1. Introduction to C

C is a general-purpose, structured programming language. Its instructions consist of terms that resemble algebraic expressions, augmented by certain English *keywords* such as `if`, `else`, `for`, `do` and `while`. In this respect C resembles other high-level structured programming languages such as Pascal and Fortran. C also contains certain additional features, however, that allow it to be used at a lower level, thus bridging the gap between machine language and the more conventional high-level languages. This flexibility allows C to be used for *systems programming* (e.g., for writing operating systems) as well as for *applications programming* (e.g., for writing a program to solve a complicated system of mathematical equations, or for writing a program to bill customers).

C is characterized by the ability to write very concise source programs, due in part to the large number of operators included within the language. It has a relatively small instruction set, though actual implementations include extensive *library functions* which enhance the basic instructions. Furthermore, the language encourages users to write additional library functions of their own. Thus the features and capabilities of the language can easily be extended by the user.

C compilers are commonly available for computers of all sizes, and C interpreters are becoming increasingly common. The compilers are usually compact, and they generate object programs that are small and highly efficient when compared with programs compiled from other high-level languages. The interpreters are less efficient, though they are easier to use when developing a new program. Many programmers begin with an interpreter, and then switch to a compiler once the program has been debugged (i.e., once all of the programming errors have been removed).

Another important characteristic of C is that its programs are highly portable, even more so than with other high-level languages. The reason for this is that C relegates most computer-dependent features to its library functions. Thus, every version of C is accompanied by its own set of library functions, which are written for the particular characteristics of the host computer. These library functions are relatively standardized, however, and each individual library function is generally accessed in the same manner from one version of C to another. Therefore, most C programs can be processed on many different computers with little or no alteration.

2. Structure of a C Program

Every C program consists of one or more modules called *functions*. One of the functions must be called `main`. The program will always begin by executing the `main` function, which may access other functions.

Each function must contain:

1. A function *heading*, which consists of the function name, followed by an optional list of *arguments*, enclosed in parentheses.
2. A list of argument *declarations*, if arguments are included in the heading.
3. A *compound statement*, which comprises the remainder of the function.

The arguments are symbols that represent information being passed between the function and other parts of the program. (Arguments are also referred to as *parameters*.)

Each compound statement is enclosed within a pair of braces, i.e., { }. The braces may contain one or more elementary statements (called *expression statements*) and other compound statements. Thus compound statements may be nested, one within another. Each expression statement must end with a semicolon (;).

Comments (remarks) may appear anywhere within a program, as long as they are placed within the delimiters /* and */ (e.g., `/* this is a comment */`). Such comments are helpful in identifying the program's principal features or in explaining the underlying logic of various program features.

These program components will be discussed in much greater detail later in this book. For now, the reader should be concerned only with an overview of the basic features that characterize most C programs.

3. Data Types

C supports several different types of data, each of which may be represented differently within the computer's memory. The basic data types are listed below. Typical memory requirements are also given. (The memory requirements for each data type will determine the permissible range of values for that data type. Note that the memory requirements for each data type may vary from one C compiler to another.)

Data Type	*Description*	*Typical Memory Requirements*
`int`	integer quantity	2 bytes or one word (varies from one compiler to another)
`char`	single character	1 byte
`float`	floating-point number (i.e., a number containing a decimal point and/or an exponent)	1 word (4 bytes)
`double`	double-precision floating-point number (i.e., more significant figures, and an exponent which may be larger in magnitude)	2 words (8 bytes)

C compilers written for personal computers or small minicomputers (i.e., computers whose natural word size is less than 32 bits) generally represent a word as 4 bytes (32 bits).

The basic data types can be augmented by the use of the data type *qualifiers* `short`, `long`, `signed` and `unsigned`. For example, integer quantities can be defined as `short int`, `long int` or `unsigned int` (these data types are usually written simply as `short`, `long` or `unsigned`, and are understood to be integers). The interpretation of a qualified integer data type will vary from one C compiler to another, though there are some commonsense relationships. Thus, a `short int` may require less memory than an ordinary `int` or it may require the same amount of memory as an ordinary `int`, but it will never exceed an ordinary `int` in word length. Similarly, a `long int` may require the same amount of memory as an ordinary `int` or it may require more memory, but it will never be less than an ordinary `int`.

If `short int` and `int` both have the same memory requirements (e.g., 2 bytes), then `long int` will generally have double the requirements (e.g., 4 bytes). Or if `int` and `long int` both have the same memory requiremements (e.g., 4 bytes) then `short int` will generally have half the memory requirements (e.g., 2 bytes). Remember that the specifics will vary from one C compiler to another.

An `unsigned` int has the same memory requirements as an ordinary `int`. However, in the case of an ordinary `int` (or a `short int` or a `long int`), the leftmost bit is reserved for the sign. With an `unsigned int`, all of the bits are used to represent the numerical value. Thus, an `unsigned int` can be approximately twice as large as an ordinary `int` (though, of course, negative values are not permitted). For example, if an ordinary `int` can vary from −32,768 to +32,767 (which is typical for a 2-byte `int`), then an `unsigned int` will be allowed to vary from 0 to 65,535. The `unsigned` qualifier can also be applied to other qualified `int`s, e.g., `unsigned short int` or `unsigned long int`.

The `char` type is used to represent individual characters. Hence, the `char` type will generally require only one byte of memory. Each `char` type has an equivalent integer interpretation, however, so that a `char` is a really a special kind of short integer (see Sec. 2.4). With most compilers, a `char` data type will permit a range of values extending from 0 to 255. Some compilers represent the `char` data type as having a range of values extending from −128 to +127. There may also be `unsigned char` data (with typical values ranging from 0 to 255), or `signed char` data (with values ranging from −128 to +127).

Some compilers permit the qualifier `long` to be applied to `float` or to `double`, e.g., `long float`, or `long double`. However, the meaning of these data types will vary from one C compiler to another. Thus, `long float` may be equivalent to `double`. Moreover, `long double` may be equivalent to `double`, or it

4. Constants

There are four basic types of constants in C. They are *integer constants*, *floating-point constants*, *character constants* and *string constants*

Moreover, there are several different kinds of integer and floating-point constants, as discussed below.

Integer and floating-point constants represent numbers. They are often referred to collectively as *numeric-type* constants. The following rules apply to all numeric-type constants.

1. Commas and blank spaces cannot be included within the constant.
2. The constant can be preceded by a minus (–) sign if desired. (Actually the minus sign is an *operator* that changes the sign of a positive constant, though it can be thought of as a part of the constant itself.)
3. The value of a constant cannot exceed specified minimum and maximum bounds. For each type of constant, these bounds will vary from one C compiler to another.

Let us consider each type of constant individually.

Integer Constants

An *integer constant* is an integer-valued number. Thus it consists of a sequence of digits. Integer constants can be written in three different number systems: decimal (base 10), octal (base 8) and hexadecimal (base 16). Beginning programmers rarely, however, use anything other than decimal integer constants.

A *decimal* integer constant can consist of any combination of digits taken from the set 0 through 9. If the constant contains two or more digits, the first digit must be something other than 0.

EXAMPLE 2.4 Several valid decimal integer constants are shown below.

```
0    1    743    5280    32767    9999
```

The following decimal integer constants are written incorrectly for the reasons stated.

