CHAPTER 1

Computer Science: The Mechanization of Abstraction

Abstraction

Though it is a new field, computer science already touches virtually every aspect of human endeavor. Its impact on society is seen in the proliferation of computers, information systems, text editors, spreadsheets, and all of the wonderful application programs that have been developed to make computers easier to use and people more productive. An important part of the field deals with how to make programming easier and software more reliable. But fundamentally, computer science is a science of *abstraction* — creating the right model for thinking about a problem and devising the appropriate mechanizable techniques to solve it.

Every other science deals with the universe as it is. The physicist's job, for example, is to understand how the world works, not to invent a world in which physical laws would be simpler or more pleasant to follow. Computer scientists, on the other hand, must create abstractions of real-world problems that can be understood by computer users and, at the same time, that can be represented and manipulated inside a computer.

Sometimes the process of abstraction is simple. For example, we can model the behavior of the electronic circuits used to build computers quite well by an abstraction called "propositional logic." The modeling of circuits by logical expressions is not exact; it simplifies, or abstracts away, many details — such as the time it takes for electrons to flow through circuits and gates. Nevertheless, the propositional logic model is good enough to help us design computer circuits well. We shall have much more to say about propositional logic in Chapter 12.

Exam scheduling

As another example, suppose we are faced with the problem of scheduling final examinations for courses. That is, we must assign course exams to time slots so that two courses may have their exams scheduled in the same time slot only if there is no student taking both. At first, it may not be apparent how we should model this problem. One approach is to draw a circle called a *node* for each course and draw a line called an *edge* connecting two nodes if the corresponding courses have a student in common. Figure 1.1 suggests a possible picture for five courses; the picture is called a *course-conflict graph.*

Given the course-conflict graph, we can solve the exam-scheduling problem by repeatedly finding and removing "maximal independent sets" from the graph. An

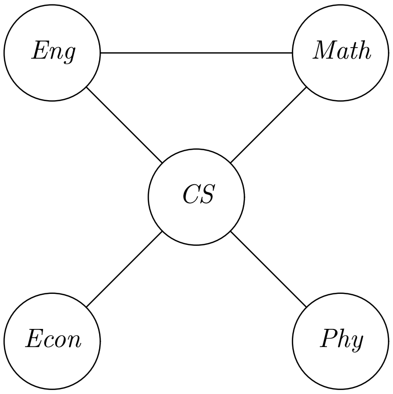


Fig. 1.1. Course-conflict graph for five courses. An edge between two courses indicates that at least one student is taking both courses.

Maximal independent set

independent set is a collection of nodes that have no connecting edges within the collection. An independent set is *maximal* if no other node from the graph can be added without including an edge between two nodes of the set. In terms of courses, a maximal independent set is any maximal set of courses with no common students. In Fig. 1.1, $\{Econ, Eng, Phy\}$ is one maximal independent set. The set of courses corresponding to the selected maximal independent set is assigned to the first time slot.

We remove from the graph the nodes in the first maximal independent set, along with all incident edges, and then find a maximal independent set among the remaining courses. One choice for the next maximal independent set is the singleton set $\{CS\}$. The course in this maximal independent set is assigned to the second time slot.

We repeat this process of finding and deleting maximal independent sets until no more nodes remain in the course-conflict graph. At this point, all courses will have been assigned to time slots. In our example, after two iterations, the only remaining node in the course-conflict graph is *Math*, and this forms the final maximal independent set, which is assigned to the third time slot. The resulting exam schedule is thus

Time Slot	Course Exams
1	*Econ, Eng, Phy*
2	*CS*
3	*Math*

This algorithm does not necessarily partition the courses among the smallest possible number of time slots, but it is simple and does tend to produce a schedule with close to the smallest number of time slots. It is also one that can be readily programmed using the techniques presented in Chapter 9.

Notice that this approach abstracts away some details of the problem that may be important. For example, it could cause one student to have five exams in five consecutive time slots. We could create a model that included limits on how many exams in a row one student could take, but then both the model and the solution

Abstraction: Not to Be Feared

The reader may cringe at the word “abstraction,” because we all have the intuition that abstract things are hard to understand; for example, abstract algebra (the study of groups, rings, and the like) is generally considered harder than the algebra we learned in high school. However, abstraction in the sense we use it implies simplification, the replacement of a complex and detailed real-world situation by an understandable model within which we can solve a problem. That is, we “abstract away” the details whose effect on the solution to a problem is minimal or nonexistent, thereby creating a model that lets us deal with the essence of the problem.

to the exam-scheduling problem would be more complicated.

Often, finding a good abstraction can be quite difficult because we are forced to confront the fundamental limitations on the tasks computers can perform and the speed with which computers can perform those tasks. In the early days of computer science, some optimists believed that robots would soon have the prodigious capability and versatility of the *Star Wars* robot C3PO. Since then we have learned that in order to have “intelligent” behavior on the part of a computer (or robot), we need to provide that computer with a model of the world that is essentially as detailed as that possessed by humans, including not only facts (“Sally’s phone number is 555-1234”), but principles and relationships (“If you drop something, it usually falls downward”).

Knowledge representation

We have made much progress on this problem of “knowledge representation.” We have devised abstractions that can be used to help build programs that do certain kinds of reasoning. One example of such an abstraction is the directed graph, in which nodes represent entities (“the species cat” or “Fluffy”) and arrows (called *arcs*) from one node to another represent relationships (“Fluffy is a cat,” “cats are animals,” “Fluffy owns Fluffy’s milk saucer”); Figure 1.2 suggests such a graph.

Another useful abstraction is formal logic, which allows us to manipulate facts by applying rules of inference, such as “If X is a cat and Y is the mother of X, then Y is a cat.” Nevertheless, progress on modeling, or abstracting, the real world or significant pieces thereof remains a fundamental challenge of computer science, one that is not likely to be solved completely in the near future.

1.1 What This Book Is About

This book will introduce the reader, who is assumed to have a working knowledge of the programming language ANSI C, to the principal ideas and concerns of computer science. The book emphasizes three important problem-solving tools:

1. *Data models*, the abstractions used to describe problems. We have already mentioned two models: logic and graphs. We shall meet many others throughout this book.

