Chapter 1

Computers and Systems

1.1 The Physical Computer

It is not necessary to understand how a computer works in order to use one—but it helps. An elementary understanding of computer architecture helps demystify the nature and rules of programming languages and enables one to use these languages more wisely. It is essential for anyone who needs to attach devices to a computer or buy one wisely. The architecture of modern computers varies greatly from type to type, and new developments happen every year. Therefore, it is impossible to describe how all computers work. The following discussion is intended to give a general idea of the elements common or universal to personal computers and workstations today.

The main logical parts of a computer, diagrammed in Figure 1.1, can be roughly compared to parts of a human body:

- The **CPU** (central processing unit) is the brain of the computer.
- The computer's **RAM** (random access memory) chips are its memory.
- The **bus** is the nervous system; it carries information between the CPU and everything else in the computer.
- The **input** devices (e.g., keyboard) are the computer's senses.
- The **output** devices (e.g., monitor) are the computer's effectors (hands, voice).

1.1.1 The Processor

In a typical modern computer, the central processing unit (CPU) is the main element on the processor chip. The CPU controls and coordinates the whole machine. It contains a set of **registers**:

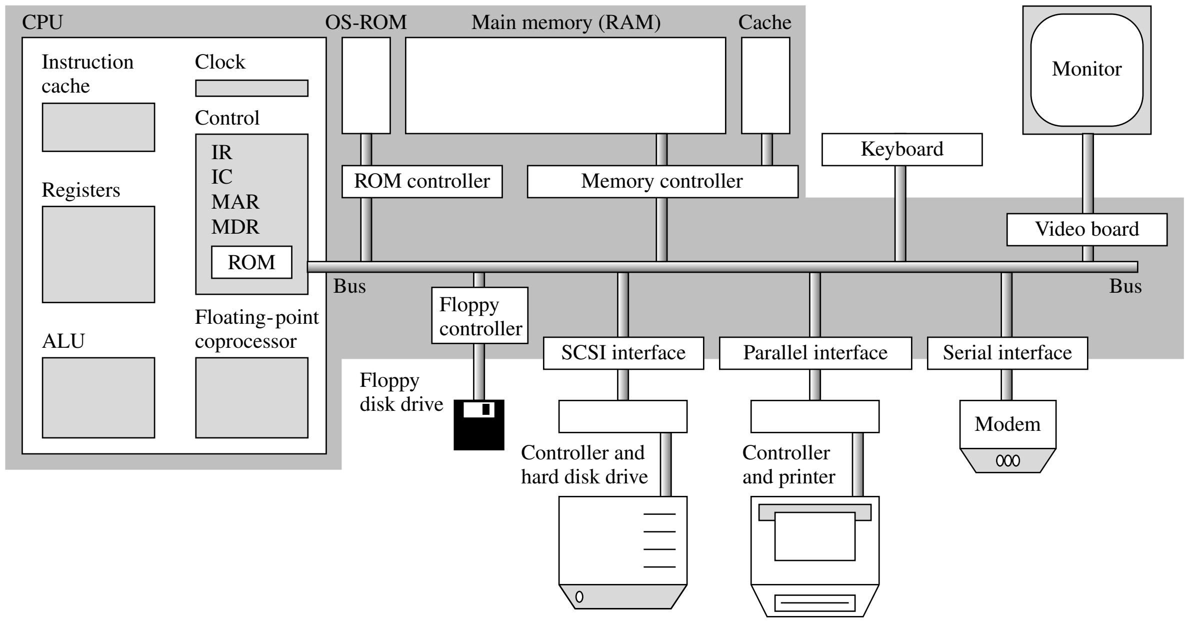


Figure 1.1. Basic architecture of a modern computer.

- The instruction register (IR) holds the current machine instruction.
- The instruction counter (IC) holds the address of the next machine instruction.
- The memory data register (MDR) holds the data currently in use.
- The memory address register (MAR) holds the address from which the data came.

One of its components is the **clock**, which ticks at a fixed rate and controls the fundamental speed at which all of the computer's operations work. The clock rate on a microprocessor chip is set as fast as is (conservatively) possible without causing processing errors. This setting is the megahertz (MHz) rating published by the manufacturer.

