PART II

Computation

Chapter 4

Expressions

Computation is central to computer programming. We discuss how the usual mathematical operations can be applied to variables and introduce the concepts of precedence and associativity that govern the meaning of an expression in both mathematical notation and in C. Parse trees are introduced to help explain the structure and meaning of expressions.

Finally, declarations, expressions, and parse trees are brought together in a discussion of program design and testing methodology. We propose a problem, develop a program for it, analyze how we should test the program, and show how to use data type rules and parse trees to find and correct an error in the coding.

4.1 Expressions and Parse Trees

An **expression** is like an entire sentence in English. It specifies how verbs (operators and functions) should be applied to nouns (variables and constants) to produce a result. In this section, we consider the rules for building expressions out of names, operators, and grouping symbols and how to interpret the meanings of those expressions. We use a kind of diagram called a *parse tree* to see the structure of an expression and to help us understand its meaning.

4.1.1 Operators and Arity

C has several classes of operators in addition to those that perform arithmetic on numbers. These include comparison and logical operators, bit-manipulation operators, and a variety of operators that are found in C but not in other languages. Each group of operators is intended for use on a particular type of object. Many will be introduced in this chapter; others will be discussed in detail in the chapters that introduce the relevant data types. For example, operators that work specifically with integers are explained in Chapter 7. Operators that deal with pointers, arrays, and structured types are left for later chapters. Issues to be considered with each class of operator include the type of data on which it operates, its precedence and associativity, any unusual rules for evaluation order, and any unexpected aspects of the meaning of an operator.

An **operand** is an expression whose value is an input to an operation. *Operators* are classed as unary, binary, or ternary according to the number of operands each takes. A **binary operator** has two operands and is written between those operands. For example, in the expression `(z/4)`, the `/` sign is a binary operator and `z` and `4` are its operands. We also say that a binary operator has arity = 2. The **arity** of an operator is the number of operands it requires.

A unary operator (arity = 1), such as `-` (negate), has one operand. Most unary operators are **prefix operators**; that is, they are written before their operand. Two unary operators, though, also may be written after the operand, as **postfix operators**. There is only one ternary operator (arity = 3), the conditional expression. The two parts of this operator are written between its three operands.

4.1.2 Precedence and Parentheses

Many expressions contain more than one operator. In such expressions, we need to know which operands go with which operators. For example, if we write

$$13/5 + 2$$

it is important to know whether this means

$$(13/5) + 2 \qquad \text{or} \qquad 13/(5 + 2)$$

because the two expressions have different answers.

Using parentheses. Parentheses are neither operators nor operands; called *grouping symbols*, they can be used to control which operands are associated with each operator. Parentheses can be used to specify whatever meaning we want; a fully parenthesized expression has only one interpretation. If we do not write parentheses, C uses a default interpretation based on the rules for precedence and associativity. This default is one of the most common sources for errors in writing expressions.

The precedence table and the default rules. Experienced programmers use parentheses only to emphasize the meaning of an expression or achieve a meaning different from the default one. They find it a great convenience to be able to omit parentheses most of the time. Further, using parentheses selectively makes expressions more readable. However, if you use the default precedence, anyone reading or writing your program must be familiar with the rules to understand the meaning of the code.

The precedence and associativity of each operator are defined by the C precedence table, given in Appendix **??**. The rules in this table describe the meaning of any unparenthesized part of an expression. **Precedence** defines the grouping order when different operators are involved, and **associativity** defines the order of grouping among adjacent operators that have the same precedence. The concepts and rules for precedence and associativity are simple and mechanical. They can be mastered even if you do not understand the meaning or use of a particular operator. The portion of the **precedence table** that covers the arithmetic operators is repeated in Figure 4.1, with a column of examples added on the right. We use it in the following discussion to illustrate the general principles.

Precedence. In the precedence table, the C operators are listed in order of precedence; the ones at the top are said to have *high precedence*; those at the bottom have *low precedence*.[1] As a convenience, the precedence classes in C also have been numbered. There are 17 different classes. (The higher the number, the higher is the precedence.) Operators that have the same precedence number are said to have *equal precedence*. You will want to mark Appendix B so that you can refer to the precedence table easily; until all the operators are familiar, you will need to consult it often.

Associativity. The rule for associativity governs consecutive operators of equal precedence, such as those in the expression `(3 * z / 10)`. All these operators with the same precedence will have the same associativity, which is either left to right or right to left.[2] With left-to-right associativity, the leftmost operator is parsed before the ones to its right. (The process of parsing is explained next.) With right-to-left associativity, the rightmost operator is parsed first. In the expression `(3 / z * 10 % 3)`, the three operators have the same precedence and all have left-to-right associativity. We therefore parse the `/` first and the `*` second, giving this result: `(((3 / z) * 10) % 3)`. The parse of this expression is diagrammed in Figure 4.3.

Almost all the unary operators have the same precedence (15) and are written before the operand (that is, in prefix position). The chart shows that they associate right to left. Therefore, in the expression `(- - x)`, the second negation operation is parsed first and the leftmost negation operation second, giving this result: `(- (- x))`. Restated simply, this rule states "the prefix operator written closest to the operand is parsed first."

[1] Most of the C operators will be described in this chapter; others will be explained in later chapters.

[2] The term *left-to-right associativity* is often shortened to *left associativity* and *right-to-left* to *right*.

Arity	Symbol	Meaning	Precedence	Associativity	Examples
Unary	-	Negation	15	right to left	-1, -temp
	+	No action	15	"	+1, +x
Binary	*	Multiplication	13	left to right	3 * x
	/	Division	13	"	x / 3.0
	%	Integer remainder (modulus)	13	"	k % 3
	+	Addition	12	"	x + 1, x + y
	-	Subtraction	12	"	x - 1, 3 - y

Figure 4.1. Arithmetic operators.

4.1.3 Parsing and Parse Trees

Parsing is the process of analyzing the structure of a statement. The compiler does this when it translates the code, and human beings do it when reading the code. One way to parse an expression is to write parentheses where the precedence and associativity rules would place them by default. This process is easy enough to carry out but can become visually confusing.

An easier way to parse is to draw a *parse tree*. The **parse tree** shows the structure of the expression and is closely related to what the C compiler does when it translates the expression. A parse tree helps us visualize the structure of an expression. It also can be used to understand the order in which operators will be executed by C, which, unfortunately, is not the same as their order of precedence.[3]

To parse an operator, either write parentheses around it and its operands or draw a tree bracket with one branch under each operand and the stem under the operator. A simple, two-pronged bracket is used for most binary operators. The bracketed or parenthesized unit becomes a single operand for the next operator, as shown in Figures 4.2 and 4.3. Brackets or parenthesized units never collide or cross each other. Figure 4.2 shows a parse tree that uses the precedence rules, and Figure 4.3 shows an expression where all operators have equal precedence and the rule for associativity is used. The steps in parsing an arbitrary expression are these:

1. Write the expression at the top, leaving some space between each operand and operator.
2. Parse parenthesized subexpressions fully before looking at the surrounding expression.
3. If there are postfix unary operators (`--` or `++`), do them first. These are mentioned here for completeness and will be explained later in the chapter.
4. Next, find the prefix unary operators, all of which associate right to left. Start at the rightmost and bracket each one with the following operand, drawing a stem under the operator. (An operand may be a variable, a literal, or a previously parsed unit.)

[3]Complications of evaluation order are covered in Sections 4.3 and 4.6.3 and Appendix **??**.

The `*` operator has highest precedence so it was parsed first and it "captured" the middle operand. The assignment has lowest precedence so it was parsed last. The arrowhead on the assignment operator bracket indicates that the value of `x` is updated with the right operand value.

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

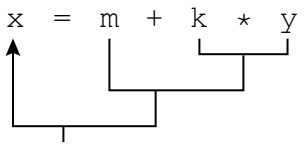


Figure 4.2. Applying precedence.

In the diagram, three brackets are used to show which operands go with each of the operators in the expression `(3 / z * 10 % 3)`. These operators have equal precedence, so they are parsed according to the associativity rule for this group of operators, which is left to right.

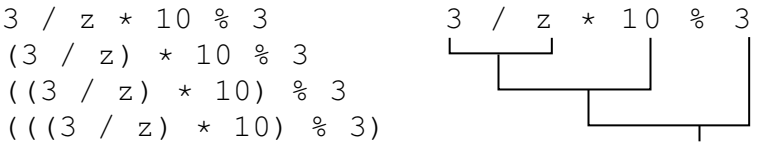


Figure 4.3. Applying associativity.

5. Now look at the unparsed operators in the expression and look up the precedence of each one in the table. Parse the operators of highest precedence first, grouping each with its neighboring operands. Then go on to those of lower precedence. For neighboring operators of the same precedence, find their associativity in the table. Proceeding in the direction specified by associativity, bracket each operator with the two adjacent operands.
6. Repeat this process until you finish all of the operators.

Figure 4.4 shows an example of parsing an expression where all the rules are applied: grouping, associativity, and precedence.

Notes on Figure 4.4. Applying the parsing rules. The steps in drawing this parse tree are as follows:

1. Subexpressions in parentheses must be parsed first. Within the parentheses, the prefix operator `-` is parsed first. Then the lower precedence operator, `+`, uses this result as its left operand to complete the parsing in the parentheses.
2. The `*` and `/` are the operators with the next highest precedence; we parse them left to right, according to their associativity rule.
3. The `-` is next; its left operand is the result of `/` and its right operand is the result of the parentheses grouping.
4. The `=` is last; it stores the answer into `x` and returns the value as the result of the expression (this is explained in the next section).

4.2 Arithmetic, Assignment, and Combination Operators

The **arithmetic operators** supported by `C` are listed in Figure 4.1. All these operators can be used on any pair of numeric operands. The **assignment operator** was described and its uses discussed in Chapter 3. These operators can be combined to give a shorthand notation for doing both actions. For example, the expression `(k += 3.5)` has the same meaning as `(k = k + 3.5)`. It means "fetch the current value of `k`, add 3.5, and store the result back into `k`." The symbol for a **combination operator** is formed by writing the desired operator symbol followed by an assignment sign, as in `+=` or `*=`. The arithmetic combinations are

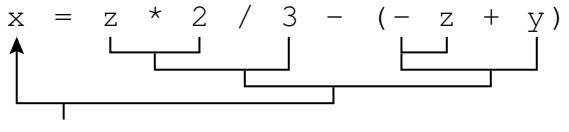


Figure 4.4. Applying the parsing rules.

All these operators associate right to left. The left operand of each must be a variable or storage location. The parse diagram for a combination operator is shown in Figure 4.6

Symbol	Example	Meaning	Precedence
`=`	`x = 2.5`	Store value in variable	2
`+=`	`x += 3.1`	Same as `(x = x + 3.1)`; add value to variable and store back into variable	2
`-=`	`x -= 1.5`	Same as `(x = x - 1.5)`; subtract value from variable and store back into variable	2
`*=`	`x *= 5`	Same as `(x = x * 5)`; multiply and store back	2
`/=`	`x /= 2`	Same as `(x = x / 2)`; divide by and store back	2

Figure 4.5. Assignment and arithmetic combinations.

listed in Figure 4.5, and a parse tree for assignment combinations in Figure 4.6. All combination operators have the same very low precedence, which means that other operators in the expression will be parsed first. If more than one assignment or combination operator is in an expression, these operators will be parsed and executed right to left.

Side effects. The assignment combinations, along with ordinary assignment and the increment and decrement operators (described in the next section), are different from all other operators: They have the **side effect** of changing the value of a memory variable. Assignment discards the old value and replaces it with a new value. The combination operators perform a calculation using the value of a memory variable then store the answer back into the variable. When using a combination operator, the right operand may be a variable or an expression, but the left operand must be a variable, since the answer will be stored in it.

Using combination operators. Figure 4.7 shows how two of the arithmetic combinations can be used in a loop. It is a preliminary version of an algorithm for expressing a number in any selected number base; the complete version is shown in Figure 4.23.

Notes on Figure 4.7. Arithmetic combinations.

Outer box.
This loop is slightly different from the counting loops and validation loops seen so far. It simply continues a process that decreases `n` until the terminal condition, `n` equals 0, occurs. More loop types are discussed in Chapter 6.

This is the parse tree and evaluation for the expression `k += m`. Note that the combined operator is diagrammed with two brackets. The upper bracket is for `+`, which adds the initial value of `k` to the value of `m`. The lower bracket shows that the sum is stored back into `k`.

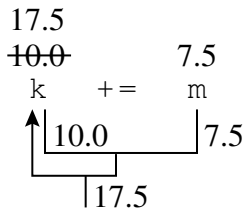


Figure 4.6. Parse tree for an assignment combination.

Each time we halve a number, we eliminate one binary digit. If we halve it repeatedly until it reaches 0 and count the iterations, we know how many binary digits are in the number.

```
#include <stdio.h>

int main( void )
{
    int n;          // Input - the number to analyze.

    puts( "\n Halving and Counting\n " );
    printf( " Enter an integer: " );
    scanf( "%i", &n );
    printf( " Your number is %i,", n);

    int k = 0;      // Initialize the counter before the loop.
    while (n > 0) { // Eliminate one binary digit each time around loop.
        n /= 2;     // Divide n in half and discard the remainder.

        k += 1;     // Count the number of times you did this.
    }

    printf( " it has %i bits when written in binary. \n\n", k );
    return 0;
}
```

Output:

```
Halving and Counting

Enter an integer:  37
Your number is 37, it has 6 bits when written in binary.
```

Figure 4.7. Arithmetic combinations.

First inner box: halving the number.
The statement `n /= 2;` divides `n` by 2, throwing away the remainder. The quotient is an integer (with no fractional part) and is stored back into `n`. Therefore, the value of `1/2` is 0, and the value of `7/2` is 3. Integer arithmetic is discussed more fully in Chapter 7.

Second inner box: incrementing the counter.
The operator `+=` is often used to increment loop counters, especially when counting by twos (or any increment not equal to 1). As seen in the next section, the operator `++` is more popular for adding 1 to a counter.

4.3 Increment and Decrement Operators

C contains four operators that are not present in most other languages because of the problems they cause, but they are very popular with C programmers because they are so convenient. These are the pre- and postincrement (`++`) and pre- and postdecrement (`--`) operators, which let us add or subtract 1 from a variable with only a few keystrokes. The increment operator most often is used in loops, to add 1 to the loop counter. Decrement sometimes is used to count backward. Both normally are used with integer operands, but they also work with variables of type `double`.

The same symbols, `++` and `--`, are used for both pre- and postincrement, but the former is written before the operand and the latter after the operand. The actions they stand for are the same, too, except that these actions are executed in a different order. Collectively, we call this group the *increment operators*; they are listed in Figure 4.8.

The increment and decrement operators cause side effects; that is, they change values in memory. The single operand of these operators must be a variable or storage location. The prefix operators both associate right to left; the postfix operators associate left to right.

Fixity	Symbol	Example	Meaning	Precedence
Postfix	`++`	`j++`	Use the operand's value in the expression, then add 1 to the variable	16
	`--`	`j--`	Use the operand's value in the expression, then subtract 1 from the variable	16
Prefix	`++`	`++j`	Add 1 to the variable then use the new value in the expression	15
	`--`	`--j`	Subtract 1 from the variable then use the new value in the expression	15

Figure 4.8. Increment and decrement operators.

4.3.1 Parsing Increment and Decrement Operators

The increment operators have two properties that, when put together, make them unique from the operators considered earlier. First, they are unary operators, and second, they modify memory. The parse diagrams for these operators reflect these differences.

To diagram an increment or decrement operator, bracket it with its single operand (which must be a variable or a reference to a variable), drawing a stem under the operator and an assignment arrowhead to show that the variable will receive a new value. That value also is passed on, down the parse tree, and can be used later in the expression. See Figure 4.9 for examples.

When postincrement or postdecrement is used, it is also possible to have prefix unary operators applied to the same operand. In this case, the postfix operator is parsed first (it has higher precedence), then the prefix operators are done right to left. Thus, the expression `(- x ++)` parses to `(- (x ++))`[4].

4.3.2 Prefix versus Postfix Operators

For simplicity, we will explain the differences between pre- and postfix operators in terms of the increment operators, but everything is equally true of decrement operators; simply change "add 1" to "subtract 1" in the explanations.

Both *prefix* and *postfix increment* operators add 1 to a variable and return the value of the variable for further use in an expression. For example, `k++` and `++k` both increase the value of `k` by 1. If a prefix

[4]The Applied C website, Program 4.A, shows a typical use of preincrement in a counting loop.

The parse trees are shown for each of these operators. The arrowheads indicate that a value is stored back into memory.

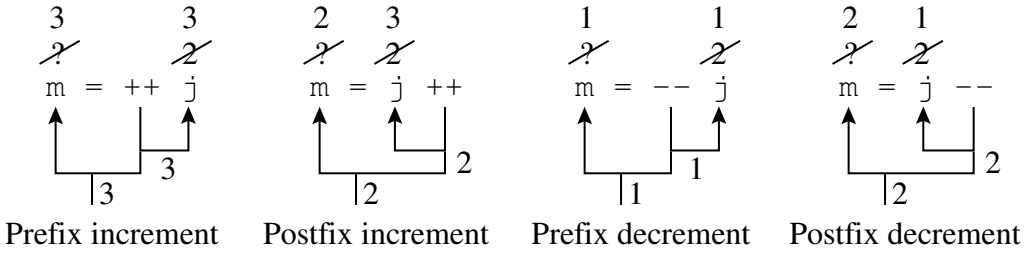


Figure 4.9. Increment and decrement trees.

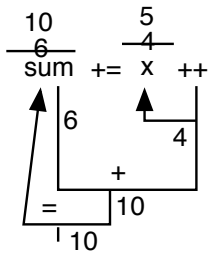


Figure 4.10. Evaluating an increment expression.

increment or a postfix increment operator is the only operator in an expression, the results will be the same. However, if an increment operation is embedded in a larger expression, the result of the larger expression will be different for prefix increment and postfix increment operators. Both kinds of increment operator return the value of `k`, to be used in the larger expression. However, the prefix form increments the variable *before* it returns the value, so that the value in memory and the one used in the expression are the same. In contrast, postfix increment returns the original, unincremented value of the variable to the larger expression and increments the value in memory *afterward*. Thus, the value used in the surrounding expression is 1 smaller than the value left in memory and 1 smaller than the value used in the prefix increment expression.

A further complication with postfix increment is that the change in memory does not have to happen right away. The compiler is permitted to postpone the store operation for a while, and many do postpone it in order to generate more efficient code.[5] This makes postfix increment and decrement somewhat tricky to use properly. However, you can depend on two things: First, if you execute `x++` three times, the value of `x` will be 3 greater than when you started. Second, by the time evaluation reaches the semicolon at the end of the statement, all incrementing and decrementing actions will be complete.

4.3.3 Mixing increment or decrement with other operators.

An increment operator is an easy, efficient way to add 1 to the value of a memory variable. Most frequently, increment and decrement operators are used in isolation. However, both also can be used in the middle of an expression because both return the value of the variable for use in further computation.

New programmers often write one line of code per idea. For example, consider the summation loop below. The first line of the loop body increments `x` and the second line uses the new value:

```
x = 1;             // Sum x for x = 1 to N
while (x <= N) {   // Leave loop when x exceeds N.
    sum += x;      // Same as sum = sum + x
    ++x;           // Add 1 to x to prepare for next iteration.
}                  // Go back to the while test.
```

An advanced programmer is more likely to write the following version, which increments `x` and uses it in the same line. The parse tree and evaluation of the assignment expression are shown in Figure 4.10, for the fourth time through the summing loop.

```
x = 1;             // Sum x for x = 1 to N
while (x < =N) sum += x++;
```

As you can see, mixing an increment with other operators in an expression makes the code "denser"—more actions happen per line—and it often permits us to shorten the code. By using two side-effect operators, we have reduced the entire loop to one line. However, this happens at the expense of some clarity and, for beginners, at the risk of writing something unintended.

[5]The exact rules for this postponement are complex. To explain them, one must first define sequence points and how they are used during evaluation. This explanation is beyond the scope of the book.

When using an increment or decrement in an expression, the difference between the prefix and postfix forms is crucial. If you use the postfix increment operator, the value used in the rest of the expression will be one smaller than if you use the prefix increment. For example, the loop shown previously sums the fractions $1/1 \ldots 1/N$. If a postfix increment were used instead of the prefix increment, it would try to sum the fractions $1/0 \ldots 1/(N-1)$ instead. Of course, dividing by 0 is a serious error that causes immediate program termination on some systems and meaningless results on others. Therefore, think carefully about whether to use a prefix or postfix operator to avoid using the wrong value in the expression or leaving the wrong value in memory. When these operators are used in isolation, such problems do not occur.

Guidance. Because of the complications with side effects, increment and decrement operators can be tricky to use when embedded in a complex expression. They are used most often in isolation to change the value of a counter in a loop, as in Figure 4.23. The following are some guidelines for their use that will help beginners avoid problems:

1. Do not mix increment or decrement operators with other operators in an expression until you are an experienced programmer.
2. Do not ever use increment on the same variable twice in an expression. The results are unpredictable and may vary from compiler to compiler or even from program to program translated by the same compiler.
3. Until you are experienced and are sure of the semantics of postfix operators, use prefix increment and decrement operators instead. The are less confusing than postfix increment and decrement operators and cause less trouble for beginners.

4.4 The sizeof operator.

When we declare an object, we say what type it will be. Types (sometimes called **data types**) are like adjectives in English: The type of an object describes its size (number of bytes in memory) and how it may be used. It tells the compiler how much storage to allocate for it and whether to use integer operations, floating-point operations, or some other kind of operations on it.

Many types are built into the C language standard; each has its own computational properties and memory requirements. Each type has different advantages and drawbacks, which will be examined in the next few chapters. Later, we will see how to define new types to describe objects that are more complex than simple letters and numbers.

The current popularity of C is based largely on its power and portability. However, the same data type can be different sizes on different machines, which adversely affects portability. For example, the type `int` commonly is two bytes on small personal computers and four bytes on larger machines. Also, some computer memories are built out of words not bytes.

The actual size of a variable, in bytes, becomes very important when you take a program debugged on one kind of system and try to use it on another; the variability in the size of a type can cause the program to fail. To address this problem, C provides the `sizeof` operator, which can be applied to any variable or any type to find out how big that type is in the local implementation. This is an important tool professional programmers use to make their programs portable. Figure 4.11 shows how to use `sizeof`. The output from this program will be different on different machines. It depends partly on the hardware and partly on the compiler itself. You should know what the answers are for any system you use. One of the exercises asks you to use `sizeof` to learn about your own compiler's types. The results from the program in Figure 4.11 are shown following the code. From this we can deduce that the workstation is using four bytes for an `int` and eight bytes for a `double`, which is usual. The literal constant e is represented as a `double`.

This short program demonstrates the proper syntax for using `sizeof`. Note that the parentheses are not needed for variables but they *are* needed for type names.

```
// ------------------------------------------------------------------------
// Demonstrate the use of sizeof to determine memory usage of data types

#include <stdio.h>
#define e 2.71828182845904523536028747135 3  // Mathematical constant e.

int main( void )                  // how to use sizeof
{
    int s_double, s_int, s_char, s_unknown, k;

    s_double = sizeof (double) ;  // use parentheses with a type name
    s_int = sizeof k ;            // no parentheses needed with a variable
    s_char = sizeof '%';

    printf( " sizeof double = %i \n sizeof int = %i \n sizeof char = %i \n",
    s_double, s_int, s_char );
    s_unknown = sizeof e ;
    printf( " sizeof e = %i \n", s_unknown );
    return 0;
}
```

Results when run on our workstation:

```
sizeof double = 8
sizeof int = 4
sizeof char = 1
sizeof e = 8
```

Figure 4.11. The size of things.

4.5 Relational Operators

The **relational operators** (`==`, `!=`, `<`, `>`, `<=`, and `>=`), with their meanings and precedence values, are listed in Figure 4.12. These operators perform comparisons on their operands and return an answer of `true` or `false`. To control program flow, we compare the values of certain variables to each other or to target values. We then use the result to select one of the clauses of an `if` statement or control a loop.

All operators listed here associate left to right.

Arity	Usage	Meaning	Precedence
Binary	`x < y`	Is `x` less than `y`?	10
	`x <= y`	Is `x` less than or equal to `y`?	10
	`x > y`	Is `x` greater than `y`?	10
	`x >= y`	Is `x` greater than or equal to `y`?	10
	`x != y`	Is `x` unequal to `y`?	9
	`x == y`	Is `x` equal to `y`?	9

Figure 4.12. Relational operators.

4.5.1 True and False

C supports two sets of operators that give true/false results: comparison operators (`==, !=, <, <=, >, >+`) and logical operators (`&&, ||, !`). The results of these operators are signed integers: `true` is represented by 1 and `false` is represented by 0.

Every integer has a ***truth value***. Values 0 and 1 are the canonical representations of *false* and *true*, but C will interpret any value as true or false if it is used in a context that requires a truth value. The zero values 0 and 0.0 and the zero character, `'\0'` are all represented by the same bit pattern as `0` and so all are interpreted as false. Any nonzero bit pattern is interpreted as *true*. For example, all of the following are `true: -1, 2.5, 'F"` and 367. Suppose we use one of these values as the entire condition of an `if` statement:

```
if ( -1 ) puts( "true" ) else puts( false" );
```

In human language, this makes no sense. However, to C, it means to find the truth value of `-1`, which is `true`. So, the answer "`true`" will be printed. Similarly, if we write

```
if ( 'F' ) puts( "true" ) else puts( false" );
```

The answer "`true`" will be printed, because `'F'` is not a synonym for zero.

Many languages (including Java and C++) have a Boolean type built into the language. It is used to represent the answers to yes-no questions such as, "Is the data value legal?" and "Are two objects the same?" For consistency with C++, the C99 standard introduced a standard type `bool` and defined the boolean values `true` and `false`. To use this type, some compilers require the program to `#include <stdbool.h>`.

A program in which the numeral `1` is used to represent both the number one and the truth value `true` can be difficult to read and interpret. For this reason, it is much better style to write `true` when you mean a truth value and reserve `1` for use as a number. We will use the symbols `false` and `true` for truth values instead of the numbers 0 and 1. We also adopt modern usage and declare truth-valued variables to have type `bool`.

4.5.2 The semantics of comparison.

A relational operator can be used to compare two values of any simple type[6] or two values that can be converted to the same simple type. The result is always an `int`, no matter what types the operands are, and it is always either a 1 (`true`) or a 0 (`false`).

A common error among inexperienced C programmers is to use the assignment operator `=` instead of the comparison operator `==`. The expression `x == y` is not the same as `x = y`. The second means "make `x` equal to the current value of `y`", while `x == y` means "compare the values of `x` and `y`". Therefore, if a comparison is done after the inadvertent assignment operation, of course, the values of the two variables *will be* equal. It may help you to pronounce these operators differently; we use "compares equal" for `==` and "gets" for `=`.

In other languages, such an error would be trapped by the compiler. In C, however, assignment is an "ordinary" operator that has precedence and associativity like other operators and actually returns a value (the same as the value it stores into memory). A C compiler has no way of knowing for sure whether a programmer meant to write a comparison or an assignment. Some compilers give a warning error comment when the `=` operator is used within the conditional part of an `if` or `while` statement; however, doing so is not necessarily an error, so the comment is only a warning, not fatal, and the compiler should produce executable code.

4.6 Logical Operators

Sometimes we want to test a more complicated condition than a simple equality or inequality. For example, we might need an input value between 0 and 10 or require two positive inputs that are not equal to each other. We can create compound conditions like these by using the logical operators `&&` (AND), `||` (OR), and `!`(NOT). The former condition can be written as `x >= 0 && x <= 10` and the latter as `x > 0 && y > 0 && x != y`. **Logical operators** let us test and combine the results of comparison expressions.

[6]Types `double` and `int` are simple. Nonsimple types are compounds with more than one part such as strings, arrays, and structures. These will be discussed in later chapters.

C uses the value 1 to represent `true` and 0 to represent `false`. The result of every comparison and logical operator is either 0 or 1. However, any value can be an input to a logical operator; all nonzero operands are interpreted as `true`. In the table, T represents any nonzero value.

Operands		**Results**		
`x`	`y`	`!x`	`x && y`	`x || y`
0	0	1	0	0
0	T	1	0	1
T	0	0	0	1
T	T	0	1	1

Figure 4.13. Truth table for the logical operators.

4.6.1 Truth-valued operators.

All the comparison and logical operators produce truth values as their results. For example, if you ask `x == y`, the answer is either 0 (`false`) or 1 (`true`). The meanings of `&&`, `||`, and `!` are summarized by the **truth table** in Figure 4.13. The first two columns of the *truth table* show all possible combinations of the truth values of two operands. `T` is used in these columns to mean `true`, because an operand can have any value, not just 1 or 0. The last three columns show the results of the three logical operations. In these columns, true answers are represented by 1 because C always uses the standard `true` to answer questions.

Let us look at a few examples to learn how to use a truth table. Assume `x = 3` and `y = -1`. Then both `x` and `y` are `true`, so we would use the last line of the table. To find the answer for `x && y`, use the fourth column. To find `x || y`, use the fifth column. As a second example, suppose `x=0` and `y=-1`; then `x` is `false` and `y` is `true`, so we use the second row. Therefore, `x || y` is `true` (1) and `x && y` is `false` (0).

4.6.2 Parse Trees for Logical Operators

The precedence and usage of the three logical operators are summarized in Figure 4.14. Note that `&&` has higher precedence than `||` and that both are quite low in the precedence table. If an expression combines arithmetic, comparisons, and logic, the arithmetic will be parsed first, the comparisons second, and the logic last. The practical effect of this precedence order is that we can omit many of the possible parentheses in expressions. Figure 4.15 gives an example of how to parse an expression with operators of all three kinds. Before beginning the parse, we note the precedence of each operator used. Beginning with the highest-precedence operator and proceeding downward, we then parenthesize or bracket each operator with its operands.

A small circle, called a **sequence point**, is written on the parse tree under every `&&` and `||` operator. The order of evaluation of the parts of a logical expression is different from other expressions, and the sequence points remind us of this difference. For these operators, the part of the tree to the left of the sequence point is always evaluated before the part on the right.[7] This fact is very important in practice because it permits us to use the left side of a logical expression to "guard" the right side, as explained in the next section.

4.6.3 Lazy Evaluation

Logical operators have a special property that makes them different from all other operators: You often can know the result of an operation without even looking at the second operand. Look again at the truth table in Figure 4.13, and note that the answer for `x && y` is always 0 when `x` is `false`. Similarly, the answer for `x || y` is always 1 when `x` is `true`. This leads to a special method of computation, called **lazy evaluation**, that can be used only for logical operators. Basically, the left operand of the logical operator is evaluated then

[7] Two other operators, the question mark and the comma, have sequence points associated with them. For all other operators in C, either the right side of the tree or its left side may be evaluated first.

Arity	Usage	Meaning	Precedence	Associativity
Unary	`!x`	logical-NOT `x` (logical opposite)	15	right to left
Binary	`x && y`	`x` logical-AND `y`	5	left to right
	`x || y`	`x` logical-OR `y`	4	"

Figure 4.14. Precedence of the logical operators.

tested. If that alone decides the value of the entire expression, the rest is skipped. We show this skipping on the parse tree by writing a diagonal *pruning mark* on the branch of the tree that is skipped. You can see these marks on the trees in Figures 4.17 and 4.19. To further emphasize the skipping, a loop is drawn around the skipped portion. Note that *no operator* within the loop is executed.

Logical-AND. Figure 4.16 summarizes how lazy evaluation works for logical-AND, and Figure 4.17 illustrates the most important use of lazy evaluation: guarding an expression. The left side of a logical-AND expression can be used to "guard" the right side. We use the left side to check for "safe" data conditions; if found, the left side is `true` and we execute the right side. If the left side is `false`, we have identified a data value that would cause trouble and use lazy evaluation to avoid executing the right side. In this way, we avoid computations that would cause a machine error or program malfunction. For example, in Figure 4.17, we want to test a condition with `x` in the denominator. But dividing by 0 is an error, so we should check the value of `x` before testing this condition. If it is 0, we skip the rest of the test; if it is nonzero, we go ahead.

Logical-OR. Figure 4.18 summarizes how lazy evaluation works for logical-OR, and Figure 4.19 illustrates how lazy evaluation can be used to improve program efficiency. Logical-OR often is used for data validation.

We parse the expression `y = a < 10 || a >= 2 * b && b != 1` using parentheses and a tree.

Parenthesizing in precedence order:

```
Level 13: y = a < 10 || a >= (2 * b) && b != 1
Level 10: y = (a < 10) || (a >= (2 * b)) && b != 1
Level 9:  y = (a < 10) || (a >= (2 * b)) && (b != 1)
Level 5:  y = (a < 10) || ((a >= (2 * b)) && (b != 1))
Level 4:  y = ((a < 10) || ((a >= (2 * b)) && (b != 1)))
Level 2:  (y = ((a < 10) || ((a >= (2 * b)) && (b != 1))))
```

Precedence level is listed above each operator.

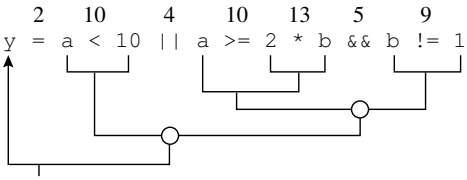


Figure 4.15. Parsing logical operators.

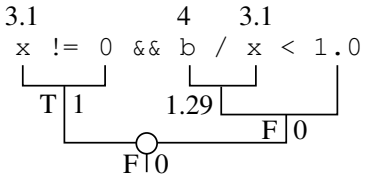
x	y	x && y
0	?	0 and skip second operand
T	0	0
T	T	1

- To evaluate an `&&` operator, first evaluate its *left* operand. This operand might be a simple variable or literal, or it might be a complicated expression.
- Look at the truth value. If it is `false`, return 0 as the answer to the `&&` operation and skip the next step.
- Otherwise, we do not yet know the outcome of the expression, so evaluate the right operand. If it is `false`, return 0. Otherwise, return 1.

