starting out with >>>

PROGRAMMING LOGIC AND DESIGN

SIXTH EDITION

TONY GADDIS

Sixth Edition

Starting Out with

Programming Logic & Design

This page intentionally left blank

Sixth Edition

Starting Out with

Programming Logic & Design

Tony Gaddis
Haywood Community College

Pearson

Content Development: Dawn Murrin
Content Management: Tracy Johnson
Content Production: Ishan Chaudhary and Carole Snyder
Product Management: Holly Stark
Product Marketing: Wayne Stevens
Rights and Permissions: Anjali Singh

Please contact https://support.pearson.com/getsupport/s/ with any queries on this content.

Cover Image by Patrick de grijs/123RF.

Library of Congress Cataloging-in-Publication Data
Starting Out with Programming Logic & Design
Library of Congress Cataloging in Publication Control Number: 2021057225

ScoutAutomatedPrintCode

Pearson

Rental:
ISBN-10: 0-13-760214-6
ISBN-13: 978-0-13-760214-8

Print offer:
ISBN-10: 0-13-760191-3
ISBN-13: 978-0-13-760191-2

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity, depth, and breadth of all learners' lived experiences.

We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, sex, sexual orientation, socioeconomic status, ability, age, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational content accurately reflects the histories and lived experiences of the learners we serve.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content prompts deeper discussions with students and motivates them to expand their own learning (and worldview).

Accessibility

We are also committed to providing products that are fully accessible to all learners. As per Pearson's guidelines for accessible educational Web media, we test and retest the capabilities of our products against the highest standards for every release, following the WCAG guidelines in developing new products for copyright year 2022 and beyond.

You can learn more about Pearson's commitment to accessibility at **https://www.pearson.com/us/accessibility.html**

Contact Us

While we work hard to present unbiased, fully accessible content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

Please contact us with concerns about any potential bias at **https://www.pearson.com/report-bias.html**

For accessibility-related issues, such as using assistive technology with Pearson products, alternative text requests, or accessibility documentation, email the Pearson Disability Support team at **disability.support@pearson.com**

Pearson

Brief Contents

Contents

Chapter 3 Decision Structures and Boolean Logic 103

Chapter 4 Repetition Structures 161

Chapter 5 Modules 227

Preface

Welcome to *Starting Out with Programming Logic and Design*, Sixth Edition. This book uses a language-independent approach to teach programming concepts and problem-solving skills, without assuming any previous programming experience. By using easy-to-understand pseudocode, flowcharts, and other tools, the student learns how to design the logic of programs without the complication of language syntax.

Fundamental topics such as data types, variables, input, output, control structures, modules, functions, arrays, and files are covered as well as object-oriented concepts, GUI development, and event-driven programming. As with all the books in the *Starting Out with . . .* series, this text is written in clear, easy-to-understand language that students find friendly and inviting.

Each chapter presents a multitude of program design examples. Short examples that highlight specific programming topics are provided, as well as more involved examples that focus on problem solving. Each chapter includes at least one *In the Spotlight* section that provides step-by-step analysis of a specific problem and demonstrates a solution to that problem.

This book is ideal for a programming logic course that is taught as a precursor to a language-specific introductory programming course, or for the first part of an introductory programming course in which a specific language is taught.

Changes in the Sixth Edition

Previous editions of this book introduced modules, which are procedures that do not return a value, in Chapter 3. Feedback from adopters and reviewers indicate that students sometimes have trouble learning about modules before they have been exposed to control structures, such as `If` statements and loops. In this edition, the chapter on modules has been moved to Chapter 5. Now, the students will learn about control structures, then modules, and then value-returning functions. This improved pedagogy gradually introduces the students to the different ways a program's flow of execution can be directed.

Brief Overview of Each Chapter

Chapter 1: Introduction to Computers and Programming

This chapter begins by giving a concise and easy-to-understand explanation of how computers work, how data is stored and manipulated, and why we write programs in high-level languages.

Chapter 2: Input, Processing, and Output

This chapter introduces the program development cycle, data types, variables, and sequence structures. The student learns to use pseudocode and flowcharts to design simple programs that read input, perform mathematical operations, and produce screen output.

Chapter 3: Decision Structures and Boolean Logic

In this chapter students explore relational operators and Boolean expressions and are shown how to control the flow of a program with decision structures. The `If-Then`, `If-Then-Else`, and `If-Then-Else If` statements are covered. Nested decision structures, logical operators, and the case structure are also discussed.

Chapter 4: Repetition Structures

This chapter shows the student how to use loops to create repetition structures. The `While`, `Do-While`, `Do-Until`, and `For` loops are presented. Counters, accumulators, running totals, and sentinels are also discussed.

Chapter 5: Modules

This chapter demonstrates the benefits of modularizing programs and using the top-down design approach. The student learns to define and call modules, pass arguments to modules, and use local variables. Hierarchy charts are introduced as a design tool.

Chapter 6: Functions

This chapter begins by discussing common library functions, such as those for generating random numbers. After learning how to call library functions and how to use values returned by functions, the student learns how to define and call his or her own functions.

Chapter 7: Input Validation

This chapter discusses the importance of validating user input. The student learns to write input validation loops that serve as error traps. Defensive programming and the importance of anticipating obvious as well as unobvious errors is discussed.

Chapter 8: Arrays

In this chapter the student learns to create and work with one- and two-dimensional arrays. Many examples of array processing are provided including examples illustrating how to find the sum, average, and highest and lowest values in an array, and how to sum the rows, columns, and all elements of a two-dimensional array. Programming techniques using parallel arrays are also demonstrated.

Chapter 9: Sorting and Searching Arrays

In this chapter the student learns the basics of sorting arrays and searching for data stored in them. The chapter covers the bubble sort, selection sort, insertion sort, and binary search algorithms.

Chapter 10: Files

This chapter introduces sequential file input and output. The student learns to read and write large sets of data, store data as fields and records, and design programs that work with both files and arrays. The chapter concludes by discussing control break processing.

Chapter 11: Menu-Driven Programs

In this chapter the student learns to design programs that display menus and execute tasks according to the user's menu selection. The importance of modularizing a menu-driven program is also discussed.

Chapter 12: Text Processing

This chapter discusses text processing at a detailed level. Algorithms that step through the individual characters in a string are discussed, and several common library functions for character and text processing are introduced.

Chapter 13: Recursion

This chapter discusses recursion and its use in problem solving. A visual trace of recursive calls is provided, and recursive applications are discussed. Recursive algorithms for many tasks are presented, such as finding factorials, finding a greatest common denominator (GCD), summing a range of values in an array, and performing a binary search. The classic Towers of Hanoi example is also presented.

Chapter 14: Object-Oriented Programming

This chapter compares procedural and object-oriented programming practices. It covers the fundamental concepts of classes and objects. Fields, methods, access specification, constructors, accessors, and mutators are discussed. The student learns how to model classes with UML and how to find the classes in a particular problem.

Chapter 15: GUI Applications and Event-Driven Programming

This chapter discusses the basic aspects of designing a GUI application. Building graphical user interfaces with visual design tools (such as Visual Studio® or NetBeans™) is discussed. The student learns how events work in a GUI application and how to write event handlers.

Appendix A: ASCII/Unicode Characters

This appendix lists the ASCII character set, which is the same as the first 127 Unicode character codes.

Appendix B: Flowchart Symbols

This appendix shows the flowchart symbols that are used in this book.

Appendix C: Pseudocode Reference

This appendix provides a quick reference for the pseudocode language that is used in the book.

Appendix D: Converting Decimal Numbers to Binary

This appendix uses a simple tutorial to demonstrate how to convert a decimal number to binary.

Appendix E: Answers to Checkpoint Questions

This appendix provides answers to the Checkpoint questions that appear throughout the text.

Organization of the Text

The text teaches programming logic and design in a step-by-step manner. Each chapter covers a major set of topics and builds knowledge as students progress through the book. Although the chapters can be easily taught in their existing sequence, there is some flexibility. Figure P-1 shows chapter dependencies. Each box represents a chapter or a group of chapters. A chapter to which an arrow points must be covered before the chapter from which the arrow originates. The dotted line indicates that only a portion of Chapter 10 depends on information presented in Chapter 8.

Features of the Text

Concept Statements. Each major section of the text starts with a concept statement. This statement concisely summarizes the main point of the section.

Example Programs. Each chapter has an abundant number of complete and partial example programs, each designed to highlight the current topic. Pseudocode, flowcharts, and other design tools are used in the example programs.

In the Spotlight. Each chapter has one or more *In the Spotlight* case studies that provide detailed, step-by-step analysis of problems, and show the student how to solve them.

