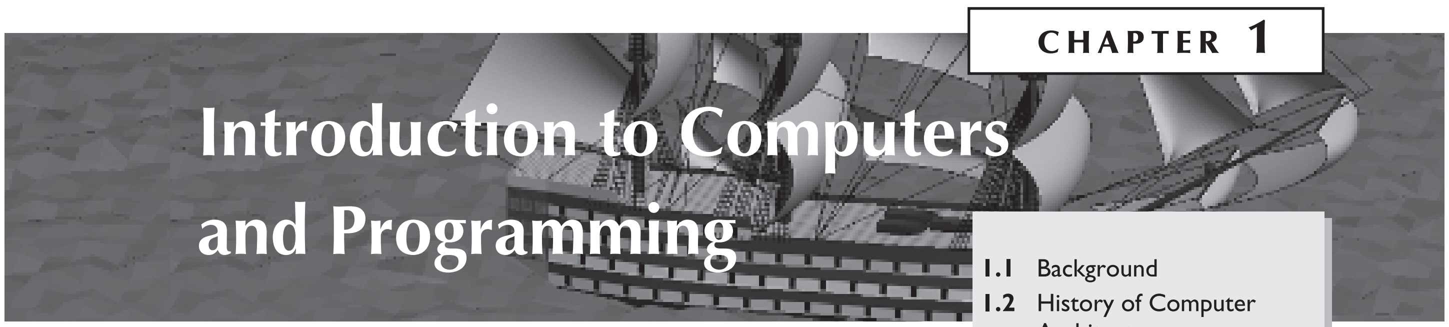
CHAPTER 1



Introduction to Computers and Programming

Chapter Objectives

This chapter presents an overview of the historical background of computing and the computer hardware and software concepts that build the foundation for the rest of this book:

- Hardware architectures
- Software categories
- Programming languages
- Anticipated outcomes

1.1 Background

Advances in technology are achieved in two steps as follows:

- A visionary conceives an idea that has never been tried before
- Engineers find or invent tools that will bring that vision to reality

The search for new software tools is therefore an inescapable part of an engineer's life. The process of creating these tools frequently spawns sub-problems, which themselves require creative solutions. The pace of change in our world is increasing, and nowhere is this phenomenon more dramatically obvious than computer science. In the span of just a few generations, computers have invaded every conceivable aspect of our lives, and there is no indication that this trend is slowing.

This book will help you become familiar with one specific programming tool: MATLAB. It is intended to bring you to a basic proficiency level so that you can confidently proceed on your own to learn the features of other programming languages that are useful to your interests.

A word of caution: Learning a programming language is very much like learning to speak a foreign language. In order to find something to eat in Munich, you must be able to express yourself in terms a German can understand. This involves knowing not only some vocabulary words, but also the grammatical rules that make those words comprehensible—in German, for example, this means putting the verbs at the ends of phrases.

If languages were a strictly theoretical exercise, you could make up your own vocabulary and grammar, and it would undoubtedly be an improvement over existing languages—especially English, with its incredibly complex spelling and pronunciation rules. However, language is not a theoretical exercise; it is a practical tool for communication, so we can't make up our own rules, but are constrained to the vocabulary and grammar expected by the people with whom we want to converse.

Similarly, this book is not an abstract text about the nature of computer languages. It is a practical guide to creating solutions to problems. Accomplishing this involves expressing your solutions in such a form that the computer can "understand" your solutions; therefore, it requires that you use the vocabulary (i.e., the appropriate key words) and grammar (the syntax) of the language.

To become proficient in this, as in any other language, it is not enough to merely know the grammar and vocabulary. You have to practice your language skills by communicating. For foreign languages, this means traveling to the country, immersing yourself in the culture, and talking with people. For computer languages, this means actually writing programs, seeing what they do, and determining how to use their capabilities to solve your engineering problems.

