Apple, IBM, Hewlett Packard, Dell, Intel, Motorola, Cisco, Microsoft, Google, Amazon, Facebook, Twitter, Groupon, Foursquare, Yahoo!, eBay and many more. These companies are major employers of people who study computer science, computer engineering, information systems or related disciplines. At the time of this writing, Apple was the most valuable company in the world. Figure 1.1 provides a few examples of the ways in which computers are improving people's lives in research, industry and society.

Name	Description
Electronic health records	These might include a patient's medical history, prescriptions, immunizations, lab results, allergies, insurance information and more. Making this information available to health care providers across a secure network improves patient care, reduces the probability of error and increases overall efficiency of the health care system.

① www.circleid.com/posts/mobile_internet_users_to_reach_134_million_by_2013/.

② www.mashable.com/2012/02/03/smartphone-sales-overtake-pcs/.

③ www.forrester.com/ER/Press/Release/0,1769,1340,00.html.

④ *Inc.*, December 2010/January 2011, pages 116–123.

Name	Description
Human Genome Project	The Human Genome Project was founded to identify and analyze the 20,000+ genes in human DNA. The project used computer programs to analyze complex genetic data, determine the sequences of the billions of chemical base pairs that make up human DNA and store the information in databases which have been made available over the Internet to researchers in many fields.
AMBERTM Alert	The AMBER (America’s Missing: Broadcast Emergency Response) Alert System is used to find abducted children. Law enforcement notifies TV and radio broadcasters and state transportation officials, who then broadcast alerts on TV, radio, computerized highway signs, the Internet and wireless devices. AMBER Alert recently partnered with Facebook, whose users can “Like” AMBER Alert pages by location to receive alerts in their news feeds.
World Community Grid	People worldwide can donate their unused computer processing power by installing a free secure software program that allows the World Community Grid (www.worldcommunitygrid.org) to harness unused capacity. This computing power, accessed over the Internet, is used in place of expensive supercomputers to conduct scientific research projects that are making a difference—providing clean water to third-world countries, fighting cancer, growing more nutritious rice for regions fighting hunger and more.
Cloud computing	Cloud computing allows you to use software, hardware and information stored in the “cloud”—i.e., accessed on remote computers via the Internet and available on demand—rather than having it stored on your personal computer. These services allow you to increase or decrease resources to meet your needs at any given time, so they can be more cost effective than purchasing expensive hardware to ensure that you have enough storage and processing power to meet your needs at their peak levels. Using cloud computing services shifts the burden of managing these applications from the business to the service provider, saving businesses money.
Medical imaging	X-ray computed tomography (CT) scans, also called CAT (computerized axial tomography) scans, take X-rays of the body from hundreds of different angles. Computers are used to adjust the intensity of the X-rays, optimizing the scan for each type of tissue, then to combine all of the information to create a 3D image. MRI scanners use a technique called magnetic resonance imaging, also to produce internal images non-invasively.
GPS	Global Positioning System (GPS) devices use a network of satellites to retrieve location-based information. Multiple satellites send timestamped signals to the GPS device, which calculates the distance to each satellite based on the time the signal left the satellite and the time the signal arrived. This information is used to determine the exact location of the device. GPS devices can provide step-by-step directions and help you locate nearby businesses (restaurants, gas stations, etc.) and points of interest. GPS is used in numerous location-based Internet services such as check-in apps to help you find your friends (e.g., Foursquare and Facebook), exercise apps such as RunKeeper that track the time, distance and average speed of your outdoor jog, dating apps that help you find a match nearby and apps that dynamically update changing traffic conditions.
Robots	Robots can be used for day-to-day tasks (e.g., iRobot’s Roomba vacuuming robot), entertainment (e.g., robotic pets), military combat, deep sea and space exploration (e.g., NASA’s Mars rover Curiosity) and more. RoboEarth (www.roboearth.org) is “a World Wide Web for robots.” It allows robots to learn from each other by sharing information and thus improving their abilities to perform tasks, navigate, recognize objects and more.
E-mail, Instant Messaging, Video Chat and FTP	Internet-based servers support all of your online messaging. E-mail messages go through a mail server that also stores the messages. Instant Messaging (IM) and Video Chat apps, such as AIM, Skype, Yahoo! Messenger, Google Talk, Trillian, Microsoft’s Messenger and others allow you to communicate with others in real time by sending your messages and live video through servers. FTP (file transfer protocol) allows you to exchange files between multiple computers (e.g., a client computer such as your desktop and a file server) over the Internet.
Internet TV	Internet TV set-top boxes (such as Apple TV, Google TV and TiVo) allow you to access an enormous amount of content on demand, such as games, news, movies, television shows and more, and they help ensure that the content is streamed to your TV smoothly.

Name	Description
Streaming music services	Streaming music services (such as Pandora, Spotify, Last.fm and more) allow you listen to large catalogues of music over the web, create customized “radio stations” and discover new music based on your feedback.
Game programming	Analysts expect global video game revenues to reach \$91 billion by 2015 (www.vg247.com/2009/06/23/global-industry-analysts-predicts-gaming-market-to-reach-91-billion-by-2015/). The most sophisticated games can cost as much as \$100 million to develop. Activision’s <i>Call of Duty: Black Ops</i> —one of the best-selling games of all time—earned \$360 million in just one day (www.forbes.com/sites/insertcoin/2011/03/11/call-of-duty-black-ops-now-the-best-selling-video-game-of-all-time/)! Online <i>social gaming</i> , which enables users worldwide to compete with one another over the Internet, is growing rapidly. Zynga—creator of popular online games such as <i>Words with Friends</i> , <i>CityVille</i> and others—was founded in 2007 and already has over 300 million monthly users. To accommodate the growth in traffic, Zynga is adding nearly 1,000 servers each week (techcrunch.com/2010/09/22/zynga-moves-1-petabyte-of-data-daily-adds-1000-servers-a-week/)!

Fig. 1.1 | A few uses for computers.

1.3 Hardware and Software

Computers can perform calculations and make logical decisions phenomenally faster than human beings can. Many of today’s personal computers can perform billions of calculations in one second—more than a human can perform in a lifetime. *Supercomputers* are already performing *thousands of trillions (quadrillions)* of instructions per second! IBM’s Sequoia supercomputer can perform over 16 quadrillion calculations per second (16.32 *petaflops*)!^① To put that in perspective, *the IBM Sequoia supercomputer can perform in one second about 1.5 million calculations for every person on the planet!* And—these “upper limits” are growing quickly!

Computers process data under the control of sequences of instructions called **computer programs**. These programs guide the computer through ordered actions specified by people called computer **programmers**. The programs that run on a computer are referred to as **software**. In this book, you’ll learn a key programming methodology that’s enhancing programmer productivity, thereby reducing software development costs—*object-oriented programming*.

