Chapter 2

C Fundamentals

This chapter is concerned with the basic elements used to construct simple C statements. These elements include the C character set, identifiers and keywords, data types, constants, variables and arrays, declarations, expressions and statements. We will see how these basic elements can be combined to form more comprehensive program components.

Some of this material is rather detailed and therefore somewhat difficult to absorb, particularly by an inexperienced programmer. Remember, however, that the purpose of this material is to introduce certain basic concepts and to provide some necessary definitions for the topics that follow in the next few chapters. Therefore, when reading this material for the first time, you need only acquire a general familiarity with the individual topics. A more comprehensive understanding will come later, from repeated references to this material in subsequent chapters.

2.1 THE C CHARACTER SET

C uses the uppercase letters A to Z, the lowercase letters a to z, the digits 0 to 9, and certain special characters as building blocks to form basic program elements (e.g., constants, variables, operators, expressions, etc.). The special characters are listed below.

+	–	*	/	=	%	&	#
!	?	^	"	'	~	\	|
<	>	(	)	[	]	{	}
:	;	.	,	_	(*blank space*)		

Most versions of the language also allow certain other characters, such as @ and $, to be included within strings and comments.

C uses certain combinations of these characters, such as \b, \n and \t, to represent special conditions such as backspace, newline and horizontal tab, respectively. These character combinations are known as *escape sequences*. We will discuss escape sequences in Sec. 2.4. For now we simply mention that each escape sequence represents a single character, even though it is written as two or more characters.

2.2 IDENTIFIERS AND KEYWORDS

Identifiers are names that are given to various program elements, such as variables, functions and arrays. Identifiers consist of letters and digits, in any order, except that *the first character must be a letter*. Both upper- and lowercase letters are permitted, though common usage favors the use of lowercase letters for most types of identifiers. Upper- and lowercase letters are not interchangeable (i.e., an uppercase letter is *not* equivalent to the corresponding lowercase letter.) The underscore character (_) can also be included, and is considered to be a letter. An underscore is often used in the middle of an identifier. An identifier may also begin with an underscore, though this is rarely done in practice.

EXAMPLE 2.1 The following names are valid identifiers.

```
x          y12        sum_1        _temperature
names      area       tax_rate     TABLE
```

The following names are *not* valid identifiers for the reasons stated.

`4th`	The first character must be a letter.
`"x"`	Illegal characters (`"`).
`order-no`	Illegal character (–).
`error flag`	Illegal character (blank space).

An identifier can be arbitrarily long. Some implementations of C recognize only the first eight characters, though most implementations recognize more (typically, 31 characters). Additional characters are carried along for the programmer's convenience.

EXAMPLE 2.2 The identifiers `file_manager` and `file_management` are both grammatically valid. Some compilers may be unable to distinguish between them, however, because the first eight letters are the same for each identifier. Therefore, only one of these identifiers should be used in a single C program.

As a rule, an identifier should contain enough characters so that its meaning is readily apparent. On the other hand, an excessive number of characters should be avoided.

EXAMPLE 2.3 A C program is being written to calculate the future value of an investment. The identifiers `value` or `future_value` are appropriate symbolic names. However, `v` or `fv` would probably be too brief, since the intended representation of these identifiers is not clear. On the other hand, the identifier `future_value_of_an_investment` would be unsatisfactory because it is too long and cumbersome.

There are certain reserved words, called *keywords*, that have standard, predefined meanings in C. These keywords can be used only for their intended purpose; they cannot be used as programmer-defined identifiers.

The standard keywords are

`auto`	`extern`	`sizeof`
`break`	`floatn`	`static`
`case`	`for`	`struct`
`char`	`goto`	`switch`
`const`	`if`	`typedef`
`continue`	`int`	`union`
`default`	`long`	`unsigned`
`do`	`register`	`void`
`double`	`return`	`volatile`
`else`	`short`	`while`
`enum`	`signed`	

Some compilers may also include some or all of the following keywords.

`ada`	`far`	`near`
`asm`	`fortran`	`pascal`
`entry`	`huge`	

Some C compilers may recognize other keywords. Consult a reference manual to obtain a complete list of keywords for your particular compiler.

