Chapter 3
Structure of a C Program

Two features set the C language apart from many other languages: expressions and pointers. Both of these concepts lie at the very heart of the language, giving C its unique look and feel.

This chapter explores the first of these concepts: expressions. Expressions are not new to you; you have used them in mathematics. However, C's use of expressions is unique to the C language.

Closely tied to the concept of expressions are operators, precedence and associativity, and statements, all of which are discussed in this chapter. This chapter also introduces a concept known as side effects and explains in detail how it affects statements in C.

Objectives

- ❑ To be able to list and describe the six expression categories
- ❑ To understand the rules of precedence and associativity in evaluating expressions
- ❑ To understand the result of side effects in expression evaluation
- ❑ To be able to predict the results when an expression is evaluated
- ❑ To understand implicit and explicit type conversion
- ❑ To understand and use the first four statement types: null, expression, return, and compound

3.1 Expressions

An **expression** is a sequence of operands and operators that reduces to a single value. Expressions can be simple or complex. An **operator** is a syntactical token that requires an action be taken. An **operand** is an object on which an operation is performed; it receives an operator's action.

A **simple expression** contains only one operator. For example 2 + 5 is a simple expression whose value is 7; similarly, -a is a simple expression. A **complex expression** contains more that one operator. An example of a complex expression is 2 + 5 * 7. To evaluate a complex expression, we reduce it to a series of simple expressions. In the previous example, we first evaluate the simple expression 5 * 7 (35) and then the expression 2 + 35, giving a result of 37.

Every language has operators whose actions are clearly specified in the language syntax. The order in which the operators in a complex expression are evaluated is determined by a set of priorities known as **precedence**; the higher the precedence, the earlier the expression containing the operator is evaluated. The table inside the front cover contains the precedence of each operator. Looking at that table, we see that in the expression 2 + 5 * 7, multiplication has a higher priority than addition so the multiply expression is evaluated first. We discuss precedence in more detail in the next section.

If two operators with the same precedence occur in a complex expression, another attribute of an operator, its associativity, takes control. **Associativity** is the parsing direction used to evaluate an expression. It can be either left-to-right or right-to-left. When two operators with the same precedence occur in an expression and their associativity is left-to-right, the left operator is evaluated first. For example, in the expression 3 * 4 / 6, there are two operators, multiplication and division, with the same precedence and left-to-right associativity. Therefore, the multiplication is evaluated before the division. We also discuss associativity in more detail in the next section.

An expression always reduces to a single value.

We can divide simple expressions into six categories based on the number of operands, relative positions of the operand and operator, and the precedence of operator. Figure 3-1 shows the categories.

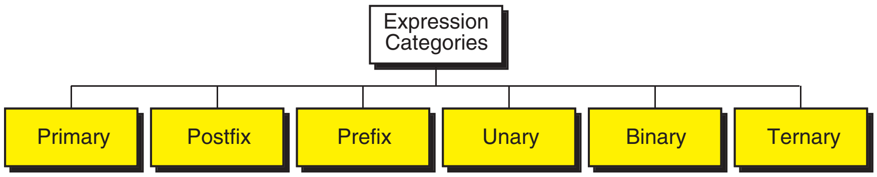


FIGURE 3-1 Expression Categories

Primary Expressions

The most elementary type of expression is a primary expression. A **primary expression** consists of only one operand with no operator. In C, the operand in the primary expression can be a name, a constant, or a parenthesized expression. Although a primary expression has no operator, the null operator in this expression has the precedence of 16 according to the table of precedence. In other words, a primary expression is evaluated first in a complex expression.

Names

A **name** is any identifier for a variable, a function, or any other object in the language. The following are examples of some names used as primary expressions:

```
a     b12     price     calc     INT_MAX     SIZE
```

Literal Constants

The second type of primary expression is the literal constant. As discussed in Chapter 2, a constant is a piece of data whose value can't change during the execution of the program. The following are examples of literal constants used as primary expressions:

```
5    123.98    'A'     "Welcome"
```

Parenthetical Expressions

The third type of primary expression is the parenthetical expression. Any value enclosed in parentheses must be reducible to a single value and is therefore a primary expression. This includes any of the complex expressions when they are enclosed in parentheses. Thus, a complex expression can be enclosed in parentheses to make it a primary expression. The following are primary expressions:

```
(2  *  3  +  4)      (a  =  23  +  b  *  6)
```

Postfix Expressions

The **postfix expression** consists of one operand followed by one operator. Its category is shown in Figure 3-2. There are several operators that create a postfix expression as you can see in the precedence table. We discuss only three of them here: function call, postfix increment, and postfix decrement.

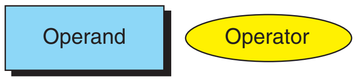


FIGURE 3-2 Postfix Expressions

Function Call

We have already used a postfix expression. In the hello world program, we wrote a message on the monitor using the *printf* function. Function calls are postfix expressions. The function name is the operand and the operator is the parentheses that follow the name. The parentheses may contain arguments or be empty. When present, the arguments are part of the operator.

Postfix Increment/Decrement

The postfix increment and postfix decrement are also postfix operators. Virtually all programs require somewhere in their code that the value 1 be added to a variable. In early languages, this additive operation could only be represented as a binary expression. C provides the same functionality in two expressions: postfix and prefix.

In the postfix increment, the variable is increased by 1. Thus, a++ results in the variable a being increased by 1. The effect is the same as a = a + 1.

(a++) *has the same effect as* (a = a + 1)

Although both the postfix increment (a++) and binary expression (a = a + 1) add 1 to the variable, there is a major difference. The *value* of the postfix increment expression is determined *before* the variable is increased. For instance, if the variable a contains 4 before the expression is evaluated, the value of the expression a++ is 4. As a result of evaluating the expression and its side effect, a contains 5. The value and side effect of the postfix increment are graphically shown in Figure 3-3.

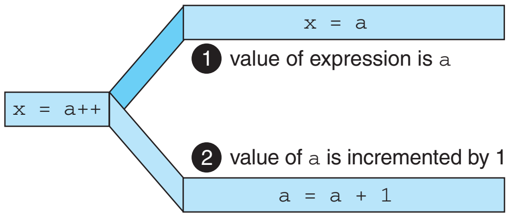


FIGURE 3-3 Result of Postfix a++

The postfix decrement (a--) also has a value and a side effect. As with the increment, the value of the expression is the value of a before the decrement; the side effect is the variable is decremented by 1.

The operand in a postfix expression must be a variable.

Program 3-1 demonstrates the effect of the postfix increment expression.

PROGRAM 3-1 Demonstrate Postfix Increment

```c
/* Example of postfix increment.
      Written by:
      Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   int a;

// Statements
   a = 4;
   printf("value of a     : %2d\n", a);
   printf("value of a++   : %2d\n",    a++);
   printf("new value of a: %2d\n\n", a);
   return 0;
}  // main
```

```
Results:
value of a     :  4
value of a++   :  4
new value of a:   5
```

Prefix Expressions

In prefix expressions, the operator comes before the operand as seen in Figure 3-4.

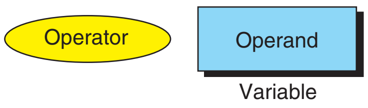


FIGURE 3-4 Prefix Expression

Prefix Increment/Decrement

In C, we have only two prefix operators that form prefix expressions: prefix increment and prefix decrement. Just like the postfix increment and postfix decrement operators, the prefix increment and prefix decrement operators are shorthand notations for adding or subtracting 1 from a variable.

The operand of a prefix expression must be a variable.

There is one major difference between the postfix and prefix operators, however: with the prefix operators, the effect takes place *before* the expression that contains the operator is evaluated. Note that this is the reverse of the postfix operation. Figure 3-5 shows the operation graphically.

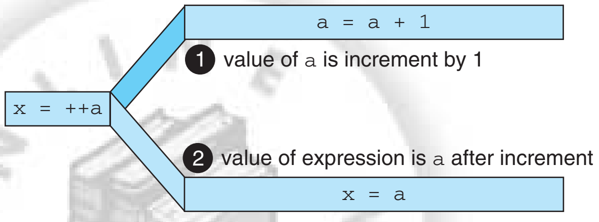

a = a + 1
1 value of a is increment by 1
x = ++a
2 value of expression is a after increment
x = a

FIGURE 3-5 Result of Prefix ++a

The effect of both the postfix and prefix increment is the same: The variable is incremented by 1. If we don't need the value of the expression—that is, if all we need is the effect of incrementing the value of a variable by 1—then it makes no difference which one we use. You will find that programmers use the postfix increment and decrement more often, if for no other reason than that the variable is shown first and is therefore easier to read.

`(++a)` has the same effect as `(a = a + 1)`

On the other hand, if we require both the value and the effect, then our application determines which one we need to use. When we need the value of the expression to be the *current value* of the variable, we use the postfix operator; when we need the value to be the *new value* of the variable (after it has been incremented or decremented), we use the prefix operator. Program 3-2 demonstrates the prefix increment expression. Study it carefully, and compare it to the results from Program 3-1.

PROGRAM 3-2 Demonstrate Prefix Increment

```
/* Example of prefix increment.
      Written by:
```

continued

PROGRAM 3-2 Demonstrate Prefix Increment *(continued)*

```
      Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   int a;

// Statements
   a = 4;
   printf("value of a    : %2d\n", a);
   printf("value of ++a  : %2d\n", ++a);
   printf("new value of a: %2d\n", a);
   return 0;
}  // main
```

```
Results:
value of a    :  4
value of ++a  :  5
new value of a:  5
```

Program 3-2 Analysis The only difference in the printouts between Program 3-1 and Program 3-2 is the use of the increment operators. The first program uses the postfix increment; the second uses the unary prefix increment. In both cases, we start with the same value for a and it has the same value at the end. But the value of the expression itself is different. To help remember the difference, use this rule: If the ++ is *before* the operand, the increment takes place *before* the expression is evaluated; if it is *after* the operand, the increment takes place *after* the expression is evaluated.