12,245	illegal character (,).
36.0	illegal character (.).
10 20 30	illegal character (blank space).
123-45-6789	illegal character (-).
0900	the first digit cannot be a zero.

An *octal* integer constant can consist of any combination of digits taken from the set 0 through 7. However the first digit must be 0, in order to identify the constant as an octal number.

EXAMPLE 2.5 Several valid octal integer constants are shown below.

```
0    01    0743    077777
```

The following octal integer constants are written incorrectly for the reasons stated.

743	Does not begin with 0.
05280	Illegal digit (8).
0777.777	Illegal character (.).

A *hexadecimal* integer constant must begin with either 0x or 0X. It can then be followed by any combination of digits taken from the sets 0 through 9 and a through f (either upper- or lowercase). Note that the letters a through f (or A through F) represent the (decimal) quantities 10 through 15, respectively.

EXAMPLE 2.6 Several valid hexadecimal integer constants are shown below.

```
0x        0X1        0X7FFF        0xabcd
```

The following hexadecimal integer constants are written incorrectly for the reasons stated.

0X12.34	Illegal character (.).
0BE38	Does not begin with 0x or 0X.
0x.4bff	Illegal character (.).
0XDEFG	Illegal character (G).

Floating-Point Constants

A *floating-point constant* is a base-10 number that contains either a decimal point or an exponent (or both).

EXAMPLE 2.8 Several valid floating-point constants are shown below.

```
0.        1.          0.2          827.602
50000.    0.000743    12.3         315.0066
2E-8      0.006e-3    1.6667E+8    .12121212e12
```

The following are *not* valid floating-point constants for the reasons stated.

1	Either a decimal point or an exponent must be present.
1,000.0	Illegal character (,).
2E+10.2	The exponent must be an integer quantity (it cannot contain a decimal point).
3E 10	Illegal character (blank space) in the exponent.

EXAMPLE 2.9 The quantity 3×10^5 can be represented in C by any of the following floating-point constants.

```
300000.    3e5     3e+5    3E5       3.0e+
.3e6       0.3E6   30E4    30.E+4    300e3
```

Similarly, the quantity 5.026×10^{-17} can be represented by any of the following floating-point constants.

```
5.026E-17    .5026e-16    50.26e-18    .0005026E-13
```

Character Constants

A *character constant* is a single character, enclosed in apostrophes (i.e., single quotation marks).

EXAMPLE 2.10 Several character constants are shown below.

```
'A'      'x'      '3'      '?'      ' '
```

Notice that the last constant consists of a blank space, enclosed in apostrophes.

Most computers, and virtually all personal computers, make use of the ASCII (i.e., American Standard Code for Information Interchange) character set, in which each individual character is numerically encoded with its own unique 7-bit combination (hence a total of $2^7 = 128$ different characters). Table 2-1 contains the ASCII character set, showing the decimal equivalent of the 7 bits that represent each character. Notice that the characters are ordered as well as encoded. In particular, the digits are ordered consecutively in their proper numerical sequence (0 to 9), and the letters are arranged consecutively in their proper alphabetical order, with uppercase characters preceding lowercase characters. This allows character-type data items to be compared with one another, based upon their relative order within the character set.

EXAMPLE 2.11 Several character constants and their corresponding values, as defined by the ASCII character set, are shown below.

Constant	*Value*
'A'	65
'x'	120
'3'	51
'?'	63
' '	32

Escape Sequences

Certain nonprinting characters, as well as the backslash (\) and the apostrophe ('), can be expressed in terms of *escape sequences*. An escape sequence always begins with a backward slash and is followed by one or more special characters. For example, a line feed (LF), which is referred to as a *newline* in C, can be represented as \n. Such escape sequences always represent single characters, even though they are written in terms of two or more characters.

The commonly used escape sequences are listed below.

Character	*Escape Sequence*
bell (alert)	\a
backspace	\b
horizontal tab	\t
vertical tab	\v
newline (line feed)	\n
form feed	\f
carriage return	\r
quotation mark (")	\"
apostrophe (')	\'
question mark (?)	\?
backslash (\)	\\
null	\0

String Constants

A *string constant* consists of any number of consecutive characters (including none), enclosed in (double) quotation marks.

EXAMPLE 2.14 Several string constants are shown below.

```
"green"          "Washington, D.C. 20005"       "270-32-3456"
"$19.95"         "THE CORRECT ANSWER IS:"       "2*(I+3)/J"
"       "        "Line 1\nLine 2\nLine 3"       ""
```

5. Variable and Arrays

A *variable* is an identifier that is used to represent some specified type of information within a designated portion of the program. In its simplest form, a variable is an identifier that is used to represent a single data item; i.e., a numerical quantity or a character constant. The data item must be assigned to the variable at some point in the program. The data item can then be accessed later in the program simply by referring to the variable name.

A given variable can be assigned different data items at various places within the program. Thus, the information represented by the variable can change during the execution of the program. However, the data type associated with the variable cannot change.

EXAMPLE 2.18 A C program contains the following lines.

```
int a, b, c;
char d;
. . .
a = 3;
b = 5;
c = a + b;
d = 'a';
. . .
a = 4;
b = 2;
c = a - b;
d = 'W';
```

The first two lines are *type declarations*, which state that a, b and c are integer variables, and that d is a char-type variable. Thus a, b and c will each represent an integer-valued quantity, and d will represent a single character. These type declarations will apply throughout the program (more about this in Sec. 2.6).

The next four lines cause the following things to happen: the integer quantity 3 is assigned to a, 5 is assigned to b, and the quantity represented by the sum a + b (i.e., 8) is assigned to c. The character 'a' is then assigned to d.

In the third line within this group, notice that the values of the variables a and b are accessed simply by writing the variables on the right-hand side of the equal sign.

The last four lines redefine the values assigned to the variables as follows: the integer quantity 4 is assigned to a, replacing the earlier value, 3; then 2 is assigned to b, replacing the earlier value, 5; then the difference between a and b (i.e., 2) is assigned to c, replacing the earlier value, 8. Finally, the character 'W' is assigned to d, replacing the earlier character, 'a'.

The *array* is another kind of variable that is used extensively in C. An array is an identifier that refers to a *collection* of data items that all have the same name. The data items must all be of the same type (e.g., all integers, all characters, etc.). The individual data items are represented by their corresponding *array elements* (i.e., the first data item is represented by the first array element, etc.). The individual array elements are distinguished from one another by the value that is assigned to a *subscript*.

EXAMPLE 2.19 Suppose that x is a 10-element array. The first element is referred to as x[0], the second as x[1], and so on. The last element will be x[9].

The subscript associated with each element is shown in square braces. Thus, the value of the subscript for the first element is 0, the value of the subscript for the second element is 1, and so on. For an n-element array, the subscripts always range from 0 to n-1.

There are several different ways to categorize arrays (e.g., integer arrays, character arrays, one-dimensional arrays, multi-dimensional arrays). For now, we will confine our attention to only one type of array: the one-dimensional, char-type array (often called a one-dimensional *character* array). This type of array is generally used to represent a string. Each array element will represent one character within the string. Thus, the entire array can be thought of as an ordered list of characters.

