Chapter 3

Operators and Expressions

We have already seen that individual constants, variables, array elements and function references can be joined together by various operators to form expressions. We have also mentioned that C includes a large number of operators which fall into several different categories. In this chapter we examine certain of these categories in detail. Specifically, we will see how arithmetic operators, unary operators, relational and logical operators, assignment operators and the conditional operator are used to form expressions.

The data items that operators act upon are called *operands*. Some operators require two operands, while others act upon only one operand. Most operators allow the individual operands to be expressions. A few operators permit only single variables as operands (more about this later).

3.1 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*.

There is no exponentiation operator in C. However, there is a *library function* (pow) to carry out exponentiation (see Sec. 3.6).

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

Notice the truncated quotient resulting from the division operation, since both operands represent integer quantities. Also, notice the integer remainder resulting from the use of the modulus operator in the last expression.

Now suppose that v1 and v2 are floating-point variables whose values are 12.5 and 2.0, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	*Value*
`v1 + v2`	14.5
`v1 - v2`	10.5
`v1 * v2`	25.0
`v1 / v2`	6.25

Finally, suppose that c1 and c2 are character-type variables that represent the characters P and T, respectively. Several arithmetic expressions that make use of these variables are shown below, together with their resulting values (based upon the ASCII character set).

Expression	*Value*
`c1`	80
`c1 + c2`	164
`c1 + c2 + 5`	169
`c1 + c2 + '5'`	217

Note that P is encoded as (decimal) 80, T is encoded as 84, and 5 is encoded as 53 in the ASCII character set, as shown in Table 2-1.

If one or both operands represent negative values, then the addition, subtraction, multiplication and division operations will result in values whose signs are determined by the usual rules of algebra. Integer division will result in truncation toward zero; i.e., the resultant will always be smaller in magnitude than the true quotient.

The interpretation of the remainder operation is unclear when one of the operands is negative. Most versions of C assign the sign of the first operand to the remainder. Thus, the condition

```
a = ((a / b) * b) + (a % b)
```

will always be satisfied, regardless of the signs of the values represented by a and b.

Beginning programmers should exercise care in the use of the remainder operation when one of the operands is negative. In general, it is best to avoid such situations.

EXAMPLE 3.2 Suppose that a and b are integer variables whose values are 11 and −3, respectively. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	*Value*
`a + b`	8
`a - b`	14
`a * b`	−33
`a / b`	−3
`a % b`	2

If a had been assigned a value of −11 and b had been assigned 3, then the value of a / b would still be −3 but the value of a % b would be −2. Similarly, if a and b had both been assigned negative values (−11 and −3, respectively), then the value of a / b would be 3 and the value of a % b would be −2.

Note that the condition

```
a = ((a / b) * b) + (a % b)
```

will be satisfied in each of the above cases. Most versions of C will determine the sign of the remainder in this manner, though this feature is unspecified in the formal definition of the language.

EXAMPLE 3.3 Here is an illustration of the results that are obtained with floating-point operands having different signs. Let `r1` and `r2` be floating-point variables whose assigned values are –0.66 and 4.50. Several arithmetic expressions involving these variables are shown below, together with their resulting values.

Expression	*Value*
`r1 + r2`	`3.84`
`r1 - r2`	`-5.16`
`r1 * r2`	`-2.97`
`r1 / r2`	`-0.1466667`

Operands that differ in type may undergo type conversion before the expression takes on its final value. In general, the final result will be expressed in the highest precision possible, consistent with the data types of the operands. The following rules apply when neither operand is `unsigned`.

1. If both operands are floating-point types whose precisions differ (e.g., a `float` and a `double`), the lower-precision operand will be converted to the precision of the other operand, and the result will be expressed in this higher precision. Thus, an operation between a `float` and a `double` will result in a `double`; a `float` and a `long double` will result in a `long double`; and a `double` and a `long double` will result in a `long double`. (Note: In some versions of C, all operands of type `float` are automatically converted to `double`.)
2. If one operand is a floating-point type (e.g., `float`, `double` or `long double`) and the other is a `char` or an `int` (including `short int` or `long int`), the `char` or `int` will be converted to the floating-point type and the result will be expressed as such. Hence, an operation between an `int` and a `double` will result in a `double`.
3. If neither operand is a floating-point type but one is a `long int`, the other will be converted to `long int` and the result will be `long int`. Thus, an operation between a `long int` and an `int` will result in a `long int`.
4. If neither operand is a floating-point type or a `long int`, then both operands will be converted to `int` (if necessary) and the result will be `int`. Thus, an operation between a `short int` and an `int` will result in an `int`.

A detailed summary of these rules is given in Appendix D. Conversions involving `unsigned` operands are also explained in Appendix D.

EXAMPLE 3.4 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-type variable that represents the character w. Several expressions which include the use of these variables are shown below. Each expression involves operands of two different types. Assume that the ASCII character set is being used.

Expression	*Value*	*Type*
`i + f`	`12.5`	double-precision
`i + c`	`126`	integer
`i + c - '0'`	`78`	integer
`(i + c) - (2 * f / 5)`	`123.8`	double-precision

Note that w is encoded as (decimal) 119 and 0 is encoded as 48 in the ASCII character set, as shown in Table 2-1.

The value of an expression can be converted to a different data type if desired. To do so, the expression must be preceded by the name of the desired data type, enclosed in parentheses, i.e.,

```
(data type) expression
```

This type of construction is known as a *cast*.