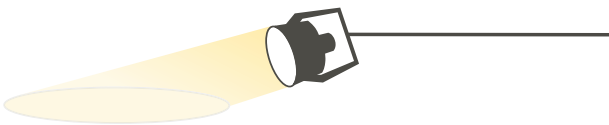
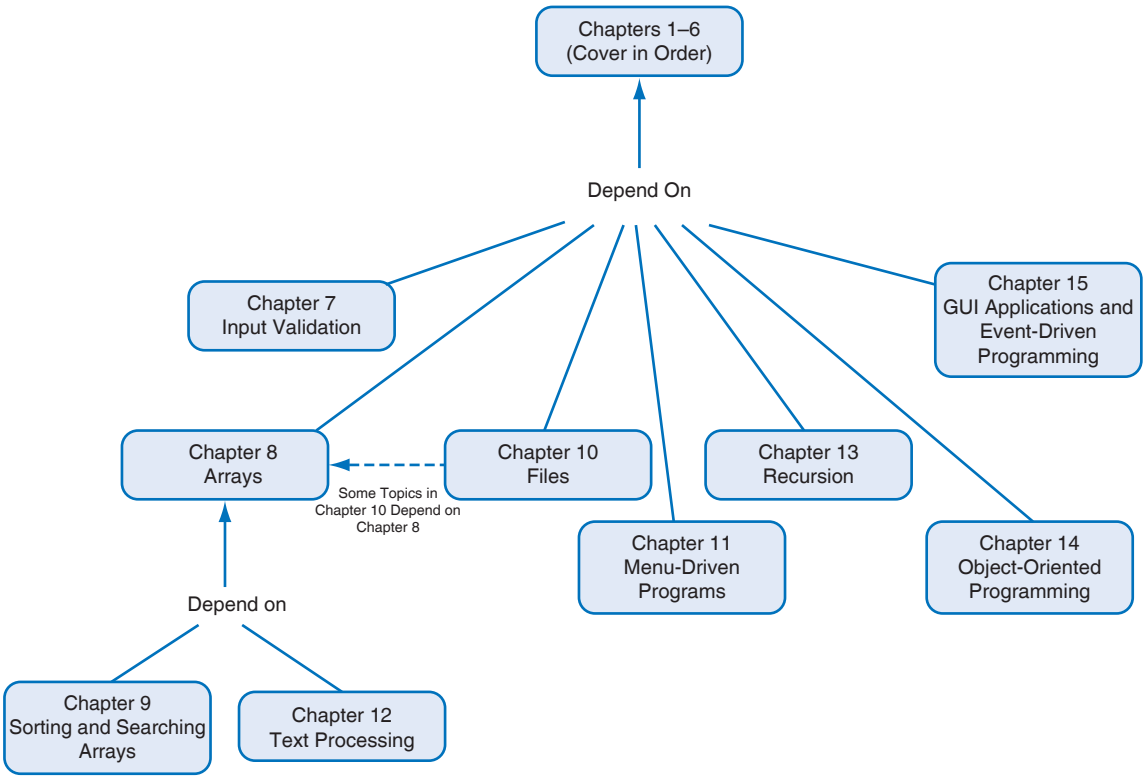


Figure P-1 Chapter dependencies



VideoNotes. A series of online videos, developed specifically for this book, are available for viewing at `www.pearson.com/cs-resources`. Icons appear throughout the text alerting the student to videos about specific topics.

NOTE: Notes appear at several places throughout the text. They are short explanations of interesting or often misunderstood points relevant to the topic at hand.

TIP: Tips advise the student on the best techniques for approaching different programming or animation problems.

WARNING! Warnings caution students about programming techniques or practices that can lead to malfunctioning programs or lost data.

Programming Language Companions. Many of the pseudocode programs shown in this book have also been written in Java, Python, and C++. These programs appear in the programming language companions that are available at `www.pearson.com/cs-resources`. Icons appear next to each pseudocode program that also appears in the language companions.

Checkpoints. Checkpoints are questions placed at intervals throughout each chapter. They are designed to query the student's knowledge quickly after learning a new topic.

Review Questions. Each chapter presents a thorough and diverse set of Review Questions and exercises. They include Multiple Choice, True/False, Short Answer, and Algorithm Workbench.

Debugging Exercises. Most chapters provide a set of Debugging Exercises in which the student examines a set of pseudocode algorithms and identifies logical errors.

Programming Exercises. Each chapter offers a pool of Programming Exercises designed to solidify the student's knowledge of the topics currently being studied.

Supplements

Student Online Resources

Many student resources are available for this book from the publisher. The following items are available on the Gaddis Series resource page at `www.pearson.com/cs-resources`:

- **Access to the book's companion VideoNotes**

  An extensive series of online VideoNotes have been developed to accompany this text. Throughout the book, VideoNote icons alert the student to videos covering specific topics. Additionally, one programming exercise at the end of each chapter has an accompanying VideoNote explaining how to develop the problem's solution.

- **Access to the Language Companions for Python, Java, and C++**

  Programming language companions specifically designed to accompany this textbook are available for download. The companions introduce the Java™, Python®, and C++ programming languages, and correspond on a chapter-by-chapter basis with the textbook. Many of the pseudocode programs that appear in the textbook also appear in the companions, implemented in a specific programming language.

- **A link to download the Flowgorithm flowcharting application**

  Flowgorithm is a free application, developed by Devin Cook at Sacramento State University, which allows you to create programs using simple flowcharts. It supports the flowcharting conventions used in this textbook, as well as several other standard conventions. When you create a flowchart with Flowgorithm, you can execute the program and generate Gaddis Pseudocode. You can also generate source code in Java, Python, Visual Basic, C#, Ruby, JavaScript, and several other languages. For more information, see `www.flowgorithm.org`.

- **A link to download the RAPTOR flowcharting environment**

  RAPTOR is a flowchart-based programming environment developed by the US Air Force Academy Department of Computer Science. For more information, see `https://raptor.martincarlisle.com`.

Instructor Resources

The following supplements are available to qualified instructors only:

- Answers to all of the Review Questions
- Solutions for the Programming Exercises
- PowerPoint® presentation slides for each chapter
- Test bank

Visit the Pearson Instructor Resource Center `www.pearson.com` or contact your local Pearson representative for information on how to access them.

This page intentionally left blank

Acknowledgments

There have been many helping hands in the development and publication of this text. I would like to thank the following faculty reviewers:

Reviewers for This Edition

Taz Daughtrey
Central Virginia Community College

Donna Sandsmark
University of California, San Diego

Holly Tajlil
Sacramento State University

Deborah Wilson
Ashland University

Reviewers of Previous Editions

Reni Abraham
Houston Community College

Alan Anderson
Gwinnett Technical College

Cherie Aukland
Thomas Nelson Community College

Steve Browning
Freed Hardeman University

John P. Buerck
Saint Louis University

Jill Canine
Ivy Tech Community College of Indiana

Tony Cantrell
Georgia Northwestern Technical College

Steven D. Carver
Ivy Tech Community College

Stephen Robert Cheskiewicz
Keystone College and Wilkes University

Katie Danko
Grand Rapids Community College

Richard J. Davison
College of the Albemarle

Sameer Dutta
Grambling State University

Norman P. Hahn
Thomas Nelson Community College

John Haley
Athens Technical College

Keith Hallmark
Calhoun Community College

Ronald J. Harkins
Miami University, OH

Dianne Hill
Jackson College

Vai Kumar
Pensacola State College

Coronicca Oliver
Coastal Georgia Community College

Robert S. Overall, III
Nashville State Community College

Dale T. Pickett
Baker College of Clinton Township

Tonya Pierce
Ivy Tech Community College

J. Shawn Pope
Tulsa Community College

Maryam Rahnemoonfar
Texas A&M University

Linda Reeser
Arizona Western College

Homayoun Sharafi
Prince George's Community College

Emily Shepard
Central Carolina Community College

Larry Strain
Ivy Tech Community College–Bloomington

Donald Stroup
Ivy Tech Community College

John Thacher
Gwinnett Technical College

Jim Turney
Austin Community College

Scott Vanselow
Edison State College

I would like to thank the faculty, staff, and administration at Haywood Community College for the opportunity to build a career teaching the subjects that I love. I would also like to thank my family and friends for their support in all my projects.

It is a great honor to be published by Pearson, and I am extremely fortunate to have Tracy Johnson as my Content Manager. She and her colleagues Holly Stark, Erin Sullivan, Sandra Rodriguez, Wayne Stevens, Scott Disanno, Bob Engelhardt, Ishan Chaudhary, Carole Snyder, and Mahalakshmi Usha have worked tirelessly to produce and promote this book. Thanks to you all!

