

Computer Science and Programming 1

The computer is no better than its program.

Elting Elmore Morison
Men, Machines and Modern Times

Science and technology, and the various forms of art, all unite humanity in a single and interconnected system.

Zhores Medvedev *The Medvedev Papers*

I want to reach that state of condensation of sensations which constitutes a picture.

Henri Matisse
Notes d'un Peintre

In this chapter we introduce you to computer science. Ideally, we would begin with a simple definition that could be expanded and refined throughout the book. Unfortunately, computer science, like other disciplines, has no simple definition. For example, we might say that biology is the study of life. But that doesn't explain much about the content of such subdisciplines as animal behavior, immunology, or genetics—all of which are part of biology. Nor does it explain much about the contributions that these disciplines make to biology in general. Similarly, is English the study of grammar and spelling, the reading of Shakespeare's plays, or the writing of poems and stories? In many cases it is easier to consider the subfields within an area of study than it is to define the area of study. So it is with computer science.

1.1 What Is Computer Science?

In some respects, computer science is a new discipline; it has grown and evolved along with the growth of computing technology and the cheaper, faster, and more accessible processing power of modern-day computers. As recently as 1970, many colleges and universities did not even have departments of computer science. But computer science has benefited from work done in such older disciplines as mathematics, psychology, electrical engineering, physics, and linguistics. Computer science inherits characteristics from all these fields in ways that we'll touch on in this book, but the thread that links these and the many subdisciplines of computer science is computer programming.

Some people prefer the term used in many European languages, *informatics*, over what is called *computer science* in the United States. Computer science is more the study of managing and processing information than it is the study of computers. Computer science is more than programming, but programming is at the core of information processing and computer science.

This book will guide you through the study of the design, building, and analysis of computer programs. Although you won't become an expert by working through this book, you will lay a foundation on which expertise can be built. Wherever possible, the programming examples will solve problems that are difficult to solve without a computer: a program might find the smallest of 10,000 numbers, rather than the smallest of 2 numbers. Longer examples are taken from various core areas of computer science. As this is a book about the design and analysis of computer programs, it must be used in conjunction with a computer. Reading alone cannot convey the same understanding that using, reading, and writing programs can.

1.1.1 The Tapestry of Computer Science

This chapter introduces computer science using a tapestry metaphor. A tapestry has much in common with computer science. A tapestry has many intricate scenes that form a whole. Similarly, computer science is a broad discipline with many intricate subdisciplines. In studying a tapestry, we can step back and view the work as a whole, move closer to concentrate on some particularly alluring or colorful region, and even study the quality of the fabric itself. We'll similarly explore computer science—studying some things in detail, but stepping back to view the whole when appropriate. We'll view programs as tapestries too. You'll study programs written by others, add to these programs to make them more useful, and write your own programs. You'll see that creating and developing programs is not only useful but is immensely satisfying, and often entertaining as well.

Several unifying threads run through a tapestry, and the various scenes and sections originate from and build on these threads. Likewise in computer science, we find basic themes and concepts on which the field is built and that we use to write programs and solve problems. In this chapter we introduce the themes of computer science, which are like the scenes in a tapestry, and the concepts, which are like the unifying threads.

Contexture is a word meaning both “an arrangement of interconnected parts” and “the act of weaving (assembling) parts into a whole.” It can apply to tapestries and to computer programming. This book uses a contextural approach in which programming is the vehicle for learning about computer science. Although it is possible to study computer science without programming, it would be like studying food and cooking without eating, which would be neither as enjoyable nor as satisfying.

Computer science is *not* just programming. Too often this is the impression left after an initial exposure to the field. I want you to learn something of what a well-read and well-rounded computer scientist knows. You should have an understanding of what has been done, what might be done, and what cannot be done by programming a computer. After a brief preview of what is ahead, we'll get to it.

Alan Turing (1913–1954)

Alan Turing was one of the founders of computer science, studying it before there were computers! To honor his work, the highest achievement in the field of computer science—and the equivalent in stature to a Nobel prize—is the Turing award, given by the Association for Computing Machinery (the ACM).



In 1937, Turing published the paper *On Computable Numbers, with an Application to the Entscheidungsproblem*. In this paper he invented an abstract machine, now known as a *Turing Machine*, that is (theoretically) capable of doing any calculation that today's supercomputers can. He used this abstract machine to show that there are certain problems in mathematics whose proofs cannot be found. This also shows that there are certain problems that cannot be solved with any computer. In particular, a program cannot be written that will determine whether

an arbitrary program will eventually stop. This is called the **halting problem**.

During World War II, Turing was instrumental in breaking a German coding machine called the *Enigma*. He was also very involved with the design of the first computers in England and the United States. During this time, Turing practiced one of his loves—long-distance running. A newspaper account said of his second-place finish (by 1 foot) in a 3-mile race in a time of 15:51: “Antithesis of the popular notion of a scientist is tall, modest, 34-year-old bachelor Alan M. Turing... Turing is the club's star distance runner... [and] is also credited with the original idea for the Automatic Computing Engine, popularly known as the Electronic Brain.”

Turing was also fond of playing “running-chess,” in which each player alternated moves with a run around Turing's garden. Turing was gay and, unfortunately, the 1940s and 50s were not a welcome time for homosexuals. He was found guilty of committing “acts of gross indecency” in 1952 and sentenced to a regimen of hormones as a “cure.” More than a year after finishing this “therapy,” and with no notice, Turing committed suicide in 1954.

For a full account of Turing's life see[Hod83].

1.2 Algorithms

To develop an initial understanding of the themes and concepts that make up the computer science tapestry, we'll work through an example. Consider two similar tasks of arranging objects into some predetermined order:

Arranging Cards

- Arrange cards into four groups by suit: spades, hearts, clubs, diamonds
- Sort each group. To sort a group:
 - For each rank (2, 3, 4, ..., 10, J, Q, K, A) put the 2 first, followed by the 3, the 4, ..., followed by the 10, J, Q, K, A (if any rank is missing, skip it)

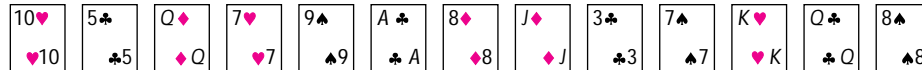
Figure 1.1 Arranging cards in order.