1.2 History of Computer Architectures

Computing concepts developed as tools to solve previously intractable problems. This section will trace the growth of computing architectures, review the basic organization of computer hardware components, and emphasize the implementation of the data storage and processing capabilities by highlighting three milestones on the road to today's computers: Babbage's difference engine, Colossus, and the von Neumann architecture.

1.2.1 Babbage's Difference Engine

Charles Babbage (1791–1871) is generally recognized as the earliest pioneer of the modern computer. Babbage's **difference engine**—a relatively simple device that can subtract adjacent values in a column of numbers—is a good example of a computing device designed to improve the speed and repeatability of mathematical operations. Babbage was concerned about the process engineers used to develop the tables of logarithms and trigonometric functions. In his day, the only way to develop these tables was for mathematicians to calculate the values in the tables by hand. While the algorithms were simple—combining tables of the differences between adjacent values—the opportunity for human error was unacceptably high. In 1854 Babbage designed a difference engine that could automate the process of generating tables of mathematical functions. Since the objective was to create numerical tables, the output device was to be a set of copper plates ready for a printing press. The memory devices for storing numerical values were wheels arranged in vertical columns. The arithmetic operations were accomplished by ratchet devices cranked by hand.

Sadly, the manufacturing tools and materials available then prevented him from actually building his machine. However, in 1991 the Science Museum in London built a machine to his specifications, as shown in Figure 1.1. With only minor changes to the design, they were able to make the machine work. Although limited in its flexibility, the machine was able to compute difference equations up to the seventh order with up to 13 significant digits.

1.2.2 Colossus

Colossus was a computing machine developed to solve large, complex problems quickly. Early in the Second World War, Britain was losing the Battle of the Atlantic—German U-boats were sinking an enormous number of cargo ships that were resupplying the Allied war effort. The Government

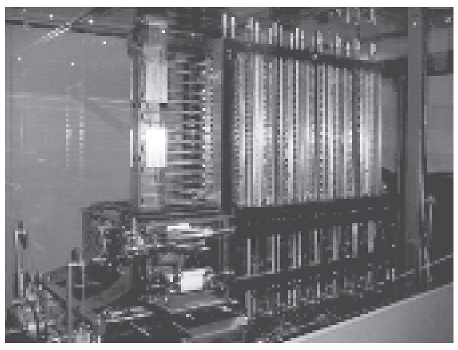


Figure 1.1 *Babbage's difference engine*

Code and Cypher School was established at Bletchley Hall in Britain with the goal of breaking the German codes used to communicate with their U-boats in the North Atlantic. They were using Enigma machines, relatively simple devices that encrypted messages by shifting characters in the alphabet. However, to crack the code they needed to exhaustively evaluate text shifted by arbitrary amounts. Although the algorithm was known, the manual solution took too long, and it was often too late to make use of the information. A computer later named Colossus (see Figure 1.2) was designed by Max Newman and was custom built for this purpose. While not a general-purpose processor, Colossus was fast enough to crack all but the most sophisticated Enigma codes. Sadly, due to security concerns, the machine was destroyed when the war ended. However, the dawn of ubiquitous computing was breaking, and general-purpose computers were soon to be available.

1.2.3 The von Neumann Architecture

These and other contemporary achievements demonstrated the ability of special-purpose machines to solve specific problems. However, the creativity of John von Neumann ushered in the current era of general-purpose computing in which computers are flexible enough to solve an

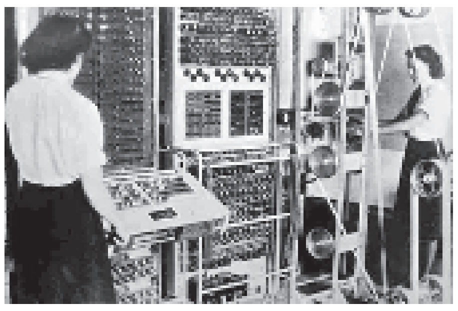


Figure 1.2 *Colossus*

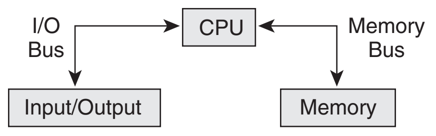


Figure 1.3 *von Neumann architecture*

astonishing array of different problems. Dr. von Neumann proposed a computer architecture that separated the Central Processing Unit (CPU) from the computer memory and the Input/Output (I/O) devices (see Figure 1.3).