This page intentionally left blank

About the Author

Tony Gaddis is the principal author of the *Starting Out with . . .* series of textbooks. Tony has twenty years of experience teaching computer science courses at Haywood Community College. He is a highly acclaimed instructor who was previously selected as the North Carolina Community College "Teacher of the Year" and has received the Teaching Excellence award from the National Institute for Staff and Organizational Development. The *Starting Out with . . .* series includes introductory books covering Programming Logic and Design, C++, Java, Microsoft® Visual Basic, C#®, Python, App Inventor, and Alice, all published by Pearson.

This page intentionally left blank

Sixth Edition

Starting Out with

Programming Logic & Design

This page intentionally left blank

CHAPTER 1

Introduction to Computers and Programming

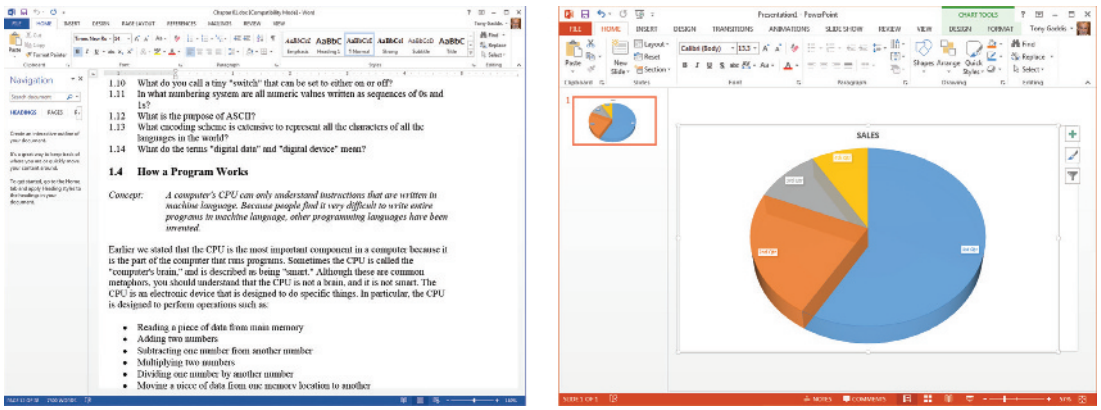
TOPICS

1.1 Introduction

Think about some of the different ways that people use computers. In school, students use computers for tasks such as writing papers, searching for articles, sending email, and participating in online classes. At work, people use computers to analyze data, make presentations, conduct business transactions, communicate with customers and coworkers, control machines in manufacturing facilities, and many other things. At home, people use computers for tasks such as paying bills, shopping online, communicating with friends and family, and playing computer games. And don't forget that smart phones, tablets, home automation devices, car navigation systems, and many other devices are computers too. The uses of computers are almost limitless in our everyday lives.

Computers can do such a wide variety of things because they can be programmed. This means that computers are not designed to do just one job, but to do any job that their programs tell them to do. A *program* is a set of instructions that a computer follows to perform a task. For example, Figure 1-1 shows screens from two commonly used programs: Microsoft Word and PowerPoint.

Figure 1-1 Commonly used programs (courtesy of Microsoft Corporation)



Programs are commonly referred to as *software*. Software is essential to a computer because without software, a computer can do nothing. All of the software that we use to make our computers useful is created by individuals known as programmers or software developers. A *programmer*, or *software developer*, is a person with the training and skills necessary to design, create, and test computer programs. Computer programming is an exciting and rewarding career. Today, you will find programmers working in business, medicine, government, law enforcement, agriculture, academics, entertainment, and almost every other field.

This book introduces you to the fundamental concepts of computer programming. Before we begin exploring those concepts, you need to understand a few basic things about computers and how they work. This chapter will build a solid foundation of knowledge that you will continually rely on as you study computer science. First, we will discuss the physical components that computers are commonly made of. Next, we will look at how computers store data and execute programs. Finally, we will discuss the major types of software that computers use.

1.2 Hardware

CONCEPT: **The physical devices that a computer is made of are referred to as the computer's hardware. Most computer systems are made of similar hardware devices.**

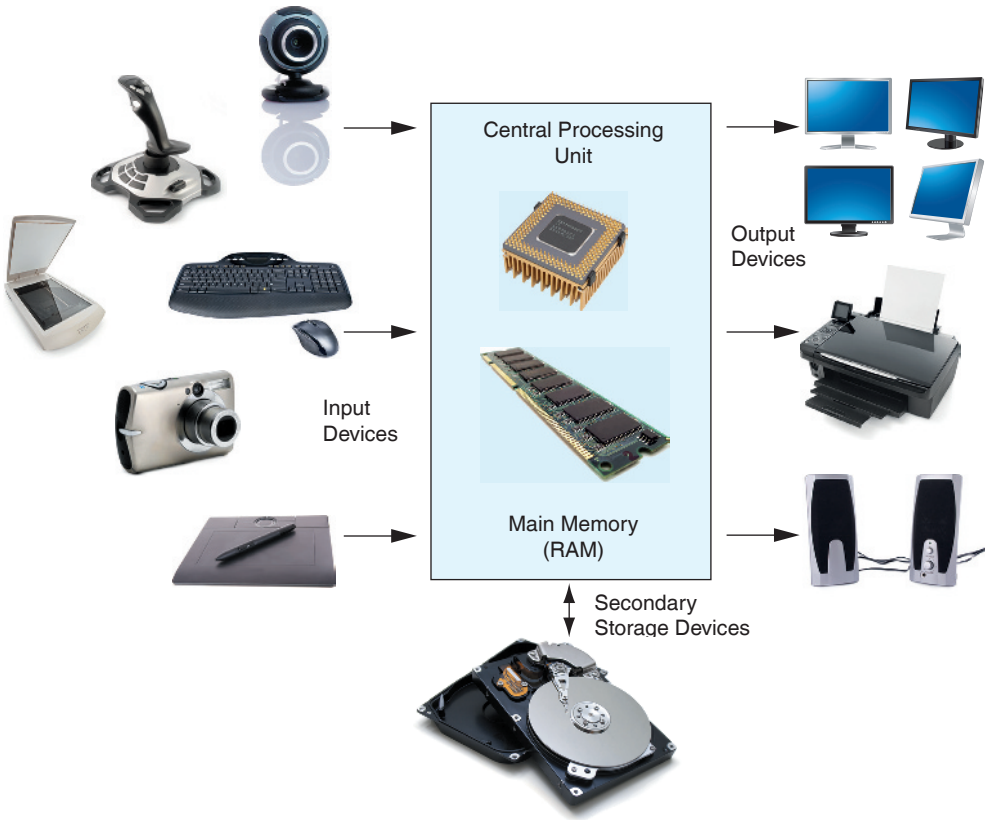
The term *hardware* refers to all of the physical devices, or *components*, that a computer is made of. A computer is not one single device, but a system of devices that all work together. Like the different instruments in a symphony orchestra, each device in a computer plays its own part.

If you have ever shopped for a computer, you've probably seen sales literature listing components such as microprocessors, memory, disk drives, video displays, graphics cards, and so on. Unless you already know a lot about computers, or at least have a friend who

does, understanding what these different components do can be confusing. As shown in Figure 1-2, a typical computer system consists of the following major components:

- The central processing unit (CPU)
- Main memory
- Secondary storage devices
- Input devices
- Output devices

Figure 1-2 Typical components of a computer system



(***Photo credits:*** **Webcam** Iko/Shutterstock, **Joystick** Nikita Rogul/Shutterstock, **Scanner** Feng Yu/Shutterstock, **Keyboard** Chiyacat/Shutterstock, **Camera** Elkostas/Shutterstock, **Tablet** Tkemot/Shutterstock, **Hard disk** Vitaly Korovin/Shutterstock, **Speakers** StockPhotosArt/Shutterstock, **Printer** Jocic/Shutterstock, **Monitors** Art gallery/Shutterstock, **RAM** Peter Guess/Shutterstock, **Chip** Aquila/Shutterstock).

Let's take a closer look at each of these components.

The CPU

When a computer is performing the tasks that a program tells it to do, we say that the computer is *running* or *executing* the program. The *central processing unit*, or *CPU*, is the part of a computer that actually runs programs. (The CPU is often referred to as the *processor*.) The CPU is the most important component in a computer because without it, the computer could not run software.

In the earliest computers, CPUs were huge devices made of electrical and mechanical components such as vacuum tubes and switches. Figure 1-3 shows such a device. The two women in the photo are working with the historic ENIAC computer. The *ENIAC*, considered by many to be the world's first programmable electronic computer, was built in 1945 to calculate artillery ballistic tables for the U.S. Army. This machine, which was primarily one big CPU, was 8 feet tall, 100 feet long, and weighed 30 tons.