Figure 4.16. Lazy truth table for logical–AND.

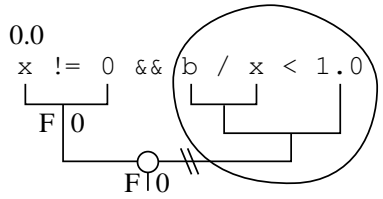
We evaluate a logical expression twice with different operand values.

(1) Evaluation with `x = 3.1` and `b = 4`. All parts of the expression are evaluated because the left operand is `true`.



The `/` and `<` operators are evaluated and their results are written under the operators on the tree.

(2) Evaluation with the values `x = 0.0` and `b =` anything. Skipping happened because the left operand is `false`.



The "pruning mark" on the tree and the looped line show the part of the expression that was skipped.

Figure 4.17. Lazy evaluation of logical–AND.

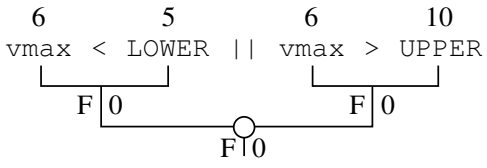
x	y	x || y
T	?	1 and skip second operand
0	0	0
0	T	1

- To evaluate an `||` operator, first evaluate its *left* operand. This operand might be a simple variable or literal or it might be a more complicated expression.
- Look at the truth value. If it is `true`, return 1 as the answer to the `||` operation and skip the next step.
- Otherwise, we do not yet know the outcome of the expression so evaluate the right operand. If it is `false`, return 0. Otherwise, return 1.

Figure 4.18. Lazy truth table for logical–OR.

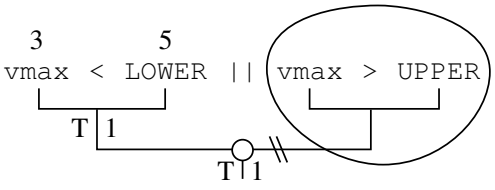
We evaluate a logical expression twice with different operand values. The first time, the entire expression is evaluated. The second time, a shortcut is taken.

(1) Evaluation with `vmax = 6`. All parts of the expression are evaluated because the left operand is `false`.



The `<` and `>` operators are both evaluated and their results are written under the operators on the tree.

(2) Evaluation with `vmax = 3`. The left side is `true`, which causes skipping.



The pruning mark on the tree and the looped line show the part of the expression that was skipped.

Figure 4.19. Lazy evaluation of logical-OR.

If one validity condition is fast to compute and another is slow, we can save a little computation time by putting the fast test on the left side of the logical-OR and the slow test on the right. Or we could put the most common problem on the left and an unusual problem on the right.

Figure 4.19 shows two evaluations of a logical-OR expression that is used in the next program. If the input data fails the first test, the second test determines the result of the expression. If the data passes the first test, we save time by skipping the second test and return the result of the left side.

4.7 Integer Operations

Modern computers have two separate sets of machine instructions that perform arithmetic: one for integers, the other for floating-point numbers. We say that the operators, `*`, `/`, `+`, and `-`, are *generic*, because they have more than one possible translation. When a C compiler translates a generic operator, it uses the types of the operands to select either an integer or a floating-point operation. The compiler will choose integer operations when both operands are integers; the result will also be an integer. In all other cases, a floating-point operation will be chosen.

4.7.1 Integer Division and Modulus

Division. Like the other arithmetic operators, the division operator `/` is generic; its meaning depends on the types of its operands. However, each of the operators `*`, `+`, and `-` symbolizes a single mathematical operation, even though it is performed in two different ways on the two kinds of number representations. In contrast, `/` represents two different mathematical operations: **real division** (where the answer has a fractional part) and **integer division** (where there are two parts of the answer, the quotient and the remainder). Here, the instruction used makes a significant difference. With floating-point operands, only real division is meaningful. However, with integer operands, both real and integer division are useful and a programmer might wish to use either one. In C, if both operands are integers, the integer quotient is calculated and the remainder is forgotten. A programmer who needs the remainder of the answer can use the modulus operator, `%`, which is described shortly. If the entire fractional answer is needed, one of the operands must first be converted to floating-point representation. (Data type conversion techniques are covered later in this chapter.)

Division by 0. Attempting to **divide by 0** will cause an error that the computer hardware can detect. In most cases, this error will cause the program to terminate.[8] It always is wise to check for this condition before doing the division operation in order to make the program as robust as possible; that is, it does something sensible even when given incorrect data.

Indeterminate answers (optional topic). A further complication of integer division is that the C standard allows two different correct answers for `x/y` in some cases. When x is not an even multiple of y and either is negative, the answer can be the integer either just larger than or just smaller than the true quotient (this choice is made by the compiler, not the programmer). This indeterminacy is provided by the standard to accommodate different kinds of hardware division instructions. The implication is that programs using division with signed integers may be nonportable because the answer produced depends on the hardware. This "feature" of the language is worse in theory than in practice; we tested C compilers running on a variety of hardware platforms and found that they all truncate the answer toward 0 (rather than negative infinity). However, the careful C programmer should be aware of the potential problem here.

Modulus. When integer division is performed, the answer has two parts: a quotient and a remainder. C has no provision for returning a two-part answer from one operation, so it provides two operators. The **integer modulus** operator, named *mod* and written `%`, performs the division and returns the remainder,

[8] More advanced techniques, beyond the scope of this book, can be used to take special action after the termination.

x	y	`x % y`
10	3	1
9	3	0
8	3	2
7	3	1
6	3	0
5	3	2

x	y	`x % y`
3	10	3
3	9	3
7	2873	7
7	7	0
7	1	0
7	0	Undefined

Figure 4.20. The modulus operation is cyclic.

while `/` returns the quotient. Figure 4.20 shows the results of using `%` for several positive values; Figure 4.21 is a visual presentation of how mod is computed. Note the following properties of this operator:

- Mod is defined for integers x and y in terms of integer division as: `x % y` $= x - y \times (x/y)$.
- If x is a multiple of y, the answer is 0.
- If x is smaller than y, the answer is x.
- The operation `x % y` is a cyclic function whose answer (for positive operands) always is between 0 and $y - 1$.
- This operator has no meaning for floating-point operands.
- If y is 0, `x % y` is undefined. At run time a division by 0 error will occur.
- The results of `/` with negative values is not fully defined by the standard; implementations may vary. For example, $-5/3$ can equal either -1 (the usual answer) or -2. Since the definition of `%` depends on the definition of `/`, the result of `x % y` is indeterminate if either x or y is negative. Therefore, `-5 % 3` can equal either -2 or 1.

A program that uses integer `/` and `%` is given in the next section, and one that uses the `%` operator to help format the output into columns is in Figure 5.26.

4.7.2 Applying Integer Division and Modulus

We normally count and express numbers in base 10, probably because we have 10 fingers. However, any number greater than 1 can be used as the base for a positional notation.[9] Computers use base 2 (binary) internally and the `C` language lets us write numbers in bases 8 (octal) and 16 (hexadecimal) as well as base 10 (decimal). These number representations and the simple algorithms for converting numbers from one base to another are described in Appendix E. The next program shows how one can use a computer to convert a number from its internal representation (binary) to any desired base. The algorithm used is based on the meaning of a positional notation:

- Each digit in a number represents a multiple of its place value.
- The place value of the rightmost position is 1.
- The place value of each other position is the base times the place value of the digit to its right.

Therefore, given a number N (in the computer's internal representation) and a base B, `N % B` is the digit whose place value is $B^0 = 1$ when expressed in positional notation using base B. We can use these facts to convert a number N to the equivalent value N' in base B.

The algorithm is a simple loop that generates the digits of N' from right to left. On the first pass through the conversion loop, we compute `N % B`, which is the rightmost digit of N'. Having done so, we are no longer

[9] Theoretically, negative bases can be used, too, but they are beyond the scope of this text.

To calculate `a % b`, we distribute `a` markers in `b` columns. The answer is the number of markers in the last, partially filled row. (If the last row is filled, the answer is 0.)

```
operation: 12 % 5       15 % 5       4 % 5       5 % 4     10 % 4
           ---------------------------------------------------------
           x x x x x    x x x x x    x x x x .   x x x x    x x x x
           x x x x x    x x x x x                x . . .    x x x x
           x x . . .    x x x x x                           x x . .
           ---------------------------------------------------------
answer:    1 2 3 4 0    1 2 3 4 0    1 2 3 4 0   1 2 3 0    1 2 3 0
             ↑                ↑          ↑         ↑          ↑
```

Figure 4.21. Visualizing the modulus operator.

interested in the 1's place, so we compute $N = N/B$ to eliminate it and prepare for the next iteration. We continue this pattern of using `%` to generate digits and integer division to reduce the size of N until nothing is left to convert. An example is given in Figure 4.22, a program that implements this algorithm is in Figure 4.22 and the flow diagram for this program in Figure 4.24. This program also illustrates the use of an increment operator, logical opertor, and assignment combination operator.

Notes on Figure 4.23. Number conversion.

First box: selecting a valid base and the number to convert.

- We have restricted the acceptable number bases to the range 2...10, which restricts the possible digits in the answer to the range 0...9. This algorithm could be used to convert a number to any base. We demonstrate it here only for bases of less than 10 because we wish to focus attention on the conversion algorithm, not on the representation of digits greater than 9.
- We use 2 as the minimum base value; 1 and 0 cannot be used as bases for a positional notation. The following output shows an example of error handling:

```
Read an integer and express in a given base.
Please enter a number to convert and
a target base between 2 and 10: 3 45
Base must be between 2 and 10.
```

- Any integer, positive, negative or zero, can be accepted as input for the number to be converted. However, 0 must be treated as a special case. The following output shows an example

Any number N can be expressed in positional notation as a series of digits:

$$N = \ldots D_3 D_2 D_1 D_0$$

If the number's base is B, then each digit is between 0 and $B-1$ and the value of the number is

$$N = \ldots B^3 \times D_3 + B^2 \times D_2 + B^1 \times D_1 + D_0$$

Now, if $N = 1234$ and $B = 10$, we can generate all the digits of N by repeatedly taking `N % B` and reducing N by a factor of B each time:

$$\begin{array}{llll} D_0 = & 1234 \,\%\, 10 = 4 & N_1 = & 1234/10 = 123 \\ D_1 = & 123 \,\%\, 10 = 3 & N_2 = & 123/10 = 12 \\ D_2 = & 12 \,\%\, 10 = 2 & N_3 = & 12/10 = 1 \\ D_3 = & 1 \,\%\, 10 = 1 & N_4 = & 1/10 = 0 \end{array}$$

Figure 4.22. Positional notation and base conversion.

```
Read an integer and express in a given base.
Please enter a number to convert and
a target base between 2 and 10: 0  3
0 is 0 in every base.
```

Second box (outer): converting to the selected base.

- We start by generating the coefficient of B^0, so we set `power = 0`. After each iteration, we increment `power` to prepare for converting the coefficients of B^1, B^2, and so forth.
- We reduce the size of N on each iteration by dividing it by the target base; we continue until N is reduced to 0.
- Note that a loop test need not always involve the loop counter; it is necessary only that the value being tested change somewhere within the loop.
- We first print the number itself (in base 10) and then an `=` sign before the first term of the answer. We indent and print a `+` before all other terms.

Two examples of the output.

```
Read an integer and express in a given base.
Please enter a number to convert and
a target base between 2 and 10: 45 3
```

Convert an integer to a selected base and print it using place-value notation.

```
#include <stdio.h>

int main( void )
{
    int n;          // input:  the number to convert
    int base;       // input:  base to which we will convert n
    int rhdigit;    // right-hand digit of n-prime
    int power;      // loop counter

    printf( " Read an integer and express it in a given base.\n"
            " Please enter a number to convert and\n"
            " a target base between 2 and 10:  " );
    scanf( "%i %i", &n, &base );

    if (base < 2 || base > 10) printf( " Base must be between 2 and 10\n" );
    else if (n==0) printf ( " 0 is 0 in every base.\n" );

    power = 0;
    // --- Generate and print digits of converted number, right to left.
    while (n != 0) {
        if (power == 0) printf( "%12li = ", n );
        else printf( "             +  ");

        rhdigit = n % base;      // Isolate right-hand digit of n.
        n /= base;               // then eliminate right-hand digit.

        printf( "%hi * %hi^%i \n", rhdigit, base, power );
        ++power;
    }

    return 0;
}
```

Figure 4.23. Number conversion.

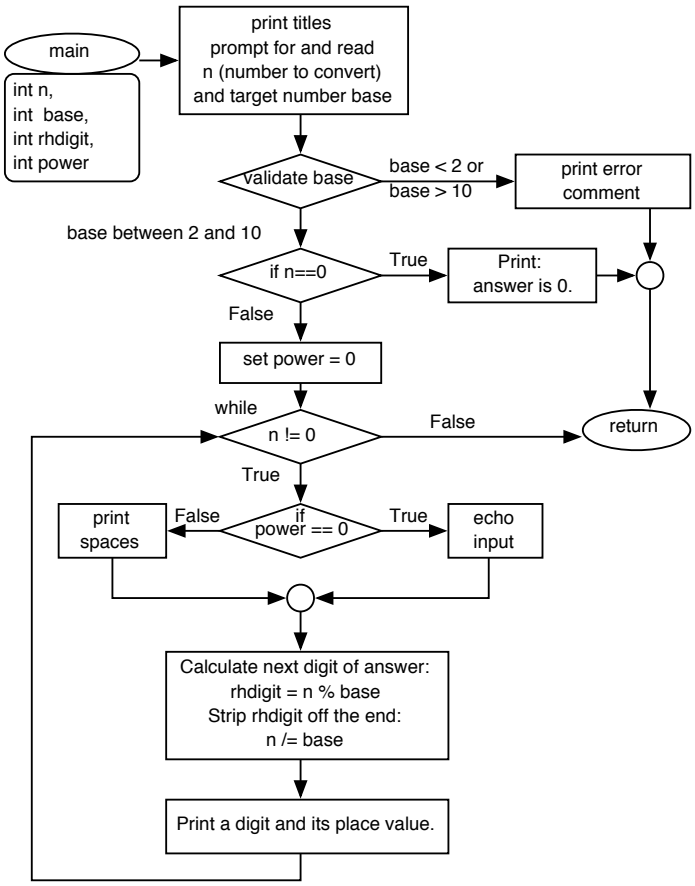


Figure 4.24. A flow diagram for the base conversion.

```
          45 =   0 * 3^0
             +   0 * 3^1
             +   2 * 3^2
             +   1 * 3^3

Read an integer and express in a given base.
Please enter a number to convert and
a target base between 2 and 10: -45 10

         -45 =  -5 * 10^0
             +  -4 * 10^1
```

Inner box: decomposing the number, digit by digit. As explained, we use `%` to generate each digit of N'. Then we use integer division to reduce N by removing the extracted digit and shifting the others one position to the right. This prepares N for the next iteration. Real division would not work here; the algorithm relies on the fact that the remainder is discarded.

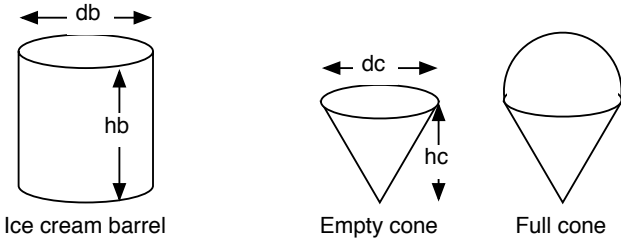
4.8 Techniques for Debugging

4.8.1 Using Assignments and Printouts to Debug

One cause for logic errors is complexity, and that complexity also makes the errors difficult to find and difficult to correct. Sometimes an algorithm is complex; that can be addressed by breaking it into modular parts, as shown in Chapter 5. Here, we address the problem of debugging complex formulas.

Problem scope: Determine how many barrels of ice cream to buy to fill one cone for each guest at the annual picnic. The cone portion should be filled, and a half-sphere of ice cream should be on top.

Inputs: Diameter and height of cones. Diameter and height of a 3-gallon barrel of ice cream. (All in cm.)



Constants: Number of guests (number of cones needed).
Formulas:

$$\begin{aligned}
\text{volume of ice cream needed} &= \text{number of guests} * \text{volume of one full cone} \\
\text{barrels needed} &= \text{volume of ice cream needed/volume of barrel} \\
\text{volume of full cone} &= \text{volume of hemisphere} + \text{volume of empty cone} \\
\text{volume of barrel} &= \pi * \text{hb} * \text{db}^2/4 \\
\text{volume of cone} &= \pi * (\text{dc}/2)^2 * \text{hc}/3 \\
\text{volume of hemisphere} &= 2 * \pi * (\text{dc}/2)^3/3
\end{aligned}$$

Output required: Number of 3-gallon barrels of ice cream to buy.

Debugging outputs: Volume of one barrel, volume of an empty cone, volume of the half-sphere of ice cream on top of the cone, and total volume of the full cone.

Figure 4.25. Problem specification: Ice cream for the picnic.

Suppose we wish to write a program for the specification in Figure 4.25. We could combine all the formulas given into one line:

$$\text{barrels needed} = \text{Number of guests} * \pi * (dc^3/12 + (dc/2)^2 * hc/3)/(hb * db^2/4)$$

However, the resulting formula is quite complex and difficult to compute by hand or in one's head. It would make much more sense to calculate each formula, as given. Even better, you might notice that the expression `dc/2` is used twice, and compute it separately also. The program in Figure 4.26 uses a sequence of assignment statements to calculate the parts of the long. messy formula.

Notes on Figure 4.26. How much ice cream?

First box.
In addition to the four variables that we need for input and the one for output, we define four variables for intermediate results.

Second box.
An important aid in debugging is to be certain that the inputs that were actually read by the program are the ones that the user intended to enter. Echoing the inputs as part of the output is sound programming practice.

Third box.

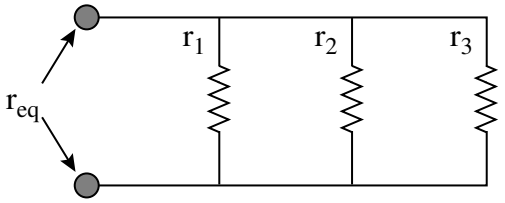
- We implement the separate formulas instead of the combined formula to make both programming and debugging easier and less error prone. It is a little more work for the computer this way, but in almost all circumstances, simplicity is better than complexity.
- We compute each formula and store its result for later use. We also print each result, so that the user can hand-verify each part of the computation.
- These formulas are not computed in the same order as they are given in the specification because, in C, each part must be computed before it is used. The specification starts with the general formula and goes on to the details. We must compute the details before we can compute the general formula.
- We want to print the volume of the full cone, but do not need it in any further computation. Moreover, the formula is simple (it involves only one operation). So we choose to write the formula as part of the `printf()` statement, rather than as a separate assignment statement. This technique is good style as long as the results are not very complex.
- Sample output:

```
#include <stdio.h>
#define  GUESTS   100
#define  PI   3.1415927

int main( void )
{
    double h_cone, d_cone;      // Height and diameter of part
    double h_barrel, d_barrel;  // Height and diameter of barrel
    double n_barrels;           // Number of barrels needed.
    double r_cone, v_cone;     // Cone's radius and volume
    double v_barrel, v_hemi;   // Volume of barrel and ice cream on top.

    printf( " How much ice cream do we need?\n"
            " Enter Diameter and height of ice cream barrel: " );
    scanf( "%lg %lg", & d_barrel, & h_barrel);
    printf( " Enter Diameter and height of an empty cone: " );
    scanf( "%lg %lg", & d_cone, & h_cone);

    printf( "     Barrel is %g cm. wide, %g cm. tall.\n", d_barrel, h_barrel );
    printf( "     Cones are %g cm. wide, %g cm. tall.\n", d_cone, h_cone );

    v_barrel = PI * h_barrel * d_barrel * d_barrel / 4;
    printf( "     Barrel volume = %g\n", v_barrel );
    r_cone = d_cone / 2;
    v_cone = PI * r_cone * r_cone * h_cone / 3;
    printf( "     Cone volume = %g\n", v_cone );
    v_hemi = 2 * PI * r_cone * r_cone * r_cone / 3;
    printf( "     Top volume = %g\n", v_hemi );
    printf( "     Full cone volume = %g cm^3\n", v_hemi + v_cone );

    n_barrels = GUESTS * (v_hemi + v_cone ) / v_barrel ;
    printf( " You need %g barrels of ice cream.\n", n_barrels );

    return 0;
}
```

Figure 4.26. How much ice cream?

Problem scope: Find the electrical resistance equivalent, r_{eq}, for three resistances wired in parallel.
Input: Three resistance values, r_1, r_2, and r_3.
Formula:

$$r\_eq = \frac{r_1 * r_2 * r_3}{r_1*r_2 + r_1*r_3 + r_2*r_3}$$



Constants: None.
Output required: The three inputs and their equivalent resistance.
Computational requirements: The equivalent resistance should be a real number, not an integer.

Figure 4.27. Problem specification: Computing resistance.

```
How many gallons of ice cream do we need?
Enter Diameter and height of ice cream barrel: 30 40
Enter Diameter and height of an empty cone: 10 15
    Barrel is 30 cm. wide, 40 cm. tall.
    Cones are 10 cm. wide, 15 cm. tall.
    Barrel volume = 28274.3 cm^3
    Cone volume = 392.699 cm^3
    Top volume = 261.799 cm^3
    Full cone volume = 654.498 cm^3
You need 2.31481 barrels of ice cream.
```

- If you have an on-line debugger and know how to use it, you can set a breakpoint after each computation instead of printing the result. Both ways will give you the information you need. However, if you need to ask someone who is not present for assistance, it is essential to provide that person with printouts of both the program and the results.

Fourth box.

- Finally we are able to implement one form of the general formula. It is too long and complex to fit easily into a `printf()` statement, so we compute the answer and store it in a variable first, then print it.
- When we wrote this program, we were uncertain whether the answers were right or wrong, So we used the intermediate printouts with a calculator to verify that each part was correct and made sense. Then we verified that the final result was correctly computed from the intermediate results.

4.8.2 Case Study: Using a Parse Tree to Debug

When a program seems to work but gives the wrong answers, the problem sometimes lies in the expressions that calculate those answers. Drawing a parse tree can help **debug** the program; that is, help a programmer find the error. We illustrate this technique through a case study for which Figure 4.27 gives the full specification. In this problem, a circuit is wired with three resistances connected in parallel, as shown in the specification. We must calculate the electrical resistance equivalent, r_{eq}, for this part of the circuit.

Step 1. Making a test plan. Making a **test plan** first is a good way to understand the problem. It forces us to analyze the formulas, look at the details, and think about what kind of data might cause problems. We look at the problem specification to decide what the test plan should be.

The first test case should be something that can be computed in one's head. We note that the arithmetic is very simple if all the resistances are 2 ohms, so we enter this case in the test chart, which follows. We want

to test inputs with fractional values and note that we can easily compute the answer for three resistances of 0.1 ohm each. Then we notice that, if two inputs are 0, the denominator of the fraction will be 0. Since division by 0 is undefined, this will cause trouble. Since no limitations on input values are specified, it is unclear what to do about this. Resistances that are 0 or negative make no realistic sense, so we decide to warn the user and trust him or her to enter valid data. Next, we enter an arbitrary set of values, just to see what the output will look like for the typical case. We use a pocket calculator and a pencil to do the computation by hand. We now have three tests in our test plan, which is enough for a very simple program.

r_1	r_2	r_3	r_{eq}
2	2	2	$8.0/12.0 = 0.666667$
0.1	0.1	0.1	$0.001/0.03 = 0.033333$
75	40	2.5	2.28137

Step 2. Starting the program. Write the parts that remain the same from application to application. We write the `#include` statement and the first and last lines of `main()` with the opening and closing messages. The dots (...) represent the unfinished parts of the program.

```
#include <stdio.h>
...                // Space for #defines.
int main( void )
{   ...            // Space for declarations.
    puts( "\n Computing Equivalent Resistance \n" );
    ...            // Space for I/O and computations.
    puts( "\n Normal termination." );
}
```

Step 3. Reading the input. We need to read three resistance values; we could do this with three calls on `scanf()` or with one. We choose to use one call because one input step is faster for the user and we do not think the user will be confused by giving three answers for one prompt in this situation. To store the three values, we need three variables, which we name `r1, r2`, and `r3`. We declare these as type `double` (not `int`) because the answer will have a fractional part. We write a declaration for three doubles:

```
double r1, r2, r3; // input variables for resistances
```

Now we are ready for the prompt and the input. In the format for `scanf()`, we write three percent signs because we will be reading three values. Since we are reading `double` values, we write `lg` (the letter l, not the numeral 1) after the percent signs. We remember to write the ampersand before the name of each variable that needs to receive an input value.

```
printf( "\n Enter resistances #1, #2, and #3 (ohms).\n"
        " All resistances must be greater than 0: " );
scanf( "%lg%lg%lg", &r1, &r2, &r3 );
```

Finally, we write a `printf()` statement to echo the three input values. In the output format, we again write three percent signs, for three values. However, the correct output code for type `double` is `g`, without the `l`. At the end of the format we remember to write a newline character. After the format we list the names of the variables to print, without ampersands. (Reading into a variable requires an ampersand; writing does not.)

```
printf( "\n    r1= %g    r2= %g    r3= %g\n", r1, r2, r3 );
```

```
// ------------------------------------------------------------------
// Compute the equivalent resistance of three resistors in parallel
//
#include <stdio.h>

int main( void )
{
    double r1, r2, r3;  // input variables for three resistances
    double r_eq;        // equivalent resistance

    puts( "\n Computing Equivalent Resistance\n" );
    printf( "\n Enter values of resistances #1, #2, and #3 (ohms).\n"
            " All resistances must be greater than 0: " );
    scanf( "%lg%lg%lg", &r1, &r2, &r3 );
    printf( "     r1= %g    r2= %g    r3= %g\n", r1, r2, r3 );

    r_eq = r1 * r2 * r3 / r1 * r2 + r1 * r3 + r2 * r3;
    printf( " The equivalent resistance is %g\n\n", r_eq );

    return 0;
}
```

Figure 4.28. Computing resistance.

Step 4. Computation and output. Now we transcribe the mathematical formula into C notation, changing the fraction bar to a division sign and writing the subscripts as part of the variable names. In the process, we note that we need a variable for the result and declare another `double`. Then we write a `printf()` statement to print the result. We have

```
double r_eq;             // equivalent resistance
...
r_eq = r1 * r2 * r3 / r1 * r2 + r1 * r3 + r2 * r3;
printf( " The equivalent resistance is %g\n", r_eq );
```

Step 5. Putting it together and testing it. We now type in all the parts of the program. The code compiled successfully after correcting a few typographical errors; the result is shown in Figure 4.28. We then ran the program and entered the first data set. The output was

```
Computing Equivalent Resistance

Enter values of resistances #1, #2, and #3 (ohms).
All resistances must be greater than 0: 2 2 2
    r1= 2    r2= 2    r3= 2
The equivalent resistance is 16
```

Comparing the answer to the answer in our test plan, we see that it is wrong. The correct answer is 0.667. What could account for the error? There are three possibilities:

1. The input was read incorrectly. This could happen if the format were inappropriate for the data type of the variable or if we forgot to write an ampersand in front of the variable name.
2. The answer was printed incorrectly. This could happen if the format were inappropriate for the data type of the variable or if we wrote the wrong variable name or put an ampersand in front of it.
3. The answer was computed incorrectly.

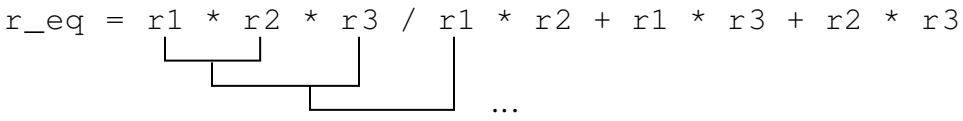


Figure 4.29. Finding an expression error.

We eliminate the first possibility immediately; we echoed the data and know it was read correctly. This is why every program should echo its inputs. Then we look carefully at the final `printf()` statement and see no errors. We think this is not the problem. The remaining possibility is that the computation is wrong, so we need to analyze the formula we wrote.

Step 6. Correcting the error. The best way to analyze a computation is with a parse tree, so we copy the expression on a piece of paper and begin drawing the tree (see Figure 4.29).

1. The `*` and `/` are the highest precedence operators in the expression so they are parsed first using left-to-right associativity, as shown in Figure 4.29.
2. At the `/` sign, we see that the right operand is `r1`, which does not correspond to the mathematical formula. The error becomes clear; the denominator for `/` should be the entire subexpression `r1 * r2 + r1 * r3 + r2 * r3`, not just `r1`.
3. The error is corrected by adding parentheses as shown in Figure 4.30, so that it now corresponds to the mathematical formula.

We correct the program, recompile it, and retest it. The results are

```
Computing Equivalent Resistance
Enter values of resistances #1, #2, and #3 (ohms).
All resistances must be greater than 0: 2 2 2
    r1= 2    r2= 2    r3= 2
The equivalent resistance is 0.667

Normal termination.
```

We see that the first answer is now correct. Before going on, we also consider the appearance of the output. Is it neat and readable? Does every number have a label? Should the vertical or horizontal spacing be adjusted? We decide to add a blank line before the final answer, so we insert a `\n` at the beginning of the format. The boxed section of the program now is

```
r_eq = r1 * r2 * r3 / (r1 * r2 + r1 * r3 + r2 * r3);
printf( "\n The equivalent resistance is %g\n", r_eq );
```

We recompile the program and go on with the test plan. The program produces correct results for the other two test cases:

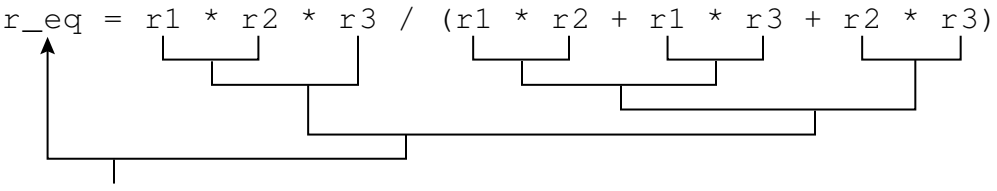


Figure 4.30. Parsing the corrected expression.

```
Enter values of resistances #1, #2, and #3 (ohms).
All resistances must be greater than 0: .1 .1 .1
    r1= 0.1    r2= 0.1    r3= 0.1

The equivalent resistance is 0.033
-------------------------------------------------

Enter values of resistances #1, #2, and #3 (ohms)
All resistances must be greater than 0: 75 40 2.5
    r1= 75    r2= 40    r3= 2.5

The equivalent resistance is 2.281
```

The output is correct, neat, and readable, so we declare the program finished.

4.9 What You Should Remember

4.9.1 Major Concepts

This chapter is concerned with how to create and name objects, how to use types to describe their properties, and how to combine the objects with operators to form expressions. These concepts are summarized here.

- Operators:
  - Operators are like verbs: They represent actions and can be applied to objects of appropriate types. An expression is like a sentence: It combines operators with the names of objects to specify a computation.
  - Precedence, associativity, and parentheses control the structure of an expression. The precedence of C operators follows normal mathematical conventions.
  - A few operators have side effects; that is, they modify the value of some variable in memory. These include the assignment operator, assignment combinations, increment, and decrement.
  - Integer division is not the same as division using real numbers; any remainder from an integer division is forgotten. The remainder, if needed, must be computed by using the modulus operator (`%`).
  - The result of a comparison is always either `true` (1) or `false` (0).
  - Every data value can also be interpreted as either `true` or `false`: zero is `false`, and every other value is `true`. Sometimes it surprises beginners to learn that a negative number will be interpreted as `true` when it is the operand of a logical operator, an `if` statement or a `while` statement.
- Diagrams. Diagrams are used to visualize the parts of a program and how they interact. They become increasingly important as the more complex features of C are introduced. We have now introduced three ways to visualize the aspects of a program:

  1. A flow diagram (introduced in Chapter 3) is used to depict the structure of an entire program and clarify the sequence of execution of its statements.
  2. A parse tree is used to show the structure of an expression and can be used to manually evaluate the expression.
  3. An object diagram is used to visualize a variable. Simple object diagrams were introduced in this chapter; more elaborate diagrams will be introduced as new data types are presented.

- Debugging. To debug a program, you must find and correct all the syntactic, semantic, and logical errors. The best practice is to design and write your code so that this becomes easy:

  1. Strive for simplicity at all times in every way.
  2. Do a thorough specification first.
  3. Keep your formulas short.

4. Echo the inputs.
5. Print out intermediate results or use an online debugger.
6. Hand-check the results.
7. Use a parse tree if you cannot find an error in a formula.

4.9.2 Programming Style

- Names: You must name every object and function you create. The compiler does not care what names you use as long as they are consistent. However, people do care. Obscure names, silly names, and unpronounceable names hinder comprehension. A program with bad names takes longer to debug.
- The length of a name: Extremely long and short names are poor choices. Except in unusual circumstances, a one- or two-letter name does not convey enough information to clarify its meaning. At the other extreme, very long, wordy names are distracting and often obscure the structure of an expression.
- Long expressions: Very long, complex expressions are difficult to write correctly and difficult to debug. When a formula is long and complex or has repeated subexpressions, it is a good idea to break it into several separate assignment statements.
- Parentheses: Use parentheses to clarify the structure of your expressions by enclosing meaningful subexpressions. Use them when you are uncertain about the precedence of operators. However, use parentheses sparingly; too many can be worse than too few. When three and four parentheses pile up in one part of an expression, they can be hard to "pair up" visually. In this situation, moderation is the key to good style.
- Increment and decrement operators: These operators can give nonintuitive results because of C's complicated rules about the order in which parts of an expression are evaluated. Until you fully understand the evaluation rules, restrict your use to very short expressions and avoid combining these operators with logical `&&` and `||`.
- When using division or modulus, be sure that there is no possibility that the divisor is 0. Dividing by 0 causes an immediate program crash in many systems and produces incorrect results on others. If a 0 divisor is possible, test for it.