Since the array is one-dimensional, there will be a single *subscript* (sometimes called an *index*) whose value refers to individual array elements. If the array contains n elements, the subscript will be an integer quantity whose values range from 0 to n-1. Note that an n-character string will require an (n+1)-element array, because of the null character (\0) that is automatically placed at the end of the string.

EXAMPLE 2.20 Suppose that the string "California" is to be stored in a one-dimensional character array called letter. Since "California" contains 10 characters, letter will be an 11-element array. Thus, letter[0] will represent the letter C, letter[1] will represent a, and so on, as summarized below. Note that the last (i.e., the 11th) array element, letter[10], represents the null character which signifies the end of the string.

Element Number	*Subscript Value*	*Array Element*	*Corresponding Data Item (String Character)*
1	0	letter[0]	C
2	1	letter[1]	a
3	2	letter[2]	l
4	3	letter[3]	i
5	4	letter[4]	f
6	5	letter[5]	o
7	6	letter[6]	r
8	7	letter[7]	n
9	8	letter[8]	i
10	9	letter[9]	a
11	10	letter[10]	\0

From this list we can determine, for example, that the 5th array element, `letter[4]`, represents the letter `f`, and so on. The array elements and their contents are shown schematically in Fig. 2.1.

	C	a	l	i	f	o	r	n	i	a	\0
Subscript:	0	1	2	3	4	5	6	7	8	9	10

An 11-element character array

6. Declarations

A *declaration* associates a group of variables with a specific data type. All variables must be declared before they can appear in executable statements.

A declaration consists of a data type, followed by one or more variable names, ending with a semicolon. Each array variable must be followed by a pair of square brackets, containing a positive integer which specifies the size (i.e., the number of elements) of the array.

EXAMPLE 2.21 A C program contains the following type declarations.

```
int   a, b, c;
float root1, root2;
char  flag, text[80];
```

Thus, `a`, `b` and `c` are declared to be integer variables, `root1` and `root2` are floating-point variables, `flag` is a char-type variable and `text` is an 80-element, char-type array. Note the square brackets enclosing the size specification for `text`.

These declarations could also have been written as follows.

```
int a;
int b;
int c;
float root1;
float root2;
char flag;
char text[80];
```

This form may be useful if each variable is to be accompanied by a comment explaining its purpose. In small programs however, items of the same type are usually combined in a single declaration.

7. Expressions

An *expression* represents a single data item, such as a number or a character. The expression may consist of a single entity, such as a constant, a variable, an array element or a reference to a function. It may also consist of some combination of such entities, interconnected by one or more *operators*. The use of expressions involving operators is particularly common in C, as in most other programming languages.

Expressions can also represent logical conditions that are either true or false. However, in C the conditions *true* and *false* are represented by the integer values 1 and 0, respectively. Hence logical-type expressions really represent numerical quantities.

EXAMPLE 2.27 Several simple expressions are shown below.

```
a + b
x = y
c = a + b
x <= y
x == y
++i
```

The first expression involves use of the *addition operator* (+). This expression represents the sum of the values assigned to the variables a and b.

The second expression involves the *assignment operator* (=). In this case, the expression causes the value represented by y to be assigned to x. We have already encountered the use of this operator in several earlier examples (see Examples 1.6 through 1.13, 2.25 and 2.26). C includes several additional assignment operators, as discussed in Sec. 3.4.

In the third line, the value of the expression (a + b) is assigned to the variable c. Note that this combines the features of the first two expressions (addition and assignment).

The fourth expression will have the value 1 (true) if the value of x is less than or equal to the value of y. Otherwise, the expression will have the value 0 (false). In this expression, <= is a *relational operator* that compares the values of the variables x and y.

The fifth expression is a test for equality (compare with the second expression, which is an assignment expression). Thus, the expression will have the value 1 (true) if the value of x is equal to the value of y. Otherwise, the expression will have the value 0 (false).

The last expression causes the value of the variable i to be increased by 1 (i.e., *incremented*). Thus, the expression is equivalent to

```
i = i + 1
```

8. Statements

A *statement* causes the computer to carry out some action. There are three different classes of statements in C. They are *expression statements*, *compound statements* and *control statements*.

An expression statement consists of an expression followed by a semicolon. The execution of an expression statement causes the expression to be evaluated.

EXAMPLE 2.28 Several expression statements are shown below.

```
a = 3;
c = a + b;
++i;
printf("Area = %f", area);
;
```

The first two expression statements are assignment-type statements. Each causes the value of the expression on the righ of the equal sign to be assigned to the variable on the left. The third expression statement is an incrementing-typ statement, which causes the value of i to increase by 1.

The fourth expression statement causes the printf function to be evaluated. This is a standard C library functio that writes information out of the computer (more about this in Sec. 3.6). In this case, the message Area = will b displayed, followed by the current value of the variable area. Thus, if area represents the value 100., the statement wil generate the message

```
Area = 100.
```

The last expression statement does nothing, since it consists of only a semicolon. It is simply a mechanism fo providing an empty expression statement in places where this type of statement is required. Consequently, it is called *null statement*.

A compound statement consists of several individual statements enclosed within a pair of braces { }. The individual statements may themselves be expression statements, compound statements or control statements. Thus, the compound statement provides a capability for embedding statements within other statements. Unlike an expression statement, a compound statement does *not* end with a semicolon.

EXAMPLE 2.29 A typical compound statement is shown below.

```
{
    pi = 3.141593;
    circumference = 2. * pi * radius;
    area = pi * radius * radius;
}
```

9. Arithmetic Operators

There are five *arithmetic operators* in C. They are

Operator	Purpose
+	addition
-	subtraction
*	multiplication
/	division
%	remainder after integer division

The % operator is sometimes referred to as the *modulus operator*.

The operands acted upon by arithmetic operators must represent numeric values. Thus, the operands can be integer quantities, floating-point quantities or characters (remember that character constants represent integer values, as determined by the computer's character set). The remainder operator (%) requires that both operands be integers and the second operand be nonzero. Similarly, the division operator (/) requires that the second operand be nonzero.

Division of one integer quantity by another is referred to as *integer division*. This operation always results in a truncated quotient (i.e., the decimal portion of the quotient will be dropped). On the other hand, if a division operation is carried out with two floating-point numbers, or with one floating-point number and one integer, the result will be a floating-point quotient.

EXAMPLE 3.1 Suppose that a and b are integer variables whose values are 10 and 3, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	Value
a + b	13
a - b	7
a * b	30
a / b	3
a % b	1

10. Relational and logical Operators

There are four *relational operators* in C. They are

Operator	*Meaning*
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

These operators all fall within the same precedence group, which is lower than the arithmetic and unary operators. The associativity of these operators is left to right.

Closely associated with the relational operators are the following two *equality operators*.

Operator	*Meaning*
==	equal to
!=	not equal to

The equality operators fall into a separate precedence group, beneath the relational operators. These operators also have a left-to-right associativity.

These six operators are used to form logical expressions, which represent conditions that are either true or false. The resulting expressions will be of type integer, since *true* is represented by the integer value 1 and *false* is represented by the value 0.

EXAMPLE 3.15 Suppose that i, j and k are integer variables whose values are 1, 2 and 3, respectively. Several logical expressions involving these variables are shown below.