Today, CPUs are small chips known as *microprocessors*. Figure 1-4 shows a photo of a lab technician holding a modern-day microprocessor. In addition to being much

Figure 1-3 The ENIAC computer (courtesy of US Army Center of Military History)

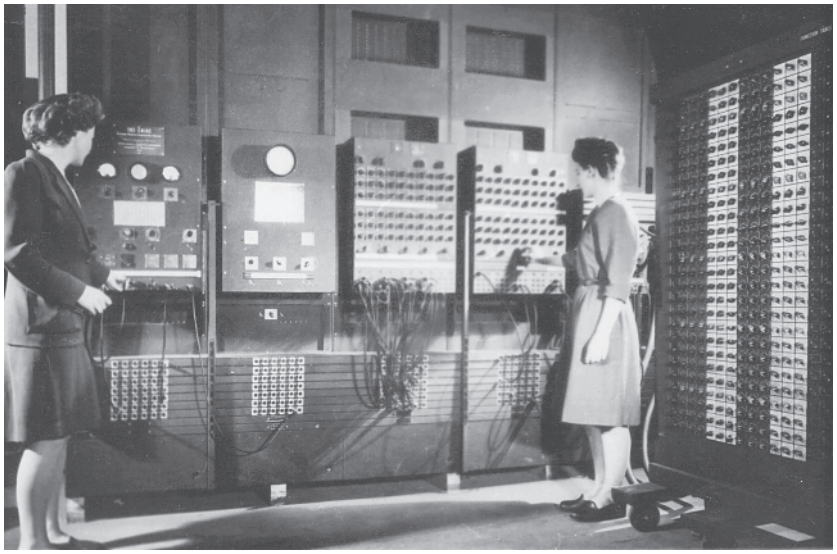


Figure 1-4 A lab technician holds a modern microprocessor (courtesy of Chris Ryan/OJO Images/Getty Images)

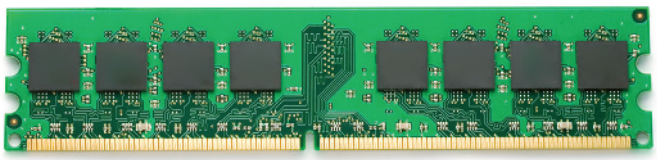
smaller than the old electro-mechanical CPUs in early computers, microprocessors are also much more powerful.

Main Memory

You can think of *main memory* as the computer's work area. This is where the computer stores a program while the program is running, as well as the data that the program is working with. For example, suppose you are using a word processing program to write an essay for one of your classes. While you do this, both the word processing program and the essay are stored in main memory.

Main memory is commonly known as *random-access memory*, or *RAM*. It is called this because the CPU is able to quickly access data stored at any random location in RAM. RAM is usually a *volatile* type of memory that is used only for temporary storage while a program is running. When the computer is turned off, the contents of RAM are erased. Inside your computer, RAM is stored in chips, similar to the ones shown in Figure 1-5.

Figure 1-5 Memory chips (photo © Garsya/Shutterstock)



NOTE: Another type of memory that is stored in chips inside the computer is *read-only memory*, or *ROM*. A computer can read the contents of ROM, but it cannot change its contents, or store additional data there. ROM is *nonvolatile*, which means that it does not lose its contents, even when the computer's power is turned off. ROM is typically used to store programs that are important for the system's operation. One example is the computer's startup program, which is executed each time the computer is started.

Secondary Storage Devices

Secondary storage is a type of memory that can hold data for long periods of time, even when there is no power to the computer. Programs are normally stored in secondary memory and loaded into main memory as needed. Important data, such as word processing documents, payroll data, and inventory records, is saved to secondary storage as well.

The most common type of secondary storage device is the disk drive. A traditional *disk drive* stores data by magnetically encoding it onto a circular disk. *Solid state drives*, which store data in solid-state memory, are increasingly becoming popular. A solid state drive has no moving parts, and operates faster than a traditional disk drive. Most computers have some sort of secondary storage device, either a traditional disk drive or a solid state drive, mounted inside their case. External disk drives, which connect to one of the computer's communication ports, are also available. External disk drives can be used to create backup copies of important data or to move data to another computer.

In addition to external disk drives, many types of devices have been created for copying data, and for moving it to other computers. *Universal Serial Bus drives*, or *USB drives*, are small devices that plug into the computer's USB port, and appear to the system as a disk drive. These drives do not actually contain a disk, however. They store data in a special type of memory known as *flash memory*. USB drives, which are also known as *memory sticks* and *flash drives*, are inexpensive, reliable, and small enough to be carried in your pocket.

NOTE: In recent years, *cloud storage* has become a popular way to store data. When you store data in the cloud, you are storing it on a remote server via the Internet, or via a company's private network. When your data is stored in the cloud, you can access it from many different devices, and from any location where you have a network connection. Cloud storage can also be used to back up important data that is stored on a computer's disk.

Input Devices

Input is any data the computer collects from people and from other devices. The component that collects the data and sends it to the computer is called an *input device*. Common input devices are the keyboard, mouse, touchscreen, scanner, microphone, and digital camera. Disk drives and optical drives can also be considered input devices because programs and data are retrieved from them and loaded into the computer's memory.

Output Devices

Output is any data the computer produces for people or for other devices. It might be a sales report, a list of names, or a graphic image. The data is sent to an *output device*, which formats and presents it. Common output devices are video displays and printers. Disk drives can also be considered output devices because the system sends data to them in order to be saved.

Checkpoint

1.1 What is a program?

1.2 What is hardware?

1.3 List the five major components of a computer system.

1.4 What part of the computer actually runs programs?

1.5 What part of the computer serves as a work area to store a program and its data while the program is running?

1.6 What part of the computer holds data for long periods of time, even when there is no power to the computer?

1.7 What part of the computer collects data from people and from other devices?

1.8 What part of the computer formats and presents data for people or other devices?

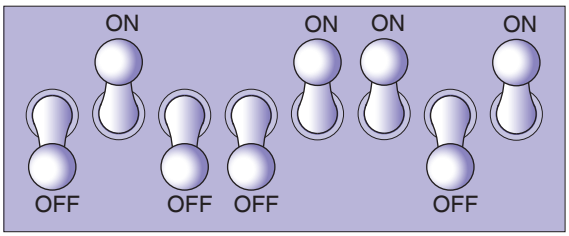
1.3 How Computers Store Data

CONCEPT: **All data that is stored in a computer is converted to sequences of 0s and 1s.**

A computer's memory is divided into tiny storage locations known as *bytes*. One byte is only enough memory to store a letter of the alphabet or a small number. In order to do anything meaningful, a computer has to have lots of bytes. Most computers today have millions, or even billions, of bytes of memory.

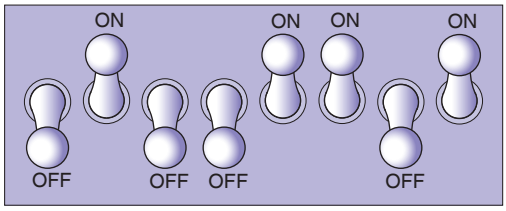
Each byte is divided into eight smaller storage locations known as bits. The term *bit* stands for *binary digit*. Computer scientists usually think of bits as tiny switches that can be either on or off. Bits aren't actual "switches," however, at least not in the conventional sense. In most computer systems, bits are tiny electrical components that can hold either a positive or a negative charge. Computer scientists think of a positive charge as a switch in the *on* position, and a negative charge as a switch in the *off* position. Figure 1-6 shows the way that a computer scientist might think of a byte of memory: as a collection of switches that are each flipped to either the on or off position.

Figure 1-6 Think of a byte as eight switches

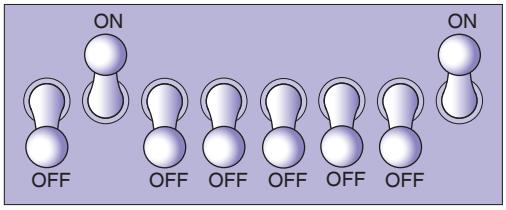


When a piece of data is stored in a byte, the computer sets the eight bits to an on/off pattern that represents the data. For example, the pattern shown on the left in Figure 1-7 shows how the number 77 would be stored in a byte, and the pattern on the right shows how the letter A would be stored in a byte. In a moment you will see how these patterns are determined.

Figure 1-7 Bit patterns for the number 77 and the letter A



The number 77 stored in a byte.



The letter A stored in a byte.