A computer consists of various devices referred to as hardware (e.g., the keyboard, screen, mouse, hard disks, memory, DVD drives and processing units). Computing costs are *dropping dramatically*, owing to rapid developments in hardware and software technologies. Computers that might have filled large rooms and cost millions of dollars decades ago are now inscribed on silicon chips smaller than a fingernail, costing perhaps a few dollars each. Ironically, silicon is one of the most abundant materials on Earth—it’s an ingredient in common sand. Silicon-chip technology has made computing so economical that computers have become a commodity.

1.3.1 Moore’s Law

Every year, you probably 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 costs of hardware supporting these technologies. For many decades, hardware costs have fallen rapidly. Every year or two, the capacities of computers have approximately *doubled* inexpensively. This remarkable trend often is called

① www.top500.org.

Moore’s Law, named for the person who identified it in the 1960s, Gordon Moore, co-founder of Intel—the leading manufacturer of the processors in today’s computers and embedded systems. Moore’s Law and related observations apply especially to the amount of memory that computers have for programs, the amount of secondary storage (such as disk storage) they have to hold programs and data over longer periods of time, and their processor speeds—the speeds at which computers execute their programs (i.e., do their work). Similar growth has occurred in the communications field, in which costs have plummeted as enormous demand for communications bandwidth (i.e., information-carrying capacity) has attracted intense competition. We know of no other fields in which technology improves so quickly and costs fall so rapidly. Such phenomenal improvement is truly fostering the *Information Revolution*.

1.3.2 Computer Organization

Regardless of differences in *physical* appearance, computers can be envisioned as divided into various **logical units** or sections (Fig. 1.2).

Logical unit	Description
Input unit	This “receiving” section obtains information (data and computer programs) from input devices and places it at the disposal of the other units for processing. Most information is entered into computers through keyboards, touch screens and mouse devices. Other forms of input include receiving voice commands, scanning images and barcodes, reading from secondary storage devices (such as hard drives, DVD drives, Blu-ray Disc™ drives and USB flash drives—also called “thumb drives” or “memory sticks”), receiving video from a webcam and having your computer receive information from the Internet (such as when you stream videos from YouTube™ or download e-books from Amazon). Newer forms of input include position data from a GPS device, and motion and orientation information from an accelerometer in a smartphone or game controller (such as Microsoft® Kinect™, Wii™ Remote and Sony’s PlayStation® Move).
Output unit	This “shipping” section takes information that the computer has processed and places it on various output devices to make it available for use outside the computer. Most information that’s output from computers today is displayed on screens, printed on paper (“going green” discourages this), played as audio or video on PCs and media players (such as Apple’s popular iPods) and giant screens in sports stadiums, transmitted over the Internet or used to control other devices, such as robots and “intelligent” appliances.
Memory unit	This rapid-access, relatively low-capacity “warehouse” section retains information that has been entered through the input unit, making it immediately available for processing when needed. The memory unit also retains processed information until it can be placed on output devices by the output unit. Information in the memory unit is <i>volatile</i> —it’s typically lost when the computer’s power is turned off. The memory unit is often called either memory or primary memory . Main memories on desktop and notebook computers commonly contain as much as 16 GB (GB stands for gigabytes; a gigabyte is approximately one billion bytes).
Arithmetic and logic unit (ALU)	This “manufacturing” section performs <i>calculations</i> , such as addition, subtraction, multiplication and division. It also contains the <i>decision</i> mechanisms that allow the computer, for example, to compare two items from the memory unit to determine whether they’re equal. In today’s systems, the ALU is usually implemented as part of the next logical unit, the CPU.
Central processing unit (CPU)	This “administrative” section coordinates and supervises 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. Many of today’s computers have multiple CPUs and, hence, can perform many operations simultaneously. A multi-core processor implements multiple processors on a single integrated-circuit chip—a <i>dual-core processor</i> has two CPUs and a <i>quad-core processor</i> has four CPUs. Today’s desktop computers have processors that can execute billions of instructions per second.

Logical unit	Description
Secondary storage unit	This is the long-term, high-capacity “warehousing” section. Programs or data not actively being used by the other units normally are placed on secondary storage devices (e.g., your <i>hard drive</i>) until they’re again needed, possibly hours, days, months or even years later. Information on secondary storage devices is <i>persistent</i> —it’s preserved even when the computer’s power is turned off. Secondary storage information takes much longer to access than information in primary memory, but the cost per unit of secondary storage is much less than that of primary memory. Examples of secondary storage devices include CD drives, DVD drives and flash drives, some of which can hold up to 768 GB. Typical hard drives on desktop and notebook computers can hold up to 2 TB (TB stands for terabytes; a terabyte is approximately one trillion bytes).

Fig. 1.2 | Logical units of a computer.

1.4 Data Hierarchy

Data items processed by computers form a **data hierarchy** that becomes larger and more complex in structure as we progress from bits to characters to fields, and so on. Figure 1.3 illustrates a portion of the data hierarchy. Figure 1.4 summarizes the data hierarchy’s levels.

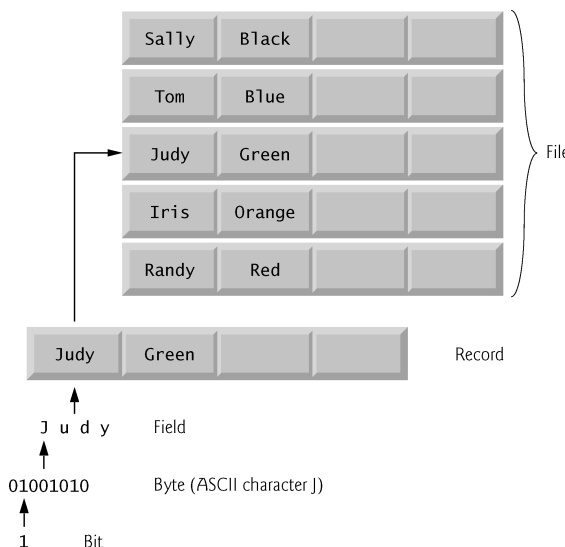


Fig. 1.3 | Data hierarchy.

Level	Description
Bits	The smallest data item in a computer can assume the value 0 or the value 1. Such a data item is called a bit (short for “binary digit”—a digit that can assume one of two values). It’s remarkable that the impressive functions performed by computers involve only the simplest manipulations of 0s and 1s— <i>examining a bit’s value</i> , <i>setting a bit’s value</i> and <i>reversing a bit’s value</i> (from 1 to 0 or from 0 to 1).
Characters	It’s tedious for people to work with data in the low-level form of bits. Instead, they prefer to work with <i>decimal digits</i> (0–9), <i>letters</i> (A–Z and a–z), and <i>special symbols</i> (e.g., \$, @, %, &, *, (,), -, +, ", ., ? and /). Digits, letters and special symbols are known as characters . The computer’s character set is the set of all the characters used to write programs and represent data items. Computers process only 1s and 0s, so every character is represented as a pattern of 1s and 0s. The Unicode [®] character set contains characters for many of the world’s languages. C++ supports several character sets, including 16-bit Unicode [®] characters that are composed of two bytes , each composed of eight bits. See Appendix B for more information on the ASCII (American Standard Code for Information Interchange) character set—the popular subset of Unicode that represents uppercase and lowercase letters, digits and some common special characters.