Together with binary encoding for storing numerical values, this was the genesis of general-purpose computing as we know it today. Although the implementation of each component has improved beyond recognition, the fundamental processing architecture remains unchanged today.

1.3 Computing Systems Today

Today's computing systems—the combination of hardware and software that collectively solve problems—retain many of the key characteristics of these inventions: they process more data than is humanly possible, quickly enough for the results to be useful, and they basically follow the von Neumann architecture. Computer **hardware** refers to the physical equipment: the keyboard, mouse, monitor, hard disk, and printer. The **software** refers to the programs that describe the steps we want the computer to perform.

1.3.1 Computer Hardware

All computers have a similar internal organization, as shown in Figure 1.4, that is closely related to the von Neumann architecture. The CPU is usually separated into two parts: the Control Unit, which manages the flow of data between the other modules, and the Arithmetic and Logic Unit (ALU), which performs all the arithmetic and logical operations required by the software.

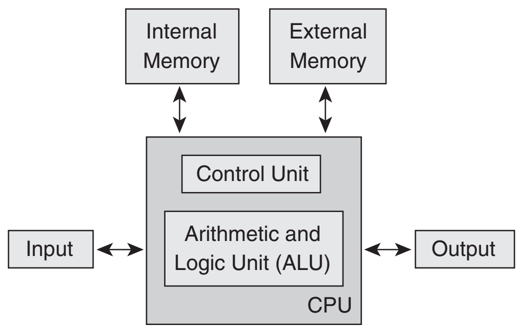


Figure 1.4 *Internal organization of a computer*

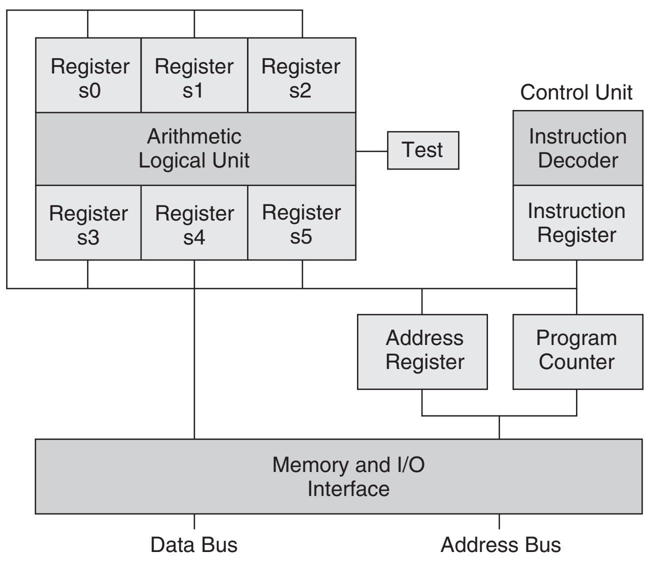


Figure 1.5 *Internal computer details*

The individual logic devices that comprise the electronic components of the computer operate in a binary mode, which is represented electrically by the presence or absence of voltage at a connection. These states, called **bits**, have the value of 1 (present) or 0 (absent). Most computer operations assemble these bits into larger collections—a **byte** being 8 bits, and **words** consisting of 16, 32, 64, or more bits. We refer to the data items coming into the computer as the **input**, and the results coming from the computations as the **output**.

Input and output (I/O) is accomplished by moving data between the memory and external equipment designed to communicate with users or other computers. In the early days, all devices had to be individually installed in the computer with dedicated wiring—a process called **hardwiring**. In contrast, today this is usually accomplished merely by plugging devices into one of many **data buses** (see Figure 1.5). A data bus is an electronic "pathway" for transporting data between devices. Since most devices expect to be able to send data on the bus as well as receive data from it, data bus design always involves a protocol that ensures that only one device is writing to the bus at any given time.

