28 In both ANSI C and traditional C, a backslash at the end of a line in a string constant has the effect of continuing it to the next line. Here is an example of this:

```
"by using a backslash at the end of the line \
a string can be extended from one line to the next"
```

Write a program that uses this construct. Many screens have 80 characters per line. What happens if you try to print a string with more than 80 characters?

29 In ANSI C, a backslash at the end of *any* line is supposed to have the effect of continuing it to the next line. This can be expected to work in string constants and macro definitions on any C compiler, either ANSI or traditional. (See the previous exercise.) However, not all ANSI C compilers support this in a more general way. After all, except in macro definitions, this construct gets little use. Does your C compiler support this in a general way? Try the following:

```
#inc\
lude <stdio.h>

int mai\
n(void)
{
   printf("Will this work?\n");
   ret\
urn 0;
}
```

30 When you invoke the compiler, the system first invokes the preprocessor. In this exercise we want to deliberately make a preprocessing error, just to see what happens. Try the following program:

```
#incl <stdixx.h>        /* two errors on this line */

int main(void)
{
   printf("Try me.\n");
   return 0;
}
```

What happens if you change `#incl` to `#include`?

Chapter 3

The Fundamental Data Types

We begin this chapter with a brief look at declarations, expressions, and assignment. Then we give a detailed explanation for each of the fundamental data types, paying particular attention to how C treats characters as small integers. In expressions with operands of different types, certain implicit conversions occur. We explain the rules for conversion and examine the cast operator, which forces explicit conversion.

3.1 Declarations, Expressions, and Assignment

Variables and constants are the objects that a program manipulates. In C, all variables must be declared before they can be used. The beginning of a program might look like this:

```
#include <stdio.h>

int main(void)
{
   int     a, b, c;                    /* declaration */
   float   x, y = 3.3, z = -7.7;       /* declaration with
                                          initializations */

   printf("Input two integers: ");     /* function call */
   scanf("%d%d", &b, &c);              /* function call */
   a = b + c;                          /* assignment */
   x = y + z;                          /* assignment */
   .....
```

Declarations associate a type with each variable that is declared, and this tells the compiler to set aside an appropriate amount of space in memory to hold values associated with variables. This also enables the compiler to instruct the machine to perform specified operations correctly. In the expression `b + c`, the operator + is being applied to two variables of type `int`, which at the machine level is a different operation than + applied to variables of type `float`, as occurs in the expression `y + z`. Of course, the programmer need not be concerned that the two + operations are mechanically different, but the C compiler has to recognize the difference and give the appropriate machine instructions.

The braces { and } surround a *block*, which consists of declarations and statements. The declarations, if any, must occur before the statements. The body of a function definition is a block, but as we shall see in Section 5.10, "Scope Rules," on page 213, there are other uses for blocks.

Expressions are meaningful combinations of constants, variables, operators, and function calls. A constant, variable, or function call by itself can also be considered an expression. Some examples of expressions are

```
a + b
sqrt(7.333)
5.0 * x - tan(9.0 / x)
```

Most expressions have a value. For example, the expression `a + b` has an obvious value, depending on the values of the variables `a` and `b`. If `a` has value 1 and `b` has value 2, then `a + b` has value 3.

The equal sign = is the basic assignment operator in C. An example of an assignment *expression* is

```
i = 7
```

The variable `i` is assigned the value 7, and the expression as a whole takes that value as well. When followed by a semicolon, an expression becomes a statement, or more explicitly, an expression statement. Some examples of statements are

```
i = 7;
printf("The plot thickens!\n");
```

The following two statements are perfectly legal, but they do no useful work. Some compilers will issue warnings about such statements; others will not.

```
3.777;
a + b;
```


Let us consider a statement that consists of a simple assignment expression followed by a semicolon. It will have the following form:

variable = *expr* ;

First, the value of the expression on the right side of the equal sign is computed. Then that value is assigned to the variable on the left side of the equal sign, and this becomes the value of the assignment expression as a whole. (Statements do not have a value.) Note that here the value of the assignment expression as a whole is not used. That is perfectly all right. The programmer is not required to use the value produced by an expression.

Even though assignment expressions sometimes resemble mathematical equations, the two notions are distinct and should not be confused. The mathematical equation

$$x + 2 = 0$$

does not become an assignment expression by typing

```
x + 2 = 0        /* wrong */
```

Here, the left side of the equal sign is an expression, not a variable, and this expression may not be assigned a value. Now consider the statement

```
x = x + 1;
```

The current value of `x` is assigned the old value of `x` plus 1. If the old value of `x` is 2, then the value of `x` after execution of the statement will be 3. Observe that as a mathematical equation

$$x = x + 1$$

is meaningless; after subtracting x from both sides of the equation, we obtain

$$0 = 1$$

Caution: Although they look alike, the assignment operator in C and the equal sign in mathematics are not comparable.

3.2 The Fundamental Data Types

C provides several fundamental data types, many of which we have already seen. We need to discuss limitations on what can be stored in each type.

Fundamental data types: long form		
`char`	`signed char`	`unsigned char`
`signed short int`	`signed int`	`signed long int`
`unsigned short int`	`unsigned int`	`unsigned long int`
`float`	`double`	`long double`

These are all keywords. They may not be used as names of variables. Of course, `char` stands for "character" and `int` stands for "integer," but only `char` and `int` can be used as keywords. Other data types such as arrays, pointers, and structures are derived from the fundamental types. They are presented in later chapters.

Usually, the keyword `signed` is not used. For example, `signed int` is equivalent to `int`, and because shorter names are easier to type, `int` is typically used. The type `char`, however, is special in this regard. (See Section 3.3, "Characters and the Data Type `char`," on page 111.) Also, the keywords `short int`, `long int`, and `unsigned int` may be, and usually are, shortened to just `short`, `long`, and `unsigned`, respectively. The keyword `signed` by itself is equivalent to `int`, but it is seldom used in this context. With all these conventions, we obtain a new list.

Fundamental data types		
`char`	`signed char`	`unsigned char`
`short`	`int`	`long`
`unsigned short`	`unsigned`	`unsigned long`
`float`	`double`	`long double`

Let us assume that the category *type* is defined to be any one of the fundamental types given in the preceding table. Using this category, we can provide the syntax of a simple declaration:

declaration ::= *type* *identifier* { , *identifier* }$_{0+}$;

The fundamental types can be grouped according to functionality. The integral types are those types that can be used to hold integer values; the floating types are those that can be used to hold real values. They are all arithmetic types.

Fundamental types grouped by functionality			
Integral types	`char`	`signed char`	`unsigned char`
	`short`	`int`	`long`
	`unsigned short`	`unsigned`	`unsigned long`
Floating types	`float`	`double`	`long double`
Arithmetic types	*Integral types + Floating types*		

These collective names are a convenience. In Chapter 6, for example, when we discuss arrays, we will explain that only integral expressions are allowed as subscripts, meaning only expressions involving integral types are allowed.

3.3 Characters and the Data Type char

In C, variables of any integral type can be used to represent characters. In particular, both `char` and `int` variables are used for this purpose. In some situations, an `int` may be required for technical reasons. (See Section 3.9, "The Use of `getchar()` and `putchar()`," on page 124.) Constants such as `'a'` and `'+'` that we think of as characters are of type `int`, not of type `char`. There are no constants of type `char` in C. This is one of the few places where C++ differs from C. In C++, character constants are of type `char`. (See exercise 14, on page 142.)

In addition to representing characters, a variable of type `char` can be used to hold small integer values. Each `char` is stored in memory in 1 byte. Other than being large enough to hold all the characters in the character set, the size of a byte is not specified in C. However, on most machines a byte is composed of 8 bits and is capable, therefore, of storing 2^8, or 256, distinct values. Only a subset of these values represents actual printing characters. These include the lower- and uppercase letters, digits, punctuation, and special characters such as % and +. The character set also includes the white space characters blank, tab, and newline.

Most machines use either ASCII or EBCDIC character codes. In the discussion that follows, we will be using the ASCII code. A table for this code appears in Appendix D,

"ASCII Character Codes." For any other code, the numbers will be different, but the ideas are analogous. The following table illustrates the correspondence between some character and integer values on an ASCII machine:

Some character constants and their corresponding integer values						
Character constants	`'a'`	`'b'`	`'c'`	...	`'z'`	
Corresponding values	97	98	99	...	112	
Character constants	`'A'`	`'B'`	`'C'`	...	`'Z'`	
Corresponding values	65	66	67	...	90	
Character constants	`'0'`	`'1'`	`'2'`	...	`'9'`	
Corresponding values	48	49	50	...	57	
Character constants	`'&'`	`'*'`	`'+'`			
Corresponding values	38	42	43			

Observe that there is no particular relationship between the value of the character constant representing a digit and the digit's intrinsic integer value. That is, the value of `'2'` is *not* 2. The property that the values for `'a'`, `'b'`, `'c'`, and so on occur in order is important. It makes convenient the sorting of characters, words, and lines into lexicographical order. Character arrays are needed for this kind of work. (See Section 6.10, "Strings," on page 270.)

Some nonprinting and hard-to-print characters require an escape sequence. The horizontal tab character, for example, is written as `\t` in character constants and in strings. Even though it is being described by the two characters \ and `t`, it represents a single character. The backslash character \ is called the *escape character* and is used to escape the usual meaning of the character that follows it. The following table contains some nonprinting and hard-to-print characters:

Special Characters		
Name of character	Written in C	Integer value
alert	`\a`	7
backslash	`\\`	92
backspace	`\b`	8
carriage return	`\r`	13
double quote	`\"`	34
formfeed	`\f`	12
horizontal tab	`\t`	9
newline	`\n`	10
null character	`\0`	0
single quote	`\'`	39
vertical tab	`\v`	11
question mark	`\?`	63

The alert character `\a` is special; it causes the bell to ring. To hear the bell, try executing a program that contains the line

```
printf("%c", '\a');        or        putchar('\a');
```

The double-quote character " has to be escaped if it is used as a character in a string. An example is

```
printf("\"abc\"");        /* "abc" is printed */
```

Similarly, the single-quote character ' has to be escaped if it is used in a constant character construct.

```
printf("%cabc%c", '\'', '\'');        /* 'abc' is printed */
```

Inside single quotes we can use either `\"` or ".

```
printf("%cabc%c", '\"', '"');        /* 'abc' is printed */
```

Inside double quotes we can use either `\'` or '.

```
printf("\'abc'");        /* 'abc' is printed */
```

In ANSI C, the effect of escaping an ordinary character is undefined. Some compilers will complain about this; others will not.

Another way to write a character constant is by means of a one-, two-, or three-octal-digit escape sequence, as in `'\007'`. This is the alert character, or the audible bell. It can be written also as `'\07'` or `'\7'`, but it cannot be written as `'7'`. ANSI C also provides hexadecimal escape sequences. An example is `'\x1a'`, which is control-z.

Next, we want to understand how characters are treated as small integers, and, conversely, how small integers are treated as characters. Consider the declaration

```
char   c = 'a';
```

The variable c can be printed either as a character or as an integer.

```
printf("%c", c);                    /*  a is printed */
printf("%d", c);                    /* 97 is printed */
```

Because c has an integer value, we may use it in arithmetic expressions.

```
printf("%c%c%c", c, c + 1, c + 2);   /* abc is printed */
```

Actually, in this regard there is nothing special about the type char. Any integral expression can be printed either in the format of a character or an integer.

```
char   c;
int    i;

for (i = 'a'; i <= 'z'; ++i)
   printf("%c", i);                  /* abc ... z is printed */
for (c = 65; c <= 90; ++c)
   printf("%c", c);                  /* ABC ... Z is printed */
for (c = '0'; c <= '9'; ++c)
   printf("%d ", c);                 /* 48 49 ... 57 is printed */
```

Next, we want to look at how a char is stored in memory at the bit level, and how strings of bits are interpreted as binary numbers. Before we describe this, recall how strings of decimal digits are interpreted as decimal numbers. Consider, for example, the decimal number 20753. Its value is given by

$$2\times10^4 + 0\times10^3 + 7\times10^2 + 5\times10^1 + 3\times10^0$$

Observe that each of the exponents corresponds to the count of a digit in the number 20753, when we count from zero starting on the right. The digit 3 with count 0 is the least significant digit, and the digit 2 with count 4 is the most significant digit. The general form of a decimal number is given by

$$d_n d_{n-1} \ldots d_2 d_1 d_0$$

where each d_i is a decimal digit. Note that the digits are numbered from least significant to most significant, counting from zero. The value of the number is given by

$$d_n\times10^n + d_{n-1}\times10^{n-1} + \ldots + d_2\times10^2 + d_1\times10^1 + d_0\times10^0$$

In a similar fashion, strings of bits, which are comprised of the binary digits 0 and 1, can be interpreted as binary numbers. The general form of a binary number, also called a base 2 number, is given by

$$b_n b_{n-1} \ldots b_2 b_1 b_0$$

where each b_i is a bit, or binary digit. The value of the number is given by

$$b_n\times2^n + b_{n-1}\times2^{n-1} + \ldots + b_2\times2^2 + b_1\times2^1 + b_0\times2^0$$

Now we are ready to look at how a char is stored in memory at the bit level. Consider the declaration

```
char   c = 'a';
```

We can think of c stored in memory in 1 byte as

```
01100001
```

where the 0s and 1s comprise the 8 bits in the byte. By convention, 0 represents a bit turned off and 1 represents a bit turned on. This string of binary digits can be considered a binary number, and its value is given by

$$1\times2^6 + 1\times2^5 + 0\times2^4 + 0\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0$$

which is 64 + 32 + 1, or 97, in decimal notation.

ANSI C provides the three types char, signed char, and unsigned char. The type char is equivalent to either signed char or unsigned char, depending on the compiler. Each of the three char types is stored in 1 byte, which can hold 256 distinct values. For a signed char the values go from -128 to 127. For an unsigned char the values go from 0 to 255. To determine the values that are appropriate for a plain char on your system, see exercise 10, on page 141.

3.4 The Data Type int

The data type int is the principal working type of the C language. This type, along with the other integral types such as char, short, and long, is designed for working with the integer values that are representable on a machine.

In mathematics, the natural numbers are 0, 1, 2, 3, ... , and these numbers, along with their negatives, comprise the integers. On a machine, only a finite portion of these integers are representable for a given integral type.

Typically, an int is stored in either 2 bytes (= 16 bits) or in 4 bytes (= 32 bits). There are other possibilities, but this is what happens in most C systems. On older PCs, an int is typically stored in 2 bytes. In newer PCs, and in workstations and mainframes, an int is typically stored in 4 bytes.

Because the size of an int varies from one C system to another, the number of distinct values that an int can hold is system-dependent. Suppose that we are on a computer that has 4-byte ints. Because 4 bytes equals 32 bits, an int can take on 2^{32} distinct states. Half of these states are used to represent negative integers and half are used to represent nonnegative integers:

$$-2^{31},\ -2^{31}+1,\ \cdots,\ -3,\ -2,\ -1,\ 0,\ 1,\ 2,\ 3,\ \cdots,\ 2^{31}-1$$

If, on the other hand, we are using a computer that has 2-byte words, then an int can take on only 2^{16} distinct states. Again, half of these states are used to represent negative integers, and half are used to represent nonnegative integers:

$$-2^{15},\ -2^{15}+1,\ \cdots,\ -3,\ -2,\ -1,\ 0,\ 1,\ 2,\ 3,\ \cdots,\ 2^{15}-1$$

Let N_{min_int} represent the smallest integer that can be stored in an int, and let N_{max_int} represent the largest integer that can be stored in an int. If i is a variable of type int, the range of values that i can take on is given by

$$N_{min\_int} \le i \le N_{max\_int}$$

with the end points of the range being machine-dependent. The typical situation is as follows:

On machines with 4-byte words:

$$N_{min\_int} = -2^{31} = -2147483648 \simeq -2 \textit{ billion}$$
$$N_{max\_int} = +2^{31} - 1 = +2147483647 \simeq +2 \textit{ billion}$$

On machines with 2-byte words:

$$N_{min\_int} = -2^{15} = -32768 \simeq -32 \textit{ thousand}$$
$$N_{max\_int} = +2^{15} - 1 = +32767 \simeq +32 \textit{ thousand}$$

On any machine, the following code is syntactically correct:

```
#define   BIG   2000000000     /* 2 billion */

int main(void)
{
   int   a, b = BIG, c = BIG;
   a = b + c;     /* out of range? */
   .....
```

However, at run-time the variable a may be assigned an incorrect value. The logical value of the expression b + c is 4 billion, which is greater than N_{max_int}. This condition is called an *integer overflow*. Typically, when an integer overflow occurs, the program continues to run, but with logically incorrect results. For this reason, the programmer must strive at all times to keep the values of integer expressions within the proper range.

In addition to decimal integer constants, there are hexadecimal integer constants such as 0x1a and octal integer constants such as 0377. (See Section 3.12, "Hexadecimal and Octal Constants," on page 134.) Many C programmers have no particular need for hexadecimal and octal numbers, but all programmers have to know that integers that begin with a leading 0 are not decimal integers. For example, 11 and 011 do not have the same value.

3.5 The Integral Types short, long, and unsigned

In C, the data type int is considered the "natural" or "usual" type for working with integers. The other integral types, such as char, short, and long, are intended for more specialized use. The data type short, for example, might be used in situations where storage is of concern. The compiler may provide less storage for a short than for an int, although it is not required to do so. In a similar fashion, the type long might be used in situations where large integer values are needed. The compiler may provide more storage for a long than for an int, although it is not required to do so. Typically, a short is stored in 2 bytes and a long is stored in 4 bytes. Thus, on machines with 4-byte words, the size of an int is the same as the size of a long, and on machines with

2-byte words, the size of an `int` is the same as the size of a `short`. If `s` is a variable of type `short`, then the range of values that `s` can take on is given by

$$N_{min\_short} \le small \le N_{max\_short}$$

where typically

$$N_{min\_short} = -2^{15} = -32768 \simeq -32\ thousand$$
$$N_{max\_short} = +2^{15} - 1 = +32767 \simeq +32\ thousand$$

If `big` is a variable of type `long`, then the range of values that `big` can take on is given by

$$N_{min\_long} \le big \le N_{max\_long}$$

where typically

$$N_{min\_long} = -2^{31} = -2147483648 \simeq -2\ billion$$
$$N_{max\_long} = +2^{31} - 1 = +2147483647 \simeq +2\ billion$$

A variable of type `unsigned` is stored in the same number of bytes as an `int`. However, as the name implies, the integer values stored have no sign. Typically, variables of type `int` and `unsigned` are stored in a machine word. If `u` is a variable of type `unsigned`, then the range of values `u` can take on is given by

$$0 \le u \le 2^{wordsize} - 1$$

The typical situation is as follows: On machines with 4-byte words

$$N_{max\_unsigned} = 2^{32} - 1 = +4294967295 \simeq +4\ billion$$

On machines with 2-byte words

$$N_{max\_unsigned} = 2^{32} - 1 = +65535 \simeq +65\ thousand$$

Arithmetic on unsigned variables is performed modulo $2^{wordsize}$. (See exercise 16, on page 143.)

Suffixes can be appended to an integer constant to specify its type. The type of an unsuffixed integer constant is either `int`, `long`, or `unsigned long`. The system chooses the first of these types that can represent the value. For example, on machines with 2-byte words, the constant `32000` is of type `int`, but `33000` is of type `long`.

Combining long and unsigned		
Suffix	Type	Example
`u or U`	`unsigned`	`37U`
`l or L`	`long`	`37L`
`ul or UL`	`unsigned long`	`37UL`

3.6 The Floating Types

ANSI C provides the three floating types: `float`, `double`, and `long double`. Variables of this type can hold real values such as 0.001, 2.0, and 3.14159. A suffix can be appended to a floating constant to specify its type. Any unsuffixed floating constant is of type `double`. Unlike other languages, the working floating type in C is `double`, not `float`.

Combining float and unsigned		
Suffix	Type	Example
`f or F`	`float`	`3.7F`
`l or L`	`long double`	`3.7L`

Integers are representable as floating constants, but they must be written with a decimal point. For example, the constants `1.0` and `2.0` are both of type `double`, whereas the constant `3` is an `int`.

In addition to the ordinary decimal notation for floating constants, there is an exponential notation, as in the example `1.234567e5`. This corresponds to the scientific notation 1.234567×10^5. Recall that

$$\begin{aligned} 1.234567 \times 105 &= 1.234567 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10 \\ &= 1.234567 \times 100000 \\ &= 123456.7 \qquad (\textit{decimal point shifted five places}) \end{aligned}$$

In a similar fashion, the number `1.234567e-3` calls for shifting the decimal point three places to the left to obtain the equivalent constant `0.001234567`.

Now we want to carefully describe the exponential notation. After we give the precise rules, we will show some examples. A floating constant such as `333.77777e-22` may not contain any embedded blanks or special characters. Each part of the constant is given a name:

Floating-point constant parts for 333.7777e-22		
Integer	Fraction	Exponent
333	77777	e-22

A floating constant may contain an integer part, a decimal point, a fractional part, and an exponential part. A floating constant *must* contain either a decimal point or an exponential part or both. If a decimal point is present, either an integer part or fractional part or both *must* be present. If no decimal point is present, then there must be an integer part along with an exponential part.

```
floating_constant ::= f_constant   {f_suffix} opt
f_constant      ::=      i_part     .     f_part   e_part
                   |     i_part     .     f_part
                   |     i_part     .
                   |                .     f_part
                   |                .     f_part   e_part
                   |     i_part           e_part
i_part ::= integer_part ::= {digit} 1+
f_part ::= fractional_part ::= {digit} 1+
e_part ::= exponential_part ::= {e|E} 1 {+|-} opt {digit}1+
f_suffix ::= floating_suffix ::= f | F | l | L
```

Some examples of floating constants are

```
3.14159
314.159e-2F    /* of type float */
0e0            /* equivalent to 0.0 */
1.             /* equivalent to 1.0, but harder to read */
```

but not

```
3.14,159       /* comma not allowed */
314159         /* decimal point or exponent part needed */
.e0            /* integer or fractional part needed */
-3.14159       /* this is floating constant expression */
```

Typically, a C compiler will provide more storage for a variable of type `double` than for one of type `float`, although it is not required to do so. On most machines, a `float` is stored in 4 bytes, and a `double` is stored in 8 bytes. The effect of this is that a `float` stores about 6 decimal places of accuracy, and a `double` stores about 15 decimal places of accuracy. An ANSI C compiler may provide more storage for a variable of type `long double` than for one of type `double`, though it is not required to do so. Many compilers implement a `long double` as a `double`. (See exercise 15, on page 143.)

The possible values that a floating type can be assigned are described in terms of attributes called *precision* and *range*. The precision describes the number of significant decimal places that a floating value carries. The range describes the limits of the largest and smallest positive floating values that can be represented in a variable of that type. A `float` on many machines has an approximate precision of 6 significant figures and an approximate range of 10^{-38} to 10^{+38}. This means that a positive `float` value is represented in the machine in the form (only approximately true)

$$0.d_1d_2d_3d_4d_5d_6 \times 10^n$$

where each d_i is a decimal digit; the first digit, d_i, is positive and $-38 \times n \times +38$. The representation of a `float` value in a machine is actually in base 2, not base 10, but the ideas as presented give the correct flavor.

