1 Introduction to Data Structures

A computer is a machine that manipulates information. The study of computer science includes the study of how information is organized in a computer, how it can be manipulated, and how it can be utilized. Thus, it is exceedingly important for a student of computer science to understand the concepts of information organization and manipulation in order to continue study of the field.

1.1 INFORMATION AND MEANING

If computer science is fundamentally the study of information, the first question that arises is, what is information? Unfortunately, although the concept of information is the bedrock of the entire field, this question cannot be answered precisely. In this sense the concept of information in computer science is similar to the concepts of point, line, and plane in geometry: they are all undefined terms about which statements can be made but which cannot be explained in terms of more elementary concepts.

In geometry it is possible to talk about the length of a line despite the fact that the concept of a line is itself undefined. The length of a line is a measure of quantity. Similarly, in computer science we can measure quantities of information. The basic unit of information is the ***bit***, whose value asserts one of two mutually exclusive possibilities. For example, if a light switch can be in one of two positions but not in both simultaneously, the fact that it is either in the "on" position or the "off" position is one bit of information. If a device can be in more than two possible states, the fact that it is in a particular state is more than one bit of information. For example, if a dial has

eight possible positions, the fact that it is in position 4 rules out seven other possibilities, whereas the fact that a light switch is on rules out only one other possibility.

Another way of thinking of this phenomenon is as follows. Suppose that we had only two-way switches but could use as many of them as we needed. How many such switches would be necessary to represent a dial with eight positions? Clearly, one switch can represent only two positions (see Figure 1.1.1a). Two switches can represent four different positions (Figure 1.1.1b), and three switches are required to represent eight different positions (Figure 1.1.1c). In general, n switches can represent 2^n different possibilities.

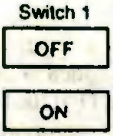
The binary digits 0 and 1 are used to represent the two possible states of a particular bit (in fact, the word "bit" is a contraction of the words "binary digit"). Given n bits, a string of n 1s and 0s is used to represent their settings. For example, the string 101011 represents six switches, the first of which is "on" (1), the second of which is "off" (0), the third on, the fourth off, and the fifth and sixth on.

We have seen that three bits are sufficient to represent eight possibilities. The eight possible configurations of these three bits (000, 001, 010, 011, 100, 101, 110, and 111) can be used to represent the integers 0 through 7. However, there is nothing about these bit settings that intrinsically implies that a particular setting represents a particular integer. Any assignment of integer values to bit settings is valid as long as no two integers are assigned to the same bit setting. Once such an assignment has been made, a particular bit setting can be unambiguously interpreted as a specific integer. Let us examine several widely used methods for interpreting bit settings as integers.

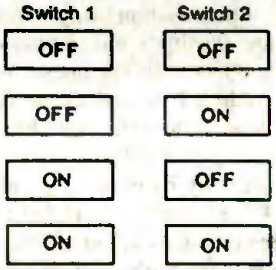
Binary and Decimal Integers

The most widely used method for interpreting bit settings as nonnegative integers is the ***binary number system***. In this system each bit position represents a power of 2. The rightmost bit position represents 2^0 which equals 1, the next position to the left represents 2^1 which is 2, the next bit position represents 2^2 which is 4, and so on. An integer is represented as a sum of powers of 2. A string of all 0s represents the number 0. If a 1 appears in a particular bit position, the power of 2 represented by that bit position is included in the sum; but if a 0 appears, that power of 2 is not included in the sum. For example, the group of bits 00100110 has 1s in positions 1, 2, and 5 (counting from right to left with the rightmost position counted as position 0). Thus 00100110 represents the integer $2^1 + 2^2 + 2^5 = 2 + 4 + 32 = 38$. Under this interpretation, any string of bits of length n represents a unique nonnegative integer between 0 and $2^n - 1$, and any nonnegative integer between 0 and $2^n - 1$ can be represented by a unique string of bits of length n.

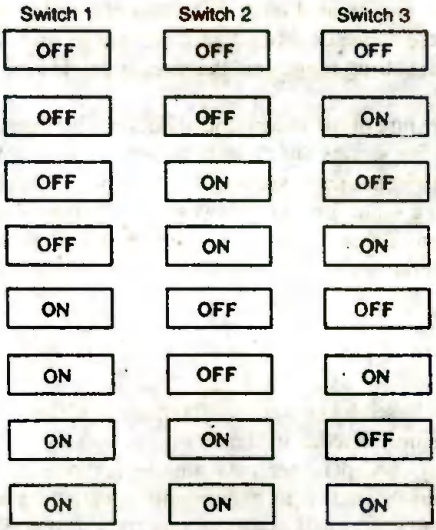
There are two widely used methods for representing negative binary numbers. In the first method, called ***ones complement notation,*** a negative number is represented by changing each bit in its absolute value to the opposite bit setting. For example, since 00100110 represents 38, 11011001 is used to represent -38. This means that the leftmost bit of a number is no longer used to represent a power of 2 but is reserved for the sign of the number. A bit string starting with a 0 represents a positive number, whereas a bit string starting with a 1 represents a negative number. Given n bits, the range of numbers that can be represented is $-2^{(n-1)} + 1$ (a 1 followed by $n - 1$ zeros) to



(a) One switch (two possibilities).



(b) Two switches (four possibilities).



(c) Three switches (eight possibilities).

Figure 1.1.1

$2^{(n-1)} - 1$ (a 0 followed by $n-1$ ones). Note that under this representation, there are two representations for the number 0: a "positive" 0 consisting of all 0s, and a "negative" 0 consisting of all 1s.

The second method of representing negative binary numbers is called ***twos complement notation***. In this notation, 1 is added to the ones complement representation of a negative number. For example, since 11011001 represents -38 in ones complement notation, 11011010 is used to represent -38 in twos complement notation. Given n bits, the range of numbers that can be represented is $-2^{(n-1)}$ (a 1 followed by $n-1$ zeros) to $2^{(n-1)} - 1$ (a 0 followed by $n-1$ ones). Note that $-2^{(n-1)}$ can be represented in twos complement notation but not in ones complement notation. However, its absolute value $2^{(n-1)}$ cannot be represented in either notation using n bits. Note also that there is only one representation for the number 0 using n bits in twos complement notation. To see this, consider 0 using eight bits: 00000000. The ones complement is 11111111, which is negative 0 in that notation. Adding 1 to produce the twos complement form yields 100000000, which is nine bits long. Since only eight bits are allowed, the leftmost bit (or "overflow") is discarded, leaving 00000000 as minus 0.

The binary number system is by no means the only method by which bits can be used to represent integers. For example, a string of bits may be used to represent integers in the decimal number system, as follows. Four bits can be used to represent a decimal digit between 0 and 9 in the binary notation described previously. A string of bits of arbitrary length may be divided into consecutive sets of four bits, with each set representing a decimal digit. The string then represents the number that is formed by those decimal digits in conventional decimal notation. For example, in this system the bit string 00100110 is separated into two strings of four bits each: 0010 and 0110. The first of these represents the decimal digit 2 and the second represents the decimal digit 6, so that the entire string represents the integer 26. This representation is called ***binary coded decimal***.

One important feature of the binary coded decimal representation of nonnegative integers is that not all bit strings are valid representations of a decimal integer. Four bits can be used to represent one of sixteen different possibilities, since there are sixteen possible states for a set of four bits. However, in the binary coded decimal integer representation, only ten of those sixteen possibilities are used. That is, codes such as 1010 and 1100, whose binary values are 10 or larger, are invalid in a binary coded decimal number.

Real Numbers

The usual method used by computers to represent real numbers is ***floating-point notation***. There are many varieties of floating-point notation and each has individual characteristics. The key concept is that a real number is represented by a number, called a ***mantissa***, times a ***base*** raised to an integer power, called an ***exponent***. The base is usually fixed, and the mantissa and exponent vary to represent different real numbers. For example, if the base is fixed at 10, the number 387.53 could be represented as 38753×10^{-2}. (Recall that 10^{-2} is .01.) The mantissa is 38753, and the exponent is -2. Other possible representations are $.38753 \times 10^{3}$ and 387.53×10^{0}. We choose the representation in which the mantissa is an integer with no trailing 0s.

In the floating-point notation that we describe (which is not necessarily implemented on any particular machine exactly as described), a real number is represented by a 32-bit string consisting of a 24-bit mantissa followed by an 8-bit exponent. The base is fixed at 10. Both the mantissa and the exponent are twos complement binary integers. For example, the 24-bit binary representation of 38753 is 000000001001011101100001, and the 8-bit twos complement binary representation of −2 is 11111110; the representation of 387.53 is 00000000100101110110000111111110. Other real numbers and their floating-point representations are as follows:

0	00000000000000000000000000000000
100	00000000000000000000000100000010
.5	00000000000000000000010111111111
000005	00000000000000000000010111111010
12000	00000000000000000000011000000011
−387.53	11111111011010001001111111111110
−12000	11111111111111111111010000000011

The advantage of floating-point notation is that it can be used to represent numbers with extremely large or extremely small absolute values. For example, in the notation presented previously, the largest number that can be represented is $(2^{23-1}) \times 10^{127}$, which is a very large number indeed. The smallest positive number that can be represented is 10^{-128}, which is quite small. The limiting factor on the precision to which numbers can be represented on a particular machine is the number of significant binary digits in the mantissa. Not every number between the largest and the smallest can be represented. Our representation allows only 23 significant bits. Thus, a number such as 10 million and 1, which requires 24 significant binary digits in the mantissa, would have to be approximated by 10 million (1×10^7), which only requires one significant digit.

Character Strings

As we all know, information is not always interpreted numerically. Items such as names, job titles, and addresses must also be represented in some fashion within a computer. To enable the representation of such nonnumeric objects, still another method of interpreting bit strings is necessary. Such information is usually represented in character string form. For example, in some computers, the eight bits 00100110 are used to represent the character '&'. A different eight-bit pattern is used to represent the character 'A', another to represent 'B', another to represent 'C', and still another for each character that has a representation in a particular machine. A Russian machine uses bit patterns to represent Russian characters, whereas an Israeli machine uses bit patterns to represent Hebrew characters. (In fact, the characters being used are transparent to the machine; the character set can be changed by using a different font set on the printer.)

If eight bits are used to represent a character, up to 256 different characters can be represented, since there are 256 different eight-bit patterns. If the string 11000000 is

used to represent the character 'A' and 11000001 is used to represent the character 'B', the character string "AB" would be represented by the bit string 1100000011000001. In general, a character string (STR) is represented by the concatenation of the bit strings that represent the individual characters of the string.

As in the case of integers, there is nothing about a particular bit string that makes it intrinsically suitable for representing a specific character. The assignment of bit strings to characters may be entirely arbitrary, but it must be adhered to consistently. It may be that some convenient rule is used in assigning bit strings to characters. For example, two bit strings may be assigned to two letters so that the one with a smaller binary value is assigned to the letter that comes earlier in the alphabet. However, such a rule is merely a convenience; it is not mandated by any intrinsic relation between characters and bit strings. In fact, computers even differ over the number of bits used to represent a character. Some computers use seven bits (and therefore allow only up to 128 possible characters), some use eight (up to 256 characters), and some use ten (up to 1024 possible characters). The number of bits necessary to represent a character in a particular computer is called the ***byte size*** and a group of bits of that number is called a ***byte***.

Note that using eight bits to represent a character means that 256 possible characters can be represented. It is not very often that one finds a computer that uses so many different characters (although it is conceivable for a computer to include upper- and lowercase letters, special characters, italics, boldface, and other type characters), so that many of the eight-bit codes are not used to represent characters.

Thus we see that information itself has no meaning. Any meaning can be assigned to a particular bit pattern, as long as it is done consistently. It is the interpretation of a bit pattern that gives it meaning. For example, the bit string 00100110 can be interpreted as the number 38 (binary), the number 26 (binary coded decimal), or the character '&'. A method of interpreting a bit pattern is often called a ***data type***. We have presented several data types: binary integers, binary coded decimal nonnegative integers, real numbers, and character strings. The key questions are how to determine what data types are available to interpret bit patterns and what data type to use in interpreting a particular bit pattern.

Hardware and Software

The ***memory*** (also called ***storage*** or ***core***) of a computer is simply a group of bits (switches). At any instant of the computer's operation any particular bit in memory is either 0 or 1 (off or on). The setting of a bit is called its ***value*** or its ***contents***.

The bits in a computer memory are grouped together into larger units such as bytes. In some computers, several bytes are grouped together into units called ***words***. Each such unit (byte or word, depending on the machine) is assigned an ***address***, that is, a name identifying a particular unit among all the units in memory. This address is usually numeric, so that we may speak of byte 746 or word 937. An address is often called a ***location***, and the contents of a location are the values of the bits that make up the unit at that location.

Every computer has a set of "native" data types. This means that it is constructed with a mechanism for manipulating bit patterns consistent with the objects they represent. For example, suppose that a computer contains an instruction to add two binary

integers and place their sum at a given location in memory for subsequent use. Then there must be a mechanism built into the computer to

1. Extract operand bit patterns from two given locations.
2. Produce a third bit pattern representing the binary integer that is the sum of the two binary integers represented by the two operands.
3. Store the resultant bit pattern at a given location.

The computer "knows" to interpret the bit patterns at the given locations as binary integers because the hardware that executes that particular instruction is designed to do so. This is akin to a light "knowing" to be on when the switch is in a particular position.

If the same machine also has an instruction to add two real numbers, there must be a separate built-in mechanism to interpret operands as real numbers. Two distinct instructions are necessary for the two operations, and each instruction carries within itself an implicit identification of the types of its operands as well as their explicit locations. Therefore it is the programmer's responsibility to know which data type is contained in each location that is used. It is the programmer's responsibility to choose between using an integer or real addition instruction to obtain the sum of two numbers.

A high-level programming language aids in this task considerably. For example, if a C programmer declares

```
int x, y;
float a, b;
```

space is reserved at four locations for four different numbers. These four locations may be referenced by the ***identifiers*** *x*, *y*, *a*, and *b*. An identifier is used instead of a numerical address to refer to a particular memory location because of its convenience for the programmer. The contents of the locations reserved for *x* and *y* will be interpreted as integers, whereas the contents of *a* and *b* will be interpreted as floating-point numbers. The compiler that is responsible for translating C programs into machine language will translate the "+" in the statement

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

into integer addition, and will translate the "+" in the statement

```
a = a + b;
```

into floating-point addition. An operator such as "+" is really a ***generic*** operator because it has several different meanings depending on its context. The compiler relieves the programmer of specifying the type of addition that must be performed by examining the context and using the appropriate version.

It is important to recognize the key role played by declarations in a high-level language. It is by means of declarations that the programmer specifies how the contents of the computer memory are to be interpreted by the program. In doing this, a declaration specifies how much memory is needed for a particular entity, how the contents of that

memory are to be interpreted, and other vital details. Declarations also specify to the compiler exactly what is meant by the operation symbols that are subsequently used.

Concept of Implementation

Thus far we have been viewing data types as a method of interpreting the memory contents of a computer. The set of native data types that a particular computer can support is determined by what functions have been wired into its hardware. However, we can view the concept of "data type" from a completely different perspective; not in terms of what a computer can do, but in terms of what the user wants done. For example, if one wishes to obtain the sum of two integers, one does not care very much about the detailed mechanism by which that sum will be obtained. One is interested in manipulating the mathematical concept of an "integer," not in manipulating hardware bits. The hardware of the computer may be used to represent an integer, and is useful only insofar as the representation is successful.

Once the concept of "data type" is divorced from the hardware capabilities of the computer, a limitless number of data types can be considered. A data type is an abstract concept defined by a set of logical properties. Once such an abstract data type is defined and the legal operations involving that type are specified, we may ***implement*** that data type (or a close approximation to it). An implementation may be a ***hardware implementation,*** in which the circuitry necessary to perform the required operations is designed and constructed as part of a computer; or it may be a ***software implementation,*** in which a program consisting of already existing hardware instructions is written to interpret bit strings in the desired fashion and to perform the required operations. Thus, a software implementation includes a specification of how an object of the new data type is represented by objects of previously existing data types, as well as a specification of how such an object is manipulated in conformance with the operations defined for it. Throughout the remainder of this text, the term "implementation" is used to mean "software implementation."

Example

We illustrate these concepts with an example. Suppose that the hardware of a computer contains an instruction

```
MOVE (source,dest,length)
```

that copies a character string of *length* bytes from an address specified by *source* to an address specified by *dest*. (We present hardware instructions using uppercase letters. The length must be specified by an integer, and for that reason we indicate it with lowercase letters. *source* and *dest* can be specified by identifiers that represent storage locations.) An example of this instruction is MOVE(a,b,3), which copies the three bytes starting at location a to the three bytes starting at location b.

Note the different roles played by the identifiers a and b in this operation. The first operand of the MOVE instruction is the contents of the location specified by the identifier a. The second operand, however, is not the contents of location b, since these

contents are irrelevant to the execution of the instruction. Rather, the location itself is the operand, since the location specifies the destination of the character string. Although an identifier always stands for a location, it is common for an identifier to be used to reference the contents of that location. It is always apparent from the context whether an identifier is referencing a location or its contents. The identifier appearing as the first operand of a MOVE instruction refers to the contents of memory, whereas the identifier appearing as the second operand refers to a location.

We also assume the computer hardware to contain the usual arithmetic and branching instructions, which we indicate by using C-like notation. For example, the instruction

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

interprets the contents of the bytes at locations *x* and *y* as binary integers, adds them, and inserts the binary representation of their sum into the byte at location *z*. (We do not operate on integers greater than one byte in length and ignore the possibility of overflow.) Here again, *x* and *y* are used to reference memory contents, whereas *z* is used to reference a memory location, but the proper interpretation is clear from the context.

Sometimes it is desirable to add a quantity to an address to obtain another address. For example, if *a* is a location in memory, we might want to reference the location four bytes beyond *a*. We cannot refer to this location as a+4, since that notation is reserved for the integer contents of location a+4. We therefore introduce the notation *a*[4] to refer to this location. We also introduce the notation *a*[*x*] to refer to the address given by adding the binary integer contents of the byte at *x* to the address *a*.

The MOVE instruction requires the programmer to specify the length of the string to be copied. Thus, its operand is a fixed-length character string (that is, the length of the string must be known). A fixed-length string and a byte-sized binary integer may be considered native data types of this particular machine.

Suppose that we wished to implement varying-length character strings on this machine. That is, we want to enable programmers to use an instruction

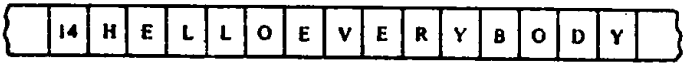
```
MOVEVAR(source,dest)
```

to move a character string from location *source* to location *dest* without being required to specify any length.

To implement this new data type, we must first decide on how it is to be represented in the memory of the machine and then indicate how that representation is to be manipulated. Clearly, it is necessary to know how many bytes must be moved to execute this instruction. Since the MOVEVAR operation does not specify this number, the number must be contained within the representation of the character string itself. A varying-length character string of length *l* may be represented by a contiguous set of *l* + 1 bytes (*l* < 256). The first byte contains the binary representation of the length *l* and the remaining bytes contain the representations of the characters in the string. Representations of three such strings are illustrated in Figure 1.1.2. (Note that the digits 5 and 9 in these figures do not stand for the bit patterns representing the characters '5'

(a)

(b)



(c)

Figure 1.1.2 Varying-length character strings.

and '9' but rather for the patterns 00000101 and 00001001 (assuming eight bits to a byte), which represent the integers five and nine. Similarly, 14 in Figure 1.1.2c stands for the bit pattern 00001110. Note also that this representation is very different from the way character strings are actually implemented in C.)

The program to implement the MOVEVAR operation can be written as follows (*i* is an auxiliary memory location):

```
MOVE(source, dest, 1);
for (i=1; i < dest; i++)
   MOVE(source[i], dest[i], 1);
```

Similarly, we can implement an operation CONCATVAR($c1$,$c2$,$c3$) to concatenate two varying-length character strings at locations $c1$ and $c2$ and place the result at $c3$. Figure 1.1.2c illustrates the concatenation of the two strings in Figure 1.1.2a and b:

```
/*     move the length      */
z = c1 + c2;
MOVE(z, c3, 1);
/*  move the first string   */
for (i = 1; i <= c1; MOVE(c1[i], c3[i], 1);
/*  move the second string  */
for (i = 1; i <= c2) {
   x = c1 + i;
   MOVE(c2[i], c3[x], 1);
} /* end for */
```

However, once the operation MOVEVAR has been defined, CONCATVAR can be implemented using MOVEVAR as follows:

```
MOVEVAR(c2, c3[c1]); /*     move the second string      */
MOVEVAR(c1, c3);     /*     move the first string       */
z = c1 + c2;         /* update the length of the result */
MOVE(z, c3, 1);
```

Figure 1.1.3 illustrates phases of this operation on the strings of Figure 1.1.2. Although this latter version is shorter, it is not really more efficient, since all the instructions used in implementing MOVEVAR are performed each time that MOVEVAR is used.

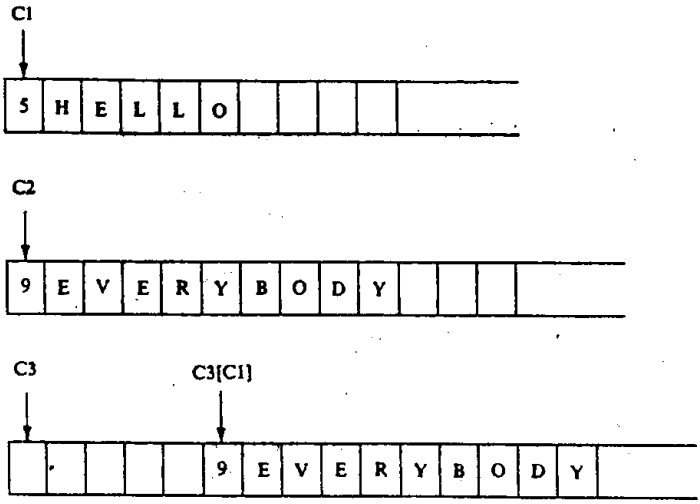
The statement $z = c1 + c2$ in both the preceding algorithms is of particular interest. The addition instruction operates independently of the use of its operands (in this case, parts of varying-length character strings). The instruction is designed to treat its operands as single-byte integers regardless of any other use that the programmer has for them. Similarly, the reference to $c3[c1]$ is to the location whose address is given by adding the contents of the byte at location $c1$ to the address $c3$. Thus the byte at $c1$ is treated as holding a binary integer, although it is also the start of a varying-length character string. This illustrates the fact that a data type is a method of treating the contents of memory and that those contents have no intrinsic meaning.

Note that this representation of varying-length character strings allows only strings whose length is less than or equal to the largest binary integer that fits into a single byte. If a byte is eight bits, this means that the largest such string is 255 (that is, $2^8 - 1$) characters long. To allow for longer strings, a different representation must be chosen and a new set of programs must be written. If we use this representation of varying-length character strings, the concatenation operation is invalid if the resulting string is more than 255 characters long. Since the result of such an operation is undefined, a wide variety of actions can be implemented if that operation is attempted. One possibility is to use only the first 255 characters of the result. Another possibility is to ignore the operation entirely and not move anything to the result field. There is also a choice of printing a warning message or of assuming that the user wants to achieve whatever result the implementor decides on.

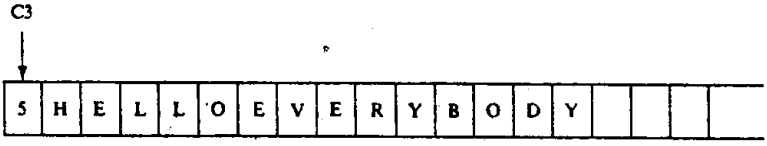
In fact, C uses an entirely different implementation of character strings that avoids this limitation on the length of the string. In C, all strings are terminated by the special character '\0'. This character, which never appears within a string, is automatically placed by the compiler at the end of every string. Since the length of the string is not known in advance, all string operations must proceed a character at a time until '\0' is encountered.

The program to implement the MOVEVAR operation, under this implementation, can be written as follows:

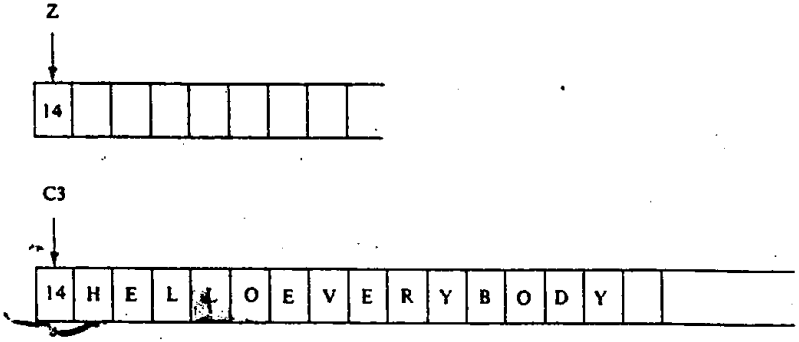
```
i = 0;
while (source[i] != '\0') {
   MOVE(source[i], dest[i], 1);
   i++;
}
dest[i] = '\0';
/* terminate the destination string with '\0' */
```



(a) MOVEVAR (C2, C3[C1]);



(b) MOVEVAR (C1, C3);



(c) Z = C1 + C2; MOVE (Z, C3, 1);

Figure 1.1.3 CONCATVAR operations.

To implement the concatenation operation, CONCATVAR(*c*1,*c*2,*c*3), we may write

```
i = 0;
/* move the first string */
while (c1[i] != '\0') {
   MOVE(c1[i], c3[i], 1);
   i++;
}
/* move the second string */
j = 0;
while (c2[j] != '\0')
   MOVE(c2[j++], c3[i++], 1);
/* terminate the destination string with a \0 */
c3[i] = '\0';
```

A disadvantage of the C implementation of character strings is that the length of a character string is not readily available without advancing through the string one character at a time until '\0' is encountered. This is more than offset by the advantage of not having an arbitrary limit placed on the length of the string.

Once a representation has been chosen for objects of a particular data type and routines have been written to operate on those representations, the programmer is free to use that data type to solve problems. The original hardware of the machine plus the programs for implementing more complex data types than those provided by the hardware can be thought of as a "better" machine than the one consisting of the hardware alone. The programmer of the original machine need not worry about how the computer is designed and what circuitry is used to execute each instruction. The programmer need know only what instructions are available and how those instructions can be used. Similarly, the programmer who uses the "extended" machine (that consists of hardware and software), or "virtual computer," as it is sometimes known, need not be concerned with the details of how various data types are implemented. All the programmer needs to know is how they can be manipulated.

Abstract Data Types

A useful tool for specifying the logical properties of a data type is the ***abstract data type***, or ***ADT***. Fundamentally, a data type is a collection of values and a set of operations on those values. That collection and those operations form a mathematical construct that may be implemented using a particular hardware or software data structure. The term "abstract data type" refers to the basic mathematical concept that defines the data type.

In defining an abstract data type as a mathematical concept, we are not concerned with space or time efficiency. Those are implementation issues. In fact, the definition of an ADT is not concerned with implementation details at all. It may not even be possible to implement a particular ADT on a particular piece of hardware or using a particular software system. For example, we have already seen that the ADT *integer* is not universally implementable. Nevertheless, by specifying the mathematical and logical properties of a data type or structure, the ADT is a useful guideline to implementors and a useful tool to programmers who wish to use the data type correctly.

There are a number of methods for specifying an ADT. The method that we use is semiformal and borrows heavily from C notation but extends that notation where necessary. To illustrate the concept of an ADT and our specification method, consider the ADT *RATIONAL*, which corresponds to the mathematical concept of a rational number. A rational number is a number that can be expressed as the quotient of two integers. The operations on rational numbers that we define are the creation of a rational number from two integers, addition, multiplication, and testing for equality. The following is an initial specification of this ADT:

```
     /*value definition*/
abstract typedef <integer, integer> RATIONAL;
condition RATIONAL[1] != 0;

     /*operator definition*/
abstract RATIONAL makerational(a,b)
int a,b;
precondition  b != 0;
postcondition makerational[0] == a;
              makerational[1] == b;

abstract RATIONAL add(a,b)                    /* written a + b */
RATIONAL a,b;
postcondition add[1] == a[1] * b[1];
              add[0] == a[0] * b[1] + b[0] * a[1];

abstract RATIONAL mult(a,b)                   /* written a * b */
RATIONAL a,b;
postcondition mult[0] == a[0] * b[0];
              mult[1] == a[1] * b[1];

abstract equal(a,b)                           /* written a == b */
RATIONAL a,b;
postcondition equal == (a[0]*b[1] == b[0]*a[1]);
```

An ADT consists of two parts: a value definition and an operator definition. The value definition defines the collection of values for the ADT and consists of two parts: a definition clause and a condition clause. For example, the value definition for the ADT *RATIONAL* states that a *RATIONAL* value consists of two integers, the second of which does not equal 0. Of course, the two integers that make up a rational number are the numerator and the denominator. We use array notation (square brackets) to indicate the parts of an abstract type.

The keywords ***abstract typedef*** introduce a value definition, and the keyword ***condition*** is used to specify any conditions on the newly defined type. In this definition, the condition specifies that the denominator may not be 0. The definition clause is required, but the condition clause may not be necessary for every ADT.

Immediately following the value definition comes the operator definition. Each operator is defined as an abstract function with three parts: a header, the optional pre-

conditions, and the postconditions. For example, the operator definition of the ADT *RATIONAL* includes the operations of creation (*makerational*), addition (*add*) and multiplication (*mult*), as well as a test for equality (*equal*). Let us consider the specification for multiplication first, since it is the simplest. It contains a header and postconditions, but no preconditions:

```
abstract RATIONAL  mult(a,b)                /* written a*b */
RATIONAL a,b;
postcondition mult[0] == a[0]*b[0];
              mult[1] == a[1]*b[1];
```

The header of this definition is the first two lines, which are just like a C function header. The keyword ***abstract*** indicates that this is not a C function but an ADT operator definition. The comment beginning with the new keyword ***written*** indicates an alternative way of writing the function.

The postcondition specifies what the operation does. In a postcondition, the name of the function (in this case, *mult*) is used to denote the result of the operation. Thus, *mult*[0] represents the numerator of the result, and *mult*[1] the denominator of the result. That is, it specifies what conditions become true after the operation is executed. In this example, the postcondition specifies that the numerator of the result of a rational multiplication equals the integer product of the numerators of the two inputs, and that the denominator equals the integer products of the two denominators.

The specification for addition (*add*) is straightforward and simply states that

$$\frac{a0}{a1} + \frac{b0}{b1} = \frac{a0 * b1 + a1 * b0}{a1 * b1}$$

The creation operation (*makerational*) creates a rational number from two integers and contains the first example of a precondition. In general, preconditions specify any restrictions that must be satisfied before the operation can be applied. In this example, the precondition states that *makerational* cannot be applied if its second parameter is 0.

The specification for equality (*equal*) is more significant and more complex in concept. In general, any two values in an ADT are "equal" if and only if the values of their components are equal. Indeed, it is usually assumed that an equality (and an inequality) operation exists and is defined that way, so that no explicit equal operator definition is required. The assignment operation (setting the value of one object to the value of another) is another example of an operation that is often assumed for an ADT and is not specified explicitly.

However, for some data types, two values with unequal components may be considered equal. Indeed, such is the case with rational numbers: for example, the rational numbers 1/2, 2/4, 3/6, and 18/36 are all equal despite the inequality of their components. Two rational numbers are considered equal if their components are equal when the numbers are reduced to lowest terms (that is, when their numerators and denominators are both divided by their greatest common divisor). One way of testing for rational equality is to reduce the two numbers to lowest terms and then test for equality of numerators and denominators. Another way of testing for rational equality is to check whether the

cross products (that is, the numerator of one times the denominator of the other) are equal. This is the method that we used in specifying the abstract *equal* operation.

The abstract specification illustrates the role of an ADT as a purely logical definition of a new data type. As collections of two integers, two ordered pairs are unequal if their components are not equal; yet as rational numbers, they may be equal. It is unlikely that any implementation of rational numbers would implement a test for equality by actually forming the cross products; they might be too large to represent as machine integers. Most likely, an implementation would first reduce the inputs to lowest terms and then test for component equality. Indeed, a reasonable implementation would insist that *makerational*, *add*, and *mult* only produce rational numbers in lowest terms. However, mathematical definitions such as abstract data type specifications need not be concerned with implementation details.

In fact, the realization that two rationals can be equal even if they are componentwise unequal forces us to rewrite the postconditions for *makerational*, *add*, and *mult*. That is, if

$$\frac{m0}{m1} = = \frac{a0}{a1} * \frac{b0}{b1}$$

it is not necessary that $m0$ equal $a0 * b0$ and that $m1$ equal $a1 * b1$, only that $m0 * a1 * b1$ equal $m1 * a0 * b0$. A more accurate ADT specification for RATIONAL is the following:

```
/*value definition*/
abstract typedef<int, int> RATIONAL;
condition RATIONAL[1] != 0;

 /*operator definition*/
abstract equal(a,b)                          /* written a == b*/
RATIONAL a,b;
postcondition equal == (a[0]*b[1] == b[0]*a[1]);

abstract RATIONAL makerational(a,b)      /* written [a,b]*/
int a,b;
precondition  b != 0;
postcondition makerational[0]*b == a*makerational[1]

abstract RATIONAL add(a,b)   /* written a + b */
RATIONAL a,b;
postcondition add == [a[0] * b[1] + b[0] * a[1], a[1]*b[1]]

abstract RATIONAL mult(a,b) /* written a * b */
RATIONAL a,b;
postcondition mult == [a[0] * b[0], a[1] * b[1] ]
```

Here, the *equal* operator is defined first, and the operator = = is extended to rational equality using the *written* clause. That operator is then used to specify the results of subsequent rational operations (*add* and *mult*).

The result of the *makerational* operation on the integers *a* and *b* produces a rational that equals *a/b*, but the definition does not specify the actual values of the resulting numerator and denominator. The specification for *makerational* also introduces the notation [a,b] for the rational formed from integers *a* and *b*, and this notation is then used in defining *add* and *mult*.

The definitions of *add* and *mult* specify that their results equal the unreduced results of the corresponding operation, but the individual components are not necessarily equal.

Notice, again, that in defining these operators we are not specifying how they are to be computed, only what their result must be. How they are computed is determined by their implementation, not by their specification.

Sequences as Value Definitions

In developing the specifications for various data types, we often use set-theoretic notation to specify the values of an ADT. In particular, it is helpful to use the notation of mathematical sequences that we now introduce.

A ***sequence*** is simply an ordered set of elements. A sequence *S* is sometimes written as the enumeration of its elements, such as

$$S = < s_0, s_1, \ldots, s_{n-1} >$$

If *S* contains *n* elements, *S* is said to be of length *n*. We assume the existence of a length function *len* such that *len*(*S*) is the length of the sequence *S*. We also assume functions *first*(*S*), which returns the value of the first element of *S* (s_0 in the foregoing example), and *last*(*S*), which returns the value of the last element of *S* (s_{n-1} in the foregoing example). There is a special sequence of length 0, called *nilseq*, that contains no elements. *first*(*nilseq*) and *last*(*nilseq*) are undefined.

We wish to define an ADT *stp*1 whose values are sequences of elements. If the sequences can be of arbitrary length and consist of elements all of which are of the same type, *tp*, then *stp*1 can be defined by

```
abstract typedef <<tp>> stp1;
```

Alternatively, we may wish to define an ADT *stp*2, whose values are sequences of fixed length whose elements are of specific types. In such a case, we would specify the definition

```
abstract typedef <tp0, tp1, tp2, ..., tpn> stp2;
```

Of course, we may want to specify a sequence of fixed length all of whose elements are of the same type. We could then write

```
abstract typedef <<tp,n>> stp3;
```

In this case *stp*3 represents a sequence of length *n*, all of whose elements are of type *tp*.

For example, using the foregoing notation we could define the following types:

```
abstract typedef <<int>> intseq;
                          /* sequence of integers of   */
                          /*       any length          */
abstract typedef <integer, char, float> seq3;
                          /*   sequence of length 3    */
                          /* consisting of an integer, */
                          /*  a  character and a       */
                          /*  floating-point number    */

abstract typedef <<int,10>> intseq;
                          /*  sequence of 10 integers  */

abstract typedef <<,2>> pair;
                          /*   arbitrary sequence of   */
                          /*          length 2         */
```

Two sequences are *equal* if each element of the first is equal to the corresponding element of the second. A ***subsequence*** is a contiguous portion of a sequence. If S is a sequence, the function $sub(S,i,j)$ refers to the subsequence of S starting at position i in S and consisting of j consecutive elements. Thus if T equals $sub(S,i,k)$, and T is the sequence $< t_0, t_1, \ldots, t_{k-1} >$, $t_0 = s_i$, $t_1 = s_{i+1}$, $\ldots$, $t_{k-1} = s_{i+k-1}$. If i is not between 0 and $len(S) - k$, then $sub(S,i,k)$ is defined as *nilseq*.

The concatenation of two sequences, written $S + T$, is the sequence consisting of all the elements of S followed by all the elements of T. It is sometimes desirable to specify insertion of an element in the middle of a sequence. $place(S,i,x)$ is defined as the sequence S with the element x inserted immediately following position i (or into the first element of the sequence if i is -1). All subsequent elements are shifted by one position. That is, $place(S,i,x)$ equals $sub(S,0,i+1) + < x > + sub(S,i+1, len(S) - i - 1)$.

Deletion of an element from a sequence can be specified in one of two ways. If x is an element of sequence S, $S - < x >$ represents the sequence S without all occurrences of element x. The sequence $delete(S,i)$ is equal to S with the element at position i deleted. $delete(S,i)$ can also be written in terms of other operations as $sub(S,0,i) + sub(S,i+1,len(S) - i - 1)$.

ADT for Varying-length Character Strings

As an illustration of the use of sequence notation in defining an ADT, we develop an ADT specification for the varying-length character string. There are four basic operations (aside from equality and assignment) normally included in systems that support such strings:

length	a function that returns the current length of the string
concat	a function that returns the concatenation of its two input strings
substr	a function that returns a substring of a given string

pos a function that returns the first position of one string as a substring of another

```
abstract typedef <<char>> STRING;

abstract length(s)
STRING s;
postcondition  length == len(s);

abstract STRING concat(s1,s2)
STRING s1,s2;
postcondition  concat == s1 + s2;

abstract STRING substr(s1,i,j)
STRING s1;
int i,j;
precondition 0 <= i < len(s1);
             0 <= j <= len(s1) - i;
postcondition substr == sub(s1,i,j);

abstract pos(s1,s2)
STRING s1,s2;
postcondition /*lastpos = len(s1) - len(s2) */
              ((pos == -1)  && (for(i = 0;
                                    i <= lastpos; i++)
                                (s2 <> sub(s1,i,len(s2)))))
              ||
              ((pos >= 0) && (pos <= lastpos)
                          && (s2 == sub(str1,pos,len(s2))
                          && (for(i = 1; i < pos; i++)
                                   (s2 <> sub(s1,i,len(s2))))));
```

The postcondition for *pos* is complex and introduces some new notation, so we review it here. First, note the initial comment whose content has the form of a C assignment statement. This merely indicates that we wish to define the symbol *lastpos* as representing the value of $len(s1) - len(s2)$ for use within the postcondition to simplify the appearance of the condition. Here, *lastpos* represents the maximum possible value of the result (that is, the last position of $s1$ where a substring whose length equals that of $s2$ can start). *lastpos* is used twice within the postcondition itself. The longer expression $len(s1) - len(s2)$ could have been used in both cases, but we chose to use a more compact symbol (*lastpos*) for clarity.

The postcondition itself states that one of two conditions must hold. The two conditions, which are separated by the || operator, are as follows:

1. The function's value (*pos*) is -1, and $s2$ does not appear as a substring of $s1$.
2. The function's value is between 0 and *lastpos*, $s2$ does appear as a substring of $s1$ beginning at the function value's position, and $s2$ does not appear as a substring of $s1$ in any earlier position.

Note the use of a pseudo-*for* loop in a condition. The condition

```
for (i = x; i <= y; i++)
    (condition(i))
```

is true if *condition*(*i*) is true for all i from x to y inclusive. It is also true if $x > y$. Otherwise, the entire *for*-condition is false.

Data Types in C

The C language contains four basic data types: *int*, *float*, *char* and *double*. In most computers, these four types are native to the machine's hardware. We have already seen how integers, floats, and characters can be implemented in hardware. A *double* variable is a double-precision floating-point number. There are three qualifiers that can be applied to *ints*: *short*, *long*, and *unsigned*. A *short* or *long* integer variable refers to the maximum size of the variable's value. The actual maximum sizes implied by *short int*, *long int*, or *int* vary from machine to machine. An *unsigned* integer is an integer that is always positive and follows the arithmetic laws of modulo 2^n, where n is the number of bits in an integer.

A variable declaration in C specifies two things. First, it specifies the amount of storage that must be set aside for objects declared with that type. For example, a variable of type *int* must have enough space to hold the largest possible integer value. Second, it specifies how data represented by strings of bits are to be interpreted. The same bits at a specific storage location can be interpreted as an integer of a floating-point number, yielding two completely different numeric values.

A variable declaration specifies that storage be set aside for an object of the specified type and that the object at that storage location can be referenced with the specified variable identifier.

Pointers in C

In fact, C allows the programmer to reference the location of objects as well as the objects (that is, the contents of those locations) themselves. For example, if x is declared as an integer, *&x* refers to the location that has been set aside to contain x. *&x* is called a ***pointer***.

It is possible to declare a variable whose data type is a pointer and whose possible values are memory locations. For example, the declarations

```
int *pi;
float *pf;
char *pc;
```

declare three pointer variables: *pi* is a pointer to an integer, *pf* is a pointer to a float number, and *pc* is a pointer to a character. The asterisk indicates that the values of the

variables being declared are pointers to values of the type specified in the declaration rather than objects of that type.