1.3.2 Computer Memory

Memory comes in many forms. Not long ago it could be nicely divided into two categories—solid state and mechanical. Solid-state memory modules were directly connected to the processor and used digital addresses to save and restore data. Mechanical memory relied upon devices that moved rewriteable storage media past sensors that converted the impressions on the storage media to digital form. Tape drives, floppy disks, hard drives, and optical disks (CDs and DVDs) share this architecture, and they are usually externally connected to the input/output system. Recently, however, these distinctions have been blurred by the arrival of devices like **flash cards** that are solid-state memory devices but attach to the computer's I/O ports and behave as if they were mechanical memory.

Today, CPUs use many forms of solid-state memory. The first instructions executed when power is turned on are usually stored in **Read-Only Memory (ROM)**, sometimes referred to as the **Basic Input/Output System (BIOS)**. These instructions are just enough to wake up the keyboard and screen in basic mode and look around for a memory device containing the real programs. These real programs are transferred from the memory device, frequently referred to as "mass memory," to **Random-Access Memory (RAM)**—large amounts of high-speed, solid-state memory used to hold all of the programs and data users need immediately.

Most processors achieve significant performance improvement by using smaller amounts of even higher speed memory as **cache**. Cache memory processors are smart devices that "guess" what instructions and data the computer needs next, and preload those guesses into cache memory where the CPU can reach them quickly. These guesses are based on the likelihood that the program will continue linearly through the program as opposed to branching to go somewhere else for the next instruction. A significant amount of today's computer architecture design effort focuses on the effective use of cache memory to improve performance.

As programs become larger and process more data, and the systems allow more than one program to run simultaneously, RAM occasionally fills up. Most operating systems today use **virtual memory**—a data file usually on the hard drive that contains an image of everything you would like to have in RAM divided into **pages**. When the CPU requires access to a page that is not actually in RAM, it has to take the time to find a special area in RAM referred to as a "page buffer" that it can safely use, write its contents back to virtual memory, and read in the page needed. No matter how smart this process might be about looking ahead and predicting required pages, there is always a huge performance loss when a computer begins using virtual memory.

Figure 1.6 illustrates some aspects of how computer memory is managed. The operating system (UNIX, Windows, Mac OS X, or whatever) consumes some memory and determines from the I/O devices available what internal software **(drivers)** must be present to enable the application programs to communicate with the outside world. As mentioned earlier, many programs

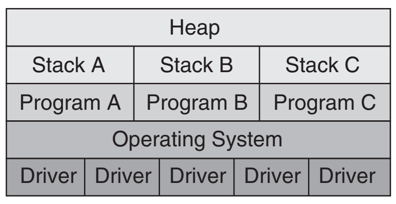


Figure 1.6 *Typical memory layout*

are loaded automatically when the operating system starts, and others are loaded upon user request. In addition to the memory needed to store the instructions, each program is allocated some **stack** space for storing local static data. The remaining memory, the **heap**, is accessible to all programs upon request to the operating system. The heap is typically used to store most of the data being manipulated by the programs. When a program finishes with a block from the heap, it is usually released by that program for other programs to use as necessary.

1.3.3 Computer Software

Computer software contains the instructions that the CPU uses to run programs. There are several important categories of software, including operating systems, software applications, and language compilers. Not all processors need all these facilities. Figure 1.7 illustrates the interactions among these categories of software and the computer hardware, and the following sections describe each in more detail.