A `double` on many machines has an approximate precision of 15 significant figures and approximate range of 10^{-308} to 10^{+308}. This means that a positive `double` value is represented in the machine in the form (only approximately true)

$$0.d_1d_2 \cdots d_{15} \times 10^n$$

where each d_i is a decimal digit, the first digit, d_i, is positive; and $-308 \le n \le +308$. Suppose `x` is a variable of type `double`. Then the statement

```
x = 123.45123451234512345;      /* 20 significant digits */
```

will result in `x` being assigned a value that is stored in the form (only approximately true)

$$0.123451234512345 \times 10^{+3} \qquad (15 \textit{ significant digits})$$

The main points you must be aware of are (1) not all real numbers are representable, and (2) floating arithmetic operations, unlike the integer arithmetic operations, need not be exact. For small computations this is usually of no concern. For very large computations, such as numerically solving a large system of ordinary differential equations, a good understanding of rounding effects, scaling, and so on may be necessary. This is the domain of numerical analysis.

3.7 The Use of typedef

The C language provides the `typedef` mechanism, which allows the programmer to explicitly associate a type with an identifier. Some examples are

```
typedef   char            uppercase;
typedef   int             INCHES, FEET;
typedef   unsigned long   size_t;        /* found in stddef.h */
```

In each of these type definitions, the named identifiers can be used later to declare variables or functions in the same way ordinary types can be used. Thus,

```
uppercase   u;
INCHES      length, width;
```

declares the variable `u` to be of type `uppercase`, which is synonymous with the type `char`, and it declares the variables `length` and `width` to be of type `INCHES`, which is synonymous with the type `int`.

What is gained by allowing the programmer to create a new nomenclature for an existing type? One gain is in abbreviating long declarations. Another is having type names that reflect the intended use. Furthermore, if there are system-sensitive declarations, such as an `int` that is 4 bytes on one system and 2 bytes on another, and these differences are critical to the program, then the use of `typedef` may make the porting of the software easier. In later chapters, after we introduce enumeration types and structure types, we will see that the `typedef` facility gets used routinely.

3.8 The sizeof Operator

C provides the unary operator `sizeof` to find the number of bytes needed to store an object. It has the same precedence and associativity as all the other unary operators. An expression of the form

`sizeof(`*object*`)`

returns an integer that represents the number of bytes needed to store the object in memory. An object can be a type such as `int` or `float`, or it can be an expression such as `a + b`, or it can be an array or structure type. The following program uses this operator. On a given machine it provides precise information about the storage requirements for the fundamental types.

```
/* Compute the size of some fundamental types. */

#include <stdio.h>

int main(void)
{
   printf("The size of some fundamental types is computed.\n\n");
   printf("       char:%3u byte \n", sizeof(char));
   printf("      short:%3u bytes\n", sizeof(short));
   printf("        int:%3u bytes\n", sizeof(int));
   printf("       long:%3u bytes\n", sizeof(long));
   printf("   unsigned:%3u bytes\n", sizeof(unsigned));
   printf("      float:%3u bytes\n", sizeof(float));
   printf("     double:%3u bytes\n", sizeof(double));
   printf("long double:%3u bytes\n", sizeof(long double));
   return 0;
}
```

Because the C language is flexible in its storage requirements for the fundamental types, the situation can vary from one machine to another. However, it is guaranteed that

```
sizeof(char) = 1
sizeof(char) <= sizeof(short) ≤ sizeof(int) ≤ sizeof(long)
sizeof(signed) = sizeof(unsigned) = sizeof(int)
sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)
```

All the signed and unsigned versions of each of the integral types are guaranteed to have the same size.

Notice that we wrote `sizeof(...)` as if it were a function. It is not, however—it is an operator. If `sizeof` is being applied to a type, then parentheses are required; otherwise they are optional. The type returned by the operator is typically `unsigned`.

3.9 The Use of getchar() and putchar()

In this section, we illustrate the use of `getchar()` and `putchar()`. These are macros defined in *stdio.h* that are used to read characters from the keyboard and to print characters on the screen, respectively. Although there are technical differences, a macro is used as a function is used. (See Section 8.6, "An Example: Macros with Arguments," on page 377.) These macros, as well as others, are often used when manipulating character data.

In memory, a `char` is stored in 1 byte, and an `int` is stored typically in either 2 or 4 bytes. Because of this, an `int` can hold all the values that can be stored in a `char`, and more. We can think of a `char` as a small integer type, and, conversely, we can think of an `int` as a large character type. This is a fundamental idea, and, unfortunately, a difficult one for beginning C programmers.

Our next program is called *double_out*. It reads characters one after another from the standard input file, which is normally connected to the keyboard, and writes each character twice to the standard output file, which is normally connected to the screen.

In file double_out.c

```
#include <stdio.h>

int main(void)
{
   int   c;

   while ((c = getchar()) != EOF) {
      putchar(c);
      putchar(c);
   }
   return 0;
}
```

Dissection of the *double_out* Program

■ `#include <stdio.h>`

One line of this header file is

■ `#define   EOF   (-1)`

The identifier `EOF` is mnemonic for "end-of-file." What is actually used to signal an end-of-file mark is system-dependent. Although the `int` value -1 is often used, different systems can have different values. By including the file *stdio.h* and using the symbolic constant `EOF`, we have made the program portable. This means that the source file can be moved to a different system and run with no changes. The header file *stdio.h* also contains the macro definitions of `getchar()` and `putchar()`.

■ `int   c;`

The variable `c` has been declared as an `int` rather than a `char`. Whatever is used to signal the end of a file, it cannot be a value that represents a character. Because `c` is an `int`, it can hold all possible character values as well as the special value `EOF`.

■ `while ((c = getchar()) != EOF) {`
`.....`

The expression

```
(c = getchar()) != EOF
```

is composed of two parts. The subexpression `c = getchar()` gets a value from the keyboard and assigns it to the variable `c`, and the value of the subexpression takes on that value as well. The symbols `!=` represent the "not equal" operator. As long as the value of the subexpression `c = getchar()` is not equal to `EOF`, the body of the `while` loop is executed. The parentheses around the subexpression `c = getchar()` are necessary. Suppose we had left out the parentheses and had typed

```
c = getchar() != EOF
```

Because of operator precedence this is equivalent to

```
c = (getchar() != EOF)
```

which is syntactically correct, but not what we want.

■ `putchar(c);`

The value of `c` is written to the standard output stream in the format of a character.

Characters have an underlying integer-valued representation that on most C systems is the numeric value of their ASCII representation. For example, the character constant `'a'` has the value 97. The values of both the lower- and uppercase letters occur in order. Because of this, the expression `'a' + 1` has the value `'b'`, the expression `'b' + 1` has the value `'c'`, and so on. Also, because there are 26 letters in the alphabet, the expression `'z' - 'a'` has the value 25. Consider the expression `'A' - 'a'`. It has a value that is the same as `'B' - 'b'`, which is the same as `'C' - 'c'`, and so on. Because of this, if the variable `c` has the value of a lowercase letter, then the expression `c + 'A' - 'a'` has the value of the corresponding uppercase letter. These ideas are incorporated into the next program, which capitalizes all lowercase letters.

In file capitalize.c

```
#include <stdio.h>

int main(void)
{
   int   c;

   while ((c = getchar()) != EOF)
      if (c >= 'a' && c <= 'z')
         putchar(c + 'A' - 'a');
      else
         putchar(c);
   return 0;
}
```

Because of operator precedence, the expressions

```
c >= 'a' && c <= 'z'
```

and

```
(c >= 'a')'&' (c <= 'z')
```

are equivalent. The symbols <= represent the operator "less than or equal." The subexpression `c >= 'a'` tests to see if the value `c` is greater than or equal to the value of `'a'`. The subexpression `c <= 'z'` tests to see if the value of `c` is less than or equal to the value `'z'`. The symbols && represent the operator "logical and." If both subexpressions are true, then the expression

```
c >= 'a' && c <= 'z'
```

is true; otherwise it is false. Thus, the expression is true if and only if `c` is a lowercase letter. If the expression is true, then the statement

```
putchar(c + 'A' - 'a');
```

is executed, causing the corresponding uppercase letter to be printed.

3.10 Mathematical Functions

There are no built-in mathematical functions in C. Functions such as

```
sqrt()   pow()   exp()   log()   sin()   cos()   tan()
```

are available in the mathematics library, which is conceptually part of the standard library. All of these functions, except the power function `pow()`, take a single argument of type `double` and return a value of type `double`. The power function takes two arguments of type `double` and returns a value of type `double`. Our next program illustrates the use of `sqrt()` and `pow()`. It asks the user to input a value for `x` and then prints it out, along with the square root of `x` and the value of `x` raised to the `x` power.

In file power_square.c

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

int main(void)
{
   double   x;

   printf("\n%s",
      "The following will be computed:\n"
      "\n"
      "   the square root of x\n"
      "   x raised to the power x\n"
      "\n");

   while (1) {
      printf("Input x: ");
      if (scanf("%lf", &x) != 1)
         break;
      if (x >= 0.0)
         printf("\n%15s%22.15e\n%15s%22.15e\n%15s%22.15e\n\n",
            "x = ", x,
            "sqrt(x) = ", sqrt(x),
            "pow(x, x) = ", pow(x, x));
      else {
         printf("\nSorry, your number must be nonnegative.\n");
         break;
      }
   }
```

```
    printf("\nBye!\n\n");
    return 0;
}
```

If we execute the program and enter 2 when prompted, here is what appears on the screen:

```
The following will be computed:

   the square root of x
   x raised to the power x

Input x: 2

          x =  2.000000000000000e+00
    sqrt(x) =  1.414213562373095e+00
 pow(x, x) =  4.000000000000000e+00

Input x:
```

Dissection of the *sqrt_pow* Program

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

These header files contain function prototypes. In particular, *math.h* contains the prototypes for the functions in the mathematics library. Although it is not recommended to do so, as an alternative to including *math.h*, we can supply our own function prototypes:

- `double   sqrt(double), pow(double, double);`

This declaration should be placed in the file just above `main()`. (Some compilers will complain if the function prototype is placed in the body of `main()` itself.)

- `while (1) {`
  `.....`

Because any nonzero value is considered to be true, the expression 1 creates an infinite `while` loop. We will use a `break` statement to exit the loop.

- `scanf("%lf", &x)`

The format `%lf` is used in the control string because `x` is a `double`. A common error is to use `%f` instead of `%lf`. Notice that we typed `2` when we illustrated the use of this program. Equivalently, we could have typed `2.0` or `2e0` or `0.2e1`. The function call `scanf("%lf", &x)` would have converted each of these to the same `double`. In C source code, `2` and `2.0` are different. The first is of type `int`, and the second is of type `double`. The input stream that is read by `scanf()` is *not* source code, so the rules for source code do not apply. When `scanf()` reads in a `double`, the number `2` is just as good as the number `2.0`. (See Section 11.2, "The Input Function `scanf()`," on page 499.)

- `if (scanf("%lf", &x) != 1)`
  `break;`

The function `scanf()` returns the number of successful conversions. To exit the `while` loop we can type "quit" or anything else that `scanf()` cannot convert to a `double`. If the value returned by `scanf()` is not 1, the `break` statement gets executed. This causes program control to exit the while statement. We discuss the `break` statement more fully in Chapter 4, "Flow of Control."

- `if (x >= 0.0)`
  `.....`

Because the square root function is defined only for nonnegative numbers, a test is made to ensure that the value of `x` is nonnegative. A call such as `sqrt(-1.0)` can cause a run-time error. (See exercise 20, on page 144.)

- `printf("\n%15s%22.15e\n%15s%22.15e\n%15s%22.15e\n\n",`
  `"x = ", x,`
  `"sqrt(x) = ", sqrt(x),`
  `"pow(x, x) = ", pow(x, x));`

Notice that we are printing `double` values in the format `%22.15e`. This results in 1 place to the left of the decimal point and 15 places to the right, 16 significant places in all. On our machine, only *n* places are valid, where *n* is between 15 and 16. (The uncertainty comes about because of the translation from binary to decimal.) You can ask for lots of decimal places to be printed, but you should not believe all that you read.

The Use of abs() and fabs()

In many languages, the function `abs()` returns the absolute value of its real argument. In this respect, C is different. In C, the function `abs()` takes an argument of type `int` and returns its absolute value as an `int`. Its function prototype is in *stdlib.h*. For mathematical code, the C programmer should use `fabs()`, which takes an argument of type `double` and returns its absolute value as a `double`. (See exercise 25, on page 145.) Its function prototype is in *math.h*. The name `fabs` stands for floating absolute value.

UNIX and the Mathematics Library

In ANSI C, the mathematics library is conceptually part of the standard library. This means that you should not have to do anything special to get access to mathematical functions. However, on older UNIX systems this is often not the case. Suppose you write a program in a file, say *pgm.c*, that uses the `sqrt()` function. The following command should then compile your program:

```
cc pgm.c
```

If, however, the ANSI C system is not properly connected, you will see something like the following printed on the screen:

```
Undefined symbol: _sqrt
```

This means that the linker looked through the libraries that were made available to it, but was unable to find a *.o* file that contained the object code for the `sqrt()` function. (See Section 11.15, "Libraries," on page 526, for further discussion of libraries.) If you give the command

```
cc pgm.c -lm
```

the mathematics library will be attached, which will allow the loader to find the necessary *.o* file. In *-lm* the letter *l* stands for "library" and the letter *m* stands for "mathematics." As older versions of UNIX give way to newer versions, the necessity of using the *-lm* option disappears.

3.11 Conversions and Casts

An arithmetic expression such as `x + y` has both a value and a type. If both `x` and `y` have type `int`, then the expression `x + y` also has type `int`. But if both `x` and `y` have type `short`, then `x + y` is of type `int`, not `short`. This is because in any expression, a `short` always gets promoted, or converted, to an `int`. In this section we want to give the precise rules for conversions.

The Integral Promotions

A `char` or `short`, either `signed` or `unsigned`, or an enumeration type can be used in any expression where an `int` or `unsigned int` may be used. (See Section 7.5, "Enumeration Types," on page 345.) If all the values of the original type can be represented by an `int`, then the value is converted to an `int`; otherwise, it is converted to an `unsigned int`. This is called an *integral promotion*. Here is an example:

```
char   c = 'A';
printf("%c\n", c);
```

The `char` variable `c` occurs by itself as an argument to `printf()`. However, because an integral promotion takes place, the type of the expression `c` is `int`, not `char`.

The Usual Arithmetic Conversions

Arithmetic conversions can occur when the operands of a binary operator are evaluated. Suppose, for example, that `i` is an `int` and `f` is a `float`. In the expression `i + f`, the operand `i` gets promoted to a `float`, and the expression `i + f` as a whole has type `float`. The rules governing this are called the *usual arithmetic conversions*; those rules follow:

The usual arithmetic conversions:

If either operand is of type `long double`, the other operand is converted to `long double`.

Otherwise, if either operand is of type `double`, the other operand is converted to `double`.

Otherwise, if either operand is of type `float`, the other operand is converted to `float`.

Otherwise, the integral promotions are performed on both operands, and the following rules are applied:

> If either operand is of type `unsigned long`, the other operand is converted to `unsigned long`.
>
> Otherwise, if one operand has type `long` and the other has type `unsigned`, then one of two possibilities occurs:
>
> > If a `long` can represent all the values of an `unsigned`, then the operand of type `unsigned` is converted to `long`.
> >
> > If a `long` cannot represent all the values of an `unsigned`, then both of the operands are converted to `unsigned long`.
>
> Otherwise, if either operand has type `long`, the other operand is converted to `long`.
>
> Otherwise, if either operand has type `unsigned`, the other is converted to `unsigned`.
>
> Otherwise, both operands have type `int`.

This arithmetic conversion has several alternate names:

- automatic conversion
- implicit conversion
- coercion
- promotion
- widening

The following table illustrates the idea of automatic conversion:

Declarations			

`char c;` `short s;` `int i;`
`long l;` `unsigned u;` `unsigned long ul;`
`float f;` `double d;` `long double ld;`			

Expression	Type	Expression	Type
`c - s / i`	`int`	`u * 7 - i`	`unsigned`
`u * 2.0 - i`	`double`	`f * 7 - i`	`float`
`c + 3`	`int`	`7 * s * ul`	`unsigned long`
`c + 5.0`	`double`	`ld + c`	`long double`
`d + s`	`double`	`u - ul`	`unsigned long`
`2 * i / l`	`long`	`u - l`	*system-dependent*

In addition to automatic conversions in mixed expressions, an automatic conversion also can occur across an assignment. For example,

```
d = i
```

causes the value of `i`, which is an `int`, to be converted to a `double` and then assigned to `d`, and `double` is the type of the expression as a whole. A promotion or widening such as `d = i` will usually be well behaved, but a narrowing or demotion such as `i = d` can lose information. Here, the fractional part of `d` will be discarded. If the remaining integer part does not fit into an `int`, then what happens is system-dependent.

Casts

In addition to implicit conversions, which can occur across assignments and in mixed expressions, there are explicit conversions called *casts*. If `i` is an `int`, then

```
(double) i
```

will cast, or convert, the value of `i` so that the expression has type `double`. The variable `i` itself remains unchanged. Casts can be applied to expressions. Some examples are

```
(long) ('A' + 1.0)
f = (float) ((int) d + 1)
d = (double) i / 3
(double) (x = 77)
```

but not

```
(double) x = 77      /* equivalent to ((double) x) = 77, error */
```

The cast operator (*type*) is a unary operator having the same precedence and right-to-left associativity as other unary operators. Thus, the expression

```
(float) i + 3
```

is equivalent to

```
((float) i) + 3
```

because the cast operator (*type*) has higher precedence than +.

3.12 Hexadecimal and Octal Constants

A number represented by a positional notation in base 16 is called a hexadecimal number. There are 16 hexadecimal digits.

Hexadecimal digits and their corresponding decimal values										
Hexadecimal digit:	0	1	...	9	A	B	C	D	E	F
Decimal value:	0	1	...	9	10	11	12	13	14	15

A positive integer written in hexadecimal notation is a string of hexadecimal digits of the form

$$h_n h_{n-1} \ldots h_2 h_1 h_0$$

where each h_i is a hexadecimal digit. It has the value

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + \ldots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

For example,

$$\begin{aligned} \texttt{A0F3C} &= \texttt{A} \times 16^4 + \texttt{0} \times 16^3 + \texttt{F} \times 16^2 + \texttt{3} \times 16^1 + \texttt{C} \times 16^0 \\ &= 10 \times 16^4 + 0 \times 16^3 + 15 \times 16^2 + 3 \times 16^1 + 12 \times 16^0 \\ &= 659260 \end{aligned}$$

Some hexadecimal numbers and their decimal equivalents are given in the following table:

Hexadecimal number	Conversion to decimal
2A	$2 \times 16 + A = 2 \times 16 + 10 = 42$
B3	$B \times 16 + 3 = 11 \times 16 + 3 = 179$
113	$1 \times 16^2 + 1 \times 16 + 3 = 275$

On machines that have 8-bit bytes, a byte is conveniently represented as two hexadecimal digits. Moreover, the representation has two simultaneously valid interpretations. First, one may consider the 8 bits in a byte as representing a number in base 2 notation. That number can be expressed uniquely as a hexadecimal number with two hexadecimal digits. The following table lists 8-bit bytes and corresponding two-digit hexadecimal numbers. For convenience, decimal numbers are listed, and for later reference octal numbers are also listed.

Decimal	Binary	Hexadecimal	Octal
0	00000000	00	000
1	00000001	01	001
2	00000010	02	002
3	00000011	03	003
.....			
31	00011111	1F	037
32	00100000	20	040
.....			
188	10111100	BC	274
.....			
254	11111110	FE	376
255	11111111	FF	377

Another interpretation of this correspondence is also useful. By definition, a *nibble* consists of 4 bits, so a byte is made up of 2 nibbles. Each nibble has a unique represen-

tation as a single hexadecimal digit, and 2 nibbles, making up a byte, are representable as 2 hexadecimal digits. For example,

```
1011 1100        corresponds to        BC
```

Note that this same correspondence occurs in the table. All of this is useful when manipulating the values of variables in bit form.

The octal digits are 0, 1, 2, ... , 7. A positive integer written in octal notation is a string of digits of the form

$$o_n o_{n-1} \ldots o_2 o_1 o_0$$

where each o_i is an octal digit. It has the value

$$o_n \times 8^n + o_{n-1} \times 8^{n-1} + \ldots + o_2 \times 8^2 + o_1 \times 8^1 + o_0 \times 8^0$$

For example,

$$75301 = 7 \times 8^4 + 5 \times 8^3 + 3 \times 8^2 + 0 \times 8^1 + 1 \times 8^0$$

On machines that have words consisting of 24 or 48 bits, it is natural to have words consisting of "bytes" with 6 bits, each "byte" made up of 2 "nibbles" of 3 bits each. In this case a "nibble" has a unique representation as a single octal digit, and a "byte" has a unique representation as two octal digits.

In C source code, positive integer constants prefaced with `0` represent integers in octal notation, and positive integer constants prefaced with `0x` or `0X` represent integers in hexadecimal notation. Just as with decimal integer constants, octal and hexadecimal constants may have suffixes appended to specify the type. The letters `A` through `F` and `a` through `f` are used to code hexadecimal digits.

hexadecimal_integer_constant ::= *h_integer_constant* { *i_suffix* }$_{opt}$
h_integer_constant ::= { `0x` | `0X` }$_1$ { *hexadecimal_digit* }$_{1+}$
hexadecimal_digit ::= `0` | `1` | ... | `9` | `a` | `A` | ... | `f` | `F`
octal_integer_constant ::= *o_integer_constant* { *i_suffix* }$_{opt}$
o_integer_constant ::= `0` { *octal_digit* }$_{1+}$
octal_digit ::= `0` | `1` | ... | `7`
i_suffix ::= *integer_suffix* ::= { `u` | `U` }$_{opt}$ { `l` | `L` }$_{opt}$

Let us write a program to illustrate these ideas. We will show the output of each `printf()` statement as a comment.