A pointer is like any other data type in C in many respects. The value of a pointer is a memory location in the way that the value of an integer is a number. Pointer values can be assigned like any other values. For example, the statement *pi* = &*x*; assigns a pointer to the integer *x* to the pointer variable *pi*.

The notation **pi* in C refers to the integer at the location referenced by the pointer *pi*. The statement *x* = **pi*; assigns the value of that integer to the integer variable *x*.

Note that C insists that a declaration of a pointer specify the data type to which the pointer points. In the foregoing declarations, each of the variables *pi*, *pf*, and *pc* are pointers to a specific data type: *int*, *float*, and *char*, respectively. The type of *pi* is not simply "pointer" but "pointer to an integer." In fact, the types of *pi* and *pf* are different: *pi* is a pointer to an integer, and *pf* is a pointer to a float number. Each data type *dt* in C generates another data type, *pdt*, called "pointer to *dt*." We call *dt* the ***base type*** of *pdt*.

The conversion of *pf* from the type "pointer to a float number" to the type "pointer to an integer" can be made by writing

```
pi = (int *) pf;
```

where the cast (***int** *) converts the value of *pf* to the type "pointer to an ***int***," or "***int** *."

The importance of each pointer being associated with a particular base type becomes clear in reviewing the arithmetic facilities that C provides for pointers. If *pi* is a pointer to an integer, then *pi* + 1 is the pointer to the integer immediately following the integer **pi* in memory, *pi* − 1 is the pointer to the integer immediately preceding **pi*, *pi* + 2 is the pointer to the second integer following **pi*, and so on. For example, suppose that a particular machine uses byte addressing, an integer requires four bytes, and the value of *pi* happens to be 100 (that is, *pi* points to the integer **pi* at location 100). Then the value of *pi* − 1 is 96, the value of *pi* + 1 is 104 and the value of *pi* + 2 is 108. The value of *(*pi* − 1) is the contents of the four bytes 96, 97, 98, and 99 interpreted as an integer; the value of *(*pi* + 1) is the contents of bytes 104, 105, 106, and 107 interpreted as an integer; and the value of *(*pi* + 2) is the integer at bytes 108, 109, 110, and 111.

Similarly, if the value of the variable *pc* is 100 (recall that *pc* is a pointer to a character) and a character is one byte long, *pc* − 1 refers to location 99, *pc* + 1 to location 101, and *pc* + 2 to location 102. Thus the result of pointer arithmetic in C depends on the base type of the pointer.

Note also the difference between **pi* + 1, which refers to 1 added to the integer **pi*, and *(*pi* + 1), which refers to the integer following the integer at location *pi*.

One area in which C pointers play a prominent role is in passing parameters to functions. Ordinarily, parameters are passed to a C function *by value*, that is, the values being passed are copied into the parameters of the called function at the time the function is invoked. If the value of a parameter is changed within the function, the value in the calling program is not changed. For example, consider the following program segment and function (the line numbers are for reference only):

```
1  x = 5;
2  printf("%d\n", x);
3  funct(x);
4  printf("%d\n", x);
      ...
5  void funct(int y)
6  {
7     ++y;
8     printf("%d\n", y);
9  } /* end funct */
```

Line 2 prints 5 and then line 3 invokes *funct*. The value of x, which is 5, is copied into y and *funct* begins execution. Line 8 then prints 6 and *funct* returns. However, when line 7 increments the value of y, the value of x remains unchanged. Thus line 4 prints 5. x and y refer to two different variables that happen to have the same value at the beginning of *funct*. y can change independently of x.

If we wish to use *funct* to modify the value of x, we must pass the address of x as follows:

```
1  x = 5;
2  printf("%d\n", x);
3  funct(&x);
4  printf("%d\n", x);
      ...
5  void funct(int *py)
6  {
7     ++(*py);
8     printf("%d\n", *py);
9  } /* end funct */
```

Line 2 again prints 5 and line 3 invokes *funct*. Now, however, the value passed is not the integer value of x, but the pointer value $\&x$. This is the address of x. The parameter of *funct* is no longer y of type **int** but py of type **int** *. (It is convenient to name pointer variables beginning with the letter p as a reminder to both the programmer and the program reader that it is a pointer. However, this is not a requirement of the C language and we could have named the pointer parameter y.) Line 7 now increments the integer at location py. py, itself, however, is not changed and retains its initial value $\&x$. Thus py points to the integer x, so that when $*py$ is incremented, x is incremented. Line 8 prints 6 and when *funct* returns, line 4 also prints 6. Pointers are the mechanism used in C to allow a called function to modify variables in a calling function.

Data Structures and C

A C programmer can think of the C language as defining a new machine with its own capabilities, data types, and operations. The user can state a problem solution in terms of the more useful C constructs rather than in terms of lower-level machine-

language constructs. Thus, problems can be solved more easily because a larger set of tools is available.

The study of data structures therefore involves two complementary goals. The first goal is to identify and develop useful mathematical entities and operations and to determine what classes of problems can be solved by using these entities and operations. The second goal is to determine representations for those abstract entities and to implement the abstract operations on these concrete representations. The first of these goals views a high-level data type as a tool that can be used to solve other problems, and the second views the implementation of such a data type as a problem to be solved using already existing data types. In determining representations for abstract entities, we must be careful to specify what facilities are available for constructing such representations. For example, it must be stated whether the full C language is available or whether we are restricted to the hardware facilities of a particular machine.

In Sections 1.2 and 1.3 we examine several data structures that already exist in C: the array and the structure. We describe the facilities that are available in C for utilizing these structures. We also focus on the abstract definitions of these data structures and how they can be useful in problem solving. Finally, we examine how they could be implemented if C were not available (although a C programmer can simply use the data structures as defined in the language without being concerned with most of these implementation details).

In the remainder of the book, we develop more complex data structures and show their usefulness in problem solving. We also show how to implement these data structures using the data structures that are already available in C. Since the problems that arise in the course of attempting to implement high-level data structures are quite complex, this will also allow us to investigate the C language more thoroughly and to gain valuable experience in the use of this language.

Often no implementation, hardware or software, can model a mathematical concept completely. For example, it is impossible to represent arbitrarily large integers on a computer, since the size of such a machine's memory is finite. Thus, it is not the data type "integer" that is represented by the hardware but rather the data type "integer between x and y," where x and y are the smallest and largest integers representable by that machine.

It is important to recognize the limitations of a particular implementation. Often it will be possible to present several implementations of the same data type, each with its own strengths and weaknesses. One particular implementation may be better than another for a specific application, and the programmer must be aware of the possible trade-offs that might be involved.

One important consideration in any implementation is its efficiency. In fact, the reason that the high-level data structures that we discuss are not built into C is the significant overhead that they would entail. There are languages of significantly higher level than C that have many of these data types already built into them, but many of them are inefficient and are therefore not in widespread use.

Efficiency is usually measured by two factors: time and space. If a particular application is heavily dependent on manipulating high-level data structures, the speed at which those manipulations can be performed will be the major determinant of

the speed of the entire application. Similarly, if a program uses a large number of such structures, an implementation that uses an inordinate amount of space to represent the data structure will be impractical. Unfortunately, there is usually a trade-off between these two efficiencies, so that an implementation that is fast uses more storage than one that is slow. The choice of implementation in such a case involves a careful evaluation of the trade-offs among the various possibilities.

EXERCISES

1.1.1. In the text, an analogy is made between the length of a line and the number of bits of information in a bit string. In what ways is this analogy inadequate?

1.1.2. Determine what hardware data types are available on the computer at your particular installation and what operations can be performed on them.

1.1.3. Prove that there are 2^n different settings for n two-way switches. Suppose that we wanted to have m settings. How many switches would be necessary?

1.1.4. Interpret the following bit settings as binary positive integers, as binary integers in twos complement, and as binary coded decimal integers. If a setting cannot be interpreted as a binary coded decimal integer, explain why.

(a)	10011001	**(d)**	01110111
(b)	1001	**(e)**	01010101
(c)	000100010001	**(f)**	100000010101

1.1.5. Write C functions *add*, *subtract*, and *multiply* that read two strings of 0s and 1s representing binary nonnegative integers, and print the string representing their sum, difference, and product, respectively.

1.1.6. Assume a ternary computer in which the basic unit of memory is a "trit" (ternary digit) rather than a bit. Such a trit can have three possible settings (0, 1, and 2) rather than just two (0 and 1). Show how nonnegative integers can be represented in ternary notation using such trits by a method analogous to binary notation using bits. Is there any nonnegative integer that can be represented using ternary notation and trits that cannot be represented using binary notation and bits? Are there any that can be represented using bits that cannot be represented using trits? Why are binary computers more common than ternary computers?

1.1.7. Write a C program to read a string of 0s and 1s representing a positive integer in binary and print a string of 0s, 1s, and 2s representing the same number in ternary notation (see the preceding exercise). Write another C program to read a ternary number and print the equivalent in binary.

1.1.8. Write an ADT specification for complex numbers $a + bi$, where $abs(a + bi)$ is $sqrt(a^2 + b^2)$, $(a + bi) + (c + di)$ is $(a + c) + (b + d)i$, $(a + bi) * (c + di)$ is $(a * c - b * d) + (a * d + b * c)i$, and $-(a + bi)$ is $(-a) + (-b)i$.

1.2 ARRAYS IN C

In this section and the next we examine several data structures that are an invaluable part of the C language. We will see how to use these structures and how they can be

implemented. These structures are ***composite*** or ***structured*** data types; that is, they are made up of simpler data structures that exist in the language. The study of these data structures involves an analysis of how simple structures combine to form the composite and how to extract a specific component from the composite. We expect that you have already seen these data structures in an introductory C programming course and that you are aware of how they are defined and used in C. In these sections, therefore, we will not dwell on the many details associated with these structures but instead will highlight those features that are interesting from a data-structure point of view.

The first of these data types is the ***array***. The simplest form of array is a ***one-dimensional array*** that may be defined abstractly as a finite ordered set of homogeneous elements. By "finite" we mean that there is a specific number of elements in the array. This number may be large or small, but it must exist. By "ordered" we mean that the elements of the array are arranged so that there is a zeroth, first, second, third, and so forth. By "homogeneous" we mean that all the elements in the array must be of the same type. For example, an array may contain all integers or all characters but may not contain both.

However, specifying the form of a data structure does not yet completely describe the structure. We must also specify how the structure is accessed. For example, the C declaration

```
int a[100];
```

specifies an array of 100 integers. The two basic operations that access an array are ***extraction*** and ***storing***. The extraction operation is a function that accepts an array, a, and an index, i, and returns an element of the array. In C, the result of this operation is denoted by the expression $a[i]$. The storing operation accepts an array, a, an index, i, and an element, x. In C this operation is denoted by the assignment statement $a[i] = x$. The operations are defined by the rule that after the foregoing assignment statement has been executed, the value of $a[i]$ is x. Before a value has been assigned to an element of the array, its value is undefined and a reference to it in an expression is illegal.

The smallest element of an array's index is called its ***lower bound*** and in C is always 0, and the highest element is called its ***upper bound***. If *lower* is the lower bound of an array and *upper* the upper bound, the number of elements in the array, called its ***range***, is given by $upper - lower + 1$. For example, in the array, a, declared previously, the lower bound is 0, the upper bound is 99, and the range is 100.

An important feature of a C array is that neither the upper bound nor the lower bound (and hence the range as well) may be changed during a program's execution. The lower bound is always fixed at 0, and the upper bound is fixed at the time the program is written.

One very useful technique is to declare a bound as a constant identifier, so that the work required to modify the size of an array is minimized. For example, consider the following program segment to declare and initialize an array:

```
int a[100];
for(i = 0; i < 100; a[i++] = 0);
```

To change the array to a larger (or smaller) size, the constant 100 must be changed in two places: once in the declarations and once in the *for* statement. Consider the following equivalent alternative:

```
#define NUMELTS 100
int a[NUMELTS];
for(i = 0; i < NUMELTS; a[i++] = 0);
```

Now only a single change in the constant definition is needed to change the upper bound.

The Array as an ADT

We can represent an array as an abstract data type with a slight extension of the conventions and notation discussed earlier. We assume the function *type(arg)*, which returns the type of its argument, *arg*. Of course, such a function cannot exist in C, since C cannot dynamically determine the type of a variable. However, since we are not concerned here with implementation, but rather with specification, the use of such a function is permissible.

Let *ARRTYPE(ub,eltype)* denote the ADT corresponding to the C array type *eltype array[ub]*. This is our first example of a parameterized ADT, in which the precise ADT is determined by the values of one or more parameters. In this case, *ub* and *eltype* are the parameters; note that *eltype* is a type indicator, not a value. We may now view any one-dimensional array as an entity of the type *ARRTYPE*. For example, *ARRTYPE*(10,*int*) would represent the type of the array *x* in the declaration *int x*[10]. We may now view any one-dimensional array as an entity of the type *ARRTYPE*. The specification follows:

```
abstract typedef <<eltype, ub>> ARRTYPE(ub, eltype);
condition type(ub) == int;

abstract eltype extract(a,i)          /*  written a[i]    */
ARRTYPE(ub, eltype) a;
int i;
precondition  0 <= i < ub;
postcondition extract == a;

abstract store(a, i, elt)             /* written a[i] = elt */
ARRTYPE (ub, eltype) a;
int i;
eltype elt;
precondition  0 <= i < ub;
postcondition a[i] == elt;
```

The *store* operation is our first example of an operation that modifies one of its parameters; in this case the array *a*. This is indicated in the postcondition by specifying the value of the array element to which *elt* is being assigned. Unless a modified value is specified in a postcondition, we assume that all parameters retain the same value after the operation is applied in a postcondition as before. It is not necessary to specify that

such values remain unchanged. Thus, in this example, all array elements other than the one to which *elt* is assigned retain the same values.

Note that once the operation *extract* has been defined, together with its bracket notation, *a*[*i*], that notation can be used in the postcondition for the subsequent *store* operation specification. Within the postcondition of *extract*, however, subscripted sequence notation must be used, since the array bracket notation itself is being defined.

Using One-Dimensional Arrays

A one-dimensional array is used when it is necessary to keep a large number of items in memory and reference all the items in a uniform manner. Let us see how these two requirements apply to practical situations.

Suppose that we wish to read 100 integers, find their average, and determine by how much each integer deviates from that average. The following program accomplishes this:

```
#define NUMELTS 100
void main()
{
   int num[NUMELTS];          /* array of numbers                */
   int i;
   int total;                 /*   sum of the numbers            */
   float avg;                 /* average of the numbers          */
   float diff;                /* difference between each         */
                              /* number and the average          */
   total = 0;
   for (i = 0; i < NUMELTS; i++) {
      /*  read the numbers into the array and add them            */
      scanf("%d", num[i]);
      total += num[i];
   } /* end for */
   avg = (float) total / NUMELTS;      /*  compute the average */
   printf("\nnumber difference");      /* print heading        */
                        /* print each number and its difference */

   for (i = 0; i < NUMELTS; i++) {
      diff = num[i] - avg;
      printf("\n %d %f", num[i], diff);
   } /* end for */
   printf("\naverage is: %f", avg);
} /* end main */
```

This program uses two groups of 100 numbers. The first group is the set of input integers and is represented by the array *num*, and the second group is the set of differences that are the successive values assigned to the variable *diff* in the second loop. The question arises, why is an array used to hold all the values of the first group simultaneously, whereas only a single variable is used to hold one value of the second group at a time?

The answer is quite simple. Each difference is computed and printed and is never needed again. Thus the variable *diff* can be reused for the difference of the next integer and the average. However, the original integers that are the values of the array *num* must all be kept in memory. Although each can be added into *total* as it is input, it must be retained until after the average is computed in order for the program to compute the difference between it and the average. Therefore, an array is used.

Of course, 100 separate variables could have been used to hold the integers. The advantage of an array, however, is that it allows the programmer to declare only a single identifier and yet obtain a large amount of space. Furthermore, in conjunction with the *for* loop, it also allows the programmer to reference each element of the group in a uniform manner instead of forcing him or her to code a statement such as

```
scanf("%d%d%d...%d", &num0, &num1, &num2, ..., &num99);
```

A particular element of an array may be retrieved through its index. For example, suppose that a company is using a program in which an array is declared by

```
int sales[10];
```

The array will hold sales figures for a ten-year period. Suppose that each line input to the program contains an integer from 0 to 9, representing a year as well as a sales figure for that year, and that it is desired to read the sales figure into the appropriate element of the array. This can be accomplished by executing the statement

```
scanf("%d%d", &yr, &sales[yr]);
```

within a loop. In this statement, a particular element of the array is accessed directly by using its index. Consider the situation if ten variables $s0, s1, \ldots, s9$ had been declared. Then even after executing *scanf*("%d",&*yr*) to set *yr* to the integer representing the year, the sales figure could not be read into the proper variable without coding something like

```
switch(yr) {
   case  0: scanf("%d", &s0);
   case  1: scanf("%d", &s1);
         .
         .
         .
   case  9: scanf("%d", &s9);
} /* end switch */
```

This is bad enough with ten elements—imagine the inconvenience if there were a hundred or a thousand.

Implementing One-Dimensional Arrays

A one-dimensional array can be implemented easily. The C declaration

```
int b[100];
```

reserves 100 successive memory locations, each large enough to contain a single integer. The address of the first of these locations is called the *base address* of the array *b* and is denoted by *base*(*b*). Suppose that the size of each individual element of the array is *esize*. Then a reference to the element *b*[0] is to the element at location *base*(*b*), a reference to *b*[1] is to the element at *base*(*b*) + *esize*, a reference to *b*[2] is to the element *base*(*b*) + 2 * *esize*. In general, a reference to *b*[i] is to the element at location *base*(*b*) + *i* * *esize*. Thus it is possible to reference any element in the array, given its index.

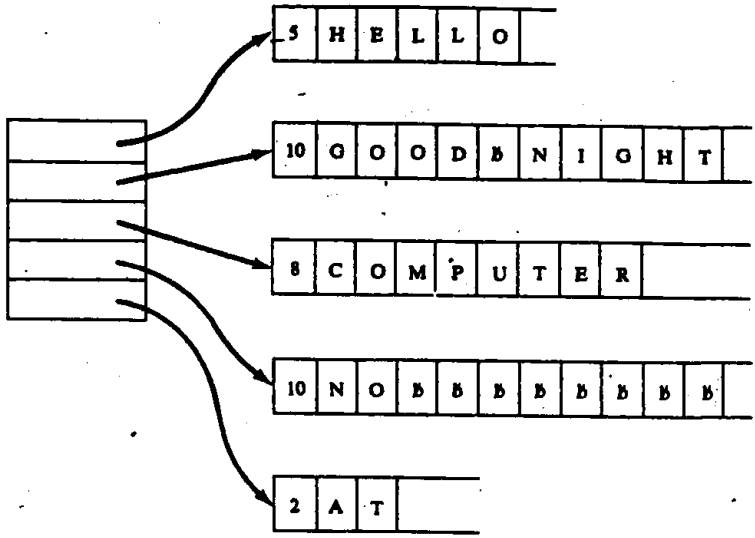
In fact, in the C language an array variable is implemented as a pointer variable. The type of the variable *b* in the above declaration is "pointer to an integer" or *int* *. An asterisk does not appear in the declaration because the brackets automatically imply that the variable is a pointer. The difference between the declarations *int* **b*; and *int* *b*[100]; is that the latter also reserves 100 integer locations starting at location *b*. In C the value of the variable *b* is *base*(*b*), and the value of the variable *b*[*i*], where *i* is an integer, is *(*b* + *i*). Recall from Section 1.1 that, since *b* is a pointer to an integer, *(*b* + *i*) is the value of the *i*th integer following the integer at location *b*. *b*[*i*], the element at location *base*(*b*) + *i* * *esize*, is equivalent to the element pointed to by *b* + *i*, which is *(*b* + *i*).

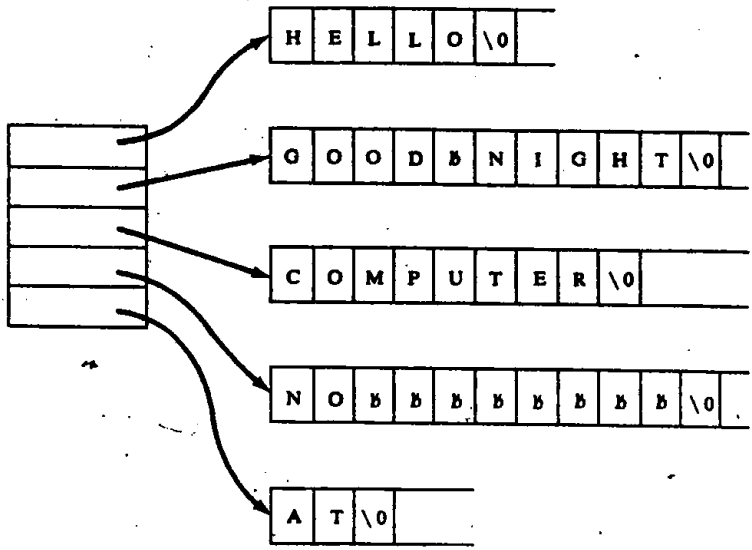
In C all elements of an array have the same fixed, predetermined size. Some programming languages, however, allow arrays of objects of differing sizes. For example, a language might allow arrays of varying-length character strings. In such cases, the above method cannot be used to implement the array. This is because this method of calculating the address of a specific element of the array depends upon knowing the fixed size (*esize*) of each preceding element. If not all the elements have the same size, a different implementation must be used.

One method of implementing an array of varying-sized elements is to reserve a contiguous set of memory locations, each of which holds an address. The contents of each such memory location are the address of the varying-length array element in some other portion of memory. For example, Figure 1.2.1a illustrates an array of five varying-length character strings under the two implementations of varying-length integers presented in Section 1.1. The arrows in the diagram indicate addresses of other portions of memory. The character '␣' indicates a blank. (However, in C a string is itself implemented as an array, so that an array of strings is actually an array of arrays—a two-dimensional rather than a one-dimensional array.)

Since the length of each address is fixed, the location of the address of a particular element can be computed in the same way that the location of a fixed-length element was computed in the previous examples. Once this location is known, its contents can be used to determine the location of the actual array element. This, of course, adds an extra level of indirection to referencing an array element by involving an extra memory reference, which in turn decreases efficiency. However, this is a small price to pay for the convenience of being able to maintain such an array.

A similar method for implementing an array of varying-sized elements is to keep all fixed-length portions of the elements in the contiguous array area, in addition to keeping the address of the varying-length portion in the contiguous area. For example, in the implementation of varying-length character strings presented in the previous section, each such string contains a fixed-length portion (a one-byte length field) and





(a)

Figure 1.2.1 Implementations of an array of varying-length strings. Continues on page 31.

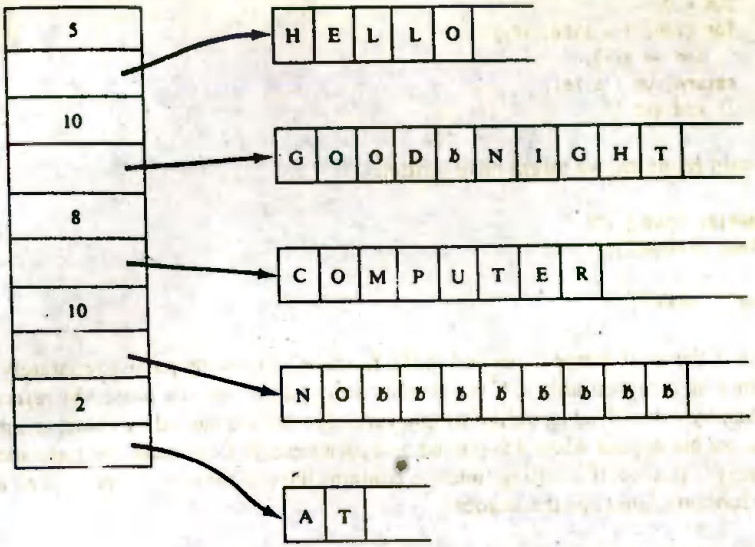


Figure 1.2.1 Concluded.

a variable-length portion (the character string itself). One implementation of an array of varying-length character strings keeps the length of the string together with the address, as shown in Figure 1.2.1b. The advantage of this method is that those parts of an element that are of fixed length can be examined without an extra memory reference. For example, a function to determine the current length of a varying-length character string can be implemented with a single memory lookup. The fixed-length information for an array element of varying length that is stored in the contiguous memory area of the array is often called a ***header***.

Arrays as Parameters

Every parameter of a C function must be declared within the function. However, the range of a one-dimensional array parameter is only specified in the main program. This is because in C new storage is not allocated for an array parameter. Rather, the parameter refers to the original array that was allocated in the calling program. For example, consider the following function to compute the average of the elements of an array:

```
float avg(float a[], int size)   /* no range is specified for the array a */
{
   int i;
   float sum;
```

```
    sum = 0;
    for (i=0; i < size; i++)
       sum += a[i];
    return(sum / size);
} /* end avg */
```

In the main program, we might have written

```
#define ARANGE 100
float a[ARANGE];
    ...
avg(a, ARANGE);
```

Note that if the array range is needed in the function, it must be passed separately.

Since an array variable in C is a pointer, array parameters are passed ***by reference*** rather than by value. That is, unlike simple variables that are passed by value, an array's contents are not copied when it is passed as a parameter in C. Instead, the base address of the array is passed. If a calling function contains the call *funct*(a), where *a* is an array and the function *funct* has the header

```
void funct(int b[])
```

the statement

```
b[i] = x;
```

inside *funct* modifies the value of *a*[*i*] inside the calling function. *b* inside *funct* refers to the same array of locations as *a* in the calling function.

Passing an array by reference rather than by value is more efficient in both time and space. The time that would be required to copy an entire array on invoking a function is eliminated. Also the space that would be needed for a second copy of the array in the called function is reduced to space for only a single pointer variable.

Character Strings in C

A ***string*** is defined in C as an array of characters. Each string is terminated by the *NULL* character, which indicates the end of the string. A string constant is denoted by any set of characters included in double-quote marks. The *NULL* character is automatically appended to the end of the characters in a string constant when they are stored. Within a program, the *NULL* character is denoted by the *escape sequence* \0. Other escape sequences that can be used are \n for a new line character, \t for a tab character, \b for a backspace character, \" for the double-quote character, \\ for the backslash character, \' for the single-quote character, \r for the carriage return character and \f for the form feed character.

A string constant represents an array whose lower bound is 0 and whose upper bound is the number of characters in the string. For example, the string "HELLO

THERE" is an array of twelve characters (the blank and \0 each counts as a character), and "I DON\'T KNOW HIM" is an array of sixteen characters (the *escape sequence* \' represents the single-quote character).

Character String Operations

Let us present C functions to implement some primitive operations on character strings. For all these functions, we assume the global declarations

```
#define STRSIZE 80
char string[STRSIZE];
```

The first function finds the current length of a string.

```
strlen(string)
char string[];
{
   int i;

   for (i=0; string[i] != '\0'; i++)
      ;
   return(i);
} /* end strlen */
```

The second function accepts two strings as parameters. The function returns an integer indicating the starting location of the first occurrence of the second parameter string within the first parameter string. If the second string does not exist within the first, -1 is returned.

```
int strpos(char s1[], char s2[])
{
   int len1, len2;
   int i, j1, j2;

   len1 = strlen(s1);
   len2 = strlen(s2);
   for (i=0; i+len2 <= len1; i++)
      for (j1=i, j2=0; j2 <= len2 && s1[j1] == s2[j2];
                                          j1++, j2++)
         if (j2 == len2)
            return(i);
   return(-1);
} /* end strpos */
```

Another common operation on strings is concatenation. The result of concatenating two strings consists of the characters of the first followed by the characters of the second. The following function sets $s1$ to the concatenation of $s1$ and $s2$.

```
void strcat(char s1[], char s2[])
{
   int i, j;

   for (i=0; s1[i] != '\0'; i++)
      ;
   for (j=0; s2[j] != '\0'; s1[i++] = s2[j++])
      ;
} /* end strcat */
```

The last operation we present on strings is the substring operation. *substr*(*s*1,*i*,*j*,*s*2) sets the string *s*2 to the *j* characters beginning at *s*1[*i*].

```
void substr(char s1[], int i, int j, char s2[])
{
   int k, m;

   for (k = i, m = 0; m < j; s2[m++] = s1[k++])
      ;
   s2[m] = '\0';
} /* end substr */
```

Two-Dimensional Arrays

The component type of an array can be another array. For example, we may define

```
int a[3][5];
```

This defines a new array containing three elements. Each of these elements is itself an array containing five integers. Figure 1.2.2 illustrates such an array. An element of this array is accessed by specifying two indices: a row number and a column number. For example, the element that is darkened in Figure 1.2.2 is in row 1 and column 3 and may be referenced as *a*[1][3]. Such an array is called a ***two-dimensional*** array. The number of rows or columns is called the ***range*** of the dimension. In the array *a*, the range of the first dimension is 3 and the range of the second dimension is 5. Thus array *a* has three rows and five columns.

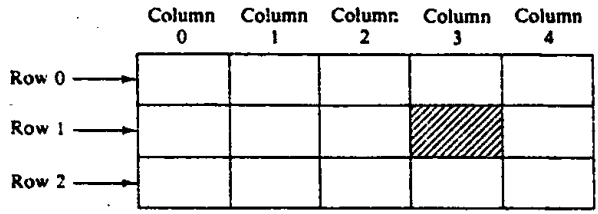


Figure 1.2.2 Two-dimensional arrays.

A two-dimensional array clearly illustrates the differences between a ***logical*** and a ***physical*** view of data. A two-dimensional array is a logical data structure that is useful in programming and problem solving. For example, such an array is useful in describing an object that is physically two-dimensional, such as a map or a checkerboard. It is also useful in organizing a set of values that are dependent upon two inputs. For example, a program for a department store that has 20 branches, each of which sells 30 items, might include a two-dimensional array declared by

```
int sales[20][30];
```

Each element *sales*[*i*][*j*] represents the amount of item *j* sold in branch *i*.

However, although it is convenient for the programmer to think of the elements of such an array as being organized in a two-dimensional table (and programming languages do indeed include facilities for treating them as a two-dimensional array), the hardware of most computers has no such facilities. An array must be stored in the memory of a computer, and that memory is usually linear. By this we mean that the memory of a computer is essentially a one-dimensional array. A single address (that may be viewed as a subscript of a one-dimensional array) is used to retrieve a particular item from memory. To implement a two-dimensional array, it is necessary to develop a method of ordering its elements in a linear fashion and of transforming a two-dimensional reference to the linear representation.

One method of representing a two-dimensional array in memory is the ***row-major*** representation. Under this representation, the first row of the array occupies the first set of memory locations reserved for the array, the second row occupies the next set, and so forth. There may also be several locations at the start of the physical array that serve as a header and that contain the upper and lower bounds of the two dimensions. (This header should not be confused with the headers discussed earlier. This header is for the entire array, whereas the headers mentioned earlier are headers for the individual array elements.) Figure 1.2.3 illustrates the row-major representation of the two-dimensional array *a* declared above and illustrated in Figure 1.2.2. Alternatively, the header need not be contiguous to the array elements but could instead contain the address of the first element of the array. Additionally, if the elements of the two-dimensional array are variable-length objects, the elements of the contiguous area could themselves contain the addresses of those objects in a form similar to those of Figure 1.2.1 for linear arrays.

Let us suppose that a two-dimensional integer array is stored in row-major sequence, as in Figure 1.2.3, and let us suppose that, for an array *ar*, *base*(*ar*) is the address of the first element of the array. That is, if *ar* is declared by

```
int ar[r1][r2];
```

where *r*1 and *r*2 are the ranges of the first and second dimension, respectively, *base*(*ar*) is the address of *ar*[0][0]. We also assume that *esize* is the size of each element in the array. Let us calculate the address of an arbitrary element, *ar*[*i*1][*i*2]. Since the element is in row *i*1, its address can be calculated by computing the address of the first element of row *i*1 and adding the quantity *i*2 * *esize* (this quantity represents how far into row *i*1 the element at column *i*2 is). But to reach the first element of row *i*1 (that is, the element

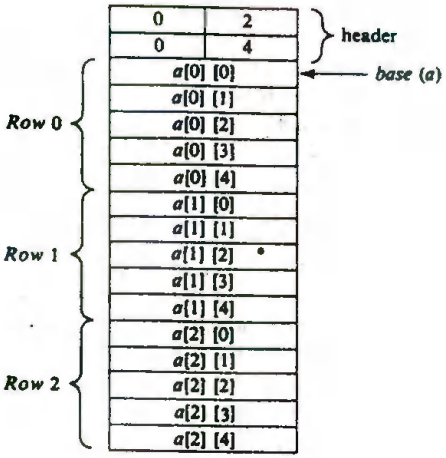


Figure 1.2.3 Representing a two-dimensional array.

$ar[i1][0]$), it is necessary to pass through $i1$ complete rows, each of which contains $r2$ elements (since there is one element from each column in each row), so that the address of the first element of row $i1$ is at $base(ar) + i1 * r2 * esize$. Therefore the address of $ar[i1][i2]$ is at

```
base(ar) + (i1 * r2 + i2) * esize
```

As an example, consider the array a of Figure 1.2.2, whose representation is illustrated in Figure 1.2.3. In this array, $r1 = 3$, $r2 = 5$, and $base(a)$ is the address of $a[0][0]$. Let us also suppose that each element of the array requires a single unit of storage, so that *esize* equals 1. (This is not necessarily true, since a was declared as an array of integers and an integer may need more than one unit of memory on a particular machine. For simplicity, however, we accept this assumption.) Then the location of $a[2][4]$ can be computed by

```
base[a] + (2 * 5 + 4) * 1
```

that is,

```
base(a) + 14
```

You may confirm the fact that $a[2][4]$ is fourteen units past $base(a)$ in Figure 1.2.3.

Another possible implementation of a two-dimensional array is as follows: An array ar, declared with upper bounds $u1$ and $u2$, consists of $u1+1$ one-dimensional arrays. The first is an array ap of $u1$ pointers. The ith element of ap is a pointer to a one-dimensional array whose elements are the elements of the one-dimensional array $ar[i]$. For example, Figure 1.2.4 illustrates such an implementation for the array a of Figure 1.2.2, where $u1$ is 3 and $u2$ is 5.

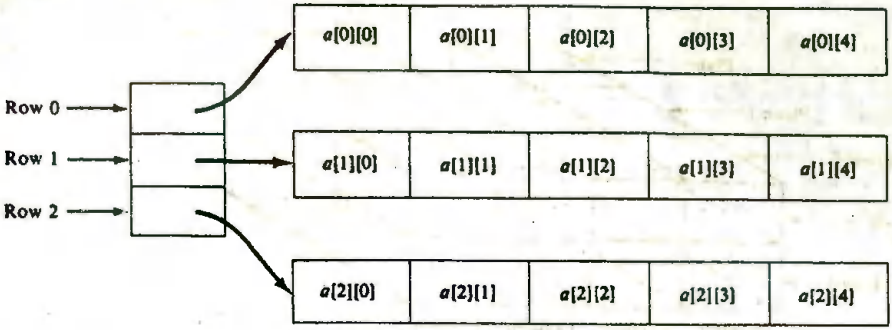


Figure 1.2.4 Alternative implementation of a two-dimensional array.

To reference $ar[i][j]$, the array ar is first accessed to obtain the pointer $ar[i]$. The array at that pointer location is then accessed to obtain $a[i][j]$.

Indeed, this second implementation is the simpler and more straightforward of the two. However, the $u1$ arrays $ar[0]$ through $ar[u1 - 1]$ would usually be allocated contiguously, with $ar[0]$ immediately followed by $ar[1]$, and so on. The first implementation avoids allocating the extra pointer array, ap, and computing the value of an explicit pointer to the desired row array. It is therefore more efficient in both space and time.

Multidimensional Arrays

C also allows arrays with more than two dimensions. For example, a three-dimensional array may be declared by

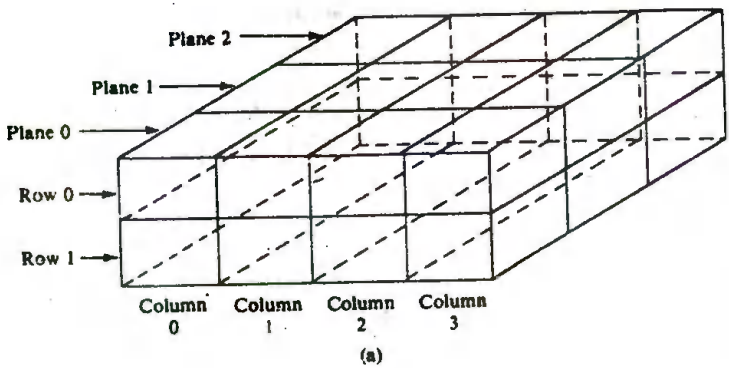
```
int b[3][2][4];
```

and is illustrated in Figure 1.2.5a. An element of this array is specified by three subscripts, such as $b[2][0][3]$. The first subscript specifies a plane number, the second subscript a row number, and the third a column number. Such an array is useful when a value is determined by three inputs. For example, an array of temperatures might be indexed by latitude, longitude, and altitude.

For obvious reasons, the geometric analogy breaks down when we go beyond three dimensions. However, C does allow an arbitrary number of dimensions. For example, a six-dimensional array may be declared by

```
int c [7][15][3][5][8][2];
```

Referencing an element of this array would require six subscripts, such as $c[2][3][0][1][6][1]$. The number of different subscripts that are allowed in a particular position (the range of a particular dimension) equals the upper bound of that dimension. The number of elements in an array is the product of the ranges of all its dimensions.



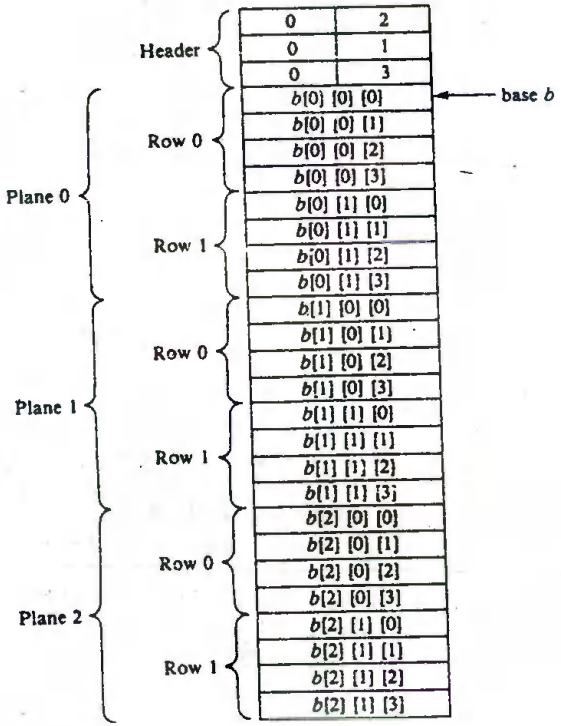


Figure 1.2.5 Three-dimensional array.

For example, the array *b* contains 3 * 2 * 4 = 24 elements, and the array *c* contains 7 * 15 * 3 * 5 * 8 * 2 = 25,200 elements.

The row-major representation of arrays can be extended to arrays of more than two dimensions. Figure 1.2.5b illustrates the representation of the array *b* of Figure 1.2.5a. The elements of the previously described six-dimensional array *c* are ordered as follows:

```
C[0][0][0][0][0][0]
C[0][0][0][0][0][1]
C[0][0][0][0][1][0]
C[0][0][0][0][1][1]
C[0][0][0][0][2][0]
      . . .
      . . .
C[6][14][2][4][5][0]
C[6][14][2][4][5][1]
C[6][14][2][4][6][0]
C[6][14][2][4][6][1]
C[6][14][2][4][7][0]
C[6][14][2][4][7][1]
```

That is, the last subscript varies most rapidly, and a subscript is not increased until all possible combinations of the subscripts to its right have been exhausted. This is similar to an odometer (mileage indicator) of a car where the rightmost digit changes most rapidly.

What mechanism is needed to access an element of an arbitrary multidimensional array? Suppose that *ar* is an *n*-dimensional array declared by

```
int ar[r1][r2]...[rn];
```

and stored in row-major order. Each element of AR is assumed to occupy *esize* storage locations, and *base*(*ar*) is defined as the address of the first element of the array (that is, *ar*[0][0] ... [0]). Then, to access the element

```
ar[i1][i2]...[in];
```

it is first necessary to pass through *i*1 complete "hyper-planes," each consisting of *r*2 * *r*3 * ... * *rn* elements to reach the first element of *ar*, whose first subscript is *i*1. Then it is necessary to pass through an additional *i*2 groups of *r*3* *r*4 * ... * *rn* elements to reach the first element of *ar*, whose first two subscripts are *i*1 and *i*2, respectively. A similar process must be carried out through the other dimensions until the first element whose first $n - 1$ subscripts match those of the desired element is reached. Finally, it is necessary to pass through *in* additional elements to reach the element desired.

Thus the address of *ar*[*i*1][*i*2] ... [*in*] may be written as *base*(*ar*) + *esize* * [*i*1 * *r*2 * ... * *rn* + *i*2 * *r*3 * ... * *rn* + ... + (*i*(*n* − 1) * *rn* + *in*)], which can be evaluated more efficiently by using the equivalent formula:

```
base(ar) + esize *
[in + rn * (i(n - 1) + r(n - 1) * (... + r3 * (i2 + r2 * i1) ...) ) ]
```

This formula may be evaluated by the following algorithm, which computes the address of the array element and places it into *addr* (assuming arrays *i* and *r* of size *n* to hold the indices and the ranges, respectively):

```
offset = 0;
for (j = 0; j < n; j++)
    offset = r[j] * offset + i[j];
addr = base(ar) + esize * offset;
```

EXERCISES

1.2.1. (a) The ***median*** of an array of numbers is the element m of the array such that half the remaining numbers in the array are greater than or equal to m and half are less than or equal to m, if the number of elements in the array is odd. If the number of elements is even, the median is the average of the two elements m_1 and m_2 such that half the remaining elements are greater than or equal to m_1 and m_2, and half the elements are less than or equal to m_1 and m_2. Write a C function that accepts an array of numbers and returns the median of the numbers in the array.