Level	Description
Fields	Just as characters are composed of bits, fields are composed of characters or bytes. A field is a group of characters or bytes that conveys meaning. For example, a field consisting of uppercase and lowercase letters could be used to represent a person's name, and a field consisting of decimal digits could represent a person's age.
Records	Several related fields can be used to compose a record . In a payroll system, for example, the record for an employee might consist of the following fields (possible types for these fields are shown in parentheses): <ul style="list-style-type: none"> • Employee identification number (a whole number) • Name (a string of characters) • Address (a string of characters) • Hourly pay rate (a number with a decimal point) • Year-to-date earnings (a number with a decimal point) • Amount of taxes withheld (a number with a decimal point) <p>Thus, a record is a group of related fields. In the preceding example, all the fields belong to the same employee. A company might have many employees and a payroll record for each one.</p>
Files	A file is a group of related records. [<i>Note: More generally, a file contains arbitrary data in arbitrary formats. In some operating systems, a file is viewed simply as a <i>sequence of bytes</i>—any organization of the bytes in a file, such as organizing the data into records, is a view created by the application programmer.</i>] It's not unusual for an organization to have many files, some containing billions, or even trillions, of characters of information.
Database	A database is an electronic collection of data that's organized for easy access and manipulation. The most popular database model is the relational database in which data is stored in simple <i>tables</i> . A table includes <i>records</i> and <i>fields</i> . For example, a table of students might include first name, last name, major, year, student ID number and grade point average. The data for each student is a record, and the individual pieces of information in each record are the fields. You can search, sort and manipulate the data based on its relationship to multiple tables or databases. For example, a university might use data from the student database in combination with databases of courses, on-campus housing, meal plans, etc.

Fig. 1.4 | Levels of the data hierarchy.

1.5 Machine Languages, Assembly Languages and High-Level Languages

Programmers write instructions in various programming languages, some directly understandable by computers and others requiring intermediate *translation* steps.

Machine Languages

Any computer can directly understand only its own **machine language** (also called *machine code*), defined by its hardware architecture. Machine languages generally consist of numbers (ultimately reduced to 1s and 0s). Such languages are cumbersome for humans.

Assembly Languages

Programming in machine language was simply too slow and tedious for most programmers. Instead, they began using English-like *abbreviations* to represent elementary operations. These abbreviations formed the basis of **assembly languages**. *Translator programs* called **assemblers** were developed to convert assembly-language programs to machine language. Although assembly-language code is clearer to humans, it's incomprehensible to computers until translated to machine language.

High-Level Languages

To speed up the programming process further, **high-level languages** were developed in which single statements could be written to accomplish substantial tasks. High-level languages, such as C++, Java, C# and Visual Basic, allow you to write instructions that look more like everyday English and contain commonly used mathematical

expressions. Translator programs called **compilers** convert high-level language programs into machine language.

The process of compiling a large high-level language program into machine language can take a considerable amount of computer time. **Interpreter** programs were developed to execute high-level language programs directly (without the need for compilation), although more slowly than compiled programs. **Scripting languages** such as the popular web languages JavaScript and PHP are processed by interpreters.



Performance Tip 1.1

Interpreters have an advantage over compilers in Internet scripting. An interpreted program can begin executing as soon as it's downloaded to the client's machine, without needing to be compiled before it can execute. On the downside, interpreted scripts generally run slower than compiled code.

1.6 C++

C++ evolved from C, which was developed by Dennis Ritchie at Bell Laboratories. C is available for most computers and is hardware independent. With careful design, it's possible to write C programs that are **portable** to most computers.

The widespread use of C with various kinds of computers (sometimes called **hardware platforms**) unfortunately led to many variations. A standard version of C was needed. The American National Standards Institute (ANSI) cooperated with the International Organization for Standardization (ISO) to standardize C worldwide; the joint standard document was published in 1990 and is referred to as *ANSI/ISO 9899: 1990*.

C11 is the latest ANSI standard for the C programming language. It was developed to evolve the C language to keep pace with increasingly powerful hardware and ever more demanding user requirements. C11 also makes C more consistent with C++. For more information on C and C11, see our book *C How to Program, 7/e* and our C Resource Center (located at www.deitel.com/C).

C++, an extension of C, was developed by Bjarne Stroustrup in 1979 at Bell Laboratories. Originally called “C with Classes”, it was renamed to C++ in the early 1980s. C++ provides a number of features that “spruce up” the C language, but more importantly, it provides capabilities for object-oriented programming.

You'll begin developing customized, reusable classes and objects in Chapter 3, Introduction to Classes, Objects and Strings. The book is object oriented, where appropriate, from the start and throughout the text.

We also provide an *optional* automated teller machine (ATM) case study in Chapters 25–26, which contains a complete C++ implementation. The case study presents a carefully paced introduction to object-oriented design using the UML—an industry standard graphical modeling language for developing object-oriented systems. We guide you through a friendly design experience intended for the novice.

C++ Standard Library

C++ programs consist of pieces called **classes** and **functions**. You can program each piece yourself, but most C++ programmers take advantage of the rich collections of 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; the second is learning how to use the classes and functions in the C++ Standard Library. We discuss many of these classes and functions. P. J. Plauger's book, *The Standard C Library* (Upper Saddle River, NJ: Prentice Hall PTR, 1992), is a must read for programmers who need a deep understanding of the ANSI C library functions included in C++. Many special-purpose class libraries are supplied by independent software vendors.



Software Engineering Observation 1.1

Use a “building-block” approach to create programs. Avoid reinventing the wheel. Use existing pieces wherever possible. Called **software reuse**, this practice is central to object-oriented programming.



Software Engineering Observation 1.2

When programming in C++, you typically will use the following building blocks: classes and functions from the C++ Standard Library, classes and functions you and your colleagues create and classes and functions from various popular third-party libraries.

The advantage of creating your own functions and classes is that you’ll know exactly how they work. You’ll 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 1.2

Using C++ Standard Library functions and classes instead of writing your own versions can improve program performance, because they’re written carefully to perform efficiently. This technique also shortens program development time.



Portability Tip 1.1

Using C++ Standard Library functions and classes instead of writing your own improves program portability, because they’re included in every C++ implementation.

1.7 Programming Languages

In this section, we provide brief comments on several popular programming languages (Fig. 1.5).