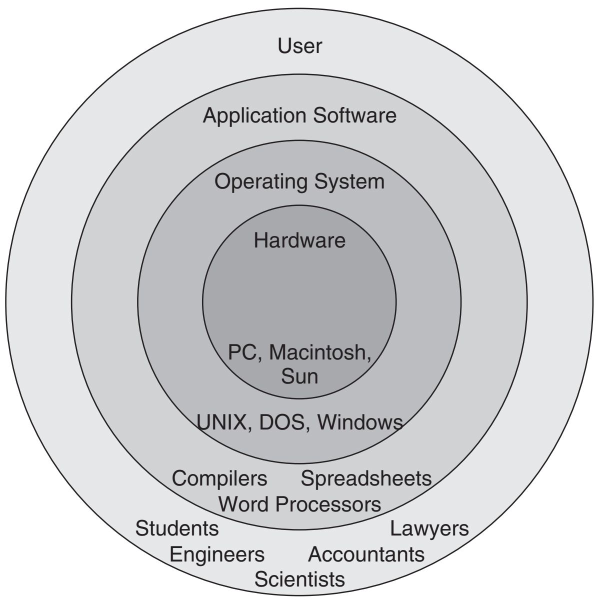


Figure 1.7 *Interactions between computer hardware and software*

Operating Systems The **operating system (OS)** serves as the manager of the computer system as a whole. It controls access to the processor by users and networked devices, and it organizes the hardware and software according to the users' specifications. The operating system is the first major software component fetched by the BIOS from mass storage, and it automatically loads and starts the myriad programs that make computers "user friendly." It also provides the tools for making the computer's peripheral devices—such as printers, scanners, and DVD drives—available to other software. Common modern operating systems are Microsoft Windows, Linux, UNIX, and Apple Mac OS.

Operating systems also contain a group of programs called **utilities** that allow you to perform functions, such as printing files, copying files from one disk to another, and listing the files that you have saved on a disk. Although these utilities are common to most operating systems, the commands themselves vary from operating system to operating system.

While computer systems give the appearance of stability, like automobiles, they require periodic maintenance to maintain peak performance.

- You should protect your computer by installing and configuring utilities that protect it from viruses, intrusive advertising, and external influences that make illegal use of the processor or its data. Refer to the documentation for your specific operating system.
- Over time, most disk drives become fragmented—the available space gets chopped up into smaller and smaller pieces—and the performance of your system begins to suffer. Defragmentation of a large disk drive may be an overnight effort, but should be done periodically.
- While very reliable, computers are not indestructible. You should establish a regular policy of backing up your personal files onto removable media. Most operating systems provide such utilities, and a number of services are now available at a modest cost that automatically back up your files to encrypted storage whenever your computer is connected to the Internet. You do not need to back up commercial software that can be reloaded from the manufacturer's installation disks.

Software Tools Software tools are commercial programs that have been written to solve specific problems. They are highly sophisticated, complex applications that use the facilities provided by the operating system to enable you to create, save, recall, manipulate, and present ideas in the form of data files on your computer. The specific nature of those files depends on the nature of the problem. If you need a well-formatted document or report, **word processors** are programs that enable you to enter and format text and graphics. They allow you to develop documents in outline form; move words, sentences, and paragraphs; and check your spelling and grammar. **Desktop publishing** combines a very powerful word processor with a high-quality printer to produce professional-grade documents.

If you need sophisticated results from tabular data, **spreadsheets** let you work easily with data that can be displayed in a grid of rows and columns. Most spreadsheet packages include plotting capabilities to create charts and graphs, so they can be especially useful in analyzing and displaying information.

If you need to store, quickly retrieve, and format large amounts of data, **database management** programs are useful tools. They are used by large organizations, such as banks, hospitals, universities, hotels, and airlines, to store and organize crucial information; they are also used to analyze large amounts of scientific data. Meteorology and oceanography are examples of scientific fields that commonly require large databases for the storage and analysis of data.

Computer-aided design (CAD) packages let you define computer models of real-world objects, assemble groups of such models, and then manipulate them graphically. CAD packages are frequently used in engineering applications, and the designs of most automobiles and aircraft are now “paperless”—the essential information is in a CAD database rather than on paper.