(b) The ***mode*** of an array of numbers is the number m in the array that is repeated most frequently. If more than one number is repeated with equal maximal frequencies, there is no mode. Write a C function that accepts an array of numbers and returns the mode or an indication that the mode does not exist.

1.2.2. Write a C program to do the following: Read a group of temperature readings. A reading consists of two numbers: an integer between −90 and 90, representing the latitude at which the reading was taken, and the observed temperature at that latitude. Print a table consisting of each latitude and the average temperature at that latitude. If there are no readings at a particular latitude, print "NO DATA" instead of an average. Then print the average temperature in the northern and southern hemispheres (the northern consists of latitudes 1 through 90 and the southern consists of latitudes −1 through −90). (This average temperature should be computed as the average of the averages, not the average of the original readings.) Also determine which hemisphere is warmer. In making the determination, take the average temperatures in all latitudes of each hemisphere for which there are data for both that latitude and the corresponding latitude in the other hemisphere. (For example, if there is data for latitude 57 but not for latitude −57, then the average temperature for latitude 57 should be ignored in determining which hemisphere is warmer.)

1.2.3. Write a program for a chain of 20 department stores, each of which sells 10 different items. Every month, each store manager submits a data card for each item consisting of a branch number (from 1 to 20), an item number (from 1 to 10), and a sales figure (less than $100,000) representing the amount of sales for that item in that branch. However, some managers may not submit cards for some items (for example, not all items are sold in all branches). You are to write a C program to read these data cards and print a table with 12 columns. The first column should contain the branch numbers from 1 to 20 and the word "TOTAL" in the last line. The next 10 columns should contain the sales figures

for each of the 10 items for each of the branches, with the total sales of each item in the last line. The last column should contain the total sales of each of the 20 branches for all items, with the grand total sales figure for the chain in the lower right-hand corner. Each column should have an appropriate heading. If no sales were reported for a particular branch and item, assume zero sales. Do not assume that your input is in any particular order.

1.2.4. Show how a checkerboard can be represented by a C array. Show how to represent the state of a game of checkers at a particular instant. Write a C function that is input to an array representing such a checkerboard and prints all possible moves that black can make from that position.

1.2.5. Write a function *printar*(*a*) that accepts an *m*-by-*n* array *a* of integers and prints the values of the array on several pages as follows: Each page is to contain 50 rows and 20 columns of the array. Along the top of each page, headings "COL 0," "COL 1," and so forth, should be printed and along the left margin of each page, headings "ROW 0," "ROW 1," and so forth, should be printed. The array should be printed by subarrays. For example, if *a* were a 100-by-100 array, the first page contains *a*[0][0] through *a*[49][19], the second page contains *a*[0][20] through *a*[49][39], the third page contains *a*[0][40] through *a*[49][59], and so on until the fifth page contains *a*[0][80] through *a*[49][99], the sixth page contains *a*[50][0] through *a*[99][19], and so on. The entire printout occupies ten pages. If the number of rows is not a multiple of 50, or the number of columns is not a multiple of 20, the last pages of the printout should contain fewer than 100 numbers.

1.2.6. Assume that each element of an array *a* stored in row-major order occupies four units of storage. If *a* is declared by each of the following, and the address of the first element of *a* is 100, find the address of the indicated array element:

a. `int a[100];` address of `a[10]`
b. `int a[200];` address of `a[100]`
c. `int a[10][20];` address of `a[0][0]`
d. `int a[10][20];` address of `a[2][1]`
e. `int a[10][20];` address of `a[5][1]`
f. `int a[10][20];` address of `a[1][10]`
g. `int a[10][20];` address of `a[2][10]`
h. `int a[10][20];` address of `a[5][3]`
i. `int a[10][20];` address of `a[9][19]`

1.2.7. Write a C function *listoff* that accepts two one-dimensional array parameters of the same size: *range* and *sub*. *range* represents the range of an integer array. For example, if the elements of *range* are

3 5 10 6 3

range represents an array *a* declared by

```
int a[3][5][10][6][3];
```

The elements of *sub* represent subscripts to the foregoing array. If *sub*[*i*] does not lie between 0 and *range*[*i*] − 1, all subscripts from the *i*th onwards are missing. In the foregoing example, if the elements of *sub* are

1 3 1 2 3

sub represents the one-dimensional array *a*[1][3][1][2]. The function *listoff* should print the offsets from the base of the array *a* represented by *range* of all the elements of *a* that are included in the array (or the offset of the single element if all subscripts are within bounds) represented by *sub*. Assume that the size (*esize*) of each element of *a* is 1. In the foregoing example, *listoff* would print the values 4, 5, and 6.

1.2.8. (a) A ***lower triangular*** array *a* is an *n*-by-*n* array in which *a*[*i*][*j*] = = 0, if *i* < *j*. What is the maximum number of nonzero elements in such an array? How can these elements be stored sequentially in memory? Develop an algorithm for accessing *a*[*i*][*j*], where *i* > = *j*. Define an ***upper triangular*** array in an analogous manner and do the same for such an array as for the lower triangular array.

(b) A ***strictly lower triangular*** array *a* is an *n*-by-*n* array in which *a*[*i*][*j*] = = 0 if *i* <= *j*. Answer the questions of part a for such an array.

(c) Let *a* and *b* be two *n*-by-*n* lower triangular arrays. Show how an *n*-by-(*n* + 1) array *c* can be used to contain the nonzero elements of the two arrays. Which elements of *c* represent the elements *a*[*i*][*j*] and *b*[*i*][*j*], respectively?

(d) A ***tridiagonal*** array *a* is an *n*-by-*n* array in which *a*[*i*][*j*] = = 0, if the absolute value of *i* − *j* is greater than 1. What is the maximum number of nonzero elements in such an array? How can these elements be stored sequentially in memory? Develop an algorithm for accessing *a*[*i*][*j*] if the absolute value of *i* − *j* is 1 or less. Do the same for an array *a* in which *a*[*i*][*j*] = = 0, if the absolute value of *i* − *j* is greater than *k*.

1.3 STRUCTURES IN C

In this section we examine the C data structure called a ***structure***. We assume that you are familiar with the structure from an introductory course. In this section we review some highlights of this data structure and point out some interesting and useful features needed for a more general study of data structures.

A structure is a group of items in which each item is identified by its own identifier, each of which is known as a ***member*** of the structure. (In many other programming languages, a structure is called a "record" and a member is called a "field." We may sometimes use these terms instead of "structure" or "member," although both terms have different meanings in C.) For example, consider the following declaration:

```
struct {
   char first[10];
   char midinit;
   char last[20];
} sname, ename;
```

This declaration creates two structure variables, *sname* and *ename*, each of which contains three members: *first*, *midinit*, and *last*. Two of the members are character strings, and the third is a single character. Alternatively, we can assign a ***tag*** to the structure and then declare the variables by means of the tag. For example, consider the following declaration that accomplishes the same thing as the declaration just given:

```
struct nametype {
   char first[10];
   char midinit;
   char last[20];
};
struct nametype sname, ename;
```

This definition creates a structure tag *nametype* containing three members, *first*, *midinit*, and *last*. Once a structure tag has been defined, variables *sname* and *ename* may be declared. For maximum program clarity, it is recommended that a tag be declared for each structure and variables then be declared using the tag.

An alternative to using a structure tag is to use the *typedef* definition in C. For example

```
typedef struct {
          char first[10];
          char midinit;
          char last[20];
        } NAMETYPE;
```

says that the identifier *NAMETYPE* is synonymous with the preceding structure specifier wherever *NAMETYPE* occurs. We can then declare

```
NAMETYPE sname, ename;
```

to achieve the declarations of the structure variables *sname* and *ename*. Note that structure tags are conventionally written in lowercase but *typedef* specifiers are written in uppercase in presenting C programs. *typedef* is sometimes used to achieve the flavor of an ADT specification within a C program.

Once a variable has been declared as a structure, each member within that variable may be accessed by specifying the variable name and the item's member identifier separated by a period. Thus, the statement

```
printf("%s", sname.first);
```

can be used to print the first name in the structure *sname*, and the statement

```
ename.midinit = 'm'
```

can be used to set the middle initial in the structure *ename* to the letter *m*. If a member of a structure is an array, a subscript may be used to access a particular element of the array, as in

```
for (i=0; i < 20; i++)
   sname.last[i] = ename.last[i];
```

A member of a structure may be declared to be another structure. For example, given the foregoing definition of *nametype* and the following definition of *addrtype*

```
struct addrtype {
    char straddr[40];
    char city[10];
    char state[3];    /*Allow room for two-character */
                      /*     abbreviation and '\0'   */
   char zip[6];       /*Allow room for five-character */
                      /*       zipcode and '\0'      */
};
```

we may declare a new structure tag *nmadtype* by

```
struct nmadtype {
   struct nametype name;
   struct addrtype address;
};
```

If we declare two variables

```
struct nmadtype nmad1, nmad2;
```

the following are valid statements:

```
nmad1.name.midinit = nmad2.name.midinit;
nmad2.address.city[4] = nmad1.name.first[1];
for (i=1; i < 10; i++)
   nmad1.name.first[i] = nmad2.name.first[i];
```

ANSI standard C allows the assignment of structures of the same type. For example, the statement *nmad1 = nmad2;* is valid and equivalent to

```
nmad1.name = nmad2.name;
nmad2.address = nmad2.address;
```

These, in turn, are equivalent to

```
for (i=0; i < 10; i++)
   nmad1.name.first[i] = nmad2.name.first[i];
nmad1.name.midinit = nmad2.name.midinit;
for (i=0; i < 20; i++)
   nmad1.name.last[i] = nmad2.name.last[i];
for (i=0; i < 40; i++)
   nmad1.address.straddr[i] = nmad2.address.straddr[i];
for (i=0; i < 10; i++)
   nmad1.address.city[i] = nmad2.address.city[i];
for (i=0; i < 2; i++)
   nmad1.address.state[i] = nmad2.address.state[i];
for (i=0; i < 5; i++)
   nmad1.address.zip[i] = nmad2.address.zip[i];
```

The reader is cautioned that many compilers, which are based on the original C language as defined by Kernighan and Ritchie, do not permit structure assignment. Thus it would be necessary to explicitly assign each member of one structure to another. In the remainder of the text we assume ANSI C compliance.

Consider another example of the use of structures, in which we define structures describing an employee and a student, respectively:

```
struct date {
   int month;
   int day;
   int year;
};
struct position {
   char deptno[2];
   char jobtitle[20];
};
struct employee {
   struct nmadtype nameaddr;
   struct position job;
   float salary;
   int numdep;
   short int hplan;
   struct date datehired;
};
struct student {
   struct nmadtype nmad;
   float gpindx;
   int credits;
   struct date dateadm;
};
```

Assuming the declarations

```
struct employee e;
struct student s;
```

a statement to give a 10 percent raise to an employee whose grade point index as a student was above 3.0 is the following where *strcmp* (*s, t*) returns 0 if strings *s* and *t* are equal.

```
if ((strcmp(e.nameaddr.name.first,s.nmad.name.first)==0) &&
    (e.nameaddr.name.midinit == s.nmad.name.midinit) &&
    (strcmp(e.nameaddr.name.last,s.nmad.name.last)==0))
   if (s.gpindx > 3.0)
     e.salary *= 1.10;
```

This statement first ensures that the employee record and the student record refer to the same person by comparing their names. Note that we cannot simply write

```
if (e.nmaddr.name == s.nmad.name)
   ...
```

since two structures cannot be compared for equality in a single operation in C.

You may have noticed that we used two different identifiers *nameaddr* and *nmad* for the name/address members of the employee and student records, respectively. It is not necessary to do so and the same identifier can be reused to name members of different structure types. This does not cause any ambiguity, since a member name must always be preceded by an expression identifying a structure of a specific type.

Implementing Structures

Let us now turn our attention from the application of structures to their implementation. Any type in C may be thought of as a pattern or a template. By this we mean that a type is a method for interpreting a portion of memory. When a variable is declared as being of a certain type, we are saying that the identifier refers to a certain portion of memory and that the contents of that memory are to be interpreted according to the pattern defined by the type. The type specifies both the amount of memory set aside for the variable and the method by which memory is interpreted.

For example, suppose that under a certain C implementation an integer is represented by four bytes, a float number by eight, and an array of ten characters by ten bytes. Then the declarations

```
int x;
float y;
char z[10];
```

specify that four bytes of memory be set aside for x, eight bytes be set aside for y, and ten bytes for z. Once those bytes are set aside for these variables, the names x, y, and z will always refer to those locations. When x is referenced, its four bytes will be interpreted as an integer; when y is referenced, its eight bytes will be interpreted as a real number; and when z is referenced, its ten bytes will be interpreted as a collection of ten characters. The amount of storage set aside for each type and the method by which the contents of memory are interpreted as specific types vary from one machine and C implementation to another. But within a given C implementation, any type always indicates a specific amount of storage and a specific method of interpreting that storage.

Now suppose that we defined a structure by

```
struct structtype {
   int field1;
   float field2;
   char field3[10];
};
```

and declared a variable

```
struct structtype r;
```

Then the amount of memory specified by the structure is the sum of the storage specified by each of its member types. Thus, the space required for the variable *r* is the sum of the space required for an integer (4 bytes), a float number (8 bytes), and an array of 10 characters (10 bytes). Therefore, 22 bytes are set aside for *r*. The first 4 of these bytes are interpreted as an integer, the next 8 as a float number, and the last 10 as an array of characters. (This is not always true. On some computers, objects of certain types may not begin anywhere in memory but are constrained to start at certain "boundaries." For example, an integer of length 4 bytes may have to start at an address divisible by 4, and a real number of length 8 bytes may have to start at an address divisible by 8. Thus, in our example, if the starting address of *r* is 200, the integer occupies bytes 200 through 203, but the real number cannot start at byte 204, since that location is not divisible by 8. Thus the real number must start at location 208 and the entire record requires 26, rather than 22, bytes. Bytes 204 through 207 are wasted space.)

For every reference to a member of a structure, an address must be calculated. Associated with each member identifier of a structure is an ***offset*** that specifies how far beyond the start of the structure the location of that field is. In the foregoing example, the offset of *field*1 is 0, the offset of *field*2 (assuming no boundary restrictions) is 4, and the offset of *field*3 is 12. Associated with each structure variable is a base address, which is the location of the start of the memory allocated to that variable. These associations are established by the compiler and are of no concern to the user. To calculate the location of a member in a structure, the offset of the member identifier is added to the base address of the structure variable.

For example, assume that the base address of *r* is 200. Then what really happens in executing a statement such as

```
r.field2 = r.field1 + 3.7;
```

is the following. First, the location of *r.field*1 is determined as the base address of *r* (200) plus the field offset of *field*1 (0), which yields 200. The 4 bytes at locations 200 through 203 are interpreted as an integer. This integer is then converted to a float number that is then added to the float number 3.7. The result is a float number that takes up 8 bytes. The location of *r.field*2 is then computed as the base address of *r* (200) plus the field offset of *field*2 (4), or 204. The contents of the 8 bytes 204 through 211 are set to the float number computed in evaluating the expression.

Note that the process of calculating the address of a structure component is very similar to that of calculating the address of an array component. In both cases an offset that depends on the component selector (the member identifier or the subscript value) is added to the base address of the compound structure (the structure or the array). In the case of a structure, the offset is associated with the field identifier by the type

definition, whereas in the case of an array, the offset is calculated based on the value of the subscript.

These two types of addressing (structure and array) may be combined. For example, to calculate the address of *r.field3*[4], we first use structure addressing to determine the base address of the array *r.field3* and then use array addressing to determine the location of the fifth element of that array. The base address of *r.field3* is given by the base address of *r* (200) plus the offset of *field3* (12), which is 212. The address of *r.field3*[4] is then determined as the base address of *r.field3* (212) plus 4 (the subscript 4 minus the lower array bound 0) times the size of each element of the array (1), which yields 212 + 4 * 1, or 216.

As an additional example, consider another variable, *rr*, declared by

```
struct structtype rr[20];
```

rr is an example of an array of structures. If the base address of *rr* is 400, then the address of *rr*[14].*field3*[6] may be computed as follows. The size of each component of *rr* is 22, so the location of *rr*[14] is 400 + 14 * 22, or 708. The base address of *rr*[14].*field3* is then 708 + 12, or 720. The address of *rr*[14].*field3*[6] is therefore 720 + 6 * 1, or 726. (Again, this ignores the possibility of boundary restrictions. For example, although the type *rectype* may require only 22 bytes, each *rectype* may have to start at an address divisible by 4, so that 2 bytes are wasted between each element of *rr* and its neighbor. If such is the case, then the size of each element of *rr* is really 24, so that the address of *rr*[14].*field3*[6] is actually 754 rather than 726.)

Unions

Thus far each structure we have looked at has had fixed members and a single format. C also allows another type of structure, the ***union***, which permits a variable to be interpreted in several different ways.

For example, consider an insurance company that offers three kinds of policies: life, auto, and home. A policy number identifies each insurance policy, of whatever kind. For all three types of insurance, it is necessary to have the policyholder's name, address, the amount of the insurance, and the monthly premium payment. For auto and home insurance policies, a deductible amount is needed. For a life insurance policy, the insured's birth date and beneficiary are needed. For an auto insurance policy, a license number, state, car model, and year are required. For a homeowner's policy, an indication of the age of the house and the presence of any security devices is required. A policy structure type for such a company may be defined as a union. We first define two auxiliary structures.

```
#define LIFE 1
#define AUTO 2
#define HOME 3
```

```
struct addr {
   char street[50];
   char city[10];
   char state[3];
   char zip[6];
};
struct date {
   int month;
   int day;
   int year;
};
struct policy {
   int polnumber;
   char name[30];
   struct addr address;
   int amount;
   float premium;
   int kind;          /* LIFE, AUTO, or HOME */
   union {
      struct {
         char beneficiary[30];
         struct date birthday;
      } life;
      struct {
         int autodeduct;
         char license[10];
         char state[3];
         char model[15];
         int year;
      } auto;
      struct {
         int homededuct;
         int yearbuilt;
      } home;
   } policyinfo;
}
```

Let us examine the union more closely. The definition consists of two parts: a fixed part and a variable part. The fixed part consists of all member declarations up to the keyword *union*, while the variable part consists of the remainder of the definition.

Now that we have examined the syntax of a union definition, let us examine its semantics. A variable declared as being of a union type T (for example, *struct policy p;*) always contains all the fixed members of T. Thus, it is always valid to reference *p.name* or *p.premium* or *p.kind*. However, the union members contained in the value of such a variable depend on what has been stored by the programmer.

It is the programmer's responsibility to make sure that the use of a member is consistent with what has been placed into that location. It is a good idea to maintain a separate fixed member in a structure containing a union whose value indicates which

alternative is currently in use. In the foregoing example, the member *kind* is used for this purpose. If its value is LIFE (1), then the structure holds a life insurance policy; if AUTO (2), an auto insurance policy; and if HOME (3), a home insurance policy. Thus the programmer would be required to execute code similar to the following to reference the union:

```
if (p.kind == LIFE)
    printf("\n%s %2d//%2d//%4d", p.policyinfo.life.beneficiary,
          p.policyinfo.life.birthday.month,
          p.policyinfo.life.birthday.day,
          p.policyinfo.life.birthday.year);
else if (p.kind == AUTO)
    printf("\n%d %s %s %s %d", p.policyinfo.auto.autodeduct,
          p.policyinfo.auto.license,
          p.policyinfo.auto.state,
          p.policyinfo.auto.model,
          p.policyinfo.auto.year);
else if (p.kind == HOME)
    printf("\n%d %d", p.policyinfo.home.homededuct,
          p.policyinfo.home.yearbuilt);
else
    printf("\nbad type %d in kind", p.kind);
```

In the foregoing example, if the value of *p.kind* is LIFE, *p* currently contains members *beneficiary* and *birthday*. It is invalid to reference *model* or *yearbuilt* while the value of *kind* is LIFE. Similarly, if the value of *kind* is AUTO, we may reference *autodeduct*, *license*, *state*, *model*, and *year* but should not reference any other member. However, the C language does not require a fixed member to indicate the current alternative of a union, nor does it enforce using a particular alternative depending on a fixed member's value.

A union allows a variable to take on several different "types" at different points in execution. It also allows an array to contain objects of different types. For example, the array *a*, declared by

```
struct policy a[100];
```

may contain life, auto, and home insurance policies. Suppose that such an array *a* is declared and that it is desired to raise the premiums of all life insurance policies and all home insurance policies for homes built before 1950 by 5 percent. This can be done as follows:

```
for (i=0; i<100; i++)
   if (a[i].kind == LIFE)
      a[i].premium = 1.05 * a[i].premium;
else if (a[i].kind == HOME &&
                        a[i].policyinfo.yearbuilt < 1950)
      a[i].premium = 1.05 * a[i].premium;
```

Implementation of Unions

To fully understand the concept of a union, it is necessary to examine its implementation. A structure may be regarded as a road map to an area of memory. It defines how the memory is to be interpreted. A union provides several different road maps for the same area of memory, and it is the responsibility of the programmer to determine which road map is in current use. In practice, the compiler allocates sufficient storage to contain the largest member of the union. It is the road map, however, that determines how that storage is to be interpreted. For example, consider the simple union and structures

```
#define INTEGER 1
#define REAL 2

struct stint {
   int f3, f4;
};
struct stfloat {
   float f5, f6;
};
struct sample {
   int f1;
   float f2;
   int utype;
   union {
      struct stint x;
      struct stfloat y;
   } funion;
};
```

Let us again assume an implementation in which an integer requires 4 bytes and a float 8 bytes. Then the three fixed members *f*1, *f*2, and *utype* occupy 16 bytes. The first member of the union, *x*, requires 8 bytes, while the second member, *y*, requires 16. The memory actually allocated for the union part of such a variable is the maximum of the space needed by any single member. In this case, therefore, 16 bytes are allocated for the union part of *sample*. Added to the 16 bytes needed for the fixed part, 32 bytes are allocated to *sample*.

The different members of a union overlay each other. In the above example, if space for *sample* is allocated starting at location 100, so that *sample* occupies bytes 100 through 131, the fixed members *sample.f* 1, *sample.f*2, and *sample.utype* occupy bytes 100 through 103, 104 through 111, and 112 through 115, respectively. If the value of the member *utype* is INTEGER (that is, 1), bytes 116 through 119 and 120 through 123 are occupied by *sample.funion.x.f*3 and *sample.funion.x.f*4, respectively, and bytes 124 through 131 are unused. If the value of *sample.utype* is REAL (that is, 2), bytes 116 through 123 are occupied by *sample.funion.y.f*5, and bytes 124 through 131 are occupied by *sample.funion.y.f*6. That is why only a single member of a union can exist at a single instant. All the members of the union use the same space, and that space

can be used by only one of them at a time. The programmer determines which member is appropriate.

Structure Parameters

In traditional C a structure may not be passed to a function by means of a call by value. To pass a structure to a function, we must pass its address to the function and refer to the structure by means of a pointer (that is, call by reference). The notation $p->x$ in C is equivalent to the notation $(*p).x$ and is frequently used to reference a member of a structure parameter. For example, the following function prints a name in a neat format and returns the number of characters printed:

```
int writename (struct nametype *name)
{
   int count, i;

   printf("\n");
   count = 0;
   for (i=0; (i < 10) && (name->first[i] != '\0'); i++) {
      printf("%c", name->first[i]);
      count++;
   } /* end for */
   printf("%c", ' ');
   count++;
   if (name->midinit != ' ') {
      printf("%c%s", name->midinit, ". ");
      count += 3;
   } /* end if */
   for (i=0; (i < 20) && (name->last[i] != '\0'); i++) {
      printf("%c", name->last[i]);
      count++;
   } /* end for */
   return(count);
} /* end writename */
```

The following list illustrates the effects of the statement x = *writename*(*&sname*) on two different values of *sname:*

Value of *sname.first:*	"Sara"	"Irene"
Value of *sname.midinit:*	'M'	' '
Value of *sname.last:*	"Binder"	"LaClaustra"
Printed output:	Sara M. Binder	Irene LaClaustra
Value of *x:*	14	16

Similarly, the statement x = *writename*(*&ename*) prints the values of *ename*'s fields and assigns the number of characters printed to x.

The original definition of C (by Kernighan and Ritchie) and many older C compilers do not allow a structure to be passed as an argument even if its value remains

unchanged. ANSI C, in addition to allowing structure assignment, does allow structures to be passed by value and returned, without applying the & operator. This involves copying the value of the entire structure when the function is called. Thus if the structure is very large it is more efficient to pass the structure by reference (that is, using the & operator). In the remainder of the text, we therefore pass all structures by reference.

We have already seen that a member of a structure may be an array or another structure. Similarly we may declare an array of structures. For example, if the types *employee* and *student* are declared as presented earlier, we can declare two arrays of employee and student structures as follows:

```
struct employee e[100];
struct student s[100];
```

The salary of the fourteenth employee is referenced by *e*[13].*salary*, and the last name is referenced by *e*[13].*nameaddr.name.last*. Similarly, the admission year of the first student is *s*[0].*dateadm.year*.

As an additional example, we present a function used at the start of a new year to give a 10 percent raise to all employees with more than ten years seniority and a 5 percent raise to all others. First, we must define a new array of structures.

```
struct employee empset[100];
```

The procedure now follows:

```
#define THISYEAR ...
void raise (struct employee e[])
{
   int i;

   for (i=0; i < 100; i++)
      if (e[i].datehired.year < THISYEAR - 10)
         e[i].salary *= 1.10;
      else
         e[i].salary *= 1.05;
} /* end raise */
```

As another example, suppose that we add an additional member, *sindex*, to the definition of the *employee* structure. This member contains an integer and indicates the student index in the array *s* of the particular employee. Let us declare *sindex* (within the *employee* record) as follows:

```
struct employee {
   struct nametype nameaddr;
      ...
   struct datehired ...;
   int sindex;
};
```

The number of credits earned by employee *i* when the employee was a student can then be referenced by *s[e[i].sindex].credits*.

The following function can be used to give a 10 percent raise to all employees whose grade point index was above 3.0 as a student and to return the number of such employees. Note that we no longer have to compare an employee name with a student name to ascertain that their records represent the same person (although these names should be equal if they do). Instead the field *sindex* can be used directly to access the appropriate student record for an employee. We assume that the main program contains the declaration

```
struct employee emp[100];
struct student stud[100];

int raise2 (struct employee e[], struct student s[])
{
   int i, j, count;

   count = 0;
   for (i=0; i < 100; i++) {
      j = e[i].sindex;
      if (s[j].gpindx > 3.0) {
         count++;
         e[i].salary *= 1.10;
      } /* end if */
   } /* end for */
   return(count);
} /* end raise2 */
```

Very often a large array of structures is used to contain an important data table for a particular application. There is generally only one table for each such array of structures. The student table *s* and the employee table *e* of the previous discussion are good examples of such data tables. In such cases, the unique tables are often used as static/external variables rather than as parameters, with a large number of functions accessing them. This increases efficiency by eliminating the overhead of parameter passing. We could easily rewrite the function *raise2* above to access *s* and *e* as static/external variables rather than as parameters by simply changing the function header to

```
int raise2()
```

The body of the function need not be changed, assuming that the tables *s* and *e* are declared in the outer program.

Representing Other Data Structures

Throughout the remainder of this text, structures are used to represent more complex data structures. Aggregating data into a structure is useful because it enables us to group objects within a single entity and to name each of these objects appropriately, according to its function.

As examples of how structures can be used in this fashion, let us consider the problems of representing rational numbers.

Rational Numbers

In Section 1.1 we presented an ADT for rational numbers. Recall that a ***rational number*** is any number that can be expressed as the quotient of two integers. Thus 1/2, 3/4, 2/3, and 2 (that is, 2/1) are all rational numbers, whereas sqr(2) and π are not. A computer usually represents a rational number by its decimal approximation. If we instruct the computer to print 1/3, the computer responds with .333333. Although this is close enough (the difference between .333333 and one-third is only one three-millionth), it is not exact. If we were to ask for the value of 1/3 + 1/3, the result would be .666666 (which equals .333333 + .333333), whereas the result of printing 2/3 might be .666667. This would mean that the result of the test 1/3 + 1/3 = = 2/3 would be false! In most instances, the decimal approximation is good enough, but sometimes it is not. It is therefore desirable to implement a representation of rational numbers for which exact arithmetic can be performed.

How can we represent a rational number exactly? Since a rational number consists of a numerator and a denominator we can represent a rational number *rational* using structures as follows:

```
struct rational {
   int numerator;
   int denominator;
};
```

An alternative way of declaring this new type is

```
typedef struct {
           int numerator;
           int denominator;
        } RATIONAL;
```

Under the first technique, a rational *r* is declared by

```
struct rational r;
```

under the second technique by

```
RATIONAL r;
```

You might think that we are now ready to define rational number arithmetic for our new representation, but there is one significant problem. Suppose that we defined two rational numbers *r*1 and *r*2 and we had given them values. How can we test if the two numbers are the same? Perhaps you might want to code

```
if (r1.numerator == r2.numerator && r1.denominator ==
                                          r2.denominator)
   ...
```

That is, if both numerators and denominators are equal, the two rational numbers are equal. However, it is possible for both numerators and denominators to be unequal, yet the two rational numbers are the same. For example, the numbers 1/2 and 2/4 are indeed equal, although their numerators (1 and 2) as well as their denominators (2 and 4) are unequal. We therefore need a new way of testing equality under our representation.

Well, why are 1/2 and 2/4 equal? The answer is that they both represent the same ratio. One out of two and two out of four are both one-half. To test rational numbers for equality, we must first reduce them to lowest terms. Once both numbers have been reduced to lowest terms, we can then test for equality by simple comparison of their numerators and denominators.

Define a ***reduced rational number*** as a rational number for which there is no integer that evenly divides both the denominator and the numerator. Thus 1/2, 2/3, and 10/1 are all reduced, while 4/8, 12/18, and 15/6 are not. In our example, 2/4 reduced to lowest terms is 1/2, so the two rational numbers are equal.

A procedure known as Euclid's algorithm can be used to reduce any fraction of the form *numerator/denominator* into its lowest terms. This procedure may be outlined as follows:

1. Let a be the larger of the *numerator* and *denominator* and let b be the smaller.
2. Divide b into a, finding a quotient q and a remainder r (that is, $a = q * b + r$).
3. Set $a = b$ and $b = r$.
4. Repeat steps 2 and 3 until b is 0.
5. Divide both the *numerator* and the *denominator* by the value of a.

As an illustration, let us reduce 1032/1976 to its lowest terms.

Step 0	*numerator* = 1032		*denominator* = 1976	
Step 1	$a = 1976$	$b = 1032$		
Step 2	$a = 1976$	$b = 1032$	$q = 1$	$r = 944$
Step 3	$a = 1032$	$b = 944$		
Steps 4 and 2	$a = 1032$	$b = 944$	$q = 1$	$r = 88$
Step 3	$a = 944$	$b = 88$		
Steps 4 and 2	$a = 944$	$b = 88$	$q = 10$	$r = 64$
Step 3	$a = 88$	$b = 64$		
Steps 4 and 2	$a = 88$	$b = 64$	$q = 1$	$r = 24$
Step 3	$a = 64$	$b = 24$		
Steps 4 and 2	$a = 64$	$b = 24$	$q = 2$	$r = 16$
Step 3	$a = 24$	$b = 16$		
Steps 4 and 2	$a = 24$	$b = 16$	$q = 1$	$r = 8$
Step 3	$a = 16$	$b = 8$		
Steps 4 and 2	$a = 16$	$b = 8$	$q = 2$	$r = 0$
Step 3	$a = 8$	$b = 0$		
Step 5	1032/8 = 129		1976/8 = 247	

Thus 1032/1976 in lowest terms is 129/247.

Let us write a function to reduce a rational number (we use the tag method for declaring rationals).

```
void reduce (struct rational *inrat, struct rational *outrat)
{
   int a, b, rem;

   if (inrat->numerator > inrat->denominator) {
       a = inrat->numerator;
       b = inrat->denominator;
   } /* end if */
   else {
      a = inrat->denominator;
      b = inrat->numerator;
   } /* end else */
   while (b != 0) {
      rem = a % b;
      a = b;
      b = rem;
   } /* end while */
   outrat->numerator /= a;
   outrat->denominator /= a;
} /* end reduce */
```

Using the function *reduce*, we can write another function *equal* that determines whether or not two rational numbers *r1* and *r2* are equal. If they are, the function returns *TRUE*; otherwise, the function returns *FALSE*.

```
#define TRUE 1
#define FALSE 0

int equal (struct rational *rat1, struct rational *rat2)
{
   struct rational r1, r2;

   reduce(rat1, &r1);
   reduce(rat2, &r2);
   if (r1.numerator == r2.numerator &&
                       r1.denominator == r2.denominator)
       return(TRUE);
   return(FALSE);
} /* end equal */
```

We may now write functions to perform arithmetic on rational numbers. We present a function to multiply two rational numbers and leave as an exercise the problem of writing similar functions to add, subtract, and divide such numbers.

```
void multiply (struct rational *r1, struct rational *r2, struct rational *r3)
/* r3 points to the result of multiplying *r1 and *r2  */
{
   struct rational rat3;

   rat3.numerator   = r1->numerator   * r2->numerator;
   rat3.denominator = r1->denominator * r2->denominator;
   reduce(&rat3, r3);
} /* end multiply */
```

Allocation of Storage and Scope of Variables

Until now we have been concerned with the declaration of variables, that is, the description of a variable's type or attribute. Two important questions, however, remain to be answered: At what point is a variable associated with actual storage (that is, ***storage allocation***)? At what point in a program may a particular variable be referenced (that is, ***scope*** of variables)?

In C variables and parameters declared within a function are known as ***automatic*** variables. Such variables are allocated storage when the function is invoked. When the function terminates, storage assigned to those variables is deallocated. Thus automatic variables exist only as long as the function is active. Furthermore, automatic variables are said to be ***local*** to the function. That is, automatic variables are known only within the function in which they are declared and may not be referenced by other functions.

Automatic variables (that is, parameters in a function header or local variables immediately following any opening brace) can be declared within any block and remain in existence until the block is terminated. The variable can be referenced throughout the entire block unless the variable identifier is redeclared within an internal block. Within the internal block, a reference to the identifier is to the inner-most declaration, and the outer variable cannot be referenced.

The second class of variables in C are the ***external*** variables. Variables that are declared outside any function are allocated storage at the point at which they are first encountered and remain in existence for the remainder of the program's execution. The scope of an external variable lasts from the point at which it is declared until the end of its containing source file. Such variables may be referred to by all functions in that source file lying beyond their declaration and are therefore said to be ***global*** to those functions.

A special case is when the programmer wishes to define a global variable in one source file and to refer to the variable in another source file. Such a variable must be explicitly declared to be external. For example, suppose that an integer array containing grades is declared in source file 1 and it is desired to refer to that array throughout a source file 2. The following declarations would then be necessary:

```
file 1          #define MAXSTUDENTS ...
                int grades[MAXSTUDENTS];

                ...

end of file 1
```

```
file 2          extern int grades[];

                float average()
                {
                    ...
                } /* end float */

                float mode()
                {
                    ...
                } /* end mode */

end of file 2
```

When file 1 and file 2 are combined into one program, storage for the array *grades* is allocated in file 1 and remains allocated until the end of file 2. Since *grades* is an external variable, it is global from the point at which it is defined in file 1 to the end of file 1 and from the point at which it is declared in file 2 to the end of file 2. Both functions *average* and *mode* may therefore refer to *grades*.

Note that the size of the array is specified only once, at the point at which the variable is originally defined. This is because a variable that is explicitly declared to be external cannot be redefined, nor can any additional storage be allocated to it. An *extern* declaration merely serves to declare for the remainder of that source file that such a variable exists and has been created earlier.

Occasionally it is desirable to define a variable within a function for which storage remains allocated throughout the execution of the program. For example, it might be useful to maintain a local counter in a function that would indicate the number of times the function is invoked. This can be done by including the word ***static*** in the variable declaration. A ***static*** internal variable is local to that function but remains in existence throughout the program's execution rather than being allocated and deallocated each time the function is invoked. When the function is exited and reentered, a static variable retains its value. Similarly, a ***static*** external variable is also allocated storage only once, but may be referred to by any function that follows it in the source file.

For purposes of optimization, it might be useful to instruct the compiler to maintain the storage for a particular variable in a high-speed register rather than in ordinary memory. Such a variable is known as a ***register variable*** and is defined by including the word ***register*** in the declaration of an automatic variable or in the formal parameter of a function. There are many restrictions on register variables that vary from machine to machine. The reader is urged to consult the appropriate manuals for details on these restrictions.

Variables may be explicitly initialized as part of a declaration. Such variables are conceptually given their initial values prior to execution. Uninitialized external and static variables are initialized to 0, whereas uninitialized automatic and register variables have undefined values.

To illustrate these rules consider the following program: (The numbers to the left of each line are for reference purposes.)

source file1.c

```
1   int x, y, z;

2   void func1()
3   {
4      int a, b;

5      x = 1;
6      y = 2;
7      z = 3;
8      a = 1;
9      b = 2;
10     printf("%d %d %d %d %d\n", x, y, z, a, b);
11  } /* end func1 */

12  void func2()
13  {
14     int a;

15     a = 5;
16     printf("%d %d %d %d\n", x, y, z, a);
17  } /* end func2 */
```

end of source file1.c

source file2.c

```
18  #include <stdio.h>
19  #include <file1.c>

20  extern int x, y, z;
21  void main()
22  {
23     func1();
24     printf("%d %d %d\n", x, y, z);
25     func2();
26     func3();
27     func3();
28     func4();
29     printf("%d %d %d\n", x, y, z);
30  } /* end main */

31  void func3()
32  {
33     static int b;        /* b is initialized to 0 */

34     y++;
35     b++;
36     printf("%d %d %d %d\n", x, y, z, b);
37  } /* end func3 */
```

```
38  void func4()
39  {
40      int x, y, z;

41      x = 10;
42      y = 20;
43      z = 30;
44      printf("%d %d %d\n", x, y, z);
45  } /* end func4 */

end of source file2.c
```

Execution of the program yields the following results:

```
a       1 2 3 1 2
b       1 2 3
c       1 2 3 5
d       1 3 3 1
e       1 4 3 2
f       10 20 30
g       1 4 3
```

Let us trace through the program. Execution begins with line 1, in which external integer variables *x*, *y*, and *z* are defined. Being externally defined, they will be known (global) throughout the remainder of *file*1.*c* (lines 1 through 17). Execution then proceeds to line 20, which declares by means of the word ***extern*** that the external integer variables *x*, *y*, and *z* are to be associated with the variables of the same name in line 1. No new storage is allocated at this point, since storage is allocated only when these variables are originally defined (line 1). Being external, *x*, *y*, and *z* will be known throughout the remainder of *file*2.*c*, with the exception of *func*4 (lines 38 through 45), where the declaration of local automatic variables *x*, *y*, and *z* (line 40) supersedes the original definition.

Execution begins with *main*(), line 21. This immediately invokes *func*1. *func*1 (lines 2 through 4) defines local automatic variables *a* and *b* (line 4) and assigns values to the global variables (lines 5 through 7) and to its local variables (lines 8 through 9). Line 10 therefore produces the first line of output (line a). Upon termination of *func*1 (line 11) storage for variables *a* and *b* is deallocated. Thus, no other function will be able to refer to these variables.

Control is then returned to the main function (line 24). The output is given in line b. It then invokes *func*2. *func*2 (lines 12 through 17) defines a local automatic variable, *a*, for which storage is allocated (line 14) and a value assigned (line 15). Line 16 refers to the external (global) variables *x*, *y*, and *z* previously defined in line 1 and assigned values in lines 5 through 7. The output is given in line b. Note that it would be illegal for *func*2 to attempt to print a value for *b*, since this variable no longer exists, being allocated only within *func*1.

The main program then invokes *func*3 twice (lines 26 through 27). *func*3 (lines 31 through 37), when called for the first time, allocates storage for the static local variable *b*

and initializes it to 0 (line 33). *b* will be known only to *func*3; however, it will remain in existence for the remainder of the program's execution. Line 34 increments the global variable *y*, and line 35 increments the local variable *b*. Line d of the output is then printed. The second time *func*3 is invoked by the main program, new storage for *b* is not allocated; thus, when *b* is incremented in line 35 the old value of *b* (from the previous invocation of *func*3) is used. The final value of *b* thus will reflect the number of times that *func*3 was invoked.

Execution then continues in the main function that invokes *func*4 (line 28). As was mentioned earlier, the definition of internal-automatic integer variables *x*, *y*, and *z* in line 40 supersedes the definition of *x*, *y*, and *z* in lines 1 and 20, and remains in force only within the scope of *func*4 (lines 38 through 45). Thus the assignment of values in lines 41 through 43 and the output (line f) resulting from line 44 refer only to these local variables. As soon as *func*4 terminates (line 45) these variables are destroyed. Subsequent references to *x*, *y*, and *z* (line 29) refer to the global *x*, *y*, and *z* (lines 1 and 20) producing the output of line g.

EXERCISES

1.3.1. Implement complex numbers, as specified in Exercise 1.1.8, using structures with real and complex parts. Write routines to add, multiply, and negate such numbers.

1.3.2. Suppose that a real number is represented by a C structure such as

```
struct realtype {
   int left;
   int right;
};
```

where *left* and *right* represent the digits to the left and right of the decimal point, respectively. If *left* is a negative integer, the represented real number is negative.

(a) Write a routine to input a real number and create a structure representing that number.

(b) Write a function that accepts such a structure and returns the real number represented by it.

(c) Write routines *add*, *subtract*, and *multiply* that accept two such structures and set the value of a third structure to represent the number that is the sum, difference, and product, respectively, of the two input records.