Expression	*Interpretation*	*Value*
i < j	true	1
(i + j) >= k	true	1
(j + k) > (i + 5)	false	0
k != 3	false	0
j == 2	true	1

EXAMPLE 3.16 Suppose that `i` is an integer variable whose value is 7, `f` is a floating-point variable whose value is 5.5, and `c` is a character variable that represents the character `'w'`. Several logical expressions that make use of these variables are shown below. Each expression involves two different type operands. (Assume that the ASCII character set applies.)

Expression	*Interpretation*	*Value*
`f > 5`	true	1
`(i + f) <= 10`	false	0
`c == 119`	true	1
`c != 'p'`	true	1
`c >= 10 * (i + f)`	false	0

In addition to the relational and equality operators, C contains two *logical operators* (also called *logical connectives*). They are

Operator	*Meaning*
&&	and
||	or

These operators are referred to as *logical and* and *logical or*, respectively.

The logical operators act upon operands that are themselves logical expressions. The net effect is to combine the individual logical expressions into more complex conditions that are either true or false. The result of a *logical and* operation will be true only if both operands are true, whereas the result of a *logical or* operation will be true if either operand is true or if both operands are true. In other words, the result of a *logical or* operation will be false only if both operands are false.

In this context it should be pointed out that *any* nonzero value, not just 1, is interpreted as true.

EXAMPLE 3.17 Suppose that `i` is an integer variable whose value is 7, `f` is a floating-point variable whose value is 5.5, and `c` is a character variable that represents the character `'w'`. Several complex logical expressions that make use of these variables are shown below.

Expression	*Interpretation*	*Value*
`(i >= 6) && (c == 'w')`	true	1
`(i >= 6) || (c == 119)`	true	1
`(f < 11) && (i > 100)`	false	0
`(c != 'p') || ((i + f) <= 10)`	true	1

The first expression is true because both operands are true. In the second expression, both operands are again true; hence the overall expression is true. The third expression is false because the second operand is false. And finally, the fourth expression is true because the first operand is true.

EXAMPLE 3.18 Suppose that `i` is an integer variable whose value is 7, and `f` is a floating-point variable whose value is 5.5. Several logical expressions which make use of these variables and the logical negation operator are shown below.

Expression	*Interpretation*	*Value*
`f > 5`	true	1
`!(f > 5)`	false	0
`i <= 3`	false	0
`!(i <= 3)`	true	1
`i > (f + 1)`	true	1
`!(i > (f + 1))`	false	0

We will see other examples illustrating the use of the logical negation operator in later chapters of this book.

The hierarchy of operator precedences covering all of the operators discussed so far has become extensive. These operator precedences are summarized below, from highest to lowest.

Operator category	*Operators*	*Associativity*
unary operators	`- ++ -- ! sizeof (type)`	R → L
arithmetic multiply, divide and remainder	`* / %`	L → R
arithmetic add and subtract	`+ -`	L → R
relational operators	`< <= > >=`	L → R
equality operators	`== !=`	L → R
logical *and*	`&&`	L → R
logical *or*	`||`	L → R

11.Conditional Operators

Simple conditional operations can be carried out with the *conditional operator* (? :). An expression that makes use of the conditional operator is called a *conditional expression*. Such an expression can be written in place of the more traditional `if-else` statement, which is discussed in Chap. 6.

A conditional expression is written in the form

```
expression 1  ?  expression 2  :  expression 3
```

When evaluating a conditional expression, *expression 1* is evaluated first. If *expression 1* is true (i.e., if its value is nonzero), then *expression 2* is evaluated and this becomes the value of the conditional expression. However, if *expression 1* is false (i.e., if its value is zero), then *expression 3* is evaluated and this becomes the value of the conditional expression. Note that only one of the embedded expressions (either *expression 2* or *expression 3*) is evaluated when determining the value of a conditional expression.

EXAMPLE 3.26 In the conditional expression shown below, assume that `i` is an integer variable.

```
(i < 0) ? 0 : 100
```

The expression `(i < 0)` is evaluated first. If it is true (i.e., if the value of `i` is less than 0), the entire conditional expression takes on the value 0. Otherwise (if the value of `i` is not less than 0), the entire conditional expression takes on the value 100.

In the following conditional expression, assume that `f` and `g` are floating-point variables.

```
(f < g) ? f : g
```

This conditional expression takes on the value of `f` if `f` is less than `g`; otherwise, the conditional expression takes on the value of `g`. In other words, the conditional expression returns the value of the smaller of the two variables.

EXAMPLE 3.27 Now suppose that `i` is an integer variable, and `f` and `g` are floating-point variables. The conditional expression

```
(f < g) ? i : g
```

involves both integer and floating-point operands. Thus, the resulting expression will be floating-point, even if the value of `i` is selected as the value of the expression (because of rule 2 in Sec. 3.1).

Conditional expressions frequently appear on the right-hand side of a simple assignment statement. The resulting value of the conditional expression is assigned to the identifier on the left.

12.Data Input and Output

We have already seen that the C language is accompanied by a collection of library functions, which includes a number of input/output functions. In this chapter we will make use of six of these functions: `getchar`, `putchar`, `scanf`, `printf`, `gets` and `puts`. These six functions permit the transfer of information between the computer and the standard input/output devices (e.g., a keyboard and a TV monitor). The first two functions, `getchar` and `putchar`, allow single characters to be transferred into and out of the computer; `scanf` and `printf` are the most complicated, but they permit the transfer of single characters, numerical values and strings; `gets` and `puts` facilitate the input and output of strings. Once we have learned how to use these functions, we will be able to write a number of complete, though simple, C programs.

An input/output function can be accessed from anywhere within a program simply by writing the function name, followed by a list of arguments enclosed in parentheses. The arguments represent data items that are sent to the function. Some input/output functions do not require arguments, though the empty parentheses must still appear.

The names of those functions that return data items may appear within expressions, as though each function reference were an ordinary variable (e.g., `c = getchar();`), or they may be referenced as separate statements (e.g., `scanf( . . . );`). Some functions do not return any data items. Such functions are referenced as though they were separate statements (e.g., `putchar( . . . );`).

Most versions of C include a collection of header files that provide necessary information (e.g., symbolic constants) in support of the various library functions. Each file generally contains information in support of a group of related library functions. These files are entered into the program via an `#include` statement at the beginning of the program. As a rule, the header file required by the standard input/output library functions is called `stdio.h` (see Sec. 8.6 for more information about the contents of these header files).

EXAMPLE 4.1 Here is an outline of a typical C program that makes use of several input/output routines from the standard C library.

```
/* sample setup illustrating the use of input/output library functions */

#include <stdio.h>

main()
{
        char c,d;                       /* declarations */
        float x,y;
        int i,j,k;

        c = getchar();                  /* character input */
        scanf("%f", &x);                /* floating-point input */
        scanf("%d %d", &i, &j);         /* integer input */
        . . .                           /* action statements */
        putchar(d);                     /* character output */
        printf("%3d %7.4f", k, y);      /* numerical output */
}
```

The program begins with the preprocessor statement `#include <stdio.h>`. This statement causes the contents of the header file `stdio.h` to be included within the program. The header file supplies required information to the library functions `scanf` and `printf`. (The syntax of the `#include` statement may vary from one version of C to another; some versions of the language use quotes instead of angle-brackets, e.g., `#include "stdio.h"`.)