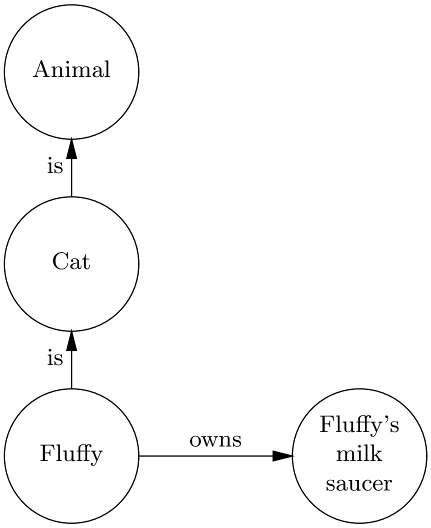


Fig. 1.2. A graph representing knowledge about Fluffy.

2. *Data structures*, the programming-language constructs used to represent data models. For example, C provides built-in abstractions, such as structures and pointers, that allow us to construct data structures to represent complex abstractions such as graphs.

3. *Algorithms*, the techniques used to obtain solutions by manipulating data as represented by the abstractions of a data model, by data structures, or by other means.

Data Models

We meet data models in two contexts. Data models such as the graphs discussed in the introduction to this chapter are abstractions frequently used to help formulate solutions to problems. We shall learn about several such data models in this book: trees in Chapter 5, lists in Chapter 6, sets in Chapter 7, relations in Chapter 8, graphs in Chapter 9, finite automata in Chapter 10, grammars in Chapter 11, and logic in Chapters 12 and 14.

Data models are also associated with programming languages and computers. For example, C has a data model that includes abstractions such as characters, integers of several sizes, and floating-point numbers. Integers and floating-point numbers in C are only approximations of integers and reals in mathematics because of the limited precision of arithmetic available in computers. The C data model also includes types such as structures, pointers, and functions, which we shall discuss in more detail in Section 1.4.

Data Structures

When the data model of the language in which we are writing a program lacks a built-in representation for the data model of the problem at hand, we must represent the needed data model using the abstractions supported by the language. For this purpose, we study *data structures*, which are methods for representing in the data model of a programming language abstractions that are not an explicit part of that language. Different programming languages may have strikingly different data models. For example, unlike C, the language Lisp supports trees directly, and the language Prolog has logic built into its data model.

Algorithms

An *algorithm* is a precise and unambiguous specification of a sequence of steps that can be carried out mechanically. The notation in which an algorithm is expressed can be any commonly understood language, but in computer science algorithms are most often expressed formally as programs in a programming language, or in an informal style as a sequence of programming language constructs intermingled with English language statements. Most likely, you have already encountered several important algorithms while studying programming. For example, there are a number of algorithms for *sorting* the elements of an array, that is, putting the elements in smallest-first order. There are clever searching algorithms such as *binary search*, which quickly finds a given element in a sorted array by repeatedly dividing in half the portion of the array in which the element could appear.

These, and many other "tricks" for solving common problems, are among the tools the computer scientist uses when designing programs. We shall study many such techniques in this book, including the important methods for sorting and searching. In addition, we shall learn what makes one algorithm better than another. Frequently, the *running time*, or time taken by an algorithm measured as a function of the size of its input, is one important aspect of the "quality" of the algorithm; we discuss running time in Chapter 3.

Other aspects of algorithms are also important, particularly their simplicity. Ideally, an algorithm should be easy to understand and easy to turn into a working program. Also, the resulting program should be understandable by a person reading the code that implements the algorithm. Unfortunately, our desires for a fast algorithm and a simple algorithm are often in conflict, and we must choose our algorithm wisely.

Underlying Threads

As we progress through this book, we shall encounter a number of important unifying principles. We alert the reader to two of these here:

1. *Design algebras*. In certain fields in which the underlying models have become well understood, we can develop notations in which design trade-offs can be expressed and evaluated. Through this understanding, we can develop a theory of design with which well-engineered systems can be constructed. Propositional logic, with the associated notation called *Boolean algebra* that we encounter in Chapter 12, is a good example of this kind of design algebra. With it, we can design efficient circuits for subsystems of the kind found in digital computers. Other examples of algebras found in this book are the algebra of sets in Chapter 7, the algebra of relations in Chapter 8, and the algebra of regular expressions in Chapter 10.

2. *Recursion* is such a useful technique for defining concepts and solving problems that it deserves special mention. We discuss recursion in detail in Chapter 2 and use it throughout the rest of the book. Whenever we need to define an object precisely or whenever we need to solve a problem, we should always ask, "What does the recursive solution look like?" Frequently that solution has a simplicity and efficiency that makes it the method of choice.

1.2 What This Chapter Is About

The remainder of this chapter sets the stage for the study of computer science. The primary concepts that will be covered are

- Data models (Section 1.3)
- The data model of the programming language C (Section 1.4)
- The principal steps in the software-creation process (Section 1.5)

We shall give examples of several different ways in which abstractions and models appear in computer systems. In particular, we mention the models found in programming languages, in certain kinds of systems programs, such as operating systems, and in the circuits from which computers are built. Since software is a vital component of today's computer systems, we need to understand the software-creation process, the role played by models and algorithms, and the aspects of software creation that computer science can address only in limited ways.

In Section 1.6 there are some conventional definitions that are used in C programs throughout this book.

1.3 Data Models

Any mathematical concept can be termed a data model. In computer science, a data model normally has two aspects:

1. The values that objects can assume. For example, many data models contain objects that have integer values. This aspect of the data model is *static*; it tells us what values objects may take. The static part of a programming language's data model is often called the *type system.*

Type system

2. The operations on the data. For example, we normally apply operations such as addition to integers. This aspect of the model is *dynamic*; it tells us the ways in which we can change values and create new values.

Programming Language Data Models

Each programming language has its own data model, and these differ from one another, often in quite substantial ways. The basic principle under which most programming languages deal with data is that each program has access to "boxes," which we can think of as regions of storage. Each box has a type, such as `int` or `char`. We may store in a box any value of the correct type for that box. We often refer to the values that can be stored in boxes as *data objects.*

Data object

Name

We may also name boxes. In general, a *name* for a box is any expression that denotes that box. Often, we think of the names of boxes as the variables of the program, but that is not quite right. For example, if `x` is a variable local to a recursive function F, then there may be many boxes named `x`, each associated with a different call to F. Then the true name of such a box is a combination of `x` and the particular call to F.

Most of the data types of C are familiar: integers, floating-point numbers, characters, arrays, structures, and pointers. These are all static notions.