Note that the keywords are all lowercase. Since uppercase and lowercase characters are not equivalent, it is possible to utilize an uppercase keyword as an identifier. Normally, however, this is not done, as it is considered a poor programming practice.

2.3 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 may refer to a separate, "extra-large" double-precision data type requiring more than two words of memory.

Two additional data types, `void` and `enum`, will be introduced later in this book (`void` is discussed in Sec. 7.2; `enum` is discussed in Sec. 14.1).

Every identifier that represents a number or a character within a C program must be associated with one of the basic data types before the identifier appears in an executable statement. This is accomplished via a *type declaration*, as described in Sec. 2.6.

2.4 CONSTANTS

There are four basic types of constants in C. They are *integer constants*, *floating-point constants*, *character constants* and *string constants* (there are also *enumeration constants*, which are discussed in Sec. 14.1). 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).

The magnitude of an integer constant can range from zero to some maximum value that varies from one computer to another (and from one compiler to another, on the same computer). A typical maximum value for most personal computers and many minicomputers is 32767 decimal (equivalent to 77777 octal or `7fff` hexadecimal), which is $2^{15} - 1$. Mainframe computers generally permit larger values, such as 2,147,483,647 (which is $2^{31} - 1$).* You should determine the appropriate value for the version of C used with your particular computer.

Unsigned and Long Integer Constants

Unsigned integer constants may exceed the magnitude of ordinary integer constants by approximately a factor of 2, though they may not be negative.* An unsigned integer constant can be identified by appending the letter `U` (either upper- or lowercase) to the end of the constant.

Long integer constants may exceed the magnitude of ordinary integer constants, but require more memory within the computer. With some computers (and/or some compilers), a long integer constant will automatically be generated simply by specifying a quantity that exceeds the normal maximum value. It is *always* possible, however, to create a long integer constant by appending the letter `L` (either upper- or lowercase) to the end of the constant.

An unsigned long integer may be specified by appending the letters `UL` to the end of the constant. The letters may be written in either upper- or lowercase. However, the `U` must precede the `L`.

EXAMPLE 2.7 Several unsigned and long integer constants are shown below.

Constant	*Number System*
`50000U`	decimal (unsigned)
`123456789L`	decimal (long)
`123456789UL`	decimal (unsigned long)
`0123456L`	octal (long)
`0777777U`	octal (unsigned)
`0X50000U`	hexadecimal (unsigned)
`0XFFFFFUL`	hexadecimal (unsigned long)

* Suppose a particular computer uses a w-bit word. Then an ordinary integer quantity may fall within the range -2^{w-1} to $+2^{w-1} - 1$, whereas an unsigned integer quantity may vary from 0 to $2^w - 1$. A short integer may substitute $w/2$ for w, and a long integer may substitute $2w$ for w. These rules may vary from one computer to another.

The maximum permissible values of unsigned and long integer constants will vary from one computer (and one compiler) to another. With some computers, the maximum permissible value of a long integer constant may be the same as that for an ordinary integer constant; other computers may allow a long integer constant to be much larger than an ordinary integer constant. You are again advised to determine the appropriate values for your particular version of C.

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.

If an exponent is present, its effect is to shift the location of the decimal point to the right, if the exponent is positive, or to the left, if the exponent is negative. If a decimal point is not included within the number, it is assumed to be positioned to the right of the last digit.

The interpretation of a floating-point constant with an exponent is essentially the same as scientific notation, except that the base 10 is replaced by the letter `E` (or e). Thus, the number 1.2×10^{-3} would be written as `1.2E-3` or `1.2e-3`. This is equivalent to `0.12e-2`, or `12e-4`, etc.