```
/* Decimal, hexadecimal, octal conversions. */

#include <stdio.h>

int main(void)
{
   printf("%d  %x  %o\n", 19, 19, 19);          /* 19  13  23 */
   printf("%d  %x  %o\n", 0x1c, 0x1c, 0x1c);    /* 28  1c  34 */
   printf("%d  %x  %o\n", 017, 017, 017);       /* 15   f  17 */
   printf("%d\n", 11 + 0x11 + 011);             /* 37         */
   printf("%x\n", 2097151);                     /* 1fffff     */
   printf("%d\n", 0x1FfFFf);                    /* 2097151    */
   return 0;
}
```

On machines with 2-byte `int`s, the last two formats must be changed to `%lx` and `%ld`, respectively. The functions `printf()` and `scanf()` use the conversion characters and `d`, `x`, and `o` in conversion specifications for decimal, hexadecimal, and octal, respectively. With `printf()`, formats of the form `%x` and `%o` cause integers to be printed out in hexadecimal and octal notation, but not prefaced with `0x` or `0`. The formats `%#x` and `%#o` can be used to get the prefixes. (See Section 11.1, "The Output Function `printf()`," on page 493, for further discussion.) *Caution:* When using `scanf()` to read in a hexadecimal number, do not type an `0x` prefix.

C.13 Summary

1 The fundamental data types are `char`, `short`, `int`, `long`, unsigned versions of these, and three floating types. The type `char` is a 1-byte integral type mostly used for representing characters.

2 The type `int` is designed to be the "natural" or "working" integral type. The other integral types such as `short`, `long`, and `unsigned` are provided for more specialized situations.

3 Three floating types, `float`, `double`, and `long double`, are provided to represent real numbers. Typically, a `float` is stored in 4 bytes and a `double` in 8 bytes. The number of bytes used to store a `long double` varies from one compiler to another. However, as compilers get updated, the trend is to store a `long double` in 16 bytes. The type `double`, not `float`, is the "working" type.

4 Unlike integer arithmetic, floating arithmetic is not always exact. Engineers and numerical analysts often have to take roundoff effects into account when doing extensive calculations with floating-point numbers.

5 The unary operator `sizeof` can be used to find the number of bytes needed to store a type or the value of an expression. For example, `sizeof(int)` is 2 on some older small machines and is 4 on most new machines that have 32-bit words.

6 The usual mathematical functions, such as `sin()`, `cos()`, and `tan()`, are available in the mathematics library. Most of the functions in the library take a single argument of type `double` and return a value of type `double`. The standard header file *math.h* should be included when using these functions.

7 Automatic conversions occur in mixed expressions and across an equal sign. Casts can be used to force explicit conversions.

8 Integer constants beginning with `0x` and `0` designate hexadecimal and octal integers, respectively.

9 Suffixes can be used to explicitly specify the type of a constant. For example, `3U` is of type `unsigned`, and `7.0F` is of type `float`.

10 A character constant such as `'A'` is of type `int` in C, but it is of type `char` in C++. This is one of the few places where C++ differs from C.

Exercises

1 Not all real numbers are machine-representable; there are too many of them. Thus, the numbers that are available on a machine have a "graininess" to them. As an example of this, the code

```
double   x = 123.45123451234512345;
double   y = 123.45123451234512300;  /* last two digits zero */

printf("%.17f\n%.17f\n", x, y);
```

causes two identical numbers to be printed. How many zeros must the initializer for `y` end with to get different numbers printed? Explain your answer.

2 The mathematical formula

$$\sin^2(x) + \cos^2(x) = 1$$

holds for all x real. Does this formula hold on your machine? Try the following program:

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

int main(void)
{
   double   two_pi = 2.0 * M_PI;   /* in math.h */
   double   h      = 0.1;          /* step size */
   double   x;

   for (x = 0.0; x < two_pi; x += h)
      printf("%5.1f: %.15e\n",
         x, sin(x) * sin(x) + cos(x) * cos(x));
   return 0;
}
```

What happens if the format `%.15e` is changed to `%.15f`? Explain.

3 Write a program that prints a table of trigonometric values for `sin()`, `cos()`, and `tan()`. The angles in your table should go from 0 to 2π in 20 steps.

4 Write a test program to find out whether the `printf()` function truncates or rounds when writing a `float` or `double` with a fractional part. The ANSI standard requires rounding, but some systems do not do this. What happens on your machine?

5 Use the following code to print out a list of powers of 2 in decimal, hexadecimal, and octal:

```
int   i, val = 1;

for (i = 1; i < 35; ++i) {
   printf("%15d%15u%15x%15o\n", val, val, val, val);
   val *= 2;
}
```

What gets printed? Explain. Powers of 2 have a special property when written in hexadecimal and octal notation. What is the property?

6 What happens when the arithmetic overflows? Try the following code on your system:

```
int   big_big = 2000000000 + 2000000000;

printf("%d   %u\n", big_big, big_big);
```

If you are working on a machine with a 2-byte `int`, change `2000000000` to `32000`. What gets printed? Explain. Does the output change if `big_big` is declared to be `unsigned` instead of `int`?

7 Study the following code carefully without running it:

```
printf("Why is 21 + 31 equal to %d?\n", 21 + 31);
```

On a machine with 4-byte `int`s, here is what gets printed:

```
Why is 21 + 31 equal to 5?
```

Can you see why? Can you deduce the moral?

8 In mathematics the Greek letter ε, called "epsilon," is often used to represent a small positive number. Although it can be arbitrarily small in mathematics, on a machine there is no such concept as "arbitrarily small." In numerical analysis it is convenient sometimes to declare `eps` (for "epsilon") as a variable of type `double` and to assign to `eps` the smallest positive number with the property that

```
1.0 < 1.0 + eps
```

is *true*. This number is machine-dependent. See if you can find `eps` on your machine. Begin by assigning the value `1e-37` to `eps`. You will find that for this value the expression is *false*.

9 If you expand the two functions `tan(sin(x))` and `sin(tan(x))` in a Taylor series about the origin, the two expansions agree for the first seven terms. (If you have access to a computer algebra system such as Maple or Mathematica, you can see this easily.) This means that the difference of the two functions is very flat at the origin. Try the following program:

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

double   f(double x);
```



```
int main(void)
{
   double   x;

   for (x = -0.25; x <= +0.25; x += 0.01)
      printf("f(%+.2f) = %+.15f\n", x, f(x));
   return 0;
}

double f(double x)
{
   return (tan(sin(x)) - sin(tan(x)));
}
```

The output of this program illustrates the flatness of `f()` near the origin. Do you see it? Also, as an experiment remove the plus signs that occur in the formats. You will find that the output does not line up properly.

10 Most machines use the two's complement representation to store integers. On these machines, the value -1 stored in an integral type turns all bits on. Assuming that your system does this, here is one way to determine whether a `char` is equivalent to a `signed char` or to an `unsigned char`. Write a program that contains the lines

```
char            c = -1;
signed char     s = -1;
unsigned char   u = -1;

printf("c = %d   s = %d   u = %d\n", c, s, u);
```

Each of the variables `c`, `s`, and `u` is stored in memory with the bit pattern `11111111`. What gets printed on your system? Can you tell from this what a `char` is equivalent to? Does your ANSI C compiler provide an option to change a plain `char` to, say, an `unsigned char`? If so, invoke the option, recompile your program, and run it again.

11 Explain why the following code prints the largest integral value on your system:

```
unsigned long   val = -1;

printf("The biggest integer value: %lu\n", val);
```

What gets printed on your system? Although technically the value that gets printed is system-dependent, on most systems the value is approximately 4 billion. Explain.

12 A variable of type `char` can be used to store small integer values. What happens if a large value is assigned to a `char` variable? Consider the code

```
char   c1 = 256, c2 = 257;              /* too big! */

printf("c1 = %d\nc2 = %d\n", c1, c2);
```

Your compiler should complain. Does it? Even if it does complain, it will probably produce executable code. Can you guess what gets printed?

13 The following table shows how many bytes are required on most machines to store some of the fundamental types. What are the appropriate values for your machine? Write and execute a program that allows you to complete the table.

Fundamental type	Memory required on machines with 4-byte words	Memory required on machines with 2-byte words	Memory required on your machine
`char`	1 byte	1 byte	
`short`	2 bytes	2 bytes	
`int`	4 bytes	2 bytes	
`unsigned`	4 bytes	2 bytes	
`long`	4 bytes	4 bytes	
`float`	4 bytes	4 bytes	
`double`	8 bytes	8 bytes	
`long double`	?	?	

14 The type of a character constant such as `'A'` is different in C and C++. In C, its type is `int`, but in C++, its type is `char`. Compile the following program first in C and then in C++. In each case, what gets printed?

```
#include <stdio.h>

int main(void)
{
   printf("sizeof('A') = %u\n", sizeof('A'));
   return 0;
}
```


15 Consider the following code:

```
char   c = 'A';

printf("sizeof(c)       = %u\n", sizeof(c));
printf("sizeof('A')     = %u\n", sizeof('A'));
printf("sizeof(c + c)   = %u\n", sizeof(c + c));
printf("sizeof(c = 'A') = %u\n", sizeof(c = 'A'));
```

Write down what you think gets printed; then write a small program to check your answer. How many lines of the output change if you compile the program in C++?

16 Let $N_{min\_u\_long}$ and $N_{max\_u\_long}$ represent the minimum and maximum values that can be stored in an `unsigned long` on your system. What are those values? *Hint:* Read the standard header file *limits.h*.

17 On a 24-hour clock, the zero hour is midnight, and the 23rd hour is 11 o'clock at night, one hour before midnight. On such a clock, when 1 is added to 23, we do not get 24, but instead we get 0. There is no 24. In a similar fashion, 22 plus 5 yields 3, because 22 plus 2 is 0 and 3 more is 3. This is an example of modular arithmetic, or more precisely, of arithmetic modulo 24. Most machines do modular arithmetic on all the integral types. This is most easily illustrated with the unsigned types. Run the following program and explain what gets printed:

```
#include <limits.h>            /* for UINT_MAX */
#include <stdio.h>

int main(void)
{
   int        i;
   unsigned   u = UINT_MAX;    /* typically 4294967295 or 65535 */

   for (i = 0; i < 10; ++i)
      printf("%u + %d = %u\n", u, i, u + i);
   for (i = 0; i < 10; ++i)
      printf("%u * %d = %u\n", u, i, u * i);
   return 0;
}
```

18 The ANSI C standard suggests that the recommendations of *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985) be followed. Does your compiler follow these recommendations? Well, one test is to try to see what happens when you assign a value to a floating variable that is out of its range. Write a small program containing the lines

```
double   x = 1e+307;                    /* big */
double   y = x * x;                     /* too big! */

printf("x = %e   y = %e\n", x, y);
```

Does the value for y get printed as `Inf` of `Infinity`? If so, there is a good chance that your compiler is following ANSI/IEEE Std 754-1985.

19 In mathematics, the numbers e and π are well known. The number e is the base of the natural logarithms and the number π is the ratio of the diameter of a circle to its circumference. Which of the two numbers, e^{π} and π^{e}, is larger? This is a standard problem for students in an honors calculus course. However, even if you have never heard of e and π and know nothing about calculus, you should be able to answer this question. *Hint:*

$e \simeq 2.71828182845904524$ and $\pi \simeq 3.14159265358979324$

20 What happens when the argument to the `sqrt()` function is negative? In some compilers a call such as `sqrt(-1.0)` causes a run-time error to occur, whereas in other compilers the special value `NaN` gets returned. The value `NaN` is called "not a number." What happens on your system? To find out, write a test program that contains the line

```
printf("sqrt(-1.0) = %f\n", sqrt(-1.0));
```

21 If the value of `x` is too large, the call `pow(x, x)` may cause a run-time error or may cause the word `Inf` or `Infinity` to appear on the screen when a `printf()` statement is used to print the value produced by `pow(x, x)`. What is the largest integer value for `x` such that the statement

```
printf("pow(%.1f, %.1f) = %.7e\n", x, x, pow(x, x));
```

does not cause a run-time error and does not cause `Inf` or `Infinity` to be printed? *Hint:* Put the statement in a `for` loop.

22 In traditional C, any `float` was automatically promoted to a `double`. In ANSI C, a compiler can promote a `float` to a `double` in arithmetic expressions, but is not required to do so. We find that most ANSI C compilers perform arithmetic on `float`s without any widening. What happens on your compiler? Try the following code:

```
float   x = 1.0, y = 2.0;
```



```
printf("%s%u\n%s%u\n%s%u\n",
   "sizeof(float)  = ", sizeof(float),
   "sizeof(double) = ", sizeof(double),
   "sizeof(x + y)  = ", sizeof(x + y));
```

23 In traditional C, the types `long float` and `double` are synonymous. However, because `long float` is harder to type, it was not popular and was rarely used. In ANSI C, the type `long float` has been eliminated. Nonetheless, many ANSI C compilers still accept it. Check to see if this is true on your compiler.

24 Write a program called *try_me* that contains the following lines:

```
int   c;

while ((c = getchar()) != EOF)
   putchar(c);
```

Create a small text file, say infile, and give the command

try_me < infile > outfile

to copy the contents of *infile* to *outfile*. (Make sure that you actually look at what is in *outfile*.) Is there any text that you can put into *infile* that will not get copied to *outfile*? What about `EOF`, or the value of `EOF`, which is typically –1? Do these cause a problem? Explain.

25 In mathematical code, the use of `abs()` instead of `fabs()` can be disastrous. Try the following program on your machine:

```
#include <math.h>       /* for fabs() */
#include <stdio.h>
#include <stdlib.h>     /* for abs() */

int main(void)
{
   double   x = -2.357;

   printf(" abs(x) = %e\n", abs(x));      /* wrong! */
   printf("fabs(x) = %e\n", fabs(x));
   return 0;
}
```

Some compilers give a warning, others do not. What happens on your machine? Here, the programmer can easily spot that something is wrong. But sometimes the use of `abs()` is deeply embedded in lots of other code, making its misuse difficult to spot.

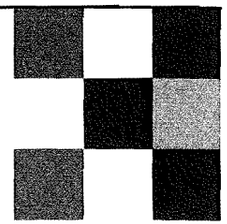
26 In C, the letters used in a hexadecimal constant can be either upper- or lowercase, or both. Consider the following code:

```
int   a = 0xabc;
int   b = 0xABc;
int   c = 0XABC;

printf("a = %d   b = %d   c = %d\n", a, b, c);
```

Write down what you think gets printed. Then write a program containing these lines to check your answer.

27 The following code is machine-dependent. If when you execute the code it does the unexpected, see if you can explain what is happening.

```
char   c = 0xff;

if (c == 0xff)
   printf("Truth!\n");
else
   printf("This needs to be explained!\n");
```



Chapter 4

Flow of Control

Statements in a program are normally executed one after another. This is called *sequential flow of control.* Often it is desirable to alter the sequential flow of control to provide for a choice of action, or a repetition of action. By means of `if`, `if-else`, and `switch` statements, a selection among alternative actions can be made. By means of `while`, `for`, and `do` statements, iterative actions can be taken. Because the relational, equality, and logical operators are heavily used in flow of control constructs, we begin with a thorough discussion of these operators. They are used in expressions that we think of as being *true* or *false.* We also discuss the compound statement, which is used to group together statements that are to be treated as a unit.

4.1 Relational, Equality, and Logical Operators

The following table contains the operators that are often used to affect flow of control:

Relational, equality, and logical operators		
Relational operators	less than greater than less than or equal to greater than or equal to	< > <= >=
Equality operators	equal to not equal to	== !=
Logical operators	(unary) negation logical and logical or	! && ||

Just as with other operators, the relational, equality, and logical operators have rules of precedence and associativity that determine precisely how expressions involving these operators are evaluated.

Operator precedence and associativity	
Operators	Associativity
() ++ (*postfix*) -- (*postfix*)	left to right
+ (*unary*) - (*unary*) ++ (*prefix*) -- (*prefix*)	right to left
* / %	left to right
+ -	left to right
< <= > >=	left to right
== !=	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= *etc*	right to left
, (*comma operator*)	left to right

The ! operator is unary. All the other relational, equality, and logical operators are binary. They operate on expressions and yield either the `int` value 0 or the `int` value 1. The reason for this is that in the C language, *false* is represented by the value zero and *true* is represented by any nonzero value. The value for *false* can be any zero value; it can be a 0 or 0.0 or the null character `'\0'` or the `NULL` pointer value. The value for *true* can be any nonzero value. Intuitively, an expression such as `a < b` is either *true* or *false*. In C, this expression will yield the `int` value 1 if it is *true* and the `int` value 0 if it is *false*.

4.2 Relational Operators and Expressions

The relational operators

```
<        >        <=        >=
```

are all binary. They each take two expressions as operands and yield either the `int` value 0 or the `int` value 1.

relational_expression ::= *expr* < *expr*
| *expr* > *expr*
| *expr* <= *expr*
| *expr* >= *expr*

Some examples are

```
a < 3
a > b
-1.3 >= (2.0 * x + 3.3)
a < b < c                    /* syntactically correct, but confusing */
```

but not

```
a =< b          /* out of order */
a < = b         /* space not allowed */
a >> b          /* this is a shift expression */
```

Consider a relational expression such as `a < b`. If `a` is less than `b`, then the expression has the `int` value 1, which we think of as being *true*. If `a` is not less than `b`, then the expression has the `int` value 0, which we think of as *false*. Mathematically, the value of `a < b` is the same as the value of `a - b < 0`. Because the precedence of the relational operators is less than that of the arithmetic operators, the expression

`a - b < 0` is equivalent to `(a - b) < 0`

On many machines, an expression such as `a < b` is implemented as `a - b < 0`. The usual arithmetic conversions occur in relational expressions.

Let *e1* and *e1* be arbitrary arithmetic expressions. The following table shows how the value of *e1* - *e2* determines the values of relational expressions:

Values of relational expressions				
a - b	a < b	a > b	a <= b	a >= b
positive	0	1	0	1
zero	0	0	1	1
negative	1	0	1	0

The following table illustrates the use of the rules of precedence and associativity to evaluate relational expressions:

Declarations and initializations		

`char c = 'w'`
`int i = 1, j = 2, k = -7`
`double x = 7e+33, y = 0.001`		

Expression	Equivalent expression	Value
`'a' + 1 < c`	`('a' + 1) < c`	1
`- i - 5 * j >= k + 1`	`((- i) - (5 * j)) >= (k + 1)`	0
`3 < j < 5`	`(3 < j) < 5`	1
`x - 3.333 <= x + y`	`(x - 3.333) <= (x + y)`	1
`x < x + y`	`x < (x + y)`	0

Two expressions in this table give surprising results in that they do not conform to rules as written mathematically. In mathematics, one writes

```
3 < j < 5
```

to indicate that the variable `j` has the property of being greater than 3 and less than 5. It can also be considered as a mathematical statement that, depending on the value of j, may or may not be true. For example, if `j = 4`, then the mathematical statement

```
3 < j < 5
```

is true, but if `j = 7`, then the mathematical statement is false. Now consider the C code

```
j = 7;
printf("%d\n", 3 < j < 5);      /* 1 gets printed, not 0 */
```


By analogy with mathematics, one might expect that the expression is *false* and that 0 is printed. However, that is not the case. Because relational operators associate from left to right,

`3 < j < 5` is equivalent to `(3 < j) < 5`

Because the expression `3 < j` is *true*, it has value 1. Thus,

`(3 < j) < 5` is equivalent to `1 < 5`

which has value 1. In C, the correct way to write a test for both `3 < j` and `j < 5` is

```
3 < j && j < 5
```

Because the relational operators have higher precedence than the binary logical operators, this is equivalent to

```
(3 < j) && (j < 5)
```

and, as we will see later, this expression is *true* if and only if both operands of the && expression are *true*.

The numbers that are representable in a machine do not have infinite precision. Sometimes, this can cause unexpected results. In mathematics the relation

`x < x + y` is equivalent to `0 < y`

Mathematically, if `y` is positive, then both of these relations are logically true. Computationally, if `x` is a floating variable with a large value such as `7e+33` and `y` is a floating variable with a small value such as `0.001`, then the relational expression

```
x < x + y
```

may be *false*, even though mathematically it is true. An equivalent expression is

```
(x - (x + y)) < 0.0
```

and it is this expression that the machine implements. If, in terms of machine accuracy, the values of `x` and `x + y` are equal, the expression will yield the `int` value 0.

4.3 Equality Operators and Expressions

The equality operators == and != are binary operators acting on expressions. They yield either the `int` value 0 or the `int` value 1. The usual arithmetic conversions are applied to expressions that are the operands of the equality operators.

equality_expression ::= *expr* == *expr* | *expr* != *expr*
expr ::= *expr*

Some examples are

```
c == 'A'
k != -2
x + y == 3 * z - 7
```

but not

```
a = b          /* an assignment statement */
a = = b - 1    /* space not allowed */
(x + y) =! 44  /* syntax error: equivalent to (x + y) = (!44) */
```

Intuitively, an equality expression such as `a == b` is either *true* or *false*. An equivalent expression is `a - b == 0`, and this is what is implemented at the machine level. If `a` equals `b`, then `a - b` has value 0 and `0 == 0` is *true*. In this case the expression `a == b` will yield the `int` value 1, which we think of as *true*. If `a` is not equal to `b`, then `a == b` will yield the `int` value 0, which we think of as *false*.

The expression `a != b` uses the not equal operator. It is evaluated in a similar fashion, except that the test here is for inequality rather than for equality. The operator semantics is given by the following table:

Values of:		
expr1 - expr2	expr1 == expr2	expr1 != expr2
zero	1	0
nonzero	0	1

The next table shows how the rules of precedence and associativity are used to evaluate some expressions with equality operators.

Declarations and initializations		
`int i = 1, j = 2, k = 3;`		
Expression	Equivalent expression	Value
`i == j`	`j == i`	0
`i != j`	`j != i`	1
`i + j + k == - 2 * - k`	`((i + j) + k) == ((- 2) * (- k))`	1

Observe that the expression

`a != b` is equivalent to `!(a == b)`

Also, note carefully that the two expressions

`a == b` and `a = b`

are *visually* similar. They are close in form but radically different in function. The expression `a == b` is a test for equality, whereas `a = b` is an assignment expression. Writing

```
if (a = 1)
    .....          /* do something */
```

instead of

```
if (a == 1)
    .....          /* do something */
```

is a common programming error. The expression in the first `if` statement is always *true*, and an error such as this can be very difficult to find.

4.4 Logical Operators and Expressions

The logical operator ! is unary, and the logical operators && and || are binary. All of these operators, when applied to expressions, yield either the `int` value 0 or the `int` value 1.

Logical negation can be applied to an expression of arithmetic or pointer type. If an expression has value zero, then its negation will yield the `int` value 1. If the expression has a nonzero value, then its negation will yield the `int` value 0.

logical_negation_expression ::= ! *expr*

Some examples are

```
!a
!(x + 7.7)
!(a < b || c < d)
```

but not

```
a!          /* out of order */
a != b      /* != is the token for the "not equal" operator */
```

The usual arithmetic conversion rules are applied to expressions that are the operands of !. The following table gives the semantics of the ! operator:

Values of:	
expr	*! expr*
zero	1
nonzero	0

Although logical negation is a very simple operator, there is one subtlety. The operator ! in C is unlike the *not* operator in ordinary logic. If *s* is a logical statement, then

not (not s) = s

whereas in C the value of !!5, for example, is 1. Because ! associates from right to left, the same as all other unary operators, the expression

!!5 is equivalent to !(!5)

and !(!5) is equivalent to !(0), which has value 1. The following table shows how some expressions with logical negation are evaluated:

Declarations and initializations		

`char c = 'A';`
`int i = 7, j = 7;`
`double x = 0.0, y = 2.3`		

Expression	Equivalent expression	Value
`! c`	`! c`	0
`! (i - j)`	`! (i - j)`	1
`! i - j`	`(! i) - j`	-7
`! ! (x + y)`	`! (! (x + y))`	1
`! x * ! ! y`	`(! x) * (!(! y))`	1

The binary logical operators && and || also act on expressions and yield either the `int` value 0 or the `int` value 1. The syntax for a logical expression is given by

logical_expression ::= *logical_negation_expression*
| *logical_or_expression*
| *logical_and_expression*
logical_or_expression ::= *expr* || *expr*
logical_and_expression ::= *expr* && *expr*

Some examples are

```
a && b
a || b
!(a < b) && c
3 && (-2 * a + 7)
```

but not

```
a &&          /* one operand missing */
a |  | b      /* extra space not allowed */
a & b         /* this is a bitwise operation */
&b            /* the address of b */
```

The operator semantics is given by the following table:

Values of:			
expr1	expr2	expr1 && expr2	expr1 || expr2
zero	zero	0	0
zero	nonzero	0	1
nonzero	zero	0	1
nonzero	nonzero	1	1

The precedence of && is higher than ||, but both operators are of lower precedence than all unary, arithmetic, and relational operators. Their associativity is left to right. The next table shows how the rules of precedence and associativity are used to compute the value of some logical expressions.

Declarations and initializations

```
char     c = 'B';
int      i = 3, j = 3, k = 3;
double   x = 0.0, y = 2.3;
```

Expression	Equivalent expression	Value
`i && j && k`	`(i && j) && k`	1
`x || i && j - 3`	`x || (i && (j - 3))`	0
`i < j && x < y`	`(i < j) && (x < y)`	0
`i < j || x < y`	`(i < j) || (x < y)`	1
`'A' <= c && c <= 'Z'`	`('A' <= c) && (c <= 'Z')`	1
`c - 1 == 'A' || c + 1 == 'Z'`	`((c - 1) == 'A') || ((c + 1) == 'Z')`	1

The usual arithmetic conversions occur in expressions that are the operands of logical operators. Note that many of the expressions in the table are of mixed type. Whenever this occurs, certain values are promoted to match the highest type present in the expression.

Short-circuit Evaluation

In the evaluation of expressions that are the operands of && and ||, the evaluation process stops as soon as the outcome *true* or *false* is known. This is called *short-circuit* evaluation. It is an important property of these operators. Suppose that *expr1* and *expr2* are expressions and that *expr1* has value zero. In the evaluation of the logical expression

expr1 && *expr2*

the evaluation of *expr2* will not occur because the value of the logical expression as a whole is already determined to be 0. Similarly, if *expr1* has nonzero value, then in the evaluation of

expr1 || *expr2*

the evaluation of *expr2* will not occur because the value of the logical expression as a whole is already determined to be 1.

Here is a simple example of how short-circuit evaluation might be used. Suppose that we want to process no more than three characters.