The ALU. The *arithmetic and logic unit* (ALU) is the part of the processor containing the many circuits that actually perform computations. Typically, an ALU includes instructions for addition, negation, and multiplication of integers; comparison; logical operations; and other actions. Many computers also have a **floating-point coprocessor**, for handling arithmetic operations on real (floating-point) numbers. Floating-point instructions are important for many scientific applications to achieve adequate accuracy at an acceptable speed. Taken together, this set of instructions forms the **machine language** for that particular processor.

Control ROM and the instruction cycle. A small read-only memory inside the control unit contains instructions (called *microcode*) that control all parts of the CPU and define the actions of the instruction cycle, the ALU, and the instruction cache (discussed next).

To use a computer, we write a **program**, which is a series of instructions in some computer language. Before the program can be used or run, those instructions must be translated into machine language and the machine-language program must be loaded into the computer's main memory. Then, one at a time, the program's instructions are brought into the processor, decoded, and executed. A processor executes the program instructions in sequence until it comes to a "jump" instruction, which causes it to start executing the instructions in another part of the program. A typical instruction brings data into one of the registers, sends data out to the memory or to an output device, or executes some computation on the data in the registers.

1.1.2 The Memory

The memory of a computer consists of a very large number of storage locations called **bits**, which is short for "binary digits." Each bit can be turned either off (to store a 0) or on (to store a 1). All memory in a computer is made out of bits or groups of bits, and all computation is done on groups of bits. The bits in memory are organized into a series of locations, each with an address. In personal computers, each addressable location is eight bits long, called a **byte**. In larger computers and older computers, the smallest addressable unit often is larger than this; in a few machines, it is smaller. Figure 1.2 is a diagram of main memory, depicted as a sequence of boxes with addresses.

A byte could contain data of various types. It could contain one character, such as *'A'*, or a small number. The range of numbers that can be stored in one byte is from 0 to 255 or from -128 to $+127$, which is not large enough for most purposes. For this reason, bytes generally are grouped into longer units. Two bytes are long enough to contain any integer between 0 and 65,536, while four bytes can hold an integer as large as 2 billion.[1]

Traditionally, a **word** is the unit of data that can pass across the bus to or from main memory at one time.[2] Small computers usually have two-byte words; workstations and larger computers have words that are four bytes or longer.

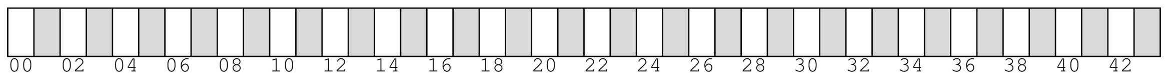
A computer might have several different types of **memory** to achieve different balances among capacity, cost, speed, and convenience. The major types are cache memory, main memory, secondary memory, and auxiliary memory. These are diagrammed in Figure 1.3 and discussed next.

Cache memory. The fastest and most expensive kind of memory is a **cache**. Some machines have small caches to speed up access to frequently used data that are stored in the main memory. The first

[1]Number representation is covered in Chapters 7 and 15.

[2]In machines where the bus transports only one byte, *word* means two bytes and *long word* means four bytes.

- Memory is a very long series of bytes; we show only the first few here.
- Every byte has an address (only the even addresses are shown here).
- In this picture, even-address bytes are white; odd-address bytes are gray.
- We diagram the addresses outside the boxes because they are part of the hardware. They are *not* stored in memory.
- The addresses in the diagram begin with the address 0 and continue through 43.



This amount of memory could store

- 44 characters (one byte each), or
- 22 short integers (two bytes each), or
- 11 long integers or single precision floating-point numbers (four bytes each).

Figure 1.2. Main memory in a byte-addressable machine.

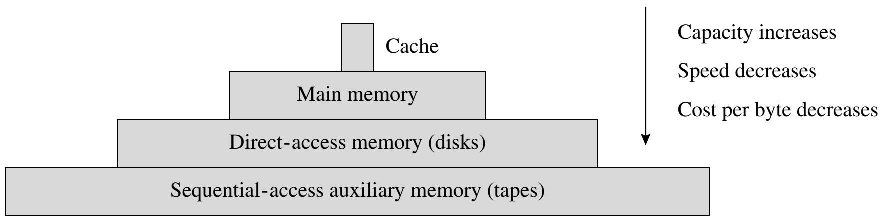


Figure 1.3. The memory hierarchy.

time an item is used, it is loaded into the cache. If it is used again soon, it is retrieved from the cache rather than from the main memory, reducing the access time. Cache memories are small because they are very expensive. Their small capacity limits the extent to which they can improve performance.