1. A hand of cards (arrange by rank and suit)
2. 100,000 exams (arrange by six-digit student ID number)

Card players often do the first task because it makes playing much simpler than if the cards in their hands are arranged in a random order. The second task is part of the administration of the Advanced Placement exams given each year to high school students. Many people are hired to sort the exam booklets by student ID number before the scores are entered into a computer. In both cases people are doing the arranging. The differences in the scale of the tasks and the techniques used to solve them will illuminate the study of computer science and problem solving.

1.2.1 Arranging 13 Cards

A hand of cards might look like this:



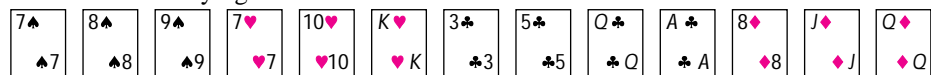
Most people arrange cards in order by suits (spades, hearts, diamonds, and clubs), and within suit by rank (2, ..., 10, J, Q, K, A) with little thought. In fact, many people perform a slightly different sequence of steps in arranging different hands of cards, modifying their basic technique depending on the order in which the cards are dealt. However, if you are asked to describe the process of arranging a hand of cards to someone who has never seen cards before, the task becomes difficult. The careful description of such processes is one of the fundamental parts of computer science. The descriptions are called **algorithms** and are the focus of much study in computer science and in this book.

The algorithm for sorting cards shown in Figure 1.1 is both correct and concise, two traits to strive for in writing algorithms. The instructions to sort a group are applicable to all groups, not just to the spades or to the diamonds. Instructions that apply in more than one situation are much more versatile than instructions that apply in a single situation.

Algorithms are often compared to recipes used in cooking: they are step-by-step plans used in some process (arranging cards or baking bread) to arrive at some end (a

sorted hand of cards or a loaf of bread). Although this analogy is apt, cooking often allows for a larger margin of error than do algorithms that are to be implemented on a computer. Phrases like “beat until smooth,” “sauté until tender,” and “season to taste” are interpreted differently by cooks. A more appropriate analogy may be seen with the instructions that are used to knit a sweater or make a shirt. In such tasks, precise instructions are given and patterns must be followed or else a sweater with a front larger than the back or a shirt with mismatched buttons and buttonholes may result.

You can easily determine that the hand below is sorted correctly, in part because there are so few cards in a hand and because grouping cards by suit makes it easier to see if the cards are sorted. Verifying that the algorithm is correct in general is much more difficult than verifying that one hand of cards is sorted.



For example, suppose that an algorithm correctly sorts 1,000 hands of cards. Does this guarantee that the algorithm will sort all hands? No, it’s possible that the next hand might not be sorted even though the first 1,000 hands were. This points out an important difference between verifying an algorithm and testing an algorithm. A *verified* algorithm has been proved to work in all situations. A *tested* algorithm has been rigorously tried with many examples to establish confidence that it works in all situations.

1.2.2 Arranging 100,000 exams

Arranging 100,000 exams by ID number is a much more cumbersome task than arranging 13 cards. Imagine being confronted with 100,000 exams to sort. Where would you begin? This task is more time-consuming and more prone to error than arranging cards. Although you probably don’t need a precise description of the card-arranging algorithm to sort cards correctly, you’ll need to think carefully about developing an algorithm to sort 100,000 exams using 40 people as assistants. Utilizing these “computational assistants” requires communication and organization beyond what is needed to arrange 13 cards in one person’s hand. A sample of 32 student ID numbers is shown here:

```
672029 662497 118183 452603 637238 249262 617834 396939
483595 613046 361999 231519 695368 689831 346006 539184
712077 816735 540778 975985 950610 846581 931662 625487
278827 821759 131232 952606 547825 385646 880295 816645
```

These represent a small fraction of the number of exam booklets that must be arranged. Consider the algorithmic description in Figure 1.2. If this algorithm is implemented correctly, it will result in 32 numbers arranged from smallest to largest. If we had a computer to assist with the task, this might be an acceptable algorithm. (We’ll see later that there are more efficient methods for use on a computer but that this is a method that works and is simple to understand.) We might be tempted to use it with 32 exams, but with 100,000 exams it would be extremely time-consuming and would make inefficient use of the resources at our disposal since using 40 people to find the smallest exam number is a literal waste of time.

Sorting Exams

Repeat the following until all 32 numbers (exams) have been arranged

- scan the list of numbers (exams) looking for the smallest exam
- move the smallest number (exam) to another pile of exams that are maintained and arranged from smallest to largest

Figure 1.2 Arranging exams in order.



1.3 Computer Science Themes and Concepts

The previous sorting example provides a context for the broad set of themes and concepts that comprise computer science.

1.3.1 Theory, Language, Architecture

Three areas mentioned in [Ble90] as forming the core of computer science serve nicely as the essential themes, linking the various scenes of the computer science tapestry together. These themes are shown in Figure 1.3. Although we can develop algorithms for both sorting tasks, it would be useful to know if there are better algorithms or if there is a “best” algorithm in the sense that it can be proven to be the most efficient. Determining whether an algorithm is “better” than another may not be relevant for arranging cards because a nonoptimal algorithm will probably still work quickly. However, a “good” algorithm is very relevant when arranging 100,000 exams. Developing algorithms and evaluating them is the part of computer science known as **theory**.

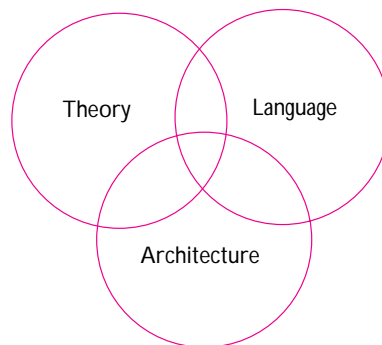


Figure 1.3 Essential computer science themes.