Dereferencing

The operations permitted on data include the usual arithmetic operations on integers and floating-point numbers, accessing operations for elements of arrays or structures, and pointer *dereferencing*, that is, finding the element pointed to by a pointer. These operations are part of the dynamics of the C data model.

The list data model

In a programming course, we would see important data models that are not part of C, such as lists, trees, and graphs. In mathematical terms, a list is a sequence of n elements, which we shall write as $(a_1, a_2, \ldots, a_n)$, where a_1 is the first element, a_2 the second, and so on. Operations on lists include inserting new elements, deleting elements, and *concatenating* lists (that is, appending one list to the end of another).

✦ **Example 1.1.** In C, a list of integers can be represented by a data structure called a *linked list* in which list elements are stored in cells. Lists and their cells can be defined by a type declaration such as

```
typedef struct CELL *LIST;
struct CELL {
    int element;
    LIST next;
};
```

This declaration defines a self-referential structure `CELL` with two fields. The first is `element`, which holds the value of an element of the list and is of type `int`.

The second field of each `CELL` is `next`, which holds a pointer to a cell. Note that the type `LIST` is really a pointer to a `CELL`. Thus, structures of type `CELL` can be linked together by their `next` fields to form what we usually think of as a linked list, as suggested in Fig. 1.3. The `next` field can be thought of as either a pointer to the next cell or as representing the entire list that follows the cell in which it appears. Similarly, the entire list can be represented by a pointer, of type `LIST`, to the first cell on the list.

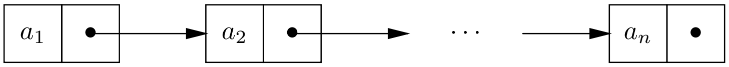


Fig. 1.3. A linked list representing the list $(a_1, a_2, \ldots, a_n)$.

Cells are represented by rectangles, the left part of which is the element, and the right part of which holds a pointer, shown as an arrow to the next cell pointed to. A dot in the box holding a pointer means that the pointer is `NULL`.[1] Lists will be covered in more detail in Chapter 6. ✦

[1] `NULL` is a symbolic constant defined in the standard header file `stdio.h` to be equal to a value that cannot be a pointer to anything. We shall use it to have this meaning throughout the book.

Data Models Versus Data Structures

Despite their similar names, a "list" and a "linked list" are very different concepts. A list is a mathematical abstraction, or data model. A linked list is a data structure. In particular, it is the data structure we normally use in C and many similar languages to represent abstract lists in programs. There are other languages in which it is not necessary to use a data structure to represent abstract lists. For example, the list $(a_1, a_2, \ldots, a_n)$ could be represented directly in the language Lisp and in the language Prolog similarly, as $[a_1, a_2, \ldots, a_n]$.

Data Models of System Software

Operating systems

Data models are found not only in programming languages but also in operating systems and applications programs. You are probably familiar with an operating system such as UNIX or MS-DOS (perhaps with Microsoft Windows).[2] The function of an operating system is to manage and schedule the resources of a computer. The data model for an operating system like UNIX has concepts such as files, directories, and processes.

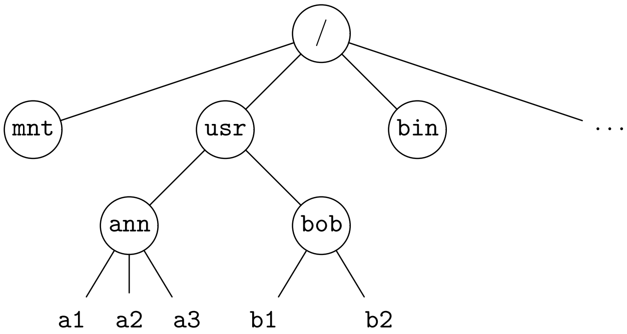


Fig. 1.4. A typical UNIX directory/file structure.

Files

1. The data itself is stored in *files*, which in the UNIX system are strings of characters.

Directories

2. Files are organized into *directories*, which are collections of files and/or other directories. The directories and files form a tree with the files at the leaves.[3] Figure 1.4 suggests the tree that might represent the directory structure of a typical UNIX operating system. Directories are indicated by circles. The root directory `/` contains directories called `mnt`, `usr`, `bin`, and so on. The directory `/usr` contains directories `ann` and `bob`; directory `ann` contains three files: `a1`, `a2`, and `a3`.

[2] If you are unfamiliar with operating systems, you can skip the next paragraphs. However, most readers have probably encountered an operating system, perhaps under another name. For example, the Macintosh "system" is an operating system, although different terminology is used. For example, a directory becomes a "folder" in Macintosh-ese.

[3] However, "links" in directories may make it appear that a file or directory is part of several different directories.

Processes

Pipes

3. *Processes* are individual executions of programs. Processes take zero or more streams as input and produce zero or more streams as output. In the UNIX system, processes can be combined by *pipes*, where the output from one process is fed as input into the next process. The resulting composition of processes can be viewed as a single process with its own input and output.

✦ **Example 1.2.** Consider the UNIX command line

```
bc | word | speak
```

The symbol `|` indicates a *pipe*, an operation that makes the output of the process on the left of this symbol be the input to the process on its right. The program `bc` is a desk calculator that takes arithmetic expressions, such as $2 + 3$, as input and produces the answer 5 as output. The program `word` translates numbers into words; `speak` translates words into phoneme sequences, which are then uttered over a loudspeaker by a voice synthesizer. Connecting these three programs together by pipes turns this UNIX command line into a single process that behaves like a "talking" desk calculator. It takes as input arithmetic expressions and produces as output the spoken answers. This example also suggests that a complex task may be implemented more easily as the composition of several simpler functions. ✦

There are many other aspects to an operating system, such as how it manages security of data and interaction with the user. However, even these few observations should make it apparent that the data model of an operating system is rather different from the data model of a programming language.

Text editors

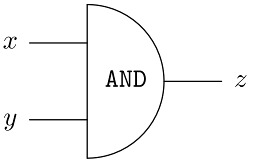
Another type of data model is found in text editors. Every text editor's data model incorporates a notion of text strings and editing operations on text. The data model usually includes the notion of *lines*, which, like most files, are character strings. However, unlike files, lines may have associated line numbers. Lines may also be organized into larger units such as paragraphs, and operations on lines are normally applicable anywhere within the line — not just at the front, like the most common file operations. The typical editor supports a notion of a "current" line (where the cursor is) and probably a current position within that line. Operations performed by the editor include various modifications to lines, such as deletion or insertion of characters within the line, deletion of lines, and creation of new lines. It is also possible in typical editors to search for features, such as specific character strings, among the lines of the file being edited.