EXAMPLE 3.5 Suppose that `i` is an integer variable whose value is 7, and `f` is a floating-point variable whose value is 8.5. The expression

```
(i + f) % 4
```

is invalid, because the first operand `(i + f)` is floating-point rather than integer. However, the expression

```
((int) (i + f)) % 4
```

forces the first operand to be an integer and is therefore valid, resulting in the integer remainder 3.

Note that the explicit type specification applies only to the first operand, not the entire expression.

The data type associated with the expression itself is not changed by a cast. Rather, it is the *value* of the expression that undergoes type conversion wherever the cast appears. This is particularly relevant when the expression consists of only a single variable.

EXAMPLE 3.6 Suppose that `f` is a floating-point variable whose value is 5.5. The expression

```
((int) f) % 2
```

contains two integer operands and is therefore valid, resulting in the integer remainder 1. Note, however, that `f` remains a floating-point variable whose value is 5.5, even though the value of `f` was converted to an integer (5) when carrying out the remainder operation.

The operators within C are grouped hierarchically according to their *precedence* (i.e., order of evaluation). Operations with a higher precedence are carried out before operations having a lower precedence. The natural order of evaluation can be altered, however, through the use of parentheses, as illustrated in Example 3.5.

Among the arithmetic operators, `*`, `/` and `%` fall into one precedence group, and `+` and `-` fall into another. The first group has a higher precedence than the second. Thus, multiplication, division and remainder operations will be carried out before addition and subtraction.

Another important consideration is the *order* in which consecutive operations within the same precedence group are carried out. This is known as *associativity*. Within each of the precedence groups described above, the associativity is left to right. In other words, consecutive addition and subtraction operations are carried out from left to right, as are consecutive multiplication, division and remainder operations.

EXAMPLE 3.7 The arithmetic expression

```
a - b / c * d
```

is equivalent to the algebraic formula $a - [(b / c) \times d]$. Thus, if the floating-point variables `a`, `b`, `c` and `d` have been assigned the values 1., 2., 3. and 4., respectively, the expression would represent the value $-1.666666\cdots$, since

$$1. - [(2. / 3.) \times 4.] = 1. - [0.666666\cdots \times 4.] = 1. - 2.666666\cdots = -1.666666\cdots$$

Notice that the division is carried out first, since this operation has a higher precedence than subtraction. The resulting quotient is then multiplied by 4., because of left-to-right associativity. The product is then subtracted from 1., resulting in the final value of $-1.666666\cdots$.

The natural precedence of operations can be altered through the use of parentheses, thus allowing the arithmetic operations within an expression to be carried out in any desired order. In fact, parentheses can be *nested*, one pair within another. In such cases the innermost operations are carried out first, then the next innermost operations, and so on.

EXAMPLE 3.8 The arithmetic expression

```
(a - b) / (c * d)
```

is equivalent to the algebraic formula $(a - b) / (c \times d)$. Thus, if the floating-point variables `a`, `b`, `c` and `d` have been assigned the values 1., 2., 3. and 4., respectively, the expression would represent the value $-0.08333333\cdots$, since

$$(1. - 2.) / (3. \times 4.) = -1. / 12. = -0.08333333\cdots$$

Compare this result with that obtained in Example 3.7.

Sometimes it is a good idea to use parentheses to clarify an expression, even though the parentheses may not be required. On the other hand, the use of overly complex expressions, such as that shown in the next example, should be avoided if at all possible. Such expressions are difficult to read, and they are often written incorrectly because of unbalanced parentheses.

EXAMPLE 3.9 Consider the arithmetic expression

```
2 * ((i % 5) * (4 + (j - 3) / (k + 2)))
```

where `i`, `j` and `k` are integer variables. If these variables are assigned the values 8, 15 and 4, respectively, then the given expression would be evaluated as

$$2 \times ((8 \,\%\, 5) \times (4 + (15 - 3) / (4 + 2))) = 2 \times (3 \times (4 + (12/6))) = 2 \times (3 \times (4 + 2)) = 2 \times (3 \times 6) = 2 \times 18 = 36$$

Suppose the value of this expression will be assigned to the integer variable `w`; i.e.,

```
w = 2 * ((i % 5) * (4 + (j - 3) / (k + 2)));
```

It is generally better to break this long arithmetic expression up into several shorter expressions, such as

```
u = i % 5;
v = 4 + (j - 3) / (k + 2);
w = 2 * (u * v);
```

where `u` and `v` are integer variables. These equivalent expressions are much more likely to be written correctly than the original lengthy expression.

Assignment expressions will be discussed in greater detail in Sec. 3.4.

3.2 UNARY OPERATORS

C includes a class of operators that act upon a single operand to produce a new value. Such operators are known as *unary operators*. Unary operators usually precede their single operands, though some unary operators are written after their operands.

Perhaps the most common unary operation is *unary minus*, where a numerical constant, variable or expression is preceded by a minus sign. (Some programming languages allow a minus sign to be included as a part of a numeric constant. In C, however, all numeric constants are positive. Thus, a negative number is actually an expression, consisting of the unary minus operator, followed by a positive numeric constant.)

Note that the unary minus operation is distinctly different from the arithmetic operator which denotes subtraction (-). The subtraction operator requires two separate operands.