If ++ is after the operand, as in a++, the increment takes place after the expression is evaluated. If ++ is before the operand, as in ++a, the increment takes place before the expression is evaluated.

Unary Expressions

A **unary expression**, like a prefix expression, consist of one operator and one operand. Also like the prefix expression, the operator comes before the operand. Although prefix expressions and unary expressions look the same, they belong to different expression categories because the prefix expression needs a variable as the operand while the unary expression can have an expression or a variable as the operand. Many of the unary expressions are also familiar to you from mathematics and will require little explanation. In this chapter we discuss the *sizeof* operator, the plus/minus operators, and the cast operator. The others will be discussed in later chapters. The format of the unary expressions is demonstrated in Figure 3-6.

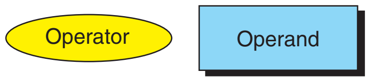


FIGURE 3-6 Unary Expressions

sizeof

The sizeof operator tells us the size, in bytes, of a type or a primary expression. By specifying the size of an object during execution, we make our program more portable to other hardware. A simple example will illustrate the point. On some personal computers, the size of the integer type is 2 bytes. On some mainframe computers, it is 4 bytes. On the very large supercomputers, it can be as large as 16 bytes. If it is important to know the exact size (in bytes) of an integer, we can use the *sizeof* operator with the integer type as shown below.

```
sizeof (int)
```

It is also possible to find the size of a primary expression. Here are two examples.

```
sizeof -345.23     sizeof x
```

Unary Plus/Minus

The unary plus and unary minus operators are what we think of as simply the plus and minus signs. In C, however, they are actually operators. Because they are operators, they can be used to compute the arithmetic value of an operand.

The plus operator does not change the value of the expression. If the expression's value is negative, it remains negative; if the expression's value is positive, it remains positive.

The minus operator changes the sign of a value algebraically—that is, to change it from plus to minus or minus to plus. Note, however, that the value of the stored variable is unchanged. The operation of these operators is seen in Table 3-1.

Expression	Contents of a Before *and* After Expression	Expression Value
+a	3	+3
-a	3	-3

continued

TABLE 3-1 Examples of Unary Plus And Minus Expressions

Expression	Contents of a Before *and* After Expression	Expression Value
+a	-5	-5
-a	-5	+5

TABLE 3-1 Examples of Unary Plus And Minus Expressions *(continued)*

Cast Operator

The third unary operator we discuss in this chapter is cast. The cast operator converts one expression type to another. For example, to convert an integer to a real number, we would use the following unary expression.

```
float (x)
```

It is important to note that, as with all of the unary operators, only the expression value is changed. The integer variable, x, is unchanged. We discuss the cast operator in detail in "Explicit Type Conversion (Cast)" in Section 3.5.

Binary Expressions

Binary expressions are formed by an operand-operator-operand combination. They are perhaps the most common expression category. Any two numbers added, subtracted, multiplied, or divided are usually formed in algebraic notation, which is a binary expression. There are many binary expressions. We cover the first two in this chapter. Figure 3-7 shows the format of a binary expression.

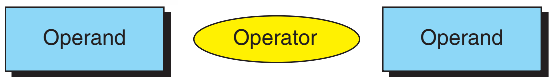


FIGURE 3-7 Binary Expressions

Multiplicative Expressions

The first binary expression we study, **multiplicative expressions**, which take its name from the first operator, include the multiply, divide, and modulus operators. These operators have the highest priority (13) among the binary operators and are therefore evaluated first among them.

The result of a multiply operator (*) is the product of the two operands. The operands can be any arithmetic type (integral or floating-point). The type of the result depends on the conversion rule that we discuss later in the chapter.

```
10 * 3                          // evaluates to 30
true * 4                        // evaluates to 4
'A' * 2                         // evaluates to 130
22.3  * 2                       // evaluates to 44.6
(2 + 3 * I) * (1 + 2 * I)       // evaluates to -4 + 7 * I
```

The result of a divide operator (/) depends on the type of the operands. If one or both operands is a floating-point type, the result is a floating-point quotient. If both operands are integral type, the result is the integral part of the quotient. The following shows some examples of the division operator.

```
10 / 3                              // evaluates to 3
true / 4                            // evaluates to 0
'A' / 2                             // evaluates to 32
22.3  / 2                           // evaluates to 11.15
```

Multiply and divide are well known, but you may not be familiar with the modulus operator (%), more commonly known as modulo. This operator divides the first operand by the second and returns the remainder rather than the quotient. Both operands must be integral types and the operator returns the remainder as an integer type. The following examples demonstrate the modulo operator.

```
10 % 3           // evaluates to 1
true % 4         // evaluates to 1
'A' % 10         // evaluates  to  5
22.3  % 2        // Error: Modulo cannot be floating-point
```

Because the division and modulus operators are related, they are often confused. Remember: The value of an expression with the division operator is the quotient; the value of a modulus operator is the remainder. Study the effect of these two operators in the following expressions:

```
3 / 5                                     // evaluates to 0
3 % 5                                     // evaluates to 3
```

Another important point to remember is, if the first integral operand is smaller than the second integral operand, and the result of division is 0, the result of the modulo operator is the first operand as shown below:

```
3 / 7                                    // evaluates to 0
3 % 7                                    // evaluates to 3
```

Both operands of the modulo operator (%) must be integral types.

Additive Expressions

In **additive expressions**, the second operand is added to or subtracted from the first operand, depending on the operator used. The operands in an additive expression can be any arithmetic types (integral or floating-point). Additive operators have lower precedence (12) than multiplicative operators (13); therefore, they are evaluated after multiplicative expressions. Two simple examples are shown below:

```
3 + 7                                  // evaluates to 10
3 - 7                                  // evaluates to -4
```

EXAMPLE 3-1 Binary Expressions

Now let's look at a short program that uses some of these expressions. Program 3-3 contains several binary expressions.

PROGRAM 3-3 Binary Expressions

```
/* This program demonstrates binary expressions.
      Written by:
      Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   int   a = 17;
   int   b = 5;
   float x = 17.67;
   float y = 5.1;

// Statements
   printf("Integral calculations\n");
   printf("%d + %d  = %d\n", a, b, a + b);
   printf("%d - %d  = %d\n", a, b, a - b);
   printf("%d * %d  = %d\n", a, b, a * b);
   printf("%d / %d  = %d\n", a, b, a / b);
   printf("%d %% %d  = %d\n", a, b, a % b);
   printf("\n");

   printf("Floating-point calculations\n");
   printf("%f + %f  = %f\n", x, y, x + y);
   printf("%f - %f  = %f\n", x, y, x - y);
   printf("%f * %f  = %f\n", x, y, x * y);
   printf("%f / %f  = %f\n", x, y, x / y);
```

continued

PROGRAM 3-3 Binary Expressions *(continued)*

```
28       return 0;
29    }  // main
```

```
Results:
Integral calculations
17 + 5 = 22
17 - 5 = 12
17 * 5 = 85
17 / 5 = 3
17 % 5 = 2

Floating-point calculations
17.670000 + 5.100000  = 22.770000
17.670000 - 5.100000  = 12.570000
17.670000 * 5.100000  = 90.116997
17.670000 / 5.100000  = 3.464706
```

Program 3-3 Analysis This simple program requires only three explanatory comments. (1) Note that even for a simple program we include all of the standard documentation comments. (2) We do not recommend that you include calculations in print statements as we have done in this program—it is not a good structured programming technique. We include them in this program because we haven't yet shown you how to save the results of a calculation. (3) Study the format string in statement 20. To print a percent sign as text in the format string, we need to code two percent signs.

Assignment Expressions

The **assignment expression** evaluates the operand on the right side of the operator (=) and places its value in the variable on the left. The assignment expression has a value and a side effect.

- The value of the total expression is the value of the expression on the right of the assignment operator (=).
- The side effect places the expression value in the variable on the left of the assignment operator.

The left operand in an assignment expression must be a single variable.

There are two forms of assignment: simple and compound.

Simple Assignment

Simple assignment is found in algebraic expressions. Three examples of simple assignments are shown below.

```
a = 5          b = x + 1          i = i + 1
```

Of course, for the effect to take place, the left variable must be able to receive it; that is, it must be a variable, not a constant. If the left operand cannot receive a value and we assign one to it, we get a compile error.

Compound Assignment

A compound assignment is a shorthand notation for a simple assignment. It requires that the left operand be repeated as a part of the right expression. Five compound assignment operators are discussed in this chapter: `*=`, `/=`, `%=`, `+=`, and `-=`.

To evaluate a compound assignment expression, first change it to a simple assignment, as shown in Table 3-2. Then perform the operation to determine the value of the expression.

Compound Expression	Equivalent Simple Expression
`x *= expression`	`x = x * expression`
`x /= expression`	`x = x / expression`
`x %= expression`	`x = x % expression`
`x += expression`	`x = x + expression`
`x -= expression`	`x = x - expression`

TABLE 3-2 Expansion of Compound Expressions

When a compound assignment is used with an expression, the expression is evaluated first. Thus, the expression

```
x *= y + 3
```

is evaluated as

```
x = x * (y + 3)
```

which, given the values x is 10 and y is 5, evaluates to 80.

Program 3-4 demonstrates the first three compound expressions.