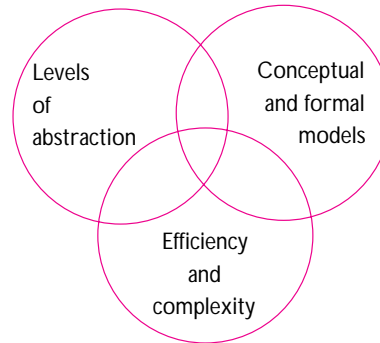


Figure 1.4 Recurring concepts

If the algorithms are to be implemented on a computer or used by people (who are in some sense “computational engines”), there must be a **language** in which the algorithms are expressed. We have noted that cooking recipes, while similar to algorithms, often leave room for ambiguity. Although English (or other natural languages) may at some point become a viable language in which to “instruct” computers, specialized computer languages are needed now. Many programming languages exist, and often the choice of language has a large impact on how well a program is written and on how fast it is developed. Languages are necessary to implement algorithms on specific kinds of computers.

Although both these arranging tasks are similar, an algorithm for one may be inappropriate for the other. Viewing a person as a computational resource (or processor), we see that the card-arranging task is done using one processor while the exam-arranging task is done using several processors. Just as some people can sort cards more quickly than others, some computer processors are faster and work differently than other processors. The term **architecture** is used to describe how a computer is put together just as it is used to describe how a building is put together. One active research area in computer science involves developing algorithms and architectures for multiprocessor computing systems.

1.3.2 Abstractions, Models, and Complexity

In this section we continue our contextual approach, whereby we weave the essential themes into the fabric that is computer science and the scenes that make up its tapestry. In addition to the themes of theory, language, and architecture, we’ll often refer to several of the recurring concepts presented in [(ed91)]. These form part of the foundation on which computer science is built; they are shown in Figure 1.4.

Both sorting tasks involve arranging things, yet the complexity of the second task makes it imperative that an efficient algorithm be used if the goal is to be achieved within a reasonable time frame. Both **efficiency** and **complexity** are parts of the computer science

tapestry we are studying. In programming and computer science, these terms concern how difficult a problem is and the computational resources, such as time and memory, that a problem requires.

We have avoided many of the details inherent in these examples that might be of concern as rough ideas evolve into detailed algorithms. If 40 people are sorting exams we might be concerned, for example, with how many are left-handed. This might affect the arrangement of the exams as they are physically moved about during the sorting process. Some playing cards are embellished with beautiful designs; it might be necessary to explain to someone who has never played cards that these designs are irrelevant in the arrangement process. In general these levels of detail are examples of **levels of abstraction** (Figure 1.4). In one sense this entire chapter mirrors the fact that we are viewing the computer science tapestry at a very high level of abstraction, with few details. Each subsequent chapter of this book involves a study of some aspect of the tapestry at a level of greater detail.

Finally, both these tasks involve numbers. We all have an idea of what a number is, although the concept of number may be different to a mathematician and to an accountant. In computer science conceptual ideas must often be formalized to be well understood. For example, telling someone who is playing hide-and-go-seek to start counting from 1 and to stop when they reach the “last number” is an interesting way to teach the concept of infinity. The finite memory of computers, however, imposes a limit on the largest number that can be represented. This difference between **conceptual and formal models** is a concept that will recur and that completes the three concepts in Figure 1.4, forming common threads of the computer science tapestry.

Pause to Reflect



1.1 The *New Hacker's Dictionary* defines *bogo-sort* as described here.

Bogo-sort: Repeatedly throw a hand of cards in the air, picking them up at random, and stopping the process when examining the hand reveals the cards are in order.

Using this “algorithm,” what is the minimum number of “throws” that yields a sorted hand? What is the danger of using this algorithm?

1.2 In the algorithm for sorting cards, nothing is stated about forming a hand from each of the separate suits. Does something need to be stated? Is too much left as “understood by the player” so that someone unfamiliar with cards couldn’t use the algorithm?

1.3 Write a concise description of the method or algorithm you use to sort a hand of cards.

1.4 Suppose that the 32 student ID numbers listed in the text are sorted. Is it a simple matter to verify that the numbers are in the correct order? Consider the same question for 100,000 numbers.

Charles Antony (Tony)Richard Hoare (b. 1934)

Perhaps best known for his invention of the sorting algorithm he modestly named Quicksort, Hoare has made profound contributions to many branches of computer



science, especially in programming and programming languages. Hoare received the ACM Turing award in 1980. In his award address he had this to say about learning from failure: “I have learned more from my failures than can ever be revealed in the cold print of a scientific article and now I would like you to learn from them, too.

Besides, failures are much more fun to hear about afterwards; they are not so funny at the time.” In a collection of essays [Hoa89], Hoare describes the programmer of the current era as part apprentice and part wizard; he urges that computer science education should focus on both theoretical foundations and practical applications. In his last essay of that collection he states “I salute the bravery of those who accept the challenge of be-

ing the first to try out new ideas; and I also respect the caution of those who prefer to stick with ideas which they know and understand and trust.”

I think Hoare may not like C++; it is too big, too full of features, and it doesn't have a formal foundation. However, according to his web page, he set himself the following task for his 1993–1994 sabbatical year: to become acquainted with Visual Basic™. Of course as other goals for that year he listed:

To complete a work on unification of theories of programming and to start new work on a range of scientific theories of computational phenomena.

In describing computer science as, in part, an engineering discipline, Hoare states:

...the major factor in the wider propagation of professional methods is education, an education which conveys a broad and deep understanding of theoretical principles as well as their practical application, an education such as can be offered by our universities and polytechnics.

For more information, see [Hoa89].