EXAMPLE 3.10 Here are several examples which illustrate the use of the unary minus operation.

```
-743          -0X7FFF       -0.2          -5E-8

-root1        -(x + y)      -3 * (x + y)
```

In each case the minus sign is followed by a numerical operand which may be an integer constant, a floating-point constant, a numeric variable or an arithmetic expression.

There are two other commonly used unary operators: The *increment operator*, ++, and the *decrement operator*, --. The increment operator causes its operand to be increased by 1, whereas the decrement operator causes its operand to be decreased by 1. The operand used with each of these operators must be a single variable.

EXAMPLE 3.11 Suppose that `i` is an integer variable that has been assigned a value of 5. The expression `++i`, which is equivalent to writing `i = i + 1`, causes the value of `i` to be increased to 6. Similarly, the expression `--i`, which is equivalent to `i = i - 1`, causes the (original) value of `i` to be decreased to 4.

The increment and decrement operators can each be utilized two different ways, depending on whether the operator is written before or after the operand. If the operator precedes the operand (e.g., `++i`), then the operand will be altered in value *before* it is utilized for its intended purpose within the program. If, however, the operator *follows* the operand (e.g., `i++`), then the value of the operand will be altered *after* it is utilized.

EXAMPLE 3.12 A C program includes an integer variable `i` whose initial value is 1. Suppose the program includes the following three `printf` statements. (See Example 1.6 for a brief explanation of the `printf` statement.)

```
printf("i = %d\n", i);
printf("i = %d\n", ++i);
printf("i = %d\n", i);
```

These `printf` statements will generate the following three lines of output. (Each `printf` statement will generate one line.)

```
i = 1
i = 2
i = 2
```

The first statement causes the original value of `i` to be displayed. The second statement increments `i` and then displays its value. The final value of `i` is displayed by the last statement.

Now suppose that the program includes the following three `printf` statements, rather than the three statements given above.

```
printf("i = %d\n", i);
printf("i = %d\n", i++);
printf("i = %d\n", i);
```

The first and third statements are identical to those shown above. In the second statement, however, the unary operator follows the integer variable rather than precedes it.

These statements will generate the following three lines of output.

```
i = 1
i = 1
i = 2
```

The first statement causes the original value of `i` to be displayed, as before. The second statement causes the current value of `i` (1) to be displayed and then incremented (to 2). The final value of `i` (2) is displayed by the last statement.

We will say much more about the use of the `printf` statement in Chap. 4. For now, simply note the distinction between the expression `++i` in the first group of statements, and the expression `i++` in the second group.

Another unary operator that is worth mentioning at this time is the `sizeof` operator. This operator returns the size of its operand, in bytes. The `sizeof` operator always precedes its operand. The operand may be an expression, or it may be a cast.

Elementary programs rarely make use of the `sizeof` operator. However, this operator allows a determination of the number of bytes allocated to various types of data items. This information can be very useful when transferring a program to a different computer or to a new version of C. It is also used for dynamic memory allocation, as explained in Sec. 10.4.

EXAMPLE 3.13 Suppose that `i` is an integer variable, `x` is a floating-point variable, `d` is a double-precision variable, and `c` is a character-type variable. The statements

```
printf("integer: %d\n", sizeof i);
printf("float: %d\n", sizeof x);
printf("double: %d\n", sizeof d);
printf("character: %d\n", sizeof c);
```

might generate the following output.

```
integer: 2
float: 4
double: 8
character: 1
```

Thus, we see that this version of C allocates 2 bytes to each integer quantity, 4 bytes to each floating-point quantity, 8 bytes to each double-precision quantity, and 1 byte to each character. These values may vary from one version of C to another, as explained in Sec. 2.3.

Another way to generate the same information is to use a cast rather than a variable within each `printf` statement. Thus, the `printf` statements could have been written as

```
printf("integer: %d\n", sizeof (integer));
printf("float: %d\n", sizeof (float));
printf("double: %d\n", sizeof (double));
printf("character: %d\n", sizeof (char));
```

These `printf` statements will generate the same output as that shown above. Note that each cast is enclosed in parentheses, as described in Sec. 3.1.

Finally, consider the array declaration

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

The statement

```
printf("Number of characters = %d", sizeof text);
```

will generate the following output.

```
Number of characters = 11
```

Thus we see that the array `text` contains 11 characters, as explained in Example 2.26.

A *cast* is also considered to be a unary operator (see Example 3.5 and the preceding discussion). In general terms, a reference to the cast operator is written as (*type*). Thus, the unary operators that we have encountered so far in this book are –, `++`, `--`, `sizeof` and (*type*).

Unary operators have a higher precedence than arithmetic operators. Hence, if a unary minus operator acts upon an arithmetic expression that contains one or more arithmetic operators, the unary minus operation will be carried out first (unless, of course, the arithmetic expression is enclosed in parentheses). Also, the associativity of the unary operators is right to left, though consecutive unary operators rarely appear in elementary programs.

EXAMPLE 3.14 Suppose that x and y are integer variables whose values are 10 and 20, respectively. The value of the expression `-x + y` will be $-10 + 20 = 10$. Note that the unary minus operation is carried out before the addition.

Now suppose that parentheses are introduced, so that the expression becomes `-(10 + 20)`. The value of this expression is $-(10 + 20) = -30$. Note that the addition now *precedes* the unary minus operation.

C includes several other unary operators. They will be discussed in later sections of this book, as the need arises.