Programming language	Description
Fortran	Fortran (FORmula TRANslator) was developed by IBM Corporation in the mid-1950s to be used for scientific and engineering applications that require complex mathematical computations. It’s still widely used and its latest versions support object-oriented programming.
COBOL	COBOL (COMmon Business Oriented Language) was developed in the late 1950s by computer manufacturers, the U.S. government and industrial computer users based on a language developed by Grace Hopper, a career U.S. Navy officer and computer scientist. COBOL is still widely used for commercial applications that require precise and efficient manipulation of large amounts of data. Its latest version supports object-oriented programming.
Pascal	Research in the 1960s resulted in <i>structured programming</i> —a disciplined approach to writing programs that are clearer, easier to test and debug and easier to modify than large programs produced with previous techniques. One of the more tangible results of this research was the development of Pascal by Professor Niklaus Wirth in 1971. It was designed for teaching structured programming and was popular in college courses for several decades.
Ada	Ada, based on Pascal, was developed under the sponsorship of the U.S. Department of Defense (DOD) during the 1970s and early 1980s. The DOD wanted a single language that would fill most of its needs. The Pascal-based language was named after Lady Ada Lovelace, daughter of the poet Lord Byron. She’s credited with writing the world’s first computer program in the early 1800s (for the Analytical Engine mechanical computing device designed by Charles Babbage). Ada also supports object-oriented programming.
BASIC	BASIC was developed in the 1960s at Dartmouth College to familiarize novices with programming techniques. Many of its latest versions are object oriented.

Programming language	Description
C	C was implemented in 1972 by Dennis Ritchie at Bell Laboratories. It initially became widely known as the UNIX operating system's development language. Today, most of the code for general-purpose operating systems is written in C or C++.
Objective-C	Objective-C is an object-oriented language based on C. It was developed in the early 1980s and later acquired by NeXT, which in turn was acquired by Apple. It has become the key programming language for the OS X operating system and all iOS-powered devices (such as iPods, iPhones and iPads).
Java	Sun Microsystems in 1991 funded an internal corporate research project led by James Gosling, which resulted in the C++-based object-oriented programming language called Java. A key goal of Java is to be able to write programs that will run on a great variety of computer systems and computer-control devices. This is sometimes called "write once, run anywhere." Java is used 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 (e.g., smartphones, tablets, television set-top boxes, appliances, automobiles and more) and for many other purposes. Java is also the key language for developing Android smartphone and tablet apps.
Visual Basic	Microsoft's Visual Basic language was introduced in the early 1990s to simplify the development of Microsoft Windows applications. Its latest versions support object-oriented programming.
C#	Microsoft's three object-oriented primary programming languages are Visual Basic (based on the original Basic), Visual C++ (based on C++) and C# (based on C++ and Java, and developed for integrating the Internet and the web into computer applications).
PHP	PHP is an object-oriented, "open-source" (see Section 1.11.2) "scripting" language supported by a community of users and developers and is used by numerous websites including Wikipedia and Facebook. PHP is platform independent—implementations exist for all major UNIX, Linux, Mac and Windows operating systems. PHP also supports many databases, including MySQL.
Perl	Perl (Practical Extraction and Report Language), one of the most widely used object-oriented scripting languages for web programming, was developed in 1987 by Larry Wall. It features rich text-processing capabilities and flexibility.
Python	Python, another object-oriented scripting language, was released publicly in 1991. Developed by Guido van Rossum of the National Research Institute for Mathematics and Computer Science in Amsterdam (CWI), Python draws heavily from Modula-3—a systems programming language. Python is "extensible"—it can be extended through classes and programming interfaces.
JavaScript	JavaScript is the most widely used scripting language. It's primarily used to add programmability to web pages—for example, animations and interactivity with the user. It's provided with all major web browsers.
Ruby on Rails	Ruby—created in the mid-1990s by Yukihiro Matsumoto—is an open-source, object-oriented programming language with a simple syntax that's similar to Perl and Python. Ruby on Rails combines the scripting language Ruby with the Rails web application framework developed by 37Signals. Their book, <i>Getting Real</i> (available free at gettingreal.37signals.com/toc.php), is a must read for web developers. Many Ruby on Rails developers have reported productivity gains over other languages when developing database-intensive web applications. Ruby on Rails was used to build Twitter's user interface.
Scala	Scala (www.scala-lang.org/node/273)—short for "scalable language"—was designed by Martin Odersky, a professor at École Polytechnique Fédérale de Lausanne (EPFL) in Switzerland. Released in 2003, Scala uses both the object-oriented programming and functional programming paradigms and is designed to integrate with Java. Programming in Scala can reduce the amount of code in your applications significantly. Twitter and Foursquare use Scala.

Fig. 1.5 | Some other programming languages.

1.8 Introduction to Object Technology

Building software quickly, correctly and economically remains an elusive goal at a time when demands for new

and more powerful software are soaring. *Objects*, or more precisely—as we’ll see in Chapter 3—the *classes* objects come from, are essentially *reusable* software components. There are date objects, time objects, audio objects, video objects, automobile objects, people objects, etc. Almost any *noun* can be reasonably represented as a software object in terms of *attributes* (e.g., name, color and size) and *behaviors* (e.g., calculating, moving and communicating). Software developers have discovered that using a modular, object-oriented design-and-implementation approach can make software-development groups much more productive than was possible with earlier techniques—object-oriented programs are often easier to understand, correct and modify.

The Automobile as an Object

Let’s begin with a simple analogy. Suppose you want to *drive a car and make it go faster by pressing its accelerator pedal*. What must happen before you can do this? Well, before you can drive a car, someone has to *design* it. A car typically begins as engineering drawings, similar to the *blueprints* that describe the design of a house. These drawings include the design for an accelerator pedal. The pedal *hides* from the driver the complex mechanisms that actually make the car go faster, just as the brake pedal hides the mechanisms that slow the car, and the steering wheel *hides* the mechanisms that turn the car. This enables people with little or no knowledge of how engines, braking and steering mechanisms work to drive a car easily.

Before you can drive a car, it must be *built* from the engineering drawings that describe it. A completed car has an *actual* accelerator pedal to make the car go faster, but even that’s not enough—the car won’t accelerate on its own (hopefully!), so the driver must *press* the pedal to accelerate the car.

Member Functions and Classes

Let’s use our car example to introduce some key object-oriented programming concepts. Performing a task in a program requires a **member function**. The member function houses the program statements that actually perform its task. It hides these statements from its user, just as the accelerator pedal of a car hides from the driver the mechanisms of making the car go faster. In C++, we create a program unit called a **class** to house the set of member functions that perform the class’s tasks. For example, a class that represents a bank account might contain one member function to *deposit* money to an account, another to *withdraw* money from an account and a third to *inquire* what the account’s current balance is. A class is similar in concept to a car’s engineering drawings, which house the design of an accelerator pedal, steering wheel, and so on.