Following the preprocessor statement is the program heading `main()` and some variable declarations. Several input/output statements are shown in the skeletal outline that follows the declarations. In particular, the assignment statement `c = getchar();` causes a single character to be entered from the keyboard and assigned to the character variable `c`. The first reference to `scanf` causes a floating-point value to be entered from the keyboard and assigned to the floating-point variable `x`, whereas the second reference to `scanf` causes two decimal integer quantities to be entered from the keyboard and assigned to the integer variables `i` and `j`, respectively.

The output statements behave in a similar manner. Thus, the reference to `putchar` causes the value of the character variable `d` to be displayed. Similarly, the reference to `printf` causes the values of the integer variable `k` and the floating-point variable `y` to be displayed.

The details of each input/output statement will be discussed in subsequent sections of this chapter. For now, you should consider only a general overview of the input/output statements appearing in this typical C program.

13.Control Statements

In most of the C programs we have encountered so far, the instructions were executed in the same order in which they appeared within the program. Each instruction was executed once and only once. Programs of this type are unrealistically simple, since they do not include any logical control structures. Thus, these programs did not include tests to determine if certain conditions are true or false, they did not require the repeated execution of groups of statements, and they did not involve the execution of individual groups of statements on a selective basis. Most C programs that are of practical interest make extensive use of features such as these.

For example, a realistic C program may require that a logical test be carried out at some particular point within the program. One of several possible actions will then be carried out, depending on the outcome of the logical test. This is known as *branching*. There is also a special kind of branching, called *selection*, in which one group of statements is selected from several available groups. In addition, the program may require that a group of instructions be executed repeatedly, until some logical condition has been satisfied. This is known as *looping*. Sometimes the required number of repetitions is known in advance; and sometimes the computation continues indefinitely until the logical condition becomes true.

All of these operations can be carried out using the various control statements included in C. We will see how this is accomplished in this chapter. The use of these statements will open the door to programming problems that are much broader and more interesting than those considered earlier.

14. if-else Statement

The `if - else` statement is used to carry out a logical test and then take one of two possible actions, depending on the outcome of the test (i.e., whether the outcome is true or false).

The `else` portion of the `if - else` statement is optional. Thus, in its simplest general form, the statement can be written as

```
if (expression) statement
```

The *expression* must be placed in parentheses, as shown. In this form, the *statement* will be executed only if the *expression* has a nonzero value (i.e., if *expression* is true). If the *expression* has a value of zero (i.e., if *expression* is false), then the *statement* will be ignored.

The *statement* can be either simple or compound. In practice, it is often a compound statement which may include other control statements.

EXAMPLE 6.5 Several representative `if` statements are shown below.

```
if (x < 0) printf("%f", x);

if (pastdue > 0)
   credit = 0;

if (x <= 3.0)   {
   y = 3 * pow(x, 2);
   printf("%f\n", y);
}

if ((balance < 1000.) || (status == 'R'))
   printf("%f", balance);

if ((a >= 0) && (b <= 5))   {
   xmid = (a + b) / 2;
   ymid = sqrt(xmid);
}
```

The first statement causes the value of the floating-point variable x to be printed (displayed) if its value is negative In the second statement, a value of zero is assigned to credit if the value of pastdue exceeds zero. The third statemen involves a compound statement, in which y is evaluated and then displayed if the value of x does not exceed 3. In the fourth statement we see a complex logical expression, which causes the value of balance to be displayed if its value is less than 1000 *or* if status has been assigned the character 'R'.

The last statement involves both a complex logical expression and a compound statement. Thus, the variables xmid and ymid will both be assigned appropriate values if the current value of a is nonnegative *and* the current value of b does not exceed 5.

The general form of an if statement which includes the else clause is

if (*expression*) *statement 1* else *statement 2*

If the *expression* has a nonzero value (i.e., if *expression* is true), then *statement 1* will be executed. Otherwise (i.e., if *expression* is false), *statement 2* will be executed.

EXAMPLE 6.6 Here are several examples illustrating the full if - else statement.

```
if (status == 'S')
   tax = 0.20 * pay;
else
   tax = 0.14 * pay;

if (pastdue > 0)   {
   printf("account number %d is overdue", accountno);
   credit = 0;
}
else
   credit = 1000.0;

if (x <= 3)
   y = 3 * pow(x, 2);
else
   y = 2 * pow(x - 3), 2);
printf("%f\n", balance);
if (circle)  {
   scanf("%f", &radius);
   area = 3.14159 * radius * radius;
   printf("Area of circle = %f", area);
}
else  {
   scanf("%f %f", &length, &width);
   area = length * width;
   printf("Area of rectangle = %f", area);
}
```

It is possible to *nest* (i.e., embed) `if - else` statements, one within another. There are several different forms that nested `if - else` statements can take. The most general form of two-layer nesting is

```
if e1 if e2 s1
      else s2
else  if e3 s3
      else s4
```

where *e1*, *e2* and *e3* represent logical expressions and *s1*, *s2*, *s3* and *s4* represent statements. Now, one complete `if - else` statement will be executed if *e1* is nonzero (true), and another complete `if - else` statement will be executed if *e1* is zero (false). It is, of course, possible that *s1*, *s2*, *s3* and *s4* will contain other `if - else` statements. We would then have multilayer nesting.

Some other forms of two-layer nesting are

```
if e1 s1
else if e2 s2

if e1 s1
else if e2 s2
     else s3

if e1 if e2 s1
      else s2
else s3

if e1 if e2 s1
      else s2
```

In the first three cases the association between the `else` clauses and their corresponding expressions is straightforward. In the last case, however, it is not clear which expression (*e1* or *e2*) is associated with the `else` clause. The answer is *e2*. The rule is that the `else` clause is always associated with the closest preceding unmatched (i.e., `else`-less) `if`. This is suggested by the indentation, though the indentation itself is not the deciding factor. Thus, the last example is equivalent to

```
if e1  {
   if e2 s1 else s2
}
```

If we wanted to associate the `else` clause with *e1* rather than *e2*, we could do so by writing

```
if e1  {
   if e2 s1
}
else s2
```

This type of nesting must be carried out carefully in order to avoid possible ambiguities.

EXAMPLE 6.7 Here is an illustration of three nested `if - else` statements.

```
if ((time >= 0.) && (time < 12.)) printf("Good Morning");
else if ((time >= 12.) && (time < 18.)) printf("Good Afternoon");
     else if ((time >= 18.) && (time < 24.)) printf("Good Evening");
          else printf("Time is out of range");
```

This example causes a different message to be displayed at various times of the day. Specifically, the message `Goo` `Morning` will be displayed if `time` has a value between 0 and 12; `Good Afternoon` will be displayed if `time` has a valu between 12 and 18; and `Good Evening` will be displayed if `time` has a value between 18 and 24. An error messag (`Time is out of range`) will be displayed if the value of `time` is less than zero, or greater than or equal to 24.

15. Loops

The `while` statement is used to carry out looping operations, in which a group of statements is executed repeatedly, until some condition has been satisfied.