Storing Numbers

A bit can be used in a very limited way to represent numbers. Depending on whether the bit is turned on or off, it can represent one of two different values. In computer systems, a bit that is turned off represents the number 0 and a bit that is turned on represents the number 1. This corresponds perfectly to the *binary numbering system*. In the binary numbering system (or *binary*, as it is usually called) all numeric values are written as sequences of 0s and 1s. Here is an example of a number that is written in binary:

```
10011101
```

The position of each digit in a binary number has a value assigned to it. Starting with the rightmost digit and moving left, the position values are 2^0, 2^1, 2^2, 2^3, and so forth, as shown in Figure 1-8. Figure 1-9 shows the same diagram with the position values calculated. Starting with the rightmost digit and moving left, the position values are 1, 2, 4, 8, and so forth.

Figure 1-8 The values of binary digits as powers of 2

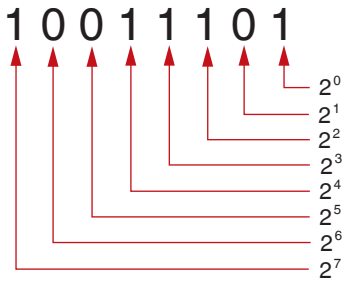
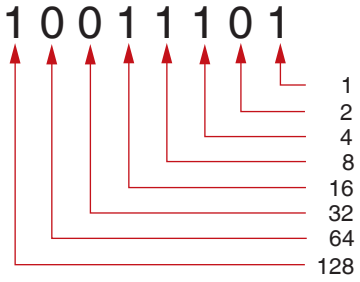


Figure 1-9 The values of binary digits



To determine the value of a binary number you simply add up the position values of all the 1s. For example, in the binary number 10011101, the position values of the 1s are 1, 4, 8, 16, and 128. This is shown in Figure 1-10. The sum of all of these position values is 157. So, the value of the binary number 10011101 is 157.

Figure 1-10 Determining the value of 10011101

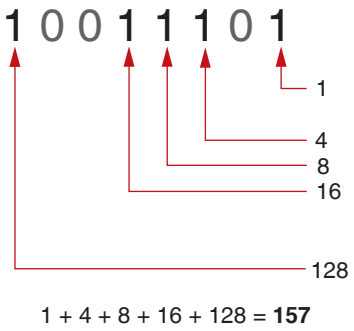
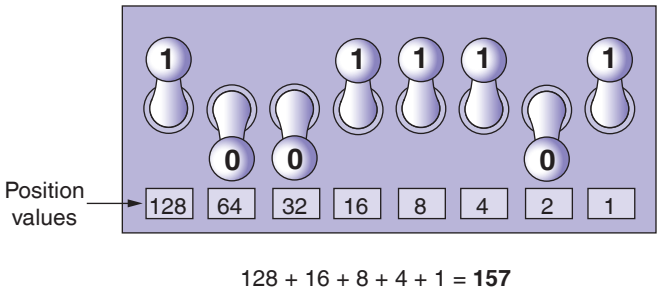


Figure 1-11 shows how you can picture the number 157 stored in a byte of memory. Each 1 is represented by a bit in the on position, and each 0 is represented by a bit in the off position.

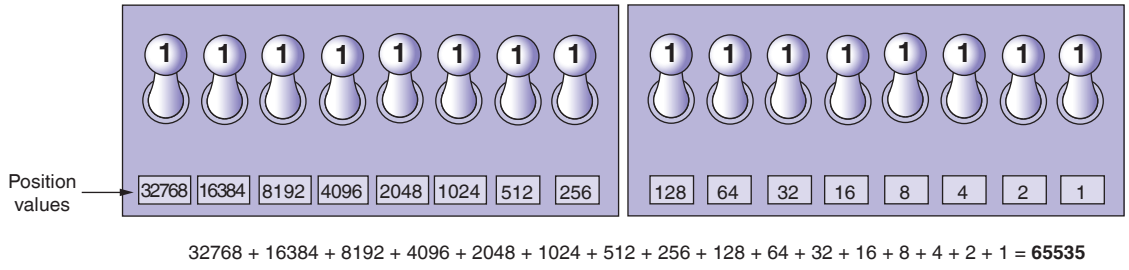
Figure 1-11 The bit pattern for 157



When all of the bits in a byte are set to 0 (turned off), then the value of the byte is 0. When all of the bits in a byte are set to 1 (turned on), then the byte holds the largest value that can be stored in it. The largest value that can be stored in a byte is $1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255$. This limit exists because there are only eight bits in a byte.

What if you need to store a number larger than 255? The answer is simple: use more than one byte. For example, suppose we put two bytes together. That gives us 16 bits. The position values of those 16 bits would be 2^0, 2^1, 2^2, 2^3, and so forth, up through 2^{15}. As shown in Figure 1-12, the maximum value that can be stored in two bytes is 65,535. If you need to store a number larger than this, then more bytes are necessary.

Figure 1-12 Two bytes used for a large number



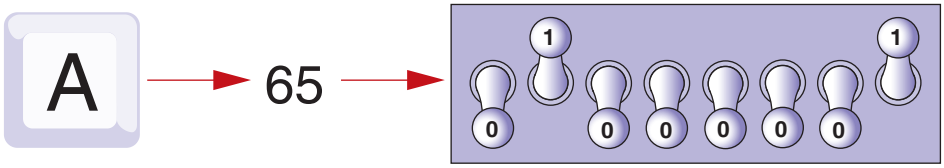
TIP: In case you're feeling overwhelmed by all this, relax! You will not have to actually convert numbers to binary while programming. Knowing that this process is taking place inside the computer will help you as you learn, and in the long term this knowledge will make you a better programmer.

Storing Characters

Any piece of data that is stored in a computer's memory must be stored as a binary number. That includes characters, such as letters and punctuation marks. When a character is stored in memory, it is first converted to a numeric code. The numeric code is then stored in memory as a binary number.

Over the years, different coding schemes have been developed to represent characters in computer memory. Historically, the most important of these coding schemes is *ASCII*, which stands for the *American Standard Code for Information Interchange*. ASCII is a set of 128 numeric codes that represent the English letters, various punctuation marks, and other characters. For example, the ASCII code for the uppercase letter A is 65. When you type an uppercase A on your computer keyboard, the number 65 is stored in memory (as a binary number, of course). This is shown in Figure 1-13.

Figure 1-13 The letter A is stored in memory as the number 65



TIP: The acronym ASCII is pronounced "askee."

In case you are curious, the ASCII code for uppercase B is 66, for uppercase C is 67, and so forth. Appendix A shows all of the ASCII codes and the characters they represent.

The ASCII character set was developed in the early 1960s, and was eventually adopted by most of all computer manufacturers. ASCII is limited, however, because it defines codes for only 128 characters. To remedy this, the Unicode character set was developed in the early 1990s. *Unicode* is an extensive encoding scheme that is compatible with ASCII, and can also represent the characters of many of the world's languages. Today, Unicode is quickly becoming the standard character set used in the computer industry.

Advanced Number Storage

Earlier you read about numbers and how they are stored in memory. While reading that section, perhaps it occurred to you that the binary numbering system can be used to represent only integer numbers, beginning with 0. Negative numbers and real numbers (such as 3.14159) cannot be represented using the simple binary numbering technique we discussed.

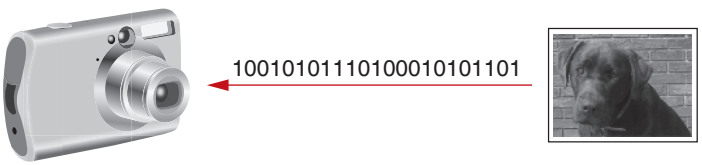
Computers are able to store negative numbers and real numbers in memory, but to do so they use encoding schemes along with the binary numbering system. Negative numbers are encoded using a technique known as *two's complement*, and real numbers are encoded in *floating-point notation*. You don't need to know how these encoding schemes work, only that they are used to convert negative numbers and real numbers to binary format.

Other Types of Data

Computers are often referred to as digital devices. The term *digital* can be used to describe anything that uses binary numbers. *Digital data* is data that is stored in binary, and a *digital device* is any device that works with binary data. In this section, we have discussed how numbers and characters are stored in binary, but computers also work with many other types of digital data.

For example, consider the pictures that you take with your digital camera. These images are composed of tiny dots of color known as *pixels*. (The term pixel stands for *picture element*.) As shown in Figure 1-14, each pixel in an image is converted to a numeric code that represents the pixel's color. The numeric code is stored in memory as a binary number.

Figure 1-14 A digital image is stored in binary format (photo on the right courtesy of Tony Gaddis)



The music that you stream from an online source, or play on an MP3 player is also digital. A digital song is broken into small pieces known as *samples*. Each sample is converted to a binary number, which can be stored in memory. The more samples that a song is divided into, the more it sounds like the original music when it is played back. A CD-quality song is divided into more than 44,000 samples per second!