Main memory. The computer's main memory (sometimes called *RAM*, for random access memory) is where the active program (or programs) is kept with its data. Sometimes the operating system, which controls the computer,[3] also is kept in RAM; otherwise it is in read-only memory.

[3]Section 1.2 covers operating systems.

OS-ROM memory. **ROM** stands for "read-only memory." An operating system (OS) in ROM is a real convenience for the user for two reasons. First, it is installed in the computer and need not be brought in from a disk. This saves time when the system is turned on. Second, ROM is read-only memory, which means its contents can be read but not changed, either accidentally or on purpose. A partly debugged program sometimes can "run wild" and try to store things in memory locations allocated to some other process that is simultaneously loaded into the computer. With the operating system in ROM, most of the system is protected from this kind of random modification. The only remaining vulnerable part of the system is the area near memory address 0, which is called **low memory** and contains the locations used to communicate between the operating system and the input/output devices.

The disadvantage of using ROM for an operating system is that it is difficult and expensive to improve the system or correct errors in it. When a company wishes to release a new version of a ROM operating system, the code must be recorded on a set of ROM chips. The computer owner must buy a set, remove the old ROM chips, and install the new ones.

Direct-access memory. Direct-access memory, also called *secondary memory*, is needed because even the largest main memory cannot store all the information we need. Data files and software packages are kept in secondary memory when not in use.[4]

A **CD-ROM** (compact disk, read-only memory) is an optical disk storage device, like an audio disc except that it is used to store various types of data, not just music data. A CD-ROM reader contains a laser that reads the minute marks etched into the surface of the disk. Once a CD-ROM has been used to record data, it cannot be reused to record different data. Large collections of data of interest to many people are recorded and distributed on CD-ROMs.

Hard disks and diskettes are the most common direct-access memory devices today. Through the years, the physical size of these devices has decreased steadily, and the amount of information they can hold has greatly increased. As technology has progressed, we have been able to store the bits closer and closer together, enabling us to simultaneously decrease size and increase capacity.

Auxiliary memory. Today's disks can store large volumes of information: 2 gigabytes (2 billion bytes) now is a common disk capacity. However, most businesses and individuals find that 2 billion bytes of secondary memory is not enough to meet all of their needs. Larger, cheaper memory devices are needed to store backup (duplicate) copies of the files on the hard disk and infrequently used files. Magnetic tapes and tape drives meet this need for auxiliary storage. A tape has very large capacity but must be read or written sequentially. This makes retrieving a file from a tape that contains hundreds of files or recording a new file on the end of the tape very slow and inconvenient.

[4]As memory capacities have expanded, our desire to store information has kept pace, so that this statement is as true today as it was when direct-access memories were the size of today's main memories.

1.1.3 Input and Output Devices

Input and output devices permit communication between human beings and the CPU. The most common input devices include keyboards, mice, and track balls. Typically, output is displayed either on a video screen or via a printer. Since humans and the computer speak different languages, some translation is needed. Using a mouse allows the human to point or click at a portion of the screen to convey a screen position and an intent to the computer. Interactions with a keyboard, the screen, or a printer take place using the English language and some hardware-level translation.

In a simple view of the computer world, hitting the Z key on a keyboard causes a *Z* to go into the computer, after which that *Z* can be sent to the video screen or a printer and reappear as a *Z*. While this usually is true, it is very far from a direct or necessary connection.

Consider the keyboard. When you type the key in the lower left (usually marked with a *Z*), the information that goes into the keyboard controller is the coordinates of that key, not a *Z*. Somewhere a code table says "bottom row, first key means *Z*." When the same keyboard is used in Germany, the key in that position is marked with a *Y*, and the code table says "bottom row, first key means *Y*." Similarly, changing the wheel on a daisy-wheel printer changes the letter that prints on the paper.