4.9.3 New and Revisited Vocabulary

The most important terms and concepts discussed in this chapter:

garbage	parse tree	truth value
precedence	test plan	truth table
associativity	debugging	lazy evaluation
arity	arithmetic operators	increment operators
precedence table	assignment operator	postfix operator
operator	combination operators	prefix operator
binary operator	side effect	integer arithmetic
operand	relational operators	modulus
expression	logical operators	intermediate printouts

C ***keywords and operators introduced or discussed in this chapter:***

`sizeof`	`=, +=, -=, *=, /=`	`++x` (preincrement)
`(...)`	`<, <=, >, >=`	`x++` (postincrement)
`+, -, *, /`	`==, !=`	`--x` (predecrement)
integer `/, %`	`&&, ||, !`	`x--` (postdecrement)

Group	Operators	Complication
Arithmetic	/	Division by 0 or 0.0 is undefined.
	/	Integer division is used if both operands are integers; the result is an integer. The fractional part is discarded.
	%	Not defined for floating-point values. For integers, the result is the remainder of an integer division.
	/, %	If both arguments are integers and one is negative, the result may be indeterminate.
Assignment combinations	+=, etc.	These operators use the value of a memory variable, then change it. Although C permits more than one of these operators to be used in a single expression, you should limit your own expressions to one.
Prefix increment and decrement	++, --	If you use a side-effect operator, do not use the same variable again in the same expression.
Postfix increment and decrement	++, --	Remember that these operators return one value for further use in expression evaluation and leave a different value in memory.
Comparison	==	Remember not to use =.
Logical	&&, ||, !	Remember that all negative and positive integers are considered `true` values. The only `false` value is 0.
Logical	&&, ||	There are special sequencing and lazy evaluation rules for expressions that contain these operators.

Figure 4.31. Difficult aspects of C operators.

4.9.4 Sticky Points and Common Errors

This has been a long chapter, filled with many facts about C semantics and C operators. The table in Figure 4.31 gives a brief review of the difficult aspects of C operators to assist you in program planning and debugging.

4.9.5 Where to Find More Information

- Program 4.A on the Applied C website shows a typical use of preincrement in a counting loop.
- The C operators for types `int` and `double` were described in this chapter. Operations on characters will be explained in Chapter 8 and operations on bits are in Chapter 15.
- Operations for nonsimple types (compound objects with more than one part) will be discussed in Chapters 10, 11, 12, and 13.
- Type conversions (casts and coercions) are explained in Chapter 7.
- Two operators, the question mark and the comma, are not needed in simple programs and are explained in Appendix **??**. Both are sequencing operators, that is, they have associated sequence points that force left-to-right evaluation.
- More information about evaluation order is given in Appendix **??**

- When post-increment or post-decrement is used to modify a variable, the time at which the variable is actually changed may vary from compiler to compiler. The C standard permits this variation, within limits, to enable code optimization. The exact rules for when the side effect must and may happen are complex. To explain them, one must first define sequence points and how they are used during evaluation. Consult a standard reference manual for a full explanation.

4.10 Exercises

4.10.1 Self-Test Exercises

1. Look at the parse tree in Figure 4.4. Make a list that shows each operator (one per line) with the left and right operands of that operator.
2. Write a single C expression to compute each of the following formulas:

   (a) Metric unit conversion: Liters = ounces / 33.81474
   (b) Circle: Circumference = $2\pi r$
   (c) Right triangle: Area = $\frac{bh}{2}$

3. Each of the following items gives two expressions that are alike except that one has parentheses and the other does not. You must determine whether the parentheses are optional. For each pair, draw the two parse trees and compare them. If the parse trees are the same, the two expressions mean the same thing and the parentheses are optional.

   (a) `d = a - c + b ;  d = a - (c + b) ;`
   (b) `e = g * f + h ;  e = g * (f + h) ;`
   (c) `d = a + b * c ;  d = a + (b * c) ;`
   (d) `e = f - g - h ;  e = (f - g) - h ;`
   (e) `d = a < b && b < c;  d = a < (b && b) < c;`

4. Using the following data values, evaluate each expression and say what will be stored in `d` or `e`:

```
int d, a = 5, b = 4, c = 32;
double e, f = 2.0, g = 27.0, h = 2.5;
```

   (a) `d = a + c - b ;`
   (b) `d = a + c * b ;`
   (c) `d = a * c - b ;`
   (d) `e = g * 3.0 * (- f * h) ;`
   (e) `d = a <= b ;`
   (f) `e = f - (g - h) ;`
   (g) `e = g / f ;`
   (h) `e = 1.0; e += h ;`
   (i) `d = (a < c) && (b == c) ;`
   (j) `d = ++a * b-- ;`

5. Using the given data values, parse and evaluate each of the following expressions and say what will be stored in `k`. Start with the original values for `k` and `m` each time. Check your precedence table to get the ordering right.

```
double k = 10.0;
double m = 5.0;
```

(a) `k *= 3.5;`
(b) `k /= m + 1;`
(c) `k += 1 / m;`
(d) `k -= ++m;`

6. In the following program, circle each error and show how to correct it:

```
#include "stdio"
#define PI 3.14159;
int main (void)
{   double v;

    printf( "Self-test Exercise/n" );
    printf( "If I don't get going I'll be late!!";
    puts( "Enter a number:  " );
    scanf( %g, v );
    w = v * Pi;
    printf( "w = %g \n", w );
}
```

7. Draw complete parse trees for the following expressions:

(a) `t = x >= y && y >= z ;`
(b) `x = (y + z) || v == 3 && !(z == y / v) ;`

8. What will be stored in `k` by the following sets of assignments? Use these variables: `int h, k, m;` .

(a) `h=2;  m=3;  k = h / m ;`
(b) `h=5;  m=16;  k = h % m;`
(c) `h=10;  m=3;  k = h / m + h % m;`
(d) `h=17;  m=5;  k = h / m;`

4.10.2 Using Pencil and Paper

1. Draw parse trees for the following expressions. Use the trees to evaluate the expressions, given the initial values shown. Assume all variables are type `double`.

(a) `a = 5; b = 4; c = 32;  d = a + c / b ;`
(b) `w = 3; x = 30; y = 5;  z = y + x / (- w * y) ;`
(c) `f = 3; g = 30; h = 5;  d = f - g - h ;`
(d) `f = 3; g = 27; h = 2;  d = f - (g - h) ;`

2. Explain why you need to know the precedence of the C operators to find the answer to question 1a. Explain why you need to know more than precedence to find the answer to question 1c. What else do you need to know?
3. Look at the parse tree in Figure 4.29. Make a list that shows each operator (one per line) with the left and right operands of that operator.
4. Parse and evaluate each of the following expressions and say whether the result of the expression is `true` or `false`. Use these variables and initial values: `int h = 0,  j = 7,  k = 1,  n = -3;`.

(a) `k && n`
(b) `!k && j`
(c) `k || j`
(d) `k || !n`
(e) `j > h && j < k`
(f) `j > h || j < k`

(g) `j > 0 && j < h || j > k`
(h) `j < h || h < k && j < k`

5. Write a single C expression to compute each of the following formulas:

   (a) Circle: Diameter $= 2r$
   (b) Flat donut: Area $= \pi \times (\text{outer\_radius}^2 - \text{inner\_radius}^2)$
   (c) Metric unit conversion: cm $= (\text{feet} \times 12 + \text{inches}) \times 2.54$

6. Parse and evaluate each of the following expressions and say what will be stored in `k` and in `m`. Start with the original value for `k` each time: `int m, k = 10;` .

   (a) `m = ++k;`
   (b) `m = k++;`
   (c) `m = -- k / 2;`
   (d) `m = 3 * k --;`

7. (Advanced) Draw parse trees for the following logical expressions and show the sequence points. Use the trees to evaluate the expressions, given the initial values shown. For each one, mark any part of the expression that is skipped because of lazy evaluation.

   (a) `w = 1; x = 5; y = 1;` `y && w != y && x`
   (b) `w = 1; x = 5; y = 3;` `w <= x && x <= y`
   (c) `x = 3; y = 0; z = 0;` `z != 0 || y && !x`
   (d) `r = 5; w = 0; x = 5; y = 0;` `y || r || x && !w`

8. What will be stored in `k` by the following sets of assignments? All variables are integers.

   (a) `h=4;` `m=5;` `k = h % m;`
   (b) `h=14;` `m=7;` `k = h % m;`
   (c) `h=7;` `m=15;` `k = h / m;`
   (d) `h=7;` `m=-5;` `k = h / m;`
   (e) `h=11;` `m=5;` `k = h / m + h % m;`

9. (Advanced) Trace the execution of the following loop and show the actual output:

```
int num = 10;
while ( num > 5 ) {
    if ( num % 3 == 0 )  num -= num / 3;
    else if ( num % 3 == 1 )  num += 2;
    else if ( num % 3 == 2 )  num /= 3;
    else  num--;
    printf( "num = %i\n", num );
}
```

4.10.3 Using the Computer

1. Your own size.

   Write a short program in which you use the `sizeof` operator to find the number of bytes used by your C compiler to store values of types `int`, `char`, and `double`.

2. Miles per gallon.

   Write a program to compute the gas consumption (miles/gallon) for your car if you are given, as input, `miles`, the number of miles since the last fill-up, and `gals`, the number of gallons of gas you just bought. Start with a formal specification for the program, including a test plan. What is the appropriate type for `miles`? For `gals`? For the answer? Explain why.

3. Centimeters.

   Write a program to convert a measurement in centimeters to inches. The numbers of centimeters should be read as input. Define int variable for centimeters, inches, and feet. Convert the centimeters to inches, then convert the inches to feet and inches (use the % operator). Print the distance in all three units. There are 2.54 centimeters in each inch and 12 inches in each foot.

   Start with a formal specification for the program, including a test plan. Turn in the source code and the output of your program when run using the data from your test plan.

4. A buggy mess.

   In the following program, circle each error and show how to correct it. There are errors on nearly every line, totaling at least 15 syntax errors, 4 syntax warnings, 1 linking error, and 3 serious logic errors. When you have found as many errors as you can, download the code from the text website, correct the errors, and try to compile the program. You have successfully debugged this code when you can get the code to compile, run, give you three different multiplication problems, and correctly tell you whether the answers are right or wrong.

```
#include stdio.h
#define SECRET = 17;

int main( void )
{
    integer number wanted;      // The number of problems you want
    integer answer;             // Your answer to the problem

    printf( " Doing the Exercises \n );
            " How many exercises do you want to do: " );
    scanf( "%i", number wanted );
    while ( number wanted > 0 );
    {
        printf( " What is %i * %i? ", SECRET, number wanted );
        scanf( "%i", answer );
        if ( answer = SECRET * number wanted) puts( "Great work." );
        else puts( "You need a calculator!"}
    }
    print( " Thank you for playing today.\n" );
    return;
}
```

5. Turf.

   You are a building contractor. As part of a project, you must install artificial turf on some sports fields and the adjacent areas. The owner has supplied length and width measurements of the field in yards and inches. Your supplier sells turf in 1-meter-wide strips that are 4 meters long. Write a program that will prompt the user for a pair of measurements in yards and inches (use integers). Convert each to meters and print the answer. (There are 39.37008 inches in a meter.) Calculate the number of strips of turf needed to cover the field. Round upward if a partial strip is needed.

   Start with a formal specification for the program, including a diagram and a test plan. Turn in the source code and the output of your program when run using the data from your test plan.

6. Holy, holy, holy day.

   A professor will assign homework today and does not want it due on anybody's holy day. The professor enters today's day of the week (0 for Sunday, 1 for Monday, etc.) and the number of days, D, to allow the students to do the work, which may be several weeks. Using the % operator, calculate the day of the week on which the work would be due. If that day is someone's holy day—Friday (Moslems), Saturday (Jews), or Sunday (Christians)—add enough days to D to reach the following Monday. Print the corrected value of D and the day of the week the work is due.

7. Ascending order.

   Write a program to input four integers and print them out in ascending numeric order. Use logical operators when you test the relationships among the numbers.

8. A piece of cake.

   Write a complete specification for a program to calculate the total volume of batter needed to half fill two layer-cake pans. The diameter of the pans is N and they are 2 inches deep. Read N as an input. Write a test plan for this program, then write the program and test it. The formula for the volume of a pan is

$$\text{Volume} = \pi \times \frac{\text{diameter}^2}{4.0} \times \text{height}$$

9. Circles.

   Write a problem specification and a complete test plan for a program that calculates facts about circles. Prompt the user to enter the diameter of the circle. If the input is less than 0.0, print an error comment. Otherwise, calculate and print the radius, circumference, and area of the circle. Make your output attractive and easy to read, and check it using your test plan.

10. Slope.

    The slope of a line in a two-dimensional plane is a measure of how steeply the line goes up or down. This can be calculated from any two points on the line, say $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ such that $x_1 < x_2$, as follows:

$$\text{Slope} = \frac{y_2 - y_1}{x_2 - x_1}$$

    Write a specification and test plan for this problem. Then write a program that will input two coordinates for each of two points, validate the second x coordinate, and print out the slope of the line.

11. What's the difference?

    Each term of an arithmetic series is a constant amount greater than the term before it. Suppose the first term in a series is a and the difference between two adjacent terms is d. Then the kth term is $a + (k-1) \times d$. The sum of the first k terms is

$$\text{Sum} = \frac{k}{2} \times (2a + d \times (k-1))$$

    Write a program that prompts the user for the first two terms of a series and the desired number of terms, k, to calculate. From these, calculate d and the sum of the first k terms. Display a, d, k, and the sum.

12. Summing squares.

    The sum of the squares of the first k positive integers is

$$1 + 4 + 9 + \ldots + k^2 = \frac{k \times (k+1) \times (2k+1)}{6}$$

    Write a program that prompts the user for k and prints out the sum of the first k squares. Make sure to validate the value of k that is entered.

13. Greatest common divisor.
    Some applications call for performing arithmetic on rational numbers (fractions). To do rational addition or subtraction, one must first convert the two operands to have a common denominator. When doing multiplication or division with fractions, it is important to reduce the result to lowest terms. For both processes, we must compute the greatest common divisor (GCD) of two integers. A good algorithm for finding the GCD was developed by Euclid 2300 years ago. In Euclid's method, you start with the two numbers, X and Y, for which you want the GCD. It does not matter which number is greater. Set $x = X$ and $y = Y$, then perform the following iterative algorithm:

    (a) Let `r = x % y`.
    (b) Now set `x=y` and `y=r`.
    (c) Repeat steps (a) and (b) until `y == 0`.

(d) At that time, `x` is the GCD of X and Y.

Write a program that will input two numbers from the user and calculate and print their greatest common divisor.

Chapter 5

Using Functions and Libraries

In this chapter, we introduce the most important tool C provides for writing manageable, debuggable programs: the function. In modern programming practice, programs are written as a collection of functions connected through well-defined interfaces. We show how to use standard library functions, functions from a personal library, and the programmer's own (programmer-defined) functions.

Functions are important because they provide a way to modularize code so that a large complex program can be written by combining many smaller parts. A **function** is a named block of code that performs a specified task when called. Many functions require one or more arguments. Each **argument** is an object or piece of information that the function can use while carrying out its specified task. Four functions were introduced in Chapter 3: `puts()`, `printf()`, `scanf()`, and `main()`. The first two perform an output task, the third performs input, while the fourth exists in every program and indicates where to begin execution.

Building a program is like building a computer. Today's computer is built by connecting boards. Each board is a group of connected chips, which consist of an integrated group of circuit components constructed by connecting logic elements.

A large program is constructed similarly. At the top level, the program includes several modules, where each module is developed separately (and stored in a separate file.) Each module is composed of object declarations and functions. These functions, in turn, call other functions.

In a well-designed program, the purpose, or task, of each function is clear and easy to describe. All its actions hang together and work at the same level of detail. No function is very long or very complex; each is short enough to comprehend in its entirety. Complexity is avoided by creating and calling other functions to do subtasks. In this way, a highly complex job can be broken into short units that interact with each other in controlled ways. This allows the whole program to be constructed and verified much more easily than a similar program written as one massive unit. Each function, then each module, is developed and debugged before it is inserted into the final, large program. This is how professional programmers have been able to develop the large, sophisticated systems we use today.

One function is special. In C, the main program is a special function. In most ways, it is like any function, but `main()` is different in three significant ways:

- Every program must have a `main()` function.
- `main()` is the only function with two standard prototypes:

```
int main( void ); // appropriate for simple programs
int main( int argc, char* argv[] ); // used by some advanced programs
```

- Both prototypes of `main()` are known to the compiler; they do not need to be declared.

5.1 Libraries

We use functions from a variety of sources. Many come to us as part of a library, which is a collection of related functions that can be incorporated into your own code. The **standard libraries** are defined by the C language

standard and are part of every C compiler. In addition, software manufacturers often create proprietary libraries that are distributed with the C translator and provide facilities not covered by the standard. Often, a group of computer users shares a collection of functions that become a **personal library**. An individual programmer might use functions from all of these sources and normally also defines his or her own functions that are tailored to the tasks of a particular program.

Using library functions lets a programmer take advantage of the skill and knowledge of experts. In modern programming practice, code libraries are used extensively to increase the reliability and quality of programs and decrease the time it takes to write them. A good programmer does not reinvent the wheel.

5.1.1 Standard Libraries

C has about a dozen standard libraries; we use six of them in this text. The first one encountered by a beginning C programmer is the standard input/output library (`stdio`), which contains the functions `scanf()` and `printf()` that we have been using since Chapter 3. This library also contains functions for the input and output of specific data types, which will be explained as those types are introduced, as well as functions for file handling, which will be explained in Chapter 14.

The second most commonly used library is the mathematics library (`math`). This library contains implementations of mathematical functions such as `sin()`, `cos()` and `log`. The program examples in this chapter illustrate the use of several of these functions; a complete list is given in Figure 5.6.

Another important library is the standard library (`stdlib`). It contains functions for generating random numbers; a definition for `abs()`, the absolute value function for integers; and a variety of general utility functions. Several of these will be introduced as the need arises in later chapters. Other libraries that we use are the time library (`time`), the string library (`string`), and the character-handling library (`ctype`).

5.1.2 Other Libraries

The standard C libraries contain many useful functions for input and output, string handling, mathematical computations, and systems programming tasks. These libraries provide expert solutions for common needs, but they cannot cover every possible need. The standard libraries contain only a fraction of the useful functions that might be written. The library functions are general-purpose utilities; many were designed for the convenience of programmers creating the UNIX operating system. They are not tailored to the needs of students or scientists and engineers who write C programs in the course of their work.

Many C implementations have additional libraries; for example, a graphics library for building screen displays. Special-purpose libraries are often included with hardware that will be connected to a computer. For example, a mobile-robotics kit such as `Lego Mindstorm` includes a library of functions that are used to communicate with the robotics hardware. Finally, many companies that create software have libraries of code relating to their products; by sharing these libraries, employees can become more efficient and products become more uniform and predictable.

Programmers define their own functions to meet their needs. Some are special-purpose functions written for one application and not relevant to other jobs. However, every programmer builds a collection of function definitions that are useful again and again. These usually are simple functions that save a little writing, simplify a repetitive task, or make a programmer's job easier. Often, such a collection is shared with coworkers and becomes a personal library. In this chapter, we suggest several functions that you may wish to put into your own personal library because they will be useful again and again. We call this library "mytools"; it is discussed in Section 5.9.2.

5.1.3 Using Libraries

Prototypes Every data object has a type. The type of a literal is evident from its form; the type of a variable is declared along with the variable name. Similarly, every function has a type, called a **prototype**, which must be known before the function can be used. The prototype defines the function's **interface**; that is, how it is supposed to interact with other functions. It declares the number and types of arguments that must be provided in every call and the kind of answer that the function computes and returns (if any). This information allows the compiler to check for syntax errors in the function calls.

Nine tenths of an iceberg floats under the surface of the water; we see only a small part on top. Similarly, each C library has two parts: a small public header file that declares the interface for the library and a large, concealed code file that supports the public interface.

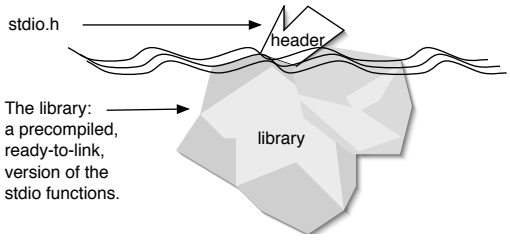


Figure 5.1. A library is like an iceberg: only a small part is visible.

Header files. Each standard library has a corresponding header file whose name ends in `.h`. The **header file** for a library contains the prototype declarations for all of the functions in that library. It may also define related types and constants. For example, the header `time.h` defines a type named `time_t`, which can be used to store the current date and time. This type is related to type `int`, and is chosen to be appropriate for the local computer hardware and software. A useful constant, `INT_MAX` (the largest representable integer) is defined in `limits.h`. Also, the mathematical constant `PI` is defined in `math.h` by many C implementations.

The header files for the libraries we have mentioned so far are

Standard input/output library: `<stdio.h>`	Standard library: `<stdlib.h>`
Mathematics library: `<math.h>`	Time library: `<time.h>`
Character handling: `<ctype.h>`	String library: `<string.h>`
Biggest and smallest integers `<limits.h>`	Personal library: `"mytools.h"`

To use one of the library functions in a program, you must include the corresponding header file in your program. This can be done explicitly, by writing an `#include` command for that library, or you can make your own header file that includes the header files for the standard libraries that will be used. Suppose you have such a file called `mytools.h`, that includes `stdio.h`. Then if you write the command `#include "mytools.h"` in your program, there is no need to write `#include <stdio.h>` separately.

In an `#include` command, angle brackets `<...>` around the name of the header file indicate that the header and its corresponding library are installed in the compiler's standard library area on the hard disk[1]. Use quotation marks instead of angle brackets for personal libraries like `mytools` that are stored in the programmer's own disk directory rather than in the standard system directory.

[1]This file must be stored where your compiler can find it. It always works to put it in the same directory as your program code, or in any directory that is on the compiler's "search path". To find out about the search path, ask your system administrator to help you.

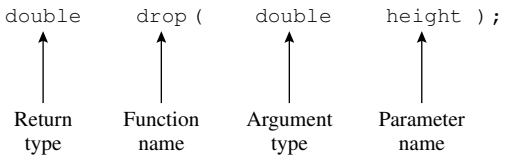


Figure 5.2. Form of a simple function prototype.

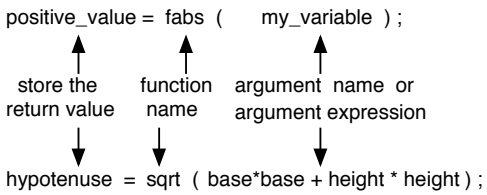


Figure 5.3. Form of a simple function call.

5.2 Function Calls

A **function call** causes the function's code to be executed. The normal sequential **flow of control** is **interrupted**, and control is transferred to the beginning of the function. At the end of the function, control returns to the point of interruption. To call a function, write the name of the function followed by a pair of parentheses enclosing a list of zero or more **arguments**. In the following discussion, we refer to the function that contains the call as the **caller** and to the called function as the **subprogram** or, if there is no ambiguity, simply the **function**. Copies of the argument values are made by the caller and sent to the subprogram, which uses these arguments in its calculations. At the end of function execution, control returns to the caller; a **function result** also may be returned.

The function call must supply an argument value for each parameter defined by the function's prototype. The form of a simple prototype declaration is shown in Figure 5.2. It starts with the type of the answer returned by the function. This is followed by the function name and a pair of parentheses. Within the parentheses are zero or more **parameter declaration** units, consisting of a type and an identifier. The type tells us what kind of argument is expected whenever the function is called. The identifier is optional in a prototype (but required in a function definition).

The C compiler checks every function call to ensure that the correct number of arguments has been provided and that every argument is an appropriate type for the function, according to the function's prototype. It also checks that the function's result is used in an appropriate context. Generally, if a mismatch is found between the function's prototype and the function call, the compiler generates an error comment and does not produce a translated version of the code. Some type mismatches are legal according to the type rules of C but they may not be meaningful in the context of the program. In such cases, the compiler generates a warning comment, continues the translation, and produces an executable program. However, the programmer should never ignore warnings; most warnings are clues about logic errors in the program.

Calling library functions. The program example in Figure 5.4 demonstrates how to include the library header files in your code and how to call the library functions. It uses standard I/O functions, the `sqrt()` function from the mathematics library, and a function from the `stdlib` library to abort execution after an input error.

Notes on Figure 5.4. Calling library functions.

First box: the `#include` ***commands.***

- We `#include <stdio.h>` so that we can use `printf()` and `scanf()`.
- We `#include <math.h>` for `sqrt()`, the square root function,
- We `#include <stdlib.h>` for the `exit()` function, which is discussed in Section 5.2.3.

Second box: calling `exit()`***.***

- The function `exit()` is defined in `stdlib`; its prototype is `void exit( int );`. It can be used to exit from the middle of a program after an error that makes continuation meaningless.
- The symbol `EXIT_FAILURE` is defined as 1. We use the symbolic name here, rather than a literal 1, as a form of program documentation.

- We use a simple `if` statement to test for an input error. If one is found, we display an error message, then call `exit(EXIT_FAILURE)` or `exit( 1 )`. When control returns to the system, it will display a termination comment with the exit code that written in the call on `exit()`.

```
Grapefruits and Gravity with Functions

Calculate the time it would take for a grapefruit
to fall from a helicopter at a given height.
Enter height of helicopter (meters): -2
Error: height must be >= 0.  You entered -2

Grapefruits has exited with status 1.
```

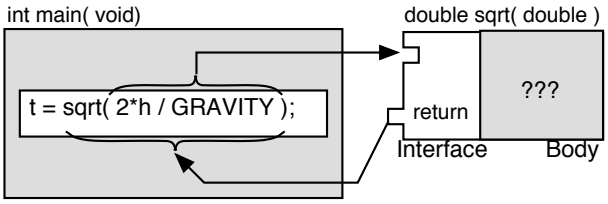
- No `else` statement is needed because `exit()` takes control immediately and directly to the end of the program. In the flow diagram (Figure 5.17), it is diagrammed as a bolt of lightning because it "short-circuits" all the normal control structures.

Third box: calling `sqrt()`.

```
// ------------------------------------------------------------------------
// Grapefruits and Gravity again, with terminal velocity, using sqrt().
//
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define GRAVITY 9.8

int main( void )
{
    double height;                  // height of fall (m)
    double time;                    // time of fall (s)
    double velo;                    // terminal velocity (m/s)

    printf( " Grapefruits and Gravity with Functions\n\n"
            " Calculate the time it would take for a grapefruit\n"
            " to fall from a helicopter at a given height.\n"
            " Enter height of helicopter (meters):  " );
    scanf( "%lg", &height );        // keyboard input for height

    if (height < 0) {               // exit gracefully after error
        printf( " Error: height must be >= 0.  You entered %g\n", height );
        exit( EXIT_FAILURE );       // abort execution
    }

    time = sqrt( 2 * height / GRAVITY );  // calculate the time of fall
    velo = GRAVITY*time;                  // terminal velocity of fruit
    printf( "      Time of fall = %g seconds\n", time );
    printf( "      Velocity of the object = %g m/s\n", velo );

    return EXIT_SUCCESS;
}
```

Figure 5.4. Calling library functions.

- The `sqrt()` function computes the square root of its argument. To call this function, we write the argument in parentheses after the function name. In this call, the argument is the value of the expression `2*h/GRAVITY`. The multiplication and division will be done, and the result will be sent to the `sqrt()` function and become the value of the `sqrt()`'s parameter. Then the square root will be calculated and returned to the caller. When a function returns a result, the caller must do something with it. In this case, it is stored in the variable named `t`, then used in the next two lines of `main()`.
- A function prototype describes the function's interface; that is, the argument(s) that must be brought into the function and the type of result that is sent back. In this case, one argument is brought in and one result is returned. We can diagram the passage of information to and from the function like this:



- In general, spacing makes no difference to the compiler. We could have written

```
time=by the standard to be an ( 2*height/GRAVITY); or
time =sqrt (2 *height / GRAVITY); or
time= sqrt (2* height/GRAVITY);
```

  However, spacing makes an important difference in the readability of a program. You should use spacing selectively to make formulas as readable as possible. Current style guidelines call for spaces after the opening parenthesis and before the closing parenthesis.

The output from a successful run.

```
Grapefruits and Gravity with Functions

Calculate the time it would take for a grapefruit
to fall from a helicopter at a given height.
Enter height of helicopter (meters):  30
     Time of fall = 2.47436 seconds
     Velocity of the object = 24.2487 m/s

Grapefruits has exited with status 0.
```

Fourth box: the `return` statement.

- In previous programs, we have written `return 0;`. Here we introduce another way to write the same thing: `return EXIT_SUCCESS`. The `stdlib.h` header file defines `EXIT_SUCCESS` to be a synonym for 0, and some programmers prefer to use the symbolic name rather than the numeric value.
- The last line of the output, above, shows the message printed by the operating system after the program terminated. The status code that is displayed is the value that was written in the return statement.

5.2.1 Call Graphs

We use flow diagrams (as in Figure 5.17) to visualize the flow of control through the statements of a single function or the transition from one function to another during execution. Another kind of diagram, a **function call graph**, is useful for showing the relationships between functions that are established by function calls. In a function call graph (see Figure 5.5), a box at the top is used to represent the function `main()`. Below it, attached to the branches of a bracket, is one box for each of the functions called by the main program.[2] As far as possible, these are listed left to right in the order in which they appear in the program. Each function has only one box; if it is called several times, there is no sign of that in the diagram. A very simple program is graphed in Figure 5.5.

[2]Various elaborations of this basic scheme are in use. In one version, the function arguments are written on the arrows. We choose to introduce the concepts by using the simplest scheme.

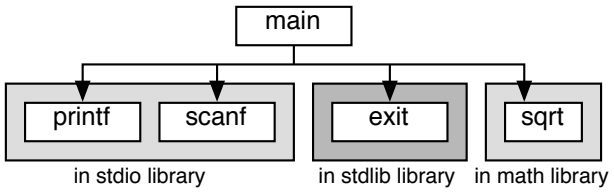


Figure 5.5. Call graph for Figure 5.4: Calling library functions.

The pattern of one box pointing at others is repeated when diagramming a more complex program; a box and a connection are drawn for each function called by a first-level function. In Figure 5.12, the main program calls a programmer-defined function, `banner()` (Figure 5.13), which in turn, calls functions named `time()` and `ctime()` from the `time` library. The resulting function call graph, shown in Figure 5.14, has boxes at three levels. In a large program, a function call graph will have many boxes at several levels and may have complex dependencies among the functions (arrows may point from a lower level to an upper level). We will use call graphs to visualize the relationships among functions in future program examples. The graph becomes more and more important as the number of functions increases and the interactions among functions become more complex.

In some programs it becomes beneficial to add information to the call graph concerning what is being passed between the functions. This would require an arrow for every parameter in every function call. If a program makes many function calls, this can get very complicated very quickly. Therefore, for the sake of clarity, we omit parameter and return information from the call graphs presented in this text.

5.2.2 Math Library Functions

Most of the functions in the math library are familiar to anyone who has studied high-school algebra and trigonometry. These include the trigonometric, exponential, log, and square root functions. Some of the

Name	Function	Argument type(s)	Return type
`fabs(x)`	Absolute value	`double`	`double`
`ceil(x)`	Round x up	`double`	`double`
`floor(x)`	Round x down	`double`	`double`
`rint(x)`	Round x to nearest integer	`double`	`double`
`cos(x)`	Cosine of x	`double`	`double`
`sin(x)`	Sine of x	`double`	`double`
`tan(x)`	Tangent of x	`double`	`double`
`acos(x)`	Arc cosine of x	`double`	`double`
`asin(x)`	Arc sine of x	`double`	`double`
`atan(x)`	Arc tangent of x	`double`	`double`
`atan2(y, x)`	Arc tangent of y/x	`double, double`	`double`
`cosh(x)`	Hyperbolic cosine of x	`double`	`double`
`sinh(x)`	Hyperbolic sine of x	`double`	`double`
`tanh(x)`	Hyperbolic tangent of x	`double`	`double`
`exp(x)`	e^x	`double`	`double`
`log(x)`	Natural log of x	`double`	`double`
`log10(x)`	Base 10 log of x	`double`	`double`
`sqrt(x)`	Square root	`double`	`double`
`pow(x, y)`	x^y	`double, double`	`double`
`fmod(x,y)`	$x - N \times y$ for largest N such that $N \times y < x$	`double, double`	`double`

Figure 5.6. Functions in the math library.