1.3.3. Assume that an integer needs four bytes, a real number needs eight bytes, and a char needs one byte. Assume the following definitions and declarations:

```
struct nametype {
   char first[10];
   char midinit;
   char last[20];
};
```

```
struct person {
   struct nametype name;
   int birthday[2];
   struct nametype parents[2];
   int income;
   int numchildren;
   char address[20];
   char city[10];
   char state[2];
};
struct person p[100];
```

If the starting address of p is 100, what are the starting addresses (in bytes) of each of the following?

(a) `p[10]`
(b) `p[20].name.midinit`
(c) `p[20].income`
(d) `p[20].address[5]`
(e) `p[5].parents[1].last[10]`

1.3.4. Assume two arrays, one of student records and the other of employee records. Each student record contains members for a last name, a first name, and a grade point index. Each employee record contains members for a last name, a first name, and a salary. Both arrays are ordered in alphabetical order by last name and first name. Two records with the same last name/first name do not appear in the same array. Write a C function to give a 10 percent raise to every employee who has a student record and whose grade-point index is greater than 3.0.

1.3.5. Write a function as in the preceding exercise, but assuming that the employee and student records are kept in two ordered external files, rather than in two ordered arrays.

1.3.6. Using the rational number representation given in the text, write routines to add, subtract, and divide such numbers.

1.3.7. The text presents a function *equal* that determines whether or not two rational numbers $r1$ and $r2$ are equal by first reducing $r1$ and $r2$ to lowest terms and then testing for equality. An alternative method would be to multiply the denominator of each by the numerator of the other and test the two products for equality. Write a function *equal2* to implement this algorithm. Which of the two methods is preferable?

1.4 CLASSES IN C++

In this section, we introduce the C++ language and the concept of a C++ ***class***. A class embodies the concept of an abstract data type by defining both the set of values of a given type and the set of operations that can be performed on those values. A variable of a class type is known as an ***object*** and the operations on that type are called ***methods***. When one object A invokes a method m on another object B, we sometimes say that "A is sending message m to B." B is viewed as receiving that message and carrying out a transformation in response to that message.

To illustrate the concept of a C++ class, consider the abstract data type *RATIONAL* that we introduced in Section 1.1. That ADT modeled a rational number as consisting of two components, a numerator and a denominator, and defined methods to check if two rational numbers were equal, to add two rationals, to multiply two rationals and to create a rational from two integers. In Section 1.3 we implemented the ADT *RATIONAL* as a C structure and presented C functions to implement its operations. We recommend that you reread the portions of Sections 1.1 and 1.3 dealing with the definition and implementation of the ADT *RATIONAL* before proceeding.

A C++ class builds on the concept of a C structure. Whereas a C structure is a collection of named fields, a C++ class is a collection of named fields and methods (or functions) that apply to objects of that class type. Additionally, the C++ language implements the concept of ***information hiding,*** restricting access to certain members of the class to methods of the class itself.

For example, the C++ definition of a class to implement the ADT *RATIONAL* might be the following:

```
class Rational{
        long numerator;
        long denominator;
        void reduce (void);
public:
        Rational add(Rational);
        Rational mult(Rational);
        Rational divide(Rational);
        int equal(Rational);
        void print(void);
        void setrational(long, long);
}
```

This class, named *Rational*, contains two data members, *numerator* and *denominator*, and seven method members, *reduce*, *add*, *mult*, *divide*, *equal*, *print*, and *setrational*. These seven methods are merely defined here; their actual implementations must be provided subsequently.

The methods *add*, *mult*, and *equal* implement the ADT functions of the same name. We have added a method *divide* to divide one rational number by another. While the corresponding ADT functions take two parameters, the methods in the class *Rational* explicitly mention only one. This is because the class object for which they are invoked is an implicit parameter for each routine. We will see how this is done shortly.

The method *setrational* is used to set the value of a *Rational* to the rational number formed by a particular numerator and denominator. Further, the members are divided into two groups: *numerator*, *denominator* and *reduce* are ***private***. That is, they can be referenced only from within the methods of the class *Rational*. This could have been made explicit by stating:

```
class Rational {
private:
      long numerator;
        ...
```

but, by default, the members defined at the beginning of a class definition are private without the need for explicitly stating this. The members *setrational*, *add*, *mult*, *divide*, *equal*, and *print*, by contrast, are ***public***. This means that they can be referenced outside the methods of the class *Rational*.

The reasons for doing this are simple. We do not want "outsiders" manipulating either the *rational* or *denominator* members. They are merely a way of implementing a rational number and are to be used solely for that purpose. An external function manipulates a *Rational*; only within the internal methods of *Rational* should we be able to access *numerator* and *denominator*. Similarly, the method *reduce* is a function to reduce the internal representation of the *Rational* (that is, the numerator and denominator) to lowest terms. We intend to use *reduce* to ensure that every rational number is kept in lowest terms. The outside world has no cause to call *reduce*. Every method that manipulates the internal numerator and denominator (that is, *setrational*, *add*, *mult*, and *divide*) is a member of the class *Rational* and will automatically ensure that the resulting number is in reduced form by calling *reduce*. We will see this when we present the implementations of these methods. There is no need for anyone else to call *reduce*, and therefore *reduce* is defined as private.

On the other hand, the methods *setrational*, *add*, *mult*, *equal*, and *print* are public. These functions form the public interface for the class *Rational*. That is, they are the methods by which the outside world can manipulate and use objects of type *Rational*.

Using the Class *Rational*

We now present an example of the use of the class *Rational*. Suppose that the class *Rational*, together with the implementations of its methods, were defined in a header file *rational.h*. Then, suppose that we wanted to write a program to do the following: Input lines of the form

```
op   ra   rb;
```

where *op* is the character + or *, and *ra* and *rb* are either integers or of the form

```
a  /  b
```

where *a* and *b* are integers. For example, the following are all valid input lines:

```
+  3  7;
```

This asks to add the integers 3 and 7 to produce 10.

```
+  3  /  4  5;
```

This asks to add the rational 3/4 and the integer 5 to produce the rational 23/4.

```
* 3 4 / 8;
```

This asks to multiply the integer 3 and the rational 4/8 to produce the rational 3/2.

```
+ 3 / 4 5 / 6;
```

This asks to add the rationals 3/4 and 5/6 to produce the rational 19/12. The program should read the line, perform the indicated computation, and print out the resulting rational.

To assist with the I/O, we assume three routines: *int readtoken(char * *), long atol(char *),* and *void error(char *).* The function *readtoken* reads the next operator or integer in character form (for example, "/" or "389"), allocates storage for the string using the *stdlib* function *calloc*, and sets a pointer of type *char* * to it. Should the end of file be encountered, *readtoken* returns a value of EOF; otherwise, it returns !EOF. The *stdlib* function *atol* converts a numerical string to an integer. The function *void error(char *)* prints its parameter as an error message and halts execution. The program is as follows:

```
#include "rational.h"
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

void main()
{
     int readtoken(char * *);
     void error(char *);

     char *optr, *token1, *token2, *token3;
     int int1, int2;
     Rational opnd1, opnd2, result;

     while (readtoken(&optr) != EOF) {    // read the operator
          readtoken(&token1);             // read the first integer's
                                          // character string
          int1 = atol(token1);            // convert the first token
                                          // to an integer
          readtoken(&token2);
          if (strcmp(token2, "/") != 0
               // convert the integer operand to a Rational
               opnd1.setrational(int1, 1);
          else {
               // get the denominator of the Rational operand
               readtoken(&token3);
               int2 = atol(token3);
               // convert the numerator and denominator to a Rational
               opnd1.setrational(int1, int2);
```

```
            readtoken(&token2);
        } /* end if */
        // get the second operand
        int1 = atol(token2);
        readtoken(&token2);
        if (strcmp(token2, "/") != 0)
            // convert the integer operand to a Rational
            opnd2.setrational(int1, 1);
        else {
            // get the denominator of the Rational operand
            readtoken(&token3);
            int2 = atol(token3);
            // convert the numerator and denominator to a Rational
            opnd2.setrational(int1, int2);
            readtoken(&token2);
        } /* end if */
        if (strcmp(token2, ";") != 0)
            error ("ERROR: ; expected, not found.");
        // apply the operator to the Rational operands
        if (*optr == '+')
            result = opnd1.add(opnd2);
        else if (*optr == '*')
            result = opnd1.mult(opnd2);
        else
            error("ERROR: illegal operator; must be * or + ");
        result.print();
    } /* end while */
} /* end main */
```

In the declarations, the variables *opnd*1, *opnd*2, and *result* are declared as of type *Rational*. This makes them objects of that class. That is, each one contains a numerator and denominator, and can be used to call the methods of class *Rational*, including *add*, *mult*, *setrational*, and *print*.

Note how the methods of *Rational* are called. If *opnd*1 and *opnd*2 are *Rationals*, then the call *opnd*1.*add*(*opnd*2) adds the two rational numbers represented by *opnd*1 and *opnd*2 and produces a *Rational* representing the result. We say that the program "sends the message" *add* to *opnd*1. Similarly, the call *result.print*() sends the message *print* to *result* and the call *opnd*1.*setrational*(*int*1, *int*2) sends the message *setrational* to *opnd*1.

Implementing the Methods

The methods of a class can be implemented within the declaration of the class or outside it. For example, the method *setrational*, which sets an object of type *Rational* to a particular value, can be implemented within the class declaration as follows:

```
class Rational
    long numerator;
    ...
```

```
public:
        Rational add(Rational);
            ...
        void print(void);
        void setrational (long n, long d)
        {
            if (d == 0)
                error ("ERROR: denominator may not be zero");
            numerator = n;
            denominator = d;
            reduce();              // reduce to lowest terms
         } /* end setrational */
} /* end Rational */
```

The body of the function *setrational* appears within the declaration of *Rational*. Note that *setrational* references the members *numerator* and *denominator*. Private members of a class, such as *numerator* and *denominator*, can be referenced directly, with no needed qualification, from a method of that class, but they cannot ordinarily be referenced by a function outside the class. Similarly, the method *reduce* is called within the method *setrational* with no qualification. This refers to the method *reduce* defined within the method *Rational*.

When *setrational* is called, as in the statement *opnd1.setrational(int1, int2)*;, references to *numerator* and *denominator* within *setrational* will refer to *opnd1.numerator* and *opnd1.denominator*, and the call *reduce*() is to *opnd1.reduce*(). Alternatively, *setrational* can be defined outside the declaration for *Rational*. *Rational* must still contain the header of the function

```
void setrational(long, long);
```

but it need not contain its body. The body can be provided after the declaration for *Rational* is completed, as follows:

```
void Rational::setrational(long n, long d)
{
    if (d == 0)
        error("ERROR: denominator may not be zero");
    numerator = n;
    denominator = d;
    reduce();                          // reduce to lowest terms
} /* end setrational */
```

Note the header line

```
void Rational::setrational(long n, long d)
```

which specifies that we are defining one of the methods within the type *Rational*. The notation "*Rational*::" introduces a scope and specifies that the function *setrational* be-

ing defined is a method of class *Rational*. Once this scope has been opened, we can utilize the private members *numerator* and *denominator* and the private method *reduce*.

Here are the specifications of the remaining methods of *RATIONAL*. The routine for *reduce* follows closely the algorithm of Section 1.3. We first make sure that *numerator* and *denominator* are both positive, keeping track of the sign.

```
void Rational::reduce (void)
{
     int a, b, rem, sign;

     if (numerator == 0)
          denominator = 1;
     sign = 1;           //assume positive
     // check if any negatives
     if (numerator < 0 && denominator < 0) {
          numerator = -numerator;
          denominator = -denominator;
     }
     if (numerator < 0) {
          numerator = -numerator;
          sign = -1;
     }
     if (denominator < 0) {
          denominator = -denominator;
          sign = -1;
     }
     if (numerator > denominator) {
          a = numerator;
          b = denominator;
     }
     else {
          a = denominator;
          b = numerator;
     }
     while (b != 0) {
          rem = a % b;
          a = b;
          b = rem;
     }
     numerator = sign * numerator / a;
     denominator = denominator / a;
} /* end reduce */
```

To add two rational numbers, we could first reduce each to lowest term, then multiply the two denominators to produce a resulting denominator, then multiply each numerator by the denominator of the other rational numbers and add the two products to produce the numerator. The result can then be reduced to lowest terms. However, this provides the danger that the product of the two denominators may be too large even for a ***long*** variable.

Instead, we use the following algorithm to add a/b to c/d. We assume that the value *rden*(*x*, *y*) denotes the denominator of *x*/*y* reduced to lowest terms:

```
k = rden(b, d);
denom = b*k;                    // the resulting denominator
num = a*k + c*(denom/d);        // the resulting numerator
```

num is the numerator of the sum, *denom* is the denominator. We leave it as an exercise to show that this algorithm is correct.

Implementing this algorithm in the context of the class *Rational* provides the following definition of the method *add:*

```
Rational Rational::add(Rational r)
{
    int k, denom, num;
    Rational rnl;

    // first reduce both rationals to lowest terms
    reduce();
    r.reduce();

    // implement the line k = rden(b, d); of the algorithm
    rnl.setrational(denominator, r.denominator);
    rnl.reduce();
    k = rnl.denominator;

    // compute the denominator of the result
    // algorithm line denom = b*k;
    denom = denominator * k;

    // compute the numerator of the result
    // algorithm line num = a*k + c*a(denom/d);
    num = numerator*k + r.numerator*(denom/r.denominator);

    // form a Rational from the result and reduce
    //  the result to lowest terms
    rnl.setrational(num, denom);
    rnl.reduce();

    return rnl;
} /* end add */
```

In multiplying two reduced rationals *a*/*b* and *c*/*d*, the straightforward method is to compute (*a***c*)/(*b***d*). However, here again we want to avoid the multiplication of *a***c* and *b***d* if at all possible, since their intermediate results may be too large. The solution is first to reduce *a*/*b* and *c*/*d* to lowest terms, then to reduce *a*/*d* and *c*/*b*. In that way we are certain that *a***c* has no terms in common with *b***d* and that the products are as small as possible.

Here is the method *mult* implementing these ideas:

```
Rational Rational::mult(Rational r)
{
     Rational rnl, rnl1, rnl2;
     int num, denom;

     // reduce both inputs to lowest terms
     reduce();
     r.reduce();

     // switch numerators and denominators and reduce
     rnl1.setrational(numerator, r.denominator);
     rnl.reduce();
     rnl2.setrational(r.numerator,denominator);
     rnl2.reduce();

     //  compute result
     num = rnl1.numerator * rnl2.numerator;
     denom = rnl1.denominator * rnl2.denominator;
     rnl.setrational(num, denom);
     return rnl;
} /* end mult */
```

The method *divide* simply multiplies by a reciprocal.

```
Rational Rational::divide(Rational r)
{
     Rational rnl;

     // Compute the reciprocal of r
     rnl.setrational(r.denominator, r.numerator);

     // Multiply by the reciprocal
     return mult(rnl);
}
```

The method *equal* reduces both rationals to lowest terms and then checks for equality of both numerators and denominators.

```
int Rational::equal(Rational r)
{
     reduce();
     r.reduce();
     if (numerator == r.numerator &&
                              denominator == r.denominator)
          return TRUE;
     else
          return FALSE;
} /* end equal */
```

To implement the method *print* we first must decide on a format for the output. A reasonable format might be to print the numerator followed by a slash followed by the denominator. We adopt this format in the routine below:

```
void Rational::print(void)
{
     cout << numerator << "/" << denominator << endl;
} /* end print */
```

This utilizes the C++ I/O facilities of the header file *iostream.h*. *cout* is an output stream and the operator << is used to send data values to the output.

Overloading

While we have a routine for adding two rational numbers, we cannot yet add a rational *r* and an integer *i* without first converting the integer to a rational using the call *setrational*(*i*, 1).

Fortunately, C++ allows function names to be ***overloaded***. That is, the same function name can apply to different functions if their parameters are of different types. Specifically, we can define another method *add* in the class *Rational* by including the line

```
Rational add(long);
```

in the ***public*** section. There are now two methods named *add*: one applied to *Rationals* and one applied to integers. Implementation of the new method is straightforward:

```
Rational Rational::add(long i)
{
     Rational r;

     r.setrational(i, 1);
     return add(r);
}
```

The implementation is as follows: First, form a new rational out of the integer using *setrational*, then call the existing *add* routine on rationals to add *r* to the rational number of the current object.

If *i* is an integer, the call *rr.add*(*i*) is a call to the second *add* method to add an integer to the rational *rr*. If *r* is a rational, the call *rr.add*(*r*) is a call to the original *add* method to add a rational to *rr*.

Inheritance

However, this technique for adding an integer is not entirely satisfactory. Under the method we have presented, the rational number of the current object is the first

operand, so we can implement the concept *r+i*. However, we cannot implement *i+r* equally directly. For addition, this is not a real problem since *r+i* equals *i+r* because of the commutative law. But consider the case of division, which is not commutative. We can write a method *Rational divide(**int**)* to compute *r/i*, but how do we compute *i/r*?

Of course, we could write a separate routine that is not part of a class, with two parameters, as follows:

```
Rational divide(long i, Rational r)
{
    Rational rr;

    rr.setrational(i, 1);
    return rr.divide(r);
}
```

However, this makes the division operator nonsymmetrical and breaks the concept of class operations.

Instead, we can note that integers are a form of rationals. We can therefore represent every integer by a rational. We do this by defining a new class *Integer* which ***inherits*** the members of the class *Rational*. In this new class, we want to make sure that the denominator is always 1, so we redefine the method *setrational*. In fact, we define two versions of *setrational*.

Here is the definition of the new class:

```
class Integer:public Rational {
public:
      void setrational(long, long);
      void setrational(long);
};
```

The class *Rational* is called the ***base class*** of the class *Integer*.

However, in order for *setrational* to access the *Rational* members *numerator* and *denominator*, those members cannot have been defined as **private**. A private member can only be accessed by the methods of the class itself, not even by the methods of an inherited class. To allow *Rational* to serve as a base class for *Integer* and give *Integer* access to its members, yet to keep those members inaccessible from the rest of the program, the members must be defined as **protected** rather than **private**. The new definition of *Rational* would then be

```
class Rational {
protected:
    long numerator;
    long denominator;
    void reduce(void);
public:

};
```

Let us now implement the two methods named *setrational* in the definition of *Integer*. The first *setrational* is included to override the routine *setrational* in the class *Rational* so that a noninteger rational is not accidentally assigned to an object of type *Integer*. It is implemented as follows:

```
void Integer::setrational(long num, long denom)
{
     if(denom != 1)
          error("ERROR: non-integer assigned to Integer variable");
     numerator = num;
     denominator = 1;
}
```

The more usual version of *Integer::setrational*, with one parameter, is as follows:

```
void Integer::setrational(long num)
{
     numerator = num;
     denominator = 1;
}
```

Now if *r* is a *Rational* and *i* is an *Integer*, then any of the following calls are valid:

```
r.add(i)
i.add(r)
r.divide(i)
i.divide(r)
```

The methods *add* and *divide*, defined for *Rational*, are inherited by *Integer* and can be invoked for an *Integer* variable.

Note that the definition of the class *Integer* begins with the line

```
class Integer:public Rational {
```

The indication **public** in this line specifies that *Integer* has access to the protected and public members of *Rational*, and they become, in turn, protected and public members of *Integer*. However, if *Rational* were a protected base class (that is, **protected** appeared instead of **public** in the opening line of the *Integer* definition), the public members of *Rational* would become protected members of *Integer*. Similarly, if *Rational* were a private base class, the public and protected members of *Rational* would become private members of *Integer*.

Constructors

A ***constructor*** is a special method of a class that is invoked whenever an object of that class is created. A constructor always is named with the same name as the class

itself. In our example above, we used the method *setrational* to initialize a *Rational* object. We could have used a constructor instead.

For example, suppose that we include in the class definition of *Rational* the following three members, all public and all named *Rational*.

```
Rational(void);
Rational(long);
Rational(long, long);
```

They are implemented as follows:

```
Rational::Rational(void)
{
    // assume the rational number is 0
    numerator = 0;
    denominator = 1;
}

Rational::Rational(long i)
{
    numerator = i;
    denominator = 1;
}

Rational::Rational(long num, long denom)
{
    numerator = num;
    denominator = denom;
}
```

Then when we declare an object to be a *Rational*, the appropriate constructor is invoked. The declaration

```
Rational r;
```

automatically initializes *r* to the rational zero (0/1) since that is what the constructor *Rational* does with no parameters. The declaration

```
Rational r(3);
```

sets *r* to the rational 3/1, since it invokes the second version of the constructor. Finally, the declaration

```
Rational r(2,5);
```

sets *r* to the rational 2/5, invoking the third version of *Rational*, with two parameters.

The operator ***new*** in C++, applied to a type designator, allocates a new object of the given type and returns a pointer to it. When ***new*** is called, the constructor is also invoked automatically. Thus the statement

```
Rational *p = new Rational;
```

declares a pointer variable *p*, allocates a new object of type *Rational*, initializes it to 0 (since that is what the constructor with no arguments does), and sets *p* to point to the object.

The statement

```
Rational *p = new Rational(2,5);
```

sets *p* to point to a newly allocated *Rational* object with value 2/5.

We can also use the constructor in a statement such as

```
r = Rational(7);
```

which sets *r* to the rational number 7/1.

EXERCISES

1.4.1. Write a method *negate* for the class *Rational* that returns the negative of a rational number.

1.4.2. Write a method *subtract* for the class *Rational* that returns the result of subtracting one rational number from another.

1.4.3. Define a class *String* that represents a string by a length and a pointer to a string of characters.

(a) Write a constructor for *String* to allocate appropriate storage for it and to initialize it to a given C string. To allocate storage for an array of characters of length *N*, use the C++ operation

```
new char[N]
```

(b) Write a constructor for *String* to allocate storage of a given size for the string but not to initialize its characters.

(c) Write a method *concat* to concatenate one *String* with another.

1.4.4. Rewrite the routines of this section to use the constructors *Rational* rather than the method *setrational*.

2

The Stack

One of the most useful concepts in computer science is that of the stack. In this chapter we shall examine this deceptively simple data structure and see why it plays such a prominent role in the areas of programming and programming languages. We shall define the abstract concept of a stack and show how that concept can be made into a concrete and valuable tool in problem solving.

2.1 DEFINITION AND EXAMPLES

A ***stack*** is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the ***top*** of the stack. We can picture a stack as in Figure 2.1.1.

Unlike that of the array, the definition of the stack provides for the insertion and deletion of items, so that a stack is a dynamic, constantly changing object. The question therefore arises, how does a stack change? The definition specifies that a single end of the stack is designated as the stack top. New items may be put on top of the stack (in which case the top of the stack moves upward to correspond to the new highest element), or items which are at the top of the stack may be removed (in which case the top of the stack moves downward to correspond to the new highest element). To answer the question, which way is up? we must decide which end of the stack is designated as its top—that is, at which end items are added or deleted. By drawing Figure 2.1.1 so that F is physically higher on the page than all the other items in the stack, we imply that F is the current top element of the stack. If any new items are added to the stack, they are placed on top of F, and if any items are deleted, F is the first to be deleted.

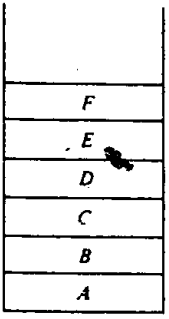


Figure 2.1.1 Stack containing stack terms.

This is also indicated by the vertical lines that extend past the items of the stack in the direction of the stack top.

Figure 2.1.2 is a motion picture of a stack as it expands and shrinks with the passage of time. Figure 2.1.2 a shows the stack as it exists at the time of the snapshot of Figure 2.1.1. In Figure 2.1.2 b, item *G* is added to the stack. According to the definition, there is only one place on the stack where it can be placed—on the top. The top element on the stack is now *G*. As the motion picture progresses through frames c, d, and e, items *I* and *J* are successively added onto the stack. Notice that the last item inserted (in this case *J*) is at the top of the stack. Beginning with frame f, however, the stack begins to shrink, as first *J*, then *I*, *H*, *G*, and *F* are successively removed. At each point, the top element is removed, since a deletion can be made only from the top. Item *G* could not be removed from the stack before items *J*, *I*, and *H* were gone. This illustrates the most important attribute of a stack, that the last element inserted into a stack is the first element deleted. Thus *J* is deleted before *I* because *J* was inserted after *I*. For this reason a stack is sometimes called a last-in, first-out (or LIFO) list.

Between frames j and k the stack has stopped shrinking and begins to expand again as item *K* is added. However, this expansion is short-lived, as the stack then shrinks to only three items in frame n.

Note that there is no way to distinguish between frame a and frame i by looking at the stack's state at the two instances. In both cases the stack contains the identical items in the same order and has the same stack top. No record is kept on the stack of the fact that four items had been pushed and popped in the meantime. Similarly, there is no way to distinguish between frames d and f, or j and l. If a record is needed of the intermediate items having been on the stack, that record must be kept elsewhere; it does not exist within the stack itself.

In fact, we have actually taken an extended view of what is really observed in a stack. The true picture of a stack is given by a view from the top looking down, rather than from a side looking in. Thus, in Figure 2.1.2, there is no perceptible difference between frames h and o. In each case the element at the top is *G*. Although the stack at frame h and the stack at frame o are not equal, the only way to determine this is to remove all the elements on both stacks and compare them individually. Although we have been looking at cross sections of stacks to make our understanding clearer, it should be noted that this is an added liberty, and there is no real provision for taking such a picture.

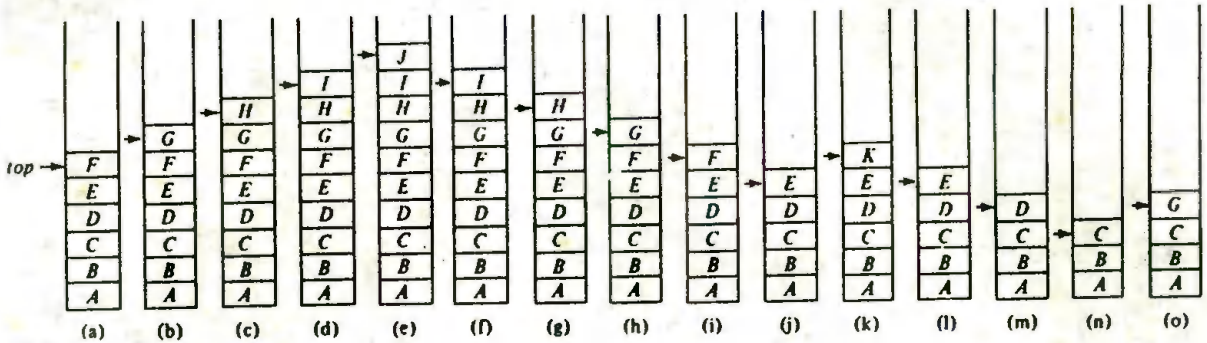


Figure 2.1.2 Motion picture of a stack.

Primitive Operations

The two changes which can be made to a stack are given special names. When an item is added to a stack, it is ***pushed*** onto the stack, and when an item is removed, it is ***popped*** from the stack. Given a stack *s*, and an item *i*, performing the operation *push* (*s*, *i*) adds the item *i* to the top of stack *s*. Similarly, the operation *pop*(*s*) removes the top element and returns it as a function value. Thus the assignment operation

```
i = pop(s);
```

removes the element at the top of *s* and assigns its value to *i*.

For example, if *s* is the stack of Figure 2.1.2, we performed the operation *push* (*s*, *G*) in going from frame a to frame b. We then performed, in turn, the following operations:

```
push (s,H);     (frame (c))
push (s,I);     (frame (d))
push (s,J);     (frame (e))
pop (s);        (frame (f))
pop (s);        (frame (g))
pop (s);        (frame (h))
pop (s);        (frame (i))
pop (s);        (frame (j))
push (s,K);     (frame (k))
pop (s);        (frame (l))
pop (s);        (frame (m))
pop (s);        (frame (n))
push (s,G);     (frame (o))
```

Because of the push operation which adds elements to a stack, a stack is sometimes called a ***pushdown list***.

There is no upper limit on the number of items that may be kept in a stack, since the definition does not specify how many items are allowed in the collection. Pushing another item onto a stack merely produces a larger collection of items. However, if a stack contains a single item and the stack is popped, the resulting stack contains no items and is called the ***empty stack***. Although the *push* operation is applicable to any stack, the *pop* operation cannot be applied to the empty stack because such a stack has no elements to pop. Therefore, before applying the *pop* operator to a stack, we must ensure that the stack is not empty. The operation *empty*(*s*) determines whether or not a stack *s* is empty. If the stack is empty, *empty*(*s*) returns the value *TRUE*; otherwise it returns the value *FALSE*.

Another operation that can be performed on a stack is to determine what the top item on a stack is without removing it. This operation is written *stacktop*(*s*) and returns the top element of stack *s*. The operation *stacktop*(*s*) is not really a new operation, since it can be decomposed into a pop and a push.

```
i = stacktop(s);
```

is equivalent to

```
i = pop(s);
push (s,i);
```

Like the operation *pop*, *stacktop* is not defined for an empty stack. The result of an illegal attempt to pop or access an item from an empty stack is called ***underflow***. Underflow can be avoided by ensuring that *empty*(*s*) is false before attempting the operation *pop*(*s*) or *stacktop*(*s*).

Example

Now that we have defined a stack and have indicated the operations that can be performed on it, let us see how we may use the stack in problem solving. Consider a mathematical expression that includes several sets of nested parentheses; for example,

```
7 - ((X * ((X + Y) / (J - 3)) + Y) / (4 - 2.5))
```

We want to ensure that the parentheses are nested correctly; that is, we want to check that

1. There are an equal number of right and left parentheses.
2. Every right parenthesis is preceded by a matching left parenthesis.

Expressions such as

$$((A + B) \quad \text{or} \quad A + B($$

violate condition 1, and expressions such as

$$)A + B(-C \quad \text{or} \quad (A + B)) - (C + D$$

violate condition 2.

To solve this problem, think of each left parenthesis as opening a scope and each right parenthesis as closing a scope. The ***nesting depth*** at a particular point in an expression is the number of scopes that have been opened but not yet closed at that point. This is the same as the number of left parentheses encountered whose matching right parentheses have not yet been encountered. Let us define the ***parenthesis count*** at a particular point in an expression as the number of left parentheses minus the number of right parentheses that have been encountered in scanning the expression from its left end up to that particular point. If the parenthesis count is nonnegative, it is the same as the nesting depth. The two conditions that must hold if the parentheses in an expression form an admissible pattern are as follows:

1. The parenthesis count at the end of the expression is 0. This implies that no scopes have been left open or that exactly as many right parentheses as left parentheses have been found.
2. The parenthesis count at each point in the expression is nonnegative. This implies that no right parenthesis is encountered for which a matching left parenthesis had not previously been encountered.

```
7 - ( ( X * ( ( Y + Y ) / ( 3 - 3 ) ) + Y ) / ( 4 - 2.5 ) )
0 0 1 2 2 2 3 4 4 4 4 3 3 4 4 4 4 3 2 2 2 2 1 1  2 2 2  2  1 0

                    ( ( A + B )
                    1 2 2 2 2 1

                    A + B  (
                    0 0 0  1

                  )  A  +  B ( - C
                 -1 -1 -1 -1 0 0 0

               ( A + B )  )  - ( C + D
               1 1 1 1 0 -1 -1 0 0 0 0
```

Figure 2.1.3 Parenthesis count at various points of strings.

In Figure 2.1.3 the count at each point in each of the previous five strings is given directly below that point. Since only the first string meets the foregoing two conditions, it is the only one among the five with a correct parentheses pattern.

Let us now change the problem slightly and assume that three different types of scope delimiters exist. These types are indicated by parentheses ((and)), brackets ([and]), and braces ({ and }). A scope ender must be of the same type as its scope opener. Thus, strings such as

$$(A + B], [(A + B]), \{A - (B]\}$$

are illegal.

It is necessary to keep track of not only how many scopes have been opened but also of their types. This information is needed because when a scope ender is encountered, we must know the symbol with which the scope was opened to ensure that it is being closed properly.

A stack may be used to keep track of the types of scopes encountered. Whenever a scope opener is encountered, it is pushed onto the stack. Whenever a scope ender is encountered, the stack is examined. If the stack is empty, the scope ender does not have a matching opener and the string is therefore invalid. If, however, the stack is nonempty, we pop the stack and check whether the popped item corresponds to the scope ender. If a match occurs, we continue. If it does not, the string is invalid. When the end of the string is reached, the stack must be empty; otherwise one or more scopes have been opened which have not been closed, and the string is invalid. The algorithm for this procedure follows. Figure 2.1.4 shows the state of the stack after reading in parts of the string $\{x + (y - [a + b]) * c - [(d + e)]\}/(h - (j - (k - [l - n])))$.

```
valid = true;            /* assume the string is valid */
s = the empty stack;
while (we have not read the entire string) {
   read the next symbol (symb) of the string;
   if (symb == '(' || symb == '[' || symb == '{')
      push(s,symb);
```

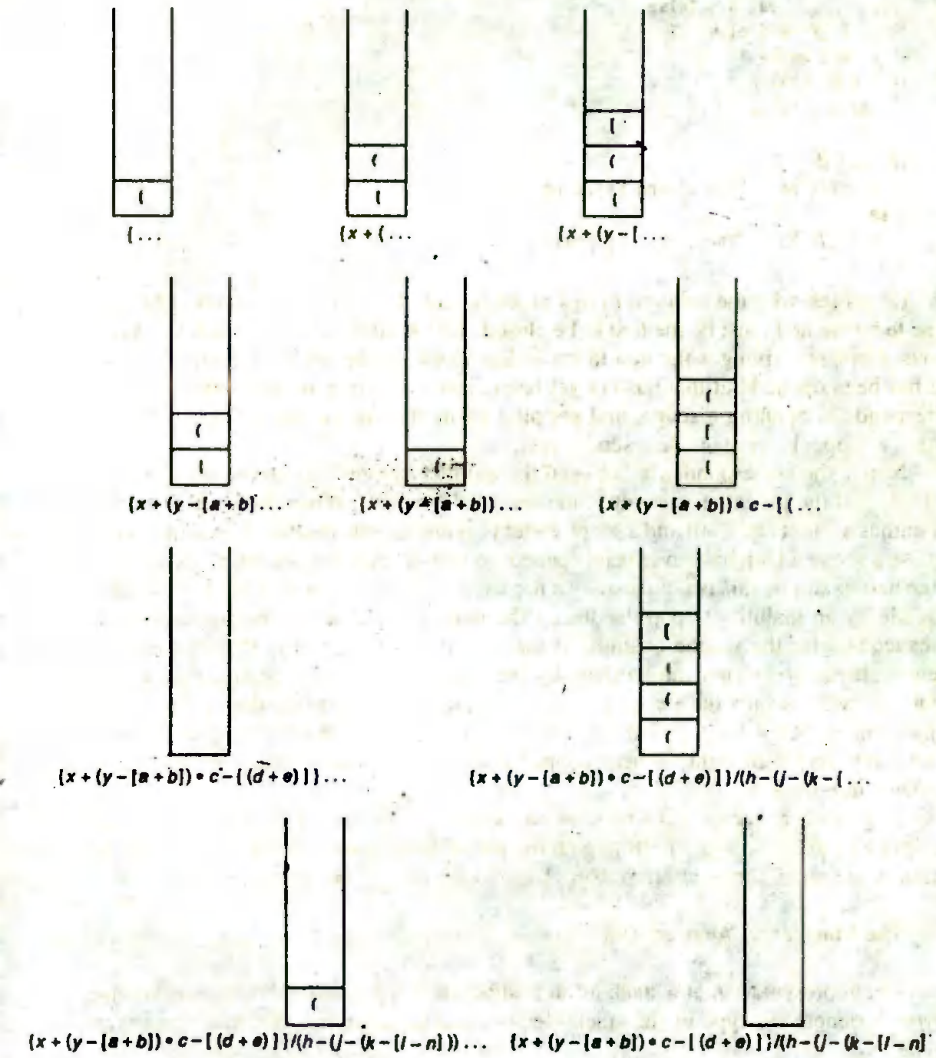


Figure 2.1.4 Parenthesis stack at various stages of processing.

```
    if (symb == ')' || symb == ']' || symb == '{')
       if (empty(s))
          valid = false;
       else {
          i = pop(s);
          if (i is not the matching opener of symb)
             valid = false;
       } /* end else */
} /* end while */
if (!empty(s))
   valid = false;

if (valid)
   printf("%s", "the string is valid");
else
   printf("%s", "the string is invalid");
```

Let us see why the solution to this problem calls for the use of a stack. The last scope to be opened must be the first to be closed. This is simulated by a stack in which the last element arriving is the first to leave. Each item on the stack represents a scope that has been opened but that has not yet been closed. Pushing an item onto the stack corresponds to opening a scope, and popping an item from the stack corresponds to closing a scope, leaving one less scope open.

Notice the correspondence between the number of elements on the stack in this example and the parenthesis count in the previous example. When the stack is empty (parenthesis count equals 0) and a scope ender is encountered, an attempt is being made to close a scope which has never been opened; so that the parenthesis pattern is invalid. In the first example, this is indicated by a negative parenthesis count, and in the second example by an inability to pop the stack. The reason that a simple parenthesis count is inadequate for the second example is that we must keep track of the actual scope openers themselves. This can be done by the use of a stack. Notice also that at any point, we examine only the element at the top of the stack. The particular configuration of parentheses below the top element is irrelevant while examining this top element. It is only after the top element has been popped that we concern ourselves with subsequent elements in a stack.

In general a stack can be used in any situation that calls for a last-in, first-out discipline or that displays a nesting pattern. We shall see more examples of the use of stacks in the remaining sections of this chapter and, indeed, throughout the text.

The Stack as an Abstract Data Type

The representation of a stack as an abstract data type is straightforward. We use *eltype* to denote the type of the stack element and parameterize the stack type with eltype.

```
abstract typedef <<eltype>> STACK (eltype);
```

```
abstract empty(s)
STACK(eltype) s;
postcondition      empty == (len(s) == 0);

abstract eltype pop(s)
STACK(eltype) s;
precondition       empty(s) == FALSE;
postcondition      pop == first(s');
                   s == sub(s', 1, len(s') - 1);

abstract push(s, elt)
STACK(eltype) s;
eltype elt;
postcondition      s == <elt> + s';
```

EXERCISES

2.1.1. Use the operations *push*, *pop*, *stacktop*, and *empty* to construct operations which do each of the following.

- (a) Set *i* to the second element from the top of the stack, leaving the stack without its top two elements.
- (b) Set *i* to the second element from the top of the stack, leaving the stack unchanged.
- (c) Given an integer *n*, set *i* to the *n*th element from the top of the stack, leaving the stack without its top *n* elements.
- (d) Given an integer *n*, set *i* to the *n*th element from the top of the stack, leaving the stack unchanged.
- (e) Set *i* to the bottom element of the stack, leaving the stack empty.
- (f) Set *i* to the bottom element of the stack, leaving the stack unchanged. (*Hint:* Use another, auxiliary stack.)
- (g) Set *i* to the third element from the bottom of the stack.

2.1.2. Simulate the action of the algorithm in this section for each of the following strings by showing the contents of the stack at each point.

- (a) $(A + B)$
- (b) $\{[A + B] - [(C - D)]\}$
- (c) $(A + B) - \{C + D\} - [F + G]$
- (d) $((H) * \{([J + K])\})$
- (e) $(((A)))$

2.1.3. Write an algorithm to determine if an input character string is of the form

$$x \quad C \quad y$$

where x is a string consisting of the letters 'A' and 'B', and where y is the reverse of x (that is, if x = "ABABBA," y must equal "ABBABA"). At each point you may read only the next character of the string.

2.1.4. Write an algorithm to determine if an input character string is of the form

$$a\ D\ b\ D\ c\ D \ldots D\ z$$

where each string *a*, *b*, . . . , *z* is of the form of the string defined in Exercise 2.1.3. (Thus a string is in the proper form if it consists of any number of such strings separated by the character 'D'.) At each point you may read only the next character of the string.

2.1.5. Design an algorithm that does not use a stack to read a sequence of *push* and *pop* operations, and determine whether or not underflow occurs on some *pop* operation. Implement the algorithm as a C program.

2.1.6. What set of conditions are necessary and sufficient for a sequence of *push* and *pop* operations on a single stack (initially empty) to leave the stack empty and not cause underflow? What set of conditions are necessary for such a sequence to leave a nonempty stack unchanged?

2.2 REPRESENTING STACKS IN C

Before programming a problem solution that uses a stack, we must decide how to represent a stack using the data structures that exist in our programming language. As we shall see, there are several ways to represent a stack in C. We now consider the simplest of these. Throughout this text, you will be introduced to other possible representations. Each of them, however, is merely an implementation of the concept introduced in Section 2.1. Each has advantages and disadvantages in terms of how close it comes to mirroring the abstract concept of a stack and how much effort must be made by the programmer and the computer in using it.

A stack is an ordered collection of items, and C already contains a data type that is an ordered collection of items: the array. Whenever a problem solution calls for the use of a stack, therefore, it is tempting to begin a program by declaring a variable *stack* as an array. However, a stack and an array are two entirely different things. The number of elements in an array is fixed and is assigned by the declaration for the array. In general, the user cannot change this number. A stack, on the other hand, is fundamentally a dynamic object whose size is constantly changing as items are popped and pushed.

However, although an array cannot be a stack, it can be the home of a stack. That is, an aray can be declared large enough for the maximum size of the stack. During the course of program execution, the stack can grow and shrink within the space reserved for it. One end of the array is the fixed bottom of the stack, while the top of the stack constantly shifts as items are popped and pushed. Thus, another field is needed that, at each point during program execution, keeps track of the current position of the top of the stack.

A stack in C may therefore be declared as a structure containing two objects: an array to hold the elements of the stack, and an integer to indicate the position of the current stack top within the array. This may be done for a stack of integers by the declarations

```
#define STACKSIZE 100
struct stack {
    int top;
    int items[STACKSIZE];
};
```

Once this has been done, an actual stack *s* may be declared by

```
struct stack s;
```

Here, we assume that the elements of the stack *s* contained in the array *s.items* are integers and that the stack will at no time contain more than *STACKSIZE* integers. In this example *STACKSIZE* is set to 100 to indicate that the stack can contain 100 elements (*items*[0] through *items*[99]).