Translation codes are arbitrary, and several codes are in common use. The most common character code for personal computers is named ASCII (American Standard Code for Information Interchange), a seven-bit code that supports upper- and lowercase alphabets, numerals, punctuation, and control characters. Each device has its own set of codes, but the codes built into one device are not necessarily the same as the codes built into the next device. Some characters, especially characters like tabs, formfeeds, and carriage returns, are handled differently by different devices and even by different pieces of system software running on the same device. For example, the character whose ASCII code is 12 is named *formfeed* in the ASCII code table. The idea of the code makers was that a printer would eject the paper when it received this code. The program in each printer's controller was supposed to look for this code and handle it, and most did. Today, also, some video controllers are programmed to respond to a formfeed character in an analogous way, by clearing the screen.

As a computer user, you need to be aware that equipment not designed to be used together might be incompatible in unexpected ways. You may find computer systems in which the label on the key that you type, the letter that shows on the screen, and the one that comes out of the printer are all different. This can happen because all three depend on software interpretation as well as hardware capabilities.

1.1.4 The Bus

The bus is the pathway between the processor and everything else. It consists of two sets of wires: one set has enough wires to transmit an address; the other set transmits data. When the processor needs a data item, it puts the address of the item in the memory address register (MAR) and issues a `fetch` command. The memory address register and the memory data register (MDR) sit at the end of the bus line (see Figure 1.1).

When a `fetch` command is given, the address goes from the MAR out over the bus's address lines and a copy of the required data comes back over the bus's data lines to the MDR. Similarly, to store information into the memory, the information and the target address are put into these two registers and a `store` command is given. The information goes out over the bus to the given address and replaces whatever used to be there.

Control of peripherals and interfaces. The input and output devices are attached to the bus lines. Each device has its own address and handles the information in its own way. Each device, in fact, has a different set of instruction codes that it can handle and a controller to carry out those instructions. A **device controller** is a small processor connected between the bus and the device that is used to control the action of the device. For example, consider a hard disk. A disk has a controller that understands how to get addresses, disk instructions, and information off the bus and how to put information and signals back on the bus. It knows how to carry out and oversee all the disk operations.

Making all the highly varied devices respond to instructions in a uniform way is the job of **device drivers**. A device driver consists of software that knows about the specific quirks and capabilities of a specific device. It translates the uniform system commands into a form that the device can handle prior to putting the commands on the bus lines. A different driver may be needed for every combination of operating system and hardware. For example, a UNIX driver for a SCSI[5] disk would translate the UNIX `read-disk` command to the SCSI format. The user becomes aware that device drivers exist when he or she wants to install a new device and must also install the appropriate driver.

Between a device's driver and its controller is an interface, a doorway between the bus and the device. There are many kinds of interfaces, with varied transmission properties, which can be classified into two general groups: serial and parallel. A **serial interface** transmits and receives bits one at a time. A **parallel interface** can transmit or receive a byte (or more) at a time, in parallel, over several wires. Parallel interfaces commonly are used for printers; serial interfaces for modems, certain printers, mice, and other slow-speed devices. A MIDI (musical instrument digital interface) is a serial interface used to communicate with electronic musical instruments such as synthesizers and keyboards.

1.1.5 Networks

Inside the computer, the various hardware components communicate with each other using the internal bus. It now is common practice to have computers communicate with each other to share resources and information. This is made possible through the use of networks, physical wires (often phone lines) along which electrical transmissions can occur. The extent of these networks is varied. A **local area network** typically joins together tens of computers in a lab or throughout a small company. A **global network**, such as the Internet, spans much greater distances and connects hundreds of thousands of machines but is truly just a joining together of the smaller networks via a set of gateway computers. A **gateway** computer is a bridge between a network such as the Internet on one side and a local network on the other side. This computer also often acts as a **firewall**, whose purpose is to keep illegal, unwanted, or dangerous transmissions out of the local environment.

[5]The small computer systems interface (SCSI) is a standard disk interface.

This is one way that a lab network might be set up. It is not drawn to scale; a hub is a small device that could fit on the corner of a table.