1.4 Language, Architecture, and Programs

Language is necessary for expressing algorithms. For computers, a precise programming language is necessary. In this section we briefly touch on the process by which an algorithm is transformed from an idea into a working computer program. This process is the same regardless of the kind of computer being used.

The final computer program differs from machine to machine in the same manner that the same idea is expressed differently in German than it is in English. Consider the German word *Geländesprung* defined:

Geländesprung: a jump made in skiing from a crouching position with the use of both poles.

An idea whose expression requires many English words can be expressed in a single German word. Different computers can offer the same economy of expression; what one computer might do in a single instruction can require several instructions (and a corresponding increase in time to execute the instructions) on another computer. For example, so-called **supercomputers** can add 100 numbers with a single instruction. On ordinary computers, one instruction can add only two numbers.

1.4.1 High- and Low-level Languages

High thoughts must have high language.
Aristophanes
Frogs

How do computers work? We don't need to know this to use computers just as we don't need to know how internal combustion engines work to drive a car. A little knowledge, however, can help to demystify what a computer is doing when it executes a program. A computer can be viewed from many levels, from the transistors that make up its circuits to the programs that are used to design the circuits.

At the lowest level, computers respond to electric signals at an extremely fast rate. Computers react to whether electricity is flowing or not; the computer merely responds to switches that are in one of two states: on or off. This method of using two states involves what is termed the **binary number system**, or the **base 2 system**. This system is based on counting using only the digits 0 and 1. The base 10 system, with which you are most familiar, uses the digits 0 through 9.

There are hundreds of different kinds of computers. You may have used Apple Macintosh computers, which are built using a computer chip called the *Power-PC*, or another kind of computer based on the Intel *Pentium* chip. Pictures of these different chips are shown at the end of the chapter in Figure 1.9 and 1.11. These chips are the foundation on which a computer is built. The chip determines how fast the computer runs and what kinds of software can be used with the computer. Since computers are constructed from different components and have different underlying architectures, they may respond differently to the same sequence of zeros and ones. Just as *chat* means “to converse informally” in English and means “a small domesticated feline (cat)” in

French, so might `00010100111010` instruct one computer to add two numbers and another computer to print the letter *q*.

Rather than instruct computers at this level of zeros and ones, languages have been developed that allow ideas to be expressed at a higher level—in a way more easily understood by people. In addition to being more easily understood, these high-level languages can be translated into particular sequences of zeros and ones for particular computers. Just as translators can translate English into both Japanese and Swahili, so can translating computer programs translate a high-level language into a low-level language for a particular computer. The concept of higher level programming languages was a breakthrough. The first computers were “programmed” literally by flipping switches by hand or physically rewiring the computer to create different on/off states corresponding to a program. The use of higher level languages made programming easier (although it is still an intellectually challenging task) and helped to make computer use more prevalent.

The computer language used in this book is called C++.¹ This language has its roots in the C programming language, which was developed in the 1970s. The language C is a high-level language² that allows low-level concepts to be expressed more readily than some other high-level languages. For example, in C it is easy to write a program to change a single bit (a 0 or a 1) in the computer’s memory. This is hard, if not impossible, to do in other high-level languages, such as Pascal.

We’re not studying C++ because it permits one bit to be changed. We’re studying C++ because with it several programming styles are possible. In particular, it can be used with a style of programming called *object-oriented programming*, often abbreviated as OOP. We will use OOP throughout this book, but it will be an aid to our study of programming and computer science rather than the principal focus. We’ll explore OOP briefly at the end of this chapter.

The intricacies of C++ are such that mastering the entire language, as well as the concepts of object-oriented programming, is a task too daunting and difficult for beginning programmers. In this book we present a significant subset of C++ and use it to write programs that permit the study of essential areas of computer science. At the same time the power of C++ is exploited where possible to allow you to create more complicated programs than would be feasible using other languages. Don’t be disheartened that you won’t learn absolutely all of C++ in this book—you’ll be building a foundation on which subsequent study can add. The few parts of the language that aren’t covered are mostly “short-cuts” that can be replaced using features of the language that are in the book.

A Concrete Example. To illustrate the difference between high- and low-level languages, we’ll study how a C++ program is translated into a low-level language. The low-level language of 0’s and 1’s that a computer understands is called **machine language**. Because different computers have different machine languages, a program is needed to translate the high-level C++ language into machine language. A **compiler** is a program that does this translation. Often the compiling process involves an intermediate step wherein the code is translated into **assembly language**.

¹This is pronounced as “see plus plus.”

²Although some computer scientists might take exception to this statement, C is clearly a much higher-level language than machine or assembly language.

```

main:                                     main:
    save %sp,-128,%sp                    pushl %ebp
    mov 7,%o0                             movl %esp,%ebp
    st %o0,[%fp-20]                       subl $12,%esp
    mov 12,%o0                             movl $7,-4(%ebp)
    st %o0,[%fp-24]                       movl $12,-8(%ebp)
    ld [%fp-20],%o0                       movl -4(%ebp),%eax
    ld [%fp-24],%o1                       imull -8(%ebp),%eax
    call .umul,0                          movl %eax,-12(%ebp)
    nop                                    xorl %eax,%eax
    st %o0,[%fp-28]                       jmp .L1
    mov 0,%i0                             .align 4
    b .LL1                                 xorl %eax,%eax
    nop                                    jmp .L1
    mov 0,%i0                             .align 4
    b .LL1                                 .L1:
    nop                                    leave
.LL1:                                     ret
    ret
    restore

```

Figure 1.5 Assembly code using g++ (Sparc on left, Pentium on right).

To keep the example simple, we'll use a program that stores two numbers in memory, then multiplies the numbers storing the product in a different memory location. The program follows.

```

int main()
{
    int x,y,z;
    x = 7; y = 12;
    z = x*y;
    return 0;
}

```

We will not discuss the C++ instructions here; we use the program only to illustrate the differences between high- and low-level languages.

The world is full of C++ compilers. Compilers exist for various kinds of computers, sizes of programs, and amounts of money. The code in this book has been tested using four different compilers. Some of these compilers cost hundreds of dollars, some are less expensive, and one is free.