In fact, if you examine any other familiar piece of software, such as a spreadsheet or a video game, a pattern emerges. Each program that is designed to be used by others has its own data model, within which the user must work. The data models we meet are often radically different from one another, both in the primitives they use to represent data and in the operations on that data that are offered to the user. Yet each data model is implemented, via data structures and the programs that use them, in some programming language.

The Data Model of Circuits

We shall also meet in this book a data model for computer circuits. This model, called *propositional logic*, is most useful in the design of computers. Computers are composed of elementary components called *gates*. Each gate has one or more inputs and one output; the value of an input or output can be only 0 or 1. A gate performs a simple function — such as AND, where the output is 1 if all the inputs are 1 and the output is 0 if one or more of the inputs are 0. At one level of abstraction, computer design is the process of deciding how to connect gates to perform the basic operations of a computer. There are many other levels of abstraction associated with computer design as well.

Figure 1.5 shows the usual symbol for an AND-gate, together with its *truth table,* which indicates the output value of the gate for each pair of input values.[4] We discuss truth tables in Chapter 12 and gates and their interconnections in Chapter 13.



x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

Fig. 1.5. An AND-gate and its truth table.

❖ **Example 1.3.** To execute the C assignment statement `a = b+c`, a computer performs the addition with an *adder* circuit. In the computer, all numbers are represented in binary notation using the two digits 0 and 1 (called *binary digits*, or *bits* for short). The familiar algorithm for adding decimal numbers, where we add the digits at the right end, generate a carry to the next place to the left, add that carry and the digits at that place, generate a carry to the next place to the left, and so on, works in binary as well.

Bit

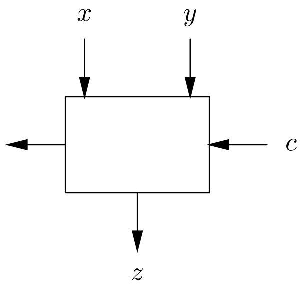


Fig. 1.6. A one-bit adder: dz is the sum $x + y + c$.

One-bit adder

Out of a few gates, we can build a *one-bit adder* circuit, as suggested in Fig. 1.6. Two input bits, x and y, and a *carry-in* bit c, are summed, resulting in a *sum* bit z and a *carry-out* bit d. To be precise, d is 1 if two or more of c, x, and y are 1, while z is 1 if an odd number (one or three) of c, x, and y are 1, as suggested by

[4] Note that if we think of 1 as "true" and 0 as "false," then the AND-gate performs the same logical operation as the `&&` operator of C.

x	y	c	d	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Fig. 1.7. Truth table for the one-bit adder.

The Ripple-Carry Addition Algorithm

We all have used the ripple-carry algorithm to add numbers in decimal. To add $456 + 829$, for example, one performs the steps suggested below:

$$\begin{array}{rrrr} & & 1 & \\ & 4 & 5 & 6 \\ & 8 & 2 & 9 \\ \hline & & & 5 \end{array} \qquad \begin{array}{rrrr} & 0 & & \\ & 4 & 5 & 6 \\ & 8 & 2 & 9 \\ \hline & & 8 & 5 \end{array} \qquad \begin{array}{rrrr} & & & \\ & 4 & 5 & 6 \\ & 8 & 2 & 9 \\ \hline 1 & 2 & 8 & 5 \end{array}$$

That is, at the first step, we add the rightmost digits, $6+9 = 15$. We write down the 5 and carry the 1 to the second column. At the second step, we add the carry-in, 1, and the two digits in the second place from the right, to get $1+5+2 = 8$. We write down the 8, and the carry is 0. In the third step, we add the carry-in, 0, and the digits in the third place from the right, to get $0 + 4 + 8 = 12$. We write down the 2, but since we are at the leftmost place, we do not carry the 1, but rather write it down as the leftmost digit of the answer.

Binary ripple-carry addition works the same way. However, at each place, the carry and the "digits" being added are all either 0 or 1. The one-bit adder thus describes completely the addition table for a single place. That is, if all three bits are 0, then the sum is 0, and so we write down 0 and carry 0. If one of the three is 1, the sum is 1; we write down 1 and carry 0. If two of the three are 1, the sum is 2, or 10 in binary; we write down 0 and carry 1. If all three are 1, then the sum is 3, or 11 in binary, and so we write down 1 and carry 1. For example, to add 101 to 111 using binary ripple-carry addition, the steps are

$$\begin{array}{rrrr} & & 1 & \\ & 1 & 0 & 1 \\ & 1 & 1 & 1 \\ \hline & & & 0 \end{array} \qquad \begin{array}{rrrr} & 1 & & \\ & 1 & 0 & 1 \\ & 1 & 1 & 1 \\ \hline & & 0 & 0 \end{array} \qquad \begin{array}{rrrr} & & & \\ & 1 & 0 & 1 \\ & 1 & 1 & 1 \\ \hline 1 & 1 & 0 & 0 \end{array}$$

the table of Fig. 1.7. The carry-out bit followed by the sum bit — that is, dz — forms a two-bit binary number, which is the total number of x, y, and c that are 1. In this sense, the one-bit adder adds its inputs.

Many computers represent integers as 32-bit numbers. An adder circuit can

Ripple-carry adder

then be composed of 32 one-bit adders, as suggested in Fig. 1.8. This circuit is often called a *ripple-carry adder,* because the carry ripples from right to left, one bit at a time. Note that the carry into the rightmost (low-order bit) one-bit adder is always 0. The sequence of bits $x_{31}x_{30}\cdots x_0$ represents the bits of the first number being added, and $y_{31}y_{30}\cdots y_0$ is the second addend. The sum is $dz_{31}z_{30}\cdots z_0$; that is, the leading bit is the carry-out of the leftmost one-bit adder, and the following bits of the sum are the sum bits of the adders, from the left.