The general form of the `while` statement is

```
while (expression) statement
```

The ***statement*** will be executed repeatedly, as long as the ***expression*** is true (i.e., as long ***expression*** has a nonzero value). This ***statement*** can be simple or compound, though it is usually a compound statement. It must include some feature that eventually alters the value of the ***expression***, thus providing a stopping condition for the loop.

EXAMPLE 6.8 Consecutive Integer Quantities Suppose we want to display the consecutive digits 0, 1, 2, . . . , 9, with one digit on each line. This can be accomplished with the following program.

```
#include <stdio.h>

main()     /* display the integers 0 through 9 */

{
    int digit = 0;

    while (digit <= 9)  {
          printf("%d\n", digit);
          ++digit;
    }
}
```

Initially, `digit` is assigned a value of 0. The `while` loop then displays the current value of `digit`, increases its value by 1 and then repeats the cycle, until the value of `digit` exceeds 9. The net effect is that the body of the loop will be repeated 10 times, resulting in 10 consecutive lines of output. Each line will contain a successive integer value, beginning with 0 and ending with 9. Thus, when the program is executed, the following output will be generated.

```
0
1
2
3
4
5
6
7
8
9
```

This program can be written more concisely as

```
#include <stdio.h>

main()      /* display the integers 0 through 9 */
{
    int digit = 0;

    while (digit <= 9)
          printf("%d\n", digit++);
}
```

When executed, this program will generate the same output as the first program.

When a loop is constructed using the `while` statement described in Sec. 6.3, the test for continuation of the loop is carried out at the *beginning* of each pass. Sometimes, however, it is desirable to have a loop with the test for continuation at the *end* of each pass. This can be accomplished by means of the `do - while` statement.
The general form of the `do - while` statement is

```
do statement while (expression);
```

The ***statement*** will be executed repeatedly, as long as the value of ***expression*** is true (i.e., is nonzero). Notice that ***statement*** will always be executed at least once, since the test for repetition does not occur until the end of the first pass through the loop. The ***statement*** can be either simple or compound, though most applications will require it to be a compound statement. It must include some feature that eventually alters the value of ***expression*** so the looping action can terminate.

EXAMPLE 6.11 Consecutive Integer Quantities In Example 6.8 we saw two complete C programs that use the `while` statement to display the consecutive digits 0, 1, 2, . . . , 9. Here is another program to do the same thing, using the `do - while` statement in place of the `while` statement.

```
#include <stdio.h>

main()    /* display the integers 0 through 9 */
{
    int digit = 0;

    do
    printf("%d\n", digit++);
    while (digit <= 9);
}
```

As in the earlier example, `digit` is initially assigned a value of 0. The `do - while` loop displays the current value of `digit`, increases its value by 1, and then tests to see if the current value of `digit` exceeds 9. If so, the loop terminates; otherwise, the loop continues, using the new value of `digit`. Note that the test is carried out at the end of each pass through the loop. The net effect is that the loop will be repeated 10 times, resulting in 10 successive lines of output. Each line will appear exactly as shown in Example 6.8.

Comparing this program with the second program presented in Example 6.8, we see about the same level of complexity in both programs. Neither of the conditional looping structures (i.e., `while` or `do - while`) appears more desirable than the other.

The `for` statement is the third and perhaps the most commonly used looping statement in C. This statement includes an expression that specifies an initial value for an index, another expression that determines whether or not the loop is continued, and a third expression that allows the index to be modified at the end of each pass.

The general form of the `for` statement is

```
for (expression 1; expression 2; expression 3) statement
```

where *expression 1* is used to initialize some parameter (called an *index*) that controls the looping action, *expression 2* represents a condition that must be true for the loop to continue execution, and *expression 3* is used to alter the value of the parameter initially assigned by *expression 1*. Typically, *expression 1* is an assignment expression, *expression 2* is a logical expression and *expression 3* is a unary expression or an assignment expression.

When the `for` statement is executed, *expression 2* is evaluated and tested at the *beginning* of each pass through the loop, and *expression 3* is evaluated at the *end* of each pass. Thus, the `for` statement is equivalent to

```
expression 1;
while (expression 2)  {
   statement
   expression 3;
}
```

The looping action will continue as long as the value of ***expression 2*** is not zero, that is, as long as the logical condition represented by ***expression 2*** is true.

The `for` statement, like the `while` and the `do - while` statements, can be used to carry out looping actions where the number of passes through the loop is not known in advance. Because of the features that are built into the `for` statement, however, it is particularly well suited for loops in which the number of passes *is* known in advance. As a rough rule of thumb, `while` loops are generally used when the number of passes is *not* known in advance, and `for` loops are generally used when the number of passes *is* known in advance.

EXAMPLE 6.14 Consecutive Integer Quantities We have already seen several different versions of a C program that will display the consecutive digits 0, 1, 2, . . . , 9, with one digit on each line (see Examples 6.8 and 6.11). Here is another program which does the same thing. Now, however, we will make use of the `for` statement rather than the `while` statement or the `do - while` statement, as in the earlier examples.

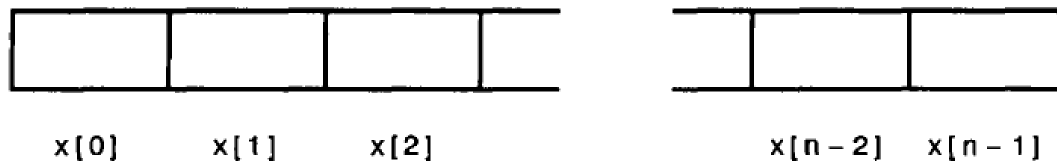
```
#include <stdio.h>

main()   /* display the numbers 0 through 9 */

{
    int digit;

    for (digit = 0; digit <= 9; ++digit)
        printf("%d\n", digit);
}
```

The first line of the `for` statement contains three expressions, enclosed in parentheses. The first expression assigns an initial value 0 to the integer variable `digit`; the second expression continues the looping action as long as the current value of `digit` does not exceed 9 at the *beginning* of each pass; and the third expression increases the value of `digit` by 1 at the *end* of each pass through the loop. The `printf` function, which is included in the `for` loop, produces the desired output, as shown in Example 6.8.

16. Arrays

Many applications require the processing of multiple data items that have common characteristics (e.g., a set of numerical data, represented by $x_1, x_2, \ldots, x_n$). In such situations it is often convenient to place the data items into an *array*, where they will all share the same name (e.g., x). The individual data items can be characters, integers, floating-point numbers, etc. However, they must all be of the same type and the same storage class.

Each array element (i.e., each individual data item) is referred to by specifying the array name followed by one or more *subscripts*, with each subscript enclosed in square brackets. Each subscript must be expressed as a nonnegative integer. In an *n*-element array, the array elements are `x[0]`, `x[1]`, `x[2]`, . . . , `x[n - 1]`, as illustrated in Fig. 9.1. The value of each subscript can be expressed as an integer constant, an integer variable or a more complex integer expression.



x is an n-element, one-dimensional array

The number of subscripts determines the dimensionality of the array. For example, `x[i]` refers to an element in the one-dimensional array `x`. Similarly, `y[i][j]` refers to an element in the two-dimensional array `y`. (We can think of a two-dimensional array as a table, where `y[i][j]` is the *j*th element of the *i*th row.) Higher-dimensional arrays can be also be formed, by adding additional subscripts in the same manner (e.g., `z[i][j][k]`).