The assembly code generated by the same compiler running on two different machines is shown in Figure 1.5. The compiler used is g++ running on two different machines: a Sun Sparcstation and a Pentium-based computer.³

There is one column of assembly code for each machine. Note that although the programs are of roughly the same length, there are few similarities in the assembly instructions.

³The characteristics of these machines are not important, but the same compiler runs on both machines, which facilitates a comparison.

Among the instructions are *ld*, *call*, and *nop* for the Sun assembly and *pushl*, *subl*, and *xorl* for the Pentium. The important point of Figure 1.5 is that you do *not* need to worry about assembly code to write programs in C++ or in any other high-level language. It is comforting to know that we can ignore most of the low-level details in writing programs and studying computer science and, perhaps, enticing to know that the details are there for those who are interested.

1.5 Creating and Developing Programs

How is a computer program created? Usually a problem arises whose solution requires computation. An algorithm for the solution is developed into a running program in several steps. The steps that lead to the program's execution on a computer are also important. We'll look at developing a program for the problem of multiplying two numbers as shown in Figure 1.6. The process of developing an idea into an algorithm that is eventually realized as a working computer program is illustrated in Figure 1.7.

From Problem to Algorithm. Consider the steps labeled 1 and 2 in Figure 1.7. The problem of multiplying two specific numbers (1285 and 57) has been generalized to the problem of multiplying two arbitrary numbers (Y and Z). The two views of the problem, one concrete and one general, represent two levels of abstraction. A solution to the general problem will be useful for any two numbers, not just for 1285 and 57. If you can develop a general solution that is useful in many situations, it is usually worth it. Sometimes, however, a solution to a specific problem is needed and solving a general version would take too long or be too difficult.

To write a program for solving this general problem, we must develop an algorithm for multiplication. Consider multiplying rational numbers (fractions), integers, real numbers, and complex numbers as illustrated in Figure 1.6.

You may not be familiar with each of these types of numbers, but each uses a different method for multiplication. If we're going to write a program to multiply, we'll need to determine what **type** of number is being used. The general form of $X \times Y$ can be used to express multiplication regardless of which type of number is multiplied. One of the advantages of C++ is that this conceptual similarity in notation is formalized in code: the same symbol, `*`, can be used to multiply many types of numbers.

In addition to the type of number, considerations in the development of the algorithm might include the size of the numbers being multiplied (an efficient algorithm would

Rational	Integer	Real	Complex
$3/4 * 8/9$	$1,285 * 57$	$3.14 * 6.023$	$(3 + 5i) * (2 - 7i)$
$2/3$	73,245	18.91222	$41 - 11i$

Figure 1.6 Multiplying different types of numbers.

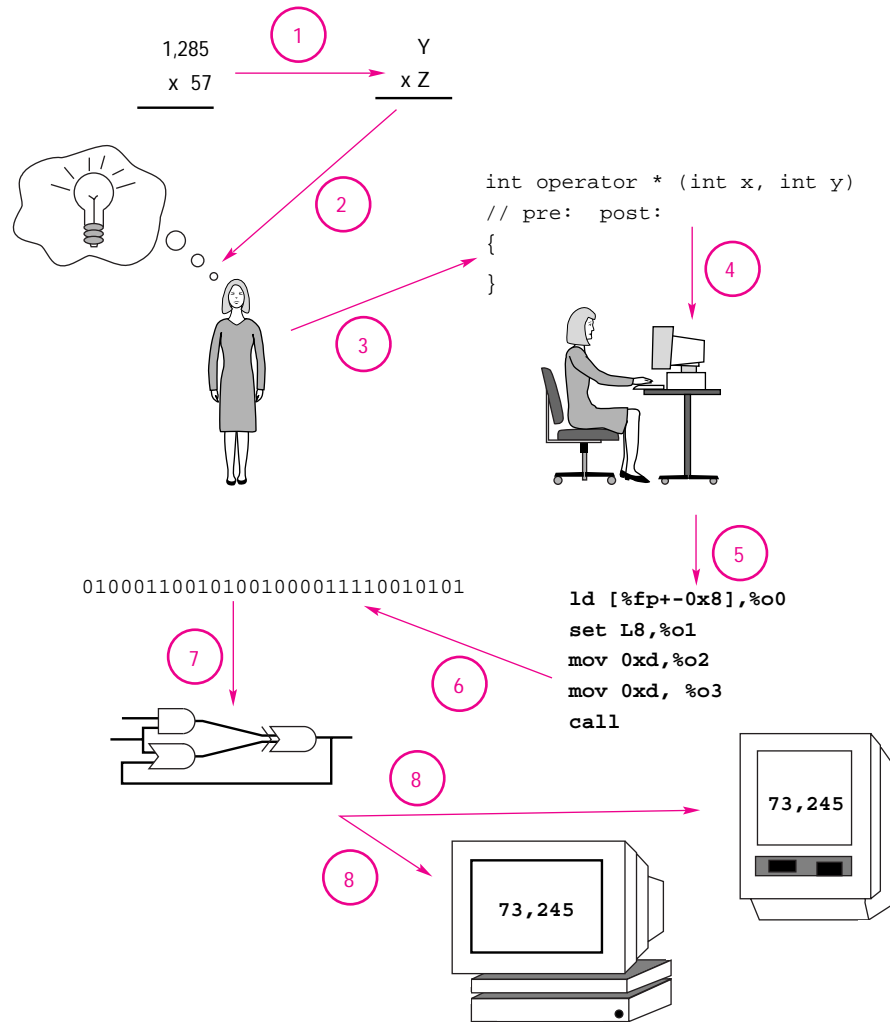


Figure 1.7 The steps of transition from problem to program.

be more important if the numbers were hundreds of digits long as opposed to three digits long), how many times numbers will be multiplied, and whether the result of multiplying the numbers can exceed the memory constraints of the computer. Although it's impossible for numbers to get "too big" conceptually, the inherent finiteness of a computer's memory requires that a formal model of computation take this into account.