Checkpoint

1.9 What amount of memory is enough to store a letter of the alphabet or a small number?

1.10 What do you call a tiny "switch" that can be set to either on or off?

1.11 In what numbering system are all numeric values written as sequences of 0s and 1s?

1.12 What is the purpose of ASCII?

1.13 What encoding scheme is extensive to represent all the characters of all the languages in the world?

1.14 What do the terms "digital data" and "digital device" mean?

1.4 How a Program Works

CONCEPT: **A computer's CPU can only understand instructions that are written in machine language. Because people find it very difficult to write entire programs in machine language, other programming languages have been invented.**

Earlier, we stated that the CPU is the most important component in a computer because it is the part of the computer that runs programs. Sometimes the CPU is called the "computer's brain," and is described as being "smart." Although these are common metaphors, you should understand that the CPU is not a brain, and it is not smart. The CPU is an electronic device that is designed to do specific things. In particular, the CPU is designed to perform operations such as the following:

- Reading a piece of data from main memory
- Adding two numbers
- Subtracting one number from another number
- Multiplying two numbers
- Dividing one number by another number
- Moving a piece of data from one memory location to another
- Determining whether one value is equal to another value
- And so forth . . .

As you can see from this list, the CPU performs simple operations on pieces of data. The CPU does nothing on its own, however. It has to be told what to do, and that's the purpose of a program. A program is nothing more than a list of instructions that cause the CPU to perform operations.

Each instruction in a program is a command that tells the CPU to perform a specific operation. Here's an example of an instruction that might appear in a program:

```
10110000
```

To you and me, this is only a series of 0s and 1s. To a CPU, however, this is an instruction to perform an operation.[1] It is written in 0s and 1s because CPUs only understand instructions that are written in *machine language*, and machine language instructions are always written in binary.

[1] The example shown is an actual instruction for an Intel microprocessor. It tells the microprocessor to move a value into the CPU.

A machine language instruction exists for each operation that a CPU is capable of performing. For example, there is an instruction for adding numbers; there is an instruction for subtracting one number from another; and so forth. The entire set of instructions that a CPU can execute is known as the CPU's *instruction set*.

NOTE: There are several microprocessor companies today that manufacture CPUs. Some of the more well-known microprocessor companies are Intel, AMD, and Motorola. If you look carefully at your computer, you might find a tag showing a logo for its microprocessor.

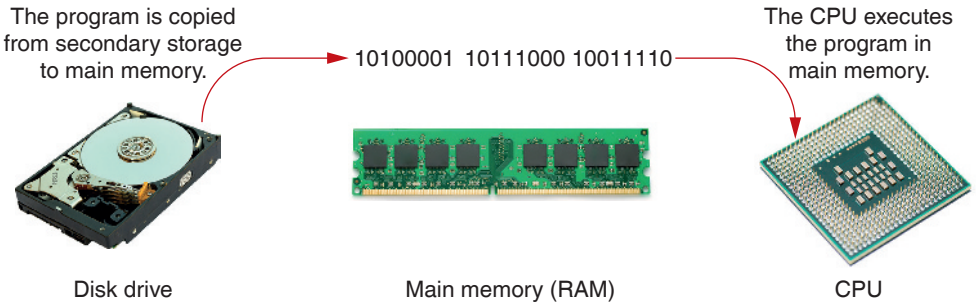
Each brand of microprocessor has its own unique instruction set, which is typically understood only by microprocessors of the same brand. For example, Intel microprocessors understand the same instructions, but they do not understand instructions for Motorola microprocessors.

The machine language instruction that was previously shown is an example of only one instruction. It takes a lot more than one instruction, however, for the computer to do anything meaningful. Because the operations that a CPU knows how to perform are so basic in nature, a meaningful task can be accomplished only if the CPU performs many operations. For example, if you want your computer to calculate the amount of interest that you will earn from your savings account this year, the CPU will have to perform a large number of instructions, carried out in the proper sequence. It is not unusual for a program to contain thousands, or even a million or more machine language instructions.

Programs are usually stored on a secondary storage device such as a disk drive. When you install a program on your computer, the program is typically downloaded from a Web site, or installed from an online app store.

Although a program can be stored on a secondary storage device such as a disk drive, it has to be copied into main memory, or RAM, each time the CPU executes it. For example, suppose you have a word processing program on your computer's disk. To execute the program you use the mouse to double-click the program's icon. This causes the program to be copied from the disk into main memory. Then, the computer's CPU executes the copy of the program that is in main memory. This process is illustrated in Figure 1-15.

Figure 1-15 A program is copied into main memory and then executed (courtesy of Lefteris Papaulakis/Shutterstock, Garsya/Shutterstock and marpan/Shutterstock)

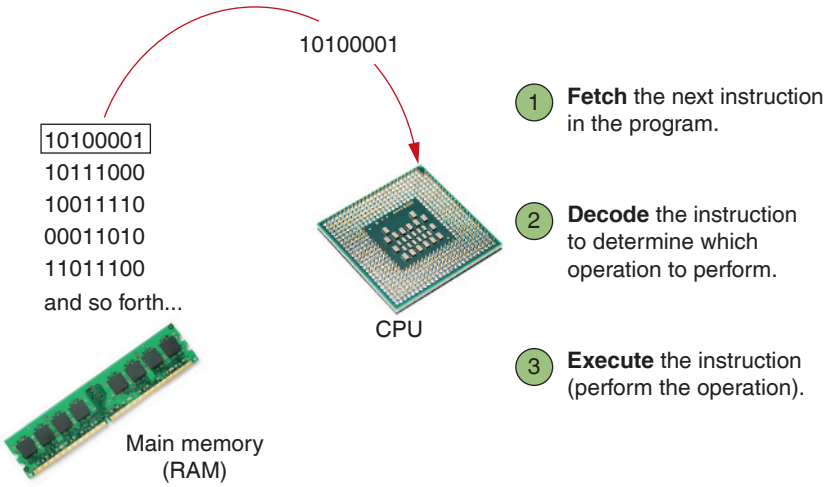


When a CPU executes the instructions in a program, it is engaged in a process that is known as the *fetch-decode-execute cycle*. This cycle, which consists of three steps, is repeated for each instruction in the program. The steps are:

1. **Fetch.** A program is a long sequence of machine language instructions. The first step of the cycle is to fetch, or read, the next instruction from memory into the CPU.
2. **Decode.** A machine language instruction is a binary number that represents a command that tells the CPU to perform an operation. In this step the CPU decodes the instruction that was just fetched from memory, to determine which operation it should perform.
3. **Execute.** The last step in the cycle is to execute, or perform, the operation.

Figure 1-16 illustrates these steps.

Figure 1-16 The fetch-decode-execute cycle (courtesy of Garsya/Shutterstock, marpan/Shutterstock)



From Machine Language to Assembly Language

Computers can only execute programs that are written in machine language. As previously mentioned, a program can have thousands, or even a million or more binary instructions, and writing such a program would be very tedious and time consuming. Programming in machine language would also be very difficult because putting a 0 or a 1 in the wrong place will cause an error.

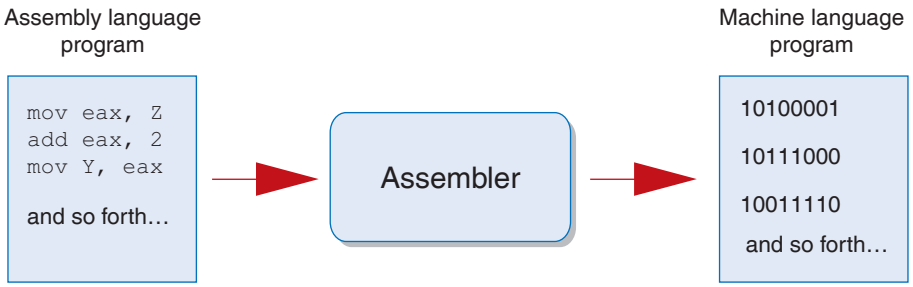
Although a computer's CPU only understands machine language, it is impractical for people to write programs in machine language. For this reason, *assembly language* was created in the early days of computing[2] as an alternative to machine language. Instead of using binary numbers for instructions, assembly language uses short words that are known as *mnemonics*. For example, in assembly language, the mnemonic `add` typically means to add numbers, `mul` typically means to multiply numbers, and `mov` typically means to move a value to a location in memory. When programmers use assembly language to write programs, they can write short mnemonics instead of binary numbers.

[2] The first assembly language was most likely developed in the 1940s at Cambridge University for use with a historical computer known as the EDSAC.