3.3 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

When carrying out relational and equality operations, operands that differ in type will be converted in accordance with the rules discussed in Sec. 3.1.

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.

Each of the logical operators falls into its own precedence group. *Logical and* has a higher precedence than *logical or*. Both precedence groups are lower than the group containing the equality operators. The associativity is left to right. The precedence groups are summarized below.

C also includes the unary operator `!` that negates the value of a logical expression; i.e., it causes an expression that is originally true to become false, and vice versa. This operator is referred to as the *logical negation* (or *logical not*) operator.

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

A more complete listing is given in Table 3-1, later in this chapter.

EXAMPLE 3.19 Consider once again the variables `i`, `f` and `c`, as described in Examples 3.16 and 3.17; i.e., `i` = 7, `f` = 5.5 and `c` = `'w'`. Some logical expressions that make use of these variables are shown below.

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

Each of these expressions has been presented before (the first in Example 3.16, and the other two in Example 3.17), though pairs of parentheses were included in the previous examples. The parentheses are not necessary because of the natural operator precedences. Thus, the arithmetic operations will automatically be carried out before the relational or equality operations, and the relational and equality operations will automatically be carried out before the logical connectives.

Consider the last expression in particular. The first operation to be carried out will be addition (i.e., `i + f`); then the relational comparison (i.e., `i + f <= 10`); then the equality comparison (i.e., `c != 'p'`); and finally, the *logical or* condition.

Complex logical expressions that consist of individual logical expressions joined together by the logical operators && and || are evaluated left to right, but only until the overall true/false value has been established. Thus, a complex logical expression will not be evaluated in its entirety if its value can be established from its constituent operands.

EXAMPLE 3.20 Consider the complex logical expression shown below.

```
error > .0001 && count < 100
```

If `error > .0001` is false, then the second operand (i.e., `count < 100`) will not be evaluated, because the entire expression will be considered false.

On the other hand, suppose the expression had been written

```
error > .0001 || count < 100
```

If `error > .0001` is true, then the entire expression will be true. Hence, the second operand will not be evaluated. If `error > .0001` is false, however, then the second expression (i.e., `count < 100`) must be evaluated to determine if the entire expression is true or false.

3.4 ASSIGNMENT OPERATORS

There are several different assignment operators in C. All of them are used to form *assignment expressions*, which assign the value of an expression to an identifier.

The most commonly used assignment operator is =. Assignment expressions that make use of this operator are written in the form

identifier = *expression*

where *identifier* generally represents a variable, and *expression* represents a constant, a variable or a more complex expression.

EXAMPLE 3.21 Here are some typical assignment expressions that make use of the = operator.

```
a = 3
x = y
delta = 0.001
sum = a + b
area = length * width
```

The first assignment expression causes the integer value 3 to be assigned to the variable `a`, and the second assignment causes the value of `y` to be assigned to `x`. In the third assignment, the floating-point value `0.001` is assigned to `delta`. The last two assignments each result in the value of an arithmetic expression being assigned to a variable (i.e., the value of `a + b` is assigned to `sum`, and the value of `length * width` is assigned to `area`).

Remember that the *assignment operator* = and the *equality operator* == are *distinctly different*. The assignment operator is used to assign a value to an identifier, whereas the equality operator is used to determine if two expressions have the same value. These operators cannot be used in place of one another. Beginning programmers often incorrectly use the assignment operator when they want to test for equality. This results in a logical error that is usually difficult to detect.

Assignment expressions are often referred to as *assignment statements*, since they are usually written as complete statements. However, assignment expressions can also be written as expressions that are included within other statements (more about this in later chapters).

If the two operands in an assignment expression are of different data types, then the value of the expression on the right (i.e., the right-hand operand) will automatically be converted to the type of the identifier on the left. The entire assignment expression will then be of this same data type.

Under some circumstances, this automatic type conversion can result in an alteration of the data being assigned. For example:

- A floating-point value may be truncated if assigned to an integer identifier.
- A double-precision value may be rounded if assigned to a floating-point (single-precision) identifier.
- An integer quantity may be altered if assigned to a shorter integer identifier or to a character identifier (some high-order bits may be lost).

Moreover, the value of a character constant assigned to a numeric-type identifier will be dependent upon the particular character set in use. This may result in inconsistencies from one version of C to another.

The careless use of type conversions is a frequent source of error among beginning programmers.

EXAMPLE 3.22 In the following assignment expressions, suppose that `i` is an integer-type variable.

Expression	*Value*
`i = 3.3`	3
`i = 3.9`	3
`i = -3.9`	-3

Now suppose that `i` and `j` are both integer-type variables, and that `j` has been assigned a value of 5. Several assignment expressions that make use of these two variables are shown below.

Expression	*Value*	
`i = j`	5	
`i = j / 2`	2	
`i = 2 * j / 2`	5	(left-to-right associativity)
`i = 2 * (j / 2)`	4	(truncated division, followed by multiplication)

Finally, assume that i is an integer-type variable, and that the ASCII character set applies.

Expression	*Value*
`i = 'x'`	120
`i = '0'`	48
`i = ('x' - '0') / 3`	24
`i = ('y' - '0') / 3`	24

Multiple assignments of the form

identifier 1 = *identifier 2* = ··· = *expression*

are permissible in C. In such situations, the assignments are carried out from right to left. Thus, the multiple assignment

identifier 1 = *identifier 2* = *expression*

is equivalent to

identifier 1 = (*identifier 2* = *expression*)

and so on, with right-to-left nesting for additional multiple assignments.