Arrays are defined in much the same manner as ordinary variables, except that each array name must be accompanied by a size specification (i.e., the number of elements). For a one-dimensional array, the size is specified by a positive integer expression, enclosed in square brackets. The expression is usually written as a positive integer constant.

In general terms, a one-dimensional array definition may be expressed as

```
storage-class   data-type   array[ expression];
```

where *storage-class* refers to the storage class of the array, *data-type* is the data type, *array* is the array name, and *expression* is a positive-valued integer expression which indicates the number of array elements. The *storage-class* is optional; default values are `automatic` for arrays that are defined within a function or a block, and `external` for arrays that are defined outside of a function.

EXAMPLE 9.1 Several typical one-dimensional array definitions are shown below.

```
int x[100];
char text[80];
static char message[25];
static float n[12];
```

The first line states that `x` is a 100-element integer array, and the second defines `text` to be an 80-element character array. In the third line, `message` is defined as a static 25-element character array, whereas the fourth line establishes `n` as a static 12-element floating-point array.

EXAMPLE 9.3 Shown below are several array definitions that include the assignment of initial values.

```
int digits[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

static float x[6] = {0, 0.25, 0, -0.50, 0, 0};

char color[3] = {'R', 'E', 'D'};
```

Note that x is a static array. The other two arrays (`digits` and `color`) are assumed to be external arrays by virtue of their placement within the program.

The results of these initial assignments, in terms of the individual array elements, are as follows. (Remember that the subscripts in an n-element array range from 0 to n – 1.)

```
digits[0] = 1        x[0] =  0         color[0] = 'R'
digits[1] = 2        x[1] =  0.25      color[1] = 'E'
digits[2] = 3        x[2] =  0         color[2] = 'D'
digits[3] = 4        x[3] = -0.50
digits[4] = 5        x[4] =  0
digits[5] = 6        x[5] =  0
digits[6] = 7
digits[7] = 8
digits[8] = 9
digits[9] = 10
```

EXAMPLE 9.6 Consider the following two character array definitions. Each includes the initial assignment of the string constant "RED". However, the first array is defined as a three-element array, whereas the size of the second array is unspecified.

```
char color[3] = "RED";

char color[] = "RED";
```

The results of these initial assignments are not the same because of the null character, \0, which is automatically added at the end of the second string. Thus, the elements of the first array are

```
color[0] = 'R'
color[1] = 'E'
color[2] = 'D'
```

whereas the elements of the second array are

```
color[0] = 'R'
color[1] = 'E'
color[2] = 'D'
color[3] = '\0'
```

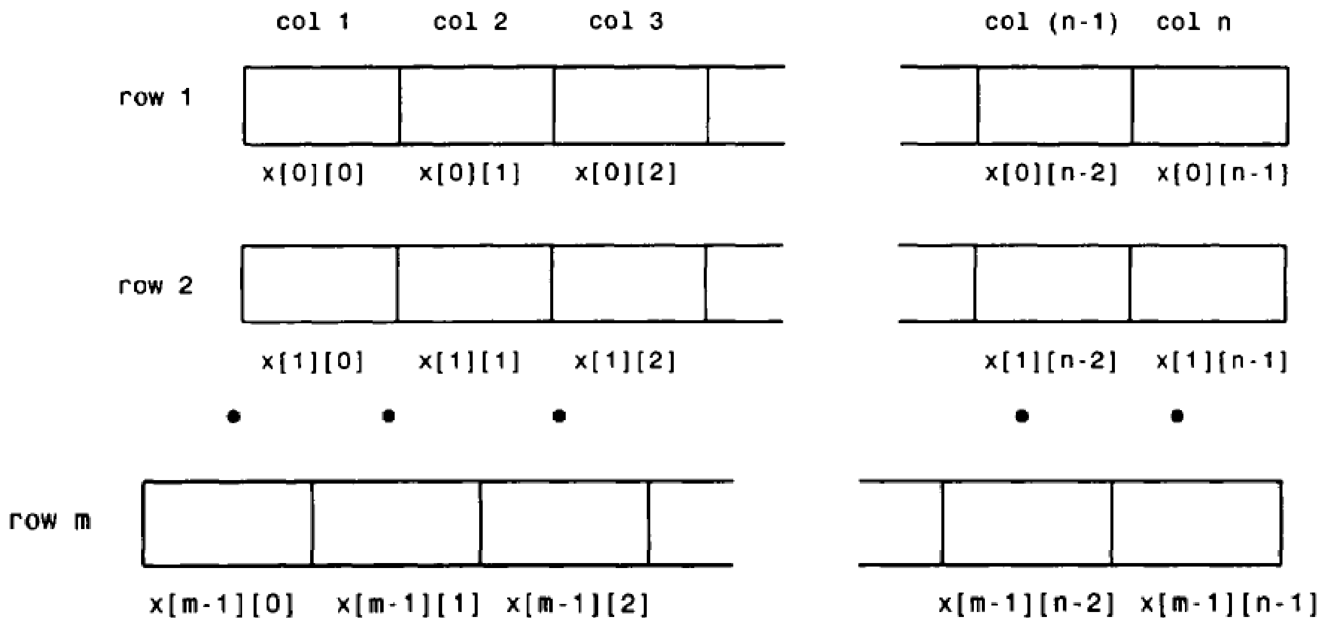
Multidimensional arrays are defined in much the same manner as one-dimensional arrays, except that a separate pair of square brackets is required for each subscript. Thus, a two-dimensional array will require two pairs of square brackets, a three-dimensional array will require three pairs of square brackets, and so on.

In general terms, a multidimensional array definition can be written as

```
storage-class  data-type  array[expression 1][expression 2] . . .[expression n];
```

where *storage-class* refers to the storage class of the array, *data-type* is its data type, *array* is the array name, and *expression 1, expression 2, . . ., expression n* are positive-valued integer expressions that indicate the number of array elements associated with each subscript. Remember that the *storage-class* is optional; the default values are *automatic* for arrays that are defined inside of a function, and *external* for arrays defined outside of a function.

We have already seen that an n-element, one-dimensional array can be thought of as a *list* of values, as illustrated in Fig. 9.1. Similarly, an m × n, two-dimensional array can be thought of as a *table* of values having m rows and n columns, as illustrated in Fig. 9.2. Extending this idea, a three-dimensional array can be visualized as a *set* of tables (e.g., a book in which each page is a table), and so on.



x is a m × n, two-dimensional array

EXAMPLE 9.15 Several typical multidimensional array definitions are shown below.

```
float table[50][50];

char page[24][80];

static double records[100][66][255];

static double records[L][M][N];
```

The first line defines table as a floating-point array having 50 rows and 50 columns (hence 50 × 50 = 2500 elements), and the second line establishes page as a character array with 24 rows and 80 columns (24 × 80 = 1920 elements). The third array can be thought of as a set of 100 static, double-precision tables, each having 66 lines and 255 columns (hence 100 × 66 × 255 = 1,683,000 elements).

The last definition is similar to the preceding definition except that the array size is defined by the symbolic constants L, M and N. Thus, the values assigned to these symbolic constants will determine the actual size of the array.