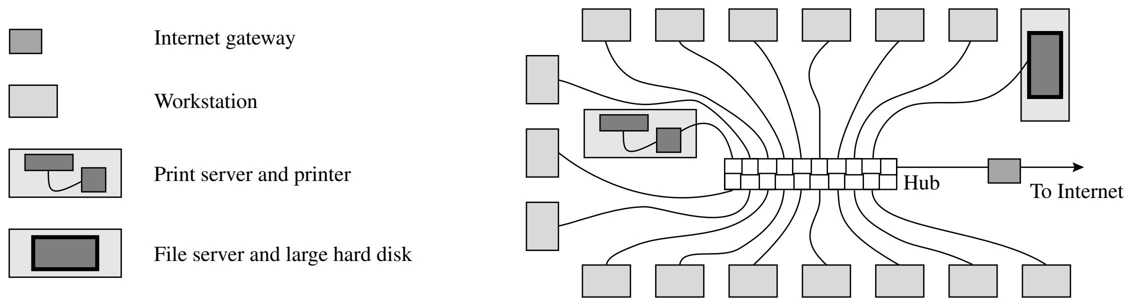


Figure 1.4. A local network in a student lab.

Sharing resources. One use of networks is to let several computers share resources such as file systems, printers, and tape drives. The computers in such a network usually are connected in a **server-client** relationship, as illustrated in Figure 1.4. The server possesses the resource that is being shared. The clients, connected via a **hub** or **switched ethernet connection**, share the use of these resources. The user of a client machine may print out documents or access files as if the devices actually were physically connected to the local machine. This can provide the illusion of greater resources than actually exist, as well as present a uniform programming environment, independent of the actual machine used. This kind of sharing is less practical over larger networks due to delays caused by data transmissions through gateway machines.

Communication. The other typical use of networks is communication. E-mail has become a popular way to send letters and short notes to friends and business associates. Chat rooms provide the opportunity for more direct, interactive communication. The World Wide Web makes a wealth of information
available to the average user at home that used to be available only in distant libraries. It also provides new commercial opportunities; people now can shop for many items on the Web and are able to buy specialty items or get bargains that were previously unavailable to them.

Networks also have changed the workplace and work habits. Many professionals use network transmissions between home and office or between two office locations to gain access to essential information when they need it, wherever they are. This may include using a laptop computer that is connected to the network via a cellular phone.

Distributed computing. Networks also are used to allow the computers to communicate between themselves. The complexity of many of today's problems requires the use of reserve computing power. This can be achieved by synchronizing the efforts of multiple computers, all working in parallel on separate components of a problem. A large distributed system may make use of thousands of computers. Synchronization and routing of information in such systems are major tasks, among the many performed by the operating system's software, as discussed next.

1.2 The Operating System

The **operating system** (OS) is the most important piece of system software and the first one you see when you turn on your machine. It is the master control program that enables you to use the hardware and communicate with the rest of the system software. The operating system has several major components, including the **system kernel**, which is the central control component; a **memory management system**, which allocates an area of memory for each program that is running; the **file system manager**, which organizes and controls use of the disks; **device drivers**, which control the hardware devices attached to the computer; and the **system libraries**, which contain all sorts of useful utility programs that can be called by user programs. In addition, a multiprocessing system (described later) has a process scheduler, which keeps track of programs waiting to be run and determines when and how long to run each one.

Command shells and windows. Operating systems can be divided into two categories: command-line interpreters and windowing systems. **Command shells**, or command-line interpreters, were invented first and can be used on small, simple machines. A command shell displays a system prompt at the left of the screen and waits for the user to type in a command. Then, the command is executed by calling up some piece of system or user software. When that program terminates, control goes back to the operating system and the system prompt again is displayed.

A newer idea is a **windowing system**. Window-based systems include the Apple Macintosh system, Microsoft Windows (which is an extension of DOS), and NextStep and X-Windows, which provide window interfaces for UNIX. Except for the Macintosh, these windowing systems can run side-by-side with a command shell and provide access to it. This is important because command-line interpreters generally provide capabilities that are not available within the window environment.

In a window environment, multiple windows can be displayed on the screen, including perhaps a command window. Windows can be used to display file directories, run programs, and so forth. The user accesses the contents of the windows using a mouse or some other point-and-click device. Seeing your files, moving and copying them, renaming them, and every other thing that you do is much easier in a window environment.