EXAMPLE 3.23 Suppose that `i` and `j` are integer variables. The multiple assignment expression

```
i = j = 5
```

will cause the integer value 5 to be assigned to both `i` and `j`. (To be more precise, 5 is first assigned to `j`, and the value of `j` is then assigned to `i`.)

Similarly, the multiple assignment expression

```
i = j = 5.9
```

will cause the integer value 5 to be assigned to both `i` and `j`. Remember that truncation occurs when the floating-point value 5.9 is assigned to the integer variable `j`.

C contains the following five additional assignment operators: `+=`, `-=`, `*=`, `/=` and `%=`. To see how they are used, consider the first operator, `+=`. The assignment expression

```
expression 1 += expression 2
```

is equivalent to

```
expression 1 = expression 1 + expression 2
```

Similarly, the assignment expression

```
expression 1 -= expression 2
```

is equivalent to

```
expression 1 = expression 1 - expression 2
```

and so on for all five operators.

Usually, *expression 1* is an identifier, such as a variable or an array element.

EXAMPLE 3.24 Suppose that `i` and `j` are integer variables whose values are 5 and 7, and `f` and `g` are floating-point variables whose values are 5.5 and -3.25. Several assignment expressions that make use of these variables are shown below. Each expression utilizes the *original* values of `i`, `j`, `f` and `g`.

Expression	*Equivalent Expression*	*Final value*
`i += 5`	`i = i + 5`	10
`f -= g`	`f = f - g`	8.75
`j *= (i - 3)`	`j = j * (i - 3)`	14
`f /= 3`	`f = f / 3`	1.833333
`i %= (j - 2)`	`i = i % (j - 2)`	0

Assignment operators have a lower precedence than any of the other operators that have been discussed so far. Therefore unary operations, arithmetic operations, relational operations, equality operations and logical operations are all carried out before assignment operations. Moreover, the assignment operations have a right-to-left associativity.

The hierarchy of operator precedences presented in the last section can now be modified as follows to include assignment operators.

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
assignment operators	= += -= *= /= %=	R → L

See Table 3-1 later in this chapter for a more complete listing.

EXAMPLE 3.25 Suppose that x, y and z are integer variables which have been assigned the values 2, 3 and 4, respectively. The expression

```
x *= -2 * (y + z) / 3
```

is equivalent to the expression

```
x = x * (-2 * (y + z) / 3)
```

Either expression will cause the value –8 to be assigned to x.

Consider the order in which the operations are carried out in the first expression. The arithmetic operations precede the assignment operation. Therefore the expression (y + z) will be evaluated first, resulting in 7. Then the value of this expression will be multiplied by –2, yielding –14. This product will then be divided by 3 and truncated, resulting in –4. Finally, this truncated quotient is multiplied by the original value of x (i.e., 2) to yield the final result of –8.

Note that all of the explicit arithmetic operations are carried out before the final multiplication and assignment are made.

C contains other assignment operators, in addition to those discussed above. We will discuss them in Chap. 13.

3.5 THE CONDITIONAL OPERATOR

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.

If the operands (i.e., *expression 2* and *expression 3*) differ in type, then the resulting data type of the conditional expression will be determined by the rules given in Sec. 3.1.

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.

EXAMPLE 3.28 Here is an assignment statement that contains a conditional expression on the right-hand side.

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

If the value of `i` is negative, then 0 will be assigned to `flag`. If `i` is not negative, however, then 100 will be assigned to `flag`.

Here is another assignment statement that contains a conditional expression on the right-hand side.

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

This statement causes the value of the smaller of `f` and `g` to be assigned to `min`.

The conditional operator has its own precedence, just above the assignment operators. The associativity is right to left.

Table 3-1 summarizes the precedences for all of the operators discussed in this chapter.

Table 3-1 Operator Precedence Groups

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
conditional operator	? :	R → L
assignment operators	= += -= *= /= %=	R → L

A complete listing of all C operators, which is more extensive than that given in Table 3-1, is shown in Appendix C.

EXAMPLE 3.29 In the following assignment statement, a, b and c are assumed to be integer variables. The statement includes operators from six different precedence groups.

```
c += (a > 0 && a <= 10) ? ++a : a/b;
```

The statement begins by evaluating the complex expression

```
(a > 0 && a <= 10)
```

If this expression is true, the expression ++a is evaluated. Otherwise, the expression a/b is evaluated. Finally, the assignment operation (+=) is carried out, causing the value of c to be increased by the value of the conditional expression.

If, for example, a, b and c have the values 1, 2 and 3, respectively, then the value of the conditional expression will be 2 (because the expression ++a will be evaluated), and the value of c will increase to 5 (c = 3 + 2). On the other hand, if a, b and c have the values 50, 10 and 20, respectively, then the value of the conditional expression will be 5 (because the expression a/b will be evaluated), and the value of c will increase to 25 (c = 20 + 5).

3.6 LIBRARY FUNCTIONS

The C language is accompanied by a number of *library functions* that carry out various commonly used operations or calculations. These library functions are not a part of the language per se, though all implementations of the language include them. Some functions return a data item to their access point; others indicate whether a condition is true or false by returning a 1 or a 0, respectively; still others carry out specific operations on data items but do not return anything. Features which tend to be computer-dependent are generally written as library functions.