Instantiation

Just as someone has to *build a car* from its engineering drawings before you can actually drive a car, you must *build an object* from a class before a program can perform the tasks that the class’s methods define. The process of doing this is called *instantiation*. An object is then referred to as an **instance** of its class.

Reuse

Just as a car’s engineering drawings can be *reused* many times to build many cars, you can *reuse* a class many times to build many objects. Reuse of existing classes when building new classes and programs saves time and effort. Reuse also helps you build more reliable and effective systems, because existing classes and components often have gone through extensive *testing*, *debugging* and *performance tuning*. Just as the notion of *interchangeable parts* was crucial to the Industrial Revolution, reusable classes are crucial to the software revolution that has been spurred by object technology.

Messages and Member Function Calls

When you drive a car, pressing its gas pedal sends a *message* to the car to perform a task—that is, to go faster. Similarly, you *send messages to an object*. Each message is implemented as a **member function call** that tells a

member function of the object to perform its task. For example, a program might call a particular bank account object's *deposit* member function to increase the account's balance.

Attributes and Data Members

A car, besides having capabilities to accomplish tasks, also has *attributes*, such as its color, its number of doors, the amount of gas in its tank, its current speed and its record of total miles driven (i.e., its odometer reading). Like its capabilities, the car's attributes are represented as part of its design in its engineering diagrams (which, for example, include an odometer and a fuel gauge). As you drive an actual car, these attributes are carried along with the car. Every car maintains its *own* attributes. For example, each car knows how much gas is in its own gas tank, but *not* how much is in the tanks of *other* cars.

An object, similarly, has attributes that it carries along as it's used in a program. These attributes are specified as part of the object's class. For example, a bank account object has a *balance attribute* that represents the amount of money in the account. Each bank account object knows the balance in the account it represents, but *not* the balances of the *other* accounts in the bank. Attributes are specified by the class's **data members**.

Encapsulation

Classes **encapsulate** (i.e., wrap) attributes and member functions into objects—an object's attributes and member functions are intimately related. Objects may communicate with one another, but they're normally not allowed to know how other objects are implemented—implementation details are *hidden* within the objects themselves. This **information hiding**, as we'll see, is crucial to good software engineering.

Inheritance

A new class of objects can be created quickly and conveniently by **inheritance**—the new class absorbs the characteristics of an existing class, possibly customizing them and adding unique characteristics of its own. In our car analogy, an object of class “convertible” certainly *is an* object of the more *general* class “automobile,” but more *specifically*, the roof can be raised or lowered.

Object-Oriented Analysis and Design (OOAD)

Soon you'll be writing programs in C++. How will you create the **code** (i.e., the program instructions) for your programs? Perhaps, like many programmers, you'll simply turn on your computer and start typing. This approach may work for small programs (like the ones we present in the early chapters of the book), but what if you were asked to create a software system to control thousands of automated teller machines for a major bank? Or suppose you were asked to work on a team of thousands of software developers building the next U.S. air traffic control system? For projects so large and complex, you should not simply sit down and start writing programs.

To create the best solutions, you should follow a detailed **analysis** process for determining your project's **requirements** (i.e., defining *what* the system is supposed to do) and developing a **design** that satisfies them (i.e., deciding *how* the system should do it). Ideally, you'd go through this process and carefully review the design (and have your design reviewed by other software professionals) before writing any code. If this process involves analyzing and designing your system from an object-oriented point of view, it's called an **object-oriented analysis and design (OOAD) process**. Languages like C++ are object oriented. Programming in such a language, called **object-oriented programming (OOP)**, allows you to implement an object-oriented design as a working system.

The UML (Unified Modeling Language)

Although many different OOAD processes exist, a single graphical language for communicating the results of *any* OOAD process has come into wide use. This language, known as the Unified Modeling Language (UML),

is now the most widely used graphical scheme for modeling object-oriented systems. We present our first UML diagrams in Chapters 3 and 4, then use them in our deeper treatment of object-oriented programming through Chapter 12. In our *optional* ATM Software Engineering Case Study in Chapters 25–26 we present a simple subset of the UML’s features as we guide you through an object-oriented design experience.

1.9 Typical C++ Development Environment

C++ systems generally consist of three parts: a program development environment, the language and the C++ Standard Library. C++ programs typically go through six phases: edit, preprocess, compile, link, load and execute. The following discussion explains a typical C++ program development environment.

Phase 1: Editing a Program

Phase 1 consists of editing a file with an *editor program*, normally known simply as an *editor* (Fig. 1.6). You type a C++ program (typically referred to as **source code**) using the editor, make any necessary corrections and save the program on a secondary storage device, such as your hard drive. C++ source code filenames often end with the .cpp, .cxx, .cc or .C extensions (note that C is in uppercase) which indicate that a file contains C++ source code. See the documentation for your C++ compiler for more information on file-name extensions.

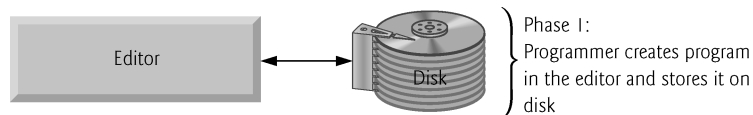


Fig. 1.6 | Typical C++ development environment—editing phase.

Two editors widely used on Linux systems are vi and emacs. C++ software packages for Microsoft Windows such as Microsoft Visual C++ (microsoft.com/express) have editors integrated into the programming environment. You can also use a simple text editor, such as Notepad in Windows, to write your C++ code.

For organizations that develop substantial information systems, **integrated development environments (IDEs)** are available from many major software suppliers. IDEs provide tools that support the software-development process, including editors for writing and editing programs and debuggers for locating **logic errors**—errors that cause programs to execute incorrectly. Popular IDEs include Microsoft[®] Visual Studio 2012 Express Edition, Dev C++, NetBeans, Eclipse, Apple’s Xcode and CodeLite.

Phase 2: Preprocessing a C++ Program

In Phase 2, you give the command to **compile** the program (Fig. 1.7). In a C++ system, a **preprocessor** program executes automatically before the compiler’s translation phase begins (so we call preprocessing Phase 2 and compiling Phase 3). The C++ preprocessor obeys commands called **preprocessing directives**, which indicate that certain manipulations are to be performed on the program before compilation. These manipulations usually include other text files to be compiled, and perform various text replacements. The most common preprocessing directives are discussed in the early chapters; a detailed discussion of preprocessor features appears in Appendix E, Preprocessor.

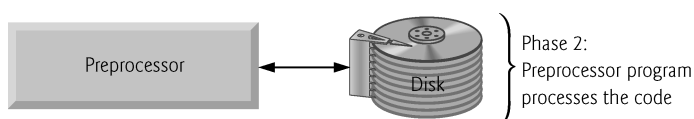


Fig. 1.7 | Typical C++ development environment—preprocessor phase.