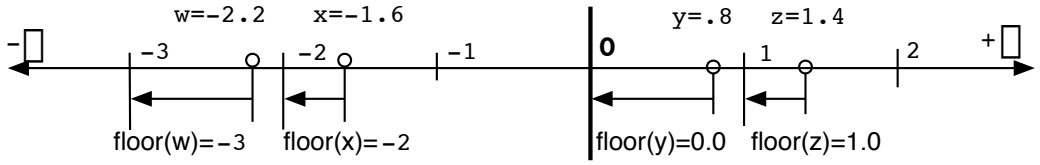


Figure 5.7. Rounding down: `floor()`.

math library functions are less familiar; these are briefly explained in the next several paragraphs, with a few important functions from other libraries.

int abs(int n). Although this is a mathematical function, it is part of the standard library (stdlib), not the math library; to use it, a program must `#include <stdlib.h>`. The result is the absolute value of n. (That is, if n is negative, the result has the same value as the argument, but with a positive sign instead of a negative sign. If the argument is zero or positive, the result is the same as the argument.)

double fabs(double x). To use this function, `#include <math.h>`. This is just like `abs()` but it works for a `double` argument instead of an `int`, and it returns a `double` result. The result is the absolute value of x. (That is, if x is negative, the result has the same value as the argument, but with a positive sign instead of a negative sign. If the argument is zero or positive, the result is the same as the argument.)

double fmod(double x, double y). To use this function, `#include <math.h>`. C has an operator, `%`, that computes the modulus function on integers; `fmod()` computes a related function for two `double`s. The result of `fmod(x, y)` is the floating-point remainder of `x/y` More precisely, `fmod(x, y) = x - k*fabs(y)` for some integer value `k`. The result has the same sign as `x` and is less than the absolute value of `y`. The function is implementation-defined if `y==0`. A few examples might make this clear:

```
fmod( 10.0, 2.0 ) = 0        and k = 5
fmod( -10.0, 2.9 ) = -1.3    and k = 3
fmod( 10.5, -1.0 ) = .5      and k = 10
fmod( 10.5, 1.1 ) = .6       and k = 9
fmod( 34.5678, .01 ) = 0.0078
```

The last example, above, hints at an application for `fmod()`. Suppose a bank calculates the amount of interest due on an account, but will add only an even number of cents to the account balance. The fractional cents must be subtracted from the calculated interest before adding the interest to the account balance. This function lets us easily calculate the fractional cents.

double atan2(double x, double y). This function computes the arc tangent of `x/y`. It is explicitly defined for `y=0` is $\pi/2$ and has the same sign as `x`. This should be used in place of `atan()` for any argument expression that might have a denominator of 0.

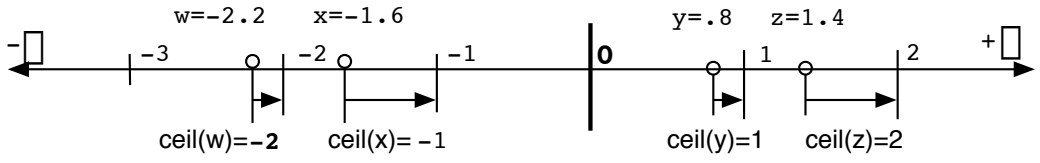


Figure 5.8. Rounding up: `ceil()`.

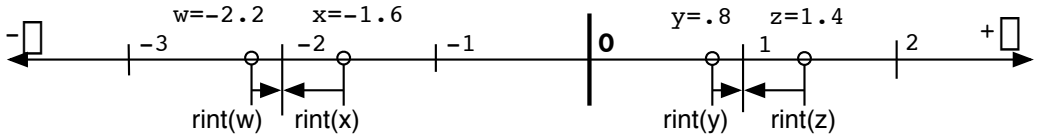


Figure 5.9. Rounding to the nearest integer: `rint()`.

Rounding and truncation. Suppose that we are given a `double` value and wish to eliminate the fractional part. The math library provides three functions that correspond to three of basic ways to do this job. The assignment operator provides a fourth way.

- The function `floor()` rounds down, that is, the result will be an integral value that is closer to $-\infty$ than the argument, as illustrated in Figure 5.7. This function returns a `double` value. For example, `floor(-1.6)` is `-2.0` and `floor(.9999999)` is `0.0`.
- The function `ceil()` (short for ceiling) rounds up, that is, the result will be an integral value that is closer to $+\infty$ than the argument, as illustrated in Figure 5.8. This function returns a `double` value.
- The function `rint()` does what we normally refer to as "rounding". It rounds to the nearest integral value and returns it as a `double` value, as illustrated in Figure 5.9. A matching function, `lrint()`, returns the result as an integer, if it is possible to represent it as an integer. For example, `rint(-1.6)` is `-2.0` and `rint(.9999999)` is `1.0`. Compare this result to `lrint(.9999999)` = `1`
- When a double value is assigned to an integer variable, the fractional part is *truncated*, that is, the decimal places are simply discarded. This is the same as rounding positive numbers down (toward 0) and rounding negative numbers up (toward 0).

5.2.3 Other Important Library Functions

The date and time. The date and time at which a program is executed is very important for many kinds of output, including student work. Modern computers have an internal battery-operated clock. Modern systems keep that clock accurate by periodically synchronizing it to a time standard accessed over the internet. C provides a variety of functions and type definitions to help programmers use the clock[3]. Most of the time library is too difficult for this chapter, but a few items can be used simply and are presented here.

- The data type `time_t` is defined[4] by the standard to be an integer that has enough bits to hold whatever encoding of date and time is used by the local system. It might be different from system to system, but it is always exactly right to store the time. The program in Figure 5.13 shows how to declare a variable of this type and use it to store the current time.
- The `time( )` function reads the system clock and returns an integer encoding of the time and date. It is described in Appendix **??** and discussed in more detail in Chapter 12. Until then, if you want to read the system clock, you should call the `time( NULL )` and store the result in a `time_t` variable.
- The `ctime( )` function is used to convert the time from the coded form to a string that can be easily printed and understood. The easiest way to use `ctime( )` is to call the function from the argument list of

[3]These are described in Appendix **??** and parts are discussed in detail in Chapter 12.

[4]Until now, we have covered only three types: `double`, `int`, and `char`. The C language actually supports many predefined types and permits the programmer to define his own. These will be introduced gradually in future chapters.

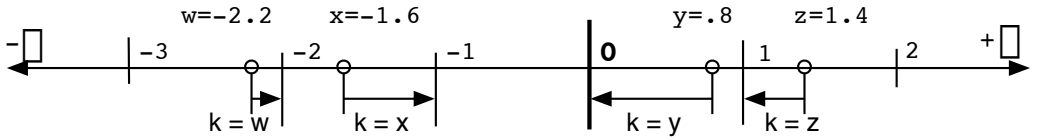


Figure 5.10. Truncation via assignment.

Library	Purpose	Name and Usage	Argument	Return type
stdlib	Absolute value	`k = abs( amount );`	`int`	`int`
stdlib	Abort execution after an error	`exit(1)`	`int`	returns to system
time	Data type for storing the time	`time_t now;`		creates a time variable
time	Read system clock	now = time(NULL)	NULL	the time encoded as an integer
time	Convert time code for printing	printf(ctime(&now))	integer	a printable string

Figure 5.11. A few functions in other libraries.

a `printf()` function. The argument to `ctime( )` must be the address of the `time_t` variable that was set by calling `time(NULL)`. Figure 5.13 shows how to do this.

Exception conditions. If a program encounters a serious error and cannot meaningfully continue, the appropriate action is to stop execution immediately. C++ and Java are newer languages that are built upon C; both include all the C operators and control structures, but provide an additional modern facility, called an exception handler, for managing errors and other unusual and unexpected conditions. C is an old language and it does not provide an exception handler. However, it does provide a function that aborts a program and "cleans up" the environment before returning to the system. In this text, the `exit()` function will be used when it would be appropriate to use an exception in a modern language.

- The function `exit()` is defined in `stdlib`; its prototype is `void exit( int );`. It can be used to exit from the middle of a program after an error that makes continuation meaningless.
- The constant `EXIT_FAILURE` is defined in `stdlib.h` as a synonym for the number 1 and the constant `EXIT_SUCCESS` is defined as a synonym for the number 0. These can be used as arguments to `exit()`, but a programmer can also use any integer as the argument or invent his or her own codes. The system should display the code on the screen after the program exits.

5.3 Programmer-Defined Functions

Functions serve three purposes in a program: They make it easy to use code written by someone else; they make it possible to reuse your own code in a new context; most important, though, they permit breaking a large program into small pieces in such a way that the interface between pieces is fixed and controllable. A programmer may (and generally does) modularize his or her program by dividing the entire job into smaller tasks and writing a **programmer-defined function** for each task.

Functions can take no arguments or many, of any combination of types, and can return or not return values. The type of a function is a composite of the type of value it returns and the set of types of its arguments.

Function	Prototype
`main()`	`int main( void );`
`exit()`	`void exit( int );`
`sqrt ()`	`double sqrt( double );`

We begin the study of programmer-defined functions by creating functions of two types: double→double and void→void.

Double→double functions. Some functions calculate and return values when they are called. For example, in Figure 5.4, the function `sqrt()` calculates a mathematical function and returns the result to `main()`, which stores it in a variable and later prints it. The functions `sqrt()`, `log()`, `sin()`, and `cos()` all accept an argument of type `double` and return an answer of type `double`. We say, informally, that these are **double→double functions** because their prototypes are of the form `double funcname(double)`. A double→double function must be called in some context where a value of type `double` makes sense. Often, these functions are called from the right side of an assignment statement or from a `printf()` statement. Examples of calls on double→double functions follow:

```
time = sqrt( 2 * height / GRAVITY );
printf( "The natural log of %g is %g\n", x, log( x ) );
```

Void functions. Some functions return no value to the calling program. Their purpose is to cause a side effect; that is, perform an input or output operation or change the value of some memory variable. These functions are called *void functions* because their prototypes start with the keyword `void` instead of a return type. *Void* means "nothing"; it is not a type name, but we need it as a placeholder to fill the space a type name normally would occupy in a prototype. We need some keyword because, if the return-type field in a function header is left blank, the return type defaults to `int`.[5]

Some `void` functions, such as `exit()`, require arguments; others, do not. The latter are called **void→void functions** and have prototypes of the form `void funcname( void )`. To call a void→void function, write the function name followed by empty parentheses and a semicolon. Often, the function call will stand by itself on a line.

5.3.1 Function syntax.

A function has two parts: a **prototype** and a **definition**. When writing a program, prototypes for all the functions are normally written at the top, between the preprocessor commands and the beginning of `main()`. Function definitions are written at the bottom of the file, following the end of `main()`.

A function definition, in turn, has two parts: a **function header** (which must correspond to the prototype) and a **function body**, which is a block of code enclosed in curly brackets. The body starts with a series of (optional) declarations. These create local variables and constants for use by the function. The local declarations are followed by a series of program statements that use the local variables and the function's arguments (if any) to compute a value or perform a task. The body may contain one or more `return` statements, which return a value to the calling program.

The complete set of rules for creating and using functions in C is extensive and complex; it is presented in some detail in Chapter 9. In this section, we begin by writing the two types of functions discussed above. We illustrate how to define these function types, write prototypes for them, call them, and draw flowcharts (using barred boxes) to show the flow of control. We give a few examples and some brief guidelines so that the student may begin using functions in his or her own programs. The next figures illustrate, in context, how to write the parts of void→void and double→double functions.

5.3.2 Defining Void→Void Functions

We show how to write void→void functions first, using Figure 5.12 to illustrate the discussion. This program uses a void→void function to print user instructions and error comments. We ask the user to enter the number of passengers in a car. If the input number is greater than 5, we print an error message and beep four times.

Notes on Figures 5.12 and 5.13. Calling void→void functions.

First box: the standard header files.

- The prototypes for the time library and the standard I/O library are brought into the program by these `#include` statements.
- The header `<time.h>` is included because one of the programmer-defined functions will call functions from the time library. When we include prototypes at the top of a program, either explicitly or by including a header file, the corresponding functions can be called anywhere in the program.

Second box: the prototypes.

- Either a prototype declaration or the actual function definition should occur in a program before any calls on the function.[6]
- The include statements bring in prototypes for the standard functions. However, we must write prototypes for the three functions defined at the bottom of this program, `beep()`, `instructions()`, and `banner()`.

[5] This default makes no sense in ISO C; it is an unfortunate holdover from pre-ISO days, when C did not even have a type `void`. The default to type `int` was kept in ISO C to maintain compatibility with old versions of C.

[6] If a function is called before it is declared, the C compiler will construct a prototype for that function that may or may not work properly.

The banner function is shown in Figure 5.13.

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

// Prototype declarations for the programmer-defined functions. ----------
void banner( void );
void instructions( void ) ;
void beep( void );

int main( void )
{
    int n_pass;                        // Number of passengers in the car.

    banner();                         // Display output headings.
    instructions();                   // Display instructions for the user.

    scanf( "%i", &n_pass );

    if (n_pass > 5) beep();     // Error message for n>5
    else printf( " OK! \n" );   // Success message for good entry
    return 0;
}

// ------------------------------------------------------------------------
void instructions( void )                              // function definition
{
    printf( " This is a legal-passenger-load tester for 6-seat sedans.\n"
            " Please input the number of passengers you will transport: " );
}

// ------------------------------------------------------------------------
void beep( void )                                      // function definition
{
      printf( " Bad data! \n\a\a\a\a" );        // error message and beeps
}
```

Figure 5.12. Using void→void functions.

This function is part of the program in Figure 5.12, and belongs at the bottom of the same source-code file.

```
// ------------------------------------------------------------------------
void banner( void )                // Print a neat header for the output.
{
   time_t now = time(NULL);
   printf( "\n-------------------------------------------------------\n" );
   printf( "    Patience S. Goodenough\n    CS 110\n    ");
   printf( ctime( &now ) );
   printf(  "-------------------------------------------------------\n" );
}
```

Figure 5.13. Printing a banner on your output.

Library functions are surrounded by shaded boxes; programmer-defined functions are shown with no surrounding box.

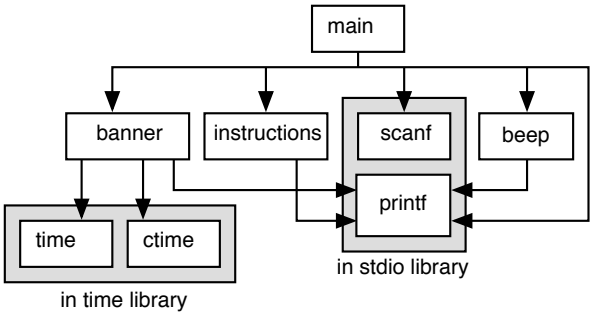


Figure 5.14. A call graph for the beep program.

- The prototype for every void→void function follows the simple pattern shown here: the word `void` followed by the name of the function, followed by the word `void` again, in parentheses. Every prototype ends in a semicolon.

Third and fourth boxes: the function calls.

- These lines all call a void→void function. The call consists of the function name followed by empty parentheses. Often, as with the calls on `instructions()` and `banner()`, a void→void function call will stand alone on a line.
- Often a void→void function is used to give general instructions or feedback to the user, as in the second box. Sometimes, one is used to inform the user that an error has happened, as is the case with the call on `beep()` in the third box. Such calls often form one clause of an `if` statement.

Fourth and fifth boxes: two easy function definitions.

- We define two void→void functions: `instructions()` and `beep()`. We use a comment line of dashes to mark the beginning of each function so that it is easy to locate on a video screen or printout.
- Each function definition starts with a header line that is the same as the prototype except that *it does not end in a semicolon.* Following each header line is a block of code enclosed in curly brackets that defines the actions of the function.
- Most void→void functions, like these two, perform input or output operations. The symbol `\a`, used in the `beep()` function, is the ASCII code for a beeping noise. Many systems will emit an audible beep when this symbol is "printed" as part of the output. Other systems may print a small box instead of emitting a sound. Some systems give no audible or visible output.

The output

Here is the output from two test runs (the second banner has been omitted):

```
------------------------------------------------------
    Patience S. Goodenough
    CS 110
    Sat Aug  9 18:21:17 2003
------------------------------------------------------
 This is a legal-passenger-load tester for 6-seat sedans.
 Please input the number of passengers you will transport: 2
 OK!

Tester has exited with status 0.
------------------------------------------------
 This is a legal-passenger-load tester for 6-seat sedans.
 Please input the number of passengers you will transport: 10
 Bad data!

Tester has exited with status 0.
```

Figure 5.13 a longer function definition.

- We define a void→void function named `banner()` that prints a neat and visible output header containing the programmer's name and the date and time when the program was executed. The date and time are produced by calling functions from the `time` library.
- The first line of this function declares a variable named `now` of type `time_t` and initializes it to the current time by calling `time( NULL )`.
- Then we call `ctime()` to convert the coded date and time into a string that can be printed. The third call on `printf()` shows how do use this function. Be sure to write the ampersand when you copy this code.
- An output header, similar to the one this function produces should be part of the output produced by every student program. You should use it with everything you hand in to the teacher. One way to do this is to start your own personal file, say `mytools.c` containing reuseable code. Make a matching file called `mytools.h` containing the prototypes for the functions in `mytools.c`. Include `mytools.h` in every main program you write, and add both `mytools` files to the project you create for your program.

5.3.3 Returning Results from a Function

A function that returns a value is fundamentally different from a `void` function. A `void` function simply causes some side effect, such as output, like `banner()` in Figure 5.19. The call on such a function forms a separate statement in the code. In contrast, a function that returns a value interacts with the rest of the program by creating information for further processing. A `return` statement is used to send a result from a function back to the caller. It is represented in the diagram in Figure 5.21 as a tab sticking out of the function's interface. A `return` statement can be placed anywhere in the function definition, and more than one `return` statement can be used in the same function.[7]

A function that returns a value can be called anywhere in a C statement that a variable name or literal of the same type would be permitted. Often, as in the call on `f()` in Figure 5.19, a function is called in an assignment statement. The return address for this call is in the middle of the statement, just before the assignment happens. When the value is returned from the call, `main()` will resume execution by assigning the returned value to the variable `z`.

If a function is called in the middle of an expression, the result of the function comes back to the calling program in that spot and is used to compute the value of the rest of the expression. The call on `exp()` in Figure 5.19 illustrates this. The function is called from the middle of a `return` statement: `return y * exp(y)`. The return address for this call is in the middle of the statement, just before the multiplication happens. After a value is returned by `exp()`, it will be multiplied by the value of `y` and the result returned to `main()`.

Finally, as in the call on `g()` in Figure 5.19, a function can be called from the argument list of another function. It is quite common to nest function calls in this way.

5.3.4 Arguments and Parameters

Function parameters introduce variability into the behavior of a function. A void→void function without parameters always does the same thing in the same way.[8] In contrast, introducing even one parameter permits the actions of a function and its results to depend on the data being processed. By parameterizing a piece of code, we can make it useful under a much more general set of circumstances.

Formal parameters are part of a function definition and specify a set of unknowns; **arguments** are part of a function call and supply values for those unknowns. In Figure 5.21, parameters are represented by notches along the left edge of each function's interface. Right-facing arrows connect each argument to the notch of the corresponding parameter; these arrows represent the direction in which information flows from the caller to the subprogram.

The function `f()` has one parameter, a `double` value named `y`. Even if other objects in the program have the same name, the parameter `y` in `f()` will be a distinct object, occupying a separate memory location. It is quite common to have two objects with the same name defined in different functions.

Looking at the list of library functions in Figure 5.6, we see that the `exp()` function has one parameter of type `double`. The name of this parameter is not known because `exp()` is a library function and its details have

[7] However, we strongly recommend using a single `return` statement at the end of the function.

[8] An exception to this occurs if the function uses global variables or user input.

been concealed from us. During the calling process, the argument value is stored in the parameter variable, making a complete object.

A function can be `void` or have one or more parameters. Its prototype defines the correct **calling sequence**; that is, the number, order, and types of the arguments that must be written in a call. The call must supply one argument expression per parameter;[9] if the number of arguments does not match the number of parameters, the program will not compile. When a function call is executed, each argument expression in the call will be evaluated and its value will be passed from the caller to the subprogram, where it will be stored in the corresponding parameter. For example, the argument in the call on `f()` is the value of the variable named `x` in the main program. This value is a `double`, so it can be stored in the `double` parameter with no conversion.
An ISO C prototype states the name of a function, the types of its parameters, and the return type. The parameters also may be named in the prototype, but such names are optional and often omitted. A function header states the same information, except that parameter names *are required* in the function header.

Inside a function, the parameter names are used to refer to the argument values; the first parameter name in the function header refers to the first argument in the function call, and so on. In Figure 5.21, when `main()` makes the call `f(x)`, the value of `x` is copied into the parameter named `y`. Within the body of `f()`, the value stored in this parameter will be used wherever the code refers to the name `y`. During execution of `f()`, this value is further copied and stored in the parameter variable of `exp()`.

Formal Parameter Names The function header (which is the first line of the function definition) the name of a function, the types of its parameters, and the return type These names provide a way for the programmer to refer to the parameters in the function's code.

Any legal name may be given to a formal parameter. It may be the same as or different from the name of a variable used in the function call, and both can be the same as or different from the optional name in the function's prototype.[10] The names chosen do not affect the meaning of the program because argument values are matched up with parameters by position, not by name. For example, the main program in Figure 5.15 uses a variable named `h` in the call on the `drop()` function (Figure 5.18). Within `drop()`, however, the parameter is named `height`, so the value of `main's h` will be stored in `drop's height`.

5.3.5 Defining a Double→Double Function

We use Figures 5.15 through 5.18 to illustrate the construction and call of a double→double function, which is somewhat more complicated than a void→void function because argument information must be passed into the function and a result must be returned to the caller.

Notes on Figure 5.15. The grapefruit returns.

First box: the prototypes.

- A prototype for a double→double function gives the function name and specifies that it requires one `double` parameter and returns a `double` result. The general form is

  ```
  double function_name( double parameter_name );
  ```

  where the parameter name is optional.
- This box contains a prototype declaration for the function named `drop()`. It states that `drop()` requires one argument of type `double` and returns a `double` result. This information permits the C compiler to check whether a call on `drop()` is written correctly.
- There is also a prototype declaration for the void→void function named `title()`, which is similar to the `instructions()` function in the previous example.

[9] Some functions accept a variable number of arguments; `scanf()` is an example. However, the details of how this is accomplished are beyond the scope of this text.

[10] However, it is good style to use the same name in the prototype and the function header.

Second box: the void→void function call. The prototype for `title()` must precede this function call. The definition of the function is at the bottom of the program, in the fourth box.

A function call interrupts the normal sequence of execution. Look at the flow diagram in Figure 5.17. The first statement is the call on `title()`. When that call is executed, control leaves the main sequence of boxes and travels along the dotted line to the beginning of the `title()` function. Then control proceeds, in sequence, to the return statement, and finally returns to where it came from along the second dotted arrow.

Third box: the double→double function call. This box calls `drop()`. The prototype for `drop()` was given in the first box and the function is defined in Figure 5.18, but would be placed in the same source file, below the definition of `title()`.

The form of any function call must follow the form of the prototype. When we call a double→double

---

A grapefruit is dropped from a helicopter hovering at height `h`. This continues development of the program in Figure 5.4. The `drop()` function is shown in Figure 5.18.

```
// ------------------------------------------------------------------------
// Modify the Grapefruits and Gravity program by using a double→double
// function to compute the travel time of the grapefruit.
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define GRAVITY 9.81

void title( void );
double drop( double height );   // Prototype declaration of drop.

int main( void )
{
    double h;                   // height of fall (m)
    double t;                   // time of fall (s)
    double v;                   // terminal velocity (m/s)

    title();                    // Call function to print titles.
    printf( " Enter height of helicopter (meters): " );
    scanf( "%lg", &h );         // keyboard input for height

    t = drop( h );              // Call drop.  Send it the argument h.
    v = GRAVITY * t;            // velocity of grapefruit at this time
    printf( "    Time of fall = %g seconds\n", t );
    printf( "    Velocity of the object = %g m/s\n", v );
    return 0;
}

// ------------------------------------------------------------------
void title( void ) {
    printf(" Grapefruits and Gravity with a Drop Function\n\n"
           " Calculate the time it would take for a grapefruit\n"
           " to fall from a helicopter at a given height.\n" );
}
```

Figure 5.15. The grapefruit returns.

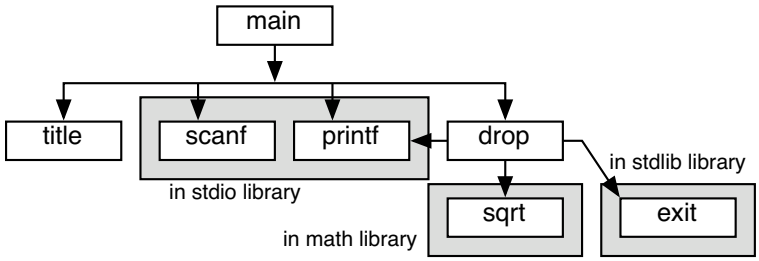


Figure 5.16. Call graph for the grapefruit returns.

function, we must supply one argument expression of type `double`. In this call, the argument expression is a simple variable name, `h`. When we call `drop()`, we send a copy of the value of `h` to the function to be used in its calculations.

A function call interrupts sequential execution and sends control into the function. Figure 5.17 depicts this interruption as a dotted arrow going from the function call to the beginning of the function. From there, control flows through the function to the return statement, which sends control back to the point of interruption, as shown by the lower dotted arrow. When a double→double function returns, it brings back a `double` value, which should be either used or stored. In this example, we store the result in the `double` variable `t`.

Program output. Here is one set of output from the grapefruit program (the banners and termination messages have been omitted):

```
Grapefruits and Gravity with a Drop Function

Calculate the time it would take for a grapefruit
to fall from a helicopter at a given height.
Enter height of helicopter (meters):  872
   Time of fall = 13.3401 seconds
   Velocity of the object = 130.733 m/s
```

Notes on Figure 5.18. Definition of the `drop()` function. We show how to write the definition of a double→double function.

The comment block and the function header.

- Every function should start with a block of comment lines, the **function comment block**, that separate the definition from other parts of the program and describe the purpose of the function. Comment marks (`/*...`) begin the first line and end the last line of the block comment. These comments provide a neat and visible heading for the function.
- The first line of a function definition is called the *function header*. It must be like the prototype except that the parameter name is required (not optional) in the header and the header does not end with a semicolon.
- A parameter can be given any convenient name, which need not be the same as the name of the argument that will appear in future function calls. A new variable is created for the parameter and can be used only by the function itself.
- When a function is called, the expression in parentheses is evaluated and its value passed into the function and used to initialize the parameter variable. This value is called the *actual argument*. Within the function, the parameter name is used to refer to the argument value. In the function call, the argument was `main()`'s variable `h`, but the `drop()` function's parameter is named `height`. This is no error. *A double→double function can be called with any* `double` *argument value.* The argument can be the result of an expression, a variable with the same name, or a variable with a different name. Within the function, the parameter name (`height`) is used to refer to the argument value.

This is a flow diagram of the program in Figures 5.15 and 5.18. Function calls are depicted using a box with an extra bar on the bottom to indicate a **transfer of control**. The dotted lines show how the control sequence is interrupted when the function call sends control to the beginning of the `drop()` function. Control flows through the function then returns via the lower dotted line to the box from which it was called. The call on `fatal()` (after an error) ends execution immediately and returns control to the operating system.

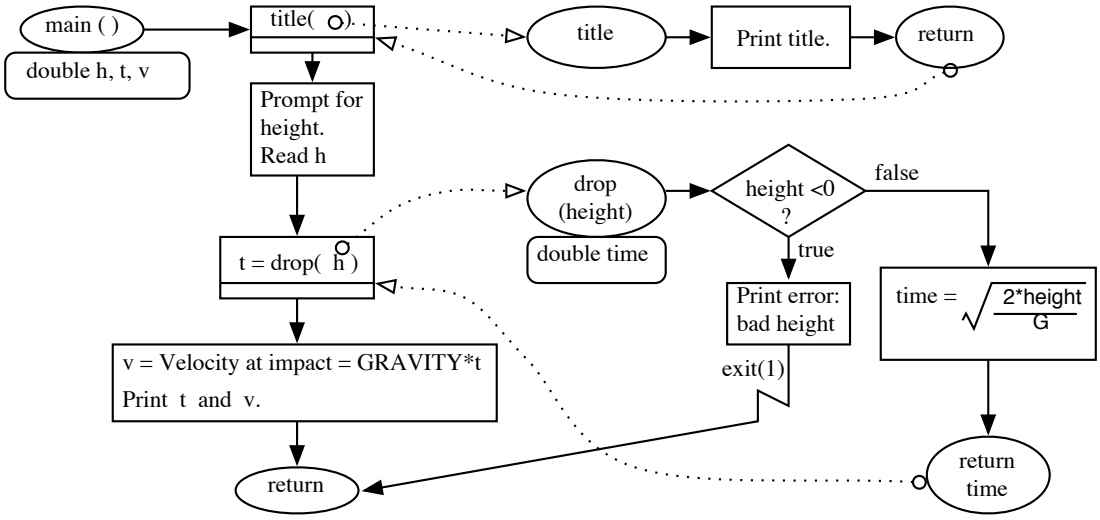


Figure 5.17. Flow diagram for the grapefruit returns.

This function is called from Figure 5.15.

```
// ------------------------------------------------------------------------ ---
// Time taken for an object dropped from height meters to hit the ground.
double drop( double height )
{
    double time;                            // create a local variable

    if (height < 0) {                       // exit gracefully after error.
        printf( " Error: height must be >= 0.  You entered %g\n", height );
        exit( 1 );                          // abort execution
    }
    time = sqrt( 2 * height / GRAVITY ); // calculate the time of fall

    return time;
}
```

Figure 5.18. Definition of the `drop()` function.

First box: a local variable declaration. The code block of a function can and usually does start with declarations for **local variables**. Memory for these variables will be allocated when the function is called and deallocated when the function returns. These variables are for use only by the code in the body of the function; no other function can use them. In the example program, we declare a local variable named `time`.

Second box: the function code. Statements within the function body may use the parameter variable and any local variables that were declared. References also may be made to constants defined globally (at the top of the program). The use of global variables is legal but strongly discouraged in C. The statements in this function are like the corresponding lines of Figure 5.4 except that they use the parameter name and local variable name instead of the names of the variables in the main program.

Inner box: calling another function. An error has been detected, so we print an appropriate comment and call `exit( 1 )` to terminate the program immediately.

Last box: the `return` ***statement.*** On completion, a double→double function must return a result of type `double` to its caller. This is done using the `return` statement. A `return` statement does two things: it sends control immediately back to the calling program and it tells what the return value (the result) of the function should be. It does not need parentheses. In Figure 5.18, the result from `sqrt()` is stored in the local variable `time`. To make that answer available to the caller, we return the value of `time`. On executing the `return` statement, the value of `time` is passed back to the caller and control is returned to the caller at the statement containing the function call, as depicted by the dotted line in Figure 5.17.

5.4 Organization of a Module

Generally, a program has a `main()` function that calls several other library and programmer-defined functions. To compile `main()`, the compiler needs to know the **prototype** of every function called. One way this information can be supplied is by putting `main()` at the bottom of the module, while the definitions of the other functions come before it. However, many programmers dislike having the main program at the end of the file and it is customary, in C, to put `main()` at the top. When this is done, prototypes for all the functions that `main()` calls must be written above[11] it. This pattern has been followed in every program example given so far. There is one major exception to this organizational guideline: When a function is so simple that its entire definition can fit on one line, the function itself often is written at the top of the file in place of a prototype.

When you call a function from one of the C libraries, you use code that is already compiled and ready to link to your own code (see Chapter 5). Header files such as `stdio.h` and `math.h` contain prototypes (not C source code) for the precompiled library functions. When your code module uses a **library** function you `#include` the library header file at the top. This causes the preprocessor to insert all the prototypes for the library functions into your module, making it possible for the compiler to properly check your calls on the library functions. The order of parts, from the top of the source file to the bottom follows. These principles lead to the following layout for the parts in a simple program:

- `#include` commands for header files.
- Constant definitions and type declarations.[12]
- Prototypes (function declarations) and one-line functions.
- `main()`, which contains function calls.
- Function definitions, possibly containing more calls.
- Figure 5.19 illustrates the principles with a complete program and two functions.

Notes on Figure 5.19. Functions, prototypes, and calls.

[11] The prototypes also may be written inside the calling function. However, we wish to discourage this practice.

[12] Type declarations will be discussed in Chapters 11, 12 and 13.

A function's prototype may be given first, then the call, and finally the full definition of the function, like function `f()` here. Alternatively, the function may be fully defined before it is called; for example, function `g()` is defined before `main()`, which calls it.