From Algorithm to Program. In step 3 we translate the algorithm into the high-level language C++. The name *operator ** has been given to the C++ instructions that perform

the multiplication. Translating the algorithm into code requires a knowledge of the programming language's syntax—the symbols and characters used in the language—as well as the meaning, or semantics, of these characters.

Once the algorithm is represented in a high-level language, a program must be entered into a computer. Step 4 consists of more than merely typing characters at a keyboard. Often the realization of the algorithm as a computer program has errors that become apparent as the program is tested. Testing can indicate that errors exist; removing the errors is another problem. Errors are often euphemistically called *bugs*.⁴ This makes the process of removing errors **debugging**. Testing and debugging can uncover errors in the original algorithm in addition to errors in the C++ representation of the algorithm.

As you become more experienced at programming you can employ techniques called **defensive programming**: attempting to ensure that your programs are robust and error-free as part of the design process rather than relying on testing and debugging exclusively. Many computer scientists are currently developing methods that will permit programs to be proved correct in the same manner that mathematical theorems are proved. Although we will not use such formal methods in our study, we introduce some of the techniques.

From High-level Program to Low-level Program. In step 5, the high-level C++ program is translated into a lower-level language called **assembly language**. The name is derived from the notion of assembling the individual low-level instructions available on a particular computer into a form understandable by people. Although some programming is still done directly in assembly, the process of translation from high-level to low-level has been refined enough that programming at this level is often unnecessary.

Step 6 shows the translation of assembly language to **machine language**, the language of zeros and ones that a particular computer understands. Specific assembly language and machine language instructions differ according to the kind of computer being used (as shown in Figure 1.5), as opposed to high-level languages like C++, which are the same on various computers. The process of translation illustrated by steps 5 and 6 is accomplished by a computer program called a **compiler** and the process is called **compiling**. A compiler translates code written in a high-level language into machine language. This translation process often includes an intermediate step in which the code is translated into assembly language.

Executing Machine Language. At the lowest level, the zeros and ones of machine language code cause switches to be turned on and off in the computer. These switches are extremely small and can be switched on and off quite rapidly. Technological advances have enabled transistors, which function as switches, to become increasingly smaller and faster. Switches are often represented by the diagrams in step 7.

The execution of a program is separate and different from the compilation of the program. Compiling a C++ program yields a low-level program, whereas executing a

⁴The derivation of the word *bug* is open to debate. Thomas Edison was reported to have discovered a “bug” in his phonograph in 1889. A literal example is the moth trapped in one of the first computers, the Harvard Mark II. The moth was placed into the system's logbook with the annotation “First actual case of bug being found” and is now on display in the Naval Museum in Dahlgren, Virginia.

machine language program results in the computer performing the tasks represented by the compiled machine code.

Coming Full Circle: Displaying the Results. Most current computers, and certainly the computers you will be using as you study computer science with this book, have a screen to display what happens when a program is run. Whether the program is a word processor or a C++ program for multiplying numbers, output is generally displayed on the screen. Note that the screens on the computers in Figure 1.7 display the answer to the original problem: $1285 \times 57 = 73,245$.



1.6 Language and Program Design

One of the “eternal truths” of computer science and the computer industry is

Software is harder than hardware.

This statement means that new computers (hardware) are developed at a faster pace and more easily than new programs (software). There is certainly some truth to this, although new programming languages and new design methods have been developed in an attempt to alleviate this disparity. Many people believe that object-oriented programming, or OOP, will be of great assistance in making software easier to develop. OOP allows pieces of code to be reused in other contexts more easily.

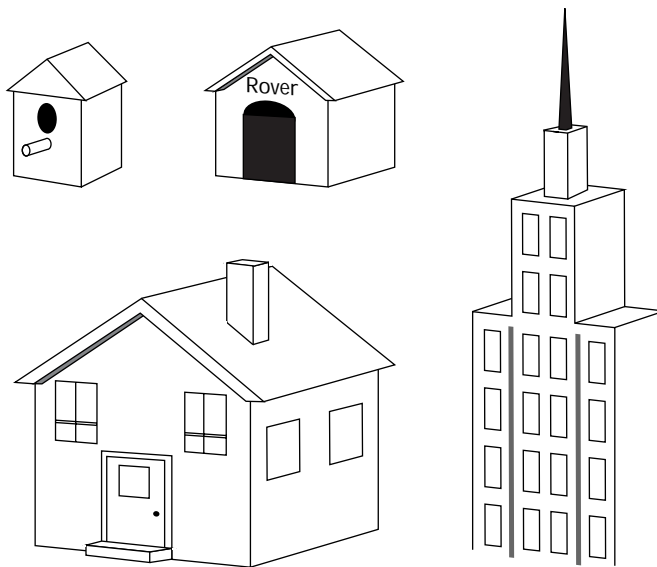


Figure 1.8 OOP: birdhouses and skyscrapers.

To try to understand what OOP is about, I use an analogy (suggested in [McC93]) that comes from construction (see Figure 1.8). Suppose you decide to build a birdhouse; you can probably nail some boards together in a couple of hours and provide a useful dwelling for your favorite flyers. You may not put much thought into the design of the house, although if you don't you may waste some wood. (A carpenter's adage is "measure twice, cut once.") Next suppose you're designing a doghouse for your favorite pet. You might take more care; you're probably more concerned with whether Rover gets wet than whether your neighborhood bluejay is inconvenienced by rain. You may buy a kit—a precut set of materials and plans for constructing the doghouse. Nevertheless, this is probably a day-long project, *if* you're used to using saws and hammers.

What about building a house? If you've been involved with house building, you know that it can take a long time, requiring contractors, plumbers, electricians, and usually a lot of headaches. However, it is certainly possible to build a house yourself. Most of the pieces of a house come prebuilt. For houses that don't use prebuilt pieces and instead require custom manufacturing, the price of construction can be very high. Finally, consider building a skyscraper such as the Empire State Building. Such a building requires careful planning and is much more complex than a typical family dwelling. Yet hundreds of such tall buildings are designed and built each year.