Phase 3: Compiling a C++ Program

In Phase 3, the compiler translates the C++ program into machine-language code—also referred to as object code (Fig. 1.8).

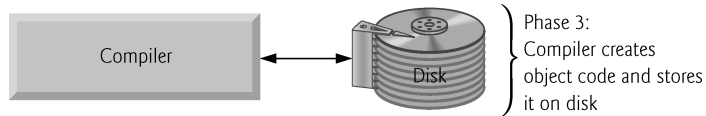


Fig. 1.8 | Typical C++ development environment—compilation phase.

Phase 4: Linking

Phase 4 is called **linking**. C++ programs typically contain references to functions and data defined elsewhere, such as in the standard libraries or in the private libraries of groups of programmers working on a particular project (Fig. 1.9). 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 program** (with no missing pieces). If the program compiles and links correctly, an executable image is produced.

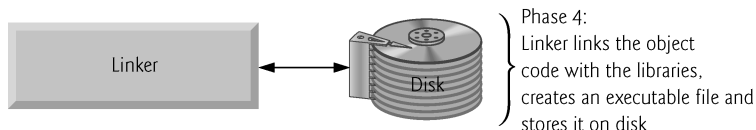


Fig. 1.9 | Typical C++ development environment—linking phase.

Phase 5: Loading

Phase 5 is called **loading**. Before a program can be executed, it must first be placed in memory (Fig. 1.10). 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.

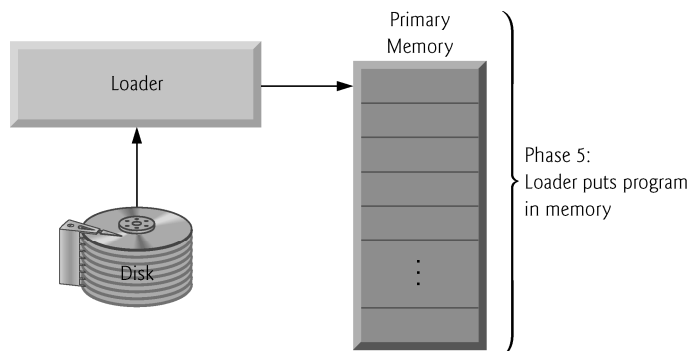


Fig. 1.10 | Typical C++ development environment—loading phase.

Phase 6: Execution

Finally, the computer, under the control of its CPU, **executes** the program one instruction at a time (Fig. 1.11). Some modern computer architectures can execute several instructions in parallel.

Problems That May Occur at Execution Time

Programs might not work on the first try. Each of the preceding phases can fail because of various errors that we’ll discuss throughout this book. For example, an executing program might try to divide by zero (an illegal

operation for integer arithmetic in C++). This would cause the C++ program to display an error message. If this occurred, you'd have to return to the edit phase, make the necessary corrections and proceed through the remaining phases again to determine that the corrections fixed the problem(s). [Note: Most programs in C++ input 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 redirected 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 redirected 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.

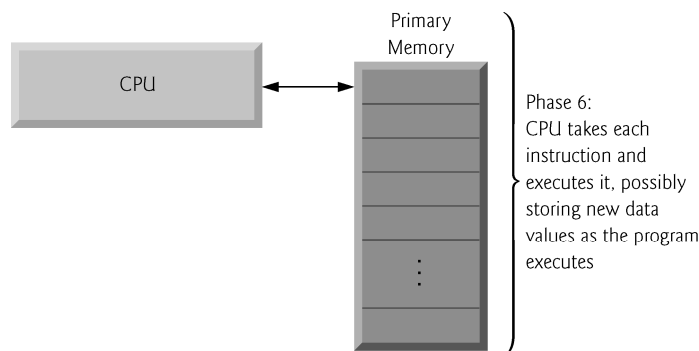


Fig. 1.11 | Typical C++ development environment—execution phase.



Common Programming Error 1.1

Errors such as division by zero occur as a program runs, so they're called **runtime errors** or **execution-time errors**. **Fatal runtime errors** cause programs to terminate immediately without having successfully performed their jobs. **Nonfatal runtime errors** allow programs to run to completion, often producing incorrect results.

1.10 Test-Driving a C++ Application

In this section, you'll run and interact with your first C++ application. You'll begin by running an entertaining guess-the-number game, which picks a number from 1 to 1000 and prompts you to guess it. If your guess is correct, the game ends. If your guess is not correct, the application indicates whether your guess is higher or lower than the correct number. There is no limit on the number of guesses you can make. [Note: For this test drive only, we've modified this application from the exercise you'll be asked to create in Chapter 6, Functions and an Introduction to Recursion. Normally this application randomly selects the correct answer as you execute the program. The modified application uses the same correct answer every time the program executes (though this may vary by compiler), so you can use the *same* guesses we use in this section and see the *same* results as we walk you through interacting with your first C++ application.]

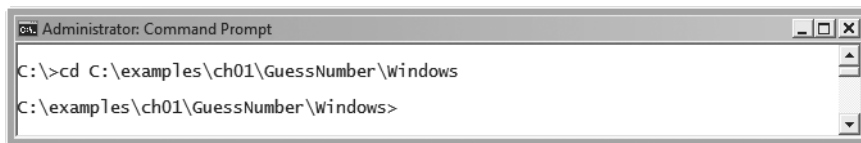
We'll demonstrate running a C++ application using the Windows Command Prompt and a shell on Linux. The application runs similarly on both platforms. Many development environments are available in which you can compile, build and run C++ applications, such as GNU C++, Microsoft Visual C++, Apple Xcode, Dev C++, CodeLite, NetBeans, Eclipse etc. Consult your instructor for information on your specific development environment.

In the following steps, you'll run the application and enter various numbers to guess the correct number.

The elements and functionality that you see in this application are typical of those you'll learn to program in this book. We use fonts to distinguish between features you see on the screen (e.g., the Command Prompt) and elements that are not directly related to the screen. We emphasize screen features like titles and menus (e.g., the File menu) in a semibold sans-serif Helvetica font and emphasize filenames, text displayed by an application and values you should enter into an application (e.g., `GuessNumber` or `500`) in a sans-serif Lucida font. As you've noticed, the **defining occurrence** of each term is set in blue, bold type. For the figures in this section, we point out significant parts of the application. To make these features more visible, we've modified the background color of the Command Prompt window (for the Windows test drive only). To modify the Command Prompt colors on your system, open a Command Prompt by selecting `Start > All Programs > Accessories > Command Prompt`, then right click the title bar and select `Properties`. In the "Command Prompt" Properties dialog box that appears, click the `Colors` tab, and select your preferred text and background colors.