There is, of course, no reason to restrict a stack to contain only integers; *items* could just as easily have been declared as *float items*[*STACKSIZE*] or *char items*[*STACKSIZE*], or whatever other type we might wish to give to the elements of the stack. In fact, should the need arise, a stack can contain objects of different types by using C unions. Thus

```
#define STACKSIZE 100
#define INT     1
#define FLOAT   2
#define STRING  3
struct stackelement {
   int etype;  /* etype equals INT, FLOAT, or STRING   */
               /*  depending on the type of the        */
               /*     corresponding element.           */
   union {
     int   ival;
     float fval;
     char *pval;  /* pointer to a string */
   } element;
};
struct stack {
   int top;
   struct stackelement items[STACKSIZE];
};
```

defines a stack whose items may be either integers, floating-point numbers, or strings, depending on the value of the corresponding *etype*. Given a stack *s* declared by

```
struct stack s;
```

we could print the top element of the stack as follows:

```
struct stackelement se;
        . . .
se = s.items[s.top];
switch (se.etype)  {
   case INTGR  : printf("% d\n", se.ival);
   case FLT    : printf("% f\n", se.fval);
   case STRING : printf("% s\n", se.pval);
}  /*end switch  */
```

For simplicity, in the remainder of this section we assume that a stack is declared to have only homogeneous elements (so that unions are not necessary). The identifier *top* must always be declared as an integer, since its value represents the position within the array *items* of the topmost stack element. Therefore, if the value of *s.top* is 4, there are five elements on the stack: *s.items*[0], *s.items*[1], *s.items*[2], *s.items*[3], and *s.items*[4]. When the stack is popped, the value of *s.top* is changed to 3 to indicate that there are now only four elements on the stack and that *s.items*[3] is the top element. On the other hand, if a new object is pushed onto the stack, the value of *s.top* must be increased by 1 to 5 and the new object inserted into *s.items*[5].

The empty stack contains no elements and can therefore be indicated by *top* equalling -1. To initialize a stack *s* to the empty state, we may initially execute *s.top* = -1;.

To determine, during the course of execution, whether or not a stack is empty the condition *s.top* = = -1 may be tested in an *if* statement as follows:

```
if (s.top == -1)
   /* stack is empty */
else
   /* stack is not empty */
```

This test corresponds to the operation *empty*(*s*) that was introduced in Section 2.1. Alternatively, we may write a function that returns *TRUE* if the stack is empty and *FALSE* if it is not empty, as follows:

```
int empty(struct stack *ps)
{
  if (ps->top == -1)
    return(TRUE);
  else
    return(FALSE);
} /* end empty */
```

Once this function exists, a test for the empty stack is implemented by the statement

```
if (empty (&s))
   /* stack is empty */
else
   /* stack is not empty */
```

Note the difference between the syntax of the call to *empty* in the algorithm of Section 2.1 and in the program segment here. In the algorithm, *s* represented a stack and the call to *empty* was expressed as

```
empty(s)
```

In this section, we are concerned with the actual implementation of the stack and its operations. Since parameters in C are passed by value, the only way to modify the

argument passed to a function is to pass the address of the argument rather than the argument itself. Further, the original definition of C (by Kernighan–Ritchie) and many older C compilers do not allow a structure to be passed as an argument even if its value remains unchanged. (Although this restriction has been omitted in ANSI C, it is generally more efficient to pass a pointer when the structure is large.) Thus in functions such as *pop* and *push* (which modify their structure arguments), as well as *empty* (which does not), we adopt the convention that we pass the address of the stack structure, rather than the stack itself.

You may wonder why we bother to define the function *empty* when we could just as easily write *if s.top* = = −1 each time that we want to test for the empty condition. The answer is that we wish to make our programs more comprehensible and to make the use of a stack independent of its implementation. Once we understand the stack concept, the phrase "*empty*(*&s*)" is more meaningful than the phrase "*s.top* = = −1." If we should later introduce a better implementation of a stack, so that "*s.top* = = −1" becomes meaningless, we would have to change every reference to the field identifier *s.top* throughout the entire program. On the other hand, the phrase "*empty*(*&s*)" would still retain its meaning, since it is an inherent attribute of the stack concept rather than of an implementation of that concept. All that would be required to revise a program to accommodate a new implementation of the stack would be a possible revision of the declaration of the structure *stack* in the main program and the rewriting of the function *empty*. (It is also possible that the form of the call to *empty* would have to be modified so that it does not use an address.)

Aggregating the set of implementation-dependent trouble spots into small, easily identifiable units is an important method of making a program more understandable and modifiable. This concept is known as ***modularization***, in which individual functions are isolated into low-level ***modules*** whose properties are easily verifiable. These low-level modules can then be used by more complex routines, which do not have to concern themselves with the details of the low-level modules but only with their function. The complex routines may themselves then be viewed as modules by still higher-level routines that use them independently of their internal details.

A programmer should always be concerned with the readability of the code he or she produces. A small amount of attention to clarity will save a large amount of time in debugging. Large- and medium-sized programs will almost never be correct the first time they are run. If precautions are taken at the time that a program is written to ensure that it is easily modifiable and comprehensible, the total time needed to get the program to run correctly is reduced sharply. For example, the *if* statement in the *empty* function could be replaced by the shorter, more efficient statement

```
return (ps->top == -1);
```

This statement is precisely equivalent to the longer statement

```
if (ps->top == -1)
     return(TRUE);
else return(FALSE);
```

This is because the value of the expression $ps->top == -1$ is *TRUE* if and only if the condition $ps->top == -1$ is *TRUE*. However, someone who reads a program will probably be much more comfortable reading the *if* statement. Often you will find that if you use "tricks" of the language in writing programs, you will be unable to decipher your own programs after putting them aside for a day or two.

Although it is true that the C programmer is often concerned with economy of code, it is also important to consider the time that will no doubt be spent in debugging. The mature professional (whether in C or other language) is constantly concerned with the proper balance between code economy and code clarity.

Implementing the *pop* Operation

The possibility of underflow must be considered in implementing the *pop* operation, since the user may inadvertently attempt to pop an element from an empty stack. Of course, such an attempt is illegal and should be avoided. However, if such an attempt should be made the user should be informed of the underflow condition. We therefore introduce a function *pop* that performs the following three actions:

1. If the stack is empty, print a warning message and halt execution.
2. Remove the top element from the stack.
3. Return this element to the calling program.

We assume that the stack consists of integers, so that the *pop* operation can be implemented as a function. This would also be the case if the stack consisted of some other type of simple variable. However, if a stack consists of a more complex structure (for example, a structure or a union), the *pop* operation would either be implemented as returning a pointer to a data element of the proper type (rather than the data element itself), or the operation would be implemented with the popped value as a parameter (in which case the address of the parameter would be passed rather than the parameter, so that the *pop* function could modify the actual argument).

```
int pop(struct stack *ps)
{
  if (empty(ps)) {
      printf("%", "stack underflow");
      exit(1);
  } /* end if */
  return(ps->items[ps->top--]);
} /* end pop */
```

Note that *ps* is already a pointer to a structure of type stack; therefore, the address operator "&" is not used in calling *empty*. In all applications in C, one must always distinguish between pointers and actual data objects.

Let us look at the *pop* function more closely. If the stack is not empty, the top element of the stack is retained as the returned value. This element is then removed from the stack by the expression $ps -> top--$. Assume that when pop is called,

ps -> *top* equals 87; that is, there are 88 items on the stack. The value of *ps* -> *items*[87] is returned, and the value of *ps* -> *top* is changed to 86. Note that *ps* -> *items*[87] still retains its old value; the array *ps* -> *items* remains unchanged by the call to *pop*. However, the stack is modified, since it now contains only 87 elements rather than 88. Recall that an array and a stack are two different objects. The array only provides a home for the stack. The stack itself contains only those elements between the zeroth element of the array and the *top*th element. Thus reducing the value of *ps* -> *top* by 1 effectively removes an element from the stack. This is true despite the fact that *ps* -> *items*[87] retains its old value.

To use the *pop* function, the programmer can declare *int x* and write

```
x = pop (&s);
```

x then contains the value popped from the stack. If the intent of the *pop* operation is not to retrieve the element on the top of the stack but only to remove it from the stack, the value of *x* will not be used again in the program.

Of course, the programmer should ensure that the stack is not empty when the *pop* operation is called. If the programmer is unsure of the state of the stack, its status may be determined by coding

```
if (!empty(&s))
   x = pop (&s);
else
   /* take remedial action */
```

If the programmer unwittingly does call *pop* with an empty stack, the function prints the error message *stack underflow* and execution halts. Although this is an unfortunate state of affairs, it is far better than what would occur had the *if* statement in the *pop* routine been omitted entirely. In that case, the value of *s.top* would be -1 and an attempt would be made to access the nonexistent element *s.items*[-1].

A programmer should always provide for the almost certain possibility of error. This can be done by including diagnostics that are meaningful in the context of the problem. By doing so, if and when an error does occur, the programmer is able to pinpoint its source and take corrective action immediately.

Testing for Exceptional Conditions

Within the context of a given problem, it may not be necessary to halt execution immediately upon the detection of underflow. Instead, it might be more desirable for the *pop* routine to signal the calling program that an underflow has occurred. The calling routine, upon detecting this signal, can take corrective action. Let us call the procedure that pops the stack and returns an indication whether underflow has occurred, *popandtest*:

```
void popandtest(struct stack *ps, int *px, int *pund)
{
   if (empty(ps)) {
      *pund = TRUE;
      return;
   } /* end if */
   *pund = FALSE;
   *px = ps->items[ps->top--];
   return;
} /* end popandtest */
```

In the calling program the programmer would write

```
popandtest(&s, &x, &und);
if (und)
   /* take corrective action */
else
   /* use value of x */
```

Implementing the Push Operation

Let us now examine the *push* operation. It seems that this operation should be quite easy to implement using the array representation of a stack. A first attempt at a *push* procedure might be the following:

```
void push(struct stack *ps, int x)
{
   ps->items[++(ps->top)] = x;
   return;
} /* end push */
```

This routine makes room for the item x to be pushed onto the stack by incrementing *s.top* by 1, and then inserts x into the array *s. items*.

The routine directly implements the *push* operation introduced in Section 2.1. Yet, as it stands, it is quite incorrect. It allows a subtle error to creep in, caused by using the array representation of the stack. Recall that a stack is a dynamic structure that is constantly allowed to grow and shrink and thus change its size. An array, on the other hand, is a fixed object of predetermined size. Thus, it is quite conceivable that a stack may outgrow the array that was set aside to contain it. This occurs when the array is full, that is, when the stack contains as many elements as the array and an attempt is made to push yet another element onto the stack. The result of such an attempt is called an ***overflow***.

Assume that the array *s.items* is full and that the C *push* routine is called. Remember that the first array position is 0 and the arbitrary size (*STACKSIZE*) chosen for the array *s.items* is 100. The full array is then indicated by the condition *s.top* == 99, so that position 99 (the 100th element of the array) is the current top of the stack.

When *push* is called, *s.top* is increased to 100 and an attempt is made to insert *x* into *s.items*[100]. Of course, the upper bound of *s.items* is 99, so that this attempt at insertion results in an unpredictable error, depending on the contents of the memory location following the last array position. An error message may be produced that is unlikely to relate to the cause of the error.

The *push* procedure must therefore be revised so that it reads as follows:

```
void push(struct stack *ps, int x)
{
   if (ps->top == STACKSIZE-1) {
      printf("%s", "stack overflow");
      exit(1);
   }
   else
      ps->items[++(ps->top)] = x;
   return;
} /* end push */
```

Here, we check whether the array is full before attempting to push another element onto the stack. The array is full if $ps \rightarrow top == stacksize - 1$.

You should again note that if and when overflow is detected in *push*, execution halts immediately after an error message is printed. This action, as in the case of *pop*, may not be the most desirable. It might, in some cases, make more sense for the calling routine to invoke the *push* operation with the instructions

```
pushandtest(&s, x, & overflow);
if (overflow)
   /* overflow has been detected, x was not  */
   /* pushed on stack. take remedial action. */
else
   /* x was successfully pushed on the stack */
   /*          continue processing.          */
```

This allows the calling program to proceed after the call to *pushandtest*, whether or not overflow was detected. The subroutine *pushandtest* is left as an exercise for the reader.

Although the overflow and underflow conditions are treated similarly in *push* and *pop*, there is a fundamental difference between them. Underflow indicates that the *pop* operation cannot be performed on the stack and may indicate an error in the algorithm or the data. No other implementation or representation of the stack will cure the underflow condition. Rather, the entire problem must be rethought. (Of course, an underflow might occur as a signal for ending one process and beginning another. But in such a case *popandtest* rather than *pop* should be used.)

Overflow, however, is not a condition that is applicable to a stack as an abstract data structure. Abstractly, it is always possible to push an element onto a stack. A stack is just an ordered set, and there is no limit to the number of elements that such a set can contain. The possibility of overflow is introduced when a stack is implemented by an array with only a finite number of elements, thereby prohibiting the growth of the stack

beyond that number. It may very well be that the algorithm that the programmer used is correct, just that the implementation of the algorithm did not anticipate that the stack would become so large. Thus, in some cases an overflow condition can be corrected by changing the value of the constant *STACKSIZE* so that the array field *items* contains more elements. There is no need to change the routines *pop* or *push*, since they refer to whatever data structure was declared for the type *stack* in the program declarations. *push* also refers to the constant *STACKSIZE*, rather than to the actual value 100.

However, more often than not, an overflow does indicate an error in the program that cannot be attributed to a simple lack of space. The program may be in an infinite loop in which items are constantly being pushed onto the stack and nothing is ever popped. Thus the stack will outgrow the array bound no matter how high that bound is set. The programmer should always check that this is not the case before indiscriminately raising the array bound. Often the maximum stack size can be determined easily from the program and its inputs, so that if the stack does overflow there is probably something wrong with the algorithm that the program represents.

Let us now look at our last operation on stacks, *stacktop*(*s*), which returns the top element of a stack without removing it from the stack. As noted in the last section, *stacktop* is not really a primitive operation because it can be decomposed into the two operations:

```
x = pop(s);
push (s,x);
```

However, this is a rather awkward way to retrieve the top element of a stack. Why not ignore the decomposition noted above and directly retrieve the proper value? Of course, a check for the empty stack and underflow must then be explicitly stated, since the test is no longer handled by a call to *pop*.

We present a C function *stacktop* for a stack of integers as follows:

```
int stacktop(struct stack *ps)
{
   if (empty(ps)) {
      printf("%s", "stack underflow");
      exit(1);
   }
   else
      return(ps->items[ps->top]);
} /*end stacktop */
```

You may wonder why we bother writing a separate routine *stacktop* when a reference to *s.items*[*s.top*] would serve just as well. There are several reasons for this. First, the routine *stacktop* incorporates a test for underflow, so that no mysterious error occurs if the stack is empty. Second, it allows the programmer to use a stack without worrying about its internal makeup. Third, if a different implementation of a stack is introduced, the programmer need not comb through all the places in the program that refer to *s.items*[*s.top*] to make those references compatible with the new implementation. Only the *stacktop* routine would need to be changed.

EXERCISES

2.2.1. Write C functions that use the routines presented in this chapter to implement the operations of Exercise 2.1.1.

2.2.2. Given a sequence of *push* and *pop* operations and an integer representing the size of an array in which a stack is to be implemented, design an algorithm to determine whether or not overflow occurs. The algorithm should not use a stack. Implement the algorithm as a C program.

2.2.3. Implement the algorithms of Exercises 2.1.3 and 2.1.4 as C programs.

2.2.4. Show how to implement a stack of integers in C by using an array *int s*[*STACKSIZE*], where *s*[0] is used to contain the index of the top element of the stack, and where *s*[1] through *s*[*STACKSIZE* − 1] contain the elements on the stack. Write a declaration and routines *pop*, *push*, *empty*, *popandtest*, *stacktop*, and *pushandtest* for this implementation.

2.2.5. Implement a stack in C in which each item on the stack is a varying number of integers. Choose a C data structure for such a stack and design *push* and *pop* routines for it.

2.2.6. Consider a language that does not have arrays but does have stacks as a data type. That is, one can declare

```
stack s;
```

and the *push*, *pop*, *popandtest*, and *stacktop* operations are defined. Show how a one-dimensional array can be implemented by using these operations on two stacks.

2.2.7. Design a method for keeping two stacks within a single linear array *S*[*spacesize*] so that neither stack overflows until all of memory is used and an entire stack is never shifted to a different location within the array. Write C routines *push*1, *push*2, *pop*1 and *pop*2 to manipulate the two stacks. (*Hint:* The two stacks grow toward each other.)

2.2.8. The Bashemin Parking Garage contains a single lane that holds up to ten cars. There is only a single entrance/exit to the garage at one end of the lane. If a customer arrives to pick up a car that is not nearest the exit, all cars blocking its path are moved out, the customer's car is driven out, and the other cars are restored in the same order that they were in originally. Write a program that processes a group of input lines. Each input line contains an 'A' for arrival or a 'D' for departure, and a license plate number. Cars are assumed to arrive and depart in the order specified by the input. The program should print a message whenever a car arrives or departs. When a car arrives, the message should specify whether or not there is room for the car in the garage. If there is no room, the car leaves without entering the garage. When a car departs, the message should include the number of times that the car was moved out of the garage to allow other cars to depart.

2.3 EXAMPLE: INFIX, POSTFIX, AND PREFIX

Basic Definitions and Examples

This section examines a major application that illustrates the different types of stacks and the various operations and functions defined upon them. The example is also an important topic of computer science in its own right.

Consider the sum of A and B. We think of applying the ***operator*** "+" to the ***operands*** A and B and write the sum as $A + B$. This particular representation is called ***infix***. There are two alternate notations for expressing the sum of A and B using the symbols A, B, and +. These are

```
+ A B     prefix
A B +     postfix
```

The prefixes "pre-," "post-," and "in-" refer to the relative position of the operator with respect to the two operands. In prefix notation the operator precedes the two operands, in postfix notation the operator follows the two operands, and in infix notation the operator is between the two operands. The prefix and postfix notations are not really as awkward to use as they might at first appear. For example, a C function to return the sum of the two arguments A and B is invoked by *add*(A, B). The operator *add* precedes the operands A and B.

Let us now consider some additional examples. The evaluation of the expression $A + B * C$, as written in standard infix notation, requires knowledge of which of the two operations, + or *, is to be performed first. In the case of + and * we "know" that multiplication is to be done before addition (in the absence of parentheses to the contrary). Thus $A + B * C$ is interpreted as $A + (B * C)$ unless otherwise specified. We say that multiplication takes ***precedence*** over addition. Suppose that we want to rewrite $A + B * C$ in postfix. Applying the rules of precedence, we first convert the portion of the expression that is evaluated first, namely the multiplication. By doing this conversion in stages we obtain

```
A + (B * C)     parentheses for emphasis
A + (BC *)      convert the multiplication
A (BC *) +      convert the addition
  ABC * +       postfix form
```

The only rules to remember during the conversion process are that operations with highest precedence are converted first and that after a portion of the expression has been converted to postfix it is to be treated as a single operand. Consider the same example with the precedence of operators reversed by the deliberate insertion of parentheses.

```
(A + B) * C     infix form
(AB +) * C      convert the addition
(AB +) C *      convert the multiplication
  AB + C *      postfix form
```

In this example the addition is converted before the multiplication because of the parentheses. In going from $(A + B) * C$ to $(AB+) * C$, A and B are the operands and + is the operator. In going from $(AB +) * C$ to $(AB +)C *$, $(AB +)$ and C are the operands and * is the operator. The rules for converting from infix to postfix are simple, providing that you know the order of precedence.

We consider five binary operations: addition, subtraction, multiplication, division, and exponentiation. The first four are available in C and are denoted by the usual op-

erators +, −, *, and /. The fifth, exponentiation, is represented by the operator $. The value of the expression *A* $ *B* is *A* raised to the *B* power, so that 3 $ 2 is 9. For these binary operators the following is the order of precedence (highest to lowest):

Exponentiation
Multiplication/division
Addition/subtraction

When unparenthesized operators of the same precedence are scanned, the order is assumed to be left to right except in the case of exponentiation, where the order is assumed to be from right to left. Thus *A* + *B* + *C* means (*A* + *B*) + *C*, whereas *A* $ *B* $ *C* means *A* $ (*B* $ *C*). By using parentheses we can override the default precedence.

We give the following additional examples of converting from infix to postfix. Be sure that you understand each of these examples (and can do them on your own) before proceeding to the remainder of this section.

Infix	Postfix
A + B	AB +
A + B − C	AB + C −
(A + B) * (C − D)	AB + CD − *
A $ B * C − D + E / F / (G + H)	AB $ C * D − EF / GH + / +
((A + B) * C − (D − E)) $ (F + G)	AB + C * DE − − FG + $
A − B/(C * D $ E)	ABCDE $ * / −

The precedence rules for converting an expression from infix to prefix are identical. The only change from postfix conversion is that the operator is placed before the operands rather than after them. We present the prefix forms of the foregoing expressions. Again, you should attempt to make the transformations on your own.

Infix	Prefix
A + B	+ AB
A + B − C	− + ABC
(A + B) * (C − D)	* + AB − CD
A $ B * C − D + E / F / (G + H)	+ − * $ ABCD / / EF + GH
((A + B) * C − (D − E)) $ (F + G)	$ − * + ABC − DE + FG
A − B/(C * D $ E)	− A / B * C $ DE

Note that the prefix form of a complex expression is not the mirror image of the postfix form, as can be seen from the second of the foregoing examples, *A* + *B* − *C*. We will henceforth consider only postfix transformations and leave to the reader as exercises most of the work involving prefix.

One point immediately obvious about the postfix form of an expression is that it requires no parentheses. Consider the two expressions *A* + (*B* * *C*) and (*A* + *B*) * *C*. Although the parentheses in one of the two expressions is superfluous [by convention *A* + *B* * *C* = *A* + (*B* * *C*)], the parentheses in the second expression are necessary to avoid confusion with the first. The postfix forms of these expressions are

Infix	Postfix
A + (B * C)	ABC * +
(A + B) * C	AB + C *

There are no parentheses in either of the two transformed expressions. The order of the operators in the postfix expressions determines the actual order of operations in evaluating the expression, making the use of parentheses unnecessary.

In going from infix to postfix we sacrifice the ability to note at a glance the operands associated with a particular operator. We gain, however, an unambiguous form of the original expression without the use of cumbersome parentheses. In fact, the postfix form of the original expression might look simpler were it not for the fact that it appears difficult to evaluate. For example, how do we know that if $A = 3$, $B = 4$, and $C = 5$ in the foregoing examples, then 3 4 5 * + equals 23 and 3 4 + 5 * equals 35?

Evaluating a Postfix Expression

The answer to the foregoing question lies in the development of an algorithm for evaluating expressions in postfix. Each operator in a postfix string refers to the previous two operands in the string. (Of course, one of these two operands may itself be the result of applying a previous operator.) Suppose that each time we read an operand we push it onto a stack. When we reach an operator, its operands will be the top two elements on the stack. We can then pop these two elements, perform the indicated operation on them, and push the result on the stack so that it will be available for use as an operand of the next operator. The following algorithm evaluates an expression in postfix using this method:

```
opndstk = the empty stack;
/* scan the input string reading one */
/* element at a time into symb       */
while (not end of input) {
   symb = next input character;
   if (symb is an operand)
      push(opndstk, symb);
   else {
      /* symb is an operator */
      opnd2 = pop(opndstk);
      opnd1 = pop(opndstk);
      value = result of applying symb to opnd1 and opnd2;
      push(opndstk, value);
   } /* end else */
} /* end while */
return(pop(opndstk));
```

Let us now consider an example. Suppose that we are asked to evaluate the following postfix expression:

6 2 3 + − 3 8 2 / + * 2 $ 3 +

We show the contents of the stack *opndstk* and the variables *symb*, *opnd1*, *opnd2*, and *value* after each successive iteration of the loop. The top of *opndstk* is to the right.

symb	*opnd1*	*opnd2*	*value*	*opndstk*
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
−	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
$	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

Each operand is pushed onto the operand stack as it is encountered. Therefore the maximum size of the stack is the number of operands that appear in the input expression. However, in dealing with most postfix expressions the actual size of the stack needed is less than this theoretical maximum, since an operator removes operands from the stack. In the previous example the stack never contained more than four elements, despite the fact that eight operands appeared in the postfix expression.

Program to Evaluate a Postfix Expression

There are a number of questions we must consider before we can actually write a program to evaluate an expression in postfix notation. A primary consideration, as in all programs, is to define precisely the form and restrictions, if any, on the input. Usually the programmer is presented with the form of the input and is required to design a program to accommodate the given data. On the other hand, we are in the fortunate position of being able to choose the form of our input. This enables us to construct a program that is not overburdened with transformation problems that overshadow the actual intent of the routine. Had we been confronted with data in a form that is awkward and cumbersome to work with, we could have relegated the transformations to various functions and used the output of these functions as input to our primary routine. In the "real world," recognition and transformation of input is a major concern.

Let us assume in this case that each input line is in the form of a string of digits and operator symbols. We assume that operands are single nonnegative digits, for example, 0, 1, 2, ..., 8, 9. For example, an input line might contain 3 4 5 * + in the first 5 columns followed by an end-of-line character ('\n'). We would like to write a program that reads input lines of this format, as long as there are any remaining, and prints for each line the original input string and the result of the evaluated expression.

Since the symbols are read as characters, we must find a method to convert the operand characters to numbers and the operator characters to operations. For example, we must have a method for converting the character '5' to the number 5 and the character '+' to the addition operation.

The conversion of a character to an integer can be handled easily in C. If *int x* is a single digit character in C, the expression x − '0' yields the numerical value of that digit. To implement the operation corresponding to an operator symbol, we use a function *oper* that accepts the character representation of an operator and two operands as input parameters, and returns the value of the expression obtained by applying the operator to the two operands. The body of the function will be presented shortly.

The body of the main program might be the following. The constant *MAXCOLS* is the maximum number of columns in an input line.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MAXCOLS 80
#define TRUE 1
#define FALSE 0

double eval(char[]);
double pop(struct stack *);
void push(struct stack *, double);
int empty(struct stack *);
int isdigit(char);
double oper(int, double, double);

void main()
{
   char expr[MAXCOLS];
   int position = 0;

   while((expr[position++] = getchar()) != '\n')
      ;
   expr[--position] = '\0';
   printf("%s%s", "the original postfix expression is", expr);

   printf("\n%f", eval(expr));
} /* end main */
```

The main part of the program is, of course, the function *eval*, which follows. That function is merely the C implementation of the evaluation algorithm, taking into account the specific environment and format of the input data and calculated outputs. *eval* calls on a function *isdigit* that determines whether or not its argument is an operand. The declaration for a stack that appears below is used by the function *eval* that follows it as well as by the routines *pop* and *push* that are called by *eval*.

```
struct stack {
    int top;
    double items[MAXCOLS];
};
```

```
double eval(char expr[])
{
   int c, position;
   double opnd1, opnd2, value;
   struct stack opndstk;

   opndstk.top = -1;
   for (position = 0; (c = expr[position]) != '\0'; position++)
      if (isdigit(c))
         /* operand-- convert the character representation */
         /* of the digit into double and push it onto      */
         /*                the stack                       */
         push(&opndstk, (double) (c-'0'));
      else {
         /*              operator                          */
         opnd2 = pop(&opndstk);
         opnd1 = pop(&opndstk);
         value = oper(c, opnd1, opnd2);
         push(&opndstk, value);
      } /* end else */
   return(pop(&opndstk));
} /* end eval */
```

For completeness we present *isdigit* and *oper*. The function *isdigit* simply checks if its argument is a digit:

```
int isdigit(char symb)
{
   return(symb >= '0' && symb <= '9');
}
```

This function is available as a predefined macro in most C systems.

The function *oper* checks that its first argument is a valid operator and, if it is, determines the results of its operation on the next two arguments. For exponentiation, we use the function *pow*(*op1*, *op2*) as defined in *math.h*.

```
double oper(int symb, double op1, double op2)
{
   switch(symb) {
      case '+' : return (op1 + op2);
      case '-' : return (op1 - op2);
      case '*' : return (op1 * op2);
      case '/' : return (op1 / op2);
      case '$' : return (pow(op1, op2));
      default  : printf("%s", "illegal operation");
                 exit(1);
   } /* end switch */
} /* end oper */
```

Limitations of the Program

Before we leave the program, we should note some of its deficiencies. Understanding what a program cannot do is as important as knowing what it can do. It should be obvious that attempting to use a program to solve a problem for which it was not intended leads to chaos. Worse still is attempting to solve a problem with an incorrect program, only to have the program produce incorrect results without the slightest trace of an error message. In these cases the programmer has no indication that the results are wrong and may therefore make faulty judgments based on those results. For this reason, it is important for the programmer to understand the limitations of the program.

A major criticism of this program is that it does nothing in terms of error detection and recovery. If the data on each input line represents a valid postfix expression, the program works. Suppose, however, that one input line has too many operators or operands or that they are not in a proper sequence. These problems could come about as a result of someone innocently using the program on a postfix expression that contains two digit numbers, yielding an excessive number of operands. Or possibly the user of the program is under the impression that the program handles negative numbers and that they are to be entered with the minus sign, the same sign that is used to represent subtraction. These minus signs are treated as subtraction operators, resulting in an excess number of operators. Depending on the specific type of error, the computer may take one of several actions (for example, halt execution or print erroneous results).

Suppose that at the final statement of the program, the stack *opndstk* is not empty. We get no error messages (because we asked for none), and *eval* returns a numerical value for an expression that was probably incorrectly stated in the first place. Suppose that one of the calls to the *pop* routine raises the *underflow* condition. Since we did not use the *popandtest* routine to pop elements from the stack, the program halts. This seems unreasonable, since faulty data on one line should not prevent the processing of additional lines. By no means are these the only problems that could arise. As exercises, you may wish to write programs that accommodate less restrictive inputs and some others that detect some of the aforementioned errors.

Converting an Expression from Infix to Postfix

We have thus far presented routines to evaluate a postfix expression. Although we have discussed a method for transforming infix to postfix, we have not as yet presented an algorithm for doing so. It is to this task that we now direct our attention. Once such an algorithm has been constructed, we will have the capability of reading an infix expression and evaluating it by first converting it to postfix and then evaluating the postfix expression.

In our previous discussion we mentioned that expressions within innermost parentheses must first be converted to postfix so that they can then be treated as single operands. In this fashion parentheses can be successively eliminated until the entire expression is converted. The last pair of parentheses to be opened within a group of parentheses encloses the first expression within that group to be transformed. This last-in, first-out behavior should immediately suggest the use of a stack.

Consider the two infix expressions $A + B * C$ and $(A + B) * C$, and their respective postfix versions $ABC * +$ and $AB + C *$. In each case the order of the operands is the same as the order of the operands in the original infix expressions. In scanning the first expression, $A + B * C$, the first operand, A, can be inserted immediately into the postfix expression. Clearly the + symbol cannot be inserted until after its second operand, which has not yet been scanned, is inserted. Therefore, it must be stored away to be retrieved and inserted in its proper position. When the operand B is scanned, it is inserted immediately after A. Now, however, two operands have been scanned. What prevents the symbol + from being retrieved and inserted? The answer is, of course, the * symbol that follows, which has precedence over +. In the case of the second expression the closing parenthesis indicates that the + operation should be performed first. Remember that in postfix, unlike infix, the operator that appears earlier in the string is the one that is applied first.

Since precedence plays such an important role in transforming infix to postfix, let us assume the existence of a function *prcd*(*op*1,*op*2), where *op*1 and *op*2 are characters representing operators. This function returns *TRUE* if *op*1 has precedence over *op*2 when *op*1 appears to the left of *op*2 in an infix expression without parentheses. *prcd* (*op*1,*op*2) returns *FALSE* otherwise. For example, *prcd*('*','+') and *prcd*('+','+') are *TRUE*, whereas *prcd*('+','*') is *FALSE*.

Let us now present an outline of an algorithm to convert an infix string without parentheses into a postfix string. Since we assume no parentheses in the input string, the only governor of the order in which operators appear in the postfix string is precedence. (The line numbers that appear in the algorithm will be used for future reference.)

```
1  opstk = the empty stack;
2  while (not end of input)  {
3     symb = next input character;
4     if (symb is an operand)
         add symb to the postfix string
5     else {
6        while(!empty(opstk) && prcd(stacktop(opstk), symb))  {
7           topsymb = pop(opstk);
8           add topsymb to the postfix string;
          }  /* end while */
9        push(opstk, symb);
      }  /* end else */
   }  /* end while */
   /* output any remaining operators */
10 while (!empty(opstk))  {
11    topsymb = pop(opstk);
12    add topsymb to the postfix string;
   }  /* end while */
```

Simulate the algorithm with such infix strings as "A * B + C * D" and "A + B * C $ D $ E" [where '$' represents exponentiation and *prcd* ('$', '$') equals *FALSE*] to

convince yourself that it is correct. Note that at each point of the simulation, an operator on the stack has a lower precedence than all the operators above it. This is because the intial empty stack trivially satisfies this condition, and an operator is pushed onto the stack (line 9) only if the operator currently on top of the stack has a lower precedence than the incoming operator.

What modification must be made to this algorithm to accommodate parentheses? The answer is, surprisingly little. When an opening parenthesis is read, it must be pushed onto the stack. This can be done by establishing the convention that *prcd*(*op*,'(') equals *FALSE*, for any operator symbol *op* other than a right parenthesis. In addition, we define *prcd*('(',*op*) to be *FALSE* for any operator symbol *op*. [The case of *op* = = ')' will be discussed shortly.] This ensures that an operator symbol appearing after a left parenthesis is pushed onto the stack.

When a closing parenthesis is read, all operators up to the first opening parenthesis must be popped from the stack into the postfix string. This can be done by defining *prcd*(*op*, ')') as *TRUE* for all operators *op* other than a left parenthesis. When these operators have been popped off the stack and the opening parenthesis is uncovered, special action must be taken. The opening parenthesis must be popped off the stack and it and the closing parenthesis discarded rather than placed in the postfix string or on the stack. Let us set *prcd*('(',')') to *FALSE*. This ensures that upon reaching an opening parenthesis, the loop beginning at line 6 is skipped, so that the opening parenthesis is not inserted into the postfix string. Execution therefore proceeds to line 9. However, since the closing parenthesis should not be pushed onto the stack, line 9 is replaced by the statement

```
9     if (empty(opstk)  || symb != ')')
         push(opstk, symb);
      else /* pop the open parenthesis and discard it */
         topsymb = pop(opstk);
```

With the foregoing conventions for the *prcd* function and the revision to line 9, the algorithm can be used to convert any infix string to postfix. We summarize the precedence rules for parentheses:

```
prcd('(',op) = FALSE       for any operator op
prcd(op,'(') = FALSE       for any operator op other than ')'
prcd(op,')') = TRUE        for any operator op other than '('
prcd(')',op) = undefined   for any operator op (an attempt
                           to compare the two indicates an
                           error).
```

We illustrate this algorithm on some examples:

Example 1: $A + B * C$

The contents of *symb*, the postfix string, and *opstk* are shown after scanning each symbol. *opstk* is shown with its top to the right.

	symb	*postfix string*	*opstk*
1	*A*	*A*	
2	+	*A*	+
3	*B*	*AB*	+
4	*	*AB*	+ *
5	*C*	*ABC*	+ *
6		*ABC* *	+
7		*ABC* * +	

Lines 1, 3, and 5 correspond to the scanning of an operand; therefore the symbol (*symb*) is immediately placed on the postfix string. In line 2 an operator is scanned and the stack is found to be empty, and the operator is therefore placed on the stack. In line 4 the precedence of the new symbol (*) is greater than the precedence of the symbol on the top of the stack (+); therefore the new symbol is pushed onto the stack. In steps 6 and 7 the input string is empty, and the stack is therefore popped and its contents are placed on the postfix string.

Example 2: $(A + B) * C$

symb	postfix string	opstk
(		(
A	*A*	(
+	*A*	(+
B	*AB*	(+
)	*AB* +	
*	*AB*+	*
C	*AB* + *C*	*
	AB + *C* *	

In this example, when the right parenthesis is encountered the stack is popped until a left parenthesis is encountered, at which point both parentheses are discarded. By using parentheses to force an order of precedence different than the default, the order of appearance of the operators in the postfix string is different than in example 1.

Example 3: $((A - (B + C)) * D) \$ (E + F)$ (See example 3 on top of page 106.)

Why does the conversion algorithm seem so involved, whereas the evaluation algorithm seems so simple? The answer is that the former converts from one order of precedence (governed by the *prcd* function and the presence of parentheses) to the natural order (that is, the operation to be executed first appears first). Because of the many combinations of elements at the top of the stack (if not empty) and possible incoming symbol, a large number of statements are necessary to ensure that every possibility is covered. In the latter algorithm, on the other hand, the operators appear in precisely the order they are to be executed. For this reason the operands can be stacked until an operator is found, at which point the operation is performed immediately.

EXAMPLE 3

symb	*postfix string*	*opstk*
(		(
(		((
A	*A*	((
−	*A*	((−
(	*A*	((− (
B	*AB*	((− (
+	*AB*	((− (+
C	*ABC*	((− (+
)	*ABC* +	((−
)	*ABC* + −	(
*	*ABC* + −	(*
D	*ABC* + −*D*	(*
)	*ABC* + −*D* *	
$	*ABC* + −*D* *	$
(	*ABC* + −*D* *	$ (
E	*ABC* + −*D* **E*	$ (
+	*ABC* + −*D* **E*	$ (+
F	*ABC* + −*D* **EF*	$ (+
)	*ABC* + −*D* **EF* +	$
	ABC + −*D* **EF* + $	

The motivation behind the conversion algorithm is the desire to output the operators in the order in which they are to be executed. In solving this problem by hand we could follow vague instructions that require us to convert from the inside out. This works very well for humans doing a problem with pencil and paper (if they do not become confused or make a mistake). However, a program or an algorithm must be more precise in its instructions. We cannot be sure that we have reached the innermost parentheses or the operator with the highest precedence until additional symbols have been scanned. At the time, we must backtrack to some previous point.

Rather than backtrack continuously, we make use of the stack to "remember" the operators encountered previously. If an incoming operator is of greater precedence than the one on top of the stack, this new operator is pushed onto the stack. This means that when all the elements in the stack are finally popped, this new operator will precede the former top in the postfix string (which is correct since it has higher precedence). If, on the other hand, the precedence of the new operator is less than that of the top of the stack, the operator at the top of the stack should be executed first. Therefore the top of the stack is popped and the incoming symbol is compared with the new top, and so on. Parentheses in the input string override the order of operations. Thus when a left parenthesis is scanned, it is pushed on the stack. When its associated right parenthesis is found, all the operators between the two parentheses are placed on the output string, because they are to be executed before any operators appearing after the parentheses.

Program to Convert an Expression from Infix to Postfix

There are two things that we must do before we actually start writing a program. The first is to define precisely the format of the input and output. The second is to construct, or at least define, those routines that the main routine depends upon. We

assume that the input consists of strings of characters, one string per input line. The end of each string is signaled by the occurrence of an end-of-line character ('\n'). For the sake of simplicity, we assume that all operands are single-character letters or digits. All operators and parentheses are represented by themselves, and '$' represents exponentiation. The output is a character string. These conventions make the output of the conversion process suitable for the evaluation process, provided that all the single character operands in the initial infix string are digits.

In transforming the conversion algorithm into a program, we make use of several routines. Among these are *empty*, *pop*, *push* and *popandtest*, all suitably modified so that the elements on the stack are characters. We also make use of a function *isoperand* that returns *TRUE* if its argument is an operand and *FALSE* otherwise. This simple function is left to the reader.

Similarly, the *prcd* function is left to the reader as an exercise. It accepts two single-character operator symbols as arguments and returns *TRUE* if the first has precedence over the second when it appears to the left of the second in an infix string and *FALSE* otherwise. The function should, of course, incorporate the parentheses conventions previously introduced.

Once these auxiliary functions have been written, we can write the conversion function *postfix* and a program that calls it. The program reads a line containing an expression in infix, calls the routine *postfix*, and prints the postfix string. The body of the main routine follows:

```
#include <stdio.h>
#include <stdlib.h>

#define MAXCOLS 80
#define TRUE 1
#define FALSE 0

void postfix(char *, char *);
int isoperand(char);
void popandtest(struct stack *, char *, int *);
int prcd(char, char);
void push(struct stack *, char);
char pop(struct stack *);


void main()
{
  char infix[MAXCOLS];
  char postr[MAXCOLS];
  int pos = 0;

  while ((infix[pos++] = getchar()) != '\n');
  infix[--pos] = '\0';
  printf("%s%s", "the original infix expression is ", infix);
  postfix(infix, postr);
  printf("%s\n", postr);
} /* end main */
```

The declaration for the operator stack and the *postfix* routine follows:

```
struct stack {
   int top;
   char items[MAXCOLS];
};

postfix(char infix[], char postr[])
{
   int position, und;
   int outpos = 0;
   char topsymb = '+';
   char symb;
   struct stack opstk;
   opstk.top = -1;      /* the empty stack */

   for (position=0; (symb = infix[position]) != '\0'; position++)
      if (isoperand(symb))
         postr[outpos++] = symb;
      else {
         popandtest(&opstk, &topsymb, &und);
         while (!und && prcd(topsymb, symb)) {
             postr[outpos++] = topsymb;
             popandtest(&opstk, &topsymb, &und);
         } /* end while */
         if (!und)
             push(&opstk, topsymb);
         if (und || (symb != ')'))
             push(&opstk, symb);
         else
             topsymb = pop(&opstk);
      } /* end else */
    while (!empty(&opstk))
       postr[outpos++] = pop(&opstk);
    postr[outpos] = '\0';
    return;

} /* end postfix */
```

The program has one major flaw in that it does not check that the input string is a valid infix expression. In fact, it would be instructive for you to examine the operation of this program when it is presented with a valid postfix string as input. As an exercise you are asked to write a program that checks whether or not an input string is a valid infix expression.

We can now write a program to read an infix string and compute its numerical value. If the original string consists of single-digit operands with no letter operands, the following program reads the original string and prints its value.