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
```

Floating-point constants have a much greater range than integer constants. Typically, the magnitude of a floating-point constant might range from a minimum value of approximately `3.4E-38` to a maximum of `3.4E+38`. Some versions of the language permit floating-point constants that cover a wider range, such as `1.7E-308` to `1.7E+308`. Also, the value `0.0` (which is less than either `3.4E-38` or `1.7E-308`) is a valid floating-point constant. You should determine the appropriate values for the version of C used on your particular computer.

Floating-point constants are normally represented as double-precision quantities in C. Hence, each floating-point constant will typically occupy 2 words (8 bytes) of memory. Some versions of C permit the specification of a "single-precision," floating-point constant, by appending the letter `F` (in either upper- or lowercase) to the end of the constant (e.g., `3E5F`). Similarly, some versions of C permit the specification of a "long" floating-point constant, by appending the letter `L` (upper- or lowercase) to the end of the constant (e.g., `0.123456789E-33L`).

The precision of floating-point constants (i.e, the number of significant figures) will vary from one version of C to another. Virtually all versions of the language permit at least six significant figures, and some versions permit as many as eighteen significant figures. You should determine the appropriate number of significant figures for your particular version of C.

Numerical Accuracy

It should be understood that integer constants are exact quantities, whereas floating-point constants are approximations. The reasons for this are beyond the current scope of discussion. However, you should understand that the floating-point constant `1.0` might be represented within the computer's memory as `0.99999999...`, even though it might appear as `1.0` when it is displayed (because of automatic rounding). Therefore floating-point values cannot be used for certain purposes, such as counting, indexing, etc., where exact values are required. We will discuss these restrictions as they arise, in later chapters of this book.

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.

Character constants have integer values that are determined by the computer's particular character set. Thus, the value of a character constant may vary from one computer to another. The constants themselves, however, are independent of the character set. This feature eliminates the dependence of a C program on any particular character set (more about this later).

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

These values will be the same for all computers that utilize the ASCII character set. The values will be different, however, for computers that utilize an alternate character set.

IBM mainframe computers, for example, utilize the EBCDIC (i.e., Extended Binary Coded Decimal Information Code) character set, in which each individual character is numerically encoded with its own unique 8-bit combination. The EBCDIC character set is distinctly different from the ASCII character set.

Table 2-1 The ASCII Character Set

ASCII Value	*Character*	*ASCII Value*	*Character*	*ASCII Value*	*Character*	*ASCII Value*	*Character*
0	NUL	32	(blank)	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

The first 32 characters and the last character are control characters. Usually, they are not displayed. However, some versions of C (some computers) support special graphics characters for these ASCII values. For example, 001 may represent the character ☺, 002 may represent ☻, and so on.

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*	*ASCII Value*
bell (alert)	\a	007
backspace	\b	008
horizontal tab	\t	009
vertical tab	\v	011
newline (line feed)	\n	010
form feed	\f	012
carriage return	\r	013
quotation mark (")	\"	034
apostrophe (')	\'	039
question mark (?)	\?	063
backslash (\)	\\	092
null	\0	000

EXAMPLE 2.12 Shown below are several character constants, expressed in terms of escape sequences.

```
'\n'        '\t'        '\b'        '\''        '\\'        '\"'
```

Note that the last three escape sequences represent an apostrophe, a backslash and a quotation mark, respectively.

Of particular interest is the escape sequence \0. This represents the *null character* (ASCII 000), which is used to indicate the end of a *string* (see below). Note that the null character constant '\0' is *not* equivalent to the character constant '0'.

An escape sequence can also be expressed in terms of one, two or three octal digits which represent single-character bit patterns. The general form of such an escape sequence is \ooo, where each o represents an octal digit (0 through 7). Some versions of C also allow an escape sequence to be expressed in terms of one or more hexadecimal digits, preceded by the letter x. The general form of a hexadecimal escape sequence is \xhh, where each h represents a hexadecimal digit (0 through 9 and a through f). The letters can be either upper- or lowercase. The use of an octal or hexadecimal escape sequence is usually less desirable than writing the character constant directly, however, since the bit patterns may be dependent upon some particular character set.

EXAMPLE 2.13 The letter A is represented by the decimal value 065 in the ASCII character set. This value is equivalent to the octal value 101. (The equivalent binary bit pattern is 001 000 001.) Hence the character constant 'A' can be expressed as the octal escape sequence '\101'.

In some versions of C, the letter A can also be expressed as a hexadecimal escape sequence. The hexadecimal equivalent of the decimal value 65 is 41. (The equivalent binary bit pattern is 0100 0001.) Hence the character constant 'A' can be expressed as '\x41', or as '\X41'.

It should be understood that the preferred way to represent this character constant is simply 'A'. In this form, the character constant is not dependent upon its equivalent ASCII representation.

Escape sequences can only be written for certain special characters, such as those listed above, or in terms of octal or hexadecimal digits. If a backslash is followed by any other character, the result may be unpredictable. Usually, however, it will simply be ignored.

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"        ""
```

Note that the string constant `"Line 1\nLine 2\nLine 3"` extends over three lines, because of the newline characters that are embedded within the string. Thus, this string would be displayed as

```
Line 1
Line 2
Line 3
```

Also, notice that the string `""` is a *null* (empty) string.