Running a C++ Application from the Windows Command Prompt

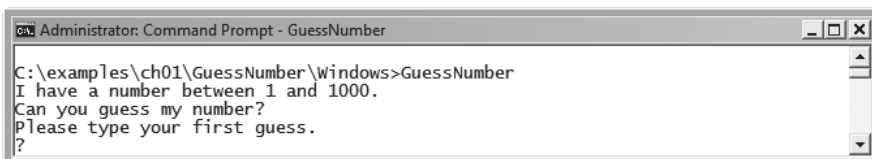
1. *Checking your setup.* It's important to read the Before You Begin section at www.deitel.com/books/cpphtp9/ to make sure that you've copied the book's examples to your hard drive correctly.
2. *Locating the completed application.* Open a Command Prompt window. To change to the directory for the completed `GuessNumber` application, type `cd C:\examples\ch01\GuessNumber\Windows`, then press `Enter` (Fig. 1.12). The command `cd` is used to change directories.



```
Administrator: Command Prompt
C:\>cd C:\examples\ch01\GuessNumber\Windows
C:\examples\ch01\GuessNumber\Windows>
```

Fig. 1.12 | Opening a Command Prompt window and changing the directory.

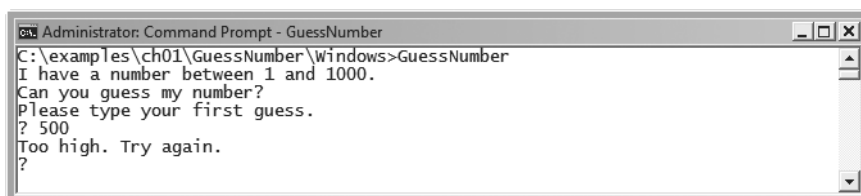
3. *Running the `GuessNumber` application.* Now that you are in the directory that contains the `GuessNumber` application, type the command `GuessNumber` (Fig. 1.13) and press `Enter`. [Note: `GuessNumber.exe` is the actual name of the application; however, Windows assumes the `.exe` extension by default.]



```
Administrator: Command Prompt - GuessNumber
C:\examples\ch01\GuessNumber\Windows>GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

Fig. 1.13 | Running the `GuessNumber` application.

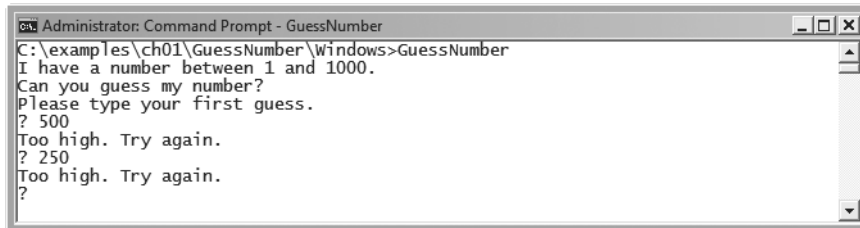
4. *Entering your first guess.* The application displays "Please type your first guess.", then displays a question mark (?) as a prompt on the next line (Fig. 1.13). At the prompt, enter `500` (Fig. 1.14).



```
Administrator: Command Prompt - GuessNumber
C:\examples\ch01\GuessNumber\Windows>GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

Fig. 1.14 | Entering your first guess.

5. *Entering another guess.* The application displays "Too high. Try again.", meaning that the value you entered is greater than the number the application chose as the correct guess. So, you should enter a lower number for your next guess. At the prompt, enter 250 (Fig. 1.15). The application again displays "Too high. Try again.", because the value you entered is still greater than the number that the application chose as the correct guess.



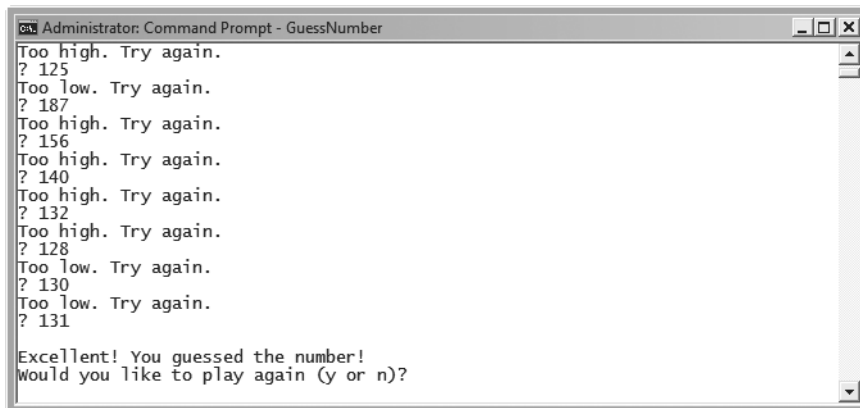
```

Administrator: Command Prompt - GuessNumber
C:\examples\ch01\GuessNumber\Windows>GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too high. Try again.
?

```

Fig. 1.15 | Entering a second guess and receiving feedback.

6. *Entering additional guesses.* Continue to play the game by entering values until you guess the correct number. The application will display "Excellent! You guessed the number!" (Fig. 1.16).



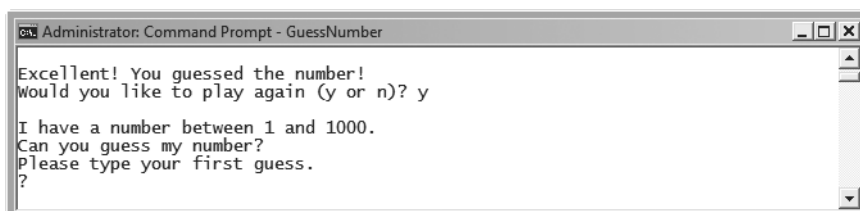
```

Administrator: Command Prompt - GuessNumber
Too high. Try again.
? 125
Too low. Try again.
? 187
Too high. Try again.
? 156
Too high. Try again.
? 140
Too high. Try again.
? 132
Too high. Try again.
? 128
Too low. Try again.
? 130
Too low. Try again.
? 131
Excellent! You guessed the number!
Would you like to play again (y or n)?

```

Fig. 1.16 | Entering additional guesses and guessing the correct number.

7. *Playing the game again or exiting the application.* After you guess correctly, the application asks if you'd like to play another game (Fig. 1.16). At the "Would you like to play again (y or n)?" prompt, entering the one character *y* causes the application to choose a new number and displays the message "Please type your first guess." followed by a question mark prompt (Fig. 1.17) so you can make your first guess in the new game. Entering the character *n* ends the application and returns you to the application's directory at the Command Prompt (Fig. 1.18). Each time you execute this application from the beginning (i.e., *Step 3*), it will choose the same numbers for you to guess.
8. *Close the Command Prompt window.*



```

Administrator: Command Prompt - GuessNumber
Excellent! You guessed the number!
Would you like to play again (y or n)? y
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?