PROGRAM 3-4 Demonstration of Compound Assignments

```
/* Demonstrate examples of compound assignments.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
```

continued

PROGRAM 3-4 Demonstration of Compound Assignments *(continued)*

```
// Local Declarations
   int x;
   int y;

// Statements
   x = 10;
   y = 5;

   printf("x: %2d  |  y: %2d ",     x, y);
   printf("  |  x *= y + 2: %2d ", x *= y + 2);
   printf("  |  x is now: %2d\n",  x);

   x = 10;
   printf("x: %2d  |  y: %2d ",     x, y);
   printf("  |  x /= y + 1: %2d ", x /= y + 1);
   printf("  |  x is now: %2d\n",  x);

   x = 10;
   printf("x: %2d  |  y: %2d ",      x, y);
   printf("  |  x %%= y - 3: %2d ", x %= y - 3);
   printf("  |  x is now: %2d\n", x);

   return 0;
}  // main
```

```
Results:
x: 10  |  y:  5   |  x *= y + 2: 70   |  x is now: 70
x: 10  |  y:  5   |  x /= y + 1:  1   |  x is now:  1
x: 10  |  y:  5   |  x %= y - 3:  0   |  x is now:  0
```

Program 3-4 Analysis Note that we have used an assignment statement in the *printf* statements to demonstrate that an assignment expression has a value. As we said before, this is not good programming style, but we use it here to match the format we used in Program 3-3. *Do not hide calculations in print statements.* Also, since we are changing the value of x with each assignment, even though it is in a *printf* statement, we need to reset it to 10 for each of the print series.

3.2 Precedence and Associativity

Precedence is used to determine the order in which different operators in a complex expression are evaluated. **Associativity** is used to determine the order in which operators with the same precedence are evaluated in a complex expression. Another way of stating this is that associativity determines how operators *with the same precedence* are grouped together to form complex expressions. Precedence is applied before associativity to determine the order in which expressions are evaluated. Associativity is then applied, if necessary.

Precedence

The concept of precedence is well founded in mathematics. For example, in algebra, multiplication and division are performed before addition and subtraction. C extends the concept to 16 levels, as shown in the Precedence Table inside the front cover.

The following is a simple example of precedence:

```
2 + 3 * 4
```

This expression is actually two binary expressions, with one addition and one multiplication operator. Addition has a precedence of 12. Multiplication has a precedence of 13. This results in the multiplication being done first, followed by the addition, as shown below in the same expression with the default parentheses added. The value of the complete expression is 14.

```
(2 + (3 * 4)) → 14
```

As another example consider the following expression:

```
-b++
```

Two different operators are in this expression. The first is the unary minus, the second is the postfix increment. The postfix increment has the higher precedence (16), so it is evaluated first. Then the unary minus, with a precedence of 15, is evaluated. To reflect the precedence, we have recoded the expression using parentheses.

```
(-(b++))
```

Assuming that the value of b is 5 initially, the expression is evaluated to −5. What is the value of b after the expression is complete? It is 6 because the operator has an effect that is separate from the value of the expression.

Program 3-5 demonstrates precedence by printing the same expression, once without parentheses and once with parentheses to change the precedence. Because the parentheses create a primary expression that must be evaluated before the binary multiply, the answer is different.

PROGRAM 3-5 Precedence

```
/* Examine the effect of precedence on an expression.
      Written by:
```

continued

PROGRAM 3-5 Precedence *(continued)*

```
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int a = 10;
   int b = 20;
   int c = 30;

// Statements
   printf ("a *  b + c  is: %d\n", a *  b + c);
   printf ("a * (b + c) is: %d\n", a * (b + c));
   return 0;
}  // main
```

```
Results:
a *  b + c  is: 230
a * (b + c) is: 500
```

Associativity

Associativity can be left-to-right or right-to-left. **Left-to-right associativity** evaluates the expression by starting on the left and moving to the right. Conversely, **right-to-left associativity** evaluates the expression by proceeding from the right to the left. Remember, however, that associativity is used only when the operators all have the same precedence.

Associativity is applied when we have more than one operator of the same precedence level in an expression.

ASSOCIATIVITY

Left-to-right Associativity

The following shows an example of left-to-right associativity. Here we have four operators of the same precedence (`* / % *`).

```
3  *  8  /  4  %  4  *  5
```

Associativity determines how the subexpressions are grouped together. All of these operators have the same precedence (13). Their associativity is from left to right. So they are grouped as follows:

```
((((3 * 8) / 4) % 4) * 5)
```

The value of this expression is 10. A graphical representation of this expression is shown in Figure 3-8.

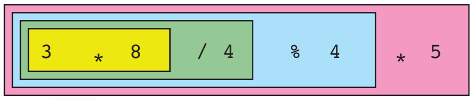


FIGURE 3-8 Left-to-right Associativity

Right-to-left Associativity

Several operators have right-to-left associativity as shown in the precedence table. For example, when more than one assignment operator occurs in an assignment expression, the assignment operators must be interpreted from right to left. This means that the rightmost expression will be evaluated first; then its value will be assigned to the operand on the left of the assignment operator and the next expression will be evaluated. Under these rules, the expression

```
a += b *= c -= 5
```

is evaluated as

```
(a += (b *= (c -= 5)))
```

which is expanded to

```
(a = a + (b = b * (c = c - 5)))
```

If `a` has an initial value of 3, `b` has an initial value of 5, and `c` has an initial value of 8, these expressions become

```
(a = 3 + (b = (5 * (c =  8 - 5)))
```

which results in `c` being assigned a value of 3, `b` being assigned a value of 15, and `a` being assigned a value of 18. The value of the complete expression is also 18. A diagram of this expression is shown in Figure 3-9.

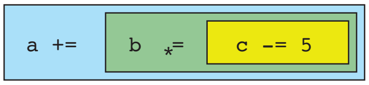


FIGURE 3-9 Right-to-left Associativity

A simple but common form of assignment is shown below. Suppose we have several variables that all need to be initialized to zero. Rather than initializing each separately, we can use a complex statement to do it.

```
a = b = c = d = 0;
```

3.3 Side Effects

A **side effect** is an action that results from the evaluation of an expression. For example, in an assignment, C first evaluates the expression on the right of the assignment operator and then places the value in the left variable. Changing the value of the left variable is a side effect. Consider the following expression:

```
x = 4;
```

This simple expression has three parts. First, on the right of the assignment operator is a primary expression that has the value 4. Second, the whole expression (x = 4) also has a value of 4. And third, as a side effect, x receives the value 4.

Let's modify the expression slightly and see the same three parts.

```
x = x + 4;
```

Assuming that x has an initial value of 3, the value of the expression on the right of the assignment operator has a value of 7. The whole expression also has a value of 7. And as a side effect, x receives the value of 7. To prove these three steps to yourself, write and run the following code fragment:

```
int x = 3;
printf("Step 1--Value of x: %d\n", x);
printf("Step 2--Value of x = x + 4: %d\n", x = x + 4);
printf("Step 3--Value of x now: %d\n", x);
```

Now, let's consider the side effect in the postfix increment expression. This expression is typically coded as shown below.

```
a++
```

As we saw earlier, the value of this expression is the value of `a` before the expression is evaluated. As a side effect, however, the value of `a` is incremented by 1.

In C, six operators generate side effects: prefix increment and decrement, postfix increment and decrement, assignment, and function call.

3.4 Evaluating Expressions

Now that we have introduced the concepts of precedence, associativity, and side effects, let's work through some examples.

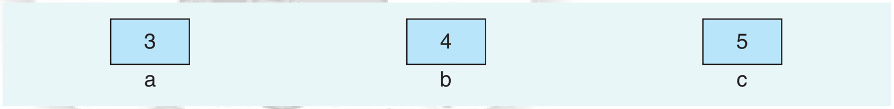
Expressions without Side Effects

EXAMPLE 3-2 Expression with no Side Effects

The first expression is shown below. It has no side effects, so the values of all of its variables are unchanged.

```
a * 4 + b / 2 - c * b
```

For this example, assume that the values of the variables are



To evaluate an expression *without side effects*, follow the simple rules shown below.

1. Replace the variables by their values. This gives us the following expression:

```
3 * 4 + 4 / 2 - 5 * 4
```

2. Evaluate the highest precedence operators, and replace them with the resulting value. In the above expression, the operators with the highest precedence are the multiply and divide (13). We therefore evaluate them first from the left and replace them with the resulting values. The expression is now

```
(3 * 4) + (4 / 2) - (5 * 4) → 12 + 2 - 20
```

3. Repeat step 2 until the result is a single value.

In this example, there is only one more precedence, binary addition and subtraction. After they are evaluated, the final value is –6. Since this expression had no side effects, all of the variables have the same values after the expression has been evaluated that they had at the beginning.

Expressions with Side Effects

EXAMPLE 3-3 Example with Side Effects

Now let's look at the rules for an expression that has side effects and parenthesized expressions. For this example, consider the expression

```
--a * (3 + b) / 2 - c++ * b
```

Assume that the variables have the values used above, a is 3, b is 4, c is 5. To evaluate this expression, use the following rules:

1. Calculate the value of the parenthesized expression (3 + b) first (precedence 16). The expression now reads

```
--a * 7 / 2 - c++ * b
```

2. Evaluate the postfix expression (c++) next (precedence 16). Remember that as a postfix expression, the value of c++ is the same as the value of c; the increment takes place after the evaluation. The expression is now

```
--a * 7 / 2 - 5 * b
```

3. Evaluate the prefix expression (--a) next (priority 15). Remember that as a prefix expression, the value of --a is the value after the side effect, which means that we first decrement a and then use its decremented value. The expression is now

```
2 * 7 / 2 - 5 * b
```

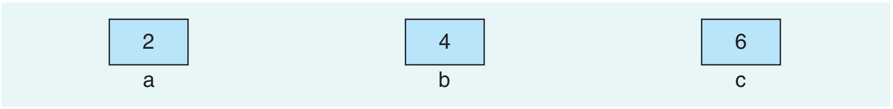
4. The multiply and division are now evaluated using their associativity rule, left to right, as shown below.

```
14 / 2 - 5 * b → 7 - 5 * 4 → 7 - 20
```

5. The last step is to evaluate the subtraction. The final expression value is -13 as shown in the final example.

```
7 - 20 → -13
```

After the side effects, the variables have the values shown below.