NOTE: There are many different versions of assembly language. It was mentioned earlier that each brand of CPU has its own machine language instruction set. Each brand of CPU typically has its own assembly language as well.

Assembly language programs cannot be executed by the CPU, however. The CPU only understands machine language, so a special program known as an *assembler* is used to translate an assembly language program to a machine language program. This process is shown in Figure 1-17. The machine language program that is created by the assembler can then be executed by the CPU.

Figure 1-17 An assembler translates an assembly language program to a machine language program



High-Level Languages

Although assembly language makes it unnecessary to write binary machine language instructions, it is not without difficulties. Assembly language is primarily a direct substitute for machine language, and like machine language, it requires that you know a lot about the CPU. Assembly language also requires that you write a large number of instructions for even the simplest program. Because assembly language is so close in nature to machine language, it is referred to as a *low-level language*.

In the 1950s, a new generation of programming languages known as *high-level languages* began to appear. A high-level language allows you to create powerful and complex programs without knowing how the CPU works, and without writing large numbers of low-level instructions. In addition, most high-level languages use words that are easy to understand. For example, if a programmer were using COBOL (which was one of the early high-level languages created in the 1950s), the programmer would write the following instruction to display the message "Hello world" on the computer screen:

```
Display "Hello world"
```

Doing the same thing in assembly language would require several instructions, and an intimate knowledge of how the CPU interacts with the computer's video circuitry. As you can see from this example, high-level languages allow programmers to concentrate on the tasks they want to perform with their programs rather than the details of how the CPU will execute those programs.

Since the 1950s, thousands of high-level languages have been created. Table 1-1 lists several of the more well-known languages. If you are working toward a degree in computer science or a related field, you are likely to study one or more of these languages.

Each high-level language has its own set of words that the programmer must learn in order to use the language. The words that make up a high-level programming language

Table 1-1 Programming languages

Language	Description
Ada	Ada was created in the 1970s, primarily for applications used by the U.S. Department of Defense. The language is named in honor of Ada Lovelace, a 19th century mathematician who published an algorithm that is considered by many to be the first computer program.
BASIC	**B**eginners **A**ll-purpose **S**ymbolic **I**nstruction **C**ode is a general-purpose language that was originally designed in the early 1960s to be simple enough for beginners to learn. Today, there are many different versions of BASIC.
FORTRAN	**FOR**mula **TRAN**slator was the first high-level programming language. It was designed in the 1950s for performing complex mathematical calculations.
COBOL	**C**ommon **B**usiness-**O**riented **L**anguage was created in the 1950s, and was designed for business applications.
Pascal	Pascal was created in 1970, and was originally designed for teaching programming. The language was named in honor of the mathematician, physicist, and philosopher Blaise Pascal.
C and C++	C and C++ (pronounced "c plus plus") are powerful, general-purpose languages developed at Bell Laboratories. The C language was created in 1972 and the C++ language was created in 1983.
C#	Pronounced "c sharp." This language was created by Microsoft around the year 2000 for developing applications based on the Microsoft .NET platform.
Java	Java was created by Sun Microsystems (a company that is now owned by Oracle) in the early 1990s. It can be used to develop programs that run on a single computer or over the Internet from a Web server.
JavaScript™	JavaScript, created in the 1990s, can be used in Web pages. Despite its name, JavaScript is not related to Java.
Python	Python is a general-purpose language created in the early 1990s. It has become popular in business and academic applications.
Ruby	Ruby is a general-purpose language that was created in the 1990s. It is increasingly becoming a popular language for programs that run on Web servers.
Rust	The Rust programming language is designed for high performance, memory safety, and concurrent execution. It was announced in 2010 by Mozilla Research.
Visual Basic	Visual Basic (commonly known as VB) is a Microsoft programming language and software development environment that allows programmers to create Windows®-based applications quickly. VB was originally created in the early 1990s.

are known as *key words* or *reserved words*. Each key word has a specific meaning, and cannot be used for any other purpose. You previously saw an example of a COBOL statement that uses the key word `display` to print a message on the screen. In the Python language the word `print` serves the same purpose.

In addition to key words, programming languages have *operators* that perform various operations on data. For example, all programming languages have math operators that perform arithmetic. In Java, as well as most other languages, the + sign is an operator that adds two numbers. The following adds 12 and 75:

```
12 + 75
```

In addition to key words and operators, each language also has its own *syntax*, which is a set of rules that must be strictly followed when writing a program. The syntax rules dictate how key words, operators, and various punctuation characters must be used in a program. When you are learning a programming language, you must learn the syntax rules for that particular language.

The individual instructions that you use to write a program in a high-level programming language are called *statements*. A programming statement can consist of key words, operators, punctuation, and other allowable programming elements, arranged in the proper sequence to perform an operation.

NOTE: Human languages also have syntax rules. Do you remember when you took your first English class, and you learned all those rules about infinitives, indirect objects, clauses, and so forth? You were learning the syntax of the English language.

Although people commonly violate the syntax rules of their native language when speaking and writing, other people usually understand what they mean. Unfortunately, computers do not have this ability. If even a single syntax error appears in a program, the program cannot be executed.

VideoNote

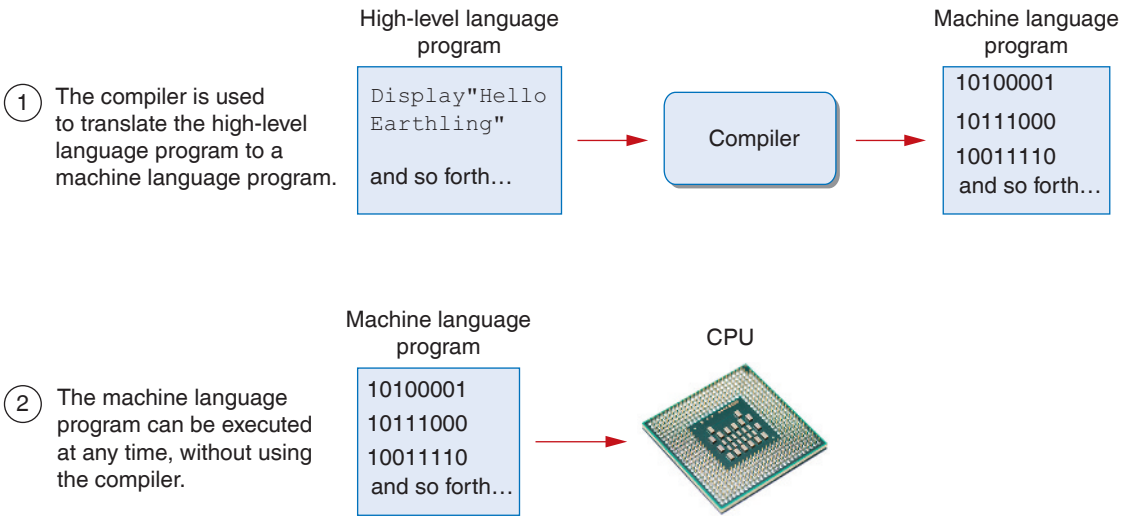
Compiling and Executing a Program

Compilers and Interpreters

Because the CPU understands only machine language instructions, programs that are written in a high-level language must be translated into machine language. Once a program has been written in a high-level language, the programmer will use a compiler or an interpreter to make the translation.

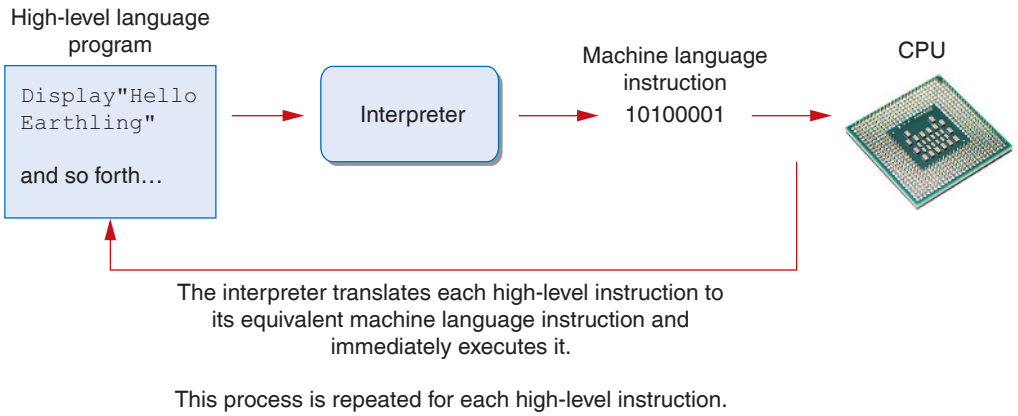
A *compiler* is a program that translates a high-level language program into a separate machine language program. The machine language program can then be executed any time it is needed. This is shown in Figure 1-18. As shown in the figure, compiling and executing are two different processes.