```
int   cnt = 0;
while (++cnt <= 3 && (c = getchar()) != EOF) {
   .....      /* do something */
```

When the expression `++cnt <= 3` is *false*, the next character will not be read.

4.5 The Compound Statement

A compound statement is a series of declarations and statements surrounded by braces.

compound_statement ::= { { *declaration* }$_{0+}$ { *statement* }$_{0+}$ }

The chief use of the compound statement is to group statements into an executable unit. When declarations come at the beginning of a compound statement, the compound statement is also called a *block*. (See Section 5.10, "Scope Rules," on page 213, for a further discussion of blocks.) In C, wherever it is syntactically correct to place a

statement, it is also syntactically correct to place a compound statement: *A compound statement is itself a statement.* An example of a compound statement is

```
{
   a += b += c;
   printf("a = %d, b = %d, c = %d\n", a, b, c);
}
```

and another example is

```
{
   a = 1;
   {
      b = 2;
      c = 3;
   }
}
```

This example has an inner compound statement within an outer compound statement. Compound statements and blocks nest. Also, we follow an indentation style where statements within the brace are offset an additional three spaces. This lets us visually identify the grouping of statements within the braces. An important use of the compound statement is to achieve the desired flow of control in `if`, `if-else`, `while`, `for`, `do`, and `switch` statements.

4.6 The Expression and Empty Statement

The expression statement is an expression followed by a semicolon. The semicolon turns the expression into a statement that must be fully evaluated before going on to the next program statement.

expr_statement ::= { *expr* }$_{opt}$;

The empty statement is written as a single semicolon. It is useful where a statement is needed syntactically, but no action is required semantically. As we shall see, this is sometimes useful in constructs that affect the flow of control, such as `if-else` statements and `for` statements. The empty statement is a special case of the expression statement.

Some examples of expression statements are

```
a = b;                  /* an assignment statement */
a + b + c;              /* legal, but no useful work gets done */
;                       /* an empty statement */
printf("%d\n", a);      /* a function call */
```

All of these examples are expression statements. However, the assignment statement and function call are important enough to be given their own designations. In an expression statement, the expression is evaluated, including all side effects. Control then passes to the next statement. A typical case is `a = b;` where `a` is assigned the value of `b`.

4.7 The if and the if-else Statements

The general form of an `if` statement is

```
if (expr)
   statement
```

If *expr* is nonzero (*true*), then *statement* is executed; otherwise, *statement* is skipped and control passes to the next statement. In the example

```
if (grade >= 90)
   printf("Congratulations!\n");
printf("Your grade is %d.\n", grade);
```

a congratulatory message is printed only when the value of `grade` is greater than or equal to 90. The second `printf()` is always executed. The syntax of an `if` statement is given by

if_statement ::= `if` (*expr*) *statement*

Usually the expression in an `if` statement is a relational, equality, or logical expression, but as the syntax shows, an expression from any domain is permissible. Some other examples of `if` statements are

```
if (y != 0.0)
   x /= y;
if (c == ' ') {
   ++blank_cnt;
   printf("found another blank\n");
}
```

but not

```
if  b == a           /* parentheses missing */
   area = a * a;
```

Where appropriate, compound statements should be used to group a series of statements under the control of a single `if` expression. The following code consists of two `if` statements:

```
if (j < k)
   min = j;
if (j < k)
   printf("j is smaller than k\n");
```

The code can be written to be more efficient and more understandable by using a single `if` statement with a compound statement for its body.

```
if (j < k) {
   min = j;
   printf("j is smaller than k\n");
}
```

The `if-else` statement is closely related to the `if` statement. It has the general form

```
if (expr)
   statement1
else
   statement2
```

If *expr* is nonzero, then *statement1* is executed and *statement2* is skipped; if *expr* is zero, then *statement1* is skipped and *statement2* is executed. In both cases control then passes to the next statement. Consider the code

```
if (x < y)
   min = x;
else
   min = y;
```


If `x < y` is *true*, then `min` will be assigned the value of `x`, and if it is *false*, then `min` will be assigned the value of `y`. The syntax is given by

if_else_statement ::= `if (`*expr*`)` *statement*
`else` *statement*

An example is

```
if (c >= 'a' && c <= 'z')
   ++lc_cnt;
else {
   ++other_cnt;
   printf("%c is not a lowercase letter\n", c);
}
```

but not

```
if (i != j) {
   i += 1;
   j += 2;
} ;
else
   i -= j;      /* syntax error */
```

The syntax error occurs because the semicolon following the right brace creates an empty statement, and consequently the `else` has nowhere to attach.

Because an `if` statement is itself a statement, it can be used as the statement part of another `if` statement. Consider the code

```
if (a == 1)
   if (b == 2)
      printf("***\n");
```

This is of the form

```
if (a == 1)
   statement
```

where *statement* is the following `if` statement

```
if (b == 2)
   printf("***\n");
```

In a similar fashion, an `if-else` statement can be used as the statement part of another `if` statement. Consider, for example,

```
if (a == 1)
   if (b == 2)
      printf("***\n");
   else
      printf("###\n");
```

Now we are faced with a semantic difficulty. This code illustrates the *dangling else problem.* It is not clear from the syntax what the `else` part is associated with. Do not be fooled by the format of the code. As far as the machine is concerned, the following code is equivalent:

```
if (a == 1)
   if (b == 2)
      printf("***\n");
else
   printf("###\n");
```

The rule is that an `else` attaches to the nearest `if`. Thus, the code is correctly formatted as we first gave it. It has the form

```
if (a == 1)
   statement
```

where *statement* is the `if-else` statement

```
if (b == 2)
   printf("***\n");
else
   printf("###\n");
```


4.8 The while Statement

Repetition of action is one reason we rely on computers. When there are large amounts of data, it is very convenient to have control mechanisms that repeatedly execute specific statements. In C, the `while`, `for`, and `do` statements provide for repetitive action.

Although we have already used the `while` statement, or `while` loop, in many examples, we now want to explain precisely how this iterative mechanism works. The syntax is given by

while_statement ::= `while (`*expr*`)` *statement*

Some examples are

```
while (i++ < n)
   factorial *= i;

while ((c = getchar()) != EOF) {
   if (c >= 'a' && c <= 'z')
      ++lowercase_letter_cnt;
   ++total_cnt;
}
```

but not

```
while (++i < LIMIT) do {      /* syntax error: do is not allowed */
   j = 2 * i + 3;
   printf("%d\n", j);
}
```

Consider a construction of the form

```
while (expr)
   statement
next statement
```

First *expr* is evaluated. If it is nonzero (*true*), then *statement* is executed and control is passed back to the beginning of the while loop. The effect of this is that the body of the while loop, namely *statement*, is executed repeatedly until *expr* is zero (*false*). At that point, control passes to *next statement.* Thus, the effect of a while loop is that its body gets executed zero or more times.

It is possible to inadvertently specify an expression that never becomes zero, and unless other means of escaping the `while` loop are introduced, the program is stuck in an infinite loop. Care should be taken to avoid this difficulty. As an example, consider the code

```
printf("Input an integer:  ");
scanf("%d", &n);
while (--n)
   .....      /* do something */
```

The intent is for a positive integer to be entered and assigned to the `int` variable `n`. Then the body of the `while` loop is to be executed repeatedly until the expression `--n` is eventually zero. However, if a negative integer is inadvertently assigned to `n`, then the loop will be infinite. To guard against this possibility, it would be better to code instead

```
while (--n > 0)
   .....          /* do something */
```

It is sometimes appropriate for a `while` loop to contain only an empty statement. A typical example would be

```
while ((c = getchar()) == ' ')
   ;     /* empty statement */
```

This code will cause blank characters in the input stream to be skipped. We could have written this as

```
while ((c = getchar()) == ' ');
```

However, it is considered good programming style to place the semicolon on the next line by itself so that it is clearly visible as an empty statement.

Our next program illustrates the ideas that we have presented in this section. The program counts various kinds of characters.

In file cnt_char.c

```
/* Count blanks, digits, letters, newlines, and others. */

#include <stdio.h>

int main(void)
{
  int   blank_cnt = 0, c, digit_cnt = 0,
        letter_cnt = 0, nl_cnt = 0, other_cnt = 0;

  while ((c = getchar()) != EOF)     /* braces not necessary */
     if (c == ' ')
        ++blank_cnt;
     else if (c >= '0' && c <= '9')
        ++digit_cnt;
     else if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
        ++letter_cnt;
     else if (c == '\n')
        ++nl_cnt;
     else
        ++other_cnt;
  printf("%10s%10s%10s%10s%10s%10s\n\n",
    "blanks", "digits", "letters", "lines", "others", "total");
  printf("%10d%10d%10d%10d%10d%10d\n\n",
     blank_cnt, digit_cnt, letter_cnt, nl_cnt, other_cnt,
     blank_cnt + digit_cnt + letter_cnt + nl_cnt + other_cnt);
  return 0;
}
```

To execute this program, using its source file for data, we give the command

```
cnt_char   <   cnt_char.c
```

Here is the output that appears on the screen:

```
    blanks    digits   letters     lines    others     total
       197        31       348        27       180       783
```

Dissection of the *cnt_char* Program

- `while ((c = getchar()) != EOF)  /* braces not necessary */`

Braces are unnecessary because the long `if-else` construct that follows is a single statement that comprises the body of the `while` loop. This `while` statement repeatedly executes the compound `if-else` statement that follows it. Braces are unnecessary because the whole `if-else` construct is a single statement.

-
```
if (c == ' ')
   ++blank_cnt;
else if (c >= '0' && c <= '9')
   ++digit_cnt;
else if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
   ++letter_cnt;
else if (c == '\n')
   ++nl_cnt;
else
   ++other_cnt;
```

This statement tests a series of conditions. When a given condition evaluates to *true*, its corresponding expression statement is executed and the remaining tests are then skipped. For each character read into the variable `c`, exactly one of the variables used for counting will be incremented.

Note that logically we could have written the `if-else` as

```
if (c == ' ')
   ++blank_cnt;
else
   if (c >= '0' && c <= '9')
      ++digit_cnt;
   else
      if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
       ++letter_cnt;
      else
       .....
```

When the code is formatted this way, each of the `if-else` pairs lines up to show the logical structure of the code. However, this coding style is not recommended because long chains of `if-else` statements can cause the code to march too far to the right.

4.9 The for Statement

The `for` statement, like the `while` statement, is used to execute code iteratively. We can explain its action in terms of the `while` statement. The construction

```
for (expr1; expr2; expr3)
   statement
next statement
```

is semantically equivalent to

```
expr1;
while (expr2) {
   statement
   expr3;
}
next statement
```

provided that *expr2* is present, and provided that a `continue` statement is not in the body of the `for` loop. From our understanding of the `while` statement, we see that the semantics of the `for` statement are the following. First, *expr1* is evaluated. Typically, *expr1* is used to initialize the loop. Then *expr2* is evaluated. If it is nonzero (*true*), then *statement* is executed, *expr3* is evaluated, and control passes back to the beginning of the `for` loop, except that evaluation of *expr1* is skipped. Typically, *expr2* is a logical expression controlling the iteration. This process continues until *expr2* is zero (*false*), at which point control passes to *next statement*.

The syntax of a `for` statement is given by

for_statement `::= for (`*expr* `;` *expr* `;` *expr*`)` *statement*

Some examples are

```
for (i = 1; i <= n; ++i)
   factorial *= i;

for (j = 2; k % j == 0; ++j) {
   printf("%d is a divisor of %d\n", j, k);
   sum += j;
}
```

but not

```
for (i = 0, i < n, i += 3)      /* semicolons are needed */
   sum += i;
```

Any or all of the expressions in a `for` statement can be missing, but the two semicolons must remain. If *expr1* is missing, then no initialization step is performed as part of the `for` loop. The code

```
i = 1;
sum = 0;
for ( ; i <= 10; ++i)
   sum += i;
```

computes the sum of the integers from 1 to 10, and so does the code

```
i = 1;
sum = 0;
for ( ; i <= 10 ; )
   sum += i++;
```

The special rule for when *expr2* is missing is that the test is always *true*. Thus, the `for` loop in the code

```
i = 1;
sum = 0;
for ( ; ; ) {
   sum += i++;
   printf("%d\n", sum);
}
```

is an infinite loop.

A `for` statement can be used as the statement part of an `if`, `if-else`, `while`, or another `for` statement. For example, the construction

```
for ( ..... )
   for ( ..... )
      for ( ..... )
         statement
```

is a single `for` statement.

In many situations, program control can be accomplished by using either a `while` or a `for` statement. Which one gets used is often a matter of taste. One major advantage of a `for` loop is that control and indexing can both be kept right at the top. When loops are nested, this can facilitate the reading of the code. The program in the next section illustrates this.

4.10 An Example: Boolean Variables

Boolean algebra plays a major role in the design of computer circuits. In this algebra all variables have only the values zero or one. Transistors and memory technologies implement zero-one value schemes with currents, voltages, and magnetic orientations. Frequently, the circuit designer has a function in mind and needs to check whether, for all possible zero-one inputs, the output has the desired behavior.

We will use `int` variables `b1`, `b2`, ... , `b5` to represent five boolean variables. They will be allowed to take on only the values 0 and 1. A boolean function of these variables is one that returns only 0 or 1. A typical example of a boolean function is the majority function; it returns 1 if a majority of the variables have value 1, and 0 otherwise. We want to create a table of values for the functions

```
b1 || b3 || b5    and    b1 && b2 || b4 && b5
```

and the majority function. Recall that logical expressions always have the `int` value 0 or 1.

In file bool_vals.c

```
/* Print a table of values for some boolean functions. */

#include <stdio.h>

int main(void)
{
   int   b1, b2, b3, b4, b5;              /* boolean variables */
   int   cnt = 0;

   printf("\n%5s%5s%5s%5s%5s%5s%7s%7s%11s\n\n",    /* headings */
      "Cnt", "b1", "b2", "b3", "b4", "b5",
      "fct1", "fct2", "majority");
   for (b1 = 0; b1 <= 1; ++b1)
      for (b2 = 0; b2 <= 1; ++b2)
         for (b3 = 0; b3 <= 1; ++b3)
            for (b4 = 0; b4 <= 1; ++b4)
               for (b5 = 0; b5 <= 1; ++b5)
                  printf("%5d%5d%5d%5d%5d%6d%7d%9d\n",
                     ++cnt, b1, b2, b3, b4, b5,
                     b1 || b3 || b5, b1 && b2 || b4 && b5,
                     b1 + b2 + b3 + b4 + b5 >= 3);
   putchar('\n');
   return 0;
}
```

The program prints a table of values for all possible inputs and corresponding outputs. It illustrates a typical use of nested `for` loops. Here is some output of the program:

```
  Cnt   b1   b2   b3   b4   b5   fct1   fct2   majority
    1    0    0    0    0    0      0      0          0
    2    0    0    0    0    1      1      0          0
    3    0    0    0    1    0      0      0          0
    .....
```


4.11 The Comma Operator

The comma operator has the lowest precedence of all the operators in C. It is a binary operator with expressions as operands. The comma operator associates from left to right.

comma_expression ::= *expr* , *expr*

In a comma expression of the form

expr1 , *expr2*

expr1 is evaluated first, and then *expr2*. The comma expression as a whole has the value and type of its right operand. An example is

```
a = 0, b = 1
```

If `b` has been declared an `int`, then this comma expression has value 1 and type `int`.

The comma operator sometimes gets used in `for` statements. It allows multiple initializations and multiple processing of indices. For example, the code

```
for (sum = 0, i = 1; i <= n; ++i)
   sum += i;
```

can be used to compute the sum of the integers from 1 to n. Carrying this idea further, we can stuff the entire body of the `for` loop inside the `for` parentheses. The previous code could be rewritten as

```
for (sum = 0, i = 1; i <= n; sum += i, ++i)
   ;
```

but not as

```
for (sum = 0, i = 1; i <= n; ++i, sum += i)
   ;
```

In the comma expression

```
++i, sum += i
```

the expression `++i` is evaluated first, and this will cause `sum` to have a different value.

The comma operator should be used only in situations where it fits naturally. So far, we have given examples to illustrate its use, but none of the code is natural. In Section 10.2, "Linear Linked Lists," on page 449, where we discuss linked lists, we will have occasion to keep track of an index and a pointer at the same time. The comma operator can be used in a natural way to do this by writing

```
for (i = 0, p = head; p != NULL; ++i, p = p -> next)
   .....
```

Examples of comma expressions are given in the following table:

Declarations and initializations		
`int i, j, k = 3;` `double x = 3.3;`		
Expression	Equivalent expression	Value
`i = 1, j = 2, ++ k + 1`	`((i = 1), (j = 2)), ((++ k) + 1)`	5
`k != 1, ++ x * 2.0 + 1`	`(k != 1), (((++ x) * 2.0) + 1)`	9.6

Most commas in programs do not represent comma operators. For example, those commas used to separate expressions in argument lists of functions or used within initializer lists are not comma operators. If a comma operator is to be used in these places, the comma expression in which it occurs must be enclosed in parentheses.

4.12 The do Statement

The do statement can be considered a variant of the `while` statement. Instead of making its test at the top of the loop, it makes it at the bottom. Its syntax is given by

do_statement `::=` `do` *statement* `while (`*expr*`) ;`

An example is


```
i = 0;
sum = 0;
/* sum a series of integer inputs until 0 is input */
do {
   sum += i;
   scanf("%d", &i);
} while (i > 0);
```

Consider a construction of the form

```
do
   statement
while (expr);
next statement
```

First *statement* is executed and *expr* is evaluated. If the value of *expr* is nonzero (*true*), then control passes back to the beginning of the do statement and the process repeats itself. When *expr* is zero (*false*), then control passes to *next statement*.

As an example, suppose that we want to read in a positive integer and that we want to insist that the integer be positive. The following code will do the job:

```
do {
   printf("Input a positive integer: ");
   scanf("%d", &n);
   if (error = (n <= 0))
      printf("\nERROR: Do it again!\n\n");
} while (error);
```

If a nonpositive integer is entered, the user will be notified with a request for a positive integer. Control will exit the loop only after a positive integer has been entered.

Because in C, only a small percentage of loops tend to be do loops, it is considered good programming style to use braces even when they are not needed. The braces in the construct

```
do {
   a single statement
} while (.....);
```

make it easier for the reader to realize that you have written a do statement rather than a `while` statement followed by an empty statement.

Now that we have discussed in detail the `if` statement, the `if-else` statement, and the various looping statements, we want to mention the following tip that applies to all of their control expressions. It is good programming style to use a relational expression, when appropriate, rather than an equality expression. In many cases this will result in more robust code. For expressions of type `float` or `double`, an equality test

can be beyond the accuracy of the machine. Here is an example, which on most machines goes into an infinite loop:

In file loop.c

```
/* A test that fails. */

#include <stdio.h>

int main(void)
{
   int      cnt = 0;
   double   sum = 0.0, x;

   for (x = 0.0; x != 9.9; x += 0.1) {   /* trouble! */
      sum += x;
      printf("cnt = %5d\n", ++cnt);
   }
   printf("sum = %f\n", sum);
   return 0;
}
```

4.13 An Example: Fibonacci Numbers

The sequence of Fibonacci numbers is defined recursively by

$$f_0 = 0, \qquad f_1 = 1, \qquad f_{i+1} = f_i + f_{i-1} \quad \text{for} \quad i = 1, 2, \ldots$$

Except for f_0 and f_1, every element in the sequence is the sum of the previous two elements. It is easy to write down the first few elements of the sequence.

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, .....
```

Fibonacci numbers have lots of uses and many interesting properties. One of the properties has to do with the Fibonacci quotients defined by

$$q_i = \{f_i\} \,/\, \{f_{i-1}\} \qquad \text{for} \quad i = 2, 3, \ldots$$

It can be shown that the sequence of quotients converges to the golden mean, which is the real number (1 + sqrt 5) / 2.

We want to write a program that prints Fibonacci numbers and quotients. If `f1` contains the value of the current Fibonacci number and `f0` contains the value of the previous Fibonacci number, then we can do the following:

Fibonacci Program

1. Save the value of `f1` (the current Fibonacci number) in a temporary.
2. Add `f0` and `f1`, and store the value in `f1`, the new Fibonacci number.
3. Store the value of the temporary in `f0` so that `f0` will contain the previous Fibonacci number.
4. Print, and then repeat this process.