Computer scientists disagree about what OOP is and whether it is appropriate for use in an introductory course. By using a carefully chosen subset of C++, it is certainly possible to develop a mastery of basic programming concepts as well as an understanding and appreciation of OOP.

1.6.1 Off-the-Shelf Components

Using off-the-shelf components is one of the reasons that constructing large buildings is possible. The phrase *off-the-shelf* is used to mean a component that is manufactured in large quantity and that can be used in a variety of situations. Nails are no longer handcrafted by blacksmiths, and even houses can be purchased in a kit form. These components are often inexpensive but serve as well (or better) than custom-built components. The same is true of building computers. One of the reasons that computers get less expensive every year is that the pieces that make up a computer get cheaper as more are produced—this is sometimes called *economy of scale*. Viewed differently, the key is *not* to do all the work yourself but to use what others provide.

Of course, off-the-shelf components don't always work. A roof of a birdhouse is different from the roof of the house you live in even though both share some common characteristics. It would be very useful to be able to order a standard roof, but then to be able to customize it easily to fulfill your specific needs.

In this book you will be using others' code in the writing of your own code, and you will be reusing the code you write for yet other programs. Code reuse is increasingly important, partly because of the graphical user interfaces (window systems) that are popular on computers. These interfaces are time-consuming to program but are very similar from program to program, so the potential for code reuse is great.

One of the goals of object-oriented programming is to provide **objects** to make code development easier. Objects are like off-the-shelf software components. You can

imagine that using such objects might be much simpler than designing them yourself. Building a house from a kit is much simpler than designing the kit itself. The same is true of programming and program design—it's simpler to use software components supplied by others than to write everything yourself. In this book, however, OOP will be used in our study of programming and the examination of computer science rather than becoming the principal focus of study.

1.6.2 Using Components

As an example of how software components might be useful, consider digital clocks, the display on a CD player, a car's odometer, and a counter for web-page hits. All of these devices require the display of numerals that are manipulated in some fashion. The numerals displayed are different according to how the device is used:

- Clocks display time; the numerals represent hours, minutes, seconds.
- CD players display information on how many tracks are available on a CD (some also display time).
- Web-page counters display information about how many times the page has been accessed.
- Odometers display mileage as recorded by a car's wheels.

It should be possible for the computer programs controlling these displays to share (reuse) the code that displays numerals, differing only in how it is determined which numerals should be displayed and, perhaps, where the numerals are displayed.

Object-oriented programming involves reusable components. In C++ the word **class** refers to a family of components sharing common characteristics. A class allows **operations** that are used to manipulate the **objects** that are components of the class. For example, the class *four-door sedan* describes many makes and models of car. A specific four-door sedan, the one in my driveway, is an object of the generalized "four-door sedan class." All objects in this class share the common characteristic of having four doors and being sedans. They share other characteristics too, such as having a steering wheel, an engine, and four wheels. These characteristics are shared by all cars, not just four-door sedans. Operations allowed by the class *four-door sedan* include being driven, storing luggage, and consuming fuel.

As another example, the display of a numeral might be a different class than the value being displayed. A numeral display class might support operations such as assigning a value to be displayed and actually "drawing" the numeral. Other classes, such as a clock class or a timer class, could supply the values to be displayed.

1.7 Chapter Review

This chapter provides an introduction to the field of computer science and places programming properly within the field. In subsequent chapters you'll begin the process

of augmenting and constructing programs. The important concepts introduced in this chapter are outlined here.

- Computer science—is more than the study of computers. It includes many sub-fields that are linked by the study of programming. Key parts of computer science include theory, language, and architecture.
- Algorithm—is a plan for solving a problem. It's related to a set of instructions to accomplish a task, such as knitting a sweater, but we'll use it to refer to a plan for accomplishing a task, such as sorting a hand of cards (and often a computer will be involved).
- Theory—refers to underlying mathematical principles on which computer science is built. For example, being able to compare different algorithms to determine which is most efficient relies on theoretical tools.
- Architecture—refers to how a computer is designed and put together. Computers have different architectures: some computers rely on using several processors at one time rather than just one.
- Language—refers to computer programming languages, which come in many forms and flavors. Both high- and low-level languages are used in writing programs, but we'll concentrate on the high-level language C++.
- Efficiency and complexity—refer to how difficult a problem is to solve using a computer and how various algorithms compare in solving problems (e.g., in how fast they run).
- Conceptual and formal models—refer to different ways of thinking. Programs can be thought of as instructions for a computer, but a mathematical notion of programming is possible too.
- Levels of abstraction—refer to different ways of observing. An idea can be turned into an algorithm, which is implemented as a C++ program, which is executed as a machine-language program. The same idea is viewed at many different levels and has particular characteristics depending on the level.
- Compiler—is a computer program that translates a high-level language such as C++ into a low-level language that can be executed on a computer.
- Bug—is a mistake in a program. Finding such mistakes is called debugging.
- Object-oriented programming—is a method of programming that, in a nutshell, relies on the use of off-the-shelf software components.
- Class—is a family of objects sharing common characteristics. The integers are a class of numbers; four-door sedans are a class of cars.

1.8 Exercises

- 1.1 The process of looking up a word in a dictionary is difficult to describe in a precise manner. Write an algorithm that can be used to find the *page* in a dictionary on which a given word occurs (if the word is in the dictionary). You may assume that each page of the dictionary has guide words indicating the first and last words on the page, but

you should assume that there are no thumb indices on the pages (so you cannot turn immediately to a specific letter section).