Program 3-6 evaluates the two expressions in this section.

PROGRAM 3-6 Evaluating Expressions

```c
/* Evaluate two complex expressions.
      Written by:
      Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   int a = 3;
   int b = 4;
   int c = 5;
   int x;
   int y;

// Statements
   printf("Initial values of the variables: \n");
   printf("a = %d\tb = %d\tc = %d\n\n", a, b, c);

   x = a * 4 + b / 2 - c * b;
   printf
     ("Value of   a *   4 + b   / 2 - c   * b:  %d\n", x);

   y = --a * (3 + b) / 2 - c++ * b;
   printf
     ("Value of --a * (3 + b) / 2 - c++ * b: %d\n", y);
   printf("\nValues of the variables are now: \n");
   printf("a = %d\tb = %d\tc = %d\n\n", a, b, c);

   return 0;
}  // main
```

```
Results:
Initial values of the variables:
a = 3   b = 4   c = 5

Value of   a *   4 + b   / 2 - c   * b:  -6
Value of --a * (3 + b) / 2 - c++ * b: -13

Values of the variables are now:
a = 2   b = 4   c = 6
```

Warning

A warning is in order: In C, if an expression variable is modified more than once during its evaluation, the result is undefined. C has no specific rule to cover when the side effect takes place, and compiler writers can implement the side effect in different ways. The result is that different compilers will give different expression results.

In C, if a single variable is modified more than once in an expression, the result is undefined.

WARNING

3.5 Type Conversion

Up to this point, we have assumed that all of our expressions involved data of the same type. But, what happens when we write an expression that involves two different data types, such as multiplying an integer and a floating-point number? To perform these evaluations, one of the types must be converted.

Implicit Type Conversion

When the types of the two operands in a binary expression are different, C automatically converts one type to another. This is known as **implicit type conversion**. For implicit type conversion, C has several complicated rules that we gradually introduce throughout the book. We mention some of the simple conversions in this section.

Conversion Rank

Before we discuss how conversions are handled, we need to discuss the concept of **conversion rank**. In C, we can assign a rank to the integral and floating-point arithmetic types. Figure 3-10 shows the ranks as we use them for conversion in this chapter. While the 1 to 9 scale we use is conceptually correct, the actual implementation is much more complex.

As shown in Figure 3-10, a *long double* real has a higher rank than a *long* integer and a short integer has a higher rank than a character.

Conversions in Assignment Expressions

A simple assignment involves an assignment operator and two operands. Depending on the difference in the rank, C tries to either promote or demote the right expression to make it the same rank as the left variable. **Promotion** occurs if the right expression has lower rank; **demotion** occurs if the right expression has a higher rank.

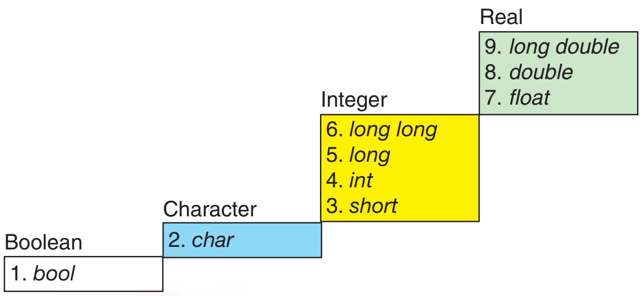


FIGURE 3-10 Conversion Rank

Promotion

There is normally no problem with promotion. The rank of the right expression is elevated to the rank of the left variable. The value of the expression is the value of the right expression after the promotion. The following examples demonstrate some simple promotions.

```
bool        b = true;
char        c = 'A';
int         i = 1234;
long double d = 3458.0004;
c = b;                          // value of c is SOH (ASCII 1)
i = c;                          // value of i is 65
d = b;                          // value of d is 1.0
d = i;                          // value of d is 1234.0
```

Demotion

Demotion may or may not create problems. If the size of the variable at the left side can accommodate the value of the expression, there is no problem; however, some of the results may surprise you.

Any integral or real value can be assigned to a Boolean type. If the value of the expression on the right is zero, *false* (0) is stored; if the result is not zero, either positive or negative, *true* (1) is stored.

When an integer or a real is assigned to a variable of type character, the least significant byte of the number is converted to a character and stored. When a real is stored in an integer, the fraction part is dropped. However, if the integral part is larger than the maximum value that can be stored, the results are invalid and unpredictable. Similarly, when we try to store a *long double* in a variable of type *float*, the results are valid if the value fits or invalid if it is too large.

The following examples demonstrate demotion.

```
bool  b = false;
char  c = 'A';
short s = 78;
int   j = INT_MAX;
int   k = 65;
...
b = c;                          // value of b is 1 (true)
s = j;                          // value of s is unpredictable
c = k + 1;                      // demotion: value of c is 'B'
```

Conversion in Other Binary Expressions

Conversion has a different set of rules for the other binary expressions. The rules are sometimes very complicated, but we can summarize them in three steps, which cover most cases:

1. The operand with the higher rank is determined using the ranking in Figure 3-10.
2. The lower-ranked operand is promoted to the rank defined in step 1. After the promotion, both expressions have the same rank.
3. The operation is performed with the expression value having the type of the promoted rank.

The following examples demonstrate some common conversions.

```
bool        b = true;
char        c = 'A';
int         i = 3650;
short       s = 78;
long double d = 3458.0004;
b + c                   // b promoted; result is 'B' ('A' + 1)
i * s;                  // result is an int
d * c;                  // result is long double
```

Let's look at a small program to see the effect of implicit conversions. In Program 3-7 we add a character, an integer, and a *float*. We can add characters to integers and floating-point values because all characters have an ASCII value that can be promoted.

PROGRAM 3-7 Implicit Type Conversion

```
/* Demonstrate automatic promotion of numeric types.
      Written by:
      Date:
*/
#include <stdio.h>
```

continued

PROGRAM 3-7 Implicit Type Conversion *(continued)*

```
 6  #include <stdbool.h>
 7
 8  int main (void)
 9  {
10  // Local Declarations
11     bool  b  = true;
12     char  c  = 'A';
13     float d  = 245.3;
14     int   i  = 3650;
15     short s  = 78;
16
17  // Statements
18     printf("bool + char is char:   %c\n", b + c);
19     printf("int * short is int:    %d\n", i * s);
20     printf("float * char is float: %f\n", d * c);
21
22     c = c + b;                     // bool promoted to char
23     d = d + c;                     // char promoted to float
24     b = false;
25     b = -d;                        // float demoted to bool
26
27     printf("\nAfter execution...\n");
28     printf("char + true:    %c\n", c);
29     printf("float + char:   %f\n", d);
30     printf("bool = -float:  %f\n", b);
31
32     return 0;
33  }  // main
```

```
Results:
bool + char is char:   B
int * short is int:    284700
float * char is float: 15944.500000

After execution...
char + true:    B
float + char:   311.299988
bool = -float:  1
```

Program 3-7 Analysis Several points in this program require explanation. First, as we stated before, it is not a good programming practice to code an expression in a print statement. We do it in this program, however, to demonstrate the promotion or demotion of the expression.

The first print series displays the value of mixed type expressions. As you examine each result, note that the value printed is in the form of the higher ranked variable. For example, in statement 18 the Boolean in the expression (b) is promoted to a character and then added to the value of the character expression (`c`). The result is the character `B`, which is then passed to the *printf* function where it is printed using the format specification `%c`.

The second print series displays the results of assignments. In the first one, we add *true* (1) to the letter A. The result is B as you would expect. In the second assignment, we add the letter B from the previous assignment to a real number. The new value of the real number is almost 66 greater than the original value. The difference occurs because real numbers are now exact. Rather than storing 245.3, the value stored was 245.299988.

Finally, note what happens when we assign a negative, real number to a boolean. The result is *true*.

Explicit Type Conversion (Cast)

Rather than let the compiler implicitly convert data, we can convert data from one type to another ourself using **explicit type conversion**. Explicit type conversion uses the unary **cast** operator, which has a precedence of 14. To cast data from one type to another, we specify the new type in parentheses before the value we want converted. For example, to convert an integer, `a`, to a *float*, we code the expression shown below.

```
(float) a
```

Note that in this operation, like any other unary operation, the value stored in `a` is still of type *int*, but the value of the expression is promoted to *float*.

One use of the cast is to ensure that the result of a divide is a real number. For example, if we calculated the average of a series of integer test scores without a cast, the result would be an integer. To force a real result, we cast the calculation as shown below.

```
average = (float) totalScores / numScores;
```

In this statement, there is an explicit conversion of `totalScores` to *float*, and then an implicit conversion of `numScores` so that it will match. The result of the divide is then a floating-point number to be assigned to `average`.

But beware! What would be the result of the following expression when `a` is 3?

```
(float) (a / 10)
```

Are you surprised to find that the result is 0.0? Since no conversions are required to divide integer 3 by integer 10, C simply divides with an integer result, 0. The integer 0 is then explicitly converted to the floating-point 0.0. To get a *float* result, we must cast one of the numbers as shown below.

```
(float) a / 10
```

One final thought about casts: Even when the compiler can correctly cast for you, it is sometimes better to code the cast explicitly as a reminder that the cast is taking place.

Program 3-8 demonstrates the use of explicit casts. In this program, we divide several mixed types. While the results are nonsense, they demonstrate the effect of casting.