```
#define MAXCOLS 80
void main()
{
   char instring[MAXCOLS], postring[MAXCOLS];
   int position = 0;
   double eval();

   while((instring[position++] = getchar()) != '\n')
      ;
   instring[--position] = '\0';
   printf("%s%s\n", "infix expresson is ", instring);
   postfix(instring, postring);
   printf("%s%f\n", "value is ", eval(postring));
} /* end main */
```

Two different versions of the stack manipulation routines (*pop*, *push*, and so forth), and associated function prototypes, are required because *postfix* uses a stack of character operators (that is, *opstk*), whereas *eval* uses a stack of float operands (that is, *opndstk*). Of course, it is possible to use a single stack that can contain both reals or characters by defining a union as described earlier in Section 1.3.

Most of our attention in this section has been devoted to transformations involving postfix expressions. An algorithm to convert an infix expression into postfix scans characters from left to right, stacking and unstacking as necessary. If it were necessary to convert from infix to prefix, the infix string could be scanned from right to left and the appropriate symbols entered in the prefix string from right to left. Since most algebraic expressions are read from left to right, postfix is a more natural choice.

The foregoing programs are merely indicative of the type of routines one could write to manipulate and evaluate postfix expressions. They are by no means comprehensive or unique. There are many variations of the foregoing routines that are equally acceptable. Some of the early high-level language compilers actually used routines such as *eval* and *postfix* to handle algebraic expressions. Since that time, more sophisticated techniques have been developed to handle these problems.

Stacks in C++ Using Templates

There are a number of drawbacks to the solution that we just presented. First, although two stacks are used in the complete solution (a stack of operators in the *postfix* routine and a stack of operands in the *eval* routine), only a single stack is used at any one time. Nevertheless, in the implementation of the solution that we presented, both stacks were created and remained in memory throughout the entire program. Second, because the stacks are not of the same type, it is necessary to declare them separately. And with the separate declarations, it is necessary to provide separate sets of primitive routines (that is, *push*, *pop*, *empty*, etc.). This, in turn, implies that when the implementation of a stack is to be changed it must be changed for each type of stack that we have created.

It would be more efficient if we could design a system around a stack of indeterminate type and define the primitive routines on such a stack. We would then create and destroy instances of such a stack as necessary. This would eliminate the need for

us to create separate primitive routines, as well as serve the purpose of allowing us to destroy a stack when it is no longer needed.

The C++ feature that supports the definition of an object of undetermined type is called a ***template***. Using a template allows the programmer to define the features of the class, while reserving the option of binding the type of the class to the class itself until a class of a particular type is actually needed. The creation of a class of a particular type is called ***instantiation***.

Let us now consider how we could create a template for stacks and then illustrate how this template could be used in the previous example: accepting an infix string, converting the infix string to a postfix string (using a stack of operators), and then evaluating the postfix string (using a stack of operands). We begin by defining the class that we shall use (in our example, the stack); however, this definition is parameterized in the snese that it depends on the attributes of a specific paramter. We denote this by using

```
template <class T>
```

as a prefix to the remainder of the definition of the class. This prefix indicates that *T* is a parameter in the subsequent definition and will vary from one use to another. The construct that allows us to create a class of (as yet) undetermined type is called a ***template***.

Thus the template definition is as follows:

```
template <class T>
// T is of ordinal type
class Stack {
private:
      int top;                          // top points to the next top element
      T *nodes;
public:
      Stack ();                         // default constructor
      int empty (void);
      void push(T &);
      T pop(void);
      T pop(int &);                     // example of overloading pop to
                                        // handle the functions of popandtest
      ~Stack ();                        // default destructor
};
```

Within the same file, we would also provide the implementation of the stack template. For example, the constructor for *Stack* would be implemented as follows:

```
//   Implementation of templates
template <class T> Stack<T>::Stack ()
{
     top = -1;
     nodes = new T[STACKSIZE];
};
```

In the example above, the maximum size of the stack is predetermined to be *STACKSIZE* and the *top* (a private variable) is initialized to -1. An array of nodes of type *T* is created when the stack is *instantiated*. Thus a stack of integers would result in an array of integers, while a stack of double would result in an array of double. Each stack of a particular type would be instantiated with an array of that type. A destructor for a stack could be implemented by

```
template <class T> Stack<T>::~Stack ()
{
    delete nodes;
};
```

The primitive routines *empty*, *push*, and *pop* are straightforward. (Note that we also overload the function *pop* by incorporating into it the functionality of *popandtest*.)

```
template <class T> int Stack<T>::empty (void)
{
    return top> =0;
};

template <class T> void Stack<T>::push(T & j)
{
    if (top == STACKSIZE) {
        cout << "Stack overflow" << endl;
        return;
    }
    nodes[++top] = j;
}

template <class T> T Stack<T>::pop(void)
{
    T p;
    if (empty ()) {
        cout << "Stack underflow" << endl;
        return p;
    }
    p = nodes[top--];
    return p;
};

// The tasks of this function were formerly performed by popandtest
template <class T> T Stack<T>::pop(int & und)
{
    T p;
```

```
        if (empty ()) {
            und = 1;
            return p;
        }
        und = 0;
        p = nodes[top--];
        return p;
};
```

To make use of the stack template, it would be necessary to include all of the prototype definitions above in a file as follows:

```
// stackt.h

#ifndef STACKT_H
#define STACKT_H

#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <mem.h>
#include <iostream.h>

#define STACKSIZE 100
```

These statements would be followed by all of the above (template definition and prototype functions) and ended with

```
#endif
```

The above constitute a definition for the template stack.

The meaning of the expression *#ifndef...* is that if the operand *STACKT_H* is not already defined, it should be defined here; otherwise, the definition may be bypassed. This is a precaution against a double definition (which might occur if there was an *#include* within an *#include*).

To make use of a stack of a particular type, it would first be necessary to ***instantiate*** it. This is done simply by declaring a stack of a particular type. By declaring such a stack, the stack class will create an instance of a stack of the specified type. Thus we could build a file that contains the following routines:

```
#define MAXCOLS 52

void postfix(char *infix, char *postr);

int prcd(char op1, char op2);
int isoperand(char op);
int isoperator(char op);
```

```
long double eval(char *postr);
long double oper(int symb, long double op1, long double op2);

int prcd(char op1, char op2)
//  body of prcd goes here

int isoperator(char op)
// body of isoperator goes here

int isoperand(char op)
// body of isoperand goes here

void postfix(char *infix, char *postr)
{
     int position, und;
     int outpos=0;
     char topsymb='+';
     char symb;
     Stack<char> opstk;

     for (position=0; (symb=infix[position]) != '\0'; position++) {
          if (isoperand(symb))
               postr[outpos++]=symb;
          else {
               topsymb=opstk.pop(und);
               while (!und && prcd(topsymb, symb)) {
                    postr[outpos++] = topsymb;
                    topsymb=opstk.pop(und);
               }
               if (!und)
                    opstk.push(topsymb);
               if (und || (symb != ')'))
                    opstk.push(symb);
               else
                    topsymb=opstk.pop();
          }
     } /* end for */
     while (!opstk.empty())
          postr[outpos++]=opstk.pop();
     postr[outpos]='\0';
} /* end postfix */

long double oper(int symb, long double op1, long double op2)
// body of isoperator goes here

long double eval(char *postr)
{
     int c, position;
     long double opnd1, opnd2, value;
     Stack<long double> opndstk;
```

```
    for (position=0; (c=postr[position]) != '\0'; position++)
        if (isoperand(c))
            opndstk.push((float) (c-'0'));
        else {
            opnd2=opndstk.pop();
            opnd1=opndstk.pop();
            value=oper(c, opnd1, opnd2);
            opndstk.push(value);
        }
    return (opndstk.pop());
} /* end eval */
```

Notice that the statements that instantiate the stacks contain the type of the stack as a parameter. Thus the statement

```
stack<long double> opndstk;
```

within the function *eval* creates a stack of *long double*, while the statement

```
stack<char> opstk;
```

within *postfix* creates a stack of type *char*. As we mentioned earlier, one set of routines for a template class stack is sufficient to manipulate a stack of any type.

Finally, all of the above could be followed by a main routine:

```
void main(void)
{
    char in[250], post[250];
    long double res;

    cin >> in;
    cout << in << endl;
    postfix (in, post);
    res = eval(post);
    cout << res << endl;
```

Two points should be noted regarding the method that we just presented. First, the set of programs above creates two classes of type *stack:* a stack of *char* and a stack of *long double*. The existence of these two classes is based on declarations that use them. Thus, since the stack of characters is present in the *postfix* routine, a class of that type will be created; and because a stack of double long is present in the *eval* routine, a class of that type will be created. The existence of these classes is really independent of the existence of any objects of the respective types. Of course, we did declare a stack of each type. But you should note that the two stack classes exist regardless of whether or not the objects of those types are declared (that is, the routines in which they are created may never be called).

The second point deals with the destructor. Although we defined a default destructor, we never really called it explicitly. This is because the default destructor is called automatically when control is returned from the routine in which an object was created.

EXERCISES

2.3.1. Transform each of the following expressions to prefix and postfix.
- (a) $A + B - C$
- (b) $(A + B)*(C - D) \$ E * F$
- (c) $(A + B)*(C \$ (D - E) + F) - G$
- (d) $A + (((B - C)*(D - E) + F) / G) \$ (H - J)$

2.3.2. Transform each of the following prefix expressions to infix.
- (a) $+ - ABC$
- (b) $+ A - BC$
- (c) $++ A - * \$ BCD / + EF * GHI$
- (d) $+ - \$ ABC * D ** EFG$

2.3.3. Transform each of the following postfix expressions to infix.
- (a) $AB + C -$
- (b) $ABC + -$
- (c) $AB - C + DEF - + \$$
- (d) $ABCDE - + \$ * EF * -$

2.3.4. Apply the evaluation algorithm in the text to evaluate the following postfix expressions. Assume $A = 1$, $B = 2$, $C = 3$.
- (a) $AB + C - BA + C \$ -$
- (b) $ABC + * CBA - + *$

2.3.5. Modify the routine *eval* to accept as input a character string of operators and operands representing a postfix expression and to create the fully parenthesized infix form of the original postfix. For example, $AB +$ would be transformed into $(A + B)$ and $AB + C -$ would be transformed into $((A + B) - C)$.

2.3.6. Write a single program combining the features of *eval* and *postfix* to evaluate an infix string. Use two stacks, one for operands and the other for operators. Do not first convert the infix string to postfix and then evaluate the postfix string, but rather evaluate as you go along.

2.3.7. Write a routine *prefix* to accept an infix string and create the prefix form of that string, assuming that the string is read from right to left and that the prefix string is created from right to left.

2.3.8. Write a C program to convert
- (a) A prefix string to postfix
- (b) A postfix string to prefix
- (c) A prefix string to infix
- (d) A postfix string to infix

2.3.9. Write a C routine *reduce* that accepts an infix string and forms an equivalent infix string with all superfluous parentheses removed. Can this be done without using a stack?

2.3.10. Assume a machine that has a single register and six instructions.

LD	*A*	Places the operand A into the register
ST	*A*	Places the contents of the register into the variable *A*
AD	*A*	Adds the contents of the variable *A* to the register
SB	*A*	Subtracts the contents of the variable *A* from the register
ML	*A*	Multiplies the contents of the register by the variable *A*
DV	*A*	Divides the contents of the register by the variable *A*

Write a program that accepts a postfix expression containing single-letter operands and the operators +, −, *, and / and prints a sequence of instructions to evaluate the expression and leave the result in the register. Use variables of the form *TEMPn* as temporary variables. For example, using the postfix expression *ABC* * + *DE* − / should print the following:

```
LD    B
ML    C
ST    TEMP1
LD    A
AD    TEMP1
ST    TEMP2
LD    D
SB    E
ST    TEMP3
LD    TEMP2
DV    TEMP3
ST    TEMP4
```

2.3.11. The template definition of a stack can be expanded to allow the *size* of the stack to be a parameter as well. Show how to define a stack template in which each instantiation of the stack will have both the element type and the size of the stack as a parameter.

2.3.12. Can a template be used to store elements of different types on the same stack? Why or why not?

3 Recursion

This chapter introduces recursion, a programming tool that is one of the most powerful and one of the least understood by beginning students of programming. We define recursion, introduce its use in C, and present several examples. We also examine an implementation of recursion using stacks. Finally, we discuss the advantages and disadvantages of using recursion in problem solving.

3.1 RECURSIVE DEFINITION AND PROCESSES

Many objects in mathematics are defined by presenting a process to produce that object. For example, π is defined as the ratio of the circumference of a circle to its diameter. This is equivalent to the following set of instructions: obtain the circumference of a circle and its diameter, divide the former by the latter, and call the result π. Clearly, the process specified must terminate with a definite result.

Factorial Function

Another example of a definition specified by a process is that of the factorial function, which plays an important role in mathematics and statistics. Given a positive integer n, *n factorial* is defined as the product of all integers between n and 1. For

example, 5 factorial equals $5 * 4 * 3 * 2 * 1 = 120$, and 3 factorial equals $3 * 2 * 1 = 6$. 0 factorial is defined as 1. In mathematics, the exclamation mark (!) is often used to denote the factorial function. We may therefore write the definition of this function as follows:

$$n! = 1 \text{ if } n = 0$$
$$n! = n * (n - 1) * (n - 2) * \ldots * 1 \text{ if } n > 0$$

The three dots are really a shorthand for all the numbers between $n - 3$ and 2 multiplied together. To avoid this shorthand in the definition of $n!$ we would have to list a formula for $n!$ for each value of n separately, as follows:

```
0! = 1
1! = 1
2! = 2 * 1
3! = 3 * 2 * 1
4! = 4 * 3 * 2 * 1
.......
```

Of course, we cannot hope to list a formula for the factorial of each integer. To avoid any shorthand and to avoid an infinite set of definitions, yet to define the function precisely, we may present an algorithm that accepts an integer n and returns the value of $n!$.

```
prod = 1;
for (x = n; x > 0; x--)
   prod *= x;
return(prod);
```

Such an algorithm is called ***iterative*** because it calls for the explicit repetition of some process until a certain condition is met. This algorithm can be translated readily into a C function that returns $n!$ when n is input as a parameter. An algorithm may be thought of as a program for an "ideal" machine without any of the practical limitations of a real computer and may therefore be used to define a mathematical function. A C function, however, cannot serve as the mathematical definition of the factorial function because of such limitations as precision and the finite size of a real machine.

Let us look more closely at the definition of $n!$ that lists a separate formula for each value of n. We may note, for example, that 4! equals $4 * 3 * 2 * 1$, which equals $4 * 3!$. In fact, for any $n > 0$, we see that $n!$ equals $n * (n - 1)!$. Multiplying n by the product of all integers from $n - 1$ to 1 yields the product of all integers from n to 1. We may therefore define

```
0! = 1
1! = 1 * 0!
2! = 2 * 1!
3! = 3 * 2!
4! = 4 * 3!
  ...
```

or, using the mathematical notation used earlier,

$0! = 1$

$n! = n * (n-1)!$

This definition may appear quite strange, since it defines the factorial function in terms of itself. This seems to be a circular definition and totally unacceptable until we realize that the mathematical notation is only a concise way of writing out the infinite number of equations necessary to define $n!$ for each n. 0! is defined directly as 1. Once 0! has been defined, defining 1! as $1 * 0!$ is not circular at all. Similarly, once 1! has been defined, defining 2! as $2 * 1!$ is equally straightforward. It may be argued that the latter notation is more precise than the definition of $n!$ as $n * (n-1) * \cdots * 1$ for $n > 0$ because it does not resort to three dots to be filled in by the (it is hoped) logical intuition of the reader. Such a definition, which defines an object in terms of a simpler case of itself, is called a ***recursive definition***.

Let us see how the recursive definition of the factorial function may be used to evaluate 5!. The definition states that 5! equals $5 * 4!$. Thus, before we can evaluate 5!, we must first evaluate 4!. Using the definition once more, we find that $4! = 4 * 3!$. Therefore, we must evaluate 3!. Repeating this process, we have that

```
1 5! = 5 * 4!
2        4! = 4 * 3!
3               3! = 3 * 2!
4                      2! = 2 * 1!
5                             1! = 1 * 0!
6                                    0! = 1
```

Each case is reduced to a simpler case until we reach the case of 0!, which is defined directly as 1. At line 6 we have a value that is defined directly and not as the factorial of another number. We may therefore backtrack from line 6 to line 1, returning the value computed in one line to evaluate the result of the previous line. This produces

```
6' 0! = 1
5' 1! = 1 * 0! = 1 * 1 = 1
4' 2! = 2 * 1! = 2 * 1 = 2
3' 3! = 3 * 2! = 3 * 2 = 6
2' 4! = 4 * 3! = 4 * 6 = 24
1' 5! = 5 * 4! = 5 * 24 = 120
```

Let us attempt to incorporate this process into an algorithm. Again, we want the algorithm to input a nonnegative integer n and to compute in a variable *fact* the nonnegative integer that is n factorial.

```
1   if (n == 0)
2      fact = 1;
3      else {
4        x = n - 1;
5        find the value of x!. Call it y;
6        fact = n * y;
7      } /* end else */
```

This algorithm exhibits the process used to compute $n!$ by the recursive definition. The key to the algorithm is, of course, line 5, where we are told to "find the value of $x!$." This requires reexecuting the algorithm with input x, since the method for computing the factorial function is the algorithm itself. To see that the algorithm eventually halts, note that at the start of line 5, x equals $n - 1$. Each time the algorithm is executed, its input is one less than the preceding time, so that (since the original input n was a nonnegative integer) 0 is eventually input to the algorithm. At that point, the algorithm simply returns 1. This value is returned to line 5, which asked for the evaluation of 0!. The multiplication of y (which equals 1) by n (which equals 1) is then executed and the result is returned. This sequence of multiplications and returns continues until the original $n!$ has been evaluated. In the next section we will see how to convert this algorithm into a C program.

Of course, it is much simpler and more straightforward to use the iterative method for evaluation of the factorial function. We present the recursive method as a simple example to introduce recursion, not as a more effective method of solving this particular problem. Indeed, all the problems in this section can be solved more efficiently by iteration. However, later in this chapter and in subsequent chapters, we will come across examples that are more easily solved by recursive methods.

Multiplication of Natural Numbers

Another example of a recursive definition is the definition of multiplication of natural numbers. The product $a * b$, where a and b are positive integers, may be defined as a added to itself b times. This is an iterative definition. An equivalent recursive definition is

$$a * b = a \text{ if } b == 1$$
$$a * b = a * (b - 1) + a \text{ if } b > 1$$

To evaluate $6 * 3$ by this definition, we first evaluate $6 * 2$ and then add 6. To evaluate $6 * 2$, we first evaluate $6 * 1$ and add 6. But $6 * 1$ equals 6 by the first part of the definition. Thus

$$6 * 3 = 6 * 2 + 6 = 6 * 1 + 6 + 6 = 6 + 6 + 6 = 18$$

The reader is urged to convert the definition above to a recursive algorithm as a simple exercise.

Note the pattern that exists in recursive definitions. A simple case of the term to be defined is defined explicitly (in the case of factorial, 0! was defined as 1; in the case of multiplication, $a * 1 = a$). The other cases are defined by applying some operation to the result of evaluating a simpler case. Thus $n!$ is defined in terms of $(n - 1)!$ and $a * b$ in terms of $a * (b - 1)$. Successive simplifications of any particular case must eventually lead to the explicitly defined trivial case. In the case of the factorial function, successively subtracting 1 from n eventually yields 0. In the case of multiplication, successively subtracting 1 from b eventually yields 1. If this were not the case, the definition would be invalid. For example, if we defined

```
n! = (n + 1)!/(n + 1)
```

or

```
a * b = a * (b + 1) - a
```

we would be unable to determine the value of 5! or 6 * 3. (You are invited to attempt to determine these values using the foregoing definitions.) This is true despite the fact that the two equations are valid. Continually adding one to n or b does not eventually produce an explicitly defined case. Even if 100! was defined explicitly, how could the value of 101! be determined?

Fibonacci Sequence

Let us examine a less familiar example. The ***Fibonacci sequence*** is the sequence of integers

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Each element in this sequence is the sum of the two preceding elements (for example, $0 + 1 = 1, 1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, \ldots$). If we let $fib(0) = 0$, $fib(1) = 1$, and so on, then we may define the Fibonacci sequence by the following recursive definition:

```
fib(n) = n if n == 0 or n == 1
fib(n) = fib(n - 2) + fib(n - 1) if n >= 2
```

To compute $fib(6)$, for example, we may apply the definition recursively to obtain

```
fib(6) = fib(4) + fib(5) = fib(2) + fib(3) + fib(5) =
fib(0) + fib(1) + fib(3) + fib(5) = 0 + 1 + fib(3) + fib(5) =
1 + fib(1) + fib(2) + fib(5) =
1 + 1 + fib(0) + fib(1) + fib(5) =
2 + 0 + 1 + fib(5) = 3 + fib(3) + fib(4) =
3 + fib(1) + fib(2) + fib(4) =
3 + 1 + fib(0) + fib(1) + fib(4) =
4 + 0 + 1 + fib(2) + fib(3) = 5 + fib(0) + fib(1) + fib(3) =
5 + 0 + 1 + fib(1) + fib(2) = 6 + 1 + fib(0) + fib(1) =
7 + 0 + 1 = 8
```

Notice that the recursive definition of the Fibonacci numbers differs from the recursive definitions of the factorial function and multiplication. The recursive definition of *fib* refers to itself twice. For example, $fib(6) = fib(4) + fib(5)$, so that in computing $fib(6)$, *fib* must be applied recursively twice. However, the computation of $fib(5)$ also involves determining $fib(4)$, so that a great deal of computational redundancy occurs in applying the definition. In the foregoing example, $fib(3)$ is computed three separate times. It would be much more efficient to "remember" the value of $fib(3)$ the first time that it is evaluated and reuse it each time that it is needed. An iterative method of computing $fib(n)$ such as the following is much more efficient:

```
if (n <= 1)
   return(n);
lofib = 0;
hifib = 1;
for (i = 2; i <= n; i++) {
   x = lofib;
   lofib = hifib;
   hifib = x + lofib;
} /* end for */
return(hifib);
```

Compare the number of additions (not including increments of the index variable *i*) that are performed in computing *fib*(6) by this algorithm and by using the recursive definition. In the case of the factorial function, the same number of multiplications must be performed in computing *n*! by the recursive and iterative methods. The same is true of the number of additions in the two methods of computing multiplication. However, in the case of the Fibonacci numbers, the recursive method is far more expensive than the iterative. We shall have more to say about the relative merits of the two methods in a later section.

Binary Search

You may have received the erroneous impression that recursion is a very handy tool for defining mathematical functions but has no influence in more practical computing activities. The next example illustrates an application of recursion to one of the most common activities in computing: that of searching.

Consider an array of elements in which objects have been placed in some order. For example, a dictionary or telephone book may be thought of as an array whose entries are in alphabetical order. A company payroll file may be in the order of employees' social security numbers. Suppose that such an array exists and that we wish to find a particular element in it. For example, we wish to look up a name in a telephone book, a word in a dictionary, or a particular employee in a personnel file. The process used to find such an entry is called a ***search***.

Since searching is such a common activity in computing, it is desirable to find an efficient method for performing it. Perhaps the crudest search method is the ***sequential*** or ***linear*** search, in which each item of the array is examined in turn and compared with the item being searched for until a match occurs. If the list is unordered and haphazardly constructed, the linear search may be the only way to find anything in it (unless, of course, the list is first rearranged). However, such a method would never be used in looking up a name in a telephone book. Rather, the book is opened to a random page and the names on that page are examined. Since the names are ordered alphabetically, such an examination would determine whether the search should proceed in the first or second half of the book.

Let us apply this idea to searching an array. If the array contains only one element, the problem is trivial. Otherwise, compare the item being searched for with the item at the middle of the array. If they are equal, the search has been completed successfully.

If the middle element is greater than the item being searched for, the search process is repeated in the first half of the array (since if the item appears anywhere it must appear in the first half); otherwise, the process is repeated in the second half. Note that each time a comparison is made, the number of elements yet to be searched is cut in half. For large arrays, this method is superior to the sequential search in which each comparison reduces the number of elements yet to be searched by only one. Because of the division of the array to be searched into two equal parts, this search method is called the ***binary search***.

Notice that we have quite naturally defined a binary search recursively. If the item being searched for is not equal to the middle element of the array, the instructions are to search a subarray using the same method. Thus the search method is defined in terms of itself with a smaller array as input. We are sure that the process will terminate because the input arrays become smaller and smaller, and the search of a one-element array is defined nonrecursively, since the middle element of such an array is its only element.

We now present a recursive algorithm to search a sorted array *a* for an element *x* between *a*[*low*] and *a*[*high*]. The algorithm returns an *index* of *a* such that *a*[*index*] equals *x* if such an *index* exists between *low* and *high*. If *x* is not found in that portion of the array, *binsrch* returns −1 (in C, no element *a*[−1] can exist).

```
1  if (low > high)
2     return(-1);
3  mid = (low + high) / 2;
4  if (x == a[mid])
5     return(mid);
6  if (x < a[mid])
7     search for x in a[low] to a[mid - 1];
8  else
9     search for x in a[mid + 1] to a[high];
```

Since the possibility of an unsuccessful search is included (that is, the element may not exist in the array), the trivial case has been altered somewhat. A search on a one-element array is not defined directly as the appropriate index. Instead that element is compared with the item being searched for. If the two items are not equal, the search continues in the "first" or "second" half—each of which contains no elements. This case is indicated by the condition *low* > *high*, and its result is defined directly as −1.

Let us apply this algorithm to an example. Suppose that the array *a* contains the elements 1, 3, 4, 5, 17, 18, 31, 33, in that order, and that we wish to search for 17 (that is, *x* equals 17) between item 0 and item 7 (that is, *low* is 0, *high* is 7). Applying the algorithm, we have

Line 1: Is *low* > *high*? It is not, so execute line 3.

Line 3: *mid* = (0 + 7)/2 = 3.

Line 4: Is *x* == *a*[3]? 17 is not equal to 5, so execute line 6.

Line 6: Is *x* < *a*[3]? 17 is not less than 5, so perform the ***else*** clause at line 8.

Line 9: Repeat the algorithm with *low* = *mid* + 1 = 4 and *high* = *high* = 7; i.e., search the upper half of the array.

Line 1: Is $4 > 7$? No, so execute line 3.

Line 3: $mid = (4 + 7)/2 = 5$.

Line 4: Is $x == a[5]$? 17 does not equal 18, so execute line 6.

Line 6: Is $x < a[5]$? Yes, since $17 < 18$, so search for x in $a[low]$ to $a[mid - 1]$.

Line 7: Repeat the algorithm with $low = low = 4$ and $high = mid - 1 = 4$. We have isolated x between the fourth and the fourth elements of a.

Line 1: Is $4 > 4$? No, so execute line 3.

Line 3: $mid = (4 + 4)/2 = 4$.

Line 4: Since $a[4] == 17$, return $mid = 4$ as the answer. 17 is indeed the fourth element of the array.

Note the pattern of calls to and returns from the algorithm. A diagram tracing this pattern appears in Figure 3.1.1. The solid arrows indicate the flow of control through the algorithm and the recursive calls. The dotted lines indicate returns. Since there are no steps to be executed in the algorithm after line 7 or 9, the returned result is returned intact to the previous execution. Finally, when control returns to the original execution, the answer is returned to the caller.

Let us examine how the algorithm searches for an item that does not appear in the array. Assume the array a as in the previous example and assume that it is searching for x, which equals 2.

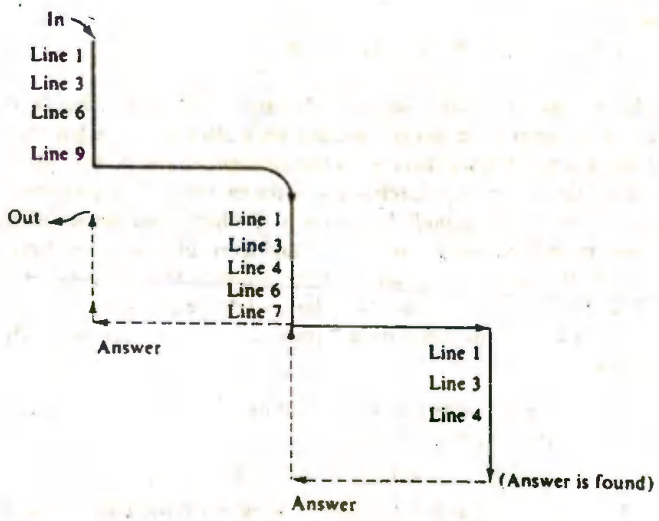


Figure 3.1.1 Diagrammatic representation of the binary search algorithm.

Line 1: Is *low* > *high*? 0 is not greater than 7, so execute line 3.

Line 3: *mid* = (0 + 7)/2 = 3.

Line 4: Is $x == a[3]$? 2 does not equal 5, so execute line 6.

Line 6: Is $x < a[3]$? Yes, $2 < 5$, so search for x in $a[low]$ to $a[mid - 1]$.

Line 7: Repeat the algorithm with *low* = *low* = 0 and *high* = *mid* − 1 = 2. If 2 appears in the array, it must appear between $a[0]$ and $a[2]$ inclusive.

Line 1: Is 0 > 2? No, execute line 3.

Line 3: *mid* = (0 + 2)/2 = 1.

Line 4: Is $2 == a[1]$? No, execute line 6.

Line 6: Is $2 < a[1]$? Yes, since $2 < 3$. Search for x in $a[low]$ to $a[mid - 1]$.

Line 7: Repeat the algorithm with *low* = *low* = 0 and *high* = *mid* − 1 = 0. If x exists in a it must be the first element.

Line 1: Is 0 > 0? No, execute line 3.

Line 3: *mid* = (0 + 0)/2 = 0.

Line 4: Is $2 == a[0]$? No, execute line 6.

Line 6: Is $2 < a[0]$? 2 is not less than 1, so perform the *else* clause at line 8.

Line 9: Repeat the algorithm with *low* = *mid* + 1 = 1 and *high* = *high* = 0.

Line 1: Is *low* > *high*? 2 is greater than 1, so − is returned. The item 2 does not exist in the array.

Properties of Recursive Definitions or Algorithms

Let us summarize what is involved in a recursive definition or algorithm. One important requirement for a recursive algorithm to be correct is that it not generate an infinite sequence of calls on itself. Clearly, any algorithm that does generate such a sequence can never terminate. For at least one argument or group of arguments, a recursive function *f* must be defined in terms that do not involve *f*. There must be a "way out" of the sequence of recursive calls. In the examples of this section the nonrecursive portions of the definitions were

```
factorial:          0! = 1
multiplication:     a * 1 = a
Fibonacci seq.:     fib(0) = 0;     fib(1) = 1
binary search:      if (low > high)
                       return(-1);
                    if (x == a[mid])
                       return(mid);
```

Without such a nonrecursive exit, no recursive function can ever be computed. Any instance of a recursive definition or invocation of a recursive algorithm must eventually reduce to some manipulation of one or more simple, nonrecursive cases.

EXERCISES

3.1.1. Write an iterative algorithm to evaluate $a * b$ by using addition, where a and b are nonnegative integers.

3.1.2. Write a recursive definition of $a + b$, where a and b are nonnegative integers, in terms of the successor function *succ*, defined as

```
succ(x)
int x;
{
   return(x++);
} /* end succ */
```

3.1.3. Let a be an array of integers. Present recursive algorithms to compute:
- (a) The maximum element of the array
- (b) The minimum element of the array
- (c) The sum of the elements of the array
- (d) The product of the elements of the array
- (e) The average of the elements of the array

3.1.4. Evaluate each of the following, using both the iterative and recursive definitions:
- (a) 6!
- (b) 9!
- (c) 100 3
- (d) 6 * 4
- (e) *fib*(10)
- (f) *fib*(11)

3.1.5. Assume that an array of ten integers contains the elements

1, 3, 7, 15, 21, 22, 36, 78, 95, 106

Use the recursive binary search to find each of the following items in the array.
- (a) 1
- (b) 20
- (c) 36

3.1.6. Write an iterative version of the binary search algorithm. (*Hint:* Modify the values of *low* and *high* directly.)

3.1.7. Ackerman's function is defined recursively on the nonnegative integers as follows:

$$a(m, n) = n + 1 \qquad \text{if } m = = 0$$
$$a(m, n) = a(m - 1, 1) \qquad \text{if } m! = 0, n = = 0$$
$$a(m, n) = a(m - 1, a(m, n - 1)) \qquad \text{if } m! = 0, n! = 0$$

- (a) Using the above definition, show that $a(2,2)$ equals 7.
- (b) Prove that $a(m,n)$ is defined for all nonnegative integers m and n.
- (c) Can you find an iterative method of computing $a(m,n)$?

3.1.8. Count the number of additions necessary to compute *fib*(n) for $0 <= n <= 10$ by the iterative and recursive methods. Does a pattern emerge?

3.1.9. If an array contains n elements, what is the maximum number of recursive calls made by the binary search algorithm?

3.2 RECURSION IN C

Factorial in C

The C language allows a programmer to write subroutines and functions that call themselves. Such routines are called ***recursive***.

The recursive algorithm to compute $n!$ may be directly translated into a C function as follows:

```
int fact(int n)
{
   int x, y;

   if (n == 0)
      return(1);
   x = n-1;
   y = fact(x);
   return(n * y);
} /* end fact */
```

In the statement $y = fact(x)$; the function *fact* calls itself. This is the essential ingredient of a recursive routine. The programmer assumes that the function being computed has already been written and uses it in its own definition. However, the programmer must ensure that this does not lead to an endless series of calls.

Let us examine the execution of this function when it is called by another program. For example, suppose that the calling program contains the statement

```
printf("%d", fact(4));
```

When the calling routine calls *fact*, the parameter n is set equal to 4. Since n is not 0, x is set equal to 3. At that point, *fact* is called a second time with an argument of 3. Therefore, the function *fact* is reentered and the local variables (x and y) and parameter (n) of the block are reallocated. Since execution has not yet left the first call of *fact*, the first allocation of these variables remains. Thus there are two generations of each of these variables in existence simultaneously. From any point within the second execution of *fact*, only the most recent copy of these variables can be referenced.

In general, each time the function *fact* is entered recursively, a new set of local variables and parameters is allocated, and only this new set may be referenced within that call of *fact*. When a return from *fact* to a point in a previous call takes place, the most recent allocation of these variables is freed, and the previous copy is reactivated. This previous copy is the one that was allocated upon the original entry to the previous call and is local to that call.

This description suggests the use of a stack to keep the successive generations of local variables and parameters. This stack is maintained by the C system and is invisible to the user. Each time that a recursive function is entered, a new allocation of its variables is pushed on top of the stack. Any reference to a local variable or parameter is through the current top of the stack. When the function returns, the stack is popped,

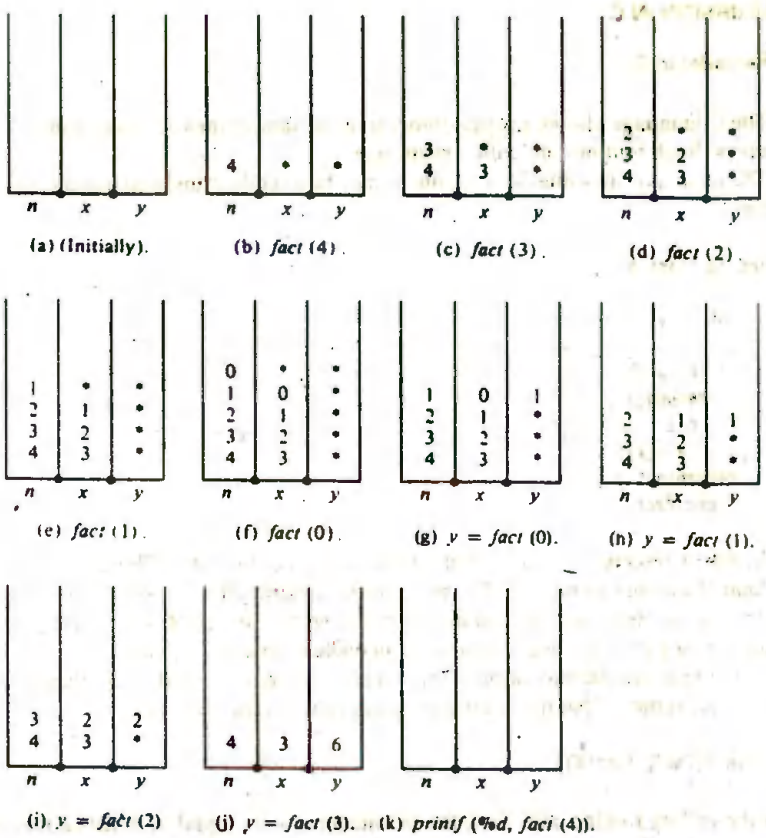


Figure 3.2.1 Stack at various times during execution. (An asterisk indicates an uninitialized value.)

the top allocation is freed, and the previous allocation becomes the current stack top to be used for referencing local variables. This mechanism is examined more closely in Section 3.4, but for now, let us see how it is applied in computing the factorial function.

Figure 3.2.1 contains a series of snapshots of the stacks for the variables n, x, and y as execution of the *fact* function proceeds. Initially, the stacks are empty, as illustrated by Figure 3.2.1a. After the first call on *fact* by the calling procedure, the situation is as shown in Figure 3.2.1b, with n equal to 4. The variables x and y are allocated but not initialized. Since n does not equal 0, x is set to 3 and *fact*(3) is called (Figure 3.2.1c). The new value of n does not equal 0; therefore x is set to 2 and *fact*(2) is called (Figure 3.2.1d).

This continues until n equals 0 (Figure 3.2.1f). At that point the value 1 is returned from the call to *fact*(0). Execution resumes from the point at which *fact*(0) was called,

which is the assignment of the returned value to the copy of y declared in *fact*(1). This is illustrated by the status of the stack shown in Figure 3.2.1g, where the variables allocated for *fact*(0) have been freed and y is set to 1.

The statement ***return***(n * y) is then executed, multiplying the top values of n and y to obtain 1 and returning this value to *fact*(2) (Figure 3.2.1h). This process is repeated twice more, until finally the value of y in *fact*(4) equals 6 (Figure 3.2.1j). The statement ***return***(n * y) is executed one more time. The product 24 is returned to the calling procedure where it is printed by the statement

```
printf("%d", fact(4));
```

Note that each time that a recursive routine returns, it returns to the point immediately following the point from which it was called. Thus, the recursive call to *fact*(3) returns to the assignment of the result to y within *fact*(4), but the recursive call to *fact*(4) returns to the *printf* statement in the calling routine.

Let us transform some of the other recursive definitions and processes of the previous section into recursive C programs. It is difficult to conceive of a C programmer writing a function to compute the product of two positive integers in terms of addition, since an asterisk performs the multiplication directly. Nevertheless, such a function can serve as another illustration of recursion in C. Following closely the definition of multiplication in the previous section, we may write:

```
int mult(int a, int b)
{
   return(b == 1 ? a : mult(a, b-1) + a);
} /* end mult */
```

Notice how similar this program is to the recursive definition of the last section. We leave it as an exercise for you to trace through the execution of this function when it is called with two positive integers. The use of stacks is a great aid in this tracing process.

This example illustrates that a recursive function may invoke itself even within a statement assigning a value to the function. Similarly, we could have written the recursive *fact* function more compactly as

```
int fact(int n)
{
   return(n == 0 ? 1 : n * fact(n-1));
} /*  end fact */
```

This compact version avoids the explicit use of local variables x (to hold the value of $n - 1$) and y (to hold the value of *fact*(x)). However, temporary locations are set aside anyway for these two values upon each invocation of the function. These temporaries are treated just as any explicit local variable. Thus, in tracing the action of a recursive routine, it may be helpful to declare all temporary variables explicitly. See if it is any easier to trace the following more explicit version of *mult*:

```
int mult(int a, int b)
{
   int c, d, sum;

   if (b == 1)
      return(a);
   c = b-1;
   d = mult(a, c);
   sum = d+a;
   return(sum);
} /* end mult */
```

Another point that should be made is that it is particularly important to check for the validity of input parameters in a recursive routine. For example, let us examine the execution of the *fact* function when it is invoked by a statement such as

```
printf("\n%d", fact(-1));
```

Of course, the *fact* function is not designed to produce a meaningful result for negative input. However, one of the most important things for a programmer to learn is that a function invariably will be presented at some time with invalid input and, unless provision is made for such input, the resultant error may be very difficult to trace.

For example, when -1 is passed as a parameter to *fact*, so that n equals -1, x is set to -2 and -2 is passed to a recursive call on *fact*. Another set of n, x, and y is allocated, n is set to -2, and x becomes -3. This process continues until the program either runs out of time or space or the value of x becomes too small. No message indicating the true cause of the error is produced.

If *fact* were originally called with a complicated expression as its argument and the expression erroneously evaluated to a negative number, a programmer might spend hours searching for the cause of the error. The problem can be remedied by revising the *fact* function to check its input explicitly, as follows:

```
int fact(int n)
{
   int x, y;

   if (n < 0) {
      printf("%s", "negative parameter in the factorial function");
      exit(1);
   } /* end if */
   if (n == 0)
      return(1);
   x = n-1;
   y = fact(x);
   return(n * y);
} /* end fact */
```

Similarly, the function *mult* must guard against a nonpositive value in the second parameter.

Fibonacci Numbers in C

We now turn our attention to the Fibonacci sequence. A C program to compute the *n*th Fibonacci number can be modeled closely after the recursive definition:

```
int fib(int n)
{
   int x, y;

   if (n <= 1)
      return(n);
   x = fib(n-1);
   y = fib(n-2);
   return(x + y);
} /* end fib */
```

Let us trace through the action of this function in computing the sixth Fibonacci number. You may compare the action of the routine with the manual computation we performed in the last section to compute *fib*(6). The stacking process is illustrated in Figure 3.2.2. When the program is first called, the variables n, x, and y are allocated, and n is set to 6 (Figure 3.2.2a). Since $n > 1$, $n - 1$ is evaluated and *fib* is called recursively. A new set of n, x, and y is allocated, and n is set to 5 (Figure 3.2.2b). This process continues (Figure 3.2.2c–f) with each successive value of n being one less than its predecessor, until *fib* is called with n equal to 1. The sixth call to *fib* returns 1 to its caller, so that the fifth allocation of x is set to 1 (Figure 3.2.2g).

The next sequential statement, $y = fib(n - 2)$, is then executed. The value of n that is used is the most recently allocated one, which is 2. Thus we again call on *fib* with an argument of 0 (Figure 3.2.2h). The value of 0 is immediately returned, so that y in *fib*(2) is set to 0 (Figure 3.2.2i). Note that each recursive call results in a return to the point of call, so that the call of *fib*(1) returns to the assignment to x, and the call of *fib*(0) returns to the assignment to y. The next statement to be executed in *fib*(2) is the statement that returns $x + y = 1 + 0 = 1$ to the statement that calls *fib*(2) in the generation of the function calculating *fib*(3). This is the assignment to x, so that x in *fib*(3) is given the value *fib*(2) = 1 (Figure 3.2.2j). The process of calling and pushing and returning and popping continues until finally the routine returns for the last time to the main program with the value 8. Figure 3.2.2 shows the stack up to the point where *fib*(5) calls on *fib*(3), so that its value can be assigned to y. The reader is urged to complete the picture by drawing the stack states for the remainder of the program execution.

This program illustrates that a recursive routine may call itself a number of times with different arguments. In fact, as long as a recursive routine uses only local variables, the programmer can use the routine just as he or she uses any other and assume that it performs its function and produces the desired value. He or she need not worry about the underlying stacking mechanism.

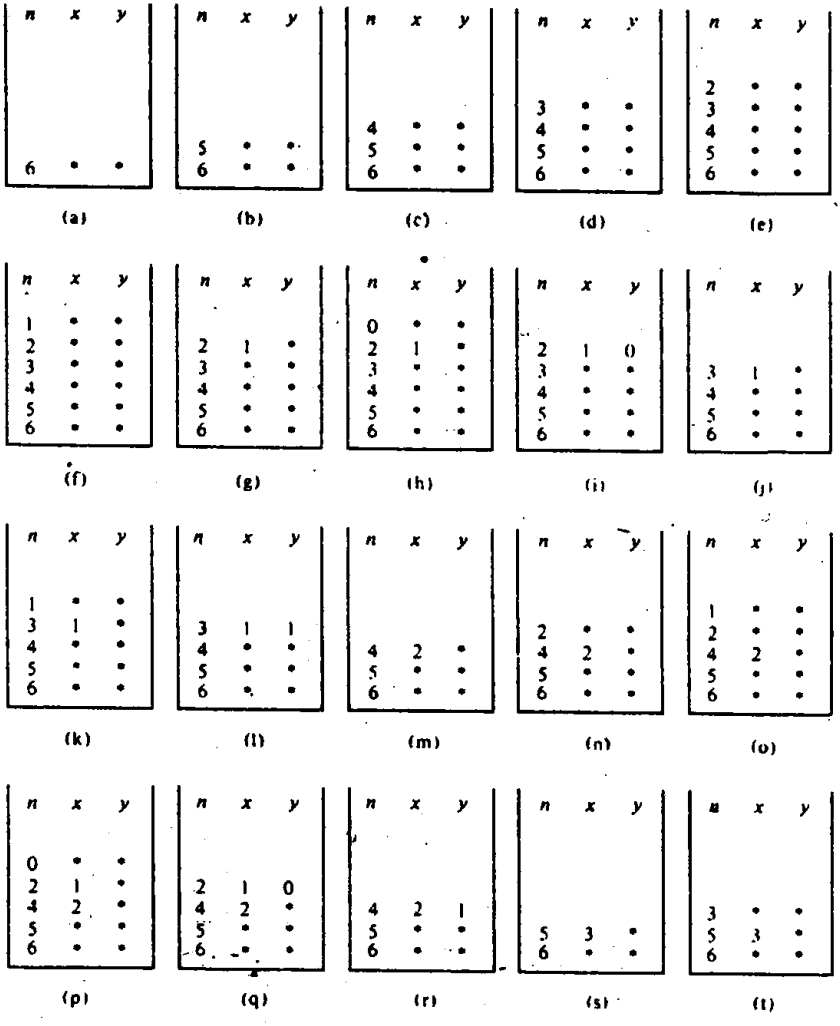


Figure 3.2.2 The recursion stack of the Fibonacci function.

Binary Search in C

Let us now present a C program for the binary search. A function to do this accepts an array a and an element x as input and returns the index i in a such that $a[i]$ equals x, or -1 if no such i exists. Thus the function *binsrch* might be invoked in a statement such as

```
i = binsrch(a, x)
```

However, in looking at the binary search algorithm of Section 3.1 as a model for a recursive C routine, we note that two other parameters are passed in the recursive calls. Lines 7 and 9 of that algorithm call for a binary search on only part of the array. Thus, for the function to be recursive the bounds between which the array is to be searched must also be specified. The routine is written as follows:

```
int binsrch(int a[], int x, int low, int high)
{
   int mid;

   if (low > high)
      return(-1);
   mid = (low + high) / 2;
   return(x == a[mid] ? mid : x < a[mid] ?
                              binsrch(a, x, low, mid-1) :
                              binsrch(a, x, mid+1, high));
} /* end binsrch */
```

When *binsrch* is first called from another routine to search for x in an array declared by

```
int a[ARRAYSIZE]
```

of which the first n elements are occupied, it is called by the statement

```
i = binsrch(a, x, 0, n-1);
```

You are urged to trace the execution of this routine and follow the stacking and unstacking using the example of the preceding section, where a is an array of 8 elements ($n = 8$) containing 1, 3, 4, 5, 17, 18, 31, 33, in that order. The value being searched for is 17 (x equals 17). Note that the array a is stacked for each recursive call. The values of *low* and *high* are the lower and upper bounds of the array a, respectively.

In the course of tracing through the *binsrch* routine, you may have noticed that the values of the two parameters a and x do not change throughout its execution. Each time that *binsrch* is called the same array is searched for the same element; it is only the upper and lower bounds of the search that change. It therefore seems wasteful to stack and unstack these two parameters each time the routine is called recursively.

One solution is to allow a and x to be global variables, declared before the program by

```
int a[ARRAYSIZE];
int x;
```

The routine is called by a statement such as

```
i = binsrch(0, n-1)
```

In this case, all references to *a* and *x* are to the global allocations of *a* and *x* declared at the beginning of the source file. This enables *binsrch* to access *a* and *x* without allocating additional space for them. All multiple allocations and freeings of space for these parameters are eliminated.

We may rewrite the *binsrch* function as follows:

```
int binsrch(int low, int high)
{
   int mid;

   if (low > high)
      return(-1);
   mid = (low + high) / 2;
   return (x == a[mid] ? mid : x < a[mid] ? binsrch(low, mid-1) :
                                            binsrch(mid+1, high));
} /* end binsrch */
```

Using this scheme, the variables *a* and *x* are referenced with the *extern* attribute and are not passed with each recursive call to *binsrch*. *a* and *x* do not change their values and are not stacked. The programmer wishing to make use of *binsrch* in a program only needs to pass the parameters *low* and *high*. The routine could be invoked with a statement such as

```
i = binsrch(low, high);
```

Recursive Chains

A recursive function need not call itself directly. Rather, it may call itself indirectly, as in the following example:

```
a(formal parameters)        b(formal parameters)
{                           {
     .                           .
     .                           .
     .                           .
   b(arguments);               a(arguments);
     .                           .
} /*end a*/                 } /*end b*/
```

In this example function *a* calls *b*, which may in turn call *a*, which may again call *b*. Thus both *a* and *b* are recursive, since they indirectly call on themselves. However, the fact that they are recursive is not evident from examining the body of either of the routines individually. The routine *a* seems to be calling a separate routine *b* and it is impossible to determine, by examining *a* alone, that it may call itself indirectly.

More than two routines may participate in a ***recursive chain***. Thus a routine *a* may call *b* which calls *c*, ... which calls z, which calls *a*. Each routine in the chain may

potentially call itself and is therefore recursive. Of course, the programmer must ensure that such a program does not generate an infinite sequence of recursive calls.

Recursive Definition of Algebraic Expressions

As an example of a recursive chain, consider the following recursive group of definitions:

1. An ***expression*** is a *term* followed by a *plus sign* followed by a *term*, or a *term* alone.
2. A ***term*** is a *factor* followed by an *asterisk* followed by a *factor*, or a *factor* alone.
3. A ***factor*** is either a *letter* or an *expression* enclosed in *parentheses*.

Before looking at some examples, note that none of the foregoing three items is defined directly in terms of itself. However, each is defined in terms of itself indirectly. An expression is defined in terms of a term, a term in terms of a factor, and a factor in terms of an expression. Similarly, a factor is defined in terms of an expression, which is defined in terms of a term, which is defined in terms of a factor. Thus the entire set of definitions forms a recursive chain.

Let us now give some examples. The simplest form of a factor is a letter. Thus *A*, *B*, *C*, *Q*, *Z*, *M* are all factors. They are also terms, since a term may be a factor alone. They are also expressions, since an expression may be a term alone. Since *A* is an expression, (*A*) is a factor and therefore a term as well as an expression. $A + B$ is an example of an expression that is neither a term nor a factor. $(A + B)$, however, is all three. $A * B$ is a term and therefore an expression, but it is not a factor. $A * B + C$ is an expression that is neither a term nor a factor. $A * (B + C)$ is a term and an expression but not a factor.

Each of the foregoing examples is a valid expression. This can be shown by applying the definition of an expression to each of them. Consider, however, the string $A + *B$. It is neither an expression, term, nor factor. It would be instructive for you to attempt to apply the definitions of expression, term, and factor to see that none of them describe the string $A + *B$. Similarly, $(A + B*)C$ and $A + B + C$ are not valid expressions according to the preceding definitions.

Let us write a program that reads and prints a character string and then prints "valid" if it is a valid expression and "invalid" if it is not. We use three functions to recognize expressions, terms, and factors, respectively. First, however, we present an auxiliary function *getsymb* that operates on three parameters: *str*, *length*, and *ppos*. *str* contains the input character string. *length* represents the number of characters in *str*. *ppos* points to an integer *pos* whose value is the position in *str* from which we last obtained a character. If *pos* < *length*, *getsymb* returns the character *str[pos]* and increments *pos* by 1. If *pos* > = *length*, *getsymb* returns a blank.

```
int getsymb(char str[], int length, int *ppos)
{
   char c;
```

```
    if (*ppos < length)
       c = str[*ppos];
    else
       c = ' ';
    (*ppos)++;
    return(c);
} /* end getsymb */
```

The function that recognizes an expression is called *expr*. It returns *TRUE* (or 1) if a valid expression begins at position *pos* of *str* and *FALSE* (or 0) otherwise. It also resets *pos* to the position following the longest expression it can find. We also assume a function *readstr* that reads a string of characters, placing the string in *str* and its length in *length*.

Having described the functions *expr* and *readstr*, we can write the main routine as follows. The standard library *ctype.h* includes a function *isalpha* called by one of the functions below.

```
#include <stdio.h>
#include <ctype.h>
#define TRUE 1
#define FALSE 0
#define MAXSTRINGSIZE 100

void readstr(char *, int);
int expr(char *, int, int *);
int term(char *, int, int *);
int getsymb(char *, int, int *);
int factor(char *, int, int *);

void main()
{
   char str[MAXSTRINGSIZE];
   int length, pos;

   readstr(str, &length);
   pos = 0;
   if (expr(str, length, &pos) == TRUE && pos >= length)
      printf("%s", "valid");
   else
      printf("%s", "invalid");
   /* The condition can fail for one (or both) of two  */
   /*  reasons. If expr(str, length, &pos) == FALSE    */
   /*  then there is no valid expression beginning at  */
   /*    pos. If pos < length there may be a valid     */
   /*    expression starting at pos but it does not    */
   /*           occupy the entire string.              */
} /* end main */
```

The functions *factor* and *term* are much like *expr* except that they are responsible for recognizing factors and terms, respectively. They also reposition *pos* to the position following the longest factor or term within the string *str* that they can find.

The code for these routines adheres closely to the definitions given earlier. Each of the routines attempts to satisfy one of the criteria for the entity being recognized. If one of these criteria is satisfied, *TRUE* is returned. If none of these criteria are satisfied, *FALSE* is returned.

```
int expr(char str[], int length, int *ppos)
{
   /*          look for a term                    */
   if (term(str, length, ppos) == FALSE)
      return(FALSE);
   /*     We have found a term; look at the       */
   /*                next symbol.                 */
   if (getsymb(str, length, ppos) != '+') {
      /*   We have found the longest expression  */
      /*  (a single term). Reposition pos so it  */
      /*     refers to the last position of      */
      /*              the expression.            */
      (*ppos)--;
      return(TRUE);
   } /* end if */
   /* At this point, we have found a term and a  */
   /*  plus sign. We must look for another term. */
   return(term(str, length, ppos));
} /* end expr */
```

The routine *term* that recognizes terms is very similar, and we present it without comments.

```
int term(char str[], int length, int *ppos)
{
   if (factor(str, length, ppos) == FALSE)
      return(FALSE);
   if (getsymb(str, length, ppos) != '*') {
      (*ppos)--;
      return(TRUE);
   } /* end if */
   return(factor(str, length, ppos));
} /* end term */
```

The function *factor* recognizes factors and should now be fairly straightforward. It uses the common library routine *isalpha* (this function is contained in the library *ctype.h*), which returns nonzero if its character parameter is a letter and zero (or *FALSE*) otherwise.

```
int factor(char str[], int length, int *ppos)
{
   int c;

   if ((c = getsymb(str, length, ppos)) != '(')
      return(isalpha(c));
   return(expr(str, length, ppos) &&
                        getsymb(str, length, ppos) == ')');
} /* end factor */
```

All three routines are recursive, since each may call itself indirectly. For example, if you trace through the actions of the program for the input string "$(a * b + c * d) + (e * (f) + g)$," you will find that each of the routines *expr*, *term*, and *factor* calls on itself.

EXERCISES

3.2.1. Determine what the following recursive C function computes. Write an iterative function to accomplish the same purpose.

```
int func(int n)
{
   if (n == 0)
      return(0);
   return(n + func(n-1));
} /* end func */
```

3.2.2. The C expression $m \% n$ yields the remainder of m upon division by n. Define the ***greatest common divisor (GCD)*** of two integers x and y by

```
gcd(x,y) = y                 if (y <= x && x % y == 0)
gcd(x,y) = gcd(y,x)          if (x < y)
gcd(x,y) = gcd(y, x % y)     otherwise
```

Write a recursive C function to compute $gcd(x,y)$. Find an iterative method for computing this function.

3.2.3. Let $comm(n,k)$ represent the number of different committees of k people that can be formed, given n people from whom to choose. For example, $comm(4,3) = 4$, since given four people A, B, C, and D there are four possible three-person committees: ABC, ABD, ACD, and BCD. Prove the identity:

```
comm(n,k) = comm(n - 1,k) + comm(n - 1,k - 1)
```

Write and test a recursive C program to compute $comm(n,k)$ for $n, k >= 1$.

3.2.4. Define a ***generalized fibonacci sequence*** of $f0$ and $f1$ as the sequence $gfib(f0,f1,0)$, $gfib(f0,f1,1)$, $gfib(f0,f1,2)$, . . . , where

```
gfib(f0, f1, 0) = f0
gfib(f0, f1, 1) = f1
gfib(f0, f1, n) = gfib(f0, f1, n - 1)
                + gfib(f0, f1, n - 2) if n > 1
```

Write a recursive C function to compute *gfib*(*f*0, *f*1,*n*). Find an iterative method for computing this function.

3.2.5. Write a recursive C function to compute the number of sequences of n binary digits that do not contain two 1s in a row. (*Hint:* Compute how many such sequences exist that start with 0, and how many exist that start with a 1.).

3.2.6. An ***order n matrix*** is an $n \times n$ array of numbers. For example,

(3)

is a 1×1 matrix,

$$\begin{matrix} 1 & 3 \\ -2 & 8 \end{matrix}$$

is a 2×2 matrix and

$$\begin{matrix} 1 & 3 & 4 & 6 \\ 2 & -5 & 0 & 8 \\ 3 & 7 & 6 & 4 \\ 2 & 0 & 9 & -1 \end{matrix}$$

is a 4×4 matrix. Define the ***minor*** of an element x in a matrix as the submatrix formed by deleting the row and column containing x. In the preceding example of a 4×4 matrix, the minor of the element 7 is the 3×3 matrix

$$\begin{matrix} 1 & 4 & 6 \\ 2 & 0 & 8 \\ 2 & 9 & -1 \end{matrix}$$

Clearly the order of a minor of any element is 1 less than the order of the original matrix. Denote the minor of an element $a[i,j]$ by *minor*($a[i,j]$).

Define the ***determinant*** of a matrix a (written *det*(a)) recursively as follows:

1. If a is a 1×1 matrix (x), *det*(a) = x.
2. If a is of an order greater than 1, compute the determinant of a as follows:

(a) Choose any row or column. For each element $a[i,j]$ in this row or column form the product

```
power(-1, i + j) * a[i,j] * det(minor(a[i,j]))
```

where i and j are the row and column positions of the element chosen, $a[i,j]$ is the element chosen, *det*(*minor*($a[i,j]$)) is the determinant of the minor of $a[i,j]$, and *power*(m,n) is the value of m raised to the nth power.

(b) *det*(a) = sum of all these products.

(More concisely, if n is the order of a,

$$det(a) = \sum_i power(-1,\ i + j) * a[i,j] * det(minor(a[i,j])),\ \text{for any } j$$

or

$$det(a) = \sum_j power(-1,\ i + j) * a[i,j] * det(minor(a[\ i,j])),\ \text{for any } i).$$

Write a C program that reads a, prints a in matrix form, and prints the value of $det(a)$, where *det* is a function that computes the determinant of a matrix.

3.2.7. Write a recursive C program to sort an array a as follows:

1. Let k be the index of the middle element of the array.
2. Sort the elements up to and including $a[k]$.
3. Sort the elements past $a[k]$.
4. Merge the two subarrays into a single sorted array.

This method is called a ***merge sort***.

3.2.8. Show how to transform the following iterative procedure into a recursive procedure. $f(i)$ is a function returning a logical value based on the value of i, and $g(i)$ is a function that returns a value with the same attributes as i.

```
void iter(int n)
{
   int i;

   i = n;
   while(f(i) == TRUE) {
      /* any group of C statements that */
      /* does not change the value of i */
      i = g(i);
   } /* end while */
} /* end iter */
```

3.3 WRITING RECURSIVE PROGRAMS

In the last section we saw how to transform a recursive definition or algorithm into a C program. It is a much more difficult task to develop a recursive C solution to a problem specification whose algorithm is not supplied. It is not only the program but also the original definitions and algorithms that must be developed. In general, when faced with the task of writing a program to solve a problem there is no reason to look for a recursive solution. Most problems can be solved in a straightforward manner using nonrecursive methods. However, some problems can be solved logically and most elegantly by recursion. In this section we shall try to identify those problems that can be solved recursively, develop a technique for finding recursive solutions, and present some examples.

Let us reexamine the factorial function. Factorial is probably a prime example of a problem that should not be solved recursively, since the iterative solution is so direct and

simple. However, let us examine the elements that make the recursive solution work. First of all, we can recognize a large number of distinct cases to solve. That is, we want to write a program to compute 0!, 1!, 2!, and so on. We can also identify a "trivial" case for which a nonrecursive solution is directly obtainable. This is the case of 0!, which is defined as 1. The next step is to find a method of solving a "complex" case in terms of a "simpler" case. This allows reduction of a complex problem to a simpler problem. The transformation of the complex case to the simpler case should eventually result in the trivial case. This would mean that the complex case is ultimately defined in terms of the trivial case.

Let us examine what this means when applied to the factorial function. 4! is a more "complex" case than 3!. The transformation that is applied to the number 4 to obtain the number 3 is simply the subtraction of 1. Repeatedly subtracting 1 from 4 eventually results in 0, which is a "trivial" case. Thus if we are able to define 4! in terms of 3!, and in general $n!$ in terms of $(n - 1)!$, we will be able to compute 4! by first working our way down to 0! and then working our way back up to 4! using the definition of $n!$ in terms of $(n - 1)!$. In the case of the factorial function we have such a definition, since

```
n! = n * (n - 1)!
```

Thus $4! = 4 * 3! = 4 * 3 * 2! = 4 * 3 * 2 * 1! = 4 * 3 * 2 * 1 * 0! = 4 * 3 * 2 * 1 * 1 = 24$.

These are the essential ingredients of a recursive routine—being able to define a "complex" case in terms of a "simpler" case and having a directly solvable (nonrecursive) "trivial" case. Once this has been done, one can develop a solution using the assumption that the simpler case has already been solved. The C version of the factorial function assumes that $(n - 1)!$ is defined and uses that quantity in computing $n!$.

Let us see how these ideas apply to other examples of the previous sections. In defining $a * b$, the case of $b = 1$ is trivial, since in that case, $a * b$ is defined as a. In general, $a * b$ may be defined in terms of $a * (b - 1)$ by the definition $a * b = a * (b - 1) + a$. Again the complex case is transformed into a simpler case by subtracting 1, eventually leading to the trivial case of $b = 1$. Here the recursion is based on the second parameter, b, alone.

In the case of the Fibonacci function, two trivial cases were defined: $fib(0) = 0$ and $fib(1) = 1$. A complex case, $fib(n)$, is then reduced to two simpler cases: $fib(n - 1)$ and $fib(n - 2)$. It is because of the definition of $fib(n)$ as $fib(n - 1) + fib(n - 2)$ that two trivial cases directly defined are necessary. $fib(1)$ cannot be defined as $fib(0) + fib(-1)$, because the Fibonacci function is not defined for negative numbers.

The binary search function is an interesting case of recursion. The recursion is based on the number of elements in the array that must be searched. Each time the routine is called recursively, the number of elements to be searched is halved (approximately). The trivial case is the one in which there are either no elements to be searched or the element being searched for is at the middle of the array. If $low > high$, the first of these two conditions holds and -1 is returned. If $x = a[mid]$, the second condition holds and mid is returned as the answer. In the more complex case of $high - low + 1$ elements to be searched, the search is reduced to taking place in one of two subregions.

1. The lower half of the array from *low* to *mid* − 1
2. The upper half of the array from *mid* + 1 to *high*

Thus a complex case (a large area to be searched) is reduced to a simpler case (an area to be searched of approximately half the size of the original area). This eventually reduces to a comparison with a single element (*a*[*mid*]) or a search within an array of no elements.

The Towers of Hanoi Problem

Thus far we have been looking at recursive definitions and examining how they fit the pattern we have established. Let us now look at a problem that is not specified in terms of recursion and see how we can use recursive techniques to produce a logical and elegant solution. The problem is the "Towers of Hanoi" problem whose initial setup is shown in Figure 3.3.1. Three pegs, *A*, *B*, and *C*, exist. Five disks of differing diameters are placed on peg *A* so that a larger disk is always below a smaller disk. The object is to move the five disks to peg *C*, using peg *B* as auxiliary. Only the top disk on any peg may be moved to any other peg, and a larger disk may never rest on a smaller one. See if you can produce a solution. Indeed, it is not even apparent that a solution exists.

Let us see if we can develop a solution. Instead of focusing our attention on a solution for five disks, let us consider the general case of *n* disks. Suppose that we had a solution for $n - 1$ disks and could state a solution for *n* disks in terms of the solution for $n - 1$ disks. Then the problem would be solved. This is true because in the trivial case of one disk (continually subtracting 1 from *n* will eventually produce 1), the solution is simple: merely move the single disk from peg *A* to peg *C*. Therefore we will have developed a recursive solution if we can state a solution for *n* disks in terms of $n - 1$. See if you can find such a relationship. In particular, for the case of five disks, suppose that we knew how to move the top four disks from peg *A* to another peg according to the rules. How could we then complete the job of moving all five? Recall that there are three pegs available.

Suppose that we could move four disks from peg *A* to peg *C*. Then we could move them just as easily to *B*, using *C* as auxiliary. This would result in the situation depicted in Figure 3.3.2a. We could then move the largest disk from *A* to *C* (Figure 3.3.2b) and finally again apply the solution for four disks to move the four disks from *B* to *C*, using

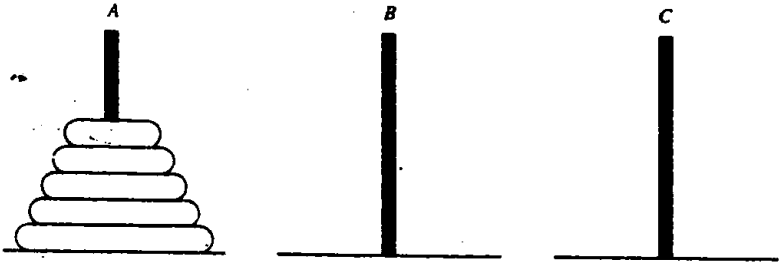


Figure 3.3.1 Initial setup of the Towers of Hanoi.

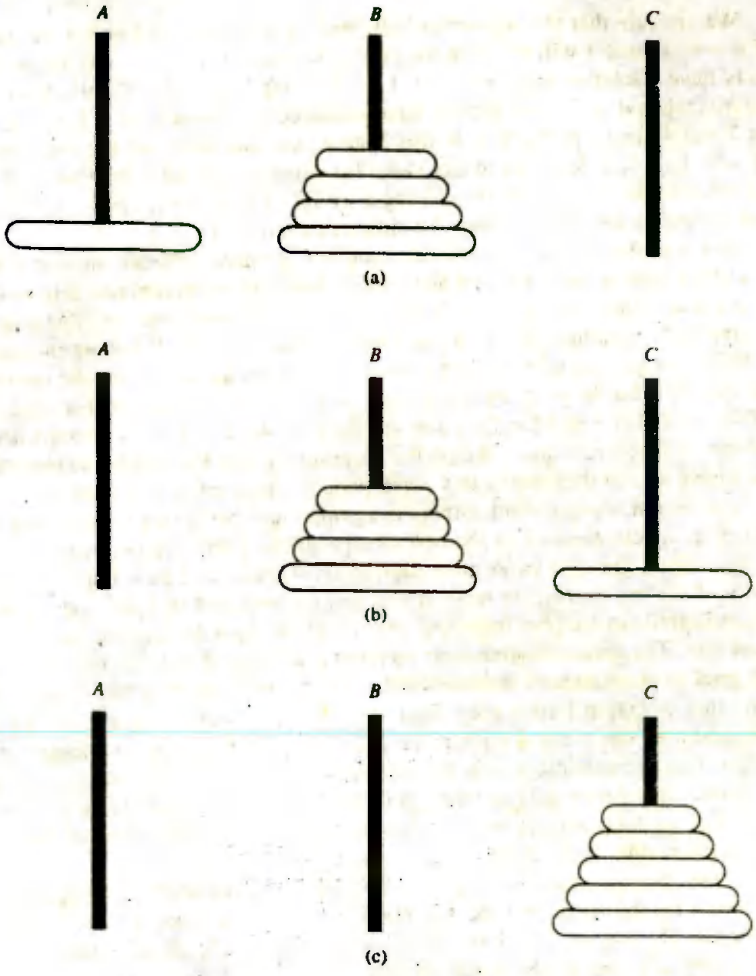


Figure 3.3.2 Recursive solution to the Towers of Hanoi.

the now empty peg A as an auxiliary (Figure 3.3.2c). Thus, we may state a recursive solution to the Towers of Hanoi problem as follows:

To move n disks from A to C, using B as auxiliary:

1. If $n == 1$, move the single disk from A to C and stop.
2. Move the top $n - 1$ disks from A to B, using C as auxiliary.
3. Move the remaining disk from A to C.
4. Move the $n - 1$ disks from B to C, using A as auxiliary.

We are sure that this algorithm will produce a correct solution for any value of n. If $n == 1$, step 1 will result in the correct solution. If $n == 2$, we know that we already have a solution for $n - 1 == 1$, so that steps 2 and 4 will perform correctly. Similarly, when $n == 3$, we already have produced a solution for $n - 1 == 2$, so that steps 2 and 4 can be performed. In this fashion, we can show that the solution works for $n == 1, 2, 3, 4, 5, \ldots$ up to any value for which we desire a solution. Notice that we developed the solution by identifying a trivial case ($n == 1$) and a solution for a general complex case (n) in terms of a simpler case ($n - 1$).

How can this solution be converted into a C program? We are no longer dealing with a mathematical function such as factorial, but rather with concrete actions such as "move a disk." How are we to represent such actions in the computer? The problem is not completely specified. What are the inputs to the program? What are its outputs to be? Whenever you are told to write a program, you must receive specific instructions about exactly what the program is expected to do. A problem statement such as "Solve the Towers of Hanoi problem" is quite insufficient. What is usually meant when such a problem is specified is that not only the program but also the inputs and outputs must be designed, so that they reasonably correspond to the problem description.

The design of inputs and outputs is an important phase of a solution and should be given as much attention as the rest of a program. There are two reasons for this. The first is that the user (who must ultimately evaluate and pass judgment on your work) will not see the elegant method that you incorporated in your program but will struggle mightily to decipher the output or to adapt the input data to your particular input conventions. The failure to agree early on input and output details has been the cause of much grief to programmers and users alike. The second reason is that a slight change in the input or output format may make the program much simpler to design. Thus, the programmer can make the job much easier if he or she is able to design an input or output format compatible with the algorithm. Of course these two considerations, convenience to the user and convenience to the programmer, often conflict sharply, and some happy medium must be found. However, the user as well as the programmer must be a full participant in the decisions on input and output formats.

Let us, then, proceed to design the inputs and outputs for this program. The only input needed is the value of n, the number of disks. At least that may be the programmer's view. The user may want the names of the disks (such as "red," "blue," "green," and so forth) and perhaps the names of the pegs (such as "left," "right," and "middle") as well. The programmer can probably convince the user that naming the disks 1, 2, 3, . . . , n and the pegs A, B, C is just as convenient. If the user is adamant, the programmer can write a small function to convert the user's names to his or her own and vice versa.

A reasonable form for the output would be a list of statements such as

```
move disk nnn from peg yyy to peg zzz
```

where *nnn* is the number of the disk to be moved, and *yyy* and *zzz* are the names of the pegs involved. The action to be taken for a solution would be to perform each of the output statements in the order that they appear in the output.

The programmer then decides to write a subroutine *towers* (being purposely vague about the parameters at this point) to print the aforementioned output. The main program would be

```
void main()
{
   int n;

   scanf("%d", &n);
   towers(parameters);
}/* end main */
```

Let us assume that the user will be satisfied to name the disks 1, 2, 3, ..., n and the pegs A, B, and C. What should the parameters to *towers* be? Clearly, they should include n, the number of disks to be moved. This not only includes information about how many disks there are but also what their names are. The programmer then notices that, in the recursive algorithm, $n - 1$ disks will have to be moved using a recursive call to *towers*. Thus, on the recursive call, the first parameter to *towers* will be $n - 1$. But this implies that the top $n - 1$ disks are numbered 1, 2, 3, ..., $n - 1$ and that the smallest disk is numbered 1. This is a good example of programming convenience determining problem representation. There is no a priori reason for labeling the smallest disk 1; logically the largest disk could have been labeled 1 and the smallest disk n. However, since it leads to a simpler and more direct program, we choose to label the disks so that the smallest disk has the smallest number.

What are the other parameters to *towers*? At first glance, it might appear that no additional parameters are necessary, since the pegs are named A, B, and C by default. However, a closer look at the recursive solution leads us to the realization that on the recursive calls disks will not be moved from A to C using B as auxiliary but rather from A to B using C (step 2) or from B to C using A (step 4). We therefore include three more parameters in *towers*. The first, *frompeg*, represents the peg from which we are removing disks; the second, *topeg*, represents the peg to which we will take the disks; and the third, *auxpeg*, represents the auxiliary peg. This situation is one which is quite typical of recursive routines; additional parameters are necessary to handle the recursive call situation. We already saw one example of this in the binary search program where the parameters *low* and *high* were necessary.

The complete program to solve the Towers of Hanoi problem, closely following the recursive solution, may be written as follows:

```
#include <stdio.h>

void towers(int, char, char, char);

void main()
{
   int n;

   scanf("%d", &n);
   towers(n, 'A', 'C', 'B');
}/* end main */
```

```
void towers(int n, char frompeg, char topeg, char auxpeg)
{
   /*  If only one disk, make the move and return.  */
   if (n == 1) {
      printf("\n%s%c%s%c", "move disk 1 from peg ", frompeg, " to peg ", topeg);
      return;
   } /* end if */
   /* Move top n-1 disks from A to B, using C as */
   /*                  auxiliary                 */
   towers(n-1, frompeg, auxpeg, topeg);
   /*      move remaining disk from A to C       */
   printf("\n%s%d%s%c%s%c", "move disk ", n, " from peg ",
                                          frompeg, " to peg ", topeg);
   /*   Move n-1 disk from B to C using A as     */
   /*                auxiliary                   */
   towers(n-1, auxpeg, topeg, frompeg);
}/* end towers */
```

Trace the actions of the foregoing program when it reads the value 4 for *n*. Be careful to keep track of the changing values of the parameters *frompeg*, *auxpeg*, and *topeg*. Verify that it produces the following output:

```
move disk 1 from peg A to peg B
move disk 2 from peg A to peg C
move disk 1 from peg B to peg C
move disk 3 from peg A to peg B
move disk 1 from peg C to peg A
move disk 2 from peg C to peg B
move disk 1 from peg A to peg B
move disk 4 from peg A to peg C
move disk 1 from peg B to peg C
move disk 2 from peg B to peg A
move disk 1 from peg C to peg A
move disk 3 from peg B to peg C
move disk 1 from peg A to peg B
move disk 2 from peg A to peg C
move disk 1 from peg B to peg C
```

Verify that the foregoing solution actually works and does not violate any of the rules.

Translation from Prefix to Postfix Using Recursion

Let us examine another problem for which the recursive solution is the most direct and elegant one. This is the problem of converting a prefix expression to postfix. Prefix and postfix notation were discussed in the last chapter. Briefly, prefix and postfix notation are methods of writing mathematical expressions without parentheses. In prefix notation each operator immediately precedes its operands. In postfix notation

each operator immediately follows its operands. To refresh your memory, here are a few conventional (infix) mathematical expressions with their prefix and postfix equivalents:

infix	*prefix*	*postfix*
$A + B$	$+AB$	$AB+$
$A + B * C$	$+A * BC$	$ABC * +$
$A * (B + C)$	$*A + BC$	$ABC + *$
$A * B + C$	$+ * ABC$	$AB * C+$
$A + B * C + D - E * F$	$- + +A * BCD * EF$	$ABC * +D + EF * -$
$(A + B) * (C + D - E) * F$	$** + AB - + CDEF$	$AB + CD + E - *F*$

The most convenient way to define postfix and prefix is by using recursion. Assuming no constants and using only single letters as variables, a prefix expression is a single letter, or an operator followed by two prefix expressions. A postfix expression may be similarly defined as a single letter, or as an operator preceded by two postfix expressions. The above definitions assume that all operations are binary—that is, that each requires two operands. Examples of such operations are addition, subtraction, multiplication, division, and exponentiation. It is easy to extend the preceding definitions of prefix and postfix to include unary operations such as negation or factorial, but in the interest of simplicity we will not do so here. Verify that each of the above prefix and postfix expressions are valid by showing that they satisfy the definitions and make sure that you can identify the two operands of each operator.

We will put these recursive definitions to use in a moment, but first let us return to our problem. Given a prefix expression, how can we convert it into a postfix expression? We can immediately identify a trivial case: if a prefix expression consists of only a single variable, that expression is its own postfix equivalent. That is, an expression such as A is valid as both a prefix and a postfix expression.

Now consider a longer prefix string. If we knew how to convert any shorter prefix string to postfix, could we convert this longer prefix string? The answer is yes, with one proviso. Every prefix string longer than a single variable contains an operator, a first operand, and a second operand (remember we are assuming binary operators only). Assume that we are able to identify the first and second operands, which are necessarily shorter than the original string. We can then convert the long prefix string to postfix by first converting the first operand to postfix, then converting the second operand to postfix and appending it to the end of the first converted operand, and finally appending the initial operator to the end of the resultant string. Thus we have developed a recursive algorithm for converting a prefix string to postfix, with the single provision that we must specify a method for identifying the operands in a prefix expression. We can summarize our algorithm as follows:

1. If the prefix string is a single variable, it is its own postfix equivalent.
2. Let *op* be the first operator of the prefix string.
3. Find the first operand, *opnd*1, of the string. Convert it to postfix and call it *post*1.

4. Find the second operand, *opnd2*, of the string. Convert it to postfix and call it *post2*.
5. Concatenate *post1*, *post2*, and *op*.

One operation that will be required in this program is that of concatenation. For example, if two strings represented by *a* and *b* represent the strings "abcde" and "xyz" respectively, the function call

```
strcat(a, b)
```

places into *a* the string "abcdexyz" (that is, the string consisting of all the elements of *a* followed by all the elements of *b*). We also require the functions *strlen* and *substr*. The function *strlen*(*str*) returns the length of the string *str*. The *substr*(*s*1,*i*,*j*,*s*2) function sets the string *s*2 to the substring of *s*1, starting at position *i* containing *j* characters. For example, after executing *substr*("abcd",1,2,*s*), *s* equals "bc". The functions *strcat*, *strlen*, and *substr* are usually standard C string library functions.

Before transforming the conversion algorithm into a C program, let us examine its inputs and outputs. We wish to write a procedure *convert* that accepts a character string. This string represents a prefix expression in which all variables are single letters and the allowable operators are '+', '−', '*', and '/'. The procedure produces a string that is the postfix equivalent of the prefix parameter.

Assume the existence of a function *find* that accepts a string and returns an integer that is the length of the longest prefix expression contained within the input string that starts at the beginning of that string. For example, *find* ("*A* + *CD*") returns 1, since "*A*" is the longest prefix string starting at the beginning of "*A* + *CD*". *find*("+ * *ABCD* + *GH*") returns 5, since "+ * *ABC*" is the longest prefix string starting at the beginning of "+ * *ABCD* + *GH*". If no such prefix string exists within the input string starting at the beginning of the input string, *find* returns 0. (For example, *find* ("* + *AB*") returns 0.) This function is used to identify the first and second operands of a prefix operator. *convert* also calls the library function *isalpha*, which determines if its parameter is a letter. Assuming the existence of the function *find*, a conversion routine may be written as follows.

```
void convert (char prefix[], char postfix[])
{
   char opnd1[MAXLENGTH], opnd2[MAXLENGTH];
   char post1[MAXLENGTH], post2[MAXLENGTH];
   char temp[MAXLENGTH];
   char op[1];
   int length;
   int i, j, m, n;

   if ((length = strlen(prefix)) == 1) {
      if (isalpha(prefix[0])) {
         /* The prefix string is a single letter. */
         postfix[0] = prefix[0];
```

```
        postfix[1] = '\0';
        return;
      } /*end if */
      printf("\nillegal prefix string");
      exit(1);
    } /* end if */
    /* The prefix string is longer than a single  */
    /* character. Extract the operator and the    */
    /*          two operand lengths.              */
    op[0] = prefix[0];
    op[1] = '\0';
    substr(prefix, 1, length-1, temp);
    m = find(temp);
    substr(prefix, m + 1, length-m-1, temp);
    n = find(temp);
    if ((op[0] != '+' && op[0] != '-' && op[0] != '*' && op[0] != '/')
                      || (m == 0) || (n == 0) || (m + n + 1 != length)) {
      printf("\nillegal prefix string");
      exit(1);
    } /* end if */
    substr(prefix, 1, m, opnd1);
    substr(prefix, m+1, n, opnd2);
    convert(opnd1, post1);
    convert(opnd2, post2);
    strcat(post1, post2);
    strcat(post1, op);
    substr(post1, 0, length, postfix);
}/* end convert */.
```

Note that several checks have been incorporated into *convert* to ensure that the parameter is a valid prefix string. One of the most difficult classes of errors to detect are those resulting from invalid inputs and the programmer's neglect to check for validity.

We now turn our attention to the function *find*, which accepts a character string and a starting position and returns the length of the longest prefix string that is contained in that input string starting at that position. The word "longest" in this definition is superfluous, since there is at most one substring starting at a given position of a given string that is a valid prefix expression.

We first show that there is at most one valid prefix expression starting at the beginning of a string. To see this, note that it is trivially true in a string of length 1. Assume that it is true for a short string. Then a long string that contains a prefix expression as an initial substring must begin with either a variable, in which case that variable is the desired substring, or with an operator. Deleting the initial operator, the remaining string is shorter than the original string and can therefore have at most a single initial prefix expression. This expression is the first operand of the initial operator. Similarly, the remaining substring (after deleting the first operand) can only have a single initial substring that is a prefix expression. This expression must be the second operand. Thus we have uniquely identified the operator and operands of the prefix expression starting

at the first character of an arbitrary string, if such an expression exists. Since there is at most one valid prefix string starting at the beginning of any string, there is at most one such string starting at any position of an arbitrary string. This is obvious when we consider the substring of the given string starting at the given position.

Notice that this proof has given us a recursive method for finding a prefix expression in a string. We now incorporate this method into the function *find:*

```
int find(char str[])
{
   char temp[MAXLENGTH];
   int length;
   int i, j, m, n;

   if ((length = strlen(str)) == 0)
      return (0);
   if (isalpha(str[0]) != 0)
      /* First character is a letter. */
      /*   That letter is the initial  */
      /*          substring.           */
      return (1);
   /* otherwise find the first operand */
   if (strlen(str) < 2)
      return (0);
   substr(str, 1, length-1, temp);
   m = find(temp);
   if (m == 0 || strlen(str) == m)
      /*   no valid prefix operand or  */
      /*       no second operand       */
      return (0);
   substr(str, m+1, length-m-1, temp);
   n = find(temp);
   if (n == 0)
      return (0);
   return (m+n+1);
} /* end find */
```

Make sure that you understand how these routines work by tracing their actions on both valid and invalid prefix expressions. More important, make sure that you understand how they were developed and how logical analysis led to a natural recursive solution that was directly translatable into a C program.

EXERCISES

3.3.1. Suppose that another provision were added to the Towers of Hanoi problem: that one disk may not rest on another disk that is more than one size larger (for example, disk 1 may only rest on disk 2 or on the ground, disk 2 may only rest on disk 3 or on the ground, and so on). Why does the solution in the text fail to work? What is faulty about the logic that led to it under the new rules?

3.3.2. Prove that the number of moves performed by *towers* in moving n disks equals $2^n - 1$. Can you find a method of solving the Towers of Hanoi problem in fewer moves? Either find such a method for some n or prove that none exists.

3.3.3. Define a postfix and prefix expression to include the possibility of unary operators. Write a program to convert a prefix expression possibly containing the unary negation operator (represented by the symbol '@') to postfix.

3.3.4. Rewrite the function *find* in the text so that it is nonrecursive and computes the length of a prefix string by counting the number of operators and single-letter operands.

3.3.5. Write a recursive function that accepts a prefix expression consisting of binary operators and single-digit integer operands and returns the value of the expression.

3.3.6. Consider the following procedure for converting a prefix expression to postfix. The routine would be called by *conv(prefix,postfix)*.