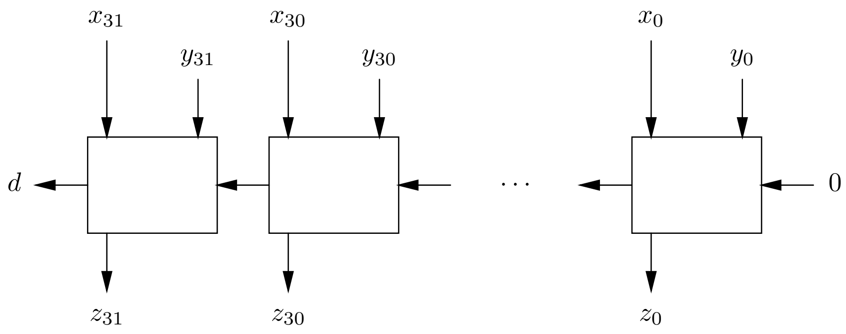


Fig. 1.8. A ripple-carry adder: $dz_{31}z_{30}\cdots z_0 = x_{31}x_{30}\cdots x_0 + y_{31}y_{30}\cdots y_0$.

The circuit of Fig. 1.8 is really an algorithm in the data model of bits and the primitive operations of gates. However, it is not a particularly good algorithm. The reason is that until we compute the carry-out of the rightmost place, we cannot compute z_1 or the carry-out of the second place. Until we compute the carry-out of the second place, we cannot compute z_2 or the carry-out of the third place, and so on. Thus, the time taken by the circuit is the length of the numbers being added — 32 in our case — multiplied by the time needed by a one-bit adder.

One might suspect that the need to "ripple" the carry through each of the one-bit adders, in turn, is inherent in the definition of addition. Thus, it may come as a surprise to the reader that computers have a much faster way of adding numbers. We shall cover such an improved algorithm for addition when we discuss the design of circuits in Chapter 13. ✦

EXERCISES

1.3.1: Explain the difference between the static and dynamic aspects of a data model.

1.3.2: Describe the data model of your favorite video game. Distinguish between static and dynamic aspects of the model. *Hint*: The static parts are not just the parts of the game board that do not move. For example, in Pac Man, the static part includes not only the map, but the "power pills," "monsters," and so on.

1.3.3: Describe the data model of your favorite text editor.

1.3.4: Describe the data model of a spreadsheet program.

1.4 The C Data Model

In this section we shall highlight important parts of the data model used by the C programming language. As an example of a C program, consider the program in Fig. 1.10 that uses the variable `num` to count the number of characters in its input.

```
#include <stdio.h>
main()
{
    int num;
    num = 0;
    while (getchar() != EOF)
        ++num; /* add 1 to num */
    printf("%d\n", num);
}
```

Fig. 1.10. C program to count number of input characters.

The first line tells the C preprocessor to include as part of the source the standard input/output file `stdio.h`, which contains the definitions of the functions `getchar` and `printf`, and the symbolic constant `EOF`, a value that represents the end of a file.

A C program itself consists of a sequence of definitions, which can be either function definitions or data definitions. One must be a definition of a function called `main`. The first statement in the function body of the program in Fig. 1.10 declares the variable `num` to be of type `int`. (All variables in a C program must be declared before their use.) The next statement initializes `num` to zero. The following `while` statement reads input characters one at a time using the library function `getchar`, incrementing `num` after each character read, until there are no more input characters. The end of file is signaled by the special value `EOF` on the input. The `printf` statement prints the value of `num` as a decimal integer, followed by a newline character.

`EOF`

The C Type System

We begin with the static part of the C data model, the type system, which describes the values that data may have. We then discuss the dynamics of the C data model, that is, the operations that may be performed on data.

In C, there is an infinite set of types, any of which could be the type associated with a particular variable. These types, and the rules by which they are constructed, form the *type system* of C. The type system contains basic types such as integers, and a collection of type-formation rules with which we can construct progressively more complex types from types we already know. The basic types of C are

1. Characters (`char`, `signed char`, `unsigned char`)
2. Integers (`int`, `short int`, `long int`, `unsigned`)
3. Floating-point numbers (`float`, `double`, `long double`)
4. Enumerations (`enum`)

Integers and floating-point numbers are considered to be *arithmetic types.*

The type-formation rules assume that we already have some types, which could be basic types or other types that we have already constructed using these rules. Here are some examples of the type formation rules in C:

1. *Array types.* We can form an array whose elements are type T with the declaration

   ```
   $T$ A[n]
   ```

   This statement declares an array `A` of n elements, each of type T. In C, array subscripts begin at 0, so the first element is `A[0]` and the last element is `A[`$n-1$`]`. Arrays can be constructed from characters, arithmetic types, pointers, structures, unions, or other arrays.

Members of a structure

2. *Structure types.* In C, a structure is a grouping of variables called *members* or *fields.* Within a structure different members can have different types, but each member must have elements of a single type. If $T_1, T_2, \ldots, T_n$ are types and $M_1, M_2, \ldots, M_n$ are member names, then the declaration

   ```
   struct $S$ {
       $T_1$ $M_1$;
       $T_2$ $M_2$;
         ...
       $T_n$ $M_n$;
   }
   ```

   defines a structure whose *tag* (i.e., the name of its type) is S and that has n members. The ith member has the name M_i and a value of type T_i, for $i = 1, 2, \ldots, n$. Example 1.1 is an illustration of a structure. This structure has tag `CELL` and two members. The first member has name `element` and has integer type. The second has name `next` and its type is a pointer to a structure of the same type.

   The structure tag S is optional, but it provides a convenient shorthand for referring to the type in later declarations. For example, the declaration

   ```
   struct $S$ myRecord;
   ```

   defines the variable `myRecord` to be a structure of type S.

3. *Union types.* A union type allows a variable to have different types at different times during the execution of a program. The declaration

   ```
   union {
       $T_1$ $M_1$;
       $T_2$ $M_2$;
         ...
       $T_n$ $M_n$;
   } x;
   ```

   defines a variable `x` that can hold a value of any of the types $T_1, T_2, \ldots, T_n$. The member names $M_1, M_2, \ldots, M_n$ help indicate which type the value of `x` should be regarded as being. That is, `x.`M_i refers to the value of `x` treated as a value of type T_i.

4. *Pointer types.* C is distinctive for its reliance on pointers. A variable of type pointer contains the address of a region of storage. We can access the value of another variable indirectly through a pointer. The declaration

   ```
   T *p;
   ```

   defines the variable `p` to be a pointer to a variable of type T. Thus `p` names a box of type pointer to T and the value in box `p` is a pointer. We often draw the value of `p` as an arrow, rather than as an object of type T itself, as shown in Fig. 1.11. What really appears in the box named `p` is the address, or location, at which an object of type T is stored in the computer.