Programming Languages All programming languages are merely tools a programmer uses to express the logic for a computer to implement. Like any spoken language, a computer language is defined by its grammar **(syntax)** and its vocabulary. There are three necessary attributes of a computer language: the scope of the logic expressed in each line of code (the **power** of the language), the clarity of each line of code from the human viewpoint, and its **portability** between different types of processor. Computer languages are frequently described in terms of **generations** that reflect the development of language power, clarity, and portability.

First-generation, or **machine languages**, are the most primitive languages, usually tied closely to the nature of the computer hardware. Since the basic logic of the CPU is binary, the syntax of machine language is expressed as sequences of 0s and 1s. This maximizes the control over the processor, but results in programs that are completely incomprehensible to anyone, including the original programmer, and are absolutely not portable.

A **second-generation** language, frequently called **assembly language**, is a means of expressing machine language in symbolic form where each line of code usually produces a single machine instruction. While programming in assembly language is easier than machine language, it is still a tedious process that requires each detailed instruction to be specified; and like the first-generation languages, it is completely tied to the nature of the CPU.

Third-generation languages such as C, FORTRAN, and BASIC have commands and instructions that are more similar to spoken languages. One line of code of these languages creates many machine level instructions. Consequently, they are much clearer expressions of the logic of a program, and the power of each instruction is significantly increased. The resulting programs are to some degree portable between processor types. Third-generation languages and beyond are referred to as **high-level languages**.

The **fourth-generation** languages that include Ada and Java take this trend to the next level. They are completely portable between supported processor types, and each line of code creates a significant amount of machine instructions. MATLAB and its close competitors, Mathematica, Mathcad, and Maple, are very powerful fourth-generation languages that combine mathematical functions and commands with extensive capabilities for presenting results in a graphical form. This combination of computation and visualization power makes them particularly useful tools for engineers.

The current language development trend is to allow the programmer to express the overall program logic in a graphical form and have the programming tools automatically convert the diagrams to working programs. Programmers involved with these implementations still need language skills to complete the implementation of the algorithms. The goal of the fifth generation of languages is to allow a programmer to use natural language. Programmers in this generation would program in the syntax of natural speech. Implementation of a fifth-generation language will require the achievement of one of the grand challenges of computer science: computerized speech understanding.

1.3.4 Running a Computer Program

For most computer languages, getting the program to run involves compilation, linking, loading, and then executing the program. These processes are outlined in Figure 1.8.

Compilation: Programs written in most high-level languages, such as C or Java, need to be compiled (i.e., translated into machine language) before the instructions can be executed by the computer. A special program called a **compiler** performs this translation. Thus,

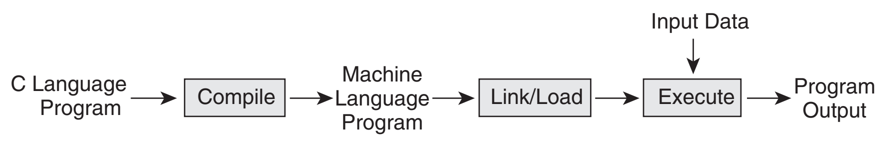


Figure 1.8 *Program compilation, linking, loading, and execution*

in order to write and execute C programs on a computer, the computer's software must include a C compiler. If any errors[1] are detected by the compiler during compilation, the compiler generates corresponding error messages. Programmers must correct the program statements and then perform the compilation step again. The errors identified during this stage are called **compile-time errors**. For example, if you want to divide the value stored in a variable called sum by 3, the correct expression in C is sum/3. If you incorrectly write the expression using the backslash, as in sum\3, you will get a compiler error. For non-trivial programs, the process of correcting statements (or **debugging**) and recompiling often must be repeated several times before the program compiles without compiler errors. When there are no compiler errors, the compiler generates a program in machine language that performs the steps specified by the original C program. The original C program is referred to as the **source code**, and the machine-language version is called the **object code**. Thus, the source code and the object code specify the same logic, but the source code is specified in a high-level language and the object code is specified in machine language.