Sometimes certain special characters (e.g., a backslash or a quotation mark) must be included as a part of a string constant. These characters *must* be represented in terms of their escape sequences. Similarly, certain nonprinting characters (e.g., tab, newline) can be included in a string constant if they are represented in terms of their corresponding escape sequences.

EXAMPLE 2.15 The following string constant includes three special characters that are represented by their corresponding escape sequences.

```
"\tTo continue, press the \"RETURN\" key\n"
```

The special characters are `\t` (horizontal tab), `\"` (double quotation marks, which appears twice), and `\n` (newline).

The compiler automatically places a null character (`\0`) at the end of every string constant, as the last character within the string (before the closing double quotation mark). This character is not visible when the string is displayed. However, we can easily examine the individual characters within a string, and test to see whether or not each character is a null character (we will see how this is done in Chap. 6). Thus, the end of every string can be readily identified. This is very helpful if the string is scanned on a character-by-character basis, as is required in many applications. Also, in many situations this end-of-string designation eliminates the need to specify a maximum string length.

EXAMPLE 2.16 The string constant shown in Example 2.15 actually contains 38 characters. This includes five blank spaces, four special characters (horizontal tab, two quotation marks and newline) represented by escape sequences, and the null character (`\0`) at the end of the string.

Remember that a character constant (e.g., `'A'`) and the corresponding single-character string constant (`"A"`) are not equivalent. Also remember that a character constant has an equivalent integer value, whereas a single-character string constant does not have an equivalent integer value and, in fact, consists of two characters — the specified character followed by the null character (`\0`).

EXAMPLE 2.17 The character constant `'w'` has an integer value of 119 in the ASCII character set. It does not have a null character at the end. In contrast, the string constant `"w"` actually consists of two characters — the lowercase letter `w` and the null character `\0`. This constant does not have a corresponding integer value.

2.5 VARIABLES 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.

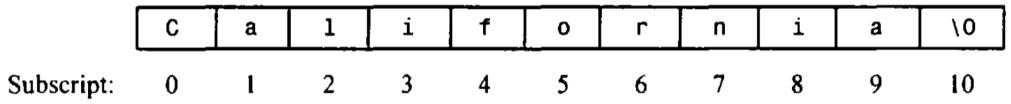
Since the array is one-dimensional, there will be a single *subscript* (sometimes called an *index*) whose value refers to individual array elements. If the array contains `n` elements, the subscript will be an integer quantity whose values range from 0 to `n-1`. Note that an `n`-character string will 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.



An 11-element character array

Fig. 2.1

We will discuss arrays in much greater detail in Chaps. 9 and 10.

2.6 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. (Recall that the permissible data types are discussed in Sec. 2.3.) 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.

Integer-type variables can be declared to be *short integer* for smaller integer quantities, or *long integer* for larger integer quantities. (Recall that some C compilers allocate less storage space to short integers, and additional storage space to long integers.) Such variables are declared by writing `short int` and `long int`, or simply `short` and `long`, respectively.

EXAMPLE 2.22 A C program contains the following type declarations.

```
short int a, b, c;
long  int r, s, t;
int   p, q;
```

Some compilers will allocate less storage space to the short integer variables a, b and c than to the integer variables p and q. Typical values are two bytes for each short integer variable, and four bytes (one word) for each ordinary integer variable. The maximum permissible values of a, b and c will be smaller than the maximum permissible values of p and q when using a compiler of this type.

Similarly, some compilers will allocate additional storage space to the long integer variables r, s and t than to the integer variables p and q. Typical values are two words (8 bytes) for each long integer variable, and one word (4 bytes) for each ordinary integer variable. The maximum permissible values of r, s and t will be larger than the maximum permissible values of p and q when using one of these compilers.