For example, there are library functions that carry out standard input/output operations (e.g., read and write characters, read and write numbers, open and close files, test for end of file, etc.), functions that perform operations on characters (e.g., convert from lower- to uppercase, test to see if a character is uppercase, etc.), functions that perform operations on strings (e.g., copy a string, compare strings, concatenate strings, etc.), and functions that carry out various mathematical calculations (e.g., evaluate trigonometric, logarithmic and exponential functions, compute absolute values, square roots, etc.). Other kinds of library functions are also available.

Library functions that are functionally similar are usually grouped together as (compiled) object programs in separate library files. These library files are supplied as a part of each C compiler. All C compilers contain similar groups of library functions, though they lack precise standardization. Thus there may be some variation in the library functions that are available in different versions of the language.

A typical set of library functions will include a fairly large number of functions that are common to most C compilers, such as those shown in Table 3-2 below. Within this table, the column labeled "type" refers to the data type of the quantity that is returned by the function. The *void* entry shown for function `srand` indicates that nothing is returned by this function.

A more extensive list, which includes all of the library functions that appear in the programming examples presented in this book, is shown in Appendix H. For complete list, see the programmer's reference manual that accompanies your particular version of C.

A library function is accessed simply by writing the function name, followed by a list of *arguments* that represent information being passed to the function. The arguments must be enclosed in parentheses and separated by commas. The arguments can be constants, variable names, or more complex expressions. The parentheses must be present, even if there are no arguments.

A function that returns a data item can appear anywhere within an expression, in place of a constant or an identifier (i.e., in place of a variable or an array element). A function that carries out operations on data items but does not return anything can be accessed simply by writing the function name, since this type of function reference constitutes an expression statement.

Table 3-2 Some Commonly Used Library Functions

Function	*Type*	*Purpose*
`abs(i)`	int	Return the absolute value of `i`.
`ceil(d)`	double	Round up to the next integer value (the smallest integer that is greater than or equal to `d`).
`cos(d)`	double	Return the cosine of `d`.
`cosh(d)`	double	Return the hyperbolic cosine of `d`.
`exp(d)`	double	Raise *e* to the power `d` ($e = 2.7182818 \cdots$ is the base of the natural (Naperian) system of logarithms).
`fabs(d)`	double	Return the absolute value of `d`.
`floor(d)`	double	Round down to the next integer value (the largest integer that does not exceed `d`).
`fmod(d1,d2)`	double	Return the remainder (i.e., the noninteger part of the quotient) of `d1/d2`, with same sign as `d1`.
`getchar()`	int	Enter a character from the standard input device.
`log(d)`	double	Return the natural logarithm of `d`.
`pow(d1,d2)`	double	Return `d1` raised to the `d2` power.
`printf(...)`	int	Send data items to the standard output device (arguments are complicated – see Chap. 4).
`putchar(c)`	int	Send a character to the standard output device.
`rand()`	int	Return a random positive integer.
`sin(d)`	double	Return the sine of `d`.
`sqrt(d)`	double	Return the square root of `d`.
`srand(u)`	void	Initialize the random number generator.
`scanf(...)`	int	Enter data items from the standard input device (arguments are complicated – see Chap. 4).
`tan(d)`	double	Return the tangent of `d`.
`toascii(c)`	int	Convert value of argument to ASCII.
`tolower(c)`	int	Convert letter to lowercase.
`toupper(c)`	int	Convert letter to uppercase.

Note: *Type* refers to the data type of the quantity that is returned by the function.
`c` denotes a character-type argument
`i` denotes an integer argument
`d` denotes a double-precision argument
`u` denotes an unsigned integer argument

EXAMPLE 3.30 Shown below is a portion of a C program that solves for the roots of the quadratic equation

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

using the well-known quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

This program uses the `sqrt` library function to evaluate the square root.

```
main()   /* solution of a quadratic equation */
{
    double a,b,c,root,x1,x2;

    /* read values for a, b and c */

    root = sqrt(b * b - 4 * a * c);
    x1 = (-b + root) / (2 * a);
    x2 = (-b - root) / (2 * a);

    /* display values for a, b, c, x1 and x2 */
}
```

In order to use a library function it may be necessary to include certain specific information within the main portion of the program. For example, forward function declarations and symbolic constant definitions are usually required when using library functions (see Secs. 7.3, 8.5 and 8.6). This information is generally stored in special files which are supplied with the compiler. Thus, the required information can be obtained simply by accessing these special files. This is accomplished with the preprocessor statement `#include`; i.e.,

```
#include <filename>
```

where *filename* represents the name of a special file.

The names of these special files are specified by each individual implementation of C, though there are certain commonly used file names such as `stdio.h`, `stdlib.h` and `math.h`. The suffix "h" generally designates a "header" file, which indicates that it is to be included at the beginning of the program. (Header files are discussed in Sec. 8.6.)

Note the similarity between the preprocessor statement `#include` and the preprocessor statement `#define`, which was discussed in Sec. 2.9.