- 1.2** Suppose that you have 10 loads of laundry, one washer, and one dryer. Washing a load takes 25 minutes, drying a load takes 25 minutes, and folding the clothes in a load takes 10 minutes, for a total of 1 hour per load (assuming that the time to transfer a load is built into the timings given.) All the laundry can be done in 10 hours using the method of completing one load before starting the next one. Devise a method for doing all 10 loads in less than 10 hours by making better use of the resources. Carefully describe the method and how long it takes to do the laundry using the method.
- 1.3** Suppose that student ID numbers consist of two digits. The exams are sorted in a large room. Consider the following description of a sorting algorithm:
- Make 100 “in-boxes” labeled 00 to 99.
 - Divide the exams among the people participating in the sort.
 - Have each person put an exam in the correct box according to ID number.
 - Collect the exams from the boxes in order (00–99).

This method will work correctly. Try to modify the method to work with four-digit ID numbers and six-digit ID numbers. In making the modification, assume you have only 100 boxes. (Hint: Consider examining only two digits at a time.)

- 1.4** The steps labeled 1–7 in Figure 1.7 illustrate the design, development, realization, and implementation of a computer program to multiply two numbers. Consider the following problem:

Develop a recipe for a chocolate cake with chocolate icing that tastes delicious and makes you swoon.

Develop analogs or parallels to the steps 1–7 for developing such a recipe. Write a detailed description of the process you might go through to develop a recipe—*not* what the recipe is.

- 1.5** Assume that a young friend of yours knows how to multiply any two one-digit numbers (i.e., knows the times tables). Write an explanation (algorithm) of how to multiply an n -digit number by a one-digit number. Can you extend this algorithm into one that can be used to multiply two many-digit numbers (such as 1285 and 57, as shown in Figure 1.7)?
- 1.6** There are many different high-level programming languages. Common languages include Pascal, FORTRAN, Scheme, BASIC, and COBOL. Can you think of a reason for why there are many languages as opposed to a single language? Why is more than one language in use today?
- 1.7** (Suggested by a description in *Computer Architecture*, by Blaauw and Brooks.) Consider clocks and watches as examples of different “architectures” used for telling time. For clocks and watches that have hands and dials, write an outline of an algorithm that can be used to tell time. How is the architecture of a wristwatch (with hands) similar to that of a grandfather clock? How is it different? What features of the face of a watch are essential for telling time? In particular, are numbers needed on the face of a watch to tell time? Make a list of different watch faces and try to distill the essential features

of a watch face into a few descriptive sentences.

Consider the inner workings of watches: list at least three different methods used to “run” a watch. How are different levels of abstraction illustrated by the concept of a watch?

How is a digital watch different from a watch with hands? How is it similar?

- 1.8 C++ (and other high-level language) programs are written in a language that is a compromise between natural languages such as English and the language of zeros and ones, which is “spoken” by computers. Consider musical compositions written for different instruments or groups of musicians. Is music written in a high-level language or a low-level language? Are there different languages for expressing musical compositions as there are different natural languages and different computer languages? Why?
- 1.9 Suppose that you are playing in a large field with several friends and one of you discovers that a house key has been lost. Write an algorithm for finding the key that is designed to find it as quickly as possible. Write another algorithm designed to take a long time to find the key. Can you reason about whether your algorithms are the best possible or worst possible algorithms for this particular task?
How is this task related to how you look for a key when you have misplaced it inside your house?



Figure 1.9 A Pentium chip.

1.10 The program used to generate the assembler output in Figure 1.5 is used in Fig 1.10 on two different computers; the assembler code below on the right is generated on a Macintosh G3 computer, the code on the left on a Pentium computer running Windows NT. Both machines use the same compiler: Metrowerks Codewarrior. A Pentium chip is shown in Figure 1.9 and a G3 chip is shown in Figure 1.11.

What is similar in these two versions of assembly language and what is different? Can you find instructions that would be common to all the different assembly codes? Why do you think different compilers generate different code for the same program?

<pre> _main push ebp mov ebp,esp sub esp,16 mov dword ptr [ebp-12],7 mov dword ptr [ebp-8],12 mov edx,dword ptr [ebp-12] imul edx,dword ptr [ebp-8] mov dword ptr [ebp-4],edx mov eax,0 leave ret near </pre>	<pre> ".main"(1) stw r31,-4(SP) stw r30,-8(SP) li r31,7 li r30,12 mullw r0,r31,r30 stw r0,-16(SP) li r3,0 lwz r31,-4(SP) lwz r30,-8(SP) blr </pre>
--	--

Figure 1.10 Windows NT code on the left, Macintosh G3 code on the right

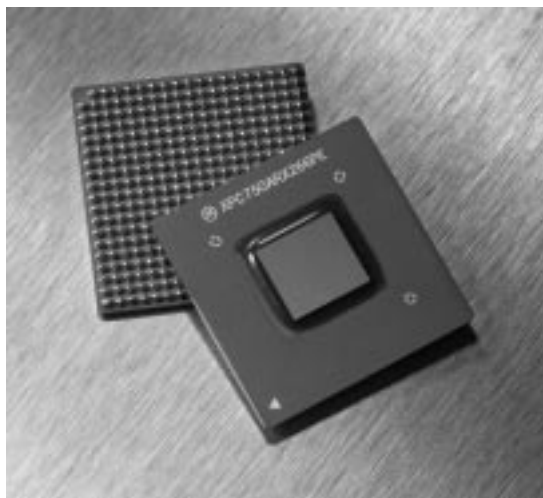


Figure 1.11 A PowerPC chip.

- 1.11** The cards that were used in the context of sorting in this chapter (ace, king, queen, etc.) provide a good example of an object. If a card is one object, and a hand and deck are other objects composed of card objects, list a few operations that might be useful in manipulating cards, hands, and decks.
- 1.12** Vending machines are objects composed of several different objects. Pick a specific kind of vending machine and list several objects that are used to “make up” the vending machine (e.g., buttons used to specify items to be bought). For each object, and for the vending machine as a whole, list several operations that might be useful in reasoning about or manipulating the objects.
Are there some characteristics that all vending machines have in common? Are there classes of vending machines, each of which differs fundamentally from other kinds of vending machines?