The above declarations could have been written as

```
short a, b, c;
long  r, s, t;
int   p, q;
```

Thus, `short` and `short int` are equivalent, as are `long` and `long int`.

An integer variable can also be declared to be *unsigned*, by writing `unsigned int`, or simply `unsigned`, as the type indicator. Unsigned integer quantities can be larger than ordinary integer quantities (approximately twice as large), but they cannot be negative.

EXAMPLE 2.23 A C program contains the following type declarations.

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

The unsigned variables x and y can represent values that are twice as large as the values represented by a and b. However, x and y cannot represent negative quantities. For example, if the computer uses 2 bytes for each integer quantity, then a and b may take on values that range from −32768 to +32767, whereas the values of x and y may vary from 0 to +65535.

Floating-point variables can be declared to be *double precision* by using the type indicator `double` or `long float` rather than `float`. In most versions of C, the exponent within a double-precision quantity is larger in magnitude than the exponent within an ordinary floating-point quantity. Hence, the quantity represented by a double-precision variable can fall within a greater range. Moreover, a double-precision quantity will usually be expressed in terms of more significant figures.

EXAMPLE 2.24 A C program contains the following type declarations.

```
float  c1, c2, c3;
double root1, root2;
```

With a particular C compiler, the double-precision variables `root1` and `root2` represent values that can vary (in magnitude) from approximately 1.7×10^{-308} to $1.7 \times 10^{+308}$. However, the floating-point variables `c1`, `c2` and `c3` are restricted (in magnitude) to the range 3.4×10^{-38} to 3.4×10^{38}. Furthermore, the values represented by `root1` and `root2` will each be expressed in terms of 18 significant figures, whereas the values represented by `c1`, `c2` and `c3` will each be expressed in terms of only 6 significant figures.

The last declaration could have been written

```
long float root1, root2;
```

though the original form (i.e., `double root1, root2;`) is more common.

Initial values can be assigned to variables within a type declaration. To do so, the declaration must consist of a data type, followed by a variable name, an equal sign (=) and a constant of the appropriate type. A semicolon must appear at the end, as usual.

EXAMPLE 2.25 A C program contains the following type declarations.

```
int    c = 12;
char   star = '*';
float  sum = 0.;
double factor = 0.21023e-6;
```

Thus, `c` is an integer variable whose initial value is 12, `star` is a char-type variable initially assigned the character `'*'`, `sum` is a floating-point variable whose initial value is `0.`, and `factor` is a double-precision variable whose initial value is 0.21023×10^{-6}.

A character-type array can also be initialized within a declaration. To do so, the array is usually written without an explicit size specification (the square brackets are empty). The array name is then followed by an equal sign, the string (enclosed in quotes), and a semicolon. This is a convenient way to assign a string to a character-type array.

EXAMPLE 2.26 A C program contains the following type declaration.

```
char text[] = "California";
```

This declaration will cause `text` to be an 11-element character array. The first 10 elements will represent the 10 characters within the word `California`, and the 11th element will represent the null character (`\0`) which is automatically added at the end of the string.

The declaration could also have been written

```
char text[11] = "California";
```

where the size of the array is explicitly specified. In such situations it is important, however, that the size be specified correctly. If the size is too small, e.g.,

```
char text[10] = "California";
```

the characters at the end of the string (in this case, the null character) will be lost. If the size is too large, e.g.,

```
char text[20] = "California";
```

the extra array elements may be assigned zeros, or they may be filled with meaningless characters.

Array declarations that include the assignment of initial values can only appear in certain places within a C program (see Chap. 9).

In Chap. 8 we shall see that variables can be categorized by *storage class* as well as by data type. The storage class specifies the portion of the program within which the variables are recognized. Moreover, the storage class associated with an array determines whether or not the array can be initialized. This is explained in Chap. 9.