Linking: Once the program has compiled correctly, additional steps are necessary to prepare the object code for **execution**. A **linker** will search libraries of built-in capabilities required by this program and collect them in a single executable file stored on the hard drive. Errors generated in this phase are typically caused by the programmer referring to program modules that are not, in fact, defined in the current context.

Loading: A **loader** is then used to copy the executable program into memory where its instructions can be executed by the computer.

Execution: New errors, synonymously called **execution errors**, **runtime errors**, **logic errors**, or **program bugs**, may be identified in this stage. Execution errors often cause the termination of a program. For example, the program statements may attempt to perform a division by zero, which usually generates an execution error. Some execution errors, however, do not stop the program from executing, but they cause incorrect results to be computed. These types of errors can be caused by programmer errors in determining the correct steps in the solutions and by errors in the data processed by the program. When execution errors occur because of errors in the program statements, you must correct the errors in the source program and then begin again with the compilation step. Even when a program appears to execute

[1]often called **bugs**, a reference to an unidentified insect that caused a short in one of the early digital computers

properly, you must check the results carefully to be sure that they are correct. The computer will perform the steps precisely as you specify them. If you specify the wrong steps, the computer will execute these wrong (but syntactically legal) steps and present you with an answer that is incorrect.

1.4 Running an Interpreted Program

An interpreted language is one that does not appear to require compilation. Rather, the environment in which it is used gives the user the impression that the instructions are taken one at a time and executed directly. The advantage of interpreted code is that the programmer can run programs a line at a time or from a stored text file, see the results immediately, and apply a number of tools to find out why the results were not as expected. Programmers can rapidly develop and execute programs (scripts) that contain commands and executable instructions that allow them to gather data, perform calculations, observe the results, and then execute other scripts. This interactive environment does not require the formal compilation, linking/loading, and execution process described earlier for high-level computer languages.

The disadvantages of interpreted code are numerous. The code is very slow to run relative to compiled code because every line must be syntactically analyzed at run-time. In order to reduce the impact of this as much as possible, the interpreter will often make use of a compilation step that is hidden from users. Also, because there is no explicit compilation step, the programmer does not have the compiler's protection from syntax errors. Typographical errors that cause unknown assets to be referenced from a program cannot be caught by the linker. In fact, all programming errors—syntactic, typographical, and logical—are postponed until the moment the interpreter tries to deal with the offending line of code. They all become run-time errors.

1.5 Anticipated Outcomes

To conclude this chapter, we list in increasing order of importance three outcomes for a diligent student: a brief introduction to MATLAB, some understanding of programming concepts, and improvement in their problem-solving skills.

1.5.1 Introduction to MATLAB

MATLAB is a highly successful engineering programming language that includes not only the capabilities needed in this text to introduce programming to novices, but also a vast collection of tools in toolboxes that enable professional engineers to be highly productive. It is very likely that you will encounter MATLAB in your career as an engineer. The concepts you learn in this book will ensure that you know what to do when faced with a MATLAB program.

1.5.2 Learning Programming Concepts

Even if you never see MATLAB again, you will certainly either need to use other programming languages or be able to converse effectively with other engineers who do. Converting to, or writing accurate specifications for, other languages is greatly simplified if you have a general idea of the capabilities of that language. When faced with a different programming language, if the student has an understanding of the basic underlying programming concepts, the transition from MATLAB to the new language becomes just one question—"How do I express the concepts I need in the new language?" We therefore have chosen to explain each programming concept in a language-independent way before discussing the MATLAB implementation of that concept.

1.5.3 Problem-Solving Skills

More important even than the computing concepts inherent in all computer languages is the ability to use those concepts as tools to solve a problem. Before we even start to program, we have to develop an idea of how to solve the problem before us. If we think about a computer program as a logical component that consumes data in one form and produces data in another, we can think about problem solving as the process of designing a collection of solutions to sub-problems. A brief illustration and example will suffice.