Because the Fibonacci numbers grow large very quickly, we are not able to compute many of them. Here is the program:

In file fibonacci.c

```
/* Print Fibonacci numbers and quotients. */

#include <stdio.h>

#define   LIMIT   46

int main(void)
{
   long   f0 = 0, f1 = 1, n, temp;

   printf("%7s%19s%29s\n%7s%19s%29s\n%7s%19s%29s\n",
          /* headings*/
      "  ", "Fibonacci", "Fibonacci",
      " n", "   number", " quotient",
      "__", "_________", "_________");
   printf("%7d%19d\n%7d%19d\n", 0, 0, 1, 1);  /* 1st 2 cases */
   for (n = 2; n <= LIMIT; ++n) {
      temp = f1;
      f1 += f0;
      f0 = temp;
      printf("%7ld%19ld%29.16f\n", n, f1, (double) f1 / f0);
   }
   return 0;
}
```

Here is some of the output of the program:

```
 n          Fibonacci number          Fibonacci quotient
--          ----------------          ------------------
 0                          0
 1                          1
 2                          1    1.0000000000000000
 3                          2    2.0000000000000000
 4                          3    1.5000000000000000
 5                          5    1.6666666666666667
 6                          8    1.6000000000000001
 7                         13    1.6250000000000000
.....
23                      28657    1.6180339901755971
24                      46368    1.6180339882053250
25                      75025    1.6180339889579021
.....
44                  701408733    1.6180339887498949
45                 1134903170    1.6180339887498949
46                 1836311903    1.6180339887498949
```

Dissection of the *fibonacci* Program

```
#define   LIMIT   46
```

When `for` loops are used, they frequently repeat an action up to some limiting value. If a symbolic constant such as `LIMIT` is used, then it serves as its own comment, and its value can be changed readily.

```
long   f0 = 0, f1 = 1, n, temp;
```

The variables `f0`, `f1`, and `temp` have been declared to be of type `long` so that the program will work properly on machines having 2-byte words, as well as on machines having 4-byte words. Note that some of the Fibonacci numbers printed are too large to be stored in a 2-byte `int`. There is no need for `n` to be of type `long`; it could just as well be of type `int`.


```
for (n = 2; n <= LIMIT; ++n) {
   temp = f1;
   f1 += f0;
   f0 = temp;
   .....
```

First `n` is initialized to 2. Then a test is made to see if `n` is less than or equal to `LIMIT`. If it is, then the body of the `for` loop is executed, `n` is incremented, and control is passed back to the top of the `for` loop. The test, the execution of the body, and the incrementing of `n` all get done repeatedly until `n` has a value greater than `LIMIT`. Each time through the loop, an appropriate Fibonacci number is computed and printed. The body of the `for` loop is the compound statement surrounded by braces. The use of the variable `temp` in the body is essential. Suppose that instead we had written

```
for (n = 2; n <= LIMIT; ++n) {   /* wrong code */
   f1 += f0;
   f0 = f1;
   .....
```

Then each time through the loop `f0` would not contain the previous Fibonacci number.

```
printf("%7ld%19ld%29.16f\n", n, f1, (double) f1 / f0);
```

Because variables of type `long` are being printed, the modifier `l` is being used with the conversion character `d`. Note that the field widths specified here match those in the `printf()` statements used at the beginning of the program. The field widths control the spacing between columns in the output. There is no magic in choosing them—just whatever looks good. Because a cast is a unary operator, it is of higher precedence than division. Thus, the expression

```
(double) f1 / f0      is equivalent to      ((double) f1) / f0
```

Because the operands of the division operator are of mixed type, the value of `f0` gets promoted to a `double` before division is performed.

4.14 The goto Statement

The goto statement is considered a harmful construct in most accounts of modern programming methodology. It causes an unconditional jump to a labeled statement somewhere in the current function. Thus, it can undermine all the useful structure provided by other flow of control mechanisms (for, while, do, if, switch).

Because a goto jumps to a labeled statement, we need to discuss this latter construct first. The syntax of a labeled statement is given by

labeled_statement ::= *label* : *statement* *label* ::= *identifier*

Some examples of labeled statements are

```
bye: exit(1);
L444: a = b + c;
bug1: bug2: bug3: printf("bug found\n");      /* multiple labels */
```

but not

```
333: a = b + c;      /* 333 is not an identifier */
```

The scope of a label is within the function in which it occurs. Label identifiers have their own name space. This means that the same identifier can be used both for a label and a variable. This practice, however, is considered bad programming style and should be avoided.

Control can be unconditionally transferred to a labeled statement by executing a goto statement of the form

```
goto label;
```

An example would be

```
goto error;

.....
error: {
   printf("An error has occurred - bye!\n");
   exit(1);
}
```


Both the goto statement and its corresponding labeled statement must be in the body of the same function. Here is a more specific piece of code that uses a goto:

```
while (scanf("%lf", &x) == 1) {
   if (x < 0.0)
      goto negative_alert;
   printf("%f  %f\n", sqrt(x), sqrt(2 * x));
}

negative_alert: printf("Negative value encountered!\n");
```

Note that this example could have been rewritten in a number of ways without using a goto.

In general, the goto should be avoided. It is a primitive method of altering flow of control, which, in a richly structured language, is unnecessary. Labeled statements and goto's are the hallmark of incremental patchwork program design. A programmer who modifies a program by adding gotos to additional code fragments soon makes the program incomprehensible.

When should a goto be used? A simple answer is "not at all." Indeed, one cannot go wrong by following this advice. In some rare instances, however, which should be carefully documented, a goto can make the program significantly more efficient. In other cases, it can simplify flow of control. This may occur when a special value is tested for in a deeply nested inner loop and, when this value is found, the program control needs to jump to the outermost level of the function.

4.15 The break and continue Statements

Two special statements,

```
break;   and   continue;
```

interrupt the normal flow of control. The break statement causes an exit from the innermost enclosing loop or switch statement. In the following example, a test for a negative argument is made, and if the test is *true*, then a break statement is used to pass control to the statement immediately following the loop.

```
while (1) {
   scanf("%lf", &x);
   if (x < 0.0)
      break;                    /* exit loop if x is negative */
   printf("%f\n", sqrt(x));
}

/* break jumps to here */
```

This is a typical use of `break`. What would otherwise be an infinite loop is made to terminate upon a given condition tested by the `if` expression.

The `continue` statement causes the current iteration of a loop to stop and causes the next iteration of the loop to begin immediately. The following code processes all characters except digits:

```
for (i = 0; i < TOTAL; ++i) {
   c = getchar();
   if (c >= '0' && c <= '9')
      continue;
   .....          /* process other characters */
/* continue transfers control to here to begin next iteration */
}
```

The `continue` statement may occur only inside `for`, `while`, and `do` loops. As the example shows, `continue` transfers control to the end of the current iteration, whereas `break` terminates the loop. With a `continue` statement, a `for` loop of the form

```
for (expr1; expr2; expr3) {
   .....
      continue;
   .....
}
```

is equivalent to

```
expr1;
while (expr2) {
   .....
      goto next;
   .....
next:
    expr3;
}
```

which is different from


```
expr1;
while (expr2) {
   .....
      continue;
   .....
   expr3;
}
```

(See exercise 29, on page 193, for a convenient way to test this.)

4.16 The switch Statement

The `switch` is a multiway conditional statement generalizing the `if-else` statement. Let us first look at the syntax for a `switch` statement.

switch_statement ::= `switch (` *integral_expression* `)`
  {*case_statement* | *default_statement* | *switch_block*}$_1$
case_statement ::= { `case` *constant_integral_expression* `:` }$_{1+}$
statement default_statement ::= `default :` *statement*
switch_block ::= {{*declaration_list*}$_{opt}$ {*case_group*}$_{0+}$ {*default_group*}$_{opt}$}
case_group ::= { `case` *constant_integral_expression* `:` }$_{1+}$ { *statement* }$_{1+}$
default_group ::= `default :` {*statement*}$_{1+}$

The following is a typical example of a `switch` statement:

```
switch (c) {
case 'a':
   ++a_cnt;
   break;
case 'b':
case 'B':
   ++b_cnt;
   break;
default:
   ++other_cnt;
}
```

Notice that the body of the `switch` statement in the example is a compound statement. This will be so in all but the most degenerate situations. The controlling expression in the parentheses following the keyword `switch` must be of integral type. In the example, it is just the `int` variable `c`. The usual automatic conversions are performed on the controlling expression. After the expression is evaluated, control jumps to the appropriate

`case` label. The constant integral expressions following the `case` labels must all be unique. Typically, the last statement before the next `case` or `default` label is a `break` statement. If there is no `break` statement, then execution "falls through" to the next statement in the succeeding case. Missing `break` statements are a frequent cause of error in `switch` statements. There may be at most one `default` label in a `switch`. Typically, it occurs last, although it can occur anywhere. The keywords `case` and `default` cannot occur outside of a `switch`.

The effect of a switch

1 Evaluate the `switch` expression.

2 Go to the `case` label having a constant value that matches the value of the expression found in step 1, or, if a match is not found, go to the `default` label, or, if there is no `default` label, terminate the `switch`.

3 Terminate the `switch` when a `break` statement is encountered, or terminate the `switch` by "falling off the end."

Let us review the various kinds of jump statements available to us. These include the `goto`, `break`, `continue`, and `return` statements. The `goto` is unrestricted in its use and should be avoided as a dangerous construct. The `break` may be used in loops and is important to the proper structuring of the `switch` statement. The `continue` is constrained to use within loops and is often unnecessary. The `return` statement must be used in functions that return values. It will be discussed in the next chapter.

4.17 The Conditional Operator

The conditional operator `?:` is unusual in that it is a ternary operator. It takes as operands three expressions.

conditional_expression `::=` *expr* `?` *expr* `:` *expr*

In a construct such as

expr1 `?` *expr2* `:` *expr3*

expr1 is evaluated first. If it is nonzero (*true*), then *expr2* is evaluated, and that is the value of the conditional expression as a whole. If *expr1* is zero (*false*), then *expr3* is evaluated, and that is the value of the conditional expression as a whole. Thus, a conditional expression can be used to do the work of an `if-else` statement. Consider, for example, the code


```
if (y < z)
   x = y;
else
   x = z;
```

The effect of the code is to assign to `x` the minimum of `y` and `z`. This also can be accomplished by writing

```
x = (y < z) ? y : z;
```

Because the precedence of the conditional operator is just above assignment, parentheses are not necessary. However, parentheses often are used to make clear what is being tested for.

The type of the conditional expression

expr1 `?` *expr2* `:` *expr3*

is determined by both *expr2* and *expr3*. If they are of different types, then the usual conversion rules are applied. Note carefully that the type of the conditional expression does not depend on which of the two expressions *expr2* or *expr3* is evaluated. The conditional operator `?:` has precedence just above the assignment operators, and it associates from right to left.

Declarations and initializations

```
char    a = 'a', b = 'b';      /* a has decimal value 97 */
int     i = 1, j = 2;
double  x = 7.07;
```

Expression	Equivalent expression	Value	Type
`i == j ? a - 1 : b + 1`	`(i == j) ? (a - 1) : (b + 1)`	99	`int`
`j % 3 == 0 ? i + 4 : x`	`((j % 3) == 0) ? (i + 4) : x`	7.07	`double`
`j % 3 ? i + 4 : x`	`(j % 3) ? (i + 4) : x`	5.0	`double`

Summary

1 Relational, equality, and logical expressions have the `int` value 0 or 1.

2 The negation operator `!` is unary. A negation expression such as `!a` has the `int` value 0 or 1. Remember: `!!a` and `a` need not have the same value.

3 A chief use of relational, equality, and logical expressions is to test data to affect flow of control.

4 Automatic type conversions can occur when two expressions are compared that are the operands of a relational, equality, or logical operator.

5 The grouping construct `{ ····· }` is a compound statement. It allows enclosed statements to be treated as a single unit.

6 An `if` statement provides a way of choosing whether or not to execute a statement.

7 The `else` part of an `if-else` statement associates with the nearest available `if`. This resolves the "dangling else" problem.

8 The `while`, `for`, and `do` statements provide for the iterative execution of code. The body of a `do` statement executes at least once.

9 To prevent an unwanted infinite loop, the programmer must make sure that the expression controlling the loop eventually becomes zero. The miscoding of controlling expressions is a common programming error.

10 The programmer often has to choose between the use of a `while` or a `for` statement. In situations where clarity dictates that both the control and the indexing be kept visible at the top of the loop, the `for` statement is the natural choice.

11 The comma operator is occasionally useful in `for` statements. Of all the operators in C, it has the lowest priority.

12 The four statement types

```
goto        break        continue        return
```

cause an unconditional transfer of flow of control. Their use should be minimized.

13 `goto`'s are considered harmful to good programming. Avoid them.

14 The `switch` statement provides a multiway conditional branch. It is useful when dealing with a large number of special cases.

Exercises

1 Give equivalent logical expressions of the following without negation:

```
!(a > b)
!(a <= b && c <= d)
!(a + 1 == b + 1)
!(a < 1 || b < 2 && c < 3)
```

2 Complete the following table:

Declarations and initializations		
`int a = 1, b = 2, c = 3;` `double x = 1.0;`		
Expression	**Equivalent expression**	**Value**
`a > b && c < d`		
`a < ! b || ! ! a`		
`a + b < ! c + c`		
`a - x || b * c && b / a`		

3 Write a program that reads characters from the standard input file until `EOF` is encountered. Use the variables `digit_cnt` and `other_cnt` to count the number of digits and the number of other characters, respectively.

4 Write a program that counts the number of times the first three letters of the alphabet (a, A, b, B, c, C) occur in a file. Do not distinguish between lower- and uppercase letters.

5 Write a program that contains the loop

```
while (scanf("%lf", &salary) == 1) {
   .....
}
```

Within the body of the loop, compute a 17% federal withholding tax and a 3% state withholding tax, and print these values along with the corresponding salary. Accumulate the sums of all salaries and taxes printed. Print these sums after the program exits the `while` loop.

6 What gets printed?

```
char   c = 'A';
int    i = 5, j = 10;

printf("%d %d %d\n", !c, !!c, !!!c);
printf("%d %d %d\n", - ! i, ! - i, ! - i - ! j);
printf("%d %d %d\n", ! (6 * j + i - c), ! i-5, ! j - 10);
```

7 Explain the effect of the following code:

```
int   i;
.....
while (i = 2) {
   printf("Some even numbers: %d %d %d\n", i, i + 2, i + 4);
   i = 0;
}
```

Contrast this code with the following:

```
int   i;
.....
if (i = 2)
   printf("Some even numbers: %d %d %d\n", i, i + 2, i + 4);
```

Both pieces of code are logically wrong. The run-time effect of one of them is so striking that the error is easy to spot, whereas the other piece of wrong code has a subtle effect that is much harder to spot. Explain.

8 What gets printed?

```
char     c = 'A';
double   x = 1e+33, y = 0.001;

printf("%d %d %d\n", c == 'a', c == 'b', c != 'c');
printf("%d\n", c == 'A' && c <= 'B' || 'C');
printf("%d\n", 1 != !!c == !!!c);
printf("%d\n", x + y > x - y);
```

9 What gets printed? Explain.

```
int   i = 7, j = 7;

if (i == 1)
   if (j == 2)
      printf("%d\n", i = i + j);
else
   printf("%d\n", i = i - j);
printf("%d\n", i);
```

10 Run a test program with this piece of code in it to find out how your compiler reports this syntax error. Explain why.

```
while (++i < LIMIT) do {     /* syntax error */
   j = 2 * i + 3;
   printf("j = %d\n", j);
}

   /* Many other languages require "do", but not C. */
```

11 Can the following code ever lead to an infinite loop? Explain. (Assume that the values of `i` and `j` are not changed in the body of the loop.)

```
printf("Input two integers: ");
scanf("%d%d", &i, &j);
while (i * j < 0 && ++i != 7 && j++ != 9) {
   .....     /* do something */
}
```

12 In Section 4.8, "The `while` Statement," on page 163, we said that if `n` has a negative value, then

```
while (--n)
   .....          /* do something */
```

is an infinite loop. Actually, this is system dependent. Can you explain why?

13 Write a program that reads in an integer value for `n` and then sums the integers from `n` to `2 * n` if `n` is nonnegative, or from `2 * n` to `n` if `n` is negative. Write the code in two versions, one using only `for` loops and the other using only `while` loops.

14 The function `putchar()` returns the int value of the character that it writes. What does the following code print?

```
for (putchar('1'); putchar('2'); putchar('3'))
   putchar('4');
```

15 For the C language, answer the following true-false questions:

Every statement ends in a semicolon.
Every statement contains at least one semicolon.
Every statement contains at most one semicolon.
There exists a statement with precisely 33 semicolons.
There exists a statement made up of 35 characters that contains 33 semicolons.

16 In Section 4.10, "An Example: Boolean Variables," on page 170, we presented a program that prints a table of values for some boolean functions. Execute the program and examine its output. For the 32 different inputs, exactly half of them (16 in number) have majority value 1. Write a program that prints a table of values for the majority function for, say, 7 boolean variables. Of the 128 different inputs, how many have majority value 1? State the general case as a theorem and try to give a proof. (Your machine can help you find theorems by checking special cases, but in general it cannot give a proof.)

17 Write three versions of a program that computes the sum of the first *n* even integers and the sum of the first *n* odd integers. The value for *n* should be entered interactively. In the first version of your program, use the code

```
for (cnt = 0, i = 1, j = 2; cnt < n; ++cnt, i += 2, j += 2)
   odd_sum += i, even_sum += j;
```


Note the prolific use of the comma operator. The code is not very good, but it will give you confidence that the comma operator works as advertised. In the second version, use one or more `for` statements but no comma operators. In the third version, use only `while` statements.

18 Choose one version of the program that you wrote in exercise 16, on page 188, and incorporate into it the following piece of code:

```
do {
   printf("Input a positive integer: ");
   scanf("%d", &n);
   if (error = (n <= 0))
      printf("\nERROR: Do it again!\n\n");
} while (error);
```

Then write another version of the program that uses a `while` statement instead of a `do` statement to accomplish the same effect.

19 Until interrupted, the following code prints `TRUE FOREVER` on the screen repeatedly. (In UNIX, type a control-c to effect an interrupt.)

```
while (1)
   printf(" TRUE FOREVER ");
```

Write a simple program that accomplishes the same thing, but use a `for` statement instead of a `while` statement. The body of the `for` statement should contain just the empty statement ";".

20 The following code is meant to give you practice with short-circuit evaluation:

```
int   a = 0, b = 0, x;

x = 0 && (a = b = 777);
printf("%d %d %d\n", a, b, x);
x = 777 || (a = ++b);
printf("%d %d %d\n", a, b, x);
```

What gets printed? First, write down your answers. Then write a test program to check them.

21 The semantics of logical expressions imply that order of evaluation is critical in some computations. Which of the following two alternate expressions is most likely to be the correct one? Explain.

```
if ((x != 0.0) && ((z - x) / x * x < 2.0))          .....
(b)     if (((z - x) / x * x < 2.0) && (x != 0.0))          .....
```

22 Suppose that we have three statements called *st1*, *st2*, and *st3*. We wish to write an `if-else` statement that will test the value of an `int` variable `i` and execute different combinations of the statements accordingly. The combinations are given in the following table:

i	execute	i	execute	i	execute
1	st1	1	st2	1	st1, st2
2	st2	2	st1, st3	2	st1, st2
3	st3	3	st1	3	st2, st3

Write programs that read in values for `i` interactively. Use appropriate `printf()` statements to check that the flow of control mimics the action described in the tables. For example, statements such as

```
if (i == 1)
   printf("statement_1 executed\n");
```

can be used to show that your programs run properly.

23 A polynomial in x of at most degree 2 is given by

$$ax^2 + bx + c$$

Its *discriminant* is defined to be

$$b^2 - 4ac$$

We are interested in the square root of the discriminant. If the discriminant is non-negative, then

$$\sqrt{b^2 - 4ac}$$

has its usual interpretation, but if the discriminant is negative, then

$\sqrt{b^2 - 4ac}$ means $i\sqrt{-(b^2 + 4ac)}$

where $i = \sqrt{-1}$ or equivalently, $i^2 = -1$.

Write a program that reads in values for a, b, and c and prints the value of the square root of the discriminant. For example, if the values 1, 2, and 3 are read in, then `i*2.828427` should be printed.

24 Write a program that repeatedly reads in values for a, b, and c and finds the roots of the polynomial

$$ax^2 + bx + c$$

Recall that the roots are real or complex numbers that solve the equation

$$ax^2 + bx + c = 0$$

When both a = 0 and b = 0, we consider the case "extremely degenerate" and leave it at that. When a = 0 and b != 0, we consider the case "degenerate." In this case the equation reduces to

$$bx + c = 0$$

and has one root given by $x = -c/b$.

When $a \neq 0$ (the general case), the roots are given by

$$root_1 = \frac{1}{2a}(-b + \sqrt{b^2 - 4ac}) \qquad root_2 = \frac{1}{2a}(-b - \sqrt{b^2 - 4ac})$$

The expression under the square root sign is the *discriminant*. If the discriminant is positive, then two real roots exist. If the discriminant is zero, then the two roots are real and equal. In this case we say that the polynomial (or the associated equation) has *multiple real roots*. If the discriminant is negative, then the roots are complex. For each set of values for a, b, and c, your program should print the computed root(s) along with one of the following messages:

```
degenerate      two real roots extremely
degenerate      two complex roots multiple real roots
```

For example, if the values 1, 2, and 3 are read in for a, b, and c, respectively, then the following should be printed:

```
two complex roots: root1 = -1.000000 + i*1.414214
                   root2 = -1.000000 - i*1.414214
```

25 A *truth table* for a boolean function is a table consisting of all possible values for its variables and the corresponding values of the boolean function itself. In Section 4.10, "An Example: Boolean Variables," on page 170, we created a truth table for the majority function and two other functions. In that table we used 1 and 0 to represent *true* and *false*, respectively. Create separate truth tables for the following boolean functions:

```
(a) b1 || b2 || b3 || b4

(b) b1 && b2 && b3 && b4

(c) !(!b1 || b2) && (!b3 || b4)
```

Use the letters T and F in your truth tables to represent *true* and *false*, respectively. *Hint:* Use the `#define` mechanism to define a `BOOLEX`, and write your program to operate on an arbitrary `BOOLEX`.

26 Write a program to check the proper pairing of braces. Your program should have two variables: one to keep track of left braces, say `left_cnt`, and the other to keep track of right braces, say `right_cnt`. Both variables should be initialized to zero. Your program should read and print each character in the input file. The appropriate variable should be incremented each time a brace is encountered. If `right_cnt` ever exceeds the value of `left_cnt`, your program should insert the character pair `??` at that point in the output. After all the characters in the input file have been processed, the two variables `left_cnt` and `right_cnt` should have the same value. If not, and `left_cnt` is larger than `right_cnt`, then a message should be printed that includes the number of right braces missing as a series of that many `}`'s. For example,

```
ERROR: Missing right braces: }}}
```

Use the macros `getchar()` and `putchar()` for input/output. Test your program by processing some files containing your own C code.

27 Extend the program that you wrote in the previous exercise so that it deals with both braces and parentheses simultaneously.

28 In Section 4.8, "The `while` Statement," on page 165, we presented a program that counts blanks, digits, and letters. Modify the program so that lower- and uppercase letters are counted separately.

29 Rewrite the following two pieces of code to avoid using `break` or `continue`:

```
while (c = getchar()) {
   if (c == 'E')
      break;
   ++cnt;
   if (c >= '0' && c <= '9')
      ++digit_cnt;
}
```

```
i = -5;
n = 50;
while (i < n) {
   ++i;
   if (i == 0)
      continue;
   total += i;
   printf("i = %d and total = %d\n", i, total);
}
```

30 Show how a `while` statement can be rewritten as a `goto` statement and an `if` statement. Which is the better construct, and why?

31 Because it leads to "spaghetti code," the `goto` is seldom used in today's programming world. The following code illustrates how just a few `goto` statements can make the flow of control incoherent:

```
d = b * b - 4.0 * a * c;
if (d == 0.0)
    goto L1;
else if (d > 0.0) {
   if (a != 0.0) {
      r1 = (-b + sqrt(d)) / (2.0 * a);
      r2 = (-b - sqrt(d)) / (2.0 * a);
      goto L4;
   }
   else
      goto L3;
}
else
   goto L2;
L1:
if (a != 0.0)
   r1 = r2 = -b / (2.0 * a);
else
   goto L3;
goto L4;
L2:
if (a != 0.0) {
   printf("imaginary roots\n");
   goto L4;
}
L3:    printf("degenerate case\n");
L4:    .....
```

Note how the programmer kept adding different cases and had to repatch the code until the program logic became obscure. Rewrite this code without using `goto` statements.