```
#include <stdio.h>
#include <math.h>
#define MIN   10.5
#define MAX   87.0

double f( double y );         // Prototype for function f, defined below.
double g( double y ) { return( y * y + y ); }  // Definition of function g.

int main( void )
{
    double x, z;

    banner();
    printf( "\n Enter a value of x between %.2f and %.2f: ", MIN, MAX );
    scanf( "%g", &x );
    z = f( x ) ;
    printf( "\n The value of x * exp(x) is: %g \n", z );
    printf( "\n The value of x * x + x  is: %g \n", g( x ) );

    return 0;
}

// ------------------------------------------------------------------
// Definition of function to calculate f = y * e to the power y. -----
double
f( double y )
{
    return y * exp( y ) ;
}
```

Figure 5.19. Functions, prototypes, and calls.

This is a function call graph for the functions, prototypes, and calls of the program in Figure 5.19. Shading around a boxes indicates that the function is in a standard library.

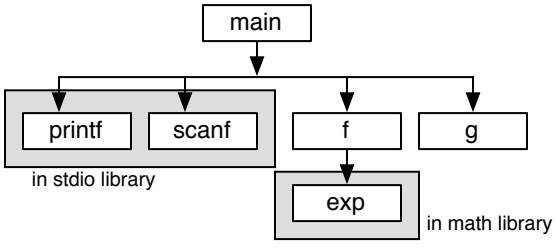


Figure 5.20. Function call graph with two levels of calls.

First box: things that precede `main()` in a code module.

- When the C compiler reaches the `#include` command, it puts a copy of the `tools.h` file into this program. This file contains prototypes for the functions in the `tools` library, including `banner()` and `bye()`, called in this program.
- Included files often contain other `#include` commands. For example, the `tools.h` file contains `#include` commands for the library header files, `stdio.h`, `math.h`, `string.h`, `time.h`, and `ctype.h`. If we include `tools.h` in a program, we need not write separate include commands for these other library header files.
- This program uses two constants, representing the minimum and maximum values acceptable for input. These constants are defined after the `#include` command and before the prototypes.
- We need a prototype for a function if a call on it comes before its definition in the file. Function `f()` is called (second box) from `main()` and defined after `main()` (fourth box). Therefore, `f()` needs a prototype, which is given on the fourth line of this box.
- The actual definition of function `g()` is given here, rather than a prototype. When a function definition comes before any use of that function, no prototype is needed. This often is done when a function is so simple that all its work is done in the `return` statement and so short that it can be written on one line.

Second and third boxes: Calls on the programmer-defined functions.

- We create `f()` and `g()` as two functions separate from `main()`, so that it is easy to change them when we need to do some other calculation. A good modular design keeps the calculation portion of a program separate from the user-interaction portion.
- In the second box, we set `z` equal to the result of calling function `f()` with the value of the variable `x`. Function `f()` was only prototyped before `main()`, so when the compiler reaches this box, the full definition of `f()` will not be known to it. However, the prototype for `f()` already was supplied, so the compiler knows that a call on `f()` should have one `double` argument and return a `double` result. This information is necessary to translate the call properly.
- In the third box, the function `g()` is called, and its return value is then passed directly to `printf()`. Since `g()` already was fully defined, the compiler has full knowledge of `g()` when it reaches this line and, therefore, is able to compile this call correctly.
- A sample output from this program, excluding the banner and closing comment, is

```
Enter a value of x between 10.50 and 87.00: 13.2

The value of x * exp(x) is: 7.13281e+06

The value of x * x + x  is: 187.44
```

Fourth box: Definition of programmer's function `f()`.

- Here we define `f()`. The return type and parameter list in the function definition must agree with the prototype given earlier.
- Function definitions should start with a blank line and a comment describing the action or purpose of the function. Discipline yourself to do this. The dashed line provides a visual separation and helps the programmer find the beginning of the function definition. This is extremely useful in a long program; make it a habit in your work.
- Compare this definition to the previous one-line definition of `g()`. The definition of `f()` begins with a descriptive header. The code itself is spread out vertically, with only one program element on each line, according to the accepted guidelines for good style. The definition of `g()` is written compactly on one line; that style is used only for very simple functions.

Inner box: A call on a library function.

- We call the function `exp()`, which is in the `math` library. We can do this because the header file, `math.h`, was included by `tools.h`, which was included in this file.
- The variable `y` is a `double`. The prototype for `exp()` says that its parameter is a `double`. The type mismatch here is not a problem. The compiler will note the mismatch and automatically compile code to convert the `double` value to a `double` format during the calling process.

This illustrates the function calls in Figure 5.19.

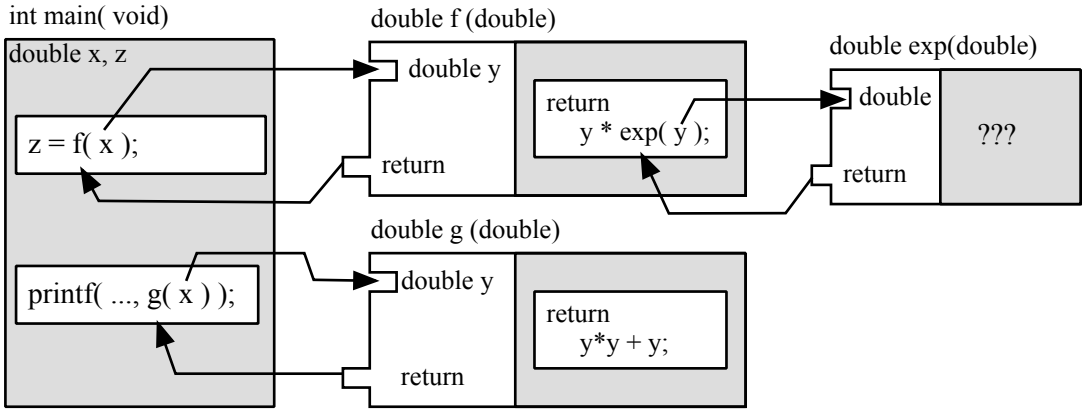


Figure 5.21. A function is a black box.

5.5 Application: Numerical Integration by Summing Rectangles

We introduce the topic of integration with a simple integration program where the function to be integrated is coded as part of the main program. The definite integral of a function can be interpreted as the area under the graph of that function over the interval of integration on the x-axis. If a function is continuous, we can approximate its integral by covering the area under the curve with a series of boxes and adding up the areas of these boxes. Several methods for numerical integration are based on a version of this idea:

- Divide the interval of integration into a series of subintervals.
- For each subinterval, approximate the area under the curve in that interval by a shape such as a rectangle or trapezoid whose area is easy to calculate.
- Calculate and add up the areas of all these shapes.

The simplest way to approximate the integral of a function is to use rectangles to approximate the area under the curve in each subinterval; the diagram in Figure 5.22 and the program in Figure 5.23 illustrate this approach. For each subinterval, we place a rectangle between the curve and the x-axis such that the upper-left corner of the rectangle touches the curve and the bottom of the rectangle lies on the x-axis. In this example, we calculate the integral of the function $f(x)$:

$$f(x) = x^2 + x, \qquad \text{where } 0 \leq x \leq b$$

by summing 100 rectangular areas, each of width $h = b/100$, as shown by Figure 5.22. For example, the area of the rectangle between $2h$ and $3h$ is $h \times f(2h)$. Generalizing this formula, we get

$$\text{Area}_k = h \times f(k \times h)$$

Now, summing over 100 rectangles, we get

$$\text{Area } = \sum_{k=0}^{99} h \times f(k \times h)$$

Notes on Figures 5.22 and 5.23: Integration by Rectangles.

First box: the function definition. This is the entire definition of the function we will integrate. Short functions can be written on one line.

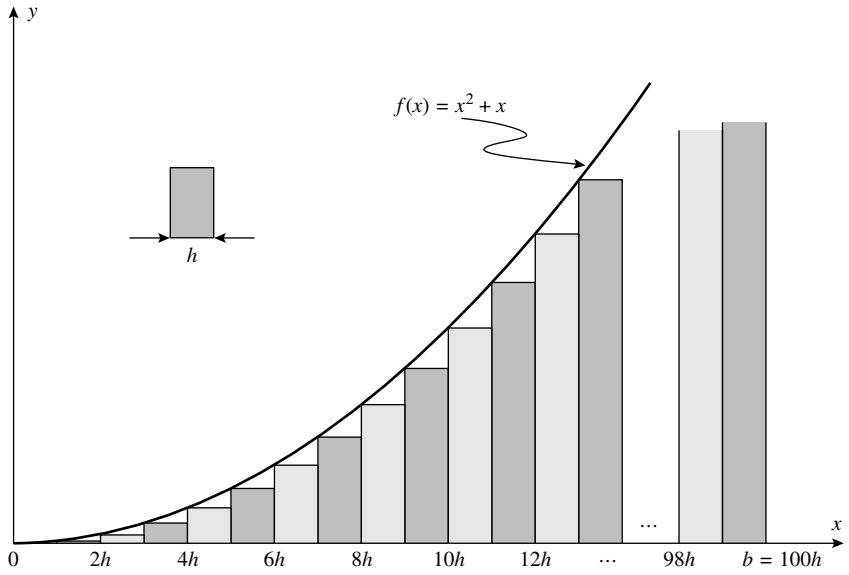


Figure 5.22. The area under a curve.

```
#include <stdio.h>
#define N 100
double f( double x ){ return x*x + x; }

int main( void )
{
    double a = 0.0;       // Lower limit of integration.
    double b = 1.0;       // Upper limit of integration.
    double h = (b - a)/N; // Width of one of the N rectangles summed.

    double x;             // Current function argument; a <= x < b.
    double sum;           // Total area of rectangles.
    int k;                // Loop counter.

    printf( " Integrate x*x + x from %g to %g. \n", a, b );

    sum = 0.0;
    k = 0;

    while(k < N) {            // Add up N rectangles.
        x = a + k * h;        // Lower left corner of kth rectangle.
        sum += h * f( x );    // Area of rectangle at x.
        ++k;
    }

    printf( " The area is %g\n", sum );

    return 0;
}
```

Figure 5.23. Integration by summing rectangles.

Second box: declarations with initializations.

- In this initial example, we integrate a single fixed function over a fixed interval using a fixed number of rectangles. Later, we present other integration programs that do the job in more general and more accurate ways.
- We integrate the function over the interval $0 \ldots 1$.
- We divide this interval into 100 parts of length `h = .01`.

Third box: initializations for the loop.

- Although we could combine these initializations with the declarations, that is bad style. For trouble-free development of complex programs, all loop initializations should be placed immediately before the loop.
- To prepare for the loop, we initialize the `sum` that will accumulate the areas.
- The loop variable for this `while` loop is `k`. The first rectangle lies between a and $a + 1 \times h$, so we initialize `k` to 0. Since $a = 0.0$, the first rectangle we sum starts at $x = 0.0 + 0 \times h = 0.0$.

Fourth box: the loop.

- We execute the loop from `k = 0` until (but not when) `k = 100`. Therefore, we do it 100 times.
- The last rectangle summed starts at $x = a + 99h$ and goes to $x = a + 100h$.
- We use the `++` operator to increment `k` each time around the loop.
- Note how convenient the `+=` operator is for adding a new term to a sum.

Fifth box: the output.

- The *exact* area under this curve for the interval $0 \ldots 1$ is .833333. Since we are calculating an approximate answer, it will be slightly different.
- The actual output is

```
Integrate x*x + x from 0 to 1.
The area is 0.82335
```

5.6 Functions with More Than One Parameter

5.6.1 Function Syntax: Two Parameters

We have studied a variety of `void` and non-`void` functions having either no parameters or one parameter and discussed the essentials of prototypes, parameters, and calls. Most functions, though, have more than one parameter, so we need to study the few remaining facets of the syntax for defining and calling functions with a more complex interface. The next few paragraphs summarize the syntax for functions with two parameters and a return value. The forms for three or more parameters follow the same pattern, with additional clauses added to the parameter list. We will use the following terminology in this discussion:

- `f-name` means any function name;
- `p-type` means the type of a parameter, v`p-name` means the name of a parameter,
- `r-type` means the type returned by the function,
- `var` means the name of a variable,
- `exp` means any expression of the correct type, possibly a simple variable name.

Prototypes. The fundamental form for the 2-argument prototype is:

```
r-type f-name( p1-type p1-name, p2-type p2-name );
```

However, parameter names are optional in a prototype, so we could also write the prototype like this:

```
r-type f-name( p1-type, p2-type );
```

For example, suppose we have a function named `cyl_vol` that takes two double parameters. Its prototype could be written in one of these two ways

This program illustrates the syntax for calling two-parameter non-`void` functions. The two functions called here are defined in Figure 5.25.

```
// ------------------------------------------------------------------------
// Calculate the volume of a cylinder.
*/
#include <stdio.h>
#include <math.h>
#define PI  3.1415927
double cyl_vol( double d, double h );          // long form of prototype
double cyl_surf( double, double );             // short form of prototype

int main( void )
{
    double diam, height;           // Inputs: dimensions of the cylinder.
    double volume;                 // Output: its volume.

    printf( "\n Calculate the Volume of a Cylinder\n\n"
            " Enter its diameter and height: " );
    scanf( "%lg%lg", &diam, &height );

    volume = cyl_vol( diam, height );
    printf( "\t The volume of this cylinder = %g\n", volume );

    printf ( "\t Its surface area = %g \tn", cyl_surf( diam, height ) );

    return 0;
}
```

Figure 5.24. Using functions with two parameters and return values.

```
double cyl_vol( double d, double h );
double cyl_vol( double, double );
```

In mathematical notation, this is called a double×double→double function.

Call syntax. A function call imitates the simpler form of the prototype; argument expressions of the appropriate type must be written, but parameter names are always omitted. The general syntax and a sample call on `cyl_vol` are:

```
General syntax:  var = f-name( exp1, exp2 );
Example:  volume = cyl_vol ( diameter, height );
```

Definition syntax. A function definition imitates the longerform of the prototype; argument types and names must both be given. This is the general syntax and function header of `cyl_vol`:

```
General syntax:  r-type f-name( p1-type p1-name, p2-type p2-name ){...
Example:  double cyl_vol ( double d, double h ){...
```

We illustrate these rules for the format of a two-parameter function using the program in Figures 5.24 and 5.25

Notes on Figures 5.24 and 5.25: Two parameters and a return value.

We illustrate the syntax for definition of two-parameter non-`void` functions. These functions are called from Figure 5.24.

```
// ------------------------------------------------------------------------
double                              // Calculate the volume of a cylinder
cyl_vol( double d, double h ) {     // with diameter=d and height=h;
    double r = d / 2;               // r is the radius of the cylinder.
    return PI * pow( r, 2 ) * h;
}
```

```
// ------------------------------------------------------------------------
double                              // Calculate surface area of cylinder
cyl_surf( double d, double h ) {    // with diameter d and height h;
    double area_end, area_side;
    double r= d / 2;                // r is the radius of the cylinder.
    area_end = PI * pow( r, 2 )     // each end is a circle.
    area_side = h * 2 * PI * r;     // length of side = circle perimiter
    return area_side + 2 * area_end;     // surface is the side + 2 ends
}
```

Figure 5.25. Functions with two parameters and return values.

First box: the prototypes. We will use two double×double→double functions to calculate two properties of a cylinder: volume and surface area. The prototype for the first, `cyl_vol()`, is written with the optional parameter names and the prototype for the second function, `cyl_surf()` is written without. As always, either prototype could be written either way.

Second box: the first function call and output.

- Since `cyl_vol()` returns a value, calls will be in the context of an assignment statement or in the argument list of another function call. This call is part of an assignment.
- First, we call the function and save the answer in `volume`. Then, on the next line, we send the value of `volume` to `printf()`. The calculation could be combined with the output and condensed into one statement by putting the call on `cyl_vol()` directly into the argument list for `printf()`, thus:

```
printf( "\t The volume of this cylinder = %.2f \n",
        cyl_vol( diam, height ) );
```

- It is a matter of personal style which way you write this code. The one-line version is more concise and takes slightly less time and space. The two-line version, however, is easier to modify, works better with an on-line debugger, and enhances seeing and understanding the technical calculation.

Third box: the second function call and output. The calculation and output can be combined and condensed into one statement by putting the call directly into the argument list for `printf()`, as shown here.

Figure 5.25: definitions of `cyl_vol` ***and*** `cyl_surf`***.***

- Following modern guidelines for style, we prefer to write the return type, alone, on the first line of the function definition. The second line starts with the function name on the left. Using this style makes it somewhat easier to find the function names when you scan a long program and allows writing a comment about the return value. Of course, these two lines also could be combined, thus:

```
double cyl_vol( double d, double h ) { ...
```

- The code in the body of this function uses only three variable names: `d`, `h`, and `r`. The first two are parameters to the function, the third is defined at the top of the function's block of code. All three objects

are local to the function; that is, these variables are defined by the function and only this function can access them. Every properly designed function follows this pattern and confines its code to use only locally defined names. All interaction between the function and outside variables happens through the parameter list or the function's return value.

Inner box: raising a number to a power. Very few functions in the `math` library require two arguments; the most commonly used is the function `pow()`, which is used to raise a number to a power. In this box, we are calculating the square of `r`, the radius.

This function's prototype is `double pow( double, double)`. The first argument is the number to be raised, the second argument is the power to which to raise it. Both may be any `double` number. For example, `pow( 100, .5 )` raises 100 to the power .5, which is the same as calculating its square root.

Output.

```
Calculate the Volume of a Cylinder

Enter its diameter and height: 2.0  10.0
The volume of this cylinder = 31.4159
Its surface area = 69.115

Cylinder has exited with status 0.
```

5.7 Application: Generating "Random" Numbers

5.7.1 Pseudo-Random Numbers

Many computer applications (experiments, games, simulations) require the computer to make some sort of random choice. To serve this need, programs called *pseudo-random number generators* have been devised. These start with some arbitrary initial value (or values), called the **seed**, and apply an algorithm to generate another value that seems unrelated to the first. Then this first result will be used as the seed for the next value and so on, as long as the user wishes to keep generating values.

The numbers generated by these algorithms are called **pseudo-random numbers** because they are not really random but the output of an algorithm and an initial value. If the same algorithm is run again with the same starting point, the same series of "random" numbers will be produced. The goal, therefore, is to find an algorithm and a seed that will produce a long series of numbers with no detectable pattern and without duplicating the seed. Repeating a seed would cause the series to enter a cycle.

The C function `rand()` generates pseudo-random integers, which might be 2 or 4 bytes long, depending on the local definition of type `int`. (The actual range of values is 0 ... `RAND_MAX`, which is commonly the same as `INT_MAX`.) This function does not implement the best known algorithm but is good enough for many purposes. The function is found in the `standard` library; to use it you must `#include <stdlib.h>`. Before calling `rand()` the first time, you must call another function, `srand()` to supply an initial seed value. This seed could be any integer value, such as a literal constant or a number entered by the user. However, in general, the user should not be bothered with selecting a seed, and a constant seed is undesirable because it always will result in the same pseudo-random series. (A constant seed can be useful during the debugging process so that error conditions can be repeated.) What therefore is needed for most applications is a handy source of numbers that are constantly changing and nonrepetitive. One such source is attached to most computers: the real-time clock. Therefore, it is quite common to read the clock to get an initial random seed. This technique is illustrated in Figure 5.26.

5.7.2 How Good Is the Standard Random Number Generator?

The next program generates a large quantity of random integers and counts the occurrences of 0. According to probability theory, if we generate numbers in the range $0 \ldots n-1$, approximately $1/n$ of the values should be counted. No single program run can confirm whether the generator is fair. However, repeated trials or larger sample sizes will give some feeling for the quality of the random number generator being used. If the results are close to the expected value most of the time, the generator is performing well; otherwise, its behavior is questionable.

Notes on Figure 5.26. Generating random numbers.

First box: initializing the random number generator.

1. C provides a function in the `time` library that permits a program to read the system's real-time clock (if there is one). The return value of `time()` is an integer encoding of the time that has type `time_t` (an integer of some system-dependent length defined in `time.h`).
2. The argument in the call to `time()` normally is the address of a variable where we want the time stored. The function `time()` stores the current time into the given address in the same way that `scanf()` stores an input value into a variable whose address is given to it.[13] However, this function also returns the same time value through the normal function return mechanism. Since we only need this information once, we use a special constant value, `NULL`, as the argument; this is legal and tells the function that we don't want a second copy of the information stored anywhere in memory.
3. Our purpose here is not to know the actual time. Rather, we use the clock as a convenient source of a seed for the random-number generator. A good seed is an unpredictable number that never is the same twice, and the time of day suits this purpose very well.
4. The type cast operator, `(unsigned)`, in front of the call on the `time()` function is used to convert the `time_t` value returned by the `time()` function into the `unsigned int` form expected by `srand()`. Using an explicit cast instead of the standard automatic coercion eliminates a compiler warning message on some systems.

Second box: data input and validation. We eliminate divisors less than 2 because they are meaningless. We also set an arbitrary upper limit on the range of numbers that will be generated. Since the limit is relatively small, we can use a short integer to store it and there is space for printing many columns of numbers. Here is an example of the error handling:

```
Please choose n between 2 and 100: 1
Number is out of range.

Error exit; press '.' and 'Enter' to continue
```

A validation loop could be used here. We take the simpler approach of using `fatal()` because little effort has been invested so far in running this program and little is wasted by restarting it after an error.

Large outer box: generating and testing the numbers. This loop calls `rand()` many times and collects some information about the results. With the constant definitions given, we will generate 500 integers in the range $0 \ldots n-1$, where $n <= 100$. These numbers will be printed in 10 columns. Occurrences of 0 will be counted.

First inner box: generating the numbers.

1. The function `rand()` returns a number between 0 and `RAND_MAX`. According to the standard, this number may vary, but it is at least 32,767. We must scale this number to the desired range `0..n-1`.
2. The modulus operator is exactly what we want for a scaling operation, since its result is between 0 and the modulus -1. We compute `num % select` and store the result back in `num`.

Second inner box: lines of output.

1. The counter `n` keeps track of the total number of random numbers produced so far.
2. We want the output printed in columns, so we use a fixed field width in the conversion specifier: `%7li`. Ten columns, each seven characters wide, will fit conveniently onto the usual 80-column line.
3. We want to print a `'\n'` after every group of `NCOL` numbers but not after every number. To do this, we count the output items as they are produced and print a newline character every time the counter is an even multiple of `NCOL`; that is, `n % NCOL == 0`.

[13] How this actually is done, using call by address, is discussed in Chapter 11.

We generate a series of pseudo-random numbers and print them in neat columns. When finished, we also print the number of zeros generated and the number expected, based on probability theory.

```
#include "mytools.h"
#define HOW_MANY 500 // Generate HOW_MANY random numbers
#define NCOL 10      // Number of columns in which to print the output.
#define MAX 100      // Upper limit on size of random numbers generated

int main( void )
{
    long num;        // a randomly generated integer
    short select;    // input:  upper limit on range of random numbers
    short n;         // # of random numbers generated
    int count;       // # of zeros generated

    banner();

    srand( (unsigned) time( NULL ) );// seed random number generator.

    printf( " Generate %i random numbers in the range 0..n-1. \n"
            " Please choose n between 2 and %i: ", HOW_MANY, MAX );
    scanf( "%hi", &select );
    if (select < 2 || select > MAX) fatal( " Number is out of range." );

    // Generate random numbers and test for zeros. ----------------------

    count = 0;                      // Count zeros generated.
    for (n = 0; n < HOW_MANY; ) {   // Generate HOW_MANY random numbers.

        num = rand();               // Generate a random long integer.
        num %= select;              // Scale to range 0..select-1.

        ++n;                        // Count the trials and...
        printf( "%5li", num );      // ...print all numbers generated.
        if (n % NCOL == 0) puts( "" ); // End line every NCOL outputs.

        if (num == 0) ++count;      // ..count the zeros.
    }

    if (count % NCOL != 0) printf( "\n" );  // End last line of output.
    printf( "\n %5i   zeros were generated.", count );
    printf( "\n %7.1f are expected on average.\n", HOW_MANY/(double)select );
    return 0;
}
```

Figure 5.26. Generating random numbers.

Third inner box: counting the zeros. To assess the "fairness" of the random-number generator, we can count the number of times a particular result shows up. If numbers in the range $0 \ldots n-1$ are being generated, then each individual number should occur `HOW_MANY / n` times. In this program, we expect each number in the possible range $1 \ldots$ `select`-1 to occur approximately 500/`select` times. The following are the first and last lines of output from three runs. Note that the number of zeros generated on two of three trials differs substantially from the number expected. This is an indication that the `rand()` function does not produce a very even distribution of numbers on our computer.

```
Generate 500 random numbers in the range 0..n-1.
Please choose n between 2 and 100: 25
    7   19   24    8   18   20    8   10    4    5
    5   18   14   19    5   11   24   17   11    5
   ...
    2    0    5   21   22    3   21    7   23   18

    18   zeros were generated.
    20.0 are expected on average.
-------------------------------------------------
Please choose n between 2 and 100: 33
    ...
    12   zeros were generated.
    15.2 are expected on average.
-------------------------------------------------
Please choose n between 2 and 100: 33
    ...
    20   zeros were generated.
    15.2 are expected on average.
```

5.8 Application: A Guessing Game

In a classic game, one player thinks of a number and a second player is given a limited number of tries to guess it. The first player must say whether the guess is too small, correct, or too large. We illustrated a simple example of this game in Figure 6.30. Now, we implement the full game in the next program example, with the computer taking the part of the first player.

5.8.1 Strategy

Even if a person has never seen this game, it does not take long to figure out an optimal strategy for the second player:

- Keep track of the smallest and largest remaining possible value.
- On each trial, guess the number midway between them.

The computer's response to each guess will eliminate half the remaining values, allowing the human player to close relentlessly in on the hidden number.

Figure 5.27 illustrates a game in which the range is $1 \ldots 1{,}000$ and the hidden number is 458. The player makes an optimal sequence of guesses, halving the range of remaining values each time: 500, 250, 375, 437, 468, 453, 461, 457, 459, 458. In this example, 10 guesses are required to home in on the hidden number. In fact, with this strategy, this also is the maximum number of trials required to find any number in the given range. Half the time the player will be lucky and it will take fewer guesses. The code for the program that implements this game is given in Figures 5.28 (`main()`) and 5.29 (`one_game()`, which handles the sequence of guesses).

5.8.2 Playing the Game

Choosing and scaling the number. We can call `rand()`, as was done in Figure 5.26, to get a number in the range `0..INT_MAX`. In this game, however, we require a random number between 1 and 1,000, so the number returned by `rand()` must be scaled and adjusted to fall within the desired range. To do this, we use the `%` (modulus) operator. For instance, `rand() % TOP` gives us a random number in the range (`0..TOP-1`). We then adjust it to the desired range simply by adding 1. This formula is used in the first box of Figure 5.29.[14]

[14] For reasons too complex to explain here, this formula has a slight bias toward lower numbers in the range. However, if the range is small compared to `RAND_MAX`, the bias is insignificant.

In this example, the total range of possible values is 1 . . . 1,000. The hidden number, $num = 458$, is represented by a dashed line. The solid vertical lines represent the guesses of a player using an optimal strategy; only the first five guesses are shown.

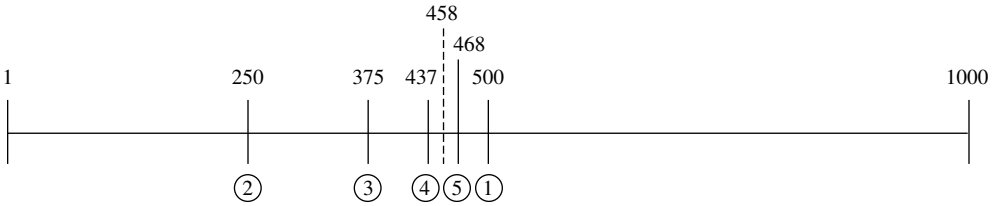


Figure 5.27. Halving the range.

Setting a limit on the number of guesses. The strategy shown in Figure 5.27 is an example of an important algorithm called **binary search**,[15] because we search for a target value by dividing the remaining set of values in half. If N values were possible in the beginning, the second player always could discover the number in T trials or fewer, where $N \leq 2^T$. Another way to say this is that

$$T = \lceil \log_2 N \rceil$$

where $\lceil \ldots \rceil$ means that we round to the next higher integer. Paraphrased, the maximum number of trials required will be the base-2 logarithm of the number of possibilities, rounded up to the next larger integer. In our case, this is

$$\lceil \log_2 1000 \rceil = \lceil 9.96578 \rceil = 10$$

To introduce an element of luck into the game, set the maximum number of guesses to something smaller than T. This, in fact, is what we do in Figure 5.28; we round *down* instead of *up*, allowing too few guesses about half of the time. This "stacks" the game in favor of the computer.

Calculating a base-2 logarithm. The C math library provides two logarithm functions: one calculates the base-10 log, the other the natural log (base e). Neither of these is what we want, but you can use the natural log function to compute the log of any other base, B, by the following formula:

$$\log_B(x) = \frac{\log_e(x)}{\log_e(B)}$$

In C, the natural log function is named `log()`, so to calculate the base-2 log of 1,000, we write

```
log( 1000 ) / log( 2 )
```

This formula is used in the second box of Figure 5.28.

Notes on Figure 5.28. Can you guess my number?

First box: guessing range.

In this game, the player will try to guess a number between 1 and `TOP`.

Second box: the number of trials.

- We calculate `tries`, the maximum number of trials the user is allowed. It is defined as a **constant** because it depends only on `TOP` and does not change from one game to another. We use a `const` variable, rather than `#define`, because the definition is not just a simple number.[16]
- The result of the division operation is a `double` value that is coerced to type `int` when stored in `tries`.
- C permits us to use a formula to define the value of a constant. Such formulas can use literal constants (such as 2) and globally defined symbols (such as `TOP`). Inside a function definition, the parameter values also can be used in a constant expression.

[15] Other examples of binary search are given in Chapters 11 and 19.

[16] For reasons beyond the scope of this book, this is more efficient.

This main program calls the function in Figure 5.29. It repeats the game as many times as the player wishes.

```
#include "mytools.h"
void one_game( int tries );

#define TOP  1000            // Top of guessing range.

int main( void )
{
    int do_it_again;                              // repeat-or-stop switch

    const int tries = log( TOP ) / log( 2 );    // One too few.

    banner();
    puts( "\n This is a guessing game.\n I will "
          "think of a number and you must guess it.\n" );

    srand( (unsigned)time( NULL ) );      // seed number generator.

    do {  one_game( tries );
          printf( "\n\n Enter 1 to continue, 0 to quit: " );
          scanf( "%i", &do_it_again );
    } while (do_it_again != 0);

    return 0;
}
```

Figure 5.28. Can you guess my number?

- We calculate the base-2 logarithm of the number of possible hidden values and use this to set the maximum number of guesses. As described, we set this maximum so that the player will succeed about half the time. The rest of the time, the player will be one guess short of success, even if he or she is using the optimal search strategy.

Third box: initializing the random number generator. As in Figure 5.26, we initialize C's random number generator with the current time of day.

Notes on Figure 5.29. Guessing a number. The `one_game()` function is called from `main()` in Figure 5.28 for each round of the game that the player wishes to play.

First box: the hidden number. The first thing this function does is choose a hidden value. The value is defined as a constant because it does not change from the beginning of a round to the end of that round. In this case, the constant expression involves the value of a global constant. To calculate a random hidden number, we use the random number generator `rand()`, scaling and adjusting its value to the required range, as discussed earlier in this section.

Second outer box: playing the game. The code in this box is almost identical to the earlier version in Figure 6.30. Correct guesses are handled by an `if...break` in the inner box. When the loop exits, the program prints a failure message if the most recent guess was wrong.

Here is some sample output (omitting output of the query loop). The first two lines were printed by the main program, the rest of the dialog was printed by the `one_game()` function. In this sample game, the player was lucky and guessed the hidden number in only five tries. We expect this to happen only once in every 32 games.

This function is called from Figure 5.28. It plays the number-guessing game once.

```
void
one_game( int tries )
{
    int k;                                          // Loop counter.
    int guess;                                      // User's input.

    const int num = 1 + rand() % TOP;               // The hidden value.

    printf( " My number is between 1 and %i;"
            " I will let you guess %i times.\n"
            " Please enter a guess at each prompt.\n", TOP, tries );

    for (k = 1; k <= tries; ++k) {
        printf( "\n Try %i: ", k );
        scanf( "%i", &guess );
        if (guess == num) break;
        if (guess > num) printf( " No, that is too high.\n" );
        else printf( " No, that is too low.\n" );
    }
    if (guess == num) printf( " YES!!  That is just right.  You win!  \n" );
    else printf( " Too bad --- I win again!\n" );
}
```

Figure 5.29. Guessing a number.

```
This is a guessing game. I will think of a number and you must guess it.

My number is between 1 and 1000; I will let you guess 9 times.
Please enter a guess at each prompt.

Try 1: 500
No, that is too high.

Try 2: 250
No, that is too high.

Try 3: 125
No, that is too high.

Try 4: 62
No, that is too low.

Try 5: 93
YES!!  That is just right.  You win!
```

5.9 What You Should Remember

5.9.1 Major Concepts

- Functions can be used to break up programs into modules of manageable complexity. In this way, a highly complex job can be broken into short units that interact with each other in controlled ways. These modules should have the following properties:
  - The purpose of each function should be clear and simple.
  - All its actions should hang together and work at the same level of detail.
  - A function should be short enough to comprehend in its entirety.
  - The length and complexity of a function can be minimized by calling other functions to do subtasks.
- Function definitions can come from the standard C libraries such as the `stdio` and `math` libraries or from a personal library. Functions also can be defined by the programmer.
- Some functions compute and return values; others do not. A function with no return value is called a `void` function. Such functions normally perform input or output of some sort. This chapter presented programmer-defined functions with zero, one, and two parameters (void→void, double→double, and double×double→double respectively). Additional types of library functions and programmer-defined functions are introduced in subsequent chapters as additional data types are discussed.
- A call to a function that returns a value normally is found in an assignment statement, an expression, or an output statement.
- The basic components of a function are the prototype and the function definition, which consists of a header and a body.
- Arguments are the means by which a calling program communicates data to a function. Parameters are the means by which a function receives the communicated data. When a function is called, the actual arguments in the function call are passed into the function and become the values of the function's formal parameters.
- After passing the parameter values, control is passed into the function. Computation starts at the top and proceeds through the function until it is finished, at which time the result is returned and control is transferred back to the calling program. A more detailed look at functions will be given in Chapter 9.
- The name of the parameter in a function does *not* need to be the same as the name of a variable used in calling that function.
- Functions allow an application program to be constructed and verified much more easily than a similar program written as one massive unit. Each function, then each module, is developed and debugged before it is inserted into the final, large program. This is how professional programmers have been able to develop the large, sophisticated systems that we use today.
- Here is a (questionable) rhyme to help you remember the four ways to extract an integer from a `double` value:

  `ceil()` goes up and `floor()` goes down; `rint()` goes nearby but assign doesn't round.

- Random numbers. Applications such as games, experiments, and quiz programs require a program to make a series of randomized selections from a preset list of numbered options. To do this, we use an algorithm called a *pseudo-random number generator*, which generates a series of integers with no apparent pattern. The random number then is scaled to be in the proper range if it does not fall within the range of selection numbers.

  The `standard` library provides the functions `srand()` and `rand()`, which together implement a pseudo-random number generator.

5.9.2 Local Libraries and Header Files

At various places in this chapter, suggestions were made about building a personal library called "mytools". In this section, we gather together the various parts that could be included in that library and show how they would be organized into a header file `mytools.h` and a code file `mytools.c`, and how a client program would use such a library.

The header file for mytools.