2.7 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
```

The operator ++, which indicates incrementing, is called a *unary* operator because it has only one *operand* (in this case, the variable `i`). C includes several other operators of this type, as discussed in Sec. 3.2.

The C language includes many different kinds of operators and expressions. Most are described in detail in Chap. 3. Others will be discussed elsewhere in this book, as the need arises.

2.8 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 right of the equal sign to be assigned to the variable on the left. The third expression statement is an incrementing-type 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 function that writes information out of the computer (more about this in Sec. 3.6). In this case, the message `Area =` will be displayed, followed by the current value of the variable `area`. Thus, if `area` represents the value 100., the statement will generate the message

```
Area = 100.
```

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

This particular compound statement consists of three assignment-type expression statements, though it is considered a single entity within the program in which it appears. Note that the compound statement does not end with a semicolon after the brace.

Control statements are used to create special program features, such as logical tests, loops and branches. Many control statements require that other statements be embedded within them, as illustrated in the following example.

EXAMPLE 2.30 The following control statement creates a conditional loop in which several actions are executed repeatedly, until some particular condition is satisfied.

```
while (count <= n)   {
    printf("x = ");
    scanf("%f", &x);
    sum += x;
    ++count;
}
```

This statement contains a compound statement, which in turn contains four expression statements. The compound statement will continue to be executed as long as the value of `count` does not exceed the value of `n`. Note that `count` increases in value during each pass through the loop.

Chapter 6 presents a detailed discussion of control statements.

2.9 SYMBOLIC CONSTANTS

A *symbolic constant* is a name that substitutes for a sequence of characters. The characters may represent a numeric constant, a character constant or a string constant. Thus, a symbolic constant allows a name to appear in place of a numeric constant, a character constant or a string. When a program is compiled, each occurrence of a symbolic constant is replaced by its corresponding character sequence.

Symbolic constants are usually defined at the beginning of a program. The symbolic constants may then appear later in the program in place of the numeric constants, character constants, etc. that the symbolic constants represent.

A symbolic constant is defined by writing

```
#define  name  text
```

where *name* represents a symbolic name, typically written in uppercase letters, and *text* represents the sequence of characters that is associated with the symbolic name. Note that *text* does not end with a semicolon, since a symbolic constant definition is not a true C statement. Moreover, if *text* were to end with a semicolon, this semicolon would be treated as though it were a part of the numeric constant, character constant or string constant that is substituted for the symbolic name.

EXAMPLE 2.31 A C program contains the following symbolic constant definitions.

```
#define  TAXRATE  0.23

#define  PI  3.141593

#define  TRUE   1
#define  FALSE  0

#define  FRIEND  "Susan"
```

Notice that the symbolic names are written in uppercase, to distinguish them from ordinary C identifiers. Also, note that the definitions do not end with semicolons.

Now suppose that the program contains the statement

```
area = PI * radius * radius;
```

During the compilation process, each occurrence of a symbolic constant will be replaced by its corresponding text. Thus, the above statement will become

```
area = 3.141593 * radius * radius;
```

Now suppose that a semicolon had been (incorrectly) included in the definition for `PI`, i.e.,

```
#define  PI  3.141593;
```

The assignment statement for `area` would then become

```
area = 3.141593; * radius * radius;
```

Note the semicolon preceding the first asterisk. This is clearly incorrect, and it will cause an error in the compilation.

The substitution of text for a symbolic constant will be carried out anywhere beyond the `#define` statement, *except* within a string. Thus, any text enclosed by (double) quotation marks will be unaffected by this substitution process.

EXAMPLE 2.32 A C program contains the following statements.

```
#define  CONSTANT  6.023E23
int c;
. . . . .
printf("CONSTANT = %f", c);
```

The `printf` statement will be unaffected by the symbolic constant definition, since the term `"CONSTANT = %f"` is a string constant. If, however, the `printf` statement were written as

```
printf("CONSTANT = %f", CONSTANT);
```

then the `printf` statement would become

```
printf("CONSTANT = %f", 6.023E23);
```

during the compilation process.

Symbolic constants are not required when writing C programs. Their use is recommended, however, since they contribute to the development of clear, orderly programs. For example, symbolic constants are more readily identified than the information that they represent, and the symbolic names usually suggest the significance of their associated data items. Furthermore, it is much easier to change the value of a single symbolic constant than to change every occurrence of some numerical constant that may appear in several places within the program.

The `#define` feature, which is used to define symbolic constants, is one of several features included in the C *preprocessor* (i.e., a program that provides the first step in the translation of a C program into machine language). A detailed discussion of the C preprocessor is included in Chap. 14 (see Sec. 14.6).