32 Here is a simple way to test the effect of a `continue` statement in the body of a `for` loop. What gets printed?

```
for (putchar('1'); putchar('2'); putchar('3')) {
   putchar('4');
   continue;
   putchar('5');
}
```


33 The mathematical operation `min(x, y)` can be represented by the conditional expression

```
(x < y) ? x : y
```

In a similar fashion, using only conditional expressions, describe the mathematical operations

```
min(x, y, z)      and      max(x, y, z, w)
```

34 Which of the following two statements is legal? Study the syntax for a switch statement to answer this, then write a test program to see if your compiler complains.

```
switch (1);                  /*version 1 */
switch (1)  switch (1);      /* version 2 */
```

35 In ANSI C, labels have their own name space, whereas in traditional C, they do not. This means that in ANSI C, the same identifier can be used for a variable and a label, although it is not considered good programming practice to do so. Write a test program containing the following code and execute it on your ANSI C compiler:

```
int   L = -3;                          /* L is a variable */
if (L < 0)
   goto L;
printf("L = %d\n", L);
L: printf("Exiting label test!\n");    /* L is a label */
```

If a traditional C compiler is available to you, check to see that your program will not compile.

36 (Advanced) Let `a` be a positive real number, and let the sequence of real numbers x_i be given by

$$x_0 = 1, \quad x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right) \quad \text{for } i = 0, 1, 2, \ldots$$

It can be shown mathematically that

$$x_i \to \sqrt{a} \quad \text{as } i \to \infty$$

This algorithm is derived from the Newton-Raphson method in numerical analysis. Write a program that reads in the value of `a` interactively and uses this algorithm to compute the square root of `a`. As you will see, the algorithm is very efficient. (Nonetheless, it is not the algorithm used by the `sqrt()` function in the standard library.)

Declare `x0` and `x1` to be of type `double`, and initialize `x1` to 1. Inside a loop do the following:

```
x0 = x1;                          /* save the current value of x1 */
x1 = 0.5 * (x1 + a / x1);         /* compute a new value of x1 */
```

The body of the loop should be executed as long as `x0` is not equal to `x1`. Each time through the loop, print out the iteration count and the values of `x1` (converging to the square root of `a`) and `a - x1 * x1` (a check on accuracy).

37 (Advanced) Modify the program you wrote in the previous exercise, so that the square roots of 1, 2, ... , n are all computed, where n is a value that is entered interactively. Print the number, the square root of the number, and the number of iterations needed to compute it. (You can look in a numerical analysis text to discover why the Newton-Raphson algorithm is so efficient. It is said to be "quadratically convergent.")

38 (Advanced) The constant *e*, which is the base of the natural logarithms, is given to 41 significant figures by

```
e = 2.71828 18284 59045 23536 02874 71352 66249 77572
```

Define

$$x_n = \left(1 + \frac{1}{n}\right)^n \quad \text{for} \quad n = 1, 2, \ldots$$

It can be shown mathematically that

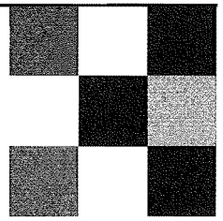
$$x_n \to e \quad \text{as} \quad n \to \infty$$

Investigate how to calculate *e* to arbitrary precision using this algorithm. You will find that the algorithm is computationally ineffective. (See exercise 36, on page 195.)

39 (Advanced) In addition to the algorithm given in the previous exercise, the value for *e* is also given by the infinite series

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots$$

The above algorithm is computationally effective. Use it to compute *e* to an arbitrary precision.



Chapter 5

Functions

The heart of effective problem solving is problem decomposition. Taking a problem and breaking it into small, manageable pieces is critical to writing large programs. In C, the function construct is used to implement this "top-down" method of programming.

A program consists of one or more files, with each file containing zero or more functions, one of them being a `main()` function. Functions are defined as individual objects that cannot be nested. Program execution begins with `main()`, which can call other functions, including library functions such as `printf()` and `sqrt()`. Functions operate with program variables, and which of these variables is available at a particular place in a function is determined by scope rules. In this chapter we discuss function definition, function declaration, scope rules, storage classes, and recursion.

5.1 Function Definition

The C code that describes what a function does is called the *function definition.* It must not be confused with the function declaration. A function definition has the following general form:

type function_name(*parameter list*) { *declarations statements* }

Everything before the first brace comprises the *header* of the function definition, and everything between the braces comprises the *body* of the function definition. The parameter list is a comma-separated list of declarations. An example of a function definition is

```
int factorial(int n)          /* header */
{                             /* body starts here */
   int   i, product = 1;
   for (i = 2; i <= n; ++i)
      product *= i;
   return product;
}
```

The first `int` tells the compiler that the value returned by the function will be converted, if necessary, to an `int`. The parameter list consists of the declaration `int n`. This tells the compiler that the function takes a single argument of type `int`. An expression such as `factorial(7)` causes the function to be invoked, or called. The effect is to execute the code that comprises the function definition, with `n` having the value 7. Thus, functions act as useful abbreviating schemes. Here is another example of a function definition:

```
void wrt_address(void)
{
   printf("%s\n%s\n%s\n%s\n%s\n\n",
      "            **********************",
      "            **   SANTA CLAUS    *",
      "            **   NORTH POLE     *",
      "            **   EARTH          *",
      "            **********************");
}
```

The first `void` tells the compiler that this function returns no value; the second `void` tells the compiler that this function takes no arguments. The expression

```
wrt_address()
```

causes the function to be invoked. For example, to call the function three times we can write

```
for (i = 0; i < 3; ++i)
   wrt_address();
```

A function definition starts with the type of the function. If no value is returned, then the type is `void`. If the type is something other than `void`, then the value returned by the function will be converted, if necessary, to this type. The name of the function is followed by a parenthesized list of parameter declarations. The parameters act as placeholders for values that are passed when the function is invoked. Sometimes, to emphasize their role as placeholders, these parameters are called the *formal parameters* of the function. The function body is a block, or compound statement, and it too may contain declarations. Some examples of function definitions are


```
void nothing(void) { }     /* this function does nothing */

double twice(double x)
{
   return (2.0 * x);
}

int all_add(int a, int b)
{
   int   c;
   .....
   return (a + b + c);
}
```

If a function definition does not specify the function type, then it is `int` by default. For example, the last function definition could be given by

```
all_add(int a, int b)
{
   .....
```

However, it is considered good programming practice to specify the function type explicitly. (See exercise 5, on page 236, for further discussion.)

Any variables declared in the body of a function are said to be "local" to that function. Other variables may be declared external to the function. These are called "global" variables. An example is

```
#include <stdio.h>

int   a = 33;     /* a is external and initialized to 33 */

int main(void)
{
   int   b = 77;              /* b is local to main() */

   printf("a = %d\n", a);     /* a is global to main() */
   printf("b = %d\n", b);
   return 0;
}
```

In traditional C, the function definition has a different syntax. The declarations of the variables in the parameter list occur after the parameter list itself and just before the first brace. An example is

```
void f(a, b, c, x, y)
int      a, b, c;
double   x, y;
{
   .....
```

The order in which the parameters are declared is immaterial. If there are no parameters, then a pair of empty parentheses is used. ANSI C compilers will accept this traditional syntax as well as the newer syntax. Thus, traditional code can still be compiled by an ANSI C compiler.

There are several important reasons to write programs as collections of many small functions. It is simpler to correctly write a small function to do one job. Both the writing and debugging are made easier. It is also easier to maintain or modify such a program. One can readily change just the set of functions that need to be rewritten, expecting the rest of the code to work correctly. Also, small functions tend to be self-documenting and highly readable. A useful heuristic for writing good programs is to write each function so that its code fits on a single page.

5.2 The return Statement

The `return` statement may or may not include an expression.

return_statement `::= return;` | `return` *expression* `;`

Some examples are

```
return;
return ++a;
return (a * b);
```

The expression being returned can be enclosed in parentheses, but this is not required.

When a `return` statement is encountered, execution of the function is terminated and control is passed back to the calling environment. If the `return` statement contains an expression, then the value of the expression is passed back to the calling environment as well. Moreover, this value will be converted, if necessary, to the type of the function as specified in the function definition.


```
float f(char a, char b, char c)
{
   int   i;
   .....
   return i;   /* value returned will be converted to a float */
}
```

There can be zero or more `return` statements in a function. If there is no `return` statement, then control is passed back to the calling environment when the closing brace of the body is encountered. This is called "falling off the end." The following function definition illustrates how two `return` statements might be used:

```
double absolute_value(double x)
{
   if (x >= 0.0)
      return x;
   else
      return -x;
}
```

Even though a function returns a value, a program does not need to use it.

```
while (.....) {
   getchar();          /* get a char, but do nothing with it */
   c = getchar();      /* c will be processed */
   .....
}
```

5.3 Function Prototypes

Functions should be declared before they are used. ANSI C provides for a new function declaration syntax called the *function prototype*. A function prototype tells the compiler the number and type of arguments that are to be passed to the function and the type of the value that is to be returned by the function. An example is

```
double   sqrt(double);
```

This tells the compiler that `sqrt()` is a function that takes a single argument of type `double` and returns a `double`. The general form of a function prototype is

type *function_name* (*parameter type list*) ;

The parameter type list is typically a comma-separated list of types. Identifiers are optional; they do not affect the prototype. For example, the function prototype

```
void f(char c, int i);        is equivalent to        void f(char, int);
```

The identifiers such as `c` and `i` that occur in parameter type lists in function prototypes are not used by the compiler. Their purpose is to provide documentation to the programmer and other readers of the code. The keyword `void` is used if a function takes no arguments. Also, the keyword `void` is used if no value is returned by the function. If a function takes a variable number of arguments, then the ellipses (`...`) are used. See, for example, the function prototype for `printf()` in the standard header file *stdio.h*.

Function prototypes allow the compiler to check the code more thoroughly. Also, values passed to functions are properly coerced, if possible. For example, if the function prototype for `sqrt()` has been specified, then the function call `sqrt(4)` will yield the correct value. Because the compiler knows that `sqrt()` takes a `double`, the `int` value 4 will be promoted to a `double` and the correct value will be returned. (See exercise 5, on page 236, for further discussion.)

In traditional C, parameter type lists are not allowed in function declarations. For example, the function declaration of `sqrt()` is given by

```
double   sqrt();        /* traditional C style */
```

Even though ANSI C compilers will accept this style, function prototypes are preferred. With this declaration, the function call `sqrt(4)` will not yield the correct value. (See exercise 5, on page 236.)

Function Prototypes in C++

In C++, function prototypes are required, and the use of `void` in the parameter type list in both function prototypes and function definitions is optional. Thus, for example, in C++

```
void f()        is equivalent to        void f(void)
```

Note carefully that this results in a conflict with traditional C. In traditional C, a function declaration such as

```
int f();
```

means that `f()` takes an unknown number of arguments. In traditional C, `void` is not a keyword. Thus, it cannot be used in a parameter list in a function declaration or function definition.

5.4 An Example: Creating a Table of Powers

In this section, we give an example of a program that is written using a number of functions. For simplicity, we will write all the functions one after another in one file. The purpose of the program is to print a table of powers.

```
#include <stdio.h>

#define   N   7

long   power(int, int);
void   prn_heading(void);
void   prn_tbl_of_powers(int);

int main(void)
{
   prn_heading();
   prn_tbl_of_powers(N);
   return 0;
}

void prn_heading(void)
{
   printf("\n::::: A TABLE OF POWERS :::::\n\n");
}

void prn_tbl_of_powers(int n)
{
   int   i, j;

   for (i = 1; i <= n; ++i) {
      for (j = 1; j <= n; ++j)
         if (j == 1)
            printf("%ld", power(i, j));
         else
            printf("%9ld", power(i, j));
      putchar('\n');
   }
}
```

```
long power(int m, int n)
{
   int    i;
   long   product = 1;

   for (i = 1; i <= n; ++i)
      product *= m;
   return product;
}
```

Here is the output of the program:

```
:::::  A TABLE OF POWERS  :::::

1        1        1        1        1        1        1
2        4        8       16       32       64      128
3        9       27       81      243      729     2187
.....
```

Note that the first column consists of integers raised to the first power, the second column consists of integers raised to the second power, and so forth. In our program we have put the function prototypes near the top of the file. This makes them visible throughout the rest of the file. We used the type `long` so that the program will produce the same output whether the machine has 2- or 4-byte words. Note that the function `power()` computes the quantity m^n, which is m raised to the nth power.

Our program illustrates in a very simple way the idea of top-down design. The programmer thinks of the tasks to be performed and codes each task as a function. If a particular task is complicated, then that task, in turn, can be subdivided into other tasks, each coded as a function. An additional benefit of this is that the program as a whole becomes more readable and self-documenting.

5.5 Function Declarations from the Compiler's Viewpoint

To the compiler, function declarations are generated in various ways: by function invocation, by function definition, and by explicit function declarations and function prototypes. If a function call, say `f(x)`, is encountered before any declaration, definition, or prototype for it occurs, then the compiler assumes a default declaration of the form


```
int   f();
```

Nothing is assumed about the parameter list for the function. Now suppose that the following function definition occurs first:

```
int f(x)          /* traditional C style */
double x;
{
   .....
```

This provides both declaration and definition to the compiler. Again, however, nothing is assumed about the parameter list. It is the programmer's responsibility to pass only a single argument of type `double`. A function call such as `f(1)` can be expected to fail because `1` is of type `int`, not `double`. Now suppose that we use, instead, an ANSI C style definition:

```
int f(double x)      /* ANSI C style */
{
   .....
```

The compiler now knows about the parameter list as well. In this case, a function call such as `f(1)` can be expected to work properly. When an `int` gets passed as an argument, it will be converted to a `double`.

A function prototype is a special case of a function declaration. A good programming style is to give either the function definition (ANSI C style) or the function prototype or both before a function is used. A major reason for including standard header files is because they contain function prototypes.

Limitations

Function definitions and prototypes have certain limitations. The function storage class specifier, if present, can be either `extern` or `static`, but not both; `auto` and `register` cannot be used. (See Section 5.11, "Storage Classes," on page 216.) The types "array of ..." and "function returning ..." cannot be returned by a function. However, a pointer representing an array or a function can be returned. (See Section 6.6, "Arrays as Function Arguments," on page 256.) The only storage class specifier that can occur in the parameter type list is `register`. Parameters cannot be initialized.

5.6 An Alternate Style for Function Definition Order

Suppose we want to write a program in a single file. If our program contains more than one function definition, we usually put `#includes` and `#defines` at the top of the file, other program elements such as templates of enumeration types (Section 7.5, "Enumeration Types," on page 345) and templates of structures and unions (Section 9.4, "Using Structures with Functions," on page 416) next, then a list of function prototypes, and finally the function definitions, starting with `main()`. Because function definitions also serve as function prototypes, an alternate style is to remove the list of function prototypes and to put the function definition of any function that gets called before the function definition of its caller. In particular, `main()` goes last in the file.

Let us illustrate this alternate style by modifying the *tbl_of_powers* program that we wrote in Section 5.4, "An Example: Creating a Table of Powers," on page 203. Here is another way of writing that program:

```
#include <stdio.h>

#define   N   7

void prn_heading(void)
{
   .....
}

long power(int m, int n)
{
   .....
}

void prn_tbl_of_powers(int n)
{
   .....
   printf("%ld", power(i, j));
   .....
}

int main(void)
{
   prn_heading();
   prn_tbl_of_powers(N);
   return 0;
}
```


Because `power()` is called by `prn_tbl_of_powers()`, the function definition for `power()` must come first. Similarly, the function definitions of the two functions called by `main()` must come before the function definition of `main()`. (See exercise 10, on page 238, for further discussion.)

Although we favor the top-down style that puts `main()` first, we will occasionally use this alternate style as well.

5.7 Function Invocation and Call-by-Value

A program is made up of one or more function definitions, with one of these being `main()`. Program execution always begins with `main()`. When program control encounters a function name, the function is called, or invoked. This means that program control passes to that function. After the function does its work, program control is passed back to the calling environment, which then continues with its work.

Functions are invoked by writing their name and an appropriate list of arguments within parentheses. Typically, these arguments match in number and type (or compatible type) the parameters in the parameter list in the function definition. The compiler enforces type compatibility when function prototypes are used. All arguments are passed "call-by-value." This means that each argument is evaluated, and its value is used locally in place of the corresponding formal parameter. Thus, if a variable is passed to a function, the stored value of that variable in the calling environment will not be changed. Here is an example that clearly illustrates the concept of "call-by-value":

```
#include <stdio.h>

int   compute_sum(int n);

int main(void)
{
   int   n = 3, sum;

   printf("%d\n", n);          /* 3 is printed */
   sum = compute_sum(n);
   printf("%d\n", n);          /* 3 is printed */
   printf("%d\n", sum);        /* 6 is printed */
   return 0;
}
```

```
int compute_sum(int n)        /* sum the integers from 1 to n */
{
   int   sum = 0;

   for ( ; n > 0; --n)        /* stored value of n is changed */
      sum += n;
   return sum;
}
```

Even though n is passed to compute_sum() and the value of n in the body of that function is changed, the value of n in the calling environment remains unchanged. It is the *value* of n that is being passed, not n itself.

The "call-by-value" mechanism is in contrast to that of "call-by-reference." In Section 6.3, "Call-by-Reference," on page 252, we will explain how to accomplish the effect of "call-by-reference." This is a way of passing addresses (references) of variables to a function that then allows the body of the function to make changes to the values of variables in the calling environment.

Function invocation means:

1 Each expression in the argument list is evaluated.

2 The value of the expression is converted, if necessary, to the type of the formal parameter, and that value is assigned to its corresponding formal parameter at the beginning of the body of the function.

3 The body of the function is executed.

4 If a return statement is executed, then control is passed back to the calling environment.

5 If the return statement includes an expression, then the value of the expression is converted, if necessary, to the type given by the type specifier of the function, and that value is passed back to the calling environment, too.

6 If the return statement does not include an expression, then no useful value is returned to the calling environment.

7 If no return statement is present, then control is passed back to the calling environment when the end of the body of the function is reached. No useful value is returned.

8 All arguments are passed "call-by-value."

5.8 Developing a Large Program

Typically, a large program is written in a separate directory as a collection of *.h* and *.c* files, with each *.c* file containing one or more function definitions. Each *.c* file can be recompiled as needed, saving time for both the programmer and the machine. (See Section 11.17, "The Use of make," on page 532 for further discussion.)

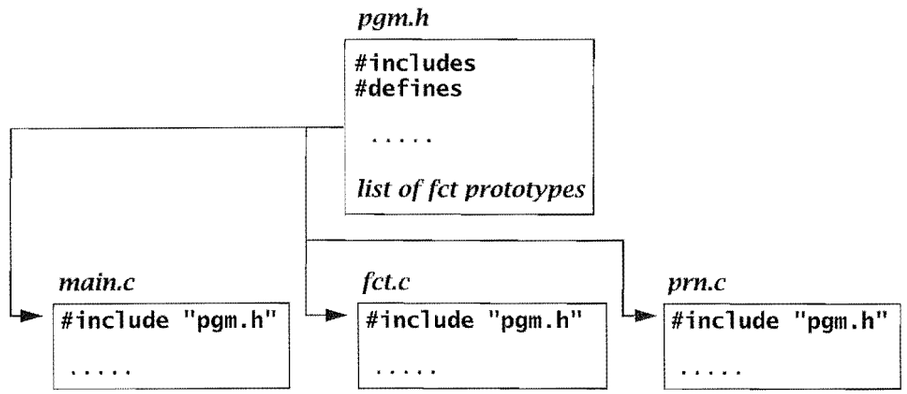
Let us suppose we are developing a large program called *pgm*. At the top of each of our *.c* files we put the line

```
#include "pgm.h"
```

When the preprocessor encounters this directive, it looks first in the current directory for the file *pgm.h*. If there is such a file, then it gets included. If not, then the preprocessor looks in other system-dependent places for the file. If the file *pgm.h* cannot be found, then the preprocessor issues an error message and compilation stops.

Our header file *pgm.h* may contain #includes, #defines, templates of enumeration types, templates of structure and union types, other programming constructs, and finally a list of function prototypes at the bottom. Thus, *pgm.h* contains program elements that are appropriate for our program as a whole. Because the header file *pgm.h* occurs at the top of each *.c* file, it acts as the "glue" that binds our program together.

Create a .h file that is included in all the .c files



Let us show a very simple example of how this works. We will write our program in a separate directory. It will consist of a *.h* file and three *.c* files. Typically, the name of the directory and the name of the program are the same. Here is our program:

In file pgm.h

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

#define   N   3

void   fct1(int k);
void   fct2(void);
void   wrt_info(char *);
```

In file main.c

```
#include "pgm.h"

int main(void)
{
   char   ans;
   int    i, n = N;

   printf("%s",
       "This program does not do very much.\n"
       "Do you want more information?  ");
   scanf(" %c", &ans);
   if (ans == 'y' || ans == 'Y')
      wrt_info("pgm");
   for (i = 0; i < n; ++i)
      fct1(i);
   printf("Bye!\n");
   return 0;
}
```

In file fct.c

```
#include "pgm.h"

void fct1(int n)
{
   int   i;

   printf("Hello from fct1()\n");
   for (i = 0; i < n; ++i)
      fct2();
}
```



```
void fct2(void)
{
   printf("Hello from fct2()\n");
}
```

In file wrt.c

```
#include "pgm.h"

void wrt_info(char *pgm_name) {
   printf("Usage:  %s\n\n", pgm_name);
   printf("%s\n",
    "This program illustrates how one can write a program\n"
    "in more than one file.  In this example, we have a\n"
    "single .h file that gets included at the top of our\n"
    "three .c files.  Thus, the .h file acts as the \"glue\"\n"
    "that binds the program together.\n"
    "\n"
    "Note that the functions fct1() and fct2() when called\n"
    "only say \"hello.\"  When writing a serious program, the\n"
    "programmer sometimes does this in a first working\n"
    "version of the code.\n"); }
```

Note that we used the type `char *` (pointer to `char`). (We will discuss pointers in Section 6.2, "Pointers," on page 248.) We compile the program with the command

```
cc  -o pgm  main.c  fct.c  prn.c
```

The compiler makes the executable file *pgm* along with three *.o* files that correspond to *.c* files. (In MS-DOS, they are *.obj* files.) The *.o* files are called *object* files. (For further discussion about these object files and how the compiler can use them, see Section 11.13, "The C Compiler," on page 522.)

For a more interesting example of a program written in separate files, see Section 7.6, "An Example: The Game of Paper, Rock, Scissors," on page 348. Other examples are available via *ftp* and the Internet. If you are connected to the Internet, try the following command:

```
ftp  bc.aw.com
```

After you have made a connection, you can change directory (*cd*) to *bc*, and then *cd* to *kelley_pohl*, and then look around. A large program is much easier to investigate if you have the source code. Then you can print out whatever is of interest and, with the help of a debugger, you can step through the program if you wish to do so. (See Section 11.19, "Other Useful Tools," on page 539.)

What Constitutes a Large Program?

For an individual, a large program might consist of just a few hundred lines of code. Which lines get counted? Usually all the lines in any *READ_ME* files (there should be at least one), the *.h* files, the *.c* files, and the *makefile.* (See Section 11.17, "The Use of make," on page 532.) In UNIX, the word count utility *wc* can be used to do this:

*wc READ_ME *.h *.c makefile*

In industry, programs are typically written by teams of programmers, and a large program might be considered anything over a hundred thousand lines.

The style of writing a program in its own directory as a collection of *.h* and *.c* files works well for any serious program, whether it is large or small, and all experienced programmers follow this style. To become proficient in the style, the programmer has to learn how to use *make* or some similar tool. (See Section 11.17, "The Use of make," on page 532.)

5.9 Using Assertions

The C system supplies the standard header file *assert.h*, which the programmer can include to obtain access to the `assert()` macro. (See Section 8.11, "The `assert()` Macro," on page 388, for further discussion.) Here, we want to show how the programmer can use assertions to facilitate the programming process. Consider the following code:

```
#include <assert.h>
#include <stdio.h>

int   f(int a, int b);
int   g(int c);
```



```
int main(void)
{
   int   a, b, c;
   .....
   scanf("%d%d", &a, &b);
   .....
   c = f(a, b);
   assert(c > 0);          /* an assertion */
   .....
}
```

If the expression passed as an argument to `assert()` is false, the system will print a message, and the program will be aborted. In this example, we are supposing that `f()` embodies an algorithm that is supposed to generate a positive integer, which gets returned. Perhaps someone else wrote the code for the function definition of `f()`, and you just want to be sure that the value returned by `f()` is positive. In the function definition for `f()`, it may be important that the arguments `a` and `b` satisfy certain conditions. Let us suppose that we expect `a` to have either 1 or -1 as its value and that we expect `b` to lie in the interval [7, 11]. We can use assertions to enforce these conditions.