PROGRAM 3-8 Explicit Casts

```
/* Demonstrate casting of numeric types.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   char   aChar    = '\0';
   int    intNum1  = 100;
   int    intNum2  =  45;
   double fltNum1  = 100.0;
   double fltNum2  =  45.0;
   double fltNum3;

// Statements
   printf("aChar numeric    :  %3d\n",    aChar);
   printf("intNum1 contains:  %3d\n",    intNum1);
   printf("intNum2 contains:  %3d\n",    intNum2);
   printf("fltNum1 contains:  %6.2f\n", fltNum1);
   printf("fltNum2 contains:  %6.2f\n", fltNum2);

   fltNum3 = (double)(intNum1 / intNum2);
   printf
      ("\n(double)(intNum1 / intNum2): %6.2f\n",
       fltNum3);

   fltNum3 = (double)intNum1 / intNum2;
   printf("(double) intNum1 / intNum2 : %6.2f\n",
          fltNum3);

   aChar = (char)(fltNum1 / fltNum2);
   printf("  (char)(fltNum1 / fltNum2): %3d\n",   aChar);

```

continued

PROGRAM 3-8 Explicit Casts *(continued)*

```
36      return 0;
37   }  // main
```

```
Results:
aChar numeric    :    0
intNum1 contains:  100
intNum2 contains:   45
fltNum1 contains:  100.00
fltNum2 contains:   45.00

(double)(intNum1 / intNum2):    2.00
(double) intNum1 / intNum2 :    2.22
  (char)(fltNum1 / fltNum2):    2
```

Program 3-8 Analysis

Study the casts carefully. The only difference between statements 24 and 29 is the use of parentheses around the calculation. In statement 24, both operands are integers so the result of the division is integer, which is then cast to a *double*. In statement 29, `intNum1` is cast to a *double*. The compiler automatically casts `intNum2` to a *double* before the division. The result is therefore a *double*. Finally, in statement 33, we cast the result of the integer division into a character.

3.6 Statements

A statement causes an action to be performed by the program. It translates directly into one or more executable computer instructions.

You may have noticed that we have used a semicolon at the end of the statements in our programs. Most statements need a semicolon at the end; some do not. When we discuss statements, we identify those that do.

Statement Type

C defines eleven types of statements, which are shown in Figure 3-11. In this chapter, we will discuss the first four; the other types will be covered in the future chapters.

Null Statement

The **null statement** is just a semicolon (the terminator) as shown below:

```
  ;                                     // null statement
```

Although they do not arise often, there are syntactical situations where we must have a statement but no action is required. In these situations, we use the null statement.

Expression Statement

An expression is turned into a statement by placing a semicolon (`;`) after it.

```
expression;                    // expression statement
```

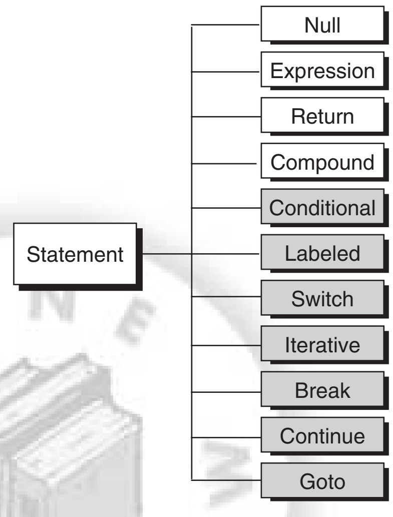


FIGURE 3-11 Types of Statements

When C sees the semicolon, it completes any pending side effects and discards the expression value before continuing with the next statement. An expression without side effects does not cause an action. Its value exists and it can be used, but unless a statement has a side effect, it does nothing.

Let us look at some **expression statements**. First, consider the expression statement

```
a = 2;
```

The effect of the expression statement is to store the value, 2, in the variable `a`. The value of the expression is 2. After the value has been stored, the expression is terminated (because there is a semicolon), and the value is discarded. C then continues with the next statement.

The next expression statement is a little more complex.

```
a = b = 3;
```

This statement actually has two expressions. If we put parentheses around them, you will be able to see them clearly.

```
a = (b = 3);
```

The parenthesized expression (`b = 3`) has a side effect of assigning the value 3 to the variable `b`. The value of this expression is 3. The expression statement now results in the expression value 3 being assigned to the variable `a`. Since the expression is terminated by the semicolon, its value, 3, is discarded. The effect of the expression statement, therefore, is that 3 has been stored in both `a` and `b`.

In Chapter 2 we examined the *scanf* and *printf* functions. These statements present interesting insights into the concepts of expressions and side effects. Consider the following *scanf* function call:

```
ioResult = scanf("%d", &x);
```

This statement has two side effects. The first is found in the *scanf* function. Reading an integer value from the keyboard and placing it into the variable `x` (note the address operator before the variable) is a side effect. The second side effect is storing the value returned by *scanf*, which represents the number of values that were converted correctly. In this case, the return value could be EOF, 0, or 1. Assuming that the user correctly keys the integer, the value will be 1. The assignment operator stores the return value in `ioResult`. The expression then terminates, and the *scanf* value is discarded.

In a similar fashion, *printf* has the effect of displaying data on the monitor and returning a value, the number of characters displayed. This is seen in the statement below.

```
numDisplayed = printf
     ("x contains %d, y contains %d\n", x, y);
```

As a general rule, however, the number of characters displayed is discarded without being stored. Therefore, we normally code the above statement as shown below.

```
printf("x contains %d, y contains %d\n", x, y);
```

Now consider the following expression statement. Assume that `a` has a value of 5 before the expression is evaluated.

```
a++;
```

In this postfix expression, the value of the expression is 5, the value of the variable, `a`, before it is changed by the side effect. Upon completion of the expression statement, `a` is incremented to 6. The value of the expression, which is still 5, is discarded since the expression is now complete.

Although they are useless, the following are also expression statements. They are useless because they have no side effect and their values are not assigned to a variable. We usually don't use them, but it is important to know they are syntactically correct expression statements. C will evaluate them, determine their value, and then discard the value.[1]

```
               b;              3;              ;
```

Return Statement

A return statement terminates a function. All functions, including *main*, must have a return statement. When there is no return statement at the end of the function, the system inserts one with a *void* return value.

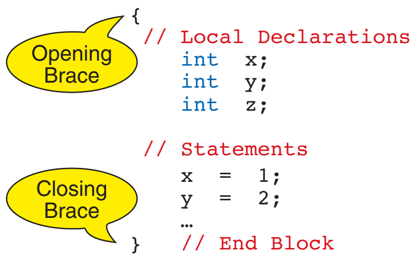
```
return expression;                  // return statement
```

In addition, the return statement can return a value to the calling function. For example, while we have not talked about it or used it, the *scanf* function returns the number of variables successfully read. In the case of *main*, it returns a value to the operating system rather than to another function. In all of our programs, *main* returns 0 to the operating system. A return value of zero tells the operating system that the program executed successfully.

Compound Statements

A **compound statement** is a unit of code consisting of zero or more statements. It is also known as a **block**. The compound statement allows a group of statements to become one single entity. You used a compound statement in your first program when you formed the body of the function *main*. All C functions contain a compound statement known as the function body.

A compound statement consists of an opening brace, an optional declaration and definition section, and an optional statement section, followed by a closing brace. Although both the declaration and the statement sections are optional, one should be present. If neither is present, then we have a null statement, which doesn't make much sense. Figure 3-12 shows the makeup of a compound statement.



```
{
 // Local Declarations
    int   x;
    int   y;
    int   z;

 // Statements
    x  =  1;
    y  =  2;
    …
}   // End Block
```

FIGURE 3-12 Compound Statement

1. In an optimized compiler, the translator determines that there is no effect and generates no code.

One important point to remember is that a compound statement does not need a semicolon. The opening and closing parentheses are simply delimiters for the compound statement. If we put a semicolon after the closing brace, the compiler thinks that we have put an extra null statement after the compound statement. This is poor style, but it does not generate any code or generate a compile error, although it may generate a warning message.

The compound statement does not need a semicolon.

C requires that the declaration section be before any statements[2] within a compound statement block. The code in the declaration section and statement section cannot be intermixed.

The Role of the Semicolon

The semicolon plays an important role in the syntax of the C language. It is used in two different auscultations.

- Every declaration in C is terminated by a semicolon.
- Most statements in C are terminated by a semicolon.

On the other hand, we must be careful not to use a semicolon when it is not needed. A semicolon should not be used with a preprocessor directive such as the include and define. In particular, a semicolon at the end of a define statement can create a problem as discussed in the next section.

Statements and Defined Constants

When we use preprocessor-defined commands, we must be very careful to make sure that we do not create an error. Remember that the defined constant is an automatic substitution. This can cause subtle problems. One common mistake is to place a semicolon at the end of the definition. Since the preprocessor uses a simple text replacement of the name with whatever expression follows, the compiler will generate a compile error if it finds a semicolon at the end of the definition. This problem is seen in the following example:

```
#define SALES_TAX_RATE 0.825;
…
salesTax = SALES_TAX_RATE * salesAmount;
```

After the substitution, the following erroneous code occurs because we coded a semicolon after the constant value:

```
salesTax = 0.0825; * salesAmount;
```

2. The one exception to this rule is in the *for* statement, which we study in Chapter 6.

This can be an extremely difficult compile error to figure out because we see the original statement and not the erroneous substitution error. One of the reasons programmers use uppercase for defined constant identifiers is to provide an automatic warning to readers that they are not looking at the real code.

3.7 Sample Programs

This section contains several programs that you should study for programming technique and style.

EXAMPLE 3-4 Calculate Quotient and Remainder

Let's write a program that calculates and prints the quotient and remainder of two integer numbers. The code is shown in Program 3-9.

PROGRAM 3-9 Calculate Quotient and Remainder

```c
/* Calculate and print quotient and remainder of two
   numbers.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int  intNum1;
   int  intNum2;
   int  intCalc;

// Statements
   printf("Enter two integral numbers: ");
   scanf ("%d %d", &intNum1, &intNum2);

   intCalc = intNum1 / intNum2;
   printf("%d / %d is %d", intNum1, intNum2, intCalc);

   intCalc = intNum1 % intNum2;
   printf(" with a remainder of: %d\n", intCalc);

   return 0;
}  // main
```

continued

PROGRAM 3-9 Calculate Quotient and Remainder *(continued)*

```
Results:
Enter two integral numbers: 13 2
13 / 2 is 6 with a remainder of: 1
```

Program 3-9 Analysis Using good programming style, the program begins with documentation about what it does, who created it, and when it was created.