   Consider the declaration

   ```
   int x, *p;
   ```

   In C, the unary operator `&` is used to obtain the address of an object, so the statement

   ```
   p = &x;
   ```

   assigns the address of `x` to `p`; that is, it makes `p` point to `x`.

   The unary operator `*` applied to `p` fetches the value of the box pointed to by `p`, so the statement

   ```
   y = *p;
   ```

   assigns to `y` the contents of whatever box `p` points to. If `y` is a variable of type `int`, then

   ```
   p = &x;
   y = *p;
   ```

   is equivalent to the assignment

   ```
   y = x;
   ```

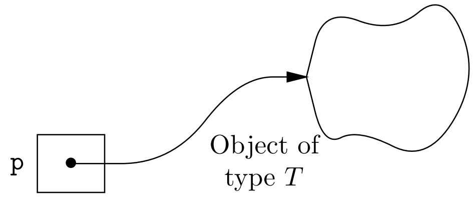


Fig. 1.11. Variable `p` is of type pointer to T.

✦ **Example 1.4.** C has the `typedef` construct to create synonyms for type names. The declaration

```
typedef int Distance;
```

```
typedef int type1[10];

typedef type1 *type2;

typedef struct {
    int field1;
    type2 field2;
} type3;

typedef type3 type4[5];
```

Fig. 1.12. Some C `typedef` declarations.

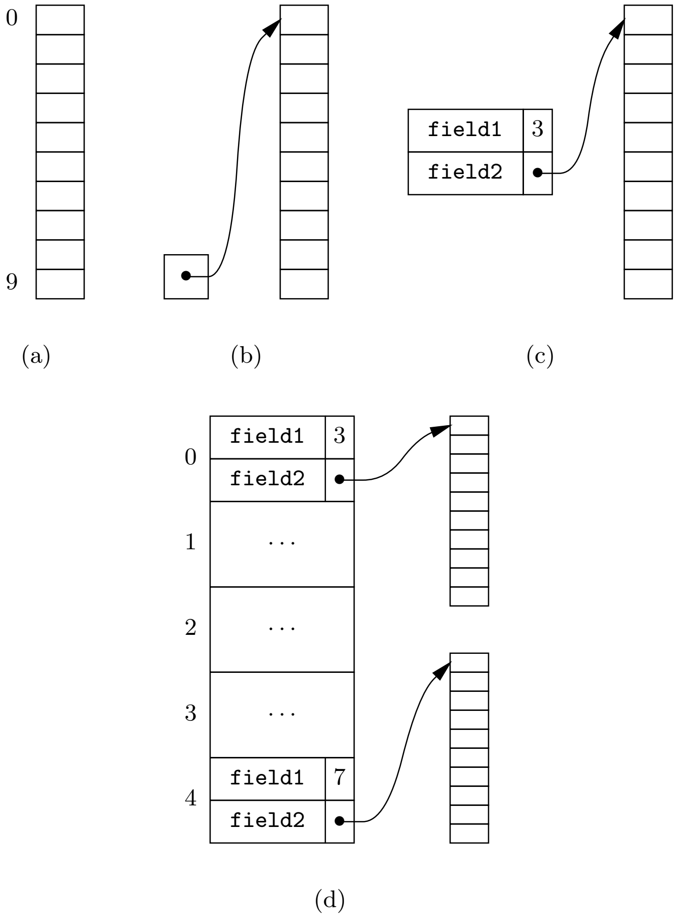


Fig. 1.13. Visualization of type declarations in Fig. 1.12.

Types, Names, Variables, and Identifiers

A number of terms associated with data objects have different meanings but are easy to confuse. First, a type describes a "shape" for data objects. In C, a new name T may be defined for an existing type using the `typedef` construct

`typedef` <type descriptor> T

Type descriptor

Here the *type descriptor* is an expression that tells us the shape of objects of the type T.

A typedef declaration for T does not actually create any objects of that type. An object of type T is created by a declaration of the form

T `x;`

Here, `x` is an *identifier,* or "variable name." Possibly, `x` is *static* (not local to any function), in which case the box for `x` is created when the program starts. If `x` is not static, then it is local to some function F. When F is called, a box whose name is "the `x` associated with this call to F" is created. More precisely, the name of the box is `x`, but only uses of the identifier `x` during the execution of this call to F refer to this box.

As mentioned in the text, there can be many boxes each of whose name involves the identifier `x`, since F may be recursive. There may even be other functions that also have used identifier `x` to name one of their variables. Moreover, names are more general than identifiers, since there are many kinds of expressions that could be used to name boxes. For instance, we mentioned that `*p` could be the name of an object pointed to by pointer `p`, and other names could be complex expressions such as `(*p).f[2]` or `p->f[2]`. The last two expressions are equivalent and refer to the array element number 2 of the field `f` of the structure pointed to by pointer `p`.

allows the name `Distance` to be used in place of the type `int`.

Consider the four typedef declarations in Fig. 1.12. In the conventional view of data in C, an object of type `type1` is an array with 10 slots, each holding an integer, as suggested in Fig. 1.13(a). Likewise, objects of type `type2` are pointers to such arrays, as in Fig. 1.13(b). Structures, like those of `type3`, are visualized as in Fig. 1.13(c), with a slot for each field; note that the name of the field (e.g., `field1`) does not actually appear with the value of the field. Finally, objects of the array type `type4` would have five slots, each of which holds an object of type `type3`, a structure we suggest in Fig. 1.13(d). ✦

Functions

Functions also have associated types, even though we do not associate boxes or "values" with functions, as we do with program variables. For any list of types $T_1, T_2, \ldots, T_n$, we can define a function with n parameters consisting of those types, in order. This list of types followed by the type of the value returned by the function (the *return-value*) is the "type" of the function. If the function has no return value, its type is `void`.

Return-value

In general, we can build types by applying the type-construction rules arbitrarily, but there are a number of constraints. For example, we cannot construct an "array of functions," although we can construct an array of pointers to functions. The complete set of rules for constructing types in C can be found in the ANSI standard.

Operations in the C Data Model

The operations on data in the C data model can be divided into three categories:

1. Operations that create or destroy a data object.
2. Operations that access and modify parts of a data object.
3. Operations that combine the values of data objects to form a new value for a data object.

Data Object Creation and Disposal

For data creation, C provides several rudimentary mechanisms. When a function is called, boxes for each of its local arguments (parameters) are created; these serve to hold the values of the arguments.