```

Fig. 1.17 | Playing the game again.

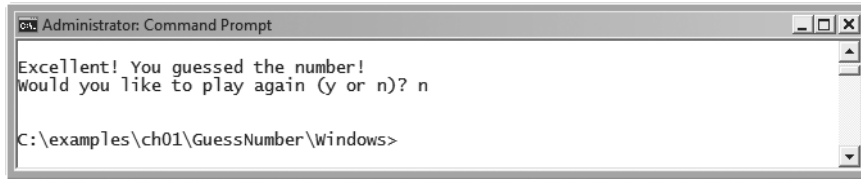


Fig. 1.18 | Exiting the game.

Running a C++ Application Using GNU C++ with Linux

For this test drive, we assume that you know how to copy the examples into your home directory. Please see your instructor if you have any questions regarding copying the files to your Linux system. Also, for the figures in this section, we use a bold highlight to point out the user input required by each step. The prompt in the shell on our system uses the tilde (~) character to represent the home directory, and each prompt ends with the dollar sign (\$) character. The prompt will vary among Linux systems.

1. *Locating the completed application.* From a Linux shell, change to the completed `GuessNumber` application directory (Fig. 1.19) by typing

```
cd Examples/ch01/GuessNumber/GNU_Linux
```

then pressing *Enter*. The command `cd` is used to change directories.

```
~$ cd examples/ch01/GuessNumber/GNU_Linux
~/examples/ch01/GuessNumber/GNU_Linux$
```

Fig. 1.19 | Changing to the `GuessNumber` application's directory.

2. *Compiling the `GuessNumber` application.* To run an application on the GNU C++ compiler, you must first compile it by typing

```
g++ GuessNumber.cpp -o GuessNumber
```

as in Fig. 1.20. This command compiles the application and produces an executable file called `GuessNumber`.

```
~/examples/ch01/GuessNumber/GNU_Linux$ g++ GuessNumber.cpp -o GuessNumber
~/examples/ch01/GuessNumber/GNU_Linux$
```

Fig. 1.20 | Compiling the `GuessNumber` application using the `g++` command.

3. *Running the `GuessNumber` application.* To run the executable file `GuessNumber`, type `./GuessNumber` at the next prompt, then press *Enter* (Fig. 1.21).

```
~/examples/ch01/GuessNumber/GNU_Linux$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

Fig. 1.21 | Running the `GuessNumber` application.

4. *Entering your first guess.* The application displays "Please type your first guess.", then displays a question mark (?) as a prompt on the next line (Fig. 1.21). At the prompt, enter 500 (Fig. 1.22). [*Note:* This is the same application that we modified and test-drove for Windows, but the outputs could vary based on the compiler being used.]

5. *Entering another guess.* The application displays "Too high. Try again.", meaning that the value you entered is greater than the number the application chose as the correct guess (Fig. 1.22). At the next prompt, enter 250 (Fig. 1.23). This time the application displays "Too low. Try again.", because the value you entered is less than the correct guess.
6. *Entering additional guesses.* Continue to play the game (Fig. 1.24) by entering values until you guess the correct number. When you guess correctly, the application displays "Excellent! You guessed the number."

```
~/examples/ch01/GuessNumber/GNU_Linux$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

Fig. 1.22 | Entering an initial guess.

```
~/examples/ch01/GuessNumber/GNU_Linux$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too low. Try again.
?
```

Fig. 1.23 | Entering a second guess and receiving feedback.

```
Too low. Try again.
? 375
Too low. Try again.
? 437
Too high. Try again.
? 406
Too high. Try again.
? 391
Too high. Try again.
? 383
Too low. Try again.
? 387
Too high. Try again.
? 385
Too high. Try again.
? 384
Excellent! You guessed the number.
Would you like to play again (y or n)?
```

Fig. 1.24 | Entering additional guesses and guessing the correct number.

7. *Playing the game again or exiting the application.* After you guess the correct number, the application asks if you'd like to play another game. At the "Would you like to play again (y or n)?" prompt, entering the one character y causes the application to choose a new number and displays the message "Please type your first guess." followed by a question mark prompt (Fig. 1.25) so you can make your first guess in the new game. Entering the character n ends the application and returns you to the application's directory in the shell (Fig. 1.26). Each time you execute this application from the beginning (i.e., *Step 3*), it will choose the same numbers for you to guess.

```
Excellent! You guessed the number.  
Would you like to play again (y or n)? y  
  
I have a number between 1 and 1000.  
Can you guess my number?  
Please type your first guess.  
?
```

Fig. 1.25 | Playing the game again.

```
Excellent! You guessed the number.  
Would you like to play again (y or n)? n  
  
~/examples/ch01/GuessNumber/GNU_Linux$
```

Fig. 1.26 | Exiting the game.

1.11 Operating Systems

Operating systems are software systems that make using computers more convenient for users, application developers and system administrators. They provide services that allow each application to execute safely, efficiently and *concurrently* (i.e., in parallel) with other applications. The software that contains the core components of the operating system is called the **kernel**. Popular desktop operating systems include Linux, Windows and OS X (formerly called Mac OS X)—we used all three in developing this book. Popular mobile operating systems used in smartphones and tablets include Google’s Android, Apple’s iOS (for iPhone, iPad and iPod Touch devices), BlackBerry OS and Windows Phone. You can develop applications in C++ for all of the following key operating systems, including several of the latest mobile operating systems.

1.11.1 Windows—A Proprietary Operating System

In the mid-1980s, Microsoft developed the **Windows operating system**, consisting of a graphical user interface built on top of DOS—an enormously popular personal-computer operating system that users interacted with by *typing* commands. Windows borrowed from many concepts (such as icons, menus and windows) developed by Xerox PARC and popularized by early Apple Macintosh operating systems. Windows 8 is Microsoft’s latest operating system—its features include enhancements to the user interface, faster startup times, further refinement of security features, touch-screen and multitouch support, and more. Windows is a *proprietary* operating system—it’s controlled by Microsoft exclusively. Windows is by far the world’s most widely used desktop operating system.

1.11.2 Linux—An Open-Source Operating System

The Linux operating system is perhaps the greatest success of the *open-source* movement. **Open-source software** departs from the *proprietary* software development style that dominated software’s early years. With open-source development, individuals and companies *contribute* their efforts in developing, maintaining and evolving software in exchange for the right to use that software for their own purposes, typically at *no charge*. Open-source code is often scrutinized by a much larger audience than proprietary software, so errors often get removed faster. Open source also encourages innovation. Enterprise systems companies, such as IBM, Oracle and many others, have made significant investments in Linux open-source development.

Some key organizations in the open-source community are the Eclipse Foundation (the Eclipse Integrated Development Environment helps programmers conveniently develop software), the Mozilla Foundation