EXAMPLE 3.31 Lowercase to Uppercase Character Conversion Here is a complete C program that reads in a lowercase character, converts it to uppercase and then displays the uppercase equivalent.

```
/* read a lowercase character and display its uppercase equivalent */

#include <stdio.h>
#include <ctype.h>

main()
{
    int lower, upper;

    lower = getchar();
    upper = toupper(lower);
    putchar(upper);
}
```

This program contains three library functions: `getchar`, `toupper` and `putchar`. The first two functions each return a single character (`getchar` returns a character that is entered from the keyboard, and `toupper` returns the uppercase equivalent of its argument). The last function (`putchar`) causes the value of the argument to be displayed. Notice that the last two functions each have one argument but the first function does not have any arguments, as indicated by the empty parentheses.

Also, notice the preprocessor statements `#include <stdio.h>` and `#include <ctype.h>`, which appear at the start of the program. These statements cause the contents of the files `stdio.h` and `ctype.h` to be inserted into the program the compilation process begins. The information contained in these files is essential for the proper functioning of the library functions `getchar`, `putchar` and `toupper`.

Review Questions

3.1 What is an expression? What are its components?

3.2 What is an operator? Describe several different types of operators that are included in C.

3.3 What is an operand? What is the relationship between operators and operands?

3.4 Describe the five arithmetic operators in C. Summarize the rules associated with their use.

3.5 Summarize the rules that apply to expressions whose operands are of different types.

3.6 How can the value of an expression be converted to a different data type? What is this called?

3.7 What is meant by operator precedence? What are the relative precedences of the arithmetic operators?

3.8 What is meant by associativity? What is the associativity of the arithmetic operators?

3.9 When should parentheses be included within an expression? When should the use of parentheses be avoided?

3.10 In what order are the operations carried out within an expression that contains nested parentheses?

3.11 What are unary operators? How many operands are associated with a unary operator?

3.12 Describe the six unary operators discussed in this chapter. What is the purpose of each?

3.13 Describe two different ways to utilize the increment and decrement operators. How do the two methods differ?

3.14 What is the relative precedence of the unary operators compared with the arithmetic operators? What is their associativity?

3.15 How can the number of bytes allocated to each data type be determined for a particular C compiler?

3.16 Describe the four relational operators included in C. With what type of operands can they be used? What type of expression is obtained?

3.17 Describe the two equality operators included in C. How do they differ from the relational operators?

3.18 Describe the two logical operators included in C. What is the purpose of each? With what type of operands can they be used? What type of expression is obtained?

3.19 What are the relative precedences of the relational, equality and logical operators with respect to one another and with respect to the arithmetic and unary operators? What are their associativities?

3.20 Describe the *logical not* (logical negation) operator. What is its purpose? Within which precedence group is it included? How many operands does it require? What is its associativity?

3.21 Describe the six assignment operators discussed in this chapter. What is the purpose of each?

3.22 How is the type of an assignment expression determined when the two operands are of different data types? In what sense is this situation sometimes a source of programming errors?

3.23 How can multiple assignments be written in C? In what order will the assignments be carried out?

3.24 What is the precedence of assignment operators relative to other operators? What is their associativity?

3.25 Describe the use of the conditional operator to form conditional expressions. How is a conditional expression evaluated?

3.26 How is the type of a conditional expression determined when its operands differ in type?

3.27 How can the conditional operator be combined with the assignment operator to form an "if - else" type statement?

3.28 What is the precedence of the conditional operator relative to the other operators described in this chapter? What is its associativity?

3.29 Describe, in general terms, the kinds of operations and calculations that are carried out by the C library functions.

3.30 Are the library functions actually a part of the C language? Explain.

3.31 How are the library functions usually packaged within a C compiler?

3.32 How are library functions accessed? How is information passed to a library function from the access point?

3.33 What are arguments? How are arguments written? How is a call to a library function written if there are no arguments?

3.34 How is specific information that may be required by the library functions stored? How is this information entered into a C program?

3.35 In what general category do the `#define` and `#include` statements fall?

Problems

3.36 Suppose a, b and c are integer variables that have been assigned the values a = 8, b = 3 and c = –5. Determine the value of each of the following arithmetic expressions.

(*a*) `a + b + c`
(*b*) `2 * b + 3 * (a - c)`
(*c*) `a / b`
(*d*) `a % b`
(*e*) `a / c`
(*f*) `a % c`
(*g*) `a * b / c`
(*h*) `a * (b / c)`
(*i*) `(a * c) % b`
(*j*) `a * (c % b)`

3.37 Suppose x, y and z are floating-point variables that have been assigned the values x = 8.8, y = 3.5 and z = –5.2. Determine the value of each of the following arithmetic expressions.

(*a*) `x + y + z`
(*b*) `2 * y + 3 * (x - z)`
(*c*) `x / y`
(d) `x % y`
(*e*) `x / (y + z)`
(*f*) `(x / y) + z`
(*g*) `2 * x / 3 * y`
(*h*) `2 * x / (3 * y)`

3.38 Suppose c1, c2 and c3 are character-type variables that have been assigned the characters E, 5 and ?, respectively. Determine the numerical value of the following expressions, based upon the ASCII character set (see Table 2-1).

(*a*) `c1`
(*b*) `c1 - c2 + c3`
(*c*) `c2 - 2`
(*d*) `c2 - '2'`
(*e*) `c3 + '#'`
(*f*) `c1 % c3`
(*g*) `'2' + '2'`
(*h*) `(c1 / c2) * c3`
(*i*) `3 * c2`
(*j*) `'3' * c2`

3.39 A C program contains the following declarations:

```
int i, j;
long ix;
short s;
float x;
double dx;
char c;
```

Determine the data type of each of the following expressions.