Program 3-9 has no global variable declarations, so after including the standard input/output library, we start immediately with *main*. Following *main* is the opening brace. The matching closing brace is found on line 26.

The most difficult part of this program is figuring out how to get the remainder. Fortunately, C has a modulo operator (%) that does the job for us. The rest of the problem is straightforward.

EXAMPLE 3-5 Print Right Digit

Another problem that requires the use of the modulo operator is to print a digit contained in an integer. Program 3-10 prints the least significant (rightmost) digit of an integer.

PROGRAM 3-10 Print Right Digit of Integer

```c
/* Print rightmost digit of an integer.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int  intNum;
   int  oneDigit;

// Statements
   printf("Enter an integral number: ");
   scanf ("%d", &intNum);

   oneDigit = intNum % 10;
   printf("\nThe right digit is: %d", oneDigit);

   return 0;
}  // main
```

```
Results:
```

continued

PROGRAM 3-10 Print Right Digit of Integer *(continued)*

```
Enter an integral number: 185

The right digit is: 5
```

EXAMPLE 3-6 Calculate Average

Program 3-11 reads four integers from the keyboard, calculates their average, and then prints the numbers with their average and the deviation (not the standard deviation, just the difference plus or minus) from the average.

PROGRAM 3-11 Calculate Average of Four Numbers

```
/* Calculate the average of four integers and print
   the numbers and their deviation from the average.
      Written by:
      Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   int    num1;
   int    num2;
   int    num3;
   int    num4;
   int    sum;
   float  average;

// Statements
   printf("\nEnter the first number  : ");
   scanf("%d", &num1);
   printf("Enter the second number : ");
   scanf("%d", &num2);
   printf("Enter the third  number : ");
   scanf("%d", &num3);
   printf("Enter the fourth number : ");
   scanf("%d", &num4);

   sum     = num1 + num2 + num3 + num4;
   average = sum / 4.0;

   printf("\n ******** average is %6.2f ******** ",
            average);
   printf("\n");
```

continued

PROGRAM 3-11 Calculate Average of Four Numbers *(continued)*

```c

   printf("\nfirst number:  %6d -- deviation: %8.2f",
            num1, num1 - average);
   printf("\nsecond number: %6d -- deviation: %8.2f",
            num2, num2 - average);
   printf("\nthird number:  %6d -- deviation: %8.2f",
            num3, num3 - average);
   printf("\nfourth number: %6d -- deviation: %8.2f",
            num4, num4 - average);

   return 0;
}  // main
```

```
Results:
Enter the first number:  23
Enter the second number: 12
Enter the third  number: 45
Enter the fourth number: 23

******** average is 25.75  ********

first number:      23 -- deviation:    -2.75
second number:     12 -- deviation:   -13.75
third number:      45 -- deviation:    19.25
fourth number:     23 -- deviation:    -2.75
```

Program 3-11 Analysis

Program 3-11 is a little more complex than the previous ones. At the beginning of *main* are several variable declarations, five integers, and a floating-point number. The first four are for the variables read from the keyboard, the fifth is for the sum, and the floating-point number is for the average.

The statements section starts by reading the data. Each read is preceded by a display so the user will know what to do. The specific instructions about what to input are known as **user prompts**. You should always tell the user what input is expected from the keyboard. After the user has keyed the data, the program continues by adding the numbers, placing the total in `sum`, and computing `average`. It then displays the results. Notice that the program displays the results in a format that allows the user to easily verify that the program ran correctly. The program not only prints the average but also repeats each input with its deviation from the average. After completing its work, the program concludes by returning to the operating system.

Look at the results carefully. Note how each series of numbers is aligned so that they can be easily read. Taking the time to align output is one of the things that distinguishes a good programmer from an average programmer. Always pay attention to how your program presents its results to the user. Paying attention to these little details pays off in the long run.

EXAMPLE 3-7 Degrees to Radians

One way to measure an angle in a circle is in degrees. For example, the angle formed by a clock at 3 o'clock is 90°. Another way to measure the angle is in radians. One radian is equal to 57.295779[3] degrees. In Program 3-12 we ask the user to input an angle in radians, and we convert it to degrees.

PROGRAM 3-12 Convert Radians to Degrees

```
/* This program prompts the user to enter an angle
   measured in radians and converts it into degrees.
      Written by:
      Date:
*/
#include <stdio.h>

#define DEGREE_FACTOR 57.295779

int main (void)
{
// Local Declarations
   double  radians;
   double  degrees;

// Statements
   printf("Enter the angle in radians: ");
   scanf("%lf", &radians);

   degrees = radians * DEGREE_FACTOR;

   printf("%6.3f radians is %6.3f degrees\n",
           radians, degrees);
   return 0;
}  // main
```

```
Results:
Enter the angle in radians: 1.57080
 1.571 radians is 90.000 degrees
```

Program 3-12 Analysis

In this short program we introduce the defined constant. Defined constants are an excellent way to document factors in a program. We could have just as easily used a memory constant. Factors are usually placed at the beginning of the program where they can easily be found and changed as necessary. Remember that the area before *main* is global and that it should be used only for declarations and constants—not variables.

3. More precisely, a radian is (180 / π).

EXAMPLE 3-8 Calculate Sales Total

Program 3-13 calculates a sale given the unit price, quantity, discount rate, and sales tax rate.

PROGRAM 3-13 Calculate Sales Total

```c
/* Calculates the total sale given the unit price,
   quantity, discount, and tax rate.
      Written by:
      Date:
*/
#include <stdio.h>

#define TAX_RATE 8.50

int main (void)
{
// Local Declarations
   int    quantity;

   float  discountRate;
   float  discountAm;
   float  unitPrice;
   float  subTotal;
   float  subTaxable;
   float  taxAm;
   float  total;

// Statements
   printf("\nEnter number of items sold:         ");
   scanf("%d", &quantity);

   printf("Enter the unit price:               ");
   scanf("%f", &unitPrice);

   printf("Enter the discount rate (per cent): ");
   scanf("%f", &discountRate);

   subTotal    = quantity * unitPrice;
   discountAm  = subTotal * discountRate / 100.0;
   subTaxable  = subTotal - discountAm;
   taxAm = subTaxable * TAX_RATE/ 100.00;
   total = subTaxable + taxAm;

   printf("\nQuantity sold:       %6d\n", quantity);
```

continued

PROGRAM 3-13 Calculate Sales Total *(continued)*

```
   printf("Unit Price of items: %9.2f\n", unitPrice);
   printf("                      ------------\n");

   printf("Subtotal :            %9.2f\n", subTotal);
   printf("Discount:            -%9.2f\n", discountAm);
   printf("Discounted total:     %9.2f\n", subTaxable);
   printf("Sales tax:           +%9.2f\n", taxAm);
   printf("Total sale:           %9.2f\n", total);

   return 0;
}  // main
```

```
Results:
Enter number of items sold:          34
Enter the unit price:                12.89
Enter the discount rate (per cent): 7

Quantity sold:           34
Unit Price of items:     12.89
                      ------------
Subtotal :              438.26
Discount:            -    30.68
Discounted total:       407.58
Sales tax:           +    34.64
Total sale:             442.23
```

Program 3-13 Analysis Look at the results of this program carefully. Do you see any problems? Just because a program runs doesn't mean that it is running correctly. In this case, the total is incorrect (407.58 + 34.64 is not equal to 442.23). The problem is created by the floating-point arithmetic and rounding errors. If we wanted absolute accuracy, we would have to do the arithmetic in integer (cents) and then divide by 100 to print the report.

EXAMPLE 3-9 Calculate Student Score

Program 3-14 calculates the average score for a student. The class has four quizzes (30%), two midterms (40%), and a final (30%). The maximum score for all quizzes and exams is 100 points.

PROGRAM 3-14 Calculate Student Score

```
/* Calculate a student's average score for a course
   with 4 quizzes, 2 midterms, and a final. The quizzes
   are weighted 30%, the midterms 40%, & the final 30%.
      Written by:
```

continued

PROGRAM 3-14 Calculate Student Score *(continued)*

```
       Date:
*/
#include <stdio.h>

#define QUIZ_WEIGHT     30
#define MIDTERM_WEIGHT  40
#define FINAL_WEIGHT    30
#define QUIZ_MAX       400.00
#define MIDTERM_MAX    200.00
#define FINAL_MAX      100.00

int main (void)
{
// Local Declarations
   int    quiz1;
   int    quiz2;
   int    quiz3;
   int    quiz4;
   int    totalQuiz;
   int    midterm1;
   int    midterm2;
   int    totalMidterm;
   int    final;

   float  quizPercent;
   float  midtermPercent;
   float  finalPercent;
   float  totalPercent;

// Statements
   printf("=========== QUIZZES =================\n");
   printf("Enter the score for the first quiz:  ");
   scanf("%d", &quiz1);
   printf("Enter the score for the second quiz: ");
   scanf("%d", &quiz2);
   printf("Enter the score for the third quiz:  ");
   scanf("%d", &quiz3);
   printf("Enter the score for the fourth quiz: ");
   scanf("%d", &quiz4);

   printf("============= MIDTERM  =============\n");
   printf("Enter the score for the first midterm:  ");
   scanf("%d", &midterm1);
   printf("Enter the score for the second midterm: ");
```