Figure 1-18 Compiling a high-level program and executing it (courtesy of marpan/Shutterstock)



An *interpreter* is a program that both translates and executes the instructions in a high-level language program. As the interpreter reads each individual instruction in the program, it converts it to a machine language instruction and then immediately executes it. This process repeats for every instruction in the program. This process is illustrated in Figure 1-19. Because interpreters combine translation and execution, they typically do not create separate machine language programs.

Figure 1-19 Executing a high-level program with an interpreter (courtesy of marpan/Shutterstock)



NOTE: Programs that are compiled generally execute faster than programs that are interpreted because a compiled program is already translated entirely to machine language when it is executed. A program that is interpreted must be translated at the time it is executed.

The statements that a programmer writes in a high-level language are called *source code*, or simply *code*. Typically, the programmer types a program's code into a text editor and then saves the code in a file on the computer's disk. Next, the programmer uses a compiler to translate the code into a machine language program, or an interpreter to translate and execute the code. If the code contains a syntax error, however, it cannot be translated. A *syntax error* is a mistake such as a misspelled key word, a missing punctuation character, or the incorrect use of an operator. When this happens the compiler or interpreter displays an error message indicating that the program contains a syntax error. The programmer corrects the error and then attempts once again to translate the program.

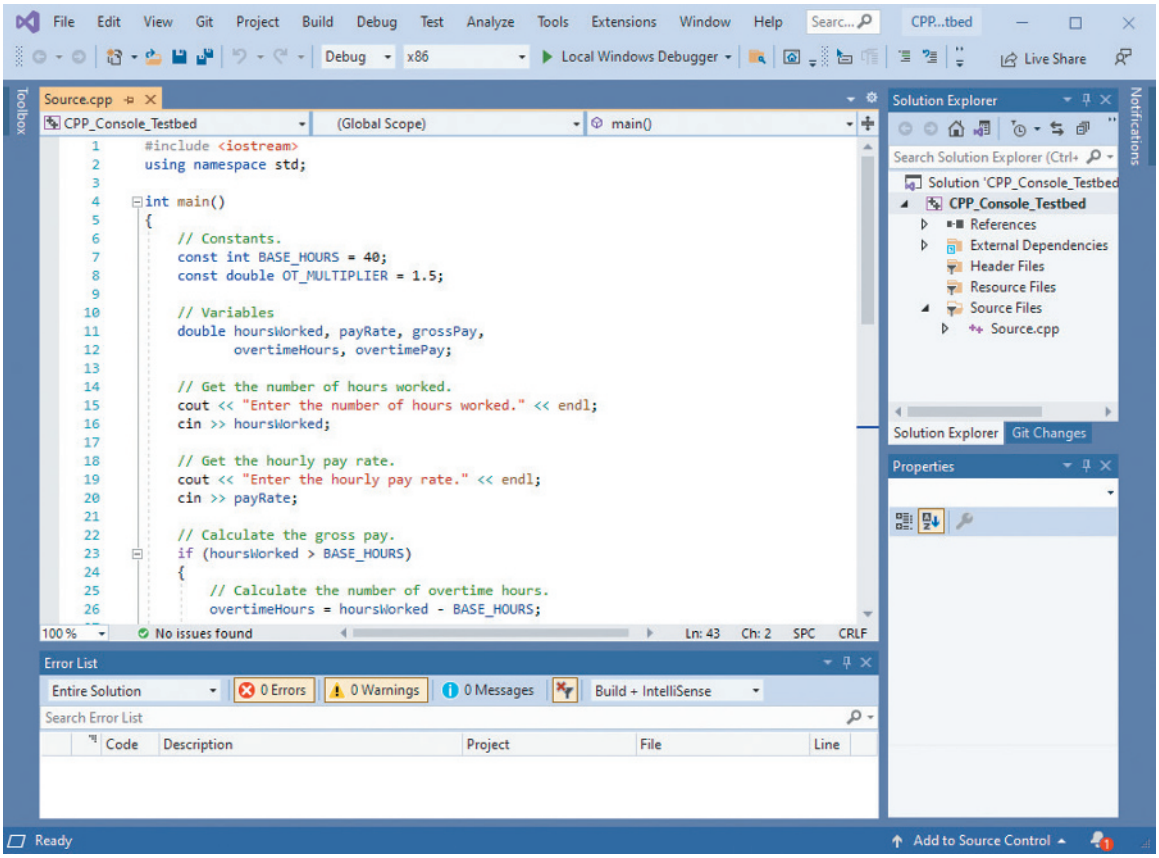
Integrated Development Environments

Although you can use a simple text editor such as Notepad (which is part of the Windows operating system) to write a program, most programmers use specialized software packages called *integrated development environments* or *IDEs*. Most IDEs combine the following programs into one software package:

- A text editor that has specialized features for writing statements in a high-level programming language
- A compiler or interpreter
- Useful tools for testing programs and locating errors

Figure 1-20 shows a screen from Microsoft Visual Studio, a popular IDE for developing programs in the C++, Visual Basic, and C# languages. Eclipse™, NetBeans, Dev-C++, and jGRASP™ are a few other popular IDEs.

Figure 1-20 An integrated development environment (photo courtesy of Microsoft Corporation)



Checkpoint

1.15 A CPU understands instructions that are written only in what language?

1.16 A program has to be copied into what type of memory each time the CPU executes it?

1.17 When a CPU executes the instructions in a program, it is engaged in what process?

1.18 What is assembly language?

1.19 What type of programming language allows you to create powerful and complex programs without knowing how the CPU works?

1.20 Each language has a set of rules that must be strictly followed when writing a program. What is this set of rules called?

1.21 What do you call a program that translates a high-level language program into a separate machine language program?

1.22 What do you call a program that both translates and executes the instructions in a high-level language program?

1.23 What type of mistake is usually caused by a misspelled key word, a missing punctuation character, or the incorrect use of an operator?

1.5 Types of Software

CONCEPT: **Programs generally fall into one of two categories: system software or application software. System software is the set of programs that control or enhance the operation of a computer. Application software makes a computer useful for everyday tasks.**

If a computer is to function, software is not optional. Everything that a computer does, from the time you turn the power switch on until you shut the system down, is under the control of software. There are two general categories of software: system software and application software. Most computer programs clearly fit into one of these two categories. Let's take a closer look at each.

System Software

The programs that control and manage the basic operations of a computer are generally referred to as *system software*. System software typically includes the following types of programs:

Operating Systems. An *operating system* is the most fundamental set of programs on a computer. The operating system controls the internal operations of the computer's hardware, manages all of the devices connected to the computer, allows data to be saved to and retrieved from storage devices, and allows other programs to run on the computer. Examples of operating systems that are widely used today are Windows, macOS, iOS, Android, and Linux.

Utility Programs. A *utility program* performs a specialized task that enhances the computer's operation or safeguards data. Examples of utility programs are virus scanners, file compression programs, and data backup programs.

Software Development Tools. *Software development tools* are the programs that programmers use to create, modify, and test software. Assemblers, compilers, and interpreters are examples of programs that fall into this category.

Application Software

Programs that make a computer useful for everyday tasks are known as *application software*. These are the programs that people normally spend most of their time running on their computers. Figure 1-1, at the beginning of this chapter, shows screens from two

commonly used applications—Microsoft Word, a word processing program, and Microsoft PowerPoint, a presentation program. Some other examples of application software are spreadsheet programs, email programs, Web browsers, and game programs.

Checkpoint

1.24 What fundamental set of programs controls the internal operations of the computer's hardware?

1.25 What do you call a program that performs a specialized task, such as a virus scanner, a file compression program, or a data backup program?

1.26 Word processing programs, spreadsheet programs, email programs, Web browsers, and game programs belong to what category of software?

Review Questions

Multiple Choice

1. A(n) _____ is a set of instructions that a computer follows to perform a task.
   a. compiler
   b. program
   c. interpreter
   d. programming language

2. The physical devices that a computer is made of are referred to as _____.
   a. hardware
   b. software
   c. the operating system
   d. tools

3. The part of a computer that runs programs is called _____.
   a. RAM
   b. secondary storage
   c. main memory
   d. the CPU

4. Today, CPUs are small chips known as _____.
   a. ENIACs
   b. microprocessors
   c. memory chips
   d. operating systems

5. The computer stores a program while the program is running, as well as the data that the program is working with, in _____.
   a. secondary storage
   b. the CPU
   c. main memory
   d. the microprocessor