```
// ------------------------------------------------------------------------
// Alice Fischer's personal tools library.                  File:  mytools.h
// Last updated on July 29, 2011.
// Standard library headers that I need. --------------------------------
#include <stdio.h>      // for puts(), printf(), and scanf()
#include <stdlib.h>     // for exit()
#include <time.h>       // for time_t, time() and ctime()
#include <stdbool.h>    // for bool data type; c99 standard

// defined on some systems but not on mine.
#define PI  3.14159265358979323846264338327951

// Prototypes for my own tool functions. --------------------------------
void banner( void );               // Print a neat header for the output.
```

The code file for mytools.

```
// ------------------------------------------------------------------------
// Alice Fischer's personal tools library.                    File:  mytools.c
// Last updated on Tue Sep 16 2003.

#include "mytools.h"
// ------------------------------------------------------------------------
void banner( void )               // Print a neat header for the output.
{
    time_t now = time(NULL);
    printf( "\n-----------------------------------------------------\n" );
    printf( "    Alice E. Fischer\n    CS 110\n    ");
    printf( ctime( &now ) );
    printf(  "-----------------------------------------------------\n" );
}
```

A program that uses mytools.

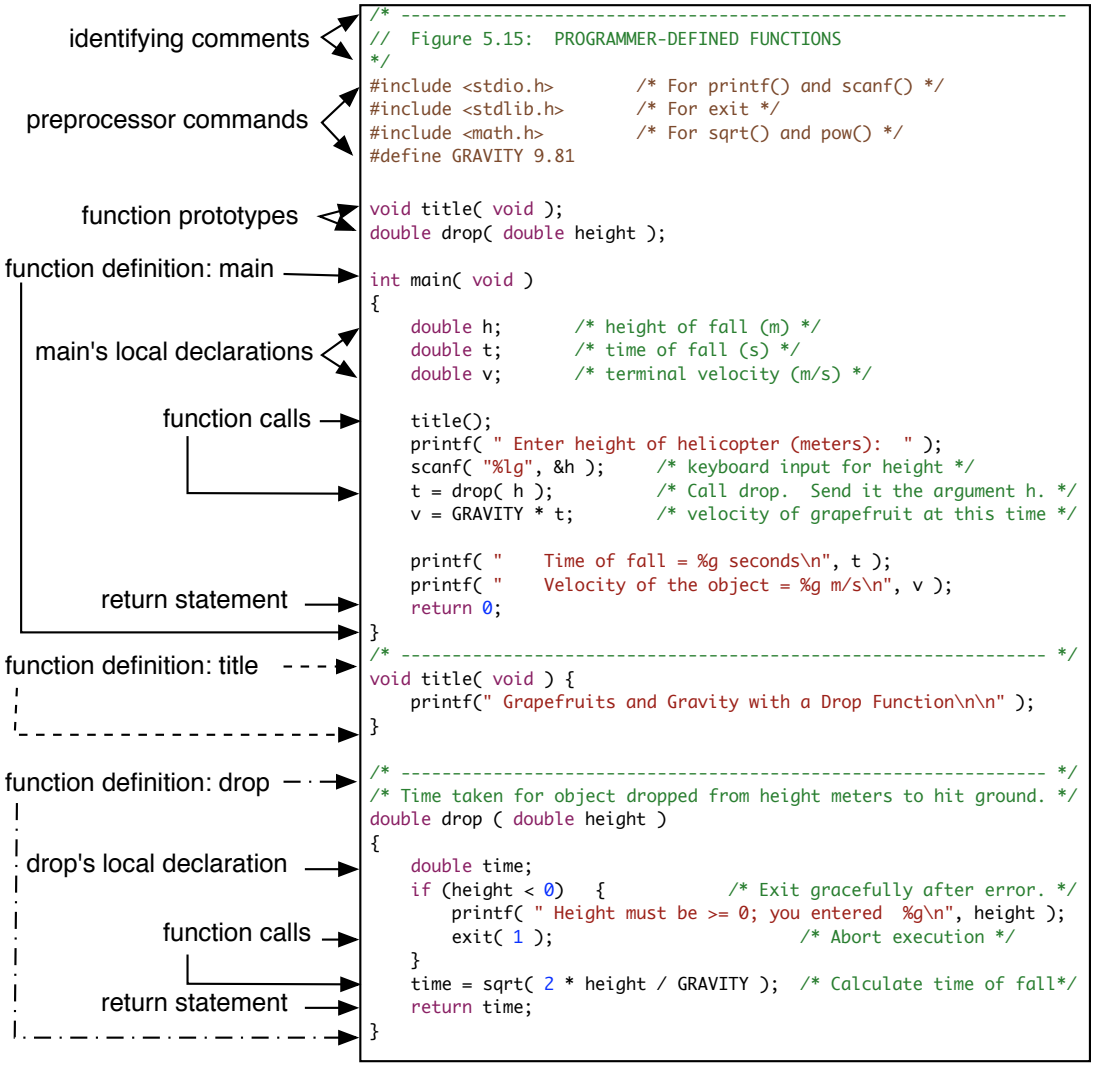
```
// ------------------------------------------------------------------------
// Demo program for using a personal library.  Both files mytools.h and
// mytools.c must be added to the project for this program.
// ------------------------------------------------------------------------

#include "mytools.h"
int main( void )
{
    int count; // Number of people in the room.
    banner();
    printf( " Demonstrating the use of a personal library.\n"
            " How many people are watching?  " );
    scanf( "%i", &count );
    if (count < 2)  puts( " Not enough.\n" );
    else puts( " Hello you-all!\n" );
    return EXIT_SUCCESS;
}
```

Sample output from this demo.

```
 -----------------------------------------------------
    Alice E. Fischer
    CS 110
    Tue Sep 16 12:26:45 2003
 -----------------------------------------------------
 Demonstrating the use of a personal library.
 How many people are watching?  3
 Hello you-all!
```

5.9.3 The Order of the Parts of a Program

The following diagram summarizes the order in which it is customary to write prototypes, function definitions, and other elements in a source code file. The prototypes are normally written first, followed by the `main()` function then all other functions, in any convenient order.




```c
/* ---------------------------------------------------------------
//  Figure 5.15:  PROGRAMMER-DEFINED FUNCTIONS
*/
#include <stdio.h>        /* For printf() and scanf() */
#include <stdlib.h>       /* For exit */
#include <math.h>         /* For sqrt() and pow() */
#define GRAVITY 9.81

void title( void );
double drop( double height );

int main( void )
{
    double h;       /* height of fall (m) */
    double t;       /* time of fall (s) */
    double v;       /* terminal velocity (m/s) */

    title();
    printf( " Enter height of helicopter (meters):  " );
    scanf( "%lg", &h );     /* keyboard input for height */
    t = drop( h );          /* Call drop.  Send it the argument h. */
    v = GRAVITY * t;        /* velocity of grapefruit at this time */

    printf( "    Time of fall = %g seconds\n", t );
    printf( "    Velocity of the object = %g m/s\n", v );
    return 0;
}
/* ---------------------------------------------------------------- */
void title( void ) {
    printf(" Grapefruits and Gravity with a Drop Function\n\n" );
}

/* ---------------------------------------------------------------- */
/* Time taken for object dropped from height meters to hit ground. */
double drop ( double height )
{
    double time;
    if (height < 0)   {            /* Exit gracefully after error. */
        printf( " Height must be >= 0; you entered  %g\n", height );
        exit( 1 );                        /* Abort execution */
    }
    time = sqrt( 2 * height / GRAVITY );  /* Calculate time of fall*/
    return time;
}
```

5.9.4 Programming Style

- Keep the main program and all function definitions short. An entire function should fit on one video screen, so that the programmer can see the declarations and the code at the same time. When a function begins to get long, this often is a sign that it should be broken into two or more functions.
- Every function definition should start with a distinctive and highly visible comment. We use a dashed line followed by a brief description of the purpose of the function. This comment block helps you to find each portion of your program quickly and easily, both on screen and on paper.
- Function names should be descriptive, distinctive, and not overly long.
- All the object names used in a function should be either parameter names or defined locally as variables. This provides maximum isolation of each function from all others, which substantially aids debugging.

5.9.5 Sticky Points and Common Errors

- The prototype must match the function header. If there is a mismatch, the program will not compile correctly.
- Prototypes end in semicolons. If the semicolon is missing, the compiler "thinks" that the prototype is the function header. This will cause many meaningless error comments.
- Definitions must *not* have a semicolon after the function header. If a semicolon is written there, the compiler "thinks" the header is a prototype and will give an error comment on the next line.
- If the prototype is missing or comes after the first function call, the compiler will construct a prototype using the types of the parameters given in the first function call. The return type always will be `int`, which may or may not be correct. If the constructed prototype is wrong, it will cause the compiler to give error comments on lines that are correct.

5.9.6 Where to Find More Information

- Chapter 9 presents the full process of top-down programming and stub testing. This discussion has been deferred until the basic mechanics of functions and function calls are better understood.
- Chapter 9 has a more complete discussion of functions and prototypes.
- Chapter 9 explains local and non-local variables and how they are implemented using activation records on the run-time stack. Figure 9.2 shows how storage might be managed at run time for the cylinder program in Figure 5.24.
- Functions and type definitions to help programmers use the clock are described in Appendix **??** and parts are discussed in detail in Chapter 12.
- Chapter 12 shows a revised version of the `banner()` function.
- The website for this chapter introduces a function named `fatal()` that combines the actions of `printf()` and `exit()`, allowing error handling to be done in one statement.
- The results of a subprogram must be passed back to the caller. Depending on the function's purpose, there may be no, one, or more results. but only one result can be returned through a `return` statement. There are several ways around this difficulty:
  - Chapter 10 discusses array parameters, which can be sent, empty, into a function, then filled in the function and returned containing a potentially large amount of data.
  - Chapter 11 shows how pointer arguments may be used to return information from a function to the caller.
  - Chapter 13 discusses compound objects (structures) that can contain and return many pieces of information packed into a single object.

5.9.7 New and Revisited Vocabulary

These are the most important terms and concepts that were introduced or discussed in this chapter:

function	function comment block	call graph
standard library	function body	pseudo-random numbers
personal library	function return	seed
header file	programmer-defined function	binary search
function call	prototype declaration	constant expression
caller	function definition	void→int function
argument	function interface	void→void function
subprogram	function header	int→void function
transfer of control	parameter	double→double function
interrupted flow	local variable	double×double→double function
exception		

The following C keywords, header files, library functions, constants, and types were discussed in this chapter:

`return` statement	`<stdlib.h>`	`time_t` (from `time.h`)
`<stdio.h>`	`EXIT_FAILURE` (from `stdlib.h`)	`time(NULL)` (from `time.h`)
`scanf()` (from `stdio`)	`EXIT_SUCCESS` (from `stdlib.h`)	`ctime()` (from `time.h`)
`printf()` (from `stdio`)	`exit()` (from `stdlib`)	`<math.h>`
`puts()` (from `stdio`)	`abs()` (from `stdlib`)	`sqrt()` (from `math`)
`<limits.h>`	`srand()`(from `stdlib`)	`floor()` (from `math`)
`INT_MAX` (from `limits.h`)	`rand()`(from `stdlib`)	`log()` (from `math`)
`RAND_MAX` (from `stdlib.h`	`<time.h>`	`pow()` (from `math`)

5.10 Exercises

5.10.1 Self-Test Exercises

1. For each part, write a double→double function that computes the formula and returns the result:
   (a) Tangent of angle $x = \dfrac{\sin(x)}{\cos(x)}$
   (b) Surface area of a sphere with radius $r = 4 \times \pi \times r^2$
2. For each part, write a double×double→double function that computes the formula and returns the result:
   (a) Hypotenuse of a right triangle: length $= \sqrt{\text{base}^2 + \text{height}^2}$
   (b) Polar to rectangular coordinates: $y = r \times \cos(\text{theta})$
   (c) Rectangular to polar coordinates: theta = arc tangent(y/x) for $x \neq 0$
3. Write the prototype that corresponds to each function definition:
   (a) `double cube( double x ) { return x*x*x; }`
   (b) `void three_beeps() { beep(); beep(); beep(); }`
   (c) `int surprise( void ) { return 17; }`
   (d) `int ratio( double a, double b) { return a/b; }`
4. Explain the difference between
   (a) An argument and a parameter
   (b) A prototype and a function header
   (c) A header file and a source code file
   (d) A personal library and a standard library
   (e) A function declaration and a function call
   (f) A function declaration and a function definition
5. What happens on your compiler when a prototype for a programmer-defined function is omitted? To find out, start with the code from Figures 5.15 and 5.18 and delete the prototype above the main program.
6. Prototypes and calls. Given the prototypes and declarations that follow, fix the errors in each of the lettered function calls.

```
void   squawk( void );
int    half( int );
double area( double, double);

int j, k;
double x, y, z;
```

   (a) `k = squawk();`
   (b) `squawk( 3 );`
   (c) `j = half( 5, k );`

(d) `j = half( int k );`
(e) `y = area( double 3.0, x );`

7. Find the error here. Using the declarations given in problem 6, find the one function call below that has an error and explain what the error is. (The other three are correct.)

   (a) `y = area( x+2, 3 );`
   (b) `y = half( half( k ) );`
   (c) `printf( "%g %g", x, half( x ) );`
   (d) `y = area( sin( x ), z );`

5.10.2 Using Pencil and Paper

1. List everything you must write in your program when you want to *use* (not define) each of the following:

   (a) A programmer-defined void→void function named `help()`
   (b) The `fatal()` function
   (c) The `sqrt()` function

2. Write a prototype that could correspond to each function call below:

   (a) `do_it();`
   (b) `x = calculate( y, z);`
   (c) `k = get_Integer();`
   (d) `x = cubeRoot( y );`

3. For each part, write a double→double function that computes the formula and returns the result:

   (a) Sine: $\sin(x) = \sqrt{1 - \cos^2(x)}$
   (b) Tangent: $\tan(2x) = \dfrac{2\tan(x)}{1 - \tan^2(x)}$
   (c) Diagonal of a square with side $s = \sqrt{2 \times s^2}$
   (d) Volume of a sphere with radius $r = \dfrac{4\pi}{3} \times r^3$

4. Write a double×double→double function that computes the formula and returns the result:

   Sum of angles: $\sin(x + y) = \sin(x)\cos(y) + \cos(x)\sin(y)$

5. What happens on your compiler when a required `#include` command is omitted? Will a program compile? Will it work correctly?

6. Prototypes and calls. Given the prototypes and declarations that follow, say whether each of the lettered function calls is legal and makes sense. If the call has an error, fix it.

```
void   squawk( void );
int    half( int );
double area( double, double);

int j, k;
double x, y, z;
```

   (a) `j = squawk();`
   (b) `half( k );`
   (c) `j = half( 5 * k );`
   (d) `y = area( double 3.0, double x );`
   (e) `y = area( z );`

(f) `y = area( pow( x, 2 ), z );`
(g) `scanf( "%i %i", &k, &half( k ) );`
(h) `y = area( x, y, z );`

7. Write the prototype that corresponds to each function definition. Then define appropriate variables and write a legal call on the function.

(a)
```
double inches( double cm ) { return cm / 2.54; }
```
(b)
```
void beeps( int n )
{   int k = 0;
     while (k < n) {
         beep();
         ++k;
     }
 }
```

5.10.3 Using the Computer

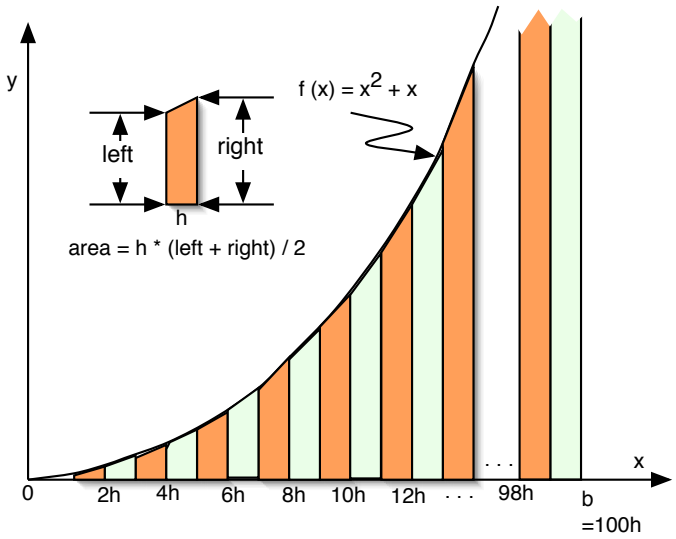
1. Geometric mean.

A *geometric progression* is a series such as 1, 2, 4, 8, ... or 1, 5, 25, 125, ... such that each number in the series is multiplied by a constant to get the next number. More formally, the constant, R, is called the **common ratio**. If the first term in the series is a, then succeeding terms will be $a * R$, $a * R^2$, $a * R^3$, ... For instance, if $a = 1$ and $R = 2$, we get 1, 2, 4, 8, ... and if $a = 1$ and $R = 5$, we get 1, 5, 25, 125, ...

Given terms $k - 1$ and $k + 1$ in a geometric progression, write a main program and at least one function to compute term k and the common ratio, R. Term k is called the *geometric mean* between the other two terms. The formulas for the kth term and common ratio are

$$R = \sqrt{\frac{t_{k+1}}{t_{k-1}}} \quad \text{and} \quad t_k = R \times t_{k-1}$$

2. Integration using trapezoids.



Modify the program in figure 5.23 to use trapezoids, rather than rectangles, to approximate the area under the curve, according to this sketch of the trapezoid method: Compare your answers to those from Figure 5.23. What can you say about their accuracy?

3. Numerical integration.

   The program in Figure 5.23 integrates the function $f(x) = x^2 + x$ for $0 \leq x \leq 1.0$. Modify this program to integrate the function $f(x) = x^2 + 2x + 2$ for $-1.0 \leq x \leq 1.0$. If you have studied symbolic integration, compare your result to the exact analytical answer.

4. Precision of numerical computations.

   Figure 5.23 gives a program that integrates a function by rectangles. Keep the function `f()` and modify the main program so that the integration process will be repeated using 10, 20, 40, 80, 160, and 320 rectangles. Print a neat table of the number of rectangles used and the results computed each time. Compare the answers. What can you say about their accuracy?

5. Sales tax.

   Write a double→double function whose parameter is a purchase price. Calculate and return T, the total price, including sales tax. Define the sales tax rate as a `const` R whose value is 6**%**. Write a main program that will read P, the before-tax amount of a purchase, call your function to calculate the after-tax price, and print out the answer. Both prices will be in units of dollars and cents. What are the appropriate types for P, R, and T? Why? Use the program in Figure 5.15 as a guide.

6. Fence me in.
   A farmer has several rectangular fields to fence. The fences will be made of three strands of barbed wire, with fence posts no more than 6 feet apart and a stronger post on every corner. Write a complete specification, with diagram, for a program that will input the length and width of a field, in feet. Make sure that each input is within a meaningful range. If so, calculate the area of the field and the total length of barbed wire required to go around the field. Also calculate the number of fence posts needed (use the `ceil()` function from the `math` library). Now write a program that will perform the calculations and display the results.

7. A spherical study.

   Write four functions to compute the following properties of a sphere, given a diameter, d, which is greater than or equal to 0.0:

   (a) Radius $r = d/2$
   (b) Surface area $= 4 \times \pi \times r^2$
   (c) Circumference $= \pi d$
   (d) Volume $= \frac{4\pi}{3} \times r^3$

   Write a main program that will input the diameter of a sphere, call all four functions, and print out the four results. Do not accept inputs less than 0.0.

8. Take-home pay.

   Write a double→double function whose parameter is an employee's gross pay for one month. Compute and return the take-home pay, given the following constants:

   - Medical plan deduction = $75.65
   - Social security tax rate = 7.51**%**
   - Federal income tax rate = 16.5**%**
   - State income tax rate = 4.5**%**
   - United Fund deduction = $15.00

   The medical deduction must be subtracted from the gross pay before the tax amounts are computed. Then the taxes should be computed and subtracted from the gross. As each one is computed, print the amount. Finally, subtract the United Fund contribution and return the remaining amount. Your main program should print the final pay amount.

9. Hourly Employee.

   Write a double×double→double function whose parameters are an employee's hourly pay rate and number of hours worked per week. Compute and return the gross pay.

   Your main program should call the hourly-pay function, then use its result to call the take-home pay function described in the previous problem. Print the take-home pay, as before.

10. Compound interest.

    (a) Write a function with three double parameters to compute the amount of money, A, that you will have in n years if you invest P dollars now at annual interest rate i. The formula is

    $$A = P(1 + i)^n$$

    (b) Write a main program that will permit the user to enter P, i, and n. Call your function to compute A. Your main program should echo the inputs and print the final dollar amount.

11. Probability.

    A statistician needs to evaluate the probability, p, of the value x occurring in a sample set with a known normal distribution. The mean of the distribution is $\mu = 10.71$ and the standard deviation is $\sigma = 1.14$.

    (a) Write a double→double function with parameter x that computes the value of the probability formula for a normal distribution, which follows. To compute e^x, use the `exp()` function from the math library; its prototype is `double exp( double x )`.

    $$p = \frac{1}{\sigma \times \sqrt{2\pi}} \times e^{-d}, \qquad \text{where } d = \frac{[(x - \mu)/\sigma]^2}{2}$$

    (b) Write a main program to input the value for x, call your probability function, and print the results.

Chapter 6

More Repetition and Decisions

This chapter continues the discussion of control statements, which are used to alter the normal top-to-bottom execution of the statements in a program. As each new kind of statement is covered, its corresponding flow diagram will be shown. We present the syntax, flowcharts, and examples of use for these statements.

There are three kinds of loop statements in C, each of which is used to repeat a block of code. The `while` loop was introduced in Chapter 3. This chapter presents two additional loop statements, `for` and `do...while`. We also discuss various ways that loops can be used.

Conditional control statements are used to determine whether to execute or skip certain blocks of code. The C language has three kinds of conditional control statements: the `if...else` statement and the simple `if` statement without an `else` clause, which are introduced in Chapter 3, and a multibranched conditional statement, the `switch` statement. In this chapter we examine the `switch` statement and a new way to use the simple `if` statement.

We briefly cover the `break` and `continue` statements, which interrupt the normal flow of control by transferring control from inside a block of code to its end or its beginning. C also supports the `goto` control statement. However, we neither explain it nor illustrate its use, because it is almost never needed in C and its unrestricted nature makes it error prone. Using a `goto` statement is considered very poor programming practice because it leads to programs that are hard to both debug and maintain.

6.1 New Loops

6.1.1 The `for` Loop

The `for` loop in C implements the same control pattern as the `while` loop; anything that can be done by a `while` loop can be done in the same way using a `for` loop and vice versa. A `for` loop could be thought of as a shorter, integrated notation for a `while` loop that brings together the initialization(s), the loop test, and the update step in a single header at the top of the loop, as shown in Figure 6.1.

Thus, `for` is not really a necessary part of the language. However, it probably is more widely used than either of the other loops because it is very convenient and it captures the nature of a loop that is controlled by a counter. After gaining some experience, most programmers prefer using the `for` statement for many kinds of loops, especially for **counted loops** like that in Figure 6.1. An extremely simple program containing a `for` loop is shown, with its output, in Figure 6.2. This illustrates how a loop should be laid out in the context of a program.

The syntax and flow diagram for `for`. The `for` statement has a header consisting of the keyword `for`, followed by a parenthesized list of three expressions separated by semicolons: an initialization expression, a condition, and an update expression. All three expressions can be arbitrarily complicated. Usually, however, the first part is a single assignment, the second is a comparison, and the third is an increment expression, as shown in in Figure 6.1.

Since `for` and `while` implement very similar control structures, a `for` loop can be diagrammed in the same manner as a `while` loop: using separate boxes for the initialization, test, and update steps. However, there is

In this simple `while` statement and its corresponding `for` statement, `k` is used as a counter to keep track of the number of loop repetitions. When `k` reaches `10`, the body will have been executed 10 times and the loop will exit. Following the loop, the values of `k` and `sum` will be printed. The output will be `10 45`.

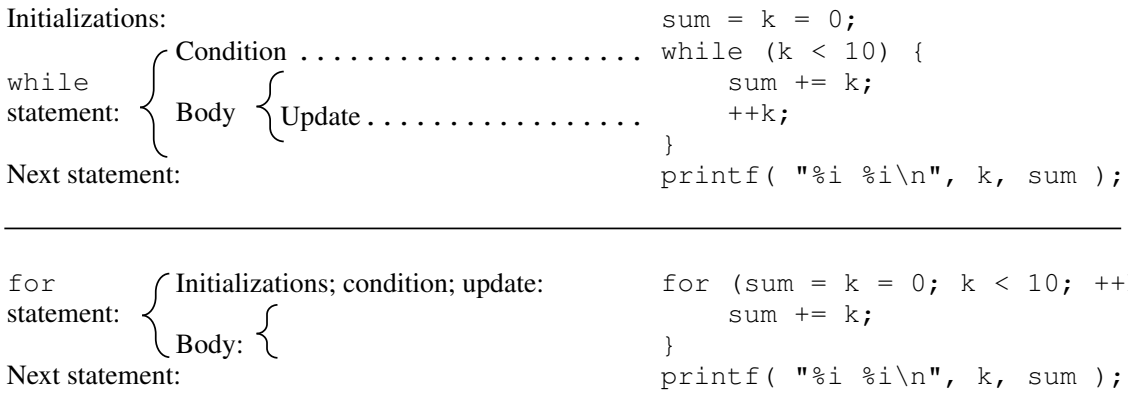


Figure 6.1. The `for` statement is a shorthand form of a `while` statement.

a more compact, single-box diagram that combines these pieces into one multipart control box with a section for each part of the loop header. (See Figure 6.3). Control enters through the initialization section at the top, then goes through the upper diagonal line into the condition section on the right. If the condition is true, control leaves through the lower exit, going through the boxes in the body of the loop, coming back into the update section of the `for` box, and finally, going through the lower diagonal line into the test again. If the test is false, control leaves the loop through the upper exit. The flow diagrams in Figure 6.4 compare the equivalent `while` and `for` loops of Figure 6.1. Both the statement syntax and the diagram for the `for` loop are more concise—fewer parts are needed and those parts are more tightly organized—than in a corresponding `while` loop.

Declaring the for-loop variable. If the loop variable will be used *only* within the loop, it is customary to declare it in the parentheses that follow the keyword `for`, as shown in Figure 6.2. This style keeps the declaration near the use of the variable and is consistent with modern usage in Java and C++.

Initializing the loop variable. An initialization statement must be written before a `while` loop so that the loop variable has some meaningful value before the loop test is performed the first time. In a `for` loop, this initialization is written as the first expression in the loop header. When the `for` loop is executed, this expression will be evaluated only once, before the loop begins.

The `,` (comma operator) can be used in a `for` loop to permit more than one item to be initialized or

```
// ---------------------------------
//   Counting with a for loop.
#include <stdio.h>
int main( void )
{
    for (int k = 0; k < 5; ++k) {
        printf( " %i \n", k );
    }
    puts( "-----\n" );
}
```

The output is

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

Figure 6.2. A simple program with a `for` statement.

The general form of a `for` flow diagram is shown here. Control passes through the loop in the order indicated by the arrows. Note that this order is not the same as the order in which the parts of the loop are written in the code.

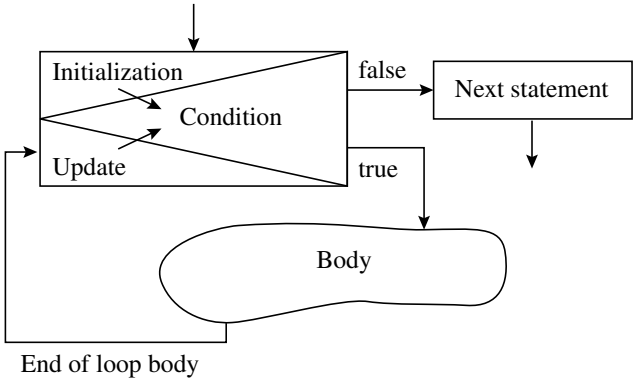


Figure 6.3. A flow diagram for the `for` statement.

updated. For example, the loop in Figure 6.1 could be written with separate initializations for `sum` and `k`:

```
for (sum = 0, k = 0; k < 10; ++k) ...
```

Technically, both assignment and comma are operators. They build expressions, not complete statements, so we are permitted to use either or both any time the language syntax calls for an expression.

The loop test and exit. The condition in a `for` statement obeys the same rules and has the same semantics as that of a `while` condition. It is executed after the initializations, when the loop is first entered. On subsequent trips through the loop, it is executed after the update. If the condition does computations or causes side effects, such as input or output, those effects will happen each time around the loop. (This programming style is not recommended. It may be concise, but it sacrifices clarity and does not work well with some on-line debuggers.)

The flow diagram for the `while` loop in Figure 6.1 is shown on the left and the diagram for the `for` loop is on the right. Note that the parts of the two loops are executed in exactly the same order, as control flows into, through, and out of the boxes along the paths indicated by the arrows.

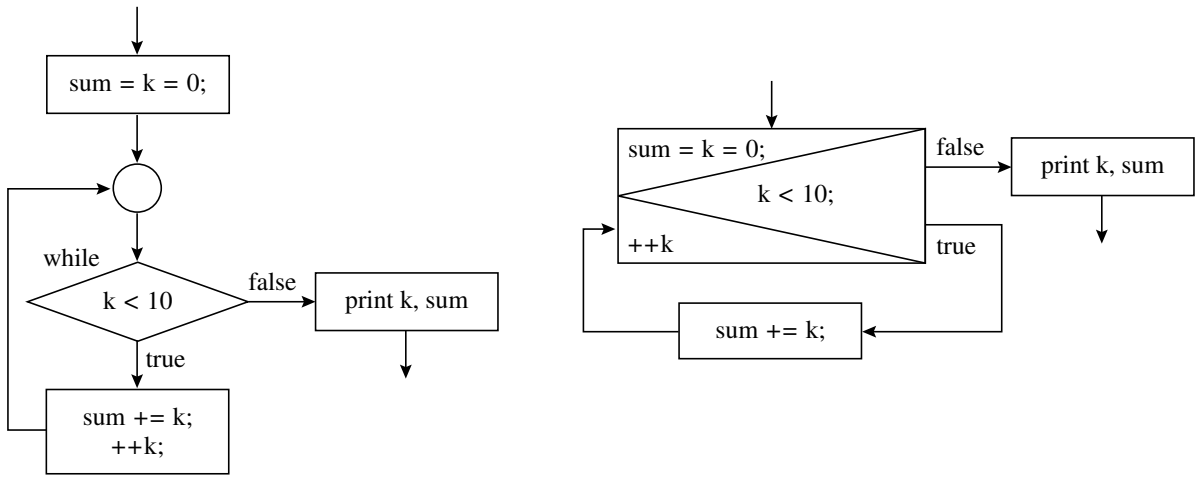


Figure 6.4. Diagrams of corresponding `while` and `for` loops.

The general form of a `do...while` flow diagram is shown on the left. Control passes through the loop body before reaching the test at the end of the loop. On the right is a diagram of a summing loop, equivalent to the loops in Figure 6.4 that use `while` and `for` statements.

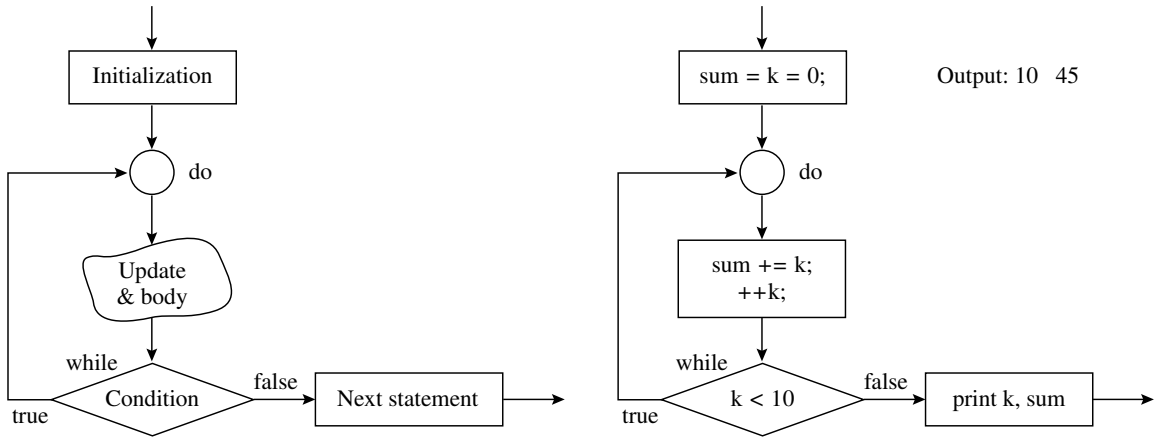


Figure 6.5. A flow diagram for the `do...while` statement.

Finally, note that the body of the loop will not be executed at all if the condition is false the first time it is tested.

Updating the loop variable. The update expression is evaluated every time around the loop, after the loop body and before re-evaluating the loop condition. Note the mismatch between *where* the update is written on the page and *when* it happens. It is written after the condition and before the body, but it happens after the body and before the condition. Beginning programmers sometimes are confused by this inverted order. Let the flow diagram be your guide to the proper order of evaluation.

In a counted `for` loop with loop variable `k`, the update expression often is as simple as `k++` or `++k`. A lot of confusion surrounds the question of which of these is correct and why. The answer is straightforward: As long as the update section is just a simple increment or decrement expression, it does not matter whether you use the prefix or postfix form of the operation. Whichever way it is written, the loop variable will be increased (or decreased) by 1 after the loop body and before the condition is retested.

The loop body. The loop body is executed after the condition and before the update expression. In Figure 6.1, compare the `for` loop body, which contains only one statement, with the two-statement body of the `while` loop. The update expression must be in the body of a `while` loop but becomes part of the loop header of a `for` loop, shortening the body of the loop by at least one statement. Often, this reduces the body of a `for` loop to a single statement, which permits us to write the entire loop on one line, without braces, like this:

```
for (sum = 0, k = 0; k < 10; ++k) sum += k;
```

6.1.2 The `do...while` Loop

The `do...while` loop implements a different control pattern than the `while` and `for` loops. The body of a `do...while` loop is executed at least once; this is illustrated by the flow diagram in Figure 6.5. The condition, written after the keyword `while`, is tested after executing the loop body. Therefore, unlike the `while` and `for` loops, the body of a `do...while` loop can initialize the variables used in the test. This makes the `do...while` loop useful for repeating a process, as shown in Figure 6.11.

6.1.3 Other Control Statements

C supports four statements whose purpose is to transfer control to another part of the program. Two of these, `break` and `continue`, are used in conjunction with loops to create additional structured control patterns. The

A `continue` statement takes control directly to the top of at `while` or `do` loop.

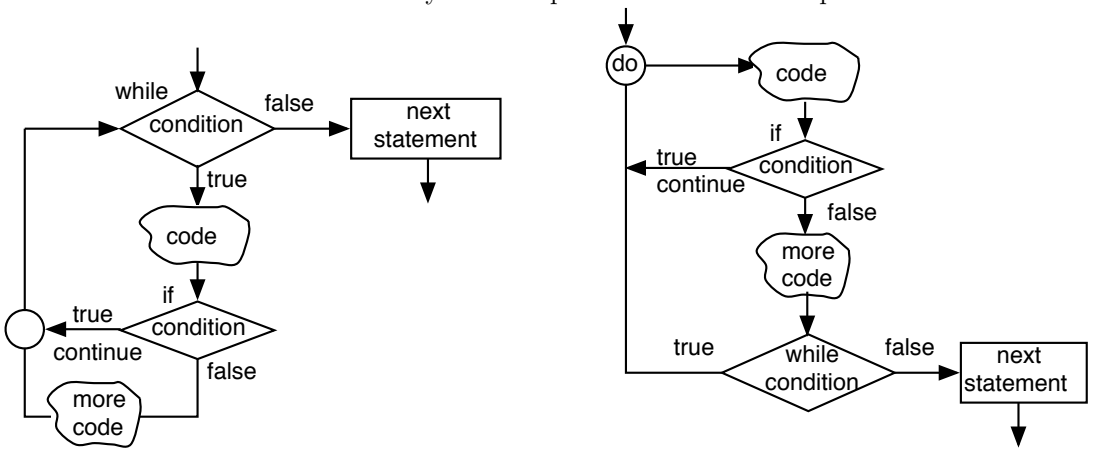


Figure 6.6. Using continue with while and do.

third, `return`, is used at the end of a function definition. The fourth, `goto`, has little or no place in modern programming and should not be used. Neither `break` nor `continue` is necessary in C; any program can be written without them. However, when used skillfully, a `break` or `continue` statement can simplify program logic and shorten the code. This is highly desirable because decreasing complexity decreases errors.

The `break` statement. The `break` statement interrupts the normal flow of control by transferring control from inside a loop or a `switch`[1] to the statement after its end. If the `break` is within a `for` loop, execution continues with the increment or update step. In a `while` loop, execution continues with the loop test. In a `do...while` loop, execution continues with the first statement in the body of the loop.

The `break` statement is diagrammed as an arrow because it interrupts the normal flow of control. An `if` statement whose true clause is a `break` statement is commonly used inside a `while` loop or a `for` loop. We will call this combination an `if...break` statement; it is diagrammed as an arrow that leaves the normal control path at an `if` statement inside the loop and rejoins the normal path at the statement after the loop. (See Figure 6.20.) This is discussed further in Section 6.2.7.

The `continue` statement. The `continue` statement interrupts the normal flow of control by transferring control from inside a loop to its beginning. If it is within a `for` loop, execution continues with the increment step, as shown on the right in Figure 6.6. In a `while` or `do...while` loop, execution continues with the loop test. The `continue` statement is also diagrammed as an arrow. An `if...continue` statement is diagrammed as an arrow that leaves the normal control path at the `if` statement and rejoins the loop at the top.

The `continue` statement is not commonly used but occasionally can be helpful in simplifying the logic when one control structure is nested within another, as when a loop contains a `switch` statement. This control pattern is used in menu processing and will be illustrated in Chapter 12 where `continue` is used within the switch to handle an invalid menu selection.

The `return` statement. By now, the `return` statement should be familiar to the reader; it has been used in every program example since Chapter 2. Here, we restate the rules for using `return`, and discuss a few usage options.

1. Executing a `return` statement causes control to leave a function immediately.
2. A `void` function may contain a simple `return` statement, in which the keyword `return` is followed immediately by a semicolon. If this statement is omitted, the function will return when control reaches the final closing brace.

[1]Section 6.3 deals with the `switch` statement.

A `continue` statement takes control to the increment step in the header of a `for` loop.

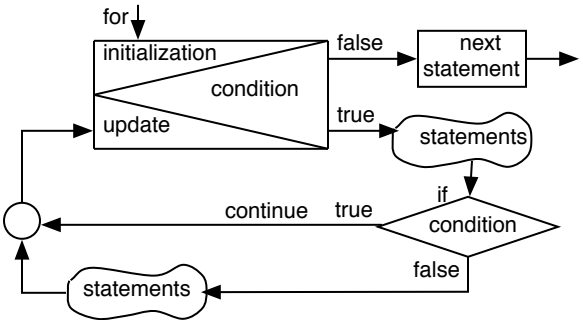


Figure 6.7. Using continue with for.

3. Every non-void function must contain a `return` statement with an expression between the keyword `return` and the semicolon. The type of this expression must match the declared return type or be coercible to that type[2]. When control reaches this statement, the expression will be evaluated and its value will be returned.
4. It is syntactically legal to have more than one `return` statement in a function. However, this is generally considered to be poor style. There is genuine debate, however, about whether multiple `return`s are sometimes appropriate. On one side, many teachers and employers prohibit this practice because it breaks the "one way in – one way out" design rule. On the other side, some experts believe that code simplicity is more important than "one in – one out" construction, and permit use of multiple return statements in a few situations where the extra return statements significantly shorten or reduce the nesting level of the code.

6.1.4 Defective Loops

If a loop does not update its loop variable, it will loop forever and be called an **infinite loop**. The most common cause of an infinite loop is that the programmer simply forgets to write a statement that updates the loop variable. Another source of error is missing or misplaced punctuation. The body of a loop starts at the right parenthesis that ends the condition. If the next character is a left curly bracket, the loop extends to the matching right curly bracket. Otherwise, the body consists of the single statement that follows the condition and the loop ends at the first semicolon. Figure 6.8 shows two loops that are infinite in nature because they do not update their loop variable. The loop on the left is like the `while` loop in Figure 6.1 except that the braces are missing. Because of this, the loop ends after the statement `sum += k;` and does not include the `++k` statement. This kind of omission will not be caught by the compiler; it is perfectly legal to write a loop with no braces and with only one statement in its body. The compiler does not check that the programmer has updated the loop variable.[3]

The loop on the right has an extraneous semicolon following the loop condition. This semicolon ends the loop, resulting in a null or empty loop body, which is legal in C. Since `k`, the loop variable, is not updated before this semicolon, it never changes. (The bracketed series of statements that follows the semicolon is outside the loop.) One way to avoid this kind of error is to write the left curly bracket that begins the loop body *on the same line* as the keyword `while`. The left bracket is a visible reminder that the line should not end in a semicolon.

[2]Type coercion is discussed in Chapter 7.

[3]A mathematical technique, called a *loop invariant*, can be used to find loops that do not accomplish the design goals. This technique is beyond the scope of an introductory textbook.

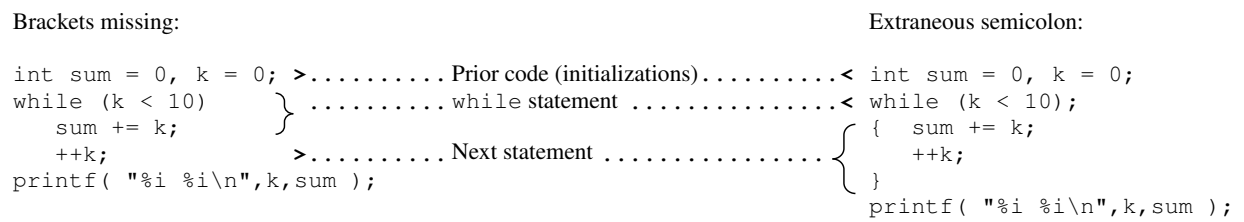


Figure 6.8. No update and no exit: Two defective loops.

6.2 Applications of Loops

Knowing the correct syntax for writing a loop is important but only part of what a programmer needs to understand. This chapter and later ones present several common applications of loops and paradigms for their implementation in C. These applications include

Sentinel loops. Introduced in Figure 6.23 and discussed in Section 6.2.1.

Query loops. Presented in Section 6.2.2.

Counted loops. Introduced in Figure 3.14 and treated in greater depth in Section 6.2.3.

Input validation loops. Introduced in Figure 3.15 and revisited in Section 6.2.4.

Nested loops. : Presented in Section 6.2.5.

Delay loops. Presented in Section 6.2.6.

Flexible-exit loops. Presented in Section 6.2.7.

Counted sentinel loops. Based on the flexible-exit loop, this pattern is presented in Section 6.2.8.

Search loops. Introduced in Section 6.2.9 and illustrated in several later chapters.

Table processing loops are introduced in Chapter 13.

End-of-file loops are introduced in Chapter 14.

6.2.1 Sentinel Loops

A **sentinel loop** keeps reading, inspecting, and processing data values until it comes across a predefined value that the programmer has designated to mean "end of data." Looping stops when the program recognizes this value, which is called a **sentinel value**, because it stands guard at the end of the data.

The value used as a sentinel depends on the application; to choose an appropriate sentinel value, the programmer must understand the nature of the data. Most functions that process strings use the null character as a sentinel value. Loops that read and process input data often use the newline character as a sentinel. If the data values are integers and a program processes only nonzero data values, then 0 can be used as a sentinel. If all data values are nonnegative, then -1 can be used as the sentinel. If every integer is admissible, the value `INT_MAX` (the largest representable integer) often is used as a sentinel.

In all cases, a sentinel value must be of the same data type as the ordinary data values being processed, because it must be stored in the same type of variable or read using the same conversion specifier in a format string. Also, a sentinel must not be contained in the set of legal data values because it must be an unambiguous signal that there are no more data sets to process.

A sentinel loop is used when the number of data items to be processed varies from session to session, depending on the user's needs, and cannot be known ahead of time. This happens in many contexts, including

- When reading a series of input data sets.

- When processing string data.[4]
- When processing data that are stored in an array or a list in which the sentinel value is stored at the end of the data[5].

Input-controlled sentinel loops are the only kind that we are ready to examine at this time[6]. An input-controlled sentinel program reads and processes input values; the sentinel value must be entered from the keyboard as a signal that there is no more input. The loop compares each input value to the sentinel and ends the input process if a match is found. Such programs follow this general form:

```
// Comments that explain the purpose of the program.
#include commands.
#define the sentinel value.

int main( void )
{
    Declaration of input variable and others.
    Output statement that identifies the program.

    Use scanf() to initialize the input variable.
    while (input != sentinel value)  {
         Process the input data.
         Prompt for and read another input.
    }
    Print program results and termination comment.
    return 0;
}
```

The user prompts must give clear instructions about the sentinel value. Otherwise, the user will be unable to end the loop. For example, consider the cash register program in Figure 6.9, which uses a simple input-controlled sentinel loop. The initial prompt gives clear instructions about how to end the processing loop. Sentinel loops are implemented with `while` or `for(;;)` statements, rather than a `do...while` statement, because it is important not to try to process the sentinel value. The `do...while` statement processes every value before making the loop test, whereas the `while` loop makes the test before processing the value. Thus, a `while` loop can watch for the sentinel value and leave the loop when it appears without processing it. Figures 6.23 and 6.24 show two ways that a `for` statement can be used to implement a sentinel loop.

6.2.2 Query Loops

If the loop variable is either initialized or updated by `scanf()` or some other input function, we say that it is an **input-controlled loop**. Such loops are very important because they allow a program to respond to the real-world environment. There are several variations on this theme, including query loops, sentinel loops (Section 6.2.1), and input validation loops (Section 6.2.4).

A useful interactive technique, the **repeat query**, is introduced in Figure 6.10 and used in Figure 6.11. (It will be refined, later, in Chapter 8.) To develop and debug a program, the programmer must test it with several sets of input so that its performance with different kinds of data can be checked. We use a repeat query loop to automate this process of rerunning a program. Until now, you have needed to execute a program once for each line in your test plan. After each run, the output must be captured and printed. At best, the process is awkward. At worst, the programmer is tempted to shortcut the testing process. A typical program with a query loop follows the general form shown in Figure 6.10.

The testing process can be simplified by writing a function that processes one line of the test plan. The main program contains a `do...while` loop that calls the function once and asks whether the user wishes to do it again. This provides a simple, convenient way to let the user decide whether to continue running the program or quit, rather than restarting it every time. Furthermore, all the output for all of the tests ends up in one place at one time. The basic technique is illustrated in Figures 6.10 through 6.12.

Notes on Figure 6.10. Form of a query loop.

[4]Strings will be introduced in Chapter 12.
[5]Arrays will be introduced in Chapter 10 and lists in Chapter 22.
[6]The other sentinel loops will be introduced in Chapters 10, 12, and 22.

Compute the sum of a series of prices entered by the user.

```
#include <stdio.h>
#define SENTINEL  0

int main ( void )
{
    double input;                               // Price of one item.
    double sum = 0;                             // Total price of all items.
    printf( " Cash Register Program.\n Enter the prices; use 0 to quit.\n> " );

    scanf ( "%lg", &input );      // Read first price.
    while (input != SENTINEL) {   // Sentinel value is 0
        sum += input;             // Process the input.
        printf( "> " );           // Get next input.
        scanf( "%lg", &input );
    }

    printf( " Your order costs $%g\n", sum );
    return 0;
}
```

Output from a sample run:

```
 Cash Register Program.
 Enter the prices; use 0 to quit.
> 3.10
> 29.98
> 2.34
> 0
 Your order costs $35.42
```

Figure 6.9. A cash register program.

Many programs perform a process repeatedly, until the user asks to stop. This is the general form of such a program. The process is performed by a function called from the main loop.

```
// Comments that explain the purpose of the program.
#include and #define commands.

int main( void )
{
    Declaration of variable for query response;

    Output statement that identifies the program;
    do {
        Process one data set or call a function to do so;
        Ask the user whether to continue (1) or quit (0);
        Read the response;
    } while (response != 0);

    Print program results and termination comment;
}
```

Figure 6.10. Form of a query loop.

This main program consists of statements to print the program titles and a loop that will repeatedly call the `work()` function, given in Figure 6.12, until the user asks to quit. This main program can be used to repeat any process by changing the titles and the `work()` function.

```
// ------------------------------------------------------------------------
// Determine the time it takes for a grapefruit to hit the ground when it
// is dropped, with no initial velocity, from a helicopter hovering
// at height h.  Also determine the velocity of the fruit at impact.
// ------------------------------------------------------------------------
#include <stdio.h>

void work( void );
double drop( double height );

int main( void )
{
    char do_it_again;     // repeat-or-stop switch

    puts( "\n Calculate the time it would take for a grapefruit\n"
          " to fall from a helicopter at a given height.\n" );

    do {
        work();
        printf( " \n Enter 'y' to continue or 'n' to quit: " );
        scanf( " %c", &do_it_again );
    } while (do_it_again != 'n');

    return 0;
}
```

Figure 6.11. Repeating a calculation.

- Programs that use this technique will have a few statements at the beginning of `main()` that may open files, clear the screen, or print output headings.
- At the end are statements to print final results and any closing message.
- In between is a `do...while` loop. The loop body consists entirely of a call on a function that performs the work of the program, followed by a prompt to ask the user whether or not to repeat the process. The response is read and immediately tested. If the user enters the code 0,[7] control leaves the loop. If 1 (or an erroneous response) is entered, the process is repeated.

Figure 6.13 shows a diagram of a query loop (on the left) used to repeat a calculation (the function diagrammed on the right). The dotted lines show how control goes from the function call to the entry point of the function and from the function return back to the call box.

Notes on Figure 6.11. Repeating a calculation. We put almost all of the program's code into a function called `work()`. The main program contains only greeting and closing messages, a loop that calls the `work()` function to do all the work, and the input statements needed to control the loop. Figure 6.13 is a flow diagram of this program.

First box: prototypes for the two programmer-defined functions. Two programmer-defined functions that follow the main program in the source code file are shown in Figure 6.12. The `work()` function is called from `main()`. It, in turn, calls `drop()`.

Outer box: the main process loop.

[7] We use a 1 or 0 response here because it is simple. In Chapter 8, we show how to process y or n responses.

The `work()` function is called from the program in Figure 6.11. The `drop()` function is a more concise version of the one in Figure 5.18.

```
// ------------------------------------------------------
// Perform one gravity calculation and print the results.
void
work( void )
{
    double h;           // height of fall (m)
    double t;           // time of fall (s)
    double v;           // terminal velocity (m/s)

    printf( " Enter height of helicopter (meters): " );
    scanf( "%lg", &h );

    t = drop( h );      // Call drop with the argument h.
    v = GRAVITY * t;    // velocity of grapefruit at impact
    printf( "    Time of fall = %g seconds\n", t );
    printf( "    Velocity of the object = %g m/s\n", v );
}

// ------------------------------------------------------
// Calculate time of fall from a given height. ----------
double
drop( double height )
{
    double answer = 0;
    if (height > 0) answer = sqrt( 2 * height / GRAVITY );
    return answer;
}
```

Figure 6.12. The `work()` and `drop()` functions.

- Compare this version of the program to the versions in Figures 3.10 and 5.15. The input, calculations, and output have been removed from `main()` and placed in a separate function, named `work()`. These lines have been replaced by a loop that will call the `work()` function repeatedly (inner box), processing several data sets. This keeps the logic of `main()` simple and easy to follow.
- We use a loop because we expect to process a series of inputs. Since we do not know how many will be needed ahead of time, we ask the user to tell us what to do after each loop repetition.
- We prompt for a 'y' to do the process again or a 'n' to quit; the response is read into the variable `do_it_again`.
- As long as the values we read for `do_it_again` are not 'n', we call the `work()` function to read another input value and perform the computation and output processing. If the user's input is an error, that is, it is neither a 'y' nor an 'n' , this program continues; it quits only if the user enters 'n'.

Notes on Figure 6.12: The `work()` and `drop()` functions.

Background. Most of the code from the `main()` program in Figure 5.15 has been moved into the `work()` function. This reduces the complexity of `main()` and makes it easy to repeat the gravity calculations with several inputs. The `work()` function contains all the declarations and code that relate to the computation. The `main()` program contains only start-up code, termination code, and the loop that calls the `work()` function. vspace*-1ex

This is a flow diagram of the program in Figures 6.11 and 6.13. The dotted lines show how the control sequence is interrupted when the function call sends control to the beginning of the function. Control flows through the function then returns via the lower dotted line to the box from which it was called.

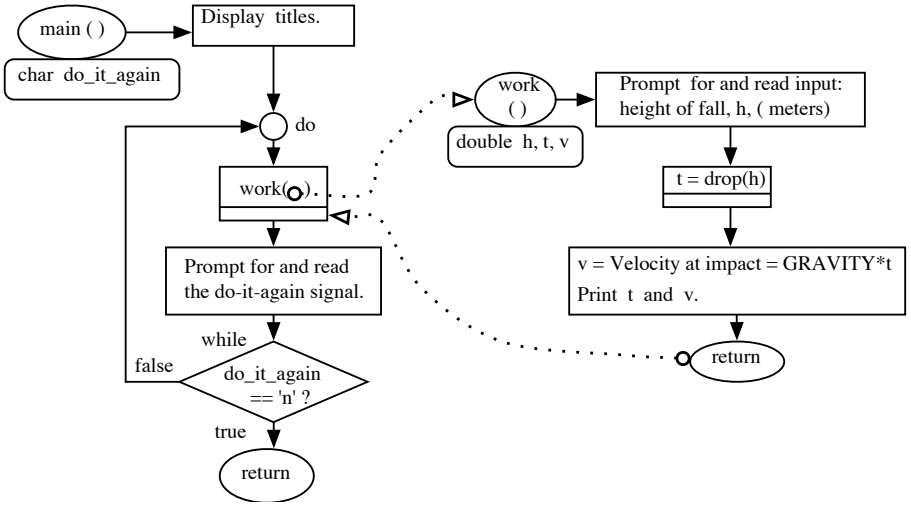


Figure 6.13. Flow diagram for repeating a process.

The flow of control. The function call in `main()` sends control into the `work()` function, where we read one input and calculate the time of fall by calling the `drop()` function. After returning from `drop()`, we calculate the terminal velocity and print the time and velocity. Then we return to `main()`. In the flow diagram (Figure 6.13), these shifts of control are represented by dotted lines.

The output. Lines printed by the `main()` program are intermixed with lines from the `work()` function. Here is a sample dialog:

```
Calculate the time it would take for a grapefruit
to fall from a helicopter at a given height.

Enter height of helicopter (meters): 20
    Time of fall = 2.01962 seconds
    Velocity of the object = 19.8057 m/s

Enter 'y' to continue or 'n' to quit:  1
Enter height of helicopter (meters): 906.5
    Time of fall = 13.597 seconds
    Velocity of the object = 133.34 m/s

Enter 'y' to continue or 'n' to quit:  2
Enter height of helicopter (meters):  2000.5
    Time of fall = 20.1987 seconds
    Velocity of the object = 198.082 m/s

Enter 'y' to continue or 'n' to quit:  0
```

As you begin to write programs, incorporate a processing loop into each one. It then will be convenient for you to test your code on a variety of inputs and demonstrate that it works correctly under all circumstances.

6.2.3 Counted Loops

Many loops are controlled by a counter. In such a **counted loop**, an initialization statement at the top of the loop usually sets some variable, say `k`, to 0 or 1. The update statement increments `k`, and the loop test asks whether `k` has reached or exceeded some goal value, `N`. To calculate the **trip count**, that is, the number of times the loop body will be executed,

- Let the initial value of the loop variable be I and the goal value be N.
- If the loop test has the form `k < N`, the trip count is $N - I$.
- If the loop test has the form `k <= N`, the trip count is $N - I + 1$.

This program computes the sum of the first N terms of the series $1/n$.

```
#include <stdio.h>
#define N  10

double f( double x) { return 1.0 / x; }    // The function to sum.

int main( void )
{
    int n;                  // Loop counter
    double sum;             // Accumulator

    printf( "\n Summing 1/n where n goes from 1 to %i \n", N );

    sum = 0;                        // Start accumulator at 0.
    for (n = 1; n <= N; ++n) {      // Sum series from 1 to N.
        sum += f( n );
    }

    printf( " The sum is %g.\n", sum );

    return 0;
}
```

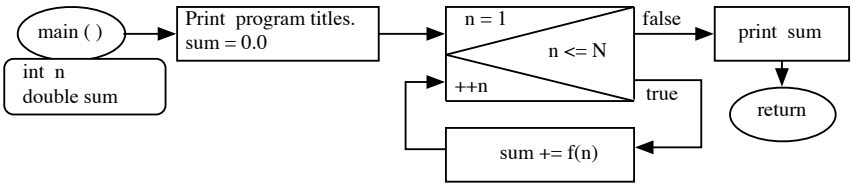


Figure 6.14. Summing a series.

The loops diagrammed in both Figures 6.4 and 6.14 are counted loops. In the first example, the loop variable is `k`, the initial value is 0, and the test is `k<10`; so this loop will be executed 10 times, with `k` taking on the values $0 \ldots 9$, successively. If an initial value of 1 and a loop test of `k <= N` were used, the loop body still would be executed 10 times, but the sum would be different, because `k` would have the values $1 \ldots 10$. Both patterns of loop control are common. A frequent source of program error is using the wrong initial value or the wrong comparison operator in the loop test.

A summation loop. Our next example, in Figure 6.14, shows how a counted loop can be used to sum a series of numbers. This example demonstrates a typical programming pattern in which two variables are used: a *counter* (to record the number of repetitions) and an *accumulator* (to hold the sum as it is accumulated). Both variables are initialized before the loop begins and both are changed on every trip through the loop. The flow diagram for this program (shown following the code) is very similar to that in Figure 6.4, since the same loop structure is being used.

Notes on Figure 6.14. SumUp.

First box: the function. The same program could be used to sum a different function by simply changing the expression in the `return` statement here and the `printf()` statement in `main()`.

Second box: the declarations.

- The type `int` is appropriate for counters such as `n`, that are used to count the repetitions of the loop.
- An accumulator is a numeric variable used to accumulate the sum of a series of values. These might be computed values, as here, or input values, illustrated in Figure 6.9. We use a variable of type `double` for

This input validation loop is a fragment of the miles-per-hour program in Figure 3.15. Some output from the loop and the following `printf()` statement follow.

```
printf( " Duration of trip in hours and minutes: " );
scanf( "%lg%lg", &hours, &minutes );
hours = hours + ( minutes / 60 );
while (hours < 0) {
    printf( " Please re-enter; time must be >= 0: " );
    scanf( "%lg%lg", &hours, &minutes );
    hours = hours + ( minutes / 60 );
}
```

Output:

```
Duration of trip in hours and minutes: -148 43
Please re-enter; time must be >= 0: 1 -70
Please re-enter; time must be >= 0: 148 -17
Average speed was 1.94291
```

Figure 6.15. Input validation using a `while` statement.

the accumulator because it will be used to compute the total of various fractions.

Third box: the loop.

- Before entering any loop, the variables used in the loop must be initialized. We set `n = 1` because we want to sum the series from 1 to 10. We set `sum = 0-->`.
- The `for` statement is ideally suited for counted loops because the loop header organizes all the information about where the counter starts and stops and how it changes at each step. Each time around the loop, we add `1.0/n` to the sum. This loop starts with `n` equal to 1 and ends when `n` reaches 11. Therefore, the loop body will be executed $11 - 1 = 10$ times, summing the fractions `1.0/1 ... 1.0/10`.
- Note that `n` will have the value 1 (not 0 or 2) the first time we compute `1.0/n`. This is important. First, we do not want to start computing the series at the wrong point. Second, we need to be careful to avoid dividing by 0.
- We are permitted to divide the `double` value 1.0 by the integer value `n`. The result is a fractional value of type `double`. It is important that the constant 1.0 be used rather than 1 in this expression, because the latter will give an incorrect result. The reason for this is discussed in Chapter 7.

Fourth box: the output.

- We use the format `%g` to print the `double` value. Up to six digits (C's default precision) will be printed with the result rounded to the sixth place. Trailing 0's, if any, are suppressed after the decimal point.
- The output from this program is

```
Summing 1/n where n goes from 1 to 10
The sum is 2.92897.
```

6.2.4 Input Validation Loops

Figure 6.15 contains a validation loop based on a `while` statement. It provides good user feedback but is long and requires duplicating lines of code. We can write a shorter, simpler validation loop that uses a `do...while` statement. However, this kind of validation loop has a severe defect: It gives the same prompt for the initial input and for re-entry after an error. This is a human-engineering issue. The error may go unnoticed if the program does not give distinctive feedback. A third kind of validation loop can be written with `for` and `if...break` statements that combines the advantages of the other two forms; it avoids duplication of code and provides informative feedback when the user makes an error (see Figure 6.15).

This program prints a multiplication table with 10 rows and 12 columns. The line number and a vertical line are printed along the left margin.

```
#include <stdio.h>
#define R 10
#define C 12

int main( void )
{
        int row, col;

        banner();
        printf( "\n\n Multiplication Table \n\n" );

        for (row = 1; row <= R; ++row) {       // Print R rows.
            printf( "%2i.  |", row );          // Print left edge of row.
            for (col = 1; col <= C; ++col) {   // Print C columns in each row.
                printf( "%4i", row * col );
            }
            printf( "\n     |\n" );            // Print blank row.
        }

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

Figure 6.16. Printing a table with nested `for` loops.

Notes on Figure 6.15: Input validation using a `while` statement.

1. The input prompt `scanf()` and calculation statements are written before the loop and again in the body of this loop. This duplication is undesirable because both sets of statements must be edited every time the prompt or input changes. The duplication is unavoidable, though, because the loop variable, `hours`, must be given a value before the `while` test at the top of the loop. It then must be given a new value within the loop.
2. The `while` test checks whether the value of `hours` is within the legal range. If not, we enter the body of the loop, print an error comment, and prompt for and read another input. Then control returns to the top of the loop to test the data again.
3. Since only a legal input will get the user out of this loop, an informative prompt is very important. If the user is not sure what data values are legal, the program may become permanently stuck inside this loop and it may be necessary to reboot the computer to regain control.

6.2.5 Nested Loops

A general rule of programming is that function follows form, that is, the form or shape of the input or output data frequently determines the way that code is written to process it. This is never more true than when processing tables. The rows and columns of a table are reflected in the code in the form of a loop to process the columns written within a loop that process the rows. We call such a control structure a *nested loop*[8]. This control structure is illustrated in a simple but general form by the program in Figure 6.16, which prints a 10-by-12 multiplication table. Its flow diagram is given in Figure 6.17.

[8]For processing two-dimensional arrays, a `for` loop within a `for` loop is the dominant control pattern. This will be fully explored in Chapter 18.

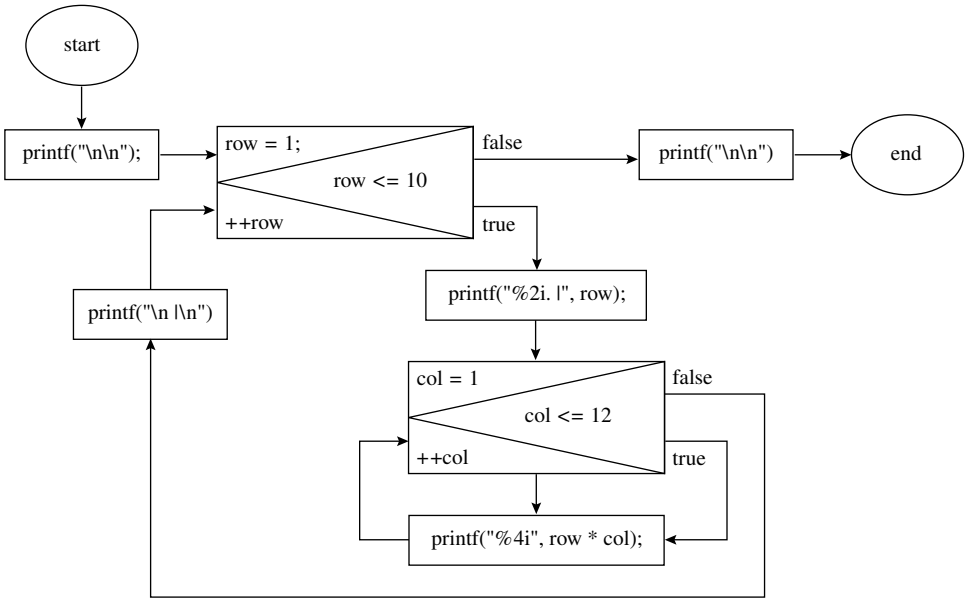


Figure 6.17. Flow diagram for the multiplication table program.

Notes on Figure 6.16. Printing a table with nested `for` loops. This program prints an R-by-C multiplication table, where `R` and `C` are `#define`d as 10 and 12.

Outer box: The `row` loop. The outer loop is executed 10 times, once for each row of the table. Its body consists of the code to process one row; it prints a row label, processes all columns (the inner loop), and finishes the row by printing a newline and a vertical bar on the next line. This is a very typical processing pattern for a nested loop that does output. The output from one repetition, one row of numbers followed by a blank row looks like this:

```
1. |   1   2   3   4   5   6   7   8   9  10  11  12
   |
```

Inner box: the column loop. The output from each repetition of the inner loop is one column of one row of the table, consisting of two or three spaces and a number. The `printf()` in the inner loop will be executed 12 times per trip through the outer loop. After the 12th number has been printed, the inner loop exits and control goes to the `printf()` at the end of the outer box.

After 120 trips. In this program, control goes through the outer loop 10 times. For each trip through the outer loop, control goes through the inner loop 12 times. Therefore, control passes through the body of the inner loop (a `printf()` statement) a total of 120 times, processing every column in every row of the table. After finishing the 10th row, control goes to the final `printf()` and the `return` statement. The complete output is

```
Multiplication Table

 1. |   1   2   3   4   5   6   7   8   9  10  11  12
    |
 2. |   2   4   6   8  10  12  14  16  18  20  22  24
    |
 3. |   3   6   9  12  15  18  21  24  27  30  33  36
    |
 4. |   4   8  12  16  20  24  28  32  36  40  44  48
    |
 5. |   5  10  15  20  25  30  35  40  45  50  55  60
    |
 6. |   6  12  18  24  30  36  42  48  54  60  66  72
    |
 7. |   7  14  21  28  35  42  49  56  63  70  77  84
```

```
    |
 8. |   8  16  24  32  40  48  56  64  72  80  88  96
    |
 9. |   9  18  27  36  45  54  63  72  81  90  99 108
    |
10. |  10  20  30  40  50  60  70  80  90 100 110 120
    |
```

6.2.6 Delay Loops

A loop that executes many times but does nothing useful can be used to make the computer wait for a while before proceeding. Such a loop is called a **delay loop**. Delay loops often are used like timers to control the length of time between repeated events (see Figure 6.18). For example, a program that controls an automated factory process might use a delay loop to regulate sending analog signals to (or receiving them from) a device such as a motor generator.