continued

PROGRAM 3-14 Calculate Student Score *(continued)*

```
   scanf("%d", &midterm2);

   printf("=============== FINAL ==============\n");
   printf("Enter the score for the final: ");
   scanf("%d", &final);
   printf("\n");

   totalQuiz = quiz1 + quiz2 + quiz3 + quiz4;
   totalMidterm = midterm1 + midterm2;

   quizPercent =
      (float)totalQuiz * QUIZ_WEIGHT / QUIZ_MAX;
   midtermPercent =
      (float)totalMidterm * MIDTERM_WEIGHT / MIDTERM_MAX;
   finalPercent =
      (float)final * FINAL_WEIGHT / FINAL_MAX;

   totalPercent =
         quizPercent + midtermPercent + finalPercent;

   printf("First Quiz  %4d\n",   quiz1);
   printf("Second Quiz %4d\n",   quiz2);
   printf("Third Quiz  %4d\n",   quiz3);
   printf("Fourth Quiz %4d\n",   quiz4);
   printf("Quiz Total  %4d\n\n", totalQuiz);

   printf("First Midterm  %4d\n",   midterm1);
   printf("Second Midterm %4d\n",   midterm2);
   printf("Total Midterms %4d\n\n", totalMidterm);

   printf("Final          %4d\n\n", final);

   printf("Quiz    %6.1f%%\n" , quizPercent);
   printf("Midterm %6.1f%%\n" , midtermPercent);
   printf("Final   %6.1f%%\n" , finalPercent);
   printf("--------------\n");
   printf("Total   %6.1f%%\n" , totalPercent);

   return 0;
}  // main
```

```
Results:
=========== QUIZZES =================
Enter the score for the first quiz:  98
```

continued

PROGRAM 3-14 Calculate Student Score *(continued)*

```
    Enter the score for the second quiz: 89
    Enter the score for the third quiz:  78
    Enter the score for the fourth quiz: 79
    ============= MIDTERM  =============
    Enter the score for the first midterm:  90
    Enter the score for the second midterm: 100
    =============== FINAL ==============
    Enter the score for the final: 92

    First Quiz    98
    Second Quiz   89
    Third Quiz    78
    Fourth Quiz   79
    Quiz Total   344

    First Midterm    90
    Second Midterm  100
    Total Midterms  190

    Final            92

    Quiz      25.8%
    Midterm   38.0%
    Final     27.6%
    --------------
    Total     91.4%
```

Program 3-14 Analysis This rather long program contains several points to consider. First, note how the program starts with a series of defined constants. Putting the definitions of constant values at the beginning of the program does two things: (1) It gives them names that we can use in the program, and (2) it makes them easy to change.

Now study the statements. Notice how they are grouped? By putting a blank line between a group of related statements, we separate them, much as we would separate paragraphs in a report. This makes it easy for the user to follow the program.

Finally, study the input and output. Notice that the user was prompted for all input with clear instructions. We even divided the input with headings. The output is also divided, making it easy to read. It would be even easier to read if we aligned all the amounts, but the techniques for doing so will not be introduced until Chapter 7.

3.8 Software Engineering

In this section we discuss three concepts that, although technically not engineering principles, are important to writing clear and understandable programs.

KISS

Keep It Simple and Short (**KISS**[4]) is an old programming principle. Unfortunately, many programmers tend to forget it, especially the simple part. They seem to feel that just because they are working on a complex problem, the solution must be complex, too. That is simply not true. Good programmers solve the problem in the simplest possible way; they do not contribute to a complex situation by writing obscure and complex code.

A trivial example will make the point. If you were writing a program that reads floating-point numbers from the keyboard, you would not program it so that the user had to enter the integral portion of the number first and then the fractional part. Although this would work, it is unnecessarily complex, even though it might be a fun way to solve the problem.

Unfortunately, C provides many operators and expression rules that make it easy for a programmer to write obscure and difficult to follow code. Your job as a programmer is to make sure that your code is always easy to read. Your code should be unambiguous: It should not be written so that it is easy to misread it.

Another old structured programming principle is that a function should not be larger than one page of code. Now, for online programming in a workstation environment, a function should be no longer than one screen—about 20 lines of code. By breaking a problem down into small, easily understood parts, we simplify it. Then we reassemble the simple components into a simple solution to a complex problem.

Blocks of code should be no longer than one screen.

One element of the C language that tends to complicate programs, especially for new programmers, is side effects. We explained in Section 3.4, "Evaluating Expressions," that side effects can lead to confusing and different results depending on the code. You need to fully understand the effects when you write C code. If you are unsure of the effects, then simplify your logic until you are sure.

Parentheses

One programming technique to simplify code is to use parentheses, even when unnecessary. While this may lead to a few extra keystrokes, it can save hours of debugging time created by a misunderstanding of the precedence

4. KISS originally had a different, rather insulting, meaning. We prefer this interpretation.

and associativity rules. Whenever a statement contains multiple expressions, use parentheses to ensure that the compiler will interpret it as you intended.

Computers do what you *tell* them to do, not what you *intended* to tell them to do. Make sure your code is as clear and simple as possible.

User Communication

You should always make sure you communicate with your user from the very first statement in your program to the very last. As mentioned previously, we recommend that you start your program with a message that identifies the program and end with a display that says the program is done.

When you give your user instructions, make sure that they are clear and understandable. In Program 3-13, we used three statements to give the user complete and detailed instructions on what we wanted entered. We could have simply said

```
"Enter data"
```

but that would have been vague and subject to interpretation. How would the user know what specific data were required? For each input in Program 3-13, we told the users exactly what data we needed in terms that they understand. If you don't tell users exactly what data to input, they may do anything they feel like, which is usually not what you wanted or expected.

One common mistake made by new programmers is to forget to tell the user anything. What do you think would be the user's response to Program 3-15 when confronted with a blank screen and a computer that is doing nothing?

PROGRAM 3-15 Program That Will Confuse the User

```
#include <stdio.h>
int main (void)
{
   int  i;
   int  j;
   int  sum;

   scanf("%d%d", &i, &j);
   sum = i + j;
   printf("The sum of %d & %d is %d\n", i, j, sum);
   return 0;
}  //  main
```

We now rewrite the program with clear user communication. With the addition of one print statement, the user knows exactly what is to be done.

PROGRAM 3-16 Program That Will Not Confuse the User

```c
#include <stdio.h>
int main (void)
{
   int  i;
   int  j;
   int  sum;

   printf("Enter two integers and key <return>\n");
   scanf("%d%d", &i, &j);
   sum = i + j;
   printf("The sum of %d & %d is %d\n", i, j, sum);
   return 0;
}  //  main
```

```
Results:
Enter two integers and key <return>
4 5
The sum of 4 & 5 is 9
```

We will return to these three concepts from time to time when we introduce new structures that tend to be confusing or misunderstood.

3.9 Tips and Common Errors

1. Be aware of expression side effects. They are one of the main sources of confusion and logical errors in a program.
2. Use decrement/increment operators wisely. Understand the difference between postfix and prefix decrement/increment operators before using them.
3. Add parentheses whenever you feel they will help to make an expression clearer.
4. It is a compile error to use a variable that has not been defined.
5. It is a compile error to forget the semicolon at the end of an expression statement.
6. It is a compile error to code a variable declaration or definition once you have started the statement section. To help yourself remember this rule, use comments to separate these two sections within a function.
7. It is most likely a compile error to terminate a defined constant (*#define*) with a semicolon. This is an especially difficult error to decipher because you will not see it in your code—you see the code you wrote, not the code that the preprocessor substituted.
8. It is a compile error when the operand on the left of the assignment operator is not a variable. For example, `a + 3` is not a variable and cannot receive the value of `b * c`.

```
(a + 3) = b * c;
```

9. It is a compile error to use the increment or decrement operators with any expression other than a variable identifier. For example, the following code is an error:

```
(a + 3)++
```

10. It is a compile error to use the modulus operator (`%`) with anything other than integers.
11. It is a logic error to use a variable before it has been assigned a value.
12. It is a logic error to modify a variable in an expression when the variable appears more than once.

3.10 Key Terms

additive expression
assignment expression
associativity
left-to-right associativity
multiplicative expression
name

binary expression
block
cast
complex expression
compound statement
conversion rank
demotion
explicit type conversion
expression
expression statement
implicit type conversion
KISS
operand
operator
postfix expression
precedence
primary expression
promotion
right-to-left associativity
side effect
simple expression
unary expression
user prompts

3.11 Summary

- An expression is a sequence of operators and operands that reduces to a single value.
- An operator is a language-specific token that requires an action to be taken.
- An operand is the recipient of the action.
- C has six kinds of expressions: primary, postfix, prefix, unary, binary, and ternary.
- The most elementary type of expression is a primary expression. A primary expression is an expression made up of only one operand. It can be a name, a constant, or a parenthesized expression.
- A postfix expression is an expression made up of an operand followed by an operator. You studied function call and postfix increment/decrement expressions in this chapter.
- A unary expression is an expression made up of an operator followed by an operand. You studied six in this chapter: prefix increment/decrement, *sizeof*, plus/minus, and cast expressions.
- A binary expression is an expression made up of two operands with an operator between them. You studied multiplicative, additive, and assignment expressions in this chapter.
- Precedence is a concept that determines the order in which different operators in a complex expression act on their operands.
- Associativity defines the order of evaluation when operators have the same precedence.
- The side effect of an expression is one of the unique phenomena in C. An expression can have a side effect in addition to a value.

- To evaluate an expression, we must follow the rules of precedence and associativity.
- A statement causes an action to be performed by the program.
- Although C has eleven different types of statements, you studied only four types in this chapter:
  1. A null statement is just a semicolon.
  2. An expression statement is an expression converted to a statement by keeping the side effect and discarding the value.
  3. A return statement terminates a function.
  4. A compound statement is a combination of statements enclosed in two braces.
- KISS means "Keep It Simple and Short."
- One of the important recommendations in software engineering is the use of parentheses when they can help clarify your code.
- Another recommendation in software engineering is to communicate clearly with the user.