```
void conv(char prefix[], char postfix[])
{
   char first[2];
   char t1[MAXLENGTH], t2[MAXLENGTH];

   first[0] = prefix[0];
   first[1] = '\0';
   substr(prefix, 1, strlen(prefix) - 1, prefix);
   if (first[0] == '+' || first[0] == '*' || first[0] == '-' ||
                                                first[0] == '/') {
      conv(prefix, t1);
      conv(prefix, t2);
      strcat(t1, t2);
      strcat(t1, first);
      substr(t1, 0, strlen(t1), postfix);
      return;
   } /* end if */
   postfix[0] = first[0];
   postfix[1] = '\0';
} /* end conv */
```

Explain how the procedure works. Is it better or worse than the method of the text? What happens if the routine is called with an invalid prefix string as input? Can you incorporate a check for such an invalid string within *convert*? Can you design such a check for the calling program after *convert* has returned? What is the value of n after *convert* returns?

3.3.7. Develop a recursive method (and program it) to compute the number of different ways in which an integer k can be written as a sum, each of whose operands is less than n.

3.3.8. Consider an array a containing positive and negative integers. Define *contigsum*(i,j) as the sum of the contiguous elements $a[i]$ through $a[j]$ for all array indexes $i<=j$. Develop a recursive procedure that determines i and j such that *contigsum*(i,j) is maximized. The recursion should consider the two halves of the array a.

3.3.9. Write a recursive C program to find the kth smallest element of an array a of numbers by choosing any element $a[i]$ of a and partitioning a into those elements smaller than, equal to, and greater than $a[i]$.

3.3.10. The eight-queens problem is to place eight queens on a chessboard so that no queen is attacking any other queen. The following is a recursive program to solve the problem. *board* is an eight by eight array that represents a chessboard. *board*[*i*][*j*] == *TRUE* if there is a queen at position [*i*][*j*], and *FALSE* otherwise. *good*() is a function that returns *TRUE* if no two queens on the chessboard are attacking each other and *FALSE* otherwise. At the end of the program, the routine *drawboard*() displays a solution to the problem.

```
#define TRUE 1
#define FALSE 0

int try(int);
void drawboard(void);

static short int board [8][8];

void main()
{
   int i, j;

   for(i=0; i<8; i++)
      for(j=0; j<8; j++)
         board[i][j] = FALSE;
   if (try(0) == TRUE)
      drawboard();
}/* end main */

int try(int n)
{
   int i;

   for(i=0; i<8; i++) {
      board[n][i] = TRUE;
      if (n == 7 && good() == TRUE)
         return(TRUE);
      if (n < 7 && good() == TRUE && try(n+1) == TRUE)
         return(TRUE);
      board[n][i] = FALSE;
   } /* end for */
   return(FALSE);
} /* end try */
```

The recursive function *try* returns *TRUE* if it is possible, given the *board* at the time that it is called, to add queens in rows *n* through 7 to achieve a solution. *try* returns *FALSE* if there is no solution that has queens at the positions in *board* that already contain *TRUE*. If *TRUE* is returned, the function also adds queens in rows *n* through 7 to produce a solution. Write the foregoing functions *good* and *drawboard*, and verify that the program produces a solution. (The idea behind the solution is as follows: *board* represents the global situation during an attempt to find a solution. The next step toward finding a solution is chosen arbitrarily (place a queen in the next untried position in row

n). Then recursively test whether it is possible to produce a solution that includes that step. If it is, return. If it is not, backtrack from the attempted next step—*board*[n][i] = *FALSE*—and try another possibility. (This method is called ***backtracking***.)

3.3.11. A 10 × 10 array *maze* of 0s and 1s represents a maze in which a traveler must find a path from *maze*[0][0] to *maze*[9][9]. The traveler may move from a square into any adjacent square in the same row or column, but may not skip over any squares or move diagonally. In addition, the traveler may not move into any square that contains a 1. *maze*[0][0] and *maze*[9][9] contain 0s. Write a routine which accepts such a *maze* and either prints a message that no path through the maze exists or which prints a list of positions representing a path from [0][0] to [9][9].

3.4 SIMULATING RECURSION

In this section we examine more closely some of the mechanisms used to implement recursion so that we can simulate these mechanisms using nonrecursive techniques. This activity is important for several reasons. First of all, many commonly used programming languages (such as FORTRAN, COBOL, and many machine languages) do not allow recursive programs. Problems such as the Towers of Hanoi and prefix-to-postfix conversion, whose solutions can be derived and stated quite simply using recursive techniques, can be programmed in these languages by simulating the recursive solution using more elementary operations. If we know that the recursive solution is correct (and it is often fairly easy to prove such a solution correct) and we have established techniques for converting a recursive solution to a nonrecursive one, we can create a correct solution in a nonrecursive language. It is not uncommon for a programmer to be able to state a recursive solution to a problem. The ability to generate a nonrecursive solution from a recursive algorithm is indispensable when using a compiler that does not support recursion.

Another reason for examining the implementation of recursion is that it will allow us to understand the implications of recursion and some of its hidden pitfalls. Although these pitfalls do not exist in mathematical definitions that employ recursion, they seem to be an inevitable accompaniment of an implementation in a real language on a real machine.

Finally, even in a language such as C that does support recursion, a recursive solution to a problem is often more expensive than a nonrecursive solution, both in terms of time and space. Frequently, this expense is a small price to pay for the logical simplicity and self-documentation of the recursive solution. However, in a production program (such as a compiler, for example) that may be run thousands of times, the recurrent expense is a heavy burden on the system's limited resources. Thus, a program may be designed to incorporate a recursive solution in order to reduce the expense of design and certification, and then carefully converted to a nonrecursive version to be put into actual day-to-day use. As we shall see, in performing such a conversion it is often possible to identify parts of the implementation of recursion that are superfluous in a particular application and thereby significantly reduce the amount of work that the program must perform.

Before examining the actions of a recursive routine, let us take a step back and examine the action of a nonrecursive routine. We will then be able to see what mech-

anisms must be added to support recursion. Before proceeding we adopt the following convention. Suppose that we have the statement

```
rout(x);
```

where *rout* is defined as a function by the header

```
rout(a)
```

x is referred to as an ***argument*** (of the calling function), and *a* is referred to as a ***parameter*** (of the called function).

What happens when a function is called? The action of calling a function may be divided into three parts:

1. Passing arguments
2. Allocating and initializing local variables
3. Transferring control to the function

Let us examine each of these three steps in turn.

1. ***Passing arguments.*** For a parameter in C, a copy of the argument is made locally within the function, and any changes to the parameter are made to that local copy. The effect of this scheme is that the original input argument cannot be altered. In this method, storage for the argument is allocated within the data area of the function.

2. ***Allocating and initializing local variables.*** After arguments have been passed, the local variables of the function are allocated. These local variables include all those declared directly in the function and any temporaries that must be created during the course of execution. For example, in evaluating the expression

```
x + y + z
```

a storage location must be set aside to hold the value of $x + y$ so that z can be added to it. Another storage location must be set aside to hold the value of the entire expression after it has been evaluated. Such locations are called ***temporaries***, since they are needed only temporarily during the course of execution. Similarly, in a statement such as

```
x = fact(n)
```

a temporary must be set aside to hold the value of *fact*(n) before that value can be assigned to x.

3. *Transferring control to the function.* At this point control may still not be passed to the function because provision has not yet been made for saving the ***return address***. If a function is given control, it must eventually restore control to the calling routine by means of a branch. However, it cannot execute that branch unless it knows

the location to which it must return. Since this location is within the calling routine and not within the function, the only way that the function can know this address is to have it passed as an argument. This is exactly what happens. Aside from the explicit arguments specified by the programmer, there is also a set of implicit arguments that contain information necessary for the function to execute and return correctly. Chief among these implicit arguments is the return address. The function stores this address within its own data area. When it is ready to return control to the calling program, the function retrieves the return address and branches to that location.

Once the arguments and the return address have been passed, control may be transferred to the function, since everything required has been done to ensure that the function can operate on the appropriate data and then return to the calling routine safely.

Return from a Function

When a function returns, three actions are performed. First, the return address is retrieved and stored in a safe location. Second, the function's data area is freed. This data area contains all local variables (including local copies of arguments), temporaries, and the return address. Finally, a branch is taken to the return address, which had been previously saved. This restores control to the calling routine at the point immediately following the instruction that initiated the call. In addition, if the function returns a value, that value is placed in a secure location from which the calling program may retrieve it. Usually this location is a hardware register that is set aside for this purpose.

Suppose that a main procedure has called a function *b* that has called *c* that has, in turn, called *d*. This is illustrated in Figure 3.4.1a, where we indicate that control currently resides somewhere within *d*. Within each function, there is a location set aside

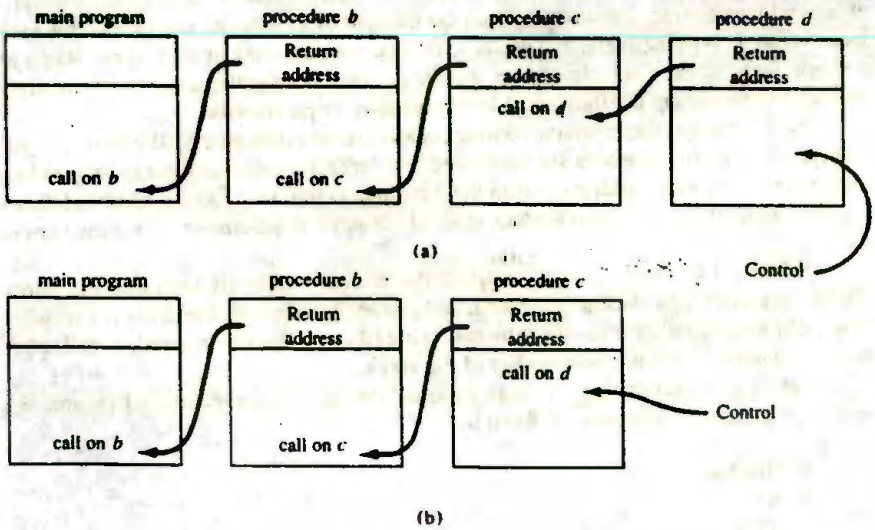


Figure 3.4.1 Series of procedures calling one another.

for the return address. Thus the return address area of *d* contains the address of the instruction in *c* immediately following the call to *d*. Figure 3.4.1b shows the situation immediately following *d*'s return to *c*. The return address within *d* has been retrieved and control has been transferred to that address.

You may have noticed that the string of return addresses forms a stack; that is, the most recent return address to be added to the chain is the first to be removed. At any point, we can only access the return address from within the function that is currently executing, which represents the top of the stack. When the stack is popped (that is, when the function returns), a new top is revealed within the calling routine. Calling a function has the effect of pushing an element onto the stack, and returning pops the stack.

Implementing Recursive Functions

What must be added to this description in the case of a recursive function? The answer is, surprisingly little. Each time a recursive function calls itself, an entirely new data area for that particular call must be allocated. As before, this data area contains all parameters, local variables, temporaries, and a return address. The point to remember is that in recursion a data area is associated not with a function alone but with a particular call to that function. Each call causes a new data area to be allocated, and each reference to an item in the function's data area is to the data area of the most recent call. Similarly, each return causes the current data area to be freed, and the data area allocated immediately prior to the current area becomes current. This behavior, of course, suggests the use of a stack.

In Section 3.1.2, where we described the action of the recursive factorial function, we used a set of stacks to represent the successive allocations of each of the local variables and parameters. These stacks may be thought of as separate stacks, one for each local variable. Alternatively, and closer to reality, we may think of all of these stacks as a single large stack. Each element of this large stack is an entire data area containing subparts representing the individual local variables or parameters.

Each time that the recursive routine is called, a new data area is allocated. The parameters within this data area are initialized to refer to the values of their corresponding arguments. The return address within the data area is initialized to the address following the call instruction. Any reference to local variables or parameters is via the current data area.

When the recursive routine returns, the returned value (if any) and the return address are saved, the data area is freed, and a branch to the return address is executed. The calling function retrieves the returned value (if any), resumes execution, and refers to its own data area that is now on top of the stack.

Let us now examine how we can simulate the actions of a recursive function. We will need a stack of data areas defined by

```
#define MAXSTACK 50;
struct stack {
   int top;
   struct dataarea item[MAXSTACK];
};
```

The *dataarea* is itself a structure containing the various items that exist in a data area and must be defined to contain the fields required for the particular function being simulated.

Simulation of Factorial

Let us look at a specific example: the factorial function. We present the code for that function, including temporary variables explicitly and omitting the test for negative input, as follows:

```
int fact(int n)
{
   int x, y;

   if (n == 0)
      return(1);
   x = n-1;
   y = fact(x);
   return(n * y);
} /* end fact */
```

How are we to define the data area for this function? It must contain the parameter n and the local variables x and y. As we shall see, no temporaries are needed. The data area must also contain a return address. In this case, there are two possible points to which we might want to return: the assignment of *fact*(x) to y, and the main program that called *fact*. Suppose that we had two labels and that we let the label *label*2 be the label of a section of code,

```
label2: y = result;
```

within the simulating program. Let the label *label*1 be the label of a statement

```
label1: return(result);
```

This reflects a convention that the variable *result* contains the value to be returned by an invocation of the *fact* function. The return address will be stored as an integer i (equal to either 1 or 2). To effect a return from a recursive call the statement

```
switch(i) {
   case 1: goto label1;
   case 2: goto label2;
} /* end case */
```

is executed. Thus, if $i == 1$, a return is executed to the main program that called *fact*, and if $i == 2$, a return is simulated to the assignment of the returned value to the variable y in the previous execution of *fact*.

The data area stack for this example can be defined as follows:

```
#define MAXSTACK 50
struct dataarea {
   int param;
   int x;
   long int y;
   short int retaddr;
};
struct stack {
   int top;
   struct dataarea item[MAXSTACK];
};
```

The field in the data area that contains the simulated parameter is called *param*, rather than *n*, to avoid confusion with the parameter *n* passed to the simulating function. We also declare a current data area to hold the values of the variables in the simulated "current" call on the recursive function. The declaration is:

```
struct dataarea currarea;
```

In addition, we declare a single variable *result* by

```
long int result;
```

This variable is used to communicate the returned value of *fact* from one recursive call of *fact* to its caller, and from *fact* to the outside calling function. Since the elements on the stack of data areas are structures and, as we mentioned earlier, it is more efficient to pass structures by reference, we do not use the function *pop* to pop a data area from *stack*. Instead, we write a function *popsub* defined by

```
void popsub(struct stack *ps, struct dataarea *parea)
```

The call *popsub*(*&s*, *&area*) pops the stacks and sets *area* to the popped element. We leave the details as an exercise.

A return from *fact* is simulated by the code

```
result = value to be returned;
i = currarea.retaddr;
popsub(&s, &currarea);
switch(i) {
   case 1: goto label1;
   case 2: goto label2;
} /* end switch */
```

A recursive call on *fact* is simulated by pushing the current data area on the stack, reinitializing the variables *currarea.param* and *currarea.retaddr* to the parameter and return address of this call, respectively, and then transferring control to the start of the simulated routine. Recall that *currarea.x* holds the value of $n - 1$ that is to be the new parameter. Recall also that on a recursive call we wish to eventually return to label 2. The code to accomplish this is

```
push(&s, &currarea);
currarea.param = currarea.x;
currarea.retaddr = 2;
goto start;  /*    start is the label of the    */
             /* start of the simulated routine. */
```

Of course, the *popsub* and *push* routines must be written so that they pop and push entire structures of type *dataarea* rather than simple variables. Another imposition of the array implementation of stacks is that the variable *currarea.y* must be initialized to some value or an error will result in the *push* routine upon assignment of *currarea.y* to the corresponding field of the top data area when the program starts.

When the simulation first begins the current area must be initialized so that *currarea.param* equals *n* and *currarea.retaddr* equals 1 (indicating a return to the calling routine). A dummy data area must be pushed onto the stack so that when *popsub* is executed in returning to the main routine, an underflow does not occur. This dummy data area must also be initialized so as not to cause an error in the *push* routine (see the last sentence of the preceding paragraph). Thus, the simulated version of the recursive *fact* routine is as follows:

```
struct dataarea {
   int param;
   int x;
   long int y;
   short int retaddr;
};
struct stack {
   int top;
   struct dataarea item[MAXSTACK];
};

int simfact(int n)
{
   struct dataarea currarea;
   struct stack s;
   short int i;
   long int result;
```

```
            s.top = -1;
            /*          initialize a dummy data area          */
            currarea.param    = 0;
            currarea.x        = 0;
            currarea.y        = 0;
            currarea.retaddr  = 0;
            /*    push the dummy data area onto the stack     */
            push (&s, &currarea);
            /*   set the parameter and the return address of  */
            /*   the current data area to their proper values. */
            currarea.param    = n;
            currarea.retaddr  = 1;
start:      /*     this is the beginning of the simulated     */
            /*                factorial routine.              */
            if (currarea.param == 0) {
            /*           simulation of return(1);             */
               result = 1;
               i = currarea.retaddr;
               popsub(&s, &currarea);
               switch(i) {
                  case 1: goto label1;
                  case 2: goto label2;
               } /* end switch */
            } /* end if */
            currarea.x = currarea.param - 1;
            /*      simulation of recursive call to fact     */
            push(&s, &currarea);
            currarea.param = currarea.x;
            currarea.retaddr = 2;
            goto start;
label2:     /*      This is the point to which we return     */
            /*    from the recursive call. Set currarea.y    */
            /*              to the returned value.           */
            currarea.y = result;
            /*          simulation of return(n * y)          */
            result = currarea.param * currarea.y;
            i = currarea.retaddr;
            popsub(&s, &currarea);
            switch(i) {
               case 1: goto label1;
               case 2: goto label2;
            } /* end switch */
label1:     /*  At this point we return to the main routine.  */
            return(result);
         } /* end simfact */
```

Trace through the execution of this program for $n = 5$ and be sure that you understand what the program does and how it does it.

Notice that no space was reserved in the data area for temporaries, since they need not be saved for later use. The temporary location that holds the value of $n * y$

in the original recursive routine is simulated by the temporary for *currarea.param* * *currarea.y* in the simulating routine. This is not the case in general. For example, if a recursive function *funct* contained a statement such as

```
x = a * funct(b)  +  c * funct(d);
```

the temporary for $a * funct(b)$ must be saved during the recursive call on $funct(d)$. However, in the example of the factorial function, it is not required to stack the temporary.

Improving the Simulated Routine

The foregoing discussion leads naturally to the question of whether all the local variables really need to be stacked at all. A variable must be saved on the stack only if its value at the point of initiation of a recursive call must be reused after return from that call. Let us examine whether the variables n, x, and y meet this requirement. Clearly n does have to be stacked. In the statement

```
y = n * fact(x);
```

the old value of n must be used in the multiplication after return from the recursive call on *fact*. However, this is not the case for x and y. In fact, the value of y is not even defined at the point of the recursive call, so clearly it need not be stacked. Similarly, although x is defined at the point of call, it is never used again after returning, so why bother saving it?

This point can be illustrated even more sharply by the following realization. If x and y were not declared within the recursive function *fact*, but rather were declared as global variables, the routine would work just as well. Thus, the automatic stacking and unstacking action performed by recursion for the local variables x and y is unnecessary.

Another interesting question to consider is whether the return address is really needed on the stack. Since there is only one textual recursive call to *fact*, there is only one return address within *fact*. The other return address is to the main routine that originally called *fact*. But suppose a dummy data area had not been stacked upon initialization of the simulation. Then a data area is placed on the stack only in simulating a recursive call. When the stack is popped in returning from a recursive call, that area is removed from the stack. However, when an attempt is made to pop the stack in simulating a return to the main procedure, an underflow will occur. We can test for this underflow by using *popandtest* rather than *popsub*, and when it does occur we can return directly to the outside calling routine rather than through a local label. This means that one of the return addresses can be eliminated. Since this leaves only a single possible return address, it need not be placed on the stack.

Thus the data area has been reduced to contain the parameter alone, and the stack may be declared by

```
#define MAXSTACK 50
struct stack {
   int top;
   int param[MAXSTACK];
};
```

The current data area is reduced to a single variable declared by

```
int currparam;
```

The program is now quite compact and comprehensible.

```
int simfact(int n)
{
   struct stack s;
   short int und;
   long int result, y;
   int currparam, x;

   s.top = -1;
   currparam = n;
start:  /*  This is the beginning of the simulated */
        /*        factorial routine.               */
   if (currparam == 0) {
      /*       simulation of return(1)        */
      result = 1;
      popandtest(&s, &currparam, &und);
      switch(und) {
         case FALSE:  goto label2;
         case TRUE:   goto label1;
      } /* end switch */
   } /* end if */
   /* currparam != 0 */
   x = currparam - 1;
   /* simulation of recursive call to fact    */
   push(&s, currparam);
   currparam = x;
   goto start;
label2: /*  This is the point to which we return   */
        /*     from the recursive call. Set        */
        /*       y to the returned value.          */
   y = result;
   /*       simulation of return (n * y);     */
   result = currparam * y;
   popandtest(&s, &currparam, &und);
   switch(und) {
      case TRUE:  goto label1;
      case FALSE: goto label2;
   } /* end switch */
label1: /*  At this point we return to the main    */
        /*                routine.                 */
   return(result);
} /* end simfact */
```

Eliminating *gotos*

Although the preceding program is certainly simpler than the previous one, it is still far from ideal. If you were to look at the program without having seen its derivation, it is probably doubtful that you could identify it as computing the factorial function. The statements

```
goto start;
```

and

```
goto label2;
```

are particularly irritating, since they interrupt the flow of thought at a time that one might otherwise come to an understanding of what is happening. Let us see if we can transform this program into a still more readable version.

Several transformations are immediately apparent. First of all, the statements

```
popandtest(&s, &currparam, &und);
switch(und) {
   case FALSE: goto label2;
   case TRUE:  goto label1;
} /* end switch */
```

are repeated twice for the two cases *currparam* == 0 and *currparam* != 0. The two sections can easily be combined into one.

A further observation is that the two variables *x* and *currparam* are assigned values from each other and are never in use simultaneously; therefore they may be combined and referred to as one variable *x*. The same is true of the variables *result* and *y*, which may be combined and referred to as the single variable *y*.

Performing these transformations leads to the following version of *simfact:*

```
struct stack {
   int top;
   int param[MAXSTACK];
};

int simfact(int n)
{
   struct stack s;
   short int und;
   int x;
   long int y;
```

```
        s.top = -1;
        x = n;
start:  /* This is the beginning of the simulated */
        /*          factorial routine.           */
        if (x == 0)
           y = 1;
        else {
           push(&s, x--);
           goto start;
        } /* end else */
label1: popandtest(&s, &x, &und);
        if (und == TRUE)
           return(y);
label2: y *= x;
        goto label1;
     } /* end simfact */
```

We are now beginning to approach a readable program. Note that the program consists of two loops:

1. The loop that consists of the entire *if* statement, labeled *start*. This loop is exited when x equals 0, at which point y is set to 1 and execution proceeds to the label *label*1.
2. The loop that begins at label *label*1 and ends with the statement ***goto*** *label*1. This loop is exited when the stack has been emptied and underflow occurs, at which point a return is executed.

These loops can easily be transformed into explicit ***while*** loops as follows:

```
        /* subtraction loop */
start:  while (x != 0)
           push(&s, x--);
        y = 1;
        popandtest(&s, &x, &und);
label1: while (und == FALSE) {
           y *= x;
           popandtest (&s, &x, &und);
        } /* end while * /
        return(y);
```

Let us examine these two loops more closely. x starts off at the value of the input parameter n and is reduced by 1 each time that the subtraction loop is repeated. Each time x is set to a new value, the old value of x is saved on the stack. This continues until x is 0. Thus, after the first loop has been executed the stack contains, from top to bottom, the integers 1 to n.

The multiplication loop merely removes each of these values from the stack and sets y to the product of the popped value and the old value of y. Since we know what the stack contains at the start of the multiplication loop, why bother popping the stack? We can use those values directly. We can eliminate the stack and the first loop entirely

and replace the multiplication loop with a loop that multiplies y by each of the integers from 1 to n in turn. The resulting program is

```
int simfact(int n)
{
   int x;
   long int y;

   for (y=x=1; x <= n; x++)
      y *= x;
   return(y);
} /* end simfact */
```

But this program is a direct C implementation of the iterative version of the factorial function as presented in Section 3.1. The only change is that x varies from 1 to n rather than from n to 1.

Simulating the Towers of Hanoi

We have shown that successive transformations of a nonrecursive simulation of a recursive routine may lead to a simpler program for solving a problem. Let us now look at a more complex example of recursion, the Towers of Hanoi problem presented in Section 3.3. We will simulate its recursion and attempt to simplify the simulation to produce a nonrecursive solution. We present again the recursive program of Section 3.3:

```
void towers(int n, char frompeg, char topeg, char auxpeg)
{
   /*  If only one disk, make the move and return.  */
   if (n == 1) {
      printf("\n%s%c%s%c", "move disk 1 from peg ", frompeg,
                                                  " to peg ", topeg);
      return;
   } /* end if */
   /* Move top n-1 disks from A to B, using C as */
   /*                  auxiliary                 */
   towers(n-1, frompeg, auxpeg, topeg);
   /*       Move remaining disk from A to C.     */
   printf("\n%s%d%s%c%s%c", "move disk ",-" from peg ",
                                          frompeg, " to peg ", topeg);
   /*   Move n-1 disk from B to C using A as     */
   /*                  auxiliary                 */
   towers(n-1, auxpeg, topeg, frompeg);
} /* end towers */
```

Make sure that you understand the problem and the recursive solution before proceeding. If you do not, reread Section 3.3.

There are four parameters in this function, each of which is subject to change in a recursive call. Therefore the data area must contain elements representing all four.

There are no local variables. There is a single temporary that is needed to hold the value of $n - 1$, but this can be represented by a similar temporary in the simulating program and does not have to be stacked. There are three possible points to which the function returns on various calls: the calling program and the two points following the recursive calls. Therefore four labels are necessary:

```
start:
label1:
label2:
label3:
```

The return address is encoded as an integer (either 1, 2, or 3) within each data area.

Consider the following nonrecursive simulation of *towers*:

```
struct dataarea {
   int nparam;
   char fromparam;
   char toparam;
   char auxparam;
   short int retaddr;
};
struct stack {
   int top;
   struct dataarea item[MAXSTACK];
};

void simtowers(int n, char frompeg, char topeg, char auxpeg)
{
   struct stack s;
   struct dataarea currarea;
   char temp;
   short int i;

   s.top = -1;
   currarea.nparam = 0;
   currarea.fromparam = ' ';
   currarea.toparam = ' ';
   currarea.auxparam = ' ';
   currarea.retaddr = 0;
   /*       Push dummy data area onto stack.       */
   push(&s, &currarea);
   /* Set the parameters and the return addresses */
   /* of the current data to their proper values. */
   currarea.nparam = n;
   currarea.fromparam = frompeg;
   currarea.toparam = topeg;
   currarea.auxparam = auxpeg;
   currarea.retaddr = 1;
```

```
start:  /* This is the start of the simulated routine. */
        if (currarea.nparam == 1) {
           printf("\n%s%c%s%c", "move disk 1 from peg ",
                   currarea.frompeg, " to peg ", currarea.toparam);
           i = currarea.retaddr;
           pop(&s, &currarea);
           switch(i) {
              case 1: goto label1;
              case 2: goto label2;
              case 3: goto label3;
           } /* end switch */
        } /* end if */
        /*      This is the first recursive call.      */
        push(&s, &currarea);
        --currarea.nparam;
        temp = currarea.auxparam;
        currarea.auxparam = currarea.toparam;
        currarea.toparam = temp;
        currarea.retaddr = 2;
        goto start;
label2: /*   We return to this point from the first    */
        /*              recursive call.                */
        printf("\n%s%d%s%c%s%c", "move disk ", currarea.nparam, " from peg ",
                                 currarea.fromparam, " to peg ", currarea.toparam);
        /*     This is the second recursive call.      */
        push(&s, &currarea);
        --currarea.nparam;
        temp = currarea.fromparam;
        currarea.fromparam = currarea.auxparam;
        currarea.auxparam = temp;
        currarea.retaddr = 3;
        goto start;
label3: /*     Return to this point from the second    */
        /*              recursive call.                */
        i = currarea.retaddr;
        pop(&s, &currarea);
        switch(i) {
           case 1: goto label1;
           case 2: goto label2;
           case 3: goto label3;
        } /* end switch */
label1: return;
     } /*  end simtowers */
```

We now simplify the program. First, notice that three labels are used for return addresses: one for each of the two recursive calls and one for the return to the main program. However, the return to the main program can be signaled by an underflow in the stack, exactly as in the second version of *simfact*. This leaves two return labels. If we

could eliminate one more such label it would no longer be necessary to stack the return address, since there would be only one point remaining to which control may be passed if the stack is popped successfully. We focus our attention on the second recursive call and the statement

```
towers(n-1, auxpeg, topeg, frompeg);
```

The actions that occur in simulating this call are as follows:

1. Push the current data area, $a1$, onto the stack.
2. Set the parameters in the new current data area, $a2$, to their respective values, $n - 1$, *auxpeg*, *topeg*, and *frompeg*.
3. Set the return label in the current data area, $a2$, to the address of the statement immediately following the call.
4. Branch to the beginning of the simulated routine.

After the simulated routine has completed, it is ready to return. The following actions occur:

5. Save the return label, *l*, from the current data area, $a2$.
6. Pop the stack and set the current data area to the popped data area, $a1$.
7. Branch to *l*.

But *l* is the label of the end of the block of code, since the second call to *towers* appears as the last statement of the function. Thus, the next step is to pop the stack again and return once more. We never again make use of the information in the current data area $a1$, since it is immediately destroyed by popping the stack as soon as it has been restored. Since there is no reason to use this data area again, there is no reason to save it on the stack in simulating the call. Data need be saved on the stack only if it is to be reused. Therefore the second call to *towers* may be simulated simply by

1. Changing the parameters in the current data area to their respective values
2. Branching to the beginning of the simulated routine

When the simulated routine returns it can return directly to the routine that called the current version. There is no reason to execute a return to the current version, only to return immediately to the previous version. Thus we have eliminated the need for stacking the return address in simulating the external call (since it can be signaled by underflow) and in simulating the second recursive call (since there is no need to save and restore the calling routine's data area at that point). The only remaining return address is the one following the first recursive call.

Since there is only one possible return address left, it is unnecessary to keep it in the data area, to be pushed and popped with the rest of the data. Whenever the stack is popped successfully, there is only one address to which a branch can be executed: the statement following the first call. If an underflow is encountered, the routine returns to the calling routine. Since the new values of the variables in the current data area will be obtained from the old values in the current data area, it is necessary to declare an additional variable, *temp*, so that values can be interchanged.

A revised nonrecursive simulation of *towers* follows:

```
struct dataarea {
   int nparam;
   char fromparam;
   char toparam;
   char auxparam;
};
struct stack {
   int top;
   struct dataarea item[MAXSTACK];
};

void simtowers(int n, char frompeg, char topeg, char auxpeg)
{
   struct stack s;
   struct dataarea currarea;
   short int und;
   char temp;

   s.top = -1;
   currarea.nparam = n;
   currarea.fromparam = frompeg;
   currarea.toparam = topeg;
   currarea.auxparam = auxpeg;
start: /* This is the start of the simulated routine. */
   if (currarea.nparam == 1) {
      printf("\n%s%c%s%c", "move disk 1 from peg ",
            currarea.frompeg, " to peg ", currarea.toparam);
      /*          simulate the return          */
      popandtest(&s, &currarea, &und);
      if (und == TRUE)
         return;
         goto retaddr;
   } /* end if */
   /*     simulate the first recursive call      */
   push(&s, &currarea);
   --currarea.nparam;
   temp = currarea.toparam;
   currarea.toparam = currarea.auxparam;
   currarea.auxparam = temp;
   goto start;
retaddr: /*     return to this point from the first     */
   /*              recursive call                 */
   printf("\n%s%d%s%c%s%c" "move disk ", currarea.nparam,
            " from peg ", currarea.fromparam, " to peg ",
                              currarea.toparam);
   /*     simulation of second recursive call     */
   --currarea.nparam;
```

```
        temp = currarea.fromparam;
        currarea.fromparam = currarea.auxparam;
        currarea.auxparam = temp;
        goto start;
    } /* end simtowers */
```

Examining the structure of the program, we see that it can easily be reorganized into a simpler format. We begin from the label *start*.

```
while (TRUE) {
    while (currarea.nparam != 1) {
        push(&s, &currarea);
        --currarea.nparam;
        temp = currarea.toparam;
        currarea.toparam = currarea.auxparam;
        currarea.auxparam = temp;
    } /* end while */
    printf("\n%s%c%s%c", "move disk 1 from peg ",
            currarea.fromparam, " to peg ", currarea.toparam)
    popandtest(&s, &currarea, &und);
    if (und == TRUE)
        return;
    printf("\n%s%d%s%c%s%c", "move disk ", currarea.nparam,
                    " from ", currarea.fromparam, " to peg ",
                                        currarea.toparam)
    --currarea.nparam;
    temp = currarea.fromparam;
    currarea.fromparam = currarea.auxparam;
    currarea.auxparam = temp;
} /* end while */
```

Trace through the actions of this program and see how it reflects the actions of the original recursive version.

EXERCISES

3.4.1. Write a nonrecursive simulation of the functions *convert* and *find* presented in Section 3.3.

3.4.2. Write a nonrecursive simulation of the recursive binary search procedure, and transform it into an iterative procedure.

3.4.3. Write a nonrecursive simulation of *fib*. Can you transform it into an iterative method?

3.4.4. Write nonrecursive simulations of the recursive routines of Sections 3.2 and 3.3 and the exercises of those sections.

3.4.5. Show that any solution to the Towers of Hanoi problem that uses a minimum number of moves must satisfy the conditions listed below. Use these facts to develop a direct iterative algorithm for Towers of Hanoi. Implement the algorithm as a C program.

1. The first move involves moving the smallest disk.
2. A minimum-move solution consists of alternately moving the smallest disk and a disk that is not the smallest.
3. At any point, there is only one possible move involving a disk that is not the smallest.
4. Define the cyclic direction from *frompeg* to *topeg* to *auxpeg* to *frompeg* as clockwise and the opposite direction (from *frompeg* to *auxpeg* to *topeg* to *frompeg*) as counterclockwise. Assume that a minimum-move solution to move a k-disk tower from *frompeg* to *topeg* always moves the smallest disk in one direction. Show that a minimum-move solution to move a $(k + 1)$-disk tower from *frompeg* to *topeg* would then always move the smallest disk in the other direction. Since the solution for one disk moves the smallest disk clockwise (the single move from *frompeg* to *topeg*), this means that for an odd number of disks the smallest disk always moves clockwise, and for an even number of disks the smallest disk always moves counterclockwise.
5. The solution is completed as soon as all the disks are on a single peg.

3.4.6. Convert the following recursive program scheme into an iterative version that does not use a stack. $f(n)$ is a function that returns *TRUE* or *FALSE* based on the value of n, and $g(n)$ is a function that returns a value of the same type as n (without modifying n).

```
int rec(int n)
{
   if (f(n) == FALSE) {
      /* any group of C statements that */
      /* do not change the value of n   */
      rec(g(n));
   } /* end if */
} /* end rec */
```

Generalize your result to the case in which *rec* returns a value.

3.4.7. Let $f(n)$ be a function and $g(n)$ and $h(n)$ be functions that return a value of the same type as n without modifying n. Let (*stmts*) represent any group of C statements that do not modify the value of n. Show that the recursive program scheme *rec* is equivalent to the iterative scheme *iter*:

```
void rec(int n)
{
   if (f(n) == FALSE) {
      (stmts);
      rec(g(n));
      rec(h(n));
   } /* end if */
} /* end rec */

struct stack {
   int top;
   int nvalues[MAXSTACK];
};
```

```
void iter(int n)
{
   struct stack s;

   s.top = -1;
   push(&s, n);
   while(empty(&s) == FALSE ) {
      n = pop(&s);
      if (f(n) == FALSE) {
         (stmts);
         push(&s, h(n));
         push(&s, g(n));
      } /* end if */
   } /* end while */
} /* end iter */
```

Show that the *if* statements in *iter* can be replaced by the following loop:

```
while (f(n) == FALSE) {
   (stmts)
   push(&s, h(n));
   n = g(n);
} /* end while */
```

3.5 EFFICIENCY OF RECURSION

In general a nonrecursive version of a program will execute more efficiently in terms of time and space than a recursive version. This is because the overhead involved in entering and exiting a block is avoided in the nonrecursive version. As we have seen, it is often possible to identify a good number of local variables and temporaries that do not have to be saved and restored through the use of a stack. In a nonrecursive program this needless stacking activity can be eliminated. However, in a recursive procedure, the compiler is usually unable to identify such variables, and they are therefore stacked and unstacked to ensure that no problems arise.

However, we have also seen that sometimes a recursive solution is the most natural and logical way of solving a problem. It is doubtful whether a programmer could have developed the nonrecursive solution to the Towers of Hanoi problem directly from the problem statement. A similar comment may be made about the problem of converting prefix to postfix, where the recursive solution flows directly from the definitions. A nonrecursive solution involving stacks is more difficult to develop and more prone to error.

Thus there is a conflict between machine efficiency and programmer efficiency. With the cost of programming increasing steadily, and the cost of computation decreasing, we have reached the point where in most cases it is not worth a programmer's time to laboriously construct a nonrecursive solution to a problem that is most naturally solved recursively. Of course an incompetent, overly clever programmer may come up

with a complicated recursive solution to a simple problem that can be solved directly by nonrecursive methods. (An example of this is the factorial function or even the binary search.) However, if a competent programmer identifies a recursive solution as being the simplest and most straightforward method for solving a particular problem, it is probably not worth the time and effort to discover a more efficient method.

However, this is not always the case. If a program is to be run very frequently (often entire computers are dedicated to continually running the same program), so that increased efficiency in execution speed significantly increases throughput, the extra investment in programming time is worthwhile. Even in such cases, it is probably better to create a nonrecursive version by simulating and transforming the recursive solution than by attempting to create a nonrecursive solution from the problem statement.

To do this most efficiently, what is required is to first write the recursive routine and then its simulated version, including all stacks and temporaries. After this has been done, eliminate all stacks and variables that are superfluous. The final version is a refinement of the original program and is certainly more efficient. Clearly, the elimination of each superfluous and redundant operation improves the efficiency of the resulting program. However, every transformation applied to a program is another opening through which an unanticipated error may creep in.

When a stack cannot be eliminated from the nonrecursive version of a program and when the recursive version does not contain any extra parameters or local variables, the recursive version can be as fast or faster than the nonrecursive version under a good compiler. The Towers of Hanoi is an example of such a recursive program. Factorial, whose nonrecursive version does not need a stack, and calculation of Fibonacci numbers, which contains an unnecessary second recursive call (and does not need a stack either), are examples where recursion should be avoided in a practical implementation. We examine another example of efficient recursion (in order tree traversal) in Section 5.2.

Another point to remember is that explicit calls to *pop*, *push*, and *empty*, as well as tests for underflow and overflow, are quite expensive. In fact, they can often outweigh the expense of the overhead of recursion. Thus, to maximize actual run-time efficiency of a nonrecursive translation, these calls should be replaced by in-line code and the overflow/underflow tests eliminated when it is known that we are operating within the array bounds.

The ideas and transformations that we have put forward in presenting the factorial function and in the Towers of Hanoi problem can be applied to more complex problems whose nonrecursive solution is not readily apparent. The extent to which a recursive solution (actual or simulated) can be transformed into a direct solution depends on the particular problem and the ingenuity of the programmer.

EXERCISES

3.5.1. Run the recursive and nonrecursive versions of the factorial function of Sections 3.2 and 3.4, and examine how much space and time each requires as n becomes larger.

3.5.2. Do the same as in Exercise 3.5.1 for the Towers of Hanoi problem.