In general terms, solutions to nontrivial problems are found by the two-pronged approach illustrated in Figure 1.9. We can consider the original information and ask ourselves what could be done with that information using existing tools, and we can also consider the objective and the different ways in which that objective might be achieved. The process of creative problem solving then becomes a search for a match between states that can be achieved from the given data and states from which the answer can be achieved.

For example, say you have a big collection of baseball cards and you want to find the names of the 10 "qualified" players with the highest lifetime

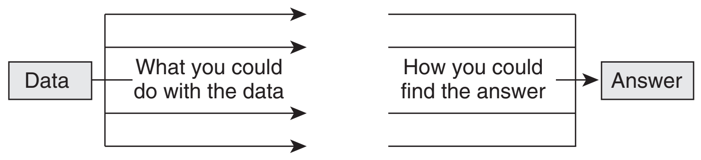

Data
What you could do with the data
How you could find the answer
Answer

Figure 1.9 *Generalized problem solving*

72 DPI M01_SMIT8708_03_SE_C01.indd Page 15 30/01/12 4:34 PM user-f396 F-396 EQA

batting averages. To qualify, the players must have been in the league at least five years, had at least 100 plate appearances per year, and made fewer than 10 errors per year. The cards contain all the relevant information for each player. You just have to organize the cards to solve the problem. Clearly there are a number of steps between the stack of cards and the solution. In no particular order, these are:

a. Write down the names of the players from some cards
b. Sort the stack of cards by the lifetime batting average
c. Select all players from the stack with five years or more in the league
d. Select all players from the stack with fewer than 10 errors per year
e. Select all players from the stack with over 100 plate appearances per year
f. Keep the first 10 players from the stack

When you think about it from right to left as shown in Figure 1.9, step a is probably the last step and step f is probably the step before that. The hard work starts when you think about it from left to right. Intuitively, when you think about sorting the stack of cards, this seems like a lengthy process. Since the sorting should probably be done on a small number of cards, you should do all the selecting before the sorting. Continuing that line of reasoning, you would reduce the total effort if the first selection pass was the criterion that eliminated most cards. You might even consider combining all three selection steps into one.

One logical way to find the players' names that you need would be to perform the steps in this order: c, d, and e in any order, followed by b, f, and then a.

Chapter Summary

This chapter presented an overview of the historical background of computing and the computer hardware and software concepts that build the foundation for the rest of this book:

- The spectrum of software products, ranging from operating systems to the many flavors of specific programming tools
- The rich variety of programming languages currently in use, and the place of interpreted programs in that spectrum as a legitimate fourth-generation language
- The basics of problem solving as a search for a path from the data provided to the answers required

Self Test

Use the following questions to check your understanding of the material in this chapter:

True or False

1. Computers were originally conceived as tools for solving specific problems.
2. Bill Gates designed the first working computer.
3. Programs cannot interact with the world outside the computer without an operating system.
4. Programs cannot interact with the world outside the computer without drivers.
5. Programs cannot interact with the world outside the computer without hardware interfaces.
6. Application programs have access to shared memory.
7. An algorithm bridges the gap between the available data and the result to be achieved.

Fill in the Blanks

1. A computer language is not a(n) _______________ exercise; it is a _____________tool for communication and problem solving.
2. Together with the use of binary encoding for storing numerical values, _______________was the genesis of general-purpose computing as we know it today.
3. Most operating systems today use__________________, which is actually a data file containing an image of everything you would like to have in RAM.
4. Operating systems contain a group of programs called __________________ that allow you to perform functions, such as printing, copying files, and listing the file names.
5. Many ________________ are loaded automatically when the operating system starts, and others are loaded upon user request.
6. Even when a program appears to execute properly, you must check the results carefully to find ____________ errors.
7. Problem solving is the process of designing a collection of ___________ .
8. The process of problem solving is a search for a match between _______________ one can achieve from the given data and __________________ from which the answer can be achieved.