3.12 Practice Sets

Review Questions

1. A unary expression consists of only one operand with no operator.
   - a. True
   - b. False
2. The left operand in an assignment expression must be a single variable.
   - a. True
   - b. False
3. Associativity is used to determine which of several different expressions is evaluated first.
   - a. True
   - b. False
4. Side effect is an action that results from the evaluation of an expression.
   - a. True
   - b. False
5. An expression statement is terminated with a period.
   - a. True
   - b. False

6. A(n) ________________ is a sequence of operands and operators that reduces to a single value.

   a. expression
   b. category
   c. formula
   d. function
   e. value

7. Which of the following is a unary expression?

   a. `i + j`
   b. `+a`
   c. `c++`
   d. `scanf(…)`
   e. `x *= 5`

8. The ________________ expression evaluates the operand on the right side of the operator and places its value in the variable on the left side of the operator.

   a. additive
   b. assignment
   c. multiplicative
   d. postfix
   e. primary

9. ________________ is used to determine the order in which different operators in a complex expression are evaluated.

   a. associativity
   b. evaluation
   c. category
   d. precedence
   e. side effect

10. ________________ is an action that results from the evaluation of an expression.

    a. associativity
    b. evaluation
    c. category
    d. precedence
    e. side effect

11. Which of the following statements about mixed expressions is false?

    a. A cast cannot be used to change an assigned value.
    b. An explicit cast can be used to change the expression type.
    c. An explicit cast on a variable changes its type in memory.
    d. An implicit cast is generated by the compiler automatically when necessary.
    e. Constant casting is done by the compiler automatically.

12. Which of the following statements about compound statements is false?
    - a. A compound statement is also known as a block.
    - b. A compound statement is enclosed in a set of braces.
    - c. A compound statement must be terminated by a semicolon.
    - d. The declaration and definition section in a compound statement is optional.
    - e. The statement section in a compound statement is optional.

Exercises

13. Which of the following expressions are not postfix expressions?
    - a. `x++`
    - b. `--x`
    - c. `scanf (…)`
    - d. `x * y`
    - e. `++x`
14. Which of the following are not unary expressions?
    - a. `++x`
    - b. `--x`
    - c. `sizeof (x)`
    - d. `+5`
    - e. `x = 4`
15. Which of the following is not a binary expression?
    - a. `3 * 5`
    - b. `x += 6`
    - c. `y = 5 + 2`
    - d. `z - 2`
    - e. `y % z`
16. Which of the following is not a valid assignment expression?
    - a. `x = 23`
    - b. `4 = x`
    - c. `y % = 5`
    - d. `x = 8 = 3`
    - e. `x = r = 5`
17. If originally `x = 4`, what is the value of `x` after the evaluation of the following expression?
    - a. `x = 2`
    - b. `x += 4`
    - c. `x += x +3`
    - d. `x *= 2`
    - e. `x /= x +2`

18. If originally `x = 3` and `y = 5`, what is the value of `x` and `y` after each of the following expressions?

    a. `x++ + y`
    b. `++x`
    c. `x++ + y++`
    d. `++x + 2`
    e. `x-- - y--`

19. What is the value of each of the following expressions?

    a. `24 - 6 * 2`
    b. `-15 * 2 + 3`
    c. `72 / 5`
    d. `72 % 5`
    e. `5 * 2 / 6 + 15 % 4`

20. What is the value of each of the following expressions?

    a. `6.2 + 5.1 * 3.2`
    b. `2.0 + 3.0 / 1.2`
    c. `4.0 * (3.0 + 2.0 / 6.0)`
    d. `6.0 / (2.0 + 4.0 * 1.2)`
    e. `2.7 + 3.2 - 5.3 * 1.1`

21. Given the following definitions, which of the following statements are valid assignments?

```
#define NUM10 10
int x;
int y = 15;
```

    a. `x = 5;`
    b. `y = 5;`
    c. `x = y = 50;`
    d. `x = 50 = y;`
    e. `x = x + 1;`
    f. `y = 1 + NUM10;`
    g. `5 = y;`

22. If originally `x = 2, y = 3`, and `z = 2`, what is the value of each of the following expressions?

    a. `x++ + y++`
    b. `++x - --z`
    c. `--x + y++`
    d. `x-- + x-- - y--`
    e. `x + y-- - x + x++ - --y`

23. If originally `x = 2, y = 3`, and `z = 1`, what is the value of each of the following expressions?

    a. `x + 2 / 6 + y`
    b. `y - 3 * z + 2`
    c. `z - (x + z) % 2 + 4`

   d. `x - 2 * (3 + z) + y`
   e. `y++ + z-- + x++`

24. If `x = 2945`, what is the value of each of the following expressions?

   a. `x % 10`
   b. `x / 10`
   c. `(x / 10) % 10`
   d. `x / 100`
   e. `(x / 100) % 10`

25. What is the output from the following code fragment?

```
int a;
int b;
a = b = 50;
printf ("%4d %4d", a, b);
a = a * 2;
b = b / 2;
printf ("%4d %4d", a, b);
```

Problems

26. Given the following pseudocode, write a program that executes it. Use floating-point types for all values.

```
Algorithm Problem26
 1 read x
 2 read y
 3 compute p = x * y
 4 compute s = x + y
 5 total = $s^2$ + p * (s - x) * (p + y)
 6 print total
End Problem26
```

27. Write a program that reads two integers from the keyboard, multiplies them, and then prints the two numbers and their product.

28. Write a program that extracts and prints the rightmost digit of the integral portion of a *float*.

29. Write a program that extracts and prints the second rightmost digit of the integral portion of a *float*.

30. Write a program that calculates the area and perimeter of a rectangle from a user-supplied (*scanf*) length and width.

31. We are all familiar with the fact that angles are measured in degrees, minutes, and seconds. Another measure of an angle is a radian. A radian is the angle formed by two radii forming an arc that is equal to the radius of their circle. One radian equals 57.295779 degrees. Write a program that converts degrees into radians. Provide good user prompts. Include the following test data in your run:

```
90° is 1.57080 radians
```

32. The formula for converting centigrade temperatures to Fahrenheit is:

$$F = 32 + \left(C \times \left(\frac{180.0}{100.0}\right)\right)$$

Write a program that asks the user to enter a temperature reading in centigrade and then prints the equivalent Fahrenheit value. Be sure to include at least one negative centigrade number in your test cases.

33. Write a program that changes a temperature reading from Fahrenheit to Celsius using the following formula:

```
Celsius = (100 / 180) * (Fahrenheit - 32)
```

Your program should prompt the user to enter a Fahrenheit temperature. It then calculates the equivalent Celsius temperature and displays the results as shown below.

```
Enter the temperature in Fahrenheit: 98.6
Fahrenheit temperature is:  98.6
Celsius temperature is:     37.0
```

34. Write the C code for each of the following formulas. Assume that all variables are defined as *double*.

a.

$$\text{KinEn} = \frac{mv^2}{2}$$

b.

$$\text{res} = \frac{b + c}{2bc}$$

35. Write the C code to calculate and print the next two numbers in each of the following series. You may use only one variable in each problem.

a. 0, 5, 10, 15, 20, 25, ?, ?
b. 0, 2, 4, 6, 8, 10, ?, ?
c. 1, 2, 4, 8, 16, 32, ?, ?

Projects

36. Write a program that converts and prints a user-supplied measurement in inches into

    a. foot (12 inches)
    b. yard (36 inches)
    c. centimeter (2.54/inch)
    d. meter (39.37 inches)

37. A Fibonacci number is a member of a set in which each number is the sum of the previous two numbers. (The Fibonacci series describes a form of a spiral.) The series begins

```
0, 1, 1, 2, 3, 5, 8, 13, 21, …
```

Write a program that calculates and prints the next three numbers in the Fibonacci series. You are to use only three variables, `fib1`, `fib2`, and `fib3`.

38. Write a program that prompts a user for an integer value in the range 0 to 32,767 and then prints the individual digits of the numbers on a line with three spaces between the digits. The first line is to start with the leftmost digit and print all five digits; the second line is to start with the second digit from the left and print four digits, and so forth. For example, if the user enters 1234, your program should print

```
0   1   2   3   4
1   2   3   4
2   3   4
3   4
4
```

39. Write a program to create a customer's bill for a company. The company sells only five different products: TV, VCR, Remote Controller, CD Player, and Tape Recorder. The unit prices are $400.00, $220, $35.20, $300.00, and $150.00, respectively. The program must read the quantity of each piece of equipment purchased from the keyboard. It then calculates the cost of each item, the subtotal, and the total cost after an 8.25% sales tax.

The input data consist of a set of integers representing the quantities of each item sold. These integers must be input into the program in a user-friendly way; that is, the program must prompt the user for each quantity as shown below. The numbers in boldface show the user's answers.

```
How Many TVs Were Sold? 3
How Many VCRs Were Sold? 5
How Many Remote Controllers Were Sold? 1
How Many CDs Were Sold? 2
How Many Tape Recorders Were Sold? 4
```

The format for the output is shown in Figure 3-13.

```
QTY     DESCRIPTION      UNIT PRICE    TOTAL PRICE
---     -----------      ----------    -----------
XX      TV                   400.00        XXXX.XX
XX      VCR                  220.00        XXXX.XX
XX      REMOTE CTRLR          35.20        XXXX.XX
XX      CD PLAYER            300.00        XXXX.XX
XX      TAPE RECORDER        150.00        XXXX.XX
                                         ---------
                           SUBTOTAL       XXXXX.XX
                           TAX             XXXX.XX
                           TOTAL          XXXXX.XX
```

FIGURE 3-13 Output Format for Project 39

Use either defined constants or memory constants for the unit prices and the tax rate. Use integer variables to store the quantities for each item. Use floating-point variables to store the total price for each item, the bill subtotal, the tax amount, and the total amount of the bill. Run your program twice with the following data:

```
SET 1 → 2  1  4  1  2
SET 2 → 3  0  2  0 21
```