```
int f(int a, int b)
{
   .....
   assert(a == 1 || a == -1);
   assert(b >= 7 && b <= 11);
   .....
}
```

Assertions are easy to write, add robustness to the code, and help other readers of the code understand its intent. The use of assertions is considered good programming methodology.

5.10 Scope Rules

The basic rule of scoping is that identifiers are accessible only within the block in which they are declared. They are unknown outside the boundaries of that block. This would be an easy rule to follow, except that programmers, for a variety of reasons, choose to use the same identifier in different declarations. We then have the question of which object the identifier refers to. Let us give a simple example of this state of affairs.

```
{
   int a = 2;              /* outer block a */
   printf("%d\n", a);      /* 2 is printed */
   {
      int a = 5;           /* inner block a */
      printf("%d\n", a);   /* 5 is printed */
   }                       /* back to the outer block */
   printf("%d\n", ++a);    /* 3 is printed */
}
```

An equivalent piece of code would be

```
{
   int a_outer = 2;
   printf("%d\n", a_outer);
   {
      int a_inner = 5;
      printf("%d\n", a_inner);
   }
   printf("%d\n", ++a_outer);
}
```

Each block introduces its own nomenclature. An outer block name is valid unless an inner block redefines it. If redefined, the outer block name is hidden, or masked, from the inner block. Inner blocks may be nested to arbitrary depths that are determined by system limitations. The following piece of code illustrates hidden variables in three nested blocks:

```
{
   int   a = 1, b = 2, c = 3;
   printf("%3d%3d%3d\n", a, b, c);          /*  1  2  3  */
   {
      int     b = 4;
      float   c = 5.0;
      printf("%3d%3d%5.1f\n", a, b, c);     /*  1  4  5.0  */
      a = b;
      {
       int   c;
       c = b;
       printf("%3d%3d%3d\n", a, b, c);      /*  4  4  4  */
      }
      printf("%3d%3d%5.1f\n", a, b, c);     /*  4  4  5.0  */
   }
   printf("%3d%3d%3d\n", a, b, c);          /*  4  2  3  */
}
```


The int variable a is declared in the outer block and is never redeclared. Therefore, it is available in both nested inner blocks. The variables b and c are redeclared in the first inner block, thus, hiding the outer block variables of the same name. Upon exiting this block, both b and c are again available as the outer block variables, with their values intact from the outer block initialization. The innermost block again redeclares c, so that both c (inner) and c (outer) are hidden by c (innermost).

Parallel and Nested Blocks

Two blocks can come one after another, in which case the second block has no knowledge of the variables declared in the first block. Such blocks, residing at the same level, are called *parallel blocks*. Functions are declared in parallel at the outermost level. The following code illustrates two parallel blocks nested in an outer block:

```
{
   int     a, b;
   .....
   {                          /* inner block 1 */
      float     b;
      .....                   /* int a is known, but not int b */
   }
.....
{                             /* inner block 2 */
   float     a;
   .....
                              /* int b is known, but not int a */
                              /* nothing in inner block 1 in known */
   }
.....
}
```

Parallel and nested blocks can be combined in arbitrarily complicated schemes. The chief reason for blocks is to allow memory for variables to be allocated where needed. If memory is scarce, block exit will release the storage allocated locally to that block, allowing the memory to be used for some other purpose. Also, blocks associate names in their neighborhood of use, making the code more readable. Functions can be viewed as named blocks with parameters and return statements allowed.

Using a Block for Debugging

One common use for a block is for debugging purposes. Imagine we are in a section of code where a variable, say v, is misbehaving. By inserting a block temporarily into the code, we can use local variables that do not interfere with the rest of the program.

```
{                                   /* debugging starts here */
   static int   cnt = 0;
   printf("*** debug: cnt = %d   v = %d\n", ++cnt, v);
}
```

The variable cnt is local to the block. It will not interfere with another variable of the same name in an outer block. Because its storage class is static, it retains its old value when the block is reentered. Here, it is being used to count the number of times the block is executed. (Perhaps we are inside a for loop.) We are assuming that the variable v has been declared in an outer block and is therefore known in this block. We are printing its value for debugging purposes. Later, after we have fixed our code, this block becomes extraneous, and we remove it.

5.11 Storage Classes

Every variable and function in C has two attributes: *type* and *storage class*. The four storage classes are automatic, external, register, and static, with corresponding keywords

```
auto          extern          register          static
```

By far the most common storage class for variables is automatic. However, the programmer needs to know about all the storage classes. They all have important uses.

The Storage Class auto

Variables declared within function bodies are automatic by default. Thus, automatic is the most common of the four storage classes. If a compound statement starts with variable declarations, then these variables can be acted on within the scope of the enclosing compound statement. A compound statement with declarations is called a *block* to distinguish it from one that does not begin with declarations.

Declarations of variables within blocks are implicitly of storage class automatic. The keyword auto can be used to explicitly specify the storage class. An example is

```
auto int     a, b, c;
auto float   f;
```

Because the storage class is automatic by default, the keyword auto is seldom used.

When a block is entered, the system allocates memory for the automatic variables. Within that block, these variables are defined and are considered "local" to the block. When the block is exited, the system releases the memory that was set aside for the automatic variables. Thus, the values of these variables are lost. If the block is reentered, the system once again allocates memory, but previous values are unknown. The body of a function definition constitutes a block if it contains declarations. If it does, then each invocation of the function sets up a new environment.

The Storage Class extern

One method of transmitting information across blocks and functions is to use external variables. When a variable is declared outside a function, storage is permanently assigned to it, and its storage class is extern. A declaration for an external variable can look just the same as a declaration for a variable that occurs inside a function or block. Such a variable is considered to be global to all functions declared after it, and upon exit from the block or function, the external variable remains in existence. The following program illustrates these ideas:

```
#include <stdio.h>

int   a = 1, b = 2, c = 3;              /* global variables */
int   f(void);                          /* function prototype */

int main(void)
{
   printf("%3d\n", f());                /* 12 is printed */
   printf("%3d%3d%3d\n", a, b, c);      /*  4  2  3 is printed */
   return 0;
}

int f(void)
{
   int   b, c;                          /* b and c are local */
                                        /* global b, c are masked */
   a = b = c = 4;
   return (a + b + c);
}
```

Note that we could have written

```
extern int   a = 1, b = 2, c = 3;    /* global variables */
```

This use of `extern` will cause some traditional C compilers to complain. In ANSI C, this use is allowed but not required. Variables defined outside a function have external storage class, even if the keyword `extern` is not used. Such variables cannot have automatic or register storage class. The keyword `static` can be used, but its use is special, as explained in Section 5.12, "Static External Variables," on page 221.

The keyword `extern` is used to tell the compiler to "look for it elsewhere, either in this file or in some other file." Let us rewrite the last program to illustrate a typical use of the keyword `extern`.

In file file1.c

```
#include <stdio.h>

int   a = 1, b = 2, c = 3;     /* external variables */
int   f(void);

int main(void)
{
   printf("%3d\n", f());
   printf("%3d%3d%3d\n", a, b, c);
   return 0;
}
```

In file file2.c

```
int f(void)
{
   extern int   a;              /* look for it elsewhere */
   int          b, c;

   a = b = c = 4;
   return (a + b + c); }
```

The two files can be compiled separately. The use of `extern` in the second file tells the compiler that the variable a will be defined elsewhere, either in this file or in some other. The ability to compile files separately is important when writing large programs.

External variables never disappear. Because they exist throughout the execution life of the program, they can be used to transmit values across functions. They may, however, be hidden if the identifier is redefined. Another way of conceiving of external variables is to think of them as being declared in a block that encompasses the whole program.

Information can be passed into a function two ways: by use of external variables and by use of the parameter mechanism. Although there are exceptions, the use of the parameter mechanism is the preferred method. It tends to improve the modularity of the code, and it reduces the possibility of undesirable side effects.

One form of side effect occurs when a function changes a global variable from within its body rather than through its parameter list. Such a construction is error prone. Correct practice is to effect changes to global variables through the parameter and `return` mechanisms. Adhering to this practice improves modularity and readability, and because changes are localized, programs are typically easier to write and maintain.

All functions have external storage class. This means that we can use the keyword `extern` in function definitions and in function prototypes. For example,

```
extern double   sin(double);
```

is a function prototype for the `sin()` function; for its function definition we can write

```
extern double   sin(double x)
{
   .....
```

The Storage Class `register`

The storage class `register` tells the compiler that the associated variables should be stored in high-speed memory registers, provided it is physically and semantically possible to do so. Because resource limitations and semantic constraints sometimes make this impossible, this storage class defaults to automatic whenever the compiler cannot allocate an appropriate physical register. Typically, the compiler has only a few such registers available. Many of these are required for system use and cannot be allocated otherwise.

Basically, the use of storage class `register` is an attempt to improve execution speed. When speed is a concern, the programmer may choose a few variables that are most frequently accessed and declare them to be of storage class `register`. Common candidates for such treatment include loop variables and function parameters. Here is an example:

```
{
   register int   i;

   for (i = 0; i < LIMIT; ++i) {
      .....
   }
}       /* block exit will free the register */
```

The declaration

```
register i;
```

is equivalent to

```
register int i;
```

If a storage class is specified in a declaration and the type is absent, then the type is `int` by default.

Note that in our example the register variable `i` was declared as close to its place of use as possible. This is to allow maximum availability of the physical registers, using them only when needed. Always remember that a register declaration is taken only as *advice* to the compiler.

The Storage Class `static`

Static declarations have two important and distinct uses. The more elementary use is to allow a local variable to retain its previous value when the block is reentered. This is in contrast to ordinary automatic variables, which lose their value upon block exit and must be reinitialized. The second and more subtle use is in connection with external declarations. We will discuss this in the next section.

As an example of the value-retention use of `static`, we will write the outline of a function that behaves differently depending on how many times it has been called.

```
void f(void)
{
   static int   cnt = 0;

   ++cnt;
   if (cnt % 2 == 0)
      .....              /* do something */
   else
      .....              /* do something different */
}
```

The first time the function is invoked, the variable `cnt` is initialized to zero. On function exit, the value of `cnt` is preserved in memory. Whenever the function is invoked again, `cnt` is not reinitialized. Instead, it retains its previous value from the last time the function was called. The declaration of `cnt` as a `static int` inside of `f()` keeps it private to `f()`. If it were declared outside of the function, then other functions could access it, too.

5.12 Static External Variables

The second and more subtle use of `static` is in connection with external declarations. With external constructs it provides a "privacy" mechanism that is very important for program modularity. By privacy, we mean visibility or scope restrictions on otherwise accessible variables or functions.

At first glance, static external variables seem unnecessary. External variables already retain their values across block and function exit. The difference is that static external variables are scope-restricted external variables. The scope is the remainder of the source file in which they are declared. Thus, they are unavailable to functions defined earlier in the file or to functions defined in other files, even if these functions attempt to use the `extern` storage class keyword.

```
void f(void)
{
   .....      /* v is not available here */
}

static int     v;     /* static external variable */

void g(void)
{
   .....      /* v can be used here */
}
```

Let us use this facility to provide a variable that is global to a family of functions but, at the same time, is private to the file. We will write two pseudo random-number generators, both of which use the same seed. (The algorithm is based on linear congruential methods; see *The Art of Computer Programming*, 2d ed., vol. 2, *Seminumerical Algorithms*, by Donald Ervin Knuth (Reading, MA: Addison-Wesley, 1981.)

```
/* A family of pseudo random number generators. */

#define   INITIAL_SEED       17
#define   MULTIPLIER         25173
#define   INCREMENT          13849
#define   MODULUS            65536
#define   FLOATING_MODULUS   65536.0

static unsigned   seed = INITIAL_SEED;       /* external, but */
                                          /* private to this file */
```

```
unsigned random(void)
{
   seed = (MULTIPLIER * seed + INCREMENT) % MODULUS;
   return seed;
}

double probability(void)
{
   seed = (MULTIPLIER * seed + INCREMENT) % MODULUS;
   return (seed / FLOATING_MODULUS);
}
```

The function `random()` produces an apparently random sequence of integer values between 0 and `MODULUS`. The function `probability()` produces an apparently random sequence of floating values between 0 and 1.

Notice that a call to `random()` or `probability()` produces a new value of the variable `seed` that depends on its old value. Because `seed` is a static external variable, it is private to this file, and its value is preserved between function calls. We can now create functions in other files that invoke these random-number generators without worrying about side effects.

A last use of `static` is as a storage class specifier for function definitions and prototypes. This use causes the scope of the function to be restricted. Static functions are visible only within the file in which they are defined. Unlike ordinary functions, which can be accessed from other files, a `static` function is available throughout its own file, but no other. Again, this facility is useful in developing private modules of function definitions.

```
static int   g(void);          /* function prototype */

void f(int a)                  /* function definition */
{
   .....                       /* g() is available here, */
                               /* but not in other files */
}

static int g(void)             /* function definition */
{
   .....
}
```


5.13 Default Initialization

In C, both external variables and static variables that are not explicitly initialized by the programmer are initialized to zero by the system. This includes arrays, strings, pointers, structures, and unions. For arrays and strings, this means that each element is initialized to zero; for structures and unions, it means that each member is initialized to zero. In contrast to this, automatic and register variables usually are not initialized by the system. This means they start with "garbage" values. Although some C systems do initialize automatic variables to zero, this feature should not be relied on; doing so makes the code nonportable.

5.14 Recursion

A function is said to be recursive if it calls itself, either directly or indirectly. In C, all functions can be used recursively. In its simplest form, the idea of recursion is straightforward. Try the following program:

```
#include <stdio.h>

int main(void)
{
   printf("  The universe is never ending!  ");
   main();
   return 0;
}
```

Here is another example of a recursive function. It computes the sum of the first *n* positive integers.

```
int sum(int n)
{
   if (n <= 1)
      return n;
   else
      return (n + sum(n - 1));
}
```

The recursive function `sum()` is analyzed as illustrated in the following table. The base case is considered, then working out from the base case, the other cases are considered.

Function call	Value returned		
`sum(1)`	`1`		
`sum(2)`	`2 + sum(1)`	or	`2 + 1`
`sum(3)`	`3 + sum(2)`	or	`3 + 2 + 1`
`sum(4)`	`4 + sum(3)`	or	`4 + 3 + 2 + 1`

Simple recursive routines follow a standard pattern. Typically, there is a base case (or cases) that is tested for upon entry to the function. Then there is a general recursive case in which one of the variables, often an integer, is passed as an argument in such a way as to ultimately lead to the base case. In `sum()`, the variable `n` was reduced by 1 each time until the base case with `n` equal to 1 was reached.

Let us write a few more recursive functions to practice this technique. For a nonnegative integer n, the factorial of n, written $n!$, is defined by

$$0! = 1, \qquad n! = n(n-1) \cdots 3 \cdot 2 \cdot 1 \quad \text{for} \quad n > 0$$

or equivalently,

$$0! = 1, \qquad n! = n((n-1)!) \quad \text{for} \quad n > 0$$

Thus, for example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. Using the recursive definition of factorial, it is easy to write a recursive version of the factorial function.

```
int factorial(int n)      /* recursive version */
{
   if (n <= 1)
      return 1;
   else
      return (n * factorial(n - 1));
}
```

This code is correct and will work properly within the limits of integer precision available on a given system. However, because the numbers $n!$ grow large very fast, the function call `factorial(n)` yields a valid result for only a few values of `n`. On our system the function call `factorial(12)` returns a correct value, but if the argument to the function is greater than 12, an incorrect value is returned. This type of programming error is common. Functions that are logically correct can return incorrect values if the

logical operations in the body of the function are beyond the integer precision available to the system.

As in `sum()`, when an integer n is passed to `factorial()`, the recursion activates n nested copies of the function before returning level by level to the original call. This means that n function calls are used for this computation. Most simple recursive functions can be easily rewritten as iterative functions.

```
int factorial(int n)      /* iterative version */
{
   int   product = 1;

   for ( ; n > 1; --n)
      product *= n;
   return product;
}
```

For a given input value, both factorial functions return the same value over the positive integers, but the iterative version requires only one function call regardless of the value passed in.

The next example illustrates a recursive function that manipulates characters. It can easily be rewritten as an equivalent iterative function. We leave this as an exercise. The program first prompts the user to type in a line. Then by means of a recursive function, it writes the line backwards.

```
/* Write a line backwards. */

#include <stdio.h>

void  wrt_it(void);

int main(void)
{
   printf("Input a line:  ");
   wrt_it();
   printf("\n\n");
   return 0;
}

void wrt_it(void)
{
   int   c;

   if ((c = getchar()) != '\n')
      wrt_it();
   putchar(c);
}
```

If the user types in the line `sing a song of sixpence` when prompted, then the following appears on the screen:

```
Input a line:  sing a song of sixpence
ecnepxis fo gnos a gnis
```

Dissection of the *wrt_bkwrds* Program

```
printf("Input a line:  ");
wrt_it();
```

First, a prompt to the user is printed. The user writes a line and terminates it by typing a carriage return, which is the character `\n`. Then the recursive function `wrt_it()` is invoked.

```
void wrt_it(void)
{
   int   c;
```

This function has no test for `EOF`; thus, `c` could be of type `char`.

```
if ((c = getchar()) != '\n')
   wrt_it();
```

If the character read in is not a newline, then `wrt_it()` is invoked again, which causes another character to be read in, and so forth. Each call has its own local storage for the variable `c`. Each `c` is local to a particular invocation of the function `wrt_it()`, and each `c` stores one of the characters in the input stream. The function calls are stacked until the newline character is read.

```
putchar(c);
```

Only after the newline character is read does anything get written. Each invocation of `wrt_it()` now prints the value stored in its local variable `c`. First the newline character is printed, then the character just before it, and so on, until the first character is printed. Thus, the input line is reversed.

Efficiency Considerations

Many algorithms have both iterative and recursive formulations. Typically, recursion is more elegant and requires fewer variables to make the same calculation. Recursion takes care of its bookkeeping by stacking arguments and variables for each invocation. This stacking of arguments, while invisible to the user, is still costly in time and space. On some machines a simple recursive call with one integer argument can require eight 32-bit words on the stack.

Let us discuss efficiency with respect to the calculation of the Fibonacci sequence, a particularly egregious example. This sequence is defined recursively by

$$f_0 = 0, \qquad f_1 = 1, \qquad f_{i+1} = f_i + f_{i-1} \qquad \text{for } i = 1, 2, \ldots$$

Except for f_0 and f_1, every element in the sequence is the sum of the previous two elements. The sequence begins 0, 1, 1, 2, 3, 5, ... Here is a function that computes Fibonacci numbers recursively:

```
int fibonacci(int n)
{
   if (n <= 1)
      return n;
   else
      return (fibonacci(n - 1) + fibonacci(n - 2));
}
```

As the following table shows, a large number of function calls is required to compute the nth Fibonacci number for even moderate values of n:

Value of n	Value of fibonacci(n)	Number of function calls required to recursively compute fibonacci(n)
0	0	1
1	1	1
2	1	3
.....		
23	28657	92735
24	46368	150049
.....		
42	267914296	866988873
43	433494437	1402817465

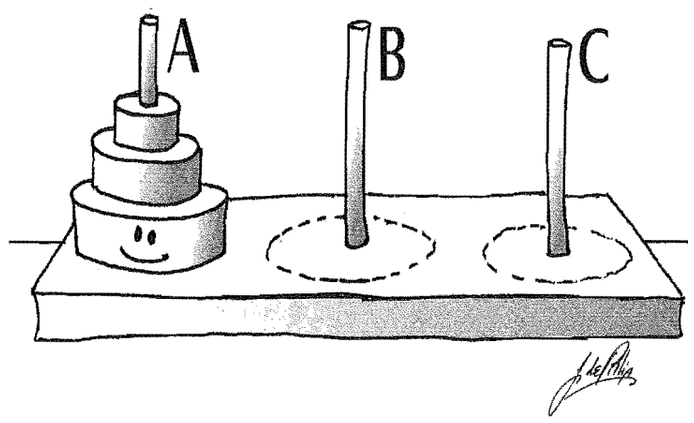
Although it is seductive to use recursion, one must be careful about run-time limitations and inefficiencies. It is sometimes necessary to recode a recursion as an equivalent iteration.

Some programmers feel that because the use of recursion is inefficient, it should not be used. The inefficiencies, however, are often of little consequence—as in the example of the quicksort algorithm given in Section 8.5, "An Example: Sorting with `qsort()`," on page 372. For many applications, recursive code is easier to write, understand, and maintain. These reasons often prescribe its use.

5.15 An Example: The Towers of Hanoi

In the game called *Towers of Hanoi*, there are three towers labeled A, B, and C. The game starts with n disks on tower A. For simplicity, suppose n is 3. The disks are numbered from 1 to 3, and without loss of generality we may assume that the diameter of each disk is the same as its number. That is, disk 1 has diameter 1 (in some unit of measure), disk 2 has diameter 2, and disk 3 has diameter 3. All three disks start on tower A in the order 1, 2, 3. The object of the game is to move all the disks on tower A to tower C. Only one disk may be moved at a time. Any of the towers A, B, or C may be used for the intermediate placement of disks. After each move, the disks on each of the towers must be in order. That is, at no time can a larger disk be placed on a smaller disk.

Towers of Hanoi



We will write a program that uses recursion to produce a list of moves that shows how to accomplish the task of transferring the n disks from tower A to tower C. Here is the program:

In file hanoi.h

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

extern int   cnt;              /* count of the number of moves */

int    get_n_from_user(void);
void   move(int n, char a, char b, char c);
```

In file main.c

```
#include "hanoi.h"

int   cnt = 0;                 /* count of the number of moves */

int main(void)
{
   int    n;                                  /* number of disks */

   n = get_n_from_user();
   assert(n > 0);
   /*
   // Move n disks from tower A to tower C,
   // using tower B as an intermediate tower.
   */
   move(n, 'A', 'B', 'C');                    /* recursive fct */
   return 0;
}
```