Multiprogramming systems. Another way to categorize systems is by whether they can run several programs at the same time or are limited to one at a time. Ordinary personal computers are limited to one process at a time. However, modern personal workstations and large computers, often called *mainframes*, have **multiprogramming** operating systems. UNIX is one of the best known and most widely used multiprogramming systems.

Workstations are capable of running a few processes concurrently, and mainframes often can support 50 or 100 users, running the programs in a **time-shared** manner. In time sharing, each user process is given a short slice of CPU time, then it waits while all the other users get their turns. This works because users spend much more time thinking and typing than running their programs. Any request by a program for input or output (I/O) also ends a time slice; the OS initiates the input or output, then selects another process to be run while the I/O happens. The process scheduler is the system component that coordinates and directs all this complex activity.

Each kind of computer must have its own custom-tailored operating system. Some systems are proprietary and have been implemented for only one manufacturer's models. For example, Apple's system for the Macintosh is jealously guarded against copying. Other systems, such as UNIX and DOS, are widely implemented or imitated. Increasingly, the computer owner has a choice about what system will be installed on his or her hardware.

The choice of hardware is important because it determines what software you can run and what diskettes you can read. Software and file systems are constructed to be compatible with a particular system environment, and they do not work with the wrong system. For example, you cannot read a UNIX diskette in a DOS system, and a C **compiler** that works under UNIX must be modified to work under DOS. Windowing systems and multiprogramming are powerful aides to program development. However, both consume large amounts of main memory, disk space, and processing time. Trying to run them with a machine that is not big enough or fast enough is a mistake.

1.3 Languages

The purpose of a computer language is to allow a human being to communicate with a computer. Human language and machine language are vastly different because human capabilities and machine capabilities are different. Each kind of computer has its own machine language that reflects the particular capabilities of that machine. Computer languages allow human beings to write instructions in a language that is more appropriate for human capabilities and can be translated into the machine language of many different kinds of machines.

1.3.1 Machine Language

Built into the CPU of each computer is a set of instructions that the hardware knows how to execute. The behavior of each instruction and its binary code are documented in the hardware manual. Technically, it is possible to program a computer by making lists of these codes. That is how programming was done 45 years ago.

A machine language program is a sequence of instructions, each of which consists of an instruction code, often followed by one or two register codes or memory addresses. In a machine, these are all represented in binary.[6] In the early days of computers, when people still wrote programs in machine

[6]Binary is the base 2 number system. Information is represented as strings of bits (see Chapter 15).

language, they did not write them in binary because people were (and are) abominably bad at writing long strings of 1's and 0's without making errors. Instead, they used the octal number system,[7] in which information is represented as strings of digits between 0 and 7. Each octal digit translates directly into three binary bits. Thus, a machine language program was written as a long series of lines, where each line was a string of octal digits.

1.3.2 Assembly Languages

When people had to use machine language, it took a very long time to write and debug a program. The next development was symbolic assembly language. Instead of writing octal codes, the programmer wrote symbolic codes for instructions and defined a name for each data-storage location. The three lines that follow show how a simple action might look when expressed in assembly language; this code adds two numbers and stores them in a variable named `sum`. The same addition expressed in C would be `sum = n1 + n2;`

```
ldreg *n1, d1   / Load first number into register d1.
add   *n2, d1   / Add second number to the register.
sto   d1, sum   / Store result in the location for sum.
```

A translator, called an **assembler**, analyzed the symbolic codes and assembled machine-code instructions by translating each symbol into its code and assigning memory locations for the data objects used by the program.

Every name and quoted string used in a program must be stored at some address in the computer's memory. To write in machine language, you manually assign an address to each object. Happily, assembly languages and high-level languages such as C free you from concern over these addresses. The programmer declares the names to be used at the top and defines each name by giving it a data type or quoted string value. When the assembler translates this into machine language, it assigns memory locations for these objects.

Assembly languages still are very important for writing programs so closely related to the hardware that high-level languages like C simply have no commands to express them. Many large systems are written primarily in a high-level language but contain some parts coded in assembly language. These portions are part of the system kernel. They work directly with parts of the machine and must operate as efficiently as possible.