(*a*) `i + c`
(*b*) `x + c`
(*c*) `dx + x`
(*d*) `((int) dx) + ix`
(*e*) `i + x`
(*f*) `s + j`
(*g*) `ix + j`
(*h*) `s + c`
(*i*) `ix + c`

3.40 A C program contains the following declarations and initial assignments:

```
int i = 8, j = 5;
float x = 0.005, y = -0.01;
char c = 'c', d = 'd';
```

Determine the value of each of the following expressions. Use the values initially assigned to the variables for each expression.

(*a*) `(3 * i - 2 * j) % (2 * d - c)`
(*b*) `2 * ((i / 5) + (4 * (j - 3)) % (i + j - 2))`

(*c*) `(i - 3 * j) % (c + 2 * d) / (x - y)`

(*d*) `-(i + j)`

(*e*) `++i`

(*f*) `i++`

(*g*) `--j`

(*h*) `++x`

(*i*) `y--`

(*j*) `i <= j`

(*k*) `c > d`

(*l*) `x >= 0`

(*m*) `x < y`

(*n*) `j != 6`

(*o*) `c == 99`

(*p*) `5 * (i + j) > 'c'`

(*q*) `(2 * x + y) == 0`

(*r*) `2 * x + (y == 0)`

(*s*) `2 * x + y == 0`

(*t*) `!(i <= j)`

(*u*) `!(c == 99)`

(*v*) `!(x > 0)`

(*w*) `(i > 0) && (j < 5)`

(*x*) `(i > 0) || (j < 5)`

(*y*) `(x > y) && (i > 0) || (j < 5)`

(*z*) `(x > y) && (i > 0) && (j < 5)`

3.41 A C program contains the following declarations and initial assignments:

```
int i = 8, j = 5, k;
float x = 0.005, y = -0.01, z;
char a, b, c = 'c', d = 'd';
```

Determine the value of each of the following assignment expressions. Use the values originally assigned to the variables for each expression.

(*a*) `k = (i + j)`

(*b*) `z = (x + y)`

(*c*) `i = j`

(*d*) `k = (x + y)`

(*e*) `k = c`

(*f*) `z = i / j`

(*g*) `a = b = d`

(*h*) `i = j = 1.1`

(*i*) `z = k = x`

(*j*) `k = z = x`

(*k*) `i += 2`

(*l*) `y -= x`

(*m*) `x *= 2`

(*n*) `i /= j`

(*o*) `i %= j`

(*p*) `i += (j - 2)`

(*q*) `k = (j == 5) ? i : j`

(*r*) `k = (j > 5) ? i : j`

(*s*) `z = (x >= 0) ? x : 0`

(*t*) `z = (y >= 0) ? y : 0`

(*u*) `a = (c < d) ? c : d`

(*v*) `i -= (j > 0) ? j : 0`

3.42 Each of the following expressions involves the use of a library function. Identify the purpose of each expression. (See Appendix H for an extensive list of library functions.)

(*a*) `abs(i - 2 * j)`
(*b*) `fabs(x + y)`
(*c*) `isprint(c)`
(*d*) `isdigit(c)`
(*e*) `toupper(d)`
(*f*) `ceil(x)`
(*g*) `floor(x + y)`
(*h*) `islower(c)`
(*i*) `isupper(j)`
(*j*) `exp(x)`
(*k*) `log(x)`
(*l*) `sqrt(x*x + y*y)`
(*m*) `isalnum(10 * j)`
(*n*) `isalpha(10 * j)`
(*o*) `isascii(10 * j)`
(*p*) `toascii(10 * j)`
(*q*) `fmod(x, y)`
(*r*) `tolower(65)`
(*s*) `pow(x - y, 3.0)`
(*t*) `sin(x - y)`
(*u*) `strlen("hello\0")`
(*v*) `strpos("hello\0", 'e')`

3.43 A C program contains the following declarations and initial assignments:

```
int i = 8, j = 5;
double x = 0.005, y = -0.01;
char c = 'c', d = 'd';
```

Determine the value of each of the following expressions, which involve the use of library functions. (See Appendix H for an extensive list of library functions.)

(*a*) `abs(i - 2 * j)`
(*b*) `fabs(x + y)`
(*c*) `isprint(c)`
(*d*) `isdigit(c)`
(*e*) `toupper(d)`
(*f*) `ceil(x)`
(*g*) `ceil(x + y)`
(*h*) `floor(x)`
(*i*) `floor(x + y)`
(*j*) `islower(c)`
(*k*) `isupper(j)`
(*l*) `exp(x)`
(*m*) `log(x)`
(*n*) `log(exp(x))`
(*o*) `sqrt(x*x + y*y)`
(*p*) `isalnum(10 * j)`
(*q*) `isalpha(10 * j)`
(*r*) `isascii(10 * j)`
(*s*) `toascii(10 * j)`
(*t*) `fmod(x, y)`
(*u*) `tolower(65)`
(*v*) `pow(x - y, 3.0)`
(*w*) `sin(x - y)`
(*x*) `strlen("hello\0")`
(*y*) `strpos("hello\0", 'e')`
(*z*) `sqrt(sin(x) + cos(y))`

3.44 Determine which of the library functions shown in Appendix H are available for your particular version of C. Are some of the functions available under a different name? What header files are required?