Another mechanism for data creation is through the use of the library routine `malloc(n)`, which returns a pointer to n consecutive character positions of unused storage that can be used to store data by the caller of `malloc`. Data objects can then be created in this storage region.

C has the analogous methods for destroying data objects. Local parameters of a function call cease to exist when the function returns. The routine `free` releases the storage created by `malloc`. In particular, the effect of calling `free(p)` is to release the storage area pointed to by `p`. It is disastrous to use `free` to get rid of an object that was not created by calling `malloc`.

Data Access and Modification

C has mechanisms for accessing the components of objects. It uses `a[i]` to access the ith element of array `a`, `x.m` to access member `m` of a structure named `x`, and `*p` to access the object pointed to by pointer `p`.

Modifying, or *writing*, values in C is done principally by the assignment operators, which allow us to change the value of an object.

✦ **Example 1.5.** If `a` is a variable of type `type4` defined in Example 1.4, then

```
    (*a[0].field2)[3] = 99;
```

assigns the value 99 to the fourth element of the array pointed to by `field2` in the structure that is the first element of the array `a`. ✦

Data Combination

C has a rich set of operators for manipulating and combining values. The principal operators are

1. *Arithmetic operators.* C provides:

a) The customary binary arithmetic operators +, −, ∗, / on integers and floating-point numbers. Integer division truncates (4/3 yields 1).

b) There are the unary + and − operators.

c) The modulus operator `i % j` produces the remainder when `i` is divided by `j`.

d) The increment and decrement operators, `++` and `--`, applied to a single integer variable add or subtract 1 from that variable, respectively. These operators can appear before or after their operand, depending on whether we wish the value of the expression to be computed before or after the change in the variable's value.

2. *Logical operators.* C does not have a Boolean type; it uses zero to represent the logical value false, and nonzero to represent true.[5] C uses:

   a) `&&` to represent `AND`. For example, the expression `x && y` returns 1 if both operands are nonzero, 0 otherwise. However, `y` is not evaluated if `x` has the value 0.

   b) `||` represents `OR`. The expression `x || y` returns 1 if either `x` or `y` is nonzero, and returns 0 otherwise. However, `y` is not evaluated if `x` is nonzero.

   c) The unary negation operator `!x` returns 0 if x is nonzero and returns 1 if $x = 0$.

   d) The conditional operator is a ternary (3-argument) operator represented by a question mark and a colon. The expression `x?y:z` returns the value of `y` if `x` is true (i.e., it is nonzero) and returns the value of `z` if `x` is false (i.e., 0).

3. *Comparison operators.* The result of applying one of the six relational comparison operators (`==`, `!=`, `<`, `>`, `<=`, and `>=`) to integers or floating point numbers is 0 if the relation is false and 1 otherwise.

4. *Bitwise manipulation operators.* C provides several useful bitwise logical operators, which treat integers as if they were bits strings equal to their binary representations. These include `&` for bitwise `AND`, `|` for bitwise inclusive-or, `^` for bitwise exclusive-or, `<<` for left shift, `>>` for right shift, and a tilde for negation.

5. *Assignment operators.* C uses `=` as the assignment operator. In addition, C allows expressions such as

```
x = x + y;
```

   to be written in a shortened form

```
x += y;
```

   Similar forms apply to the other binary arithmetic operators.

[5] We shall use `TRUE` and `FALSE` as defined constants 1 and 0, respectively, to represent Boolean values; see Section 1.6.

6. *Coercion operators.* Coercion is the process of converting a value of one type into an equivalent value of another type. For example, if `x` is a floating-point number and `i` is an integer, then `x = i` causes the integer value of `i` to be converted to a floating-point number with the same value. Here, the coercion operator is not shown explicitly, but the C compiler can deduce that conversion from integer to float is necessary and inserts the required step.

EXERCISES

1.4.1: Explain the difference between an identifier of a C program and a name (for a "box" or data object).

1.4.2: Give an example of a C data object that has more than one name.

1.4.3: If you are familiar with another programming language besides C, describe its type system and operations.

1.5 Algorithms and the Design of Programs

The study of data models, their properties, and their appropriate use is one pillar of computer science. A second, equally important pillar is the study of algorithms and their associated data structures. We need to know the best ways to perform common tasks, and we need to learn the principal techniques for designing good algorithms. Further, we need to understand how the use of data structures and algorithms fits into the process of creating useful programs. The themes of data models, algorithms, data structures, and their implementation in programs are interdependent, and each appears many times throughout the book. In this section, we shall mention some generalities regarding the design and implementation of programs.

The Creation of Software

In a programming class, when you were given a programming problem, you probably designed an algorithm to solve the problem, implemented the algorithm in some language, compiled and ran the program on some sample data, and then submitted the program to be graded.

In a commercial setting, programs are written under rather different circumstances. Algorithms, at least those simple enough and common enough to have names, are usually small parts of a complete program. Programs, in turn, are usually components of a larger system, involving hardware as well as software. Both the programs and the complete systems in which they are embedded are developed by teams of programmers and engineers; there could be hundreds of people on such a team.

Software development process

The development of a software system typically spans several phases. Although these phases may superficially bear some resemblance to the steps involved in solving the classroom programming assignment, most of the effort in building a software system to solve a given problem is not concerned with programming. Here is an idealized scenario.

Problem definition and specification. The hardest, but most important, part of the task of creating a software system is defining what the problem really is and then specifying what is needed to solve it. Usually, problem definition begins by analyzing the users' requirements, but these requirements are often imprecise and hard to write down. The system architect may have to consult with the future users of the system and iterate the specification, until both the specifier and the users are satisfied that the specification defines and solves the problem at hand. In the specification stage, it may be helpful to build a simple prototype or model of the final system, to gain insight into its behavior and intended use. Data modeling is also an important tool in the problem-definition phase.

Prototyping

Design. Once the specification is complete, a high-level design of the system is created, with the major components identified. A document outlining the high-level design is prepared, and performance requirements for the system may be included. More detailed specifications of some of the major components may also be included during this phase. A cost-effective design often calls for the reuse or modification of previously constructed components. Various software methodologies such as object-oriented technology facilitate the reuse of components.

Software reuse

Implementation. Once the design is fixed, implementation of the components can proceed. Many of the algorithms discussed in this book are useful in the implementation of new components. Once a component has been implemented, it is subject to a series of tests to make sure that it behaves as specified.