1.3.3 High-Level Languages

Programming in an assembly language is very tedious. Furthermore, an assembly language is specific to one type of machine and probably very different from assembly languages for machines of other manufacturers. Therefore, assembly language programs are not portable—that is, they are not easily converted for use on other machines. In contrast, programs in languages like FORTRAN and C are highly portable, because compilers for these languages have been created for nearly every kind of computer.

[7] Octal is base 8.

Over the past 40 years many high-level languages have been developed, some of which have stood the test of time and some of which have not. Each instruction in a high-level language translates into several at the assembly or machine language level. Programs written in a high-level language appear much more like English and are more understandable to humans. In this section, we discuss a few widely used programming languages.

The C language. C is a relatively old[8] language that recently became very popular. It has characteristics of both high-level languages such as Pascal[9] and FORTRAN[10] and low-level languages such as assembly language. You still might see several dialects of C in older programs or books. In 1988, however, the American National Standards Institute (ANSI) adopted a standard for the language, known as ANSI C. This standard was adopted with a few minor changes by the International Standards Organization (ISO) in 1990 and amended in 1994. This new standard is known as ISO C. The phrase *standard* C can be applied correctly to either version of the standard. The changeover to the standard language is nearly complete, so the beginning C student can safely focus all efforts on standard ISO C.

There are several reasons for the recent growth in the use of C:

- As the UNIX operating system has spread, so has C. C is the tool by which much of the power of UNIX is accessed. UNIX has spread because of the very large amount of valuable software that runs under it.
- From the beginning, C was a very powerful language and fun for the experienced programmer to use. However, it lacked certain important kinds of compile-time error checking and therefore was quite difficult for a beginner to use. ISO C incorporates important new error-checking features and has eliminated many hardware dependencies. It developed into a much better language and became suitable for both beginners and experienced programmers.
- C allows large programs to be written in separate modules. This makes it easier to manage large projects, greatly facilitates debugging, and makes it possible to reuse program modules that do common, useful jobs.
- The ISO C library is extensive and standardized. It contains functions for mathematical computation, input and output facilities, and various system utilities.

On the other hand, ISO C remains more error prone than languages of a similar age with similar features, such as Pascal and Ada.[11] It is popular among experienced programmers partly because of the features that cause this error-prone nature:

- C allows the programmer to write terse, compact code that can run very efficiently. Any programmer who is a slow typist appreciates this. Sometimes programmers even make a game of

[8]Created originally in 1972, the language has been updated and expanded several times.

[9]Pascal has been used most extensively as an instructional language in universities.

[10]Short for "formula translator," this language was developed primarily for doing scientific calculations.

[11]Ada is a programming language developed in the 1970s to support large-scale, portable application systems.

squeezing the unnecessary operations out of their code. On the negative side, compact code can be hard to read and understand unless comments are used liberally to explain it.

- The error-checking system in ISO C is less rigid and more permissive than in competing languages. Expert programmers claim that this permissiveness is an advantage and that the rigid mechanisms in other languages often "get in the way" when they want to do something unusual. However, these rigid systems are easier for the new programmer to understand and to use.
- C supports bit-manipulation operators that can select or change a single bit or group of bits in a number. These are very important in system programs that must interface to hardware devices that set and test values in specific memory locations. However, working with arbitrary machine addresses and bit patterns must be done with extreme care to avoid errors.
- No restrictions are placed on the use of pointers. (A **pointer** is a variable representing the location, as opposed to the value, of data.) This permits the use of some very efficient computational methods, at the potential cost of destroying information anywhere in memory when an error is made in setting a pointer value. Unfortunately, such errors are common, and many result in system crashes and the need to reboot the system.

C and FORTRAN. FORTRAN is a very old language that has been used by engineers and scientists since the infancy of computers. Originally, it was a language for scientific computation, and it still serves that purpose. Over the years, the language has been updated, revised, and expanded, but its primary focus remains high-performance numerical computation. A massive amount of scientific programming now exists in the form of FORTRAN libraries and FORTRAN application programs that are used, and shared, by scientists worldwide. The FORTRAN libraries are extremely efficient, reliable, and trusted.