EXAMPLE 9.16 Consider the following two-dimensional array definition.

```
int values[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

Note that `values` can be thought of as a table having 3 rows and 4 columns (4 elements per row). Since the initial values are assigned by rows (i.e., last subscript increasing most rapidly), the results of this initial assignment are as follows.

```
values[0][0] = 1    values[0][1] = 2     values[0][2] = 3     values[0][3] = 4
values[1][0] = 5    values[1][1] = 6     values[1][2] = 7     values[1][3] = 8
values[2][0] = 9    values[2][1] = 10    values[2][2] = 11    values[2][3] = 12
```

Remember that the first subscript ranges from 0 to 2, and the second subscript ranges from 0 to 3.

EXAMPLE 9.17 Here is a variation of the two-dimensional array definition presented in the last example.

```
int values[3][4] = {
                        {1, 2, 3, 4},
                        {5, 6, 7, 8},
                        {9, 10, 11, 12}
                    };
```

This definition results in the same initial assignments as in the last example. Thus, the four values in the first inner pair of braces are assigned to the array elements in the first row, the values in the second inner pair of braces are assigned to the array elements in the second row, etc. Note that an outer pair of braces is required, containing the inner pairs.

Now consider the following two-dimensional array definition.

```
int values[3][4] = {
                        {1, 2, 3},
                        {4, 5, 6},
                        {7, 8, 9}
                    };
```

This definition assigns values only to the first three elements in each row. Therefore, the array elements will have the following initial values.

```
values[0][0] = 1    values[0][1] = 2    values[0][2] = 3    values[0][3] = 0
values[1][0] = 4    values[1][1] = 5    values[1][2] = 6    values[1][3] = 0
values[2][0] = 7    values[2][1] = 8    values[2][2] = 9    values[2][3] = 0
```

17. Pointers

A *pointer* is a variable that represents the *location* (rather than the *value*) of a data item, such as a variable o an array element. Pointers are used frequently in C, as they have a number of useful applications. Fo example, pointers can be used to pass information back and forth between a function and its reference point In particular, pointers provide a way to return multiple data items from a function via function arguments Pointers also permit references to other functions to be specified as arguments to a given function. This ha the effect of passing functions as arguments to the given function.

Pointers are also closely associated with arrays and therefore provide an alternate way to access individua array elements. Moreover, pointers provide a convenient way to represent multidimensional arrays, allowin a single multidimensional array to be replaced by a lower-dimensional array of pointers. This feature permit a group of strings to be represented within a single array, though the individual strings may differ in length.

Within the computer's memory, every stored data item occupies one or more contiguous memory cells (i.e., adjacent words or bytes). The number of memory cells required to store a data item depends on the type of data item. For example, a single character will typically be stored in one byte (8 bits) of memory; an integer usually requires two contiguous bytes; a floating-point number may require four contiguous bytes; and a double-precision quantity may require eight contiguous bytes. (See Chap. 2 and Appendix D.)

Suppose v is a variable that represents some particular data item. The compiler will automatically assign memory cells for this data item. The data item can then be accessed if we know the location (i.e., the *address*) of the first memory cell.* The address of v's memory location can be determined by the expression &v, where & is a unary operator, called the *address operator*, that evaluates the address of its operand.

Now let us assign the address of v to another variable, pv. Thus,

```
pv = &v
```

This new variable is called a *pointer* to v, since it "points" to the location where v is stored in memory Remember, however, that pv represents v's *address*, not its value. Thus, pv is referred to as a *pointe variable*. The relationship between pv and v is illustrated in Fig. 10.1.

The data item represented by v (i.e., the data item stored in v's memory cells) can be accessed by the expression *pv, where * is a unary operator, called the *indirection operator*, that operates only on a pointer variable. Therefore, *pv and v both represent the same data item (i.e., the contents of the same memory cells). Furthermore, if we write pv = &v and u = *pv, then u and v will both represent the same value; i.e., the value of v will indirectly be assigned to u. (It is assumed that u and v are of the same data type.)

EXAMPLE 10.1 Shown below is a simple program that illustrates the relationship between two integer variables, their corresponding addresses and their associated pointers.

```
#include <stdio.h>

main()
{
    int u = 3;
    int v;
    int *pu;        /* pointer to an integer */
    int *pv;        /* pointer to an integer */

    pu = &u;        /* assign address of u to pu */
    v = *pu;        /* assign value of u to v */
    pv = &v;        /* assign address of v to pv */

    printf("\nu=%d   &u=%X   pu=%X   *pu=%d", u, &u, pu, *pu);
    printf("\n\nv=%d   &v=%X   pv=%X   *pv=%d", v, &v, pv, *pv);
}
```

Note that pu is a pointer to u, and pv is a pointer to v. Therefore pu represents the address of u, and pv represents the address of v. (Pointer declarations will be discussed in the next section.)

Execution of this program results in the following output.

```
u=3   &u=F8E   pu=F8E   *pu=3

v=3   &v=F8C   pv=F8C   *pv=3
```

In the first line, we see that u represents the value 3, as specified in the declaration statement. The address of u is determined automatically by the compiler as F8E (hexadecimal). The pointer pu is assigned this value; hence, pu also represents the (hexadecimal) address F8E. Finally, the value to which pu points (i.e., the value stored in the memory cell whose address is F8E) is 3, as expected.

Similarly, the second line shows that v also represents the value 3. This is expected, since we have assigned the value *pu to v. The address of v, and hence the value of pv, is F8C. Notice that u and v have different addresses. And finally, we see that the value to which pv points is 3, as expected.

The relationships between pu and u, and pv and v, are shown in Fig. 10.2. Note that the memory locations of the pointer variables (i.e., address EC7 for pu, and EC5 for pv) are not displayed by the program.

Pointer variables, like all other variables, must be declared before they may be used in a C program. The interpretation of a pointer declaration differs, however, from the interpretation of other variable declarations. When a pointer variable is declared, the variable name must be preceded by an asterisk (*). This identifies the fact that the variable is a pointer. The data type that appears in the declaration refers to the *object* of the pointer, i.e., the data item that is stored in the address represented by the pointer, rather than the pointer itself.

Thus, a pointer declaration may be written in general terms as

```
data-type *ptvar;
```

where *ptvar* is the name of the pointer variable, and *data-type* refers to the data type of the pointer's object. Remember that an asterisk must precede *ptvar*.

EXAMPLE 10.4 A C program contains the following declarations.

```
float u, v;
float *pv;
```

EXAMPLE 10.5 A C program contains the following declarations.

```
float u, v;
float *pv = &v;
```

The variables u and v are declared to be floating-point variables and pv is declared as a pointer variable that points to a floating-point quantity, as in Example 10.4. In addition, the address of v is initially assigned to pv.

This terminology can be confusing. Remember that these declarations are equivalent to writing

```
float u, v;        /* floating-point variable declarations */
float *pv;         /* pointer variable declaration */
. . . . .
pv = &v;           /* assign v's address to pv */
```

Note that an asterisk is not included in the assignment statement.

In general, it does not make sense to assign an integer value to a pointer variable. An exception, however, is an assignment of 0, which is sometimes used to indicate some special condition. In such situations the recommended programming practice is to define a symbolic constant NULL which represents 0, and to use NULL in the pointer initialization. This practice emphasizes the fact that the zero assignment represents a special situation.