Integration and system testing. When the components have been implemented and individually tested, the entire system is assembled and tested.

Installation and field testing. Once the developer is satisfied that the system will work to the customer's satisfaction, the system is installed on the customer's premises and the final field testing takes place.

Maintenance. At this point, we might think that the bulk of the work has been done. Maintenance remains, however, and in many situations maintenance can account for more than half the cost of system development. Maintenance may involve modifying components to eliminate unforeseen side-effects, to correct or improve system performance, or to add features. Because maintenance is such an important part of software systems design, it is important to write programs that are correct, rugged, efficient, modifiable, and — whenever possible — portable from one computer to another.

It is very important to catch errors as early as possible, preferably during the problem-definition phase. At each successive phase, the cost of fixing a design error or programming bug rises greatly. Independent reviews of requirements and designs are beneficial in reducing downstream errors.

Programming Style

An individual programmer can ease the maintenance burden greatly by writing programs that others can read and modify readily. Good programming style comes only with practice, and we recommend that you begin at once to try writing programs that are easy for others to understand. There is no magic formula that will guarantee readable programs, but there are several useful rules of thumb:

1. Modularize a program into coherent pieces.

2. Lay out a program so that its structure is clear.

3. Write intelligent comments to explain a program. Describe, clearly and precisely, the underlying data models, the data structures selected to represent them, and the operation performed by each procedure. When describing a procedure, state the assumptions made about its inputs, and tell how the output relates to the input.

4. Use meaningful names for procedures and variables.

Defined constants

5. Avoid explicit constants whenever possible. For example, do not use 7 for the number of dwarfs. Rather, use a defined constant such as `NumberOfDwarfs`, so that you can easily change all uses of this constant to 8, if you decide to add another dwarf.

Global variables

6. Avoid the use of "global variables" — that is, variables defined for the program as a whole — unless the data represented by that variable really is used by most of the procedures of the program.

Test suite

Another good programming practice is to maintain a test suite of inputs that will try to exercise every line of code while you are developing a program. Whenever new features are added to the program, the test suite can be run to make sure that the new program has the same behavior as the old on previously working inputs.

1.6 Some C Conventions Used Throughout the Book

There are several definitions and conventions that we shall find useful as we illustrate concepts with C programs. Some of these are common conventions found in the standard header file `stdio.h`, while others are defined specially for the purposes of this book and must be included with any C program that uses them.

`NULL`

1. The identifier `NULL` is a value that may appear anywhere a pointer can appear, but it is not a value that can ever point to anything. Thus, `NULL` in a field such as `next` in the cells of Example 1.1 can be used to indicate the end of a list. We shall see that `NULL` has a number of similar uses in other data structures. `NULL` is properly defined in `stdio.h`.

2. The identifiers `TRUE` and `FALSE` are defined by

```
#define TRUE 1
#define FALSE 0
```

`TRUE` **and** `FALSE`

Thus, `TRUE` can be used anywhere a condition with logical value true is wanted, and `FALSE` can be used for a condition whose value is false.

`BOOLEAN`

3. The type `BOOLEAN` is defined as

```
typedef int BOOLEAN;
```

We use `BOOLEAN` whenever we want to stress the fact that we are interested in the logical rather than the numeric value of an expression.

EOF

4. The identifier `EOF` is a value that is returned by file-reading functions such as `getchar()` when there are no more bytes left to be read from the file. An appropriate value for `EOF` is provided in `stdio.h`.

Cell definition

5. We shall define a macro that generates declarations of cells of the kind used in Example 1.1. An appropriate definition appears in Fig. 1.14. It declares cells with two fields: an `element` field whose type is given by the parameter `EltType` and a `next` field to point to a cell with this structure. The macro provides two external definitions: `CellType` is the name of structures of this type, and `ListType` is a name for the type of pointers to these cells.

```
#define DefCell(EltType, CellType, ListType)                \
typedef struct CellType *ListType;                          \
struct CellType {                                           \
    EltType element;                                        \
    ListType next;                                          \
}
```

Fig. 1.14. A macro for defining list cells.

❖ **Example 1.6.** We can define cells of the type used in Example 1.1 by the macro use

```
DefCell(int, CELL, LIST);
```

The macro then expands into

```
typedef struct CELL *LIST;
struct CELL {
    int element;
    LIST next;
}
```

As a consequence, we can use `CELL` as the type of integer cells, and we can use `LIST` as the type of pointers to these cells. For example,

```
CELL c;
LIST L;
```

defines `c` to be a cell and `L` to be a pointer to a cell. Note that the representation of a list of cells is normally a pointer to the first cell on the list, or `NULL` if the list is empty. ❖

1.7 Summary of Chapter 1

At this point you should be aware of the following concepts:

❖ How data models, data structures, and algorithms are used to solve problems

- The distinction between a list as a data model and a linked list as a data structure
- The presence of some kind of data model in every software system, be it a programming language, an operating system, or an application program
- The key elements of the data model supported by the programming language C
- The major steps in the development of a large software system

1.8 Bibliographic Notes for Chapter 1

Kernighan and Ritchie [1988] is the classic reference for the C programming language. Roberts [1994] is a good introduction to programming using C.

Stroustrup [1991] has created an object-oriented extension of C called C++ that is now widely used for implementing systems. Sethi [1989] provides an introduction to the data models of several major programming languages.

Brooks [1974] eloquently describes the technical and managerial difficulties in developing large software systems. Kernighan and Plauger [1978] provide sound advice for improving your programming style.

American National Standards Institute (ANSI) [1990]. *Programming Language C*, American National Standards Institute, New York.

Brooks, F. P. [1974]. *The Mythical Man Month*, Addison-Wesley, Reading, Mass.

Kernighan, B. W., and P. J. Plauger [1978]. *The Elements of Programming Style, second edition*, McGraw-Hill, New York.

Kernighan, B. W., and D. M. Ritchie [1988]. *The C Programming Language*, second edition, Prentice-Hall, Englewood Cliffs, New Jersey.

Roberts, E. S. [1994]. *A C-Based Introduction to Computer Science*, Addison-Wesley, Reading, Mass.

Sethi, R. [1989]. *Programming Languages: Concepts and Constructs*, Addison-Wesley, Reading, Mass.

Stroustrup, B. [1991]. *The C++ Programming Language*, second edition, Addison-Wesley, Reading, Mass.