Because many engineering departments are acquiring UNIX workstations, C is beginning to supplant FORTRAN-77 in many engineering applications. This has some advantages. FORTRAN-77 still bears the burden of being an old language. It is full of unnecessary complications and nonuniform conventions. Much of the space in a FORTRAN textbook is spent explaining how to *write* the language correctly. In contrast, a C textbook has a much simpler language to present and can spend more time explaining how to *use* the language well.

On the other hand, FORTRAN-77 is a "safer" language. A program can get into trouble in very few ways that will cause a system crash or cause the result of a seemingly correct expression to be nonsense. C is prone to these problems, even when the programmer avoids using the advanced parts of the language. When a C programmer begins to use pointers, debugging becomes substantially more difficult than it ever could be in FORTRAN-77. Nonetheless, C is here, and thousands of former FORTRAN programmers are beginning to use it. FORTRAN-to-C conversion programs exist and are being used to make the transition less costly.

C and C++. The C++ language extends C to eliminate more causes of error and provide software-engineering tools that are important for large projects. Also, C++ (but not C) is fully compatible with the FORTRAN libraries. This can be a very important consideration for a department switching from

FORTRAN to C. C++ is a superset of C. The ordinary line-by-line code in a C++ program is written in C. The extensions involve the way code is organized into modules and the way these modules are used. The C++ extensions are a powerful tool for program organization and error prevention. However, since the entire C language is included as a subset of C++, any error that you can make in C also can be made in C++. The advantages of C++ are there only for those who know how to use them. For beginners, C++ is a more difficult and confusing language than C.

The differences between C and C++ become significant only for moderate to large programs, and only when C++ is used with proper object-oriented design techniques. All software-engineering techniques presented in this book are appropriate for use with both C and C++. The way in which C language elements are presented will lead toward an understanding of the design requirements for C++.

1.4 What You Should Remember

1.4.1 Major Concepts

- This chapter provides a brief description of computer hardware and software. It describes the parts of the machine a programmer must know to comprehend the operation of a program or buy a personal computer system wisely.
- Computer languages and the process of translation are discussed, and the C language is compared to FORTRAN and C++.

1.4.2 Vocabulary

The terms and concepts that follow have been introduced and described briefly. The first and second columns contain terms related to computer hardware and operating systems; the third column relates to the programming process.

CPU	device controller	program
register	device driver	machine language
memory	serial interface	operating system
cache	parallel interface	system kernel
ROM	local area network	system libraries
CD-ROM	global network	command shell
RAM	gateway	windowing system
bit	firewall	multiprogramming
byte	server	assembler
word	client	compiler
clock	memory management system	ANSI C
bus	file system manager	ISO C
hub	floating-point coprocessor	C++

1.5 Using Pencil and Paper

1.5.1 Self-Test Exercises

1. Which terms on the vocabulary list relate to the computer's processor?
2. Which terms on the vocabulary list relate to the memory of a computer?
3. Which terms on the vocabulary list relate to the peripherals of a computer?
4. Which terms on the vocabulary list relate to a computer network?
5. Which terms on the vocabulary list relate to system software?
6. For what does each of the following abbreviations stand?

a.	ALU	f.	ANSI	k.	LAN
b.	bit	g.	I/O	l.	MIDI
c.	CPU	h.	OS	m.	SCSI
d.	ISO	i.	ROM	n.	MHz
e.	ASCII	j.	WAN	o.	RAM

1.5.2 Using Pencil and Paper

1. Choose three terms from *each column* of the vocabulary list in Section 1.4.2. In your own words, give a brief definition for each (a total of nine definitions).
2. What computer will you use for the programming exercises in this course? What kind of processor chip does it have? How big is its main memory? What input and output devices are available for it? Is it attached to a computer network?
3. Have you used a local area network? Why? Have you used the Internet? For what purposes?
4. What operating system runs on the computer that you will use for the programming exercises in this course? Is this a multiprogramming system? What compiler will you use?
5. Explain the difference between

   (a) a byte and a word.
   (b) ROM and RAM.
   (c) cache memory and main memory.
   (d) a device controller and a device driver.
   (e) a compiler and an assembler.
   (f) a command shell and a windowing system.
   (g) a LAN and a WAN.
   (h) a gateway and a hub.