In file get.c

```
#include "hanoi.h"

int get_n_from_user(void)
{
   int   n;

   printf("%s",
      "---\n"
      "TOWERS OF HANOI:\n"
      "\n"
      "There are three towers: A, B, and C.\n"
      "\n"
      "The disks on tower A must be moved to tower C.  Only one\n"
      "disk can be moved at a time, and the order on each tower\n"
      "must be preserved at each step.  Any of the towers A, B,\n"
      "or C can be used for intermediate placement of a disk.\n"
      "\n"
      "The problem starts with n disks on Tower A.\n"
      "\n"
      "Input n: ");
   if (scanf("%d", &n) != 1 || n < 1) {
      printf("\nERROR: Positive integer not found - bye!\n\n");
      exit(1);
   }
   printf("\n");
   return n;
}
```

In file move.c

```
#include "hanoi.h"

void move(int n, char a, char b, char c)
{
   if (n == 1) {
      ++cnt;
      printf("%5d: %s%d%s%c%s%c.\n", cnt,
         "Move disk ", 1, " from tower ", a, " to tower ", c);
   }
   else {
      move(n - 1, a, c, b);
      ++cnt;
      printf("%5d: %s%d%s%c%s%c.\n", cnt,
         "Move disk ", n, " from tower ", a, " to tower ", c);
      move(n - 1, b, a, c);
   }
}
```


In our header file, we declare the variable `cnt` to be of storage class `extern`, which means "look for it here or in some other file," and we provide two function prototypes at the bottom of the file. In *main.c*, just above the function definition for `main()`, we declare the external variable `cnt`, and inside of `main()`, we first get `n` from the user and then call the recursive function

```
move(n, 'A', 'B', 'C')
```

We read this as "move *n* disks from tower A to tower C, using tower B as an intermediate tower." Alternatively, we can think of this as "move *n* disks from tower A through tower B to tower C." We want to explain in detail how the function `move()` works, but before we do so, here is the output of the program if we enter 3 when prompted:

```
---
TOWERS OF HANOI:

There are three towers: A, B, and C.

The disks on tower A must be moved to tower C.  Only one
disk can be moved at a time, and the order on each tower
must be preserved at each step.  Any of the towers A, B,
or C can be used for intermediate placement of a disk.

The problem starts with n disks on Tower A.

Input n: 3

    1: Move disk 1 from tower A to tower C.
    2: Move disk 2 from tower A to tower B.
    3: Move disk 1 from tower C to tower B.
    4: Move disk 3 from tower A to tower C.
    5: Move disk 1 from tower B to tower A.
    6: Move disk 2 from tower B to tower C.
    7: Move disk 1 from tower A to tower C.
```

Dissection of the recursive function move()

- `void move(int n, char a, char b, char c)`

This is the header to the function definition for `move()`. In `main()`, where we first call `move()`, the arguments passed to the function are `n`, `'A'`, `'B'`, and `'C'`. Thus, in this first call, the parameters `a`, `b`, and `c` in the function definition have the values `'A'`, `'B'`, and `'C'`, respectively. Note, however, that when `move()` calls itself within the body of `move()`, this association need not hold.

```
if (n == 1) {
   ++cnt;
   printf("%5d: %s%d%s%c%s%c.\n", cnt,
      "Move disk ", 1, " from tower ", a, " to tower ", c);
}
```

If the number of disks on tower `a` is 1, then we increment the counter and then move the disk from tower `a` to tower `c`. That is, we print the instructions to do so.

```
else {
   move(n - 1, a, c, b);
   ++cnt;
   printf("%5d: %s%d%s%c%s%c.\n", cnt,
      "Move disk ", n, " from tower ", a, " to tower ", c);
   move(n - 1, b, a, c);
}
```

When there are `n` disks on tower `a` with `n` greater than 1, we first move `n - 1` disks from tower `a` to tower `b`, using tower `c` as an intermediate tower. Second, we increment the counter and move the last disk on tower `a`, disk `n`, from tower `a` to tower `c`. That is, we print the instructions for doing this. Finally, we move the `n - 1` disks on tower `b` to tower `c`, using tower `a` as an intermediate tower.

The code for the `move()` function is elementary, but strikingly effective. However, to actually see that the output provides valid instructions for solving the game, one needs to draw pictures that illustrate each move. This is, indeed, tedious to do. Later, after we have learned about the use of arrays, we can rewrite our program so that the output illustrates the disks on each tower at each step. Here is the output from the program when `n` has the value 3:


```
          Start: 1 2 3
1: move disk 1:   2 3                 1
2: move disk 2:     3       2         1
3: move disk 1:     3     1 2
4: move disk 3:           1 2         3
5: move disk 1:     1       2         3
6: move disk 2:     1               2 3
7: move disk 1:                   1 2 3
```

To save space, the disks on each tower are written from left to right instead of vertically. From this output, we can tell at a glance that a larger disk is never placed on a smaller one.

According to legend, each day one of the monks in the monastery that attends the three towers of Hanoi moves one of the disks. At the beginning of time, tower A started with 64 disks. When the last disk gets moved, according to legend, the world will come to an end.

Summary

1 Functions are the most general structuring concept in C. They should be used to implement "top-down" problem solving—namely, breaking up a problem into smaller and smaller subproblems until each piece is readily expressed in code.

2 A `return` statement terminates the execution of a function and passes control back to the calling environment. If the `return` statement contains an expression, then the value of the expression is passed back to the calling environment as well.

3 A function prototype tells the compiler the number and type of arguments that are to be passed to the function. The general form of a function prototype is

type function_name (*parameter type list*) ;

The parameter type list is typically a comma-separated list of types, with optional identifiers. If a function takes no arguments, then the keyword `void` is used. If the function returns no value, then the function type is `void`.

4 Arguments to functions are passed by value. They must be type compatible with the corresponding parameter types given by the function definition or the function prototype.

5 The storage class of a function is always `extern`, and its type specifier is `int` unless otherwise explicitly declared. The `return` statement must return a value compatible with the function type.

6 The principal storage class is automatic. Automatic variables appear and disappear with block entry and exit. They can be hidden when an inner block redeclares an outer block identifier.

7 Scope rules are the visibility constraints associated with identifiers. For example, if in a file we have

```
static void f(void)
{
   .....
}

static int   a, b, c;

.....
```

then `f()` will be known throughout this file but in no other, and `a`, `b`, and `c` will be known only in this file and only below the place where they are declared.

8 The external storage class is the default class for all functions and all variables declared outside of functions. These identifiers may be used throughout the program. Such identifiers can be hidden by redeclaration, but their values cannot be destroyed.

9 The keyword `extern` is used to tell the compiler to "look for it elsewhere, either in this file or in some other file."

10 The storage class `register` can be used to try to improve execution speed. It is semantically equivalent to automatic.

11 The storage class `static` is used to preserve exit values of variables. It is also used to restrict the scope of external identifiers. This latter use enhances modularization and program security by providing a form of privacy to functions and variables.

12 External and static variables that are not explicitly initialized by the programmer are initialized to zero by the system.

13 A function is said to be recursive if it calls itself, either directly or indirectly. In C, all functions can be used recursively.

Exercises

1 Write a function `double power(double x, int n)` that will compute x^n, the nth power of x. Check to see that it computes 3.5^7 correctly. (The answer is 6433.9296875.)

2 Use the library function `sqrt()` to write a function that returns the fourth root of its `int` argument `k`. The value returned should be a `double`. Use your function to make a table of values.

3 What gets printed? Explain.

```
#include <stdio.h>

int   z;

void f(int x)
{
   x = 2;
   z += x;
}

int main(void)
{
   z = 5;
   f(z);
   printf("z = %d\n", z);
   return 0;
}
```

4 In traditional C, the general form of a function definition is

type function_name(parameter list)
declarations of the parameters
{ declarations statements }

See Section 5.1, "Function Definition," on page 198, for an example. Rewrite the function definition for `f()` in the previous exercise in this style. Check to see that your compiler will accept it. Does the program produce the same output? (It should.)

5 In this exercise, we want to experiment with function declarations. We will use the function pow() from the mathematics library, but instead of including *math.h*, we will supply our own function declaration. Begin by executing the following program:

```
#include <stdio.h>

double   pow(double, double);

int main(void)
{
   printf("pow(2.0, 3.0) = %g\n", pow(2.0, 3.0));
   return 0;
}
```

Next, change the printf() statement to

```
printf("pow(2, 3) = %g\n", pow(2, 3));
```

and run the program again. Even though you have passed ints to pow() instead of doubles, do you get the correct answer? You should, because the compiler, knowing that double arguments are required, is able to coerce the values supplied into the right form. Next, replace the function prototype by

```
double   pow();        /* traditional style */
```

What happens now? Finally, remove the function declaration completely. Although your program cannot be expected to run correctly, it should compile. Does it? It is important to remember that even though the C system provides object code for functions in libraries, it is the responsibility of the programmer to provide correct function prototypes.

6 In this exercise we want to experiment with an assertion. Execute the following program so that you understand its effects. *Note:* This is *not* a typical use of assertion.

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```



```
int main(void)
{
   int   a, b, cnt = 0, i;

   srand(time(NULL));
   for (i = 0; i < 1000; ++i) {
      a = rand() % 3 + 1;          /* from 1 to 3 */
      b = rand() % 30 + 1;         /* from 1 to 30 */
      if (b - a <= 1)
       continue;
      assert(b - a > 2);
      printf("%3d\n", ++cnt);
   }
   return 0;
}
```

What values of a and b cause the assertion to fail? On average, how many times do you expect to go through the loop?

7 Use the function probability() that we wrote in Section 5.12, "Static External Variables," on page 222, to create two files, one with 100 random numbers and the other with 1,000 random numbers. All the numbers will lie in the range from 0 to 1. If the numbers are truly randomly distributed, then we expect the average to approximate 0.5 as we look at larger and larger sets of them. Compute the average of the numbers in each of your files. Typically, the average of the numbers in the larger file will be closer to 0.5 than the average of the numbers in the smaller file. Is it?

8 A polynomial of degree 2 in x is given by

$$ax^2 + bx + c$$

Write the code for the function

```
double f(double a, double b, double c, double x)
{
   .....
```

that will compute the value of an arbitrary polynomial of degree 2. Use the identity

$$(ax + b)x + c = ax^2 + bx + c$$

to minimize multiplications. Use your function to create a file of points that could be used to plot the polynomial $x^2 - 3x + 2$ on the interval [0, 3]. The variable x should go from 0 to 3 in steps of 0.1. By examining your file, can you tell where the roots of the polynomial are?

9 Experiment with the *tbl_of_powers* program presented in Section 5.4, "An Example: Creating a Table of Powers," on page 203. How many rows of the table can you compute before the powers that are printed become incorrect? Where appropriate, try using the type `double`. Can you get larger numbers this way?

10 In the *tbl_of_powers* program in Section 5.4, "An Example: Creating a Table of Powers," on page 203, we wrote the function definition for `main()` first. What happens if you remove the function prototypes and put `main()` last? Your compiler should be happy with this new arrangement. Is it? What happens if you put `main()` first but do not include any function prototypes? Can you guess what your compiler will complain about? One of our C compilers issues warnings but produces an executable file, whereas our C++ compiler issues errors and refuses to compile the code. What happens on your system?

11 Write a function `int is_prime(int n)` that returns 1 if `n` is a prime and 0 otherwise. *Hint:* If `k` and `n` are positive integers, then `k` divides `n` if and only if `n % k` has value zero.

12 Write a function, say `is_fib_prime()`, that checks whether the *n*th Fibonacci number is prime. Your function should call two other functions: the iterative version of `fibonacci()` that was given in Section 4.13, "An Example: Fibonacci Numbers," on page 175, and the function `is_prime()` that you wrote in the previous exercise. For *n* between 3 and 10, it is true that the *n*th Fibonacci number is prime if and only if *n* is prime. Use your function `is_fib_prime()` to investigate what happens when *n* is bigger than 10.

13 A famous conjecture, called the Goldbach conjecture, says that every even integer *n* greater than 2 has the property that it is the sum of two prime numbers. Computers have been used extensively to test this conjecture. No counterexample has ever been found. Write a program that will prove that the conjecture is true for all the even integers between the symbolic constants `START` and `FINISH`. For example, if you write

```
#define   START    700
#define   FINISH   1100
```


then the output of your program might look like this:

```
Every even number greater than 2 is the sum of two primes:

   700 = 17 + 683
   702 = 11 + 691
   704 =  3 + 701
   .....
  1098 =  5 + 1093
  1100 =  3 + 1097
```

Hint: Use the function `is_prime()` that you wrote in exercise 11, on page 238.

14 Write a function that finds all factors of any particular number. For example,

$9 = 3 \times 3, \qquad 17 = 17\ (prime), \qquad 52 = 2 \times 2 \times 13$

Factoring small numbers is easy, and you should have no trouble writing a program to do this. Note, however, that factoring in general is very difficult. If an integer consists of a few hundred digits, then it may be impossible to factor it, even with a very large computer.

15 This exercise gives you practice on understanding the scope of identifiers. What gets printed by the following code? First, hand simulate the code and record your answers. Then write a program to check your answers.

```
int   a = 1, b = 2, c = 3;

a += b += ++c;
printf("%5d%5d%5d\n", a, b, c);
{
   float   b = 4.0;
   int     c;

   a += c = 5 * b;
   printf("%5d%5.1f%5d\n", a, b, c);
}
printf("%5d%5d%5d\n", a, b, c);
```

16 Rewrite "The universe is never ending!" recursion so that it terminates after 17 calls. Your program should consist of a single `main()` function that calls itself recursively. *Hint:* Use a static variable as a counter.

17 The following function produces incorrect values on some systems:

```
int factorial(int n)          /* wrong */
{
   if (n == 0 || n == 1)
      return 1;
   else
      return (n * factorial(--n));
}
```

Test the function on your system. Explain why the values produced by the function are system-dependent.

18 The greatest common divisor of two positive integers is the largest integer that is a divisor of both of them. For example, 3 is the greatest common divisor of 6 and 15, and 1 is the greatest common divisor of 15 and 22. Here is a recursive function that computes the greatest common divisor of two positive integers:

```
int gcd(int p, int q)
{
   int   r;

   if ((r = p % q) == 0)
      return q;
   else
      return gcd(q, r);
}
```

First, write a program to test the function. Then write and test an equivalent iterative function.

19 In some systems the keyword `extern` is used in function declarations and prototypes in the standard header files. This was a common practice in traditional C systems, but it is usually not done in ANSI C systems. Is this done in your system? *Hint:* Look, for example, in *math.h*.

20 In the program that follows, we have purposely declared some external variables at the bottom of the file. What gets printed?

```
#include <stdio.h>

int main(void)
{
   extern int   a, b, c;      /* look for them elsewhere */

   printf("%3d%3d%3d\n", a, b, c);
   return 0;
}

int   a = 1, b = 2, c = 3;
```

Now change the last line of the program to

```
static int   a = 1, b = 2, c = 3;
```

The variables at the bottom of the file are now static external, so they should not be available in `main()`. Because the external variables referred to in `main()` are not available, your compiler should complain. Does it? (We find that most compilers get confused.)

21 When declaring a variable in traditional C, it is permissible to write the storage class specifier and the type specifier in any order. For example, we can write

```
register int     i;     or     int register     i;
```

In ANSI C, the storage class specifier is supposed to come first. Nonetheless, most ANSI C compilers will accept either order. (If they did not, then some traditional code would not compile.) Check to see if the reverse order works on your compiler.

22 Describe the behavior of the following program:

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

#define   FOREVER   1
#define   STOP      17
```

```
int main(void)
{
   void   f(void);
   while (FOREVER)
      f();
   return 0;
}

void f(void)
{
   static int   cnt = 0;
   printf("cnt = %d\n", ++cnt);
   if (cnt == STOP)
      exit(0);
}
```

23 Let n_0 be a given positive integer. For `i = 0, 1, 2, ...` define

$$n_{i+1} = n_i / 2 \quad \text{if } n_i \text{ is even;} \qquad 3n + 1 \quad \text{if } n_i \text{ is odd;}$$

The sequence stops whenever n_i has the value 1. Numbers that are generated this way are called "hailstones." Write a program that generates some hailstones. The function

```
void hailstones(int n)
{
   .....
```

should be used to compute and print the sequence generated by n. The output of your program might look as follows:

```
Hailstones generated by 77:

        77      232      116       58       29       88
        44       22       11       34       17       52
        26       13       40       20       10        5
        16        8        4        2        1

Number of hailstones generated: 23
```

You will find that all the sequences you generate are finite. Whether this is true in general is still an open question.

24 Write a coin-tossing program that uses the random-number generator `rand()` in the standard library to simulate the toss. Use the expression `rand() % 2` to generate the `int` value 0 or 1. Let "heads" be represented by 1 and "tails" by 0. Run the program for 1,000, 10,000, and 100,000 tosses, and keep track of the longest sequence of heads and the longest sequence of alternating heads and tails—that is, 101010.... If you know some probability theory, see if this simulation is in line with theory. *Caution:* In some traditional C systems, the function `rand()` is notoriously bad. All you will get is one long sequence of alternating heads and tails. If your ANSI C system has borrowed that particular function, then this exercise, although informative, will not be too interesting.

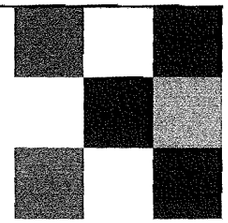
25 If a random-number generator other than `rand()` is available to you, use it to redo the previous exercise. Are the results of your experiments substantially different?

26 Simulations that involve the repeated use of a random-number generator to reproduce a probabilistic event are called Monte Carlo simulations, so called because Monte Carlo has one of the world's most famous gaming casinos. Let us use this technique to find the break-even point in a variation of the birthday game. (See exercise 36, on page 324 in Chapter 6, "Arrays, Pointers, and Strings.") In a roomful of people, at least two of them can be expected to have birthdays on the same day of the year. A common party game is to see if this is true. We wish to find the probability that any two people in a room with n people will have been born in the same month. (To do the analysis for the same day rather than the same month requires the use of an array; otherwise, the ideas are the same.) It is clear that the probability is 0 if there is only one person in the room. For two people the probability is 1/12. (We are assuming that it is equally likely for a person to be born in any month. In reality this is only an approximation.) Simulate the probability by running, say, 1,000 trials with 2, 3, ... , 20 people in the room. (Is 20 too many?) Use 12 variables to count the number of people in the room that were born in each of the 12 months. You can use the expression

```
rand() % 12 + 1
```

to compute the month of birth for each person. (Use a better random-number generator, such as `lrand48()`, if one is available to you.) This expression generates an integer that is randomly distributed in the range from 1 to 12. When any variable gets to 2, that trial is true, meaning that at least two people in the room were born in the same month. The number of true trials divided by 1,000 is the computed simulated probability. What value of n yields the break-even point? That is, find the least n for which the probability is 1/2 or more.

27 Experiment with the *Towers of Hanoi* program. How many moves are required to solve a game that starts with n disks on tower A? If n is 64, and one disk is moved each day, how many billions of years will it take to move all the disks to tower C? If the monks attending the towers of Hanoi manage to move one disk each second, will they complete all moves before the end of the world?



Chapter 6

Arrays, Pointers, and Strings

An array is a data type that uses subscripted variables and makes possible the representation of a large number of homogeneous values. In C, arrays and pointers are closely related concepts. An array name by itself is treated as a constant pointer, and pointers, like arrays, can be subscripted. A distinguishing characteristic of C is its sophisticated use of pointers and pointer arithmetic. Some languages provide "call-by-reference" so that variables in the calling environment can be changed. In C, pointers are used as parameters in function definitions to obtain the effect of "call-by-reference." Strings are one-dimensional arrays of characters. They are sufficiently important to be treated as a special topic.

6.1 One-dimensional Arrays

Programs often use homogeneous data. For example, if we want to manipulate some grades, we might declare

```
int   grade0, grade1, grade2;
```

If the number of grades is large, representing and manipulating the data by means of unique identifiers will be cumbersome. Instead, an array, which is a derived type, can be used. An array can be thought of as a simple variable with an index, or subscript, added. The brackets `[]` are used to contain the array subscripts. To use `grade[0]`, `grade[1]`, and `grade[2]` in a program, we would declare

```
int   grade[3];
```

The integer 3 in the declaration represents the number of elements in the array. The indexing of array elements always starts at 0.

A one-dimensional array declaration is a type followed by an identifier with a bracketed constant integral expression. The value of the expression, which must be positive, is the size of the array. It specifies the number of elements in the array. The array subscripts can range from 0 to *size* - 1. The lower bound of the array subscripts is 0 and the upper bound is *size* - 1. Thus, the following relationships hold:

```
int     a[size];  /* space for a[0], ... , a[size - 1] allocated */
```

lower bound = 0
upper bound = *size* - 1
size = *upper bound* + 1

It is good programming practice to define the size of an array as a symbolic constant.

```
#define   N   100

int   a[N];     /* space for a[0], ... , a[99] is allocated */
```

Given this declaration, the standard programming idiom for processing array elements is with a `for` loop. For example:

```
for (i = 0; i < N; ++i)
   sum += a[i];                /* process element a[i]
```

This iteration starts with the element `a[0]` and iteratively processes each element in turn. Because the termination condition is `i < N`, we avoid the error of falling off the end of the array. The last element processed is, correctly, `a[N - 1]`.

Initialization

Arrays may be of storage class automatic, external, or static, but not register. In traditional C, only external and static arrays can be initialized using an array initializer. In ANSI C, automatic arrays also can be initialized.

one_dimensional_array_initializer ::= { *initializer_list* }
initializer_list ::= *initializer* { , *initializer* }$_{0+}$
initializer ::= *constant_integral_expression*

Consider the example

```
float   f[5] = {0.0, 1.0, 2.0, 3.0, 4.0};
```


This initializes `f[0]` to 0.0, `f[1]` to 1.0, and so on. When a list of initializers is shorter than the number of array elements to be initialized, the remaining elements are initialized to zero. For example,

```
int   a[100] = {0};
```

initializes all the elements of a to zero. If an external or static array is not initialized explicitly, then the system initializes all elements to zero by default. In contrast, automatic arrays are not necessarily initialized by the system. Although some compilers do this, others do not. Programmers should assume that uninitialized automatic arrays start with "garbage" values.

If an array is declared without a size and is initialized to a series of values, it is implicitly given the size of the number of initializers. Thus,

```
int a[] = {2, 3, 5, -7};        and        int a[4] = {2, 3, 5, -7};
```

are equivalent declarations. This feature works with character arrays as well. However, for character arrays an alternate notation is available. A declaration such as

```
char   s[] = "abc";
```

is taken by the compiler to be equivalent to

```
char   s[] = {'a', 'b', 'c', '\0'};
```

Subscripting

If a is an array, we can write `a[`*expr*`]`, where *expr* is an integral expression, to access an element of the array. We call *expr* a subscript, or index, of a. Let us assume that the declaration

```
int   i, a[N];
```

has been made, where `N` is a symbolic constant. The expression `a[i]` can be made to refer to any element of the array by assignment of an appropriate value to the subscript `i`. A single array element `a[i]` is accessed when `i` has a value greater than or equal to 0 and less than or equal to `N - 1`. If `i` has a value outside this range, a run-time error will occur when `a[i]` is accessed. Overrunning the bounds of an array is a common programming error. The effect of the error is system-dependent, and can be quite confusing. One frequent result is that the value of some unrelated variable will be returned or modified. It is the programmer's job to ensure that all subscripts to arrays stay within bounds.