Notes on Figures 6.19 and 6.18. Delaying progress, the `delay()` function. The `delay()` function implements a delay loop (see Figure 6.19). It calls the `C` library function `time()` to read the computer's real-time clock and return the current time, in units of seconds, represented as an integer so that we can do

---

A delay loop is used here to regulate repetitions of a process.

```
#include <stdio.h>
void delay( int seconds );

int main( void )
{
    int j, max, seconds;

    printf( "This is an exercise program.\n\n"
            "How many pushups are you going to do? " );
    scanf( "%i", &max );
    if (max < 0) {
            printf( "Can't do %i pushups!\n", max );
            exit( 1 );
    }
    printf( "How many seconds between pushups? " );
    scanf( "%i", &seconds );
    if (seconds < 3) {
            printf( "Can't do a pushup in %i seconds!\n", seconds );
            exit( 1 );
    }
    printf( "OK, we will do %i pushups, one every %i seconds.\n",
            "Do one pushup each time you hear the beep.\n", max, seconds );

    for (j = 1; j <= max; ++j) {
        printf( "%i \a\n", j );   // Do one.
        delay( seconds );         // Wait specified # of seconds.
    }

    puts( "Good job.  Come again." );
}
```

Figure 6.18. Using a delay loop.

You may wish to put this function in your personal `mytools` library.

```
#include <time.h>
void delay( int seconds )
{
    time_t goal = time( NULL ) + seconds;   // Add seconds to current time.
    do {    // Nothing    } while (time( NULL ) < goal);
}
```

Figure 6.19. Delaying progress, the `delay()` function.

arithmetic with it. The type `time_t` is defined[9] by your local C system to be the right kind of integer[10] for storing the time on your system.

We add the desired number of seconds of delay to the current time to get the goal time, then store it in the variable `goal`. The loop calls `time()` continuously until the current time reaches the goal time. This loop is all test and no body. Technically, it is called a **busy wait** loop because it keeps the processor busy while waiting for time to pass. It is busy doing nothing, that is, wasting time[11]. On a typical personal computer, the `time()` function might end up being called 100,000 times or more during a delay of a few seconds. Busy waiting is an appropriate technique to use when a computer is dedicated to monitoring a single experiment or process. It is not a good technique to use on a shared computer that is serving other purposes simultaneously.

A delay loop usually is used inside another loop, which must perform a process repeatedly at a particular rate that is compatible with human response or a process being monitored. For example, the boxed loop in Figure 6.18 is used to time repetitions of an exercise. It outputs `\a` (a beep), then calls `delay()`, which waits the number of seconds specified by the user before returning. The output looks like this:

```
This is an exercise program.

How many pushups are you going to do? 5
How many seconds between pushups? 3
OK, we will do 5 pushups, one every 3 seconds.
Do one pushup each time you hear the beep.
1
2
3
4
5
Good job.  Come again.
```

6.2.7 Flexible-Exit Loops

Some languages support a kind of loop that permits the programmer to place the loop test anywhere between the beginning and the end of the loop body. Such a loop takes the place of the `while` and `do...while` loops in C and also provides other options. At the same time, it remains a one-in, one-out control structure, and therefore, is consistent with "structured programming". This sort of flexibile-exit structured loop can be imitated in C by a combination of three control statements: a `for` statement with empty loop test[12], where the only exit is by way of an `if` statement and a `break` statement somewhere within the loop body[13]. The skeleton of a `for` loop with an `if...break` is shown below and diagrammed on the left in Figure 6.20.

```
for (initialization; ; update) {
    statements
    if (condition)  break;
    more statements
}
next statement    // Control comes here after the break.
```

[9]Type definitions are discussed in Chapters 8, 12, 13, and 18.

[10]The various kinds of integers are discussed in Chapter 7.

[11]This is legal in C; a loop is not required to have any code in its body.

[12]It is legal to omit one, two, or all of the expressions in the loop header. However, the two semicolons must be present.

[13]Some practitioners use `while (1)` or `while (true)` instead of `for (;;)`. However, B. Kernighan, one of the inventors of C, wrote to me that he prefers the `for(;;)`, possibly because it avoids writing a meaningless test condition.

The only exit from this loop is through the `break` statement. The `for` header is used to initialize and update a counter but it has no exit test.

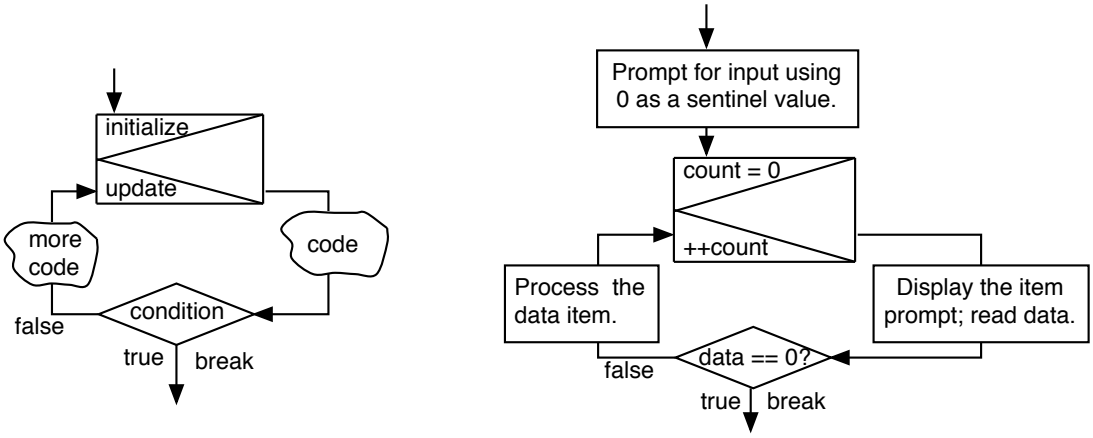


Figure 6.20. A structured loop with break.

This degenerate statement sometimes is called an **infinite `for` loop** because the loop header has no test that can end the execution of the loop. A real infinite loop is not particularly useful because it never ends. However, an infinite `for` loop normally contains an `if...break` statement and, therefore, is not infinite because the `if...break` provides a way to leave the loop. Applications of the infinite `for` loop are shown in Figure 6.21, where it is used for data validation, and in Figure 12.27, where an infinite loop is combined with a `switch` to implement a complex control structure.

A loop that exits by means of an `if...break` statement has one big advantage over an ordinary loop: the loop body can have some statements before the loop test and more statements after it. This flexibility makes it easier to write code in many situations: it provides a straightforward, clear, nonredundant way to do jobs that are awkward when done with other loops. Its primary application is in loops that depend on some property of data that is being read and processed. The first part of the loop does the input or calculation, followed immediately by the test that stops the looping when data with a specified property is found.

Notes on Figure 6.21. Input validation using a `for` statement. Compare the code in Figure 6.21 to the validation loop in Figure 6.15.

1. The original input prompt is written before the loop because it will not be repeated. (The error prompt is different.)
2. This loop can be written using a `while` statement as in Figure 6.15. However, since the `while` test is at the top of the loop, this implementation requires the input and calculation statements to be written twice, once before the loop and once in the loop body.

Compare this input validation loop to the version in Figure 6.15 that is built on the `while` statement. This form is simpler and avoids duplication. The output is identical to the output from Figure 6.15.

```
printf( " Duration of trip in hours and minutes: " );
for (;;) {
    scanf( "%lg%lg", &hours, &minutes );
    hours = hours + ( minutes / 60 );
    if (hours >= 0) break;               // Leave loop if input is valid.
    printf( " Please re-enter; time must be >= 0: " );
}
```

Figure 6.21. Input validation loop using a `for` statement.

Two designs are given for a counted sentinel loop. The version on the left is simpler. The version on the right avoids use of `break`.

```
for (k = 0; k < LIMIT; ++k)  {
    Prompt for and read an input.
    if (input == SENTINEL) break;
    Process the input data.
}
```

```
int done = 0;  // false
for (k = 0; k < LIMIT && !done; ++k) {
    Prompt for and read an input.
    if (input == SENTINEL) done = 1; // true
    else {
        Process the input data.
    }
}
```

Figure 6.22. Skeleton of a counted sentinel loop.

3. In contrast, if we use a flexible-exit loop, there is no need to write the `scanf()` and computation twice. The resulting code is simpler and clearer.
4. The input and calculation statements are done before the loop test, which is in an `if...break` statement. If the input is valid, control leaves the loop.
5. If not, we print an error prompt, return to the top of the loop, and read new data.
6. Since only a legal input will get the user out of this loop, an informative prompt is very important. If the user is not sure what data values are legal, the program may become permanently stuck inside this loop, and it may be necessary to reboot the computer to regain control.

6.2.8 Counted Sentinel Loops

Earlier in this section, we discussed a sentinel loop based on `while`. Often, we combine the sentinel test with a loop counter to make a counted sentinel loop. The general design for such a loop is given on the left in Figure 6.22. In this design, `LIMIT` is the maximum number of times the loop should be repeated, and `SENTINEL` is the designated sentinel value. To illustrate this pattern, we add a counter to the cash register program from Figure 6.9. The improved program is shown in Figure 6.23.

Moving the exit test to the top. A counted sentinel loop can be written two ways, with and without the use of `break`. Some professionals believe that the `break` statement should never be used because it is possible to overuse `break` and use it as a substitute for clean logic design. It is possible to implement the flexible-exit loop without the `break` statement by adding only a few lines of code. The loop design on the right of Figure 6.22 shows the kind of additions that are necessary to avoid the `break`.

We have added a status variable, `done`, that is set to false initially and then to true when the designated sentinel value is read. The "more code" section of the loop body is enclosed in an `else` clause so that the sentinel value will not be processed. Then the status variable is tested again and the loop ends. This is slightly less efficient and slightly longer to write than the version that uses break. Note, also, that the counter will be incremented one extra time and, therefore, we must subtract one from its value to get the true number of items that have been processed. The program in Figure 6.24 uses this logic.

6.2.9 Search Loops

A **search loop** examines a set of possibilities, looking for one that matches a given "key" value. The data items being searched can be stored in memory or calculated. The key value could be the entire item or part of it and it could be of any data type. The requirements for searching are almost identical in all cases:

- The program must know what key value to look for.
- There must be some orderly way to examine the possibilities, one at a time, until all have been checked.
- The loop must compare the key value to the current possibility. If they match, control must leave the loop.

We use a `for` loop to count the number of data items that are entered, and a `break` statement to leave the loop when the input is a designated sentinel value.

```
#include <stdio.h>
#define SENTINEL 0   // Signal for end of input.
int main( void )
{
    double input; // Price of one item.
    double sum;   // Total price of all items.
    int count;    // Number of items.

    puts( " Cash Register Program.\n"
          " Enter prices; 0 to quit." );
    for (count=sum=0;  ;++count) {
        printf( "--> " );
        scanf( "%lg", &input );
        if (input == SENTINEL) break;
        sum += input;
        printf( "\t Input:  %g\n", input );
    } printf( " Your %i items cost $%g\n", count, sum );
    return 0;
}
```

The output is :

```
Cash Register Program.
Enter prices; 0 to quit.
--> 3.17
Input:  3.17
--> 2.35
Input:  2.35
--> 0.78
Input:  0.78
--> 10.52
Input:  10.52
--> 0
Your 4 items cost $16.82
```

Figure 6.23. Breaking out of a loop.

This program uses a status flag (`done`) instead of a `break` instruction to leave the loop. Compare it the simpler logic of Figure 6.23, that relies on the `break`.

```
#include <stdio.h>
#define SENTINEL 0   // Signal for end of input.
int main( void )
{
    double input;   // Price of one item.
    double sum;     // Total price of all items.
    int count;      // Number of items.
    int done = 0;   // Set to 1 when sentinel is entered.

    puts( " Cash Register Program.\n Enter prices; 0 to quit." );
    for (count=sum=0; !done ;++count) {
        printf( "--> " );
        scanf( "%lg", &input );
        if (input == SENTINEL) done = 1;
        else {
            sum += input;
            printf( "\t Input:  %g\n", input );
        }
    } printf( " Your %i items cost $%g\n", count-1, sum );
    return 0;
}
```

Figure 6.24. Avoiding a break-out.

Two designs are given for a search loop, with and without the use of `break`.

```
Read or select the key value.
for (k = 0; k < LIMIT; ++k) {
   Calculate / select next item to test.
   if (current_data == key_value) break;
}
```

```
int done = 0;  // false
Read or select the key value.
for (k = 0; k < LIMIT && !done;  ++k) {
   Calculate / select next item to test.
   if (current_data == key_value) done = 1;
}
```

Figure 6.25. Skeleton of a search loop.

- The search loop must know how many possibilities are to be searched or have some way to know when no possibilities remain. The loop must end when this occurs.

Therefore, the search loop will terminate for two possible reasons: The key item has been found or the possibilities have been used up. The most straightforward way to implement this control pattern is to use a counted sentinel loop, with either the status flag or the `break` statement. The general pattern of a search loop is shown in Figure 6.25.

A search loop based on data input is shown in Figure 6.31. Search loops based on computed possibilities are illustrated in Figures **??** and in the Newton's method program on the text website. Search loops become increasingly important when there are large amounts of stored data. Loops that search arrays are illustrated in Chapter 18.

6.3 The `switch` Statement

C provides three ways to make a choice: the `if...else` statement, the `switch` statement, and the conditional operator. The `if...else` was fully covered in Chapter 3, this section is devoted to the `switch`, and the conditional operator is explained in Appendix **??**.

6.3.1 Syntax and Semantics

A `switch` implements the same control pattern as a series of `if...else` statements. (Refer to Figure 3.11.) In both, we wish to select one action from a set of possible actions. A `switch` could be thought of as a shorter, integrated notation for the same logic as a nested `if...else` statement. Normally, one would use `if...else` when there are only two alternatives; a `switch` is only helpful when there are more than two choices.

The syntax of a `switch` statement. To illustrate the syntax and use of the `switch` statement, we use a program whose specification is given in Figure 6.26 and program code in Figure 6.28. A `switch` statement has several parts, in this order:

1. The keyword `switch`
2. An expression in parentheses that computes a value of some integral type[14]
3. Braces enclosing a series of labeled `cases`. The case labels must be constants or constant expressions[15] of the same type as (2).
4. Each case contains a series of statements. The last statement in the series is usually, but not always, a `break` statement.
5. One of the cases may be labeled `default`. If a `default` case is present, it is traditional (but not necessary) to place it last.
6. If several cases require the same processing, several case labels may be placed before the same statement. The last label may be `default`.

[14] C has several integral types in addition to `int`; they will be introduced in the next few chapters. These include `char`, `short`, `long`, `unsigned short`, `unsigned int`, and `unsigned long`.

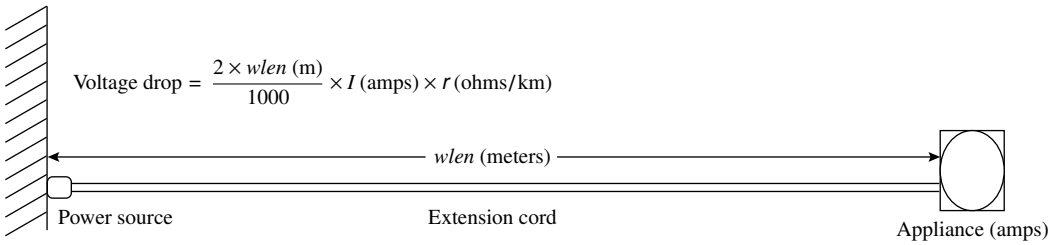
[15] A **constant expression** is composed of operators and constant operands.

Execution of a `switch` statement. When a `switch` is executed, the expression in (2) is evaluated and its value compared to the case labels. If any case label matches, control will go to the statement following that label. Control then proceeds to the following statement and through all the statements after that until it reaches the bottom of the `switch`. This is *not* normally what a programmer wishes to do. It is far more common to want the cases to be mutually exclusive. This is why each group of statements normally ends with a `break` statement that causes control to skip around all the following cases and go directly to the end of the `switch` statement. Programmers sometimes absentmindedly forget a `break` statement. In this case, the logic flows into the next case below it instead of to the end of the `switch` statement. Remember, this is not an error in the eyes of the compiler and it will not cause an error comment.

If no case label matches the value of the expression, control goes to the statement following the `default` label. If there is no `default` label, control just leaves the `switch` statement. This does not cause an error at either compile time or run time.

Problem scope: A contractor wants to automate the process of selecting an appropriate gauge wire for extension cords and temporary wiring at construction sites. Various long wires are used to supply electricity to power tools and other appliances at the site. All wires are standard annealed copper; calculations are to be made for a standard temperature of 20°C. There is a voltage drop in an extension cord due to the resistivity of its wire; heavier cords (lower-numbered gauges) have lower resistivity and incur less drop than lighter cords (higher-numbered gauges). The voltage drop is proportional to the length of the wire, so the drop can be significant in a long wire. This is an issue because appliances designed to operate at one voltage may overheat if operated at a voltage that is significantly lower. This program should evaluate a proposed wiring scheme and answer whether the wire will be adequate for its intended purpose.

Formula: An extension cord n meters long contains $2n$ meters of wire. The voltage drop is



where *wlen* is the length of the extension cord (in meters), ρ (rho) is the resistivity of the wire, and I is the current flowing in the wire, in amperes.

Constants: A voltage drop of up to 5 volts is acceptable. The resistivity, ρ, of copper wire at 20°C for the gauges used by this contractor are

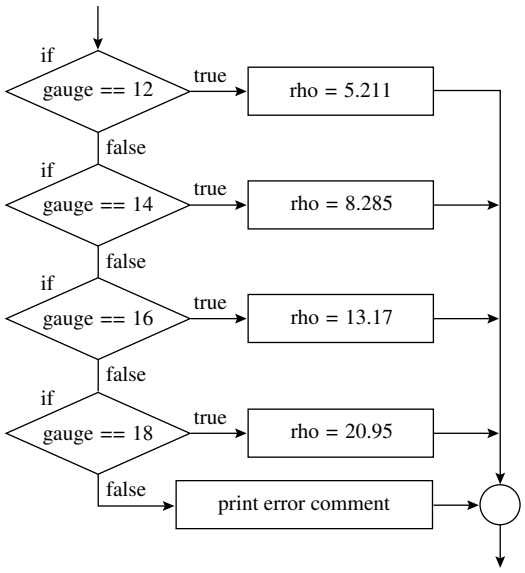
Gauge	ρ
12	5.211
14	8.285
16	13.17
18	20.95

Inputs: The contractor will type in the length of the extension cord he needs, the wire gauge, and the current rating (amps) of the appliance he will be using. The program should reject any gauge that is not in the table above.

Output required: Program headings and a termination message should be printed. Input values should be echoed. The voltage drop and the answer (gauge is adequate or not adequate) should be given. Three digits of precision are adequate.

Figure 6.26. Problem specification: Wire gauge adequacy evaluation.

We diagram a series of `if...else` statements that implements the specification in Figure 6.26.



We diagram the same logic again using a `switch` statement instead of an `if...else` statement.

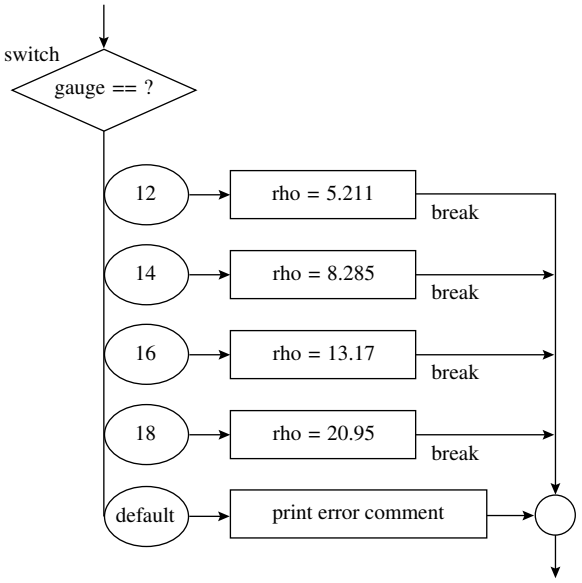


Figure 6.27. Diagrams for a nested conditional and a corresponding `switch`.

Diagramming a `switch` statement. The diagram of a nested `if...else` statement has a series of diamond-shaped `if` boxes, each enclosing a test, as shown on the left in Figure 6.27. The `true` arrow from each box leads to an action and the `false` arrow leads to the next `if` test. These tests will be made in sequence until some result is `true`.

In contrast, the diagram of a `switch` statement has a single diamond-shaped test box with a branching "out" arrow. This box encloses an expression whose value will be compared to the case labels. One branch is selected and the corresponding actions in the body of the `switch` statement are executed. Each set of actions must end in a `break` statement. Control normally enters a box from the case label and leaves by an arrow that goes directly to the connector at the end of the `switch` statement. The diagram on the right of Figure 6.27 shows the `switch` statement that is used in Figure 6.28. Compare this to the `if...else` to its left. They implement the same logic; however, the version that uses `switch` is simpler.

6.3.2 A `switch` Application

Figure 6.28 shows a program that implements this `switch` statement to solve the problem specified in Figure 6.26. It illustrates "messy" integer case labels; that is, they are not consecutive numbers starting at 0 or 1[16].

Notes on Figure 6.28: Using a `switch`.

First box: input for the switch. We display a list of available gauges and prompt the user for a choice. The user sees this set of choices:

```
Wire Gauge Adequacy Evaluation

Please choose gauge of wire:
    12 gauge
    14 gauge
    16 gauge
    18 gauge
Enter selected gauge:
```

[16]More applications of the `switch` statement are shown in Figures 13.5, 12.24, and 15.21.

The specifications for this program are in Figure 6.26.

```
#include <stdio.h>
#include <stdlib.h>
#define  MAXDROP   5.0      // volts

int main( void )
{
    int gauge;      // selected gauge of wire
    double rho;     // resistivity of selected gauge of wire
    double amps;    // current rating of appliance
    double wlen;    // length of wire needed
    double drop;    // voltage drop for selected parameters

    printf( "\n Wire Gauge Adequacy Evaluation" );

    printf( "\n Please choose gauge of wire:\n"
            "\t 12 gauge \n\t 14 gauge \n"
            "\t 16 gauge \n\t 18 gauge \n"
            " Enter selected gauge: " );
    scanf( "%i", &gauge );

    switch (gauge) {
      case 12:  rho = 5.211;   break;
      case 14:  rho = 8.285;   break;
      case 16:  rho = 13.17;   break;
      case 18:  rho = 20.95;   break;
      default:  printf( " Gauge %i is not supported.\n", gauge );
                exit(1);
    }

    printf( " Enter current rating for appliance, in amps: " );
    scanf ( "%lg", &amps );
    printf( " Enter the length of the wire, in meters: " );
    scanf ( "%lg", &wlen );
    drop = 2 * wlen / 1000 * rho * amps ;

    printf( "\n For %i gauge wire %g m long and %g amp appliance,\n"
            " voltage drop in wire = %g volts.  (Limit is %g.)  \n"
            gauge, wlen, amps, drop, MAXDROP );

    if (drop < MAXDROP)
        printf( "\n Selected gauge is adequate.\n" );
    else
        printf( "\n Selected gauge is not adequate.\n" );

    return 0;
}
```

Figure 6.28. Using a switch.

Second box: the* `switch` *statement.

- A `switch` that processes an integer input must have integer constants for case labels. This `switch` has four cases to process the four gauges plus a `default` case for errors.
- Each "correct" case contains one assignment statement that stores the resistivity value for the selected gauge wire. Each assignment is followed by a `break` that ends the case. A `default` case does not need a `break` because it is always last.
- If more extensive processing is needed, a program might call a different function to process each case.
- Error handling is done smoothly in this program. The `default` case intercepts inputs that are not supported and calls `fatal()` to print an error comment and abort the program. By calling `fatal()`, we avoid printing meaningless answers. An example would be

```
Enter selected gauge: 11
Gauge 11 is not supported.

WireGuage has exited with status 1.
```

Third box: calculating the voltage drop.

- Control goes to this box after every `break`. It does not go here after executing the `default` clause because that clause calls `fatal()`, which aborts execution.
- This box prompts for and reads the rest of the input data. This is done after the `switch` statement, not before, because the menu selection can be an invalid choice and there is no point reading the rest of the data until we know the gauge is one of those listed in the table.
- We calculate the voltage drop as soon as all the data have been read. The formula for voltage drop uses the resistivity value selected by the `switch` statement. We divide by 1,000 because ρ is given in ohms/kilometer and the wire length is given in meters. We multiply by 2 because each extension cord contains a pair of wires running its full length.

Fourth box: the answers. We ran the program and tested two cases. Each time the program was run, the greeting comment, menu, and termination comment were printed; for brevity, these are not repeated here. Dashed lines are used to separate the runs.

```
Enter selected gauge: 12
Enter the current rating for the appliance, in amps: 10
Enter the length of the wire, in meters: 30

For 12 gauge wire 30 meters long and 10 amp appliance,
voltage drop in wire = 3.1266 volts.  (Limit is 5 volts.)

Selected gauge is adequate.
-------------------------------------------------------
Enter selected gauge: 16
Enter the current rating for the appliance, in amps: 10
Enter the length of the wire, in meters: 30

For 16 gauge wire 30 meters long and 10 amp appliance,
voltage drop in wire = 7.902 volts.  (Limit is 5 volts.)

Selected gauge is not adequate.
```

6.4 Search Loop Application: Guess My Number

The program specified in Figure 6.29 and shown in Figure 6.30 is a simple interactive game in which the player is given a limited number of turns to guess (and enter) the hidden number. The game ends sooner if the player's input equals the program's hidden number (the sentinel value). The implementation uses a counted sentinel loop with `break`. The `for` statement is used to count the player's guesses and the `if...break` is used in the usual way to implement a possible early exit after a correct guess. Figure 6.31 is a flow diagram for this program[17].

Notes on Figures 6.31 and 6.30: A sentinel loop using `for` and `if...break` statements.

[17] We will revisit and elaborate on this game in Chapter 7.

1. **Problem scope:** Write a program to play an interactive guessing game with the user. The user is given a fixed number of guesses to find a hidden number.
2. **Constants:** The hidden number will be between 1 and 30; the user will be given up to 5 guesses.
3. **Inputs:** The user will enter a series of guesses.
4. **Output required:** The program will respond each time by saying the guess is too low, correct, or too high. If the guess is correct, the program should display the message"you win". If the available guesses are used up, the program should display the message"You lose".
5. **Other:** An input value outside of the specified range will be counted as a wrong guess. If the user makes optimal guesses, he can always win.

Figure 6.29. Problem specification: Guess my number.

This program illustrates the simplest form of search loop: we search the input data for a value that matches the search key. The implementation uses a counted sentinel loop with `break`.

```
#include <stdio.h>
#define TRIES 5

int main( void )
{
    int k = 0;              // Loop counter.
    int guess;              // User's input.
    int num = 17;

    printf( " Can you guess my number?  It is between 1 and 30.\n"
            " Enter a guess at each prompt; You have %i tries.\n", TRIES );

    for (k = 1; k <= TRIES; ++k) {
        printf( "\n Try %i: ", k );
        scanf( "%i", &guess );
        if (guess == num) break;
        if (guess > num) printf( " No, that is too high.\n" );
        else printf( " No, that is too low.\n" );
    }

    if (guess == num) printf( " YES!!  That is just right.  You win!  \n" );
    else printf( " Too bad --- You lose again!\n" );
    return 0;
}
```

Figure 6.30. An input-driven search loop.

This is a flow diagram of the program in Figure 6.30.

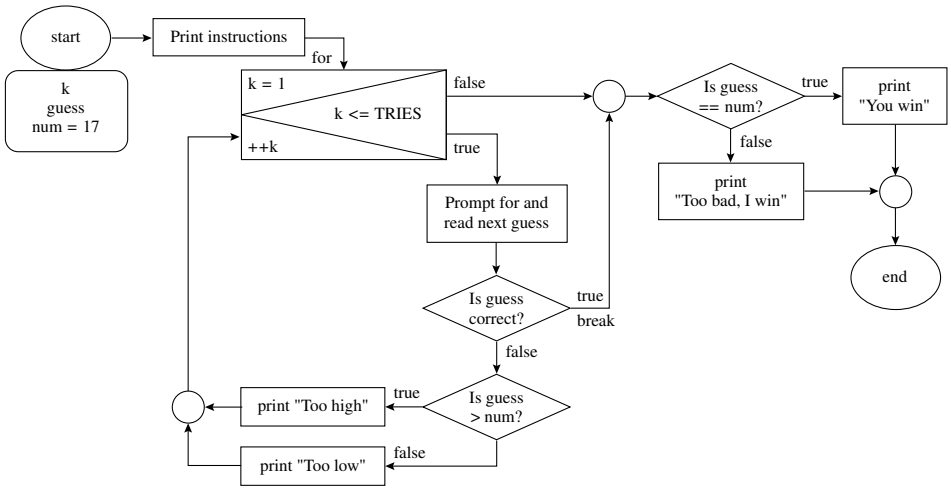


Figure 6.31. A counted sentinel loop.

First box, Figure 6.30: the concealed number. This program is a simplification of an old game that asks the user to guess a concealed number. The computer responds to each guess by telling the user whether the guess was too large, too small, or right on target. In a complete program, the concealed number would be randomly chosen and the number of guesses allowed would be barely enough (or not quite enough) to win the game every time. We give a full version of this program in Chapter 7. In this simplification, we arbitrarily choose 17 as the concealed number. The five guesses allowed are enough to uncover this number if the user makes no mistakes.

Outer box: operation of the loop.

- This `for` loop prompts for, reads, checks, and counts the guesses. It will allow the user up to five tries to enter the correct number. On each trial, the program gives feedback to guide the user in making the next guess. After the fifth unsuccessful try, the loop test will terminate the loop.
- We initialize the loop counter to 1 (rather than the usual 0) because we want to print the counter value as part of the prompt. The computer does not care about the counter values, but users prefer counters that start at 1, not 0.
- We use `<=` to compare the loop counter to the loop limit because the loop variable was initialized to 1, not 0, and we want to execute the loop when the counter equals the number of allotted trials.
- Each guess has three possibilities: It can be correct, too low, or too high. To check for three possibilities, we need two `if` statements. The first `if` statement is in the inner box. If the input matches the concealed number, the break is executed. Control will leave the `for` loop and go to the first statement after the loop. The second `if` statement prints an appropriate comment depending on whether the guess is too high (the true clause) or too low (the false clause). Control then goes back to the top of the loop.

Inner box: There are two ways to leave the loop: either the guess was correct or the guesser was unable to find the hidden number in the number of tries allowed. Here, we test the number of guesses that were used to distinguish between these two cases, then print a success or failure comment. This is a typical way to handle a loop with two exit routes.

The diagram: Figure 6.31.

- A `break` statement is represented by an arrow and a connector, not by a rectangle, diamond, or ellipse. Note the word `break` on the `true` arrow of the `if` condition diamond. The circular connector on the loop's exit arrow is for the `break`.
- The `if...break` statement is the first diamond in the `for` loop. If the guess is not correct, control stays in the loop and enters the `if...else` statement at the end of the loop body. If the guess is correct, control flows out of the loop along the `break` arrow and enters the code that follows the loop.

6.5 What You Should Remember

6.5.1 Major Concepts

- The `while` loop tests the loop exit condition before executing the loop body. The body, therefore, is executed zero or more times.
- The `while` loop is used for sentinel loops, delay loops, processing data sets of unknown size, and data validation loops when it is important to give the user an error comment different from an ordinary prompt.
- The `for` loop implements the same control pattern as the `while` loop but has a different and more compact syntax.
- The `for` loop is used for counted loops and processing any set of data whose exact size or maximum size is known ahead of time.
- The `do...while` loop executes the loop body before performing the loop exit test. Therefore, the body is always executed one or more times.
- The `do...while` statement is used to form query loops that call a `work()` function repeatedly. It also can be used for data validation.
- A nested loop is used to process the rows and columns of a table.
- A `continue` statement within any kind of loop transfers control back to the top of the loop.
- An `if...break` statement can be used to leave any kind of loop but normally is used to leave a `for` loop before the iteration limit is reached.
- Any one or all the expressions in the loop header of a `for` statement can be omitted. If the loop test is omitted, an `if...break` is used to end the loop. This combination is used for data validation loops.
- The `switch` statement has several clauses, called *cases*, each labeled by a constant. At run time, a single value is compared to each constant and the clause corresponding to the matching constant is executed. A `default` case will be executed if none of the case constants match.
- Most cases in `switch` statements (except the last one) end with a `break` statement. If a case has no `break`, the statements for that case `and` the next case will be executed.

6.5.2 Programming Style

Many programs can be improved by eliminating useless work and simplifying nested logic. The resulting code always is simpler and easier to debug and usually is substantially shorter. Some specific suggestions for improving program style follow:

- The golden rule of style is this: Indent your code consistently and properly.
- Line up the first letter of `for`, `while`, or `switch` with the curly bracket that closes the following block of statements. Indent all the statements within the block.
- Use the `switch` statement instead of a series of nested `if...else` statements when the condition being tested is simple enough.
- Do not compute the same expression twice—compute it once and save the answer in a variable. Any time you do an action twice and expect to get the same answer, you create an opportunity for disaster if the program is modified.
- If two statements are logically related, put them near each other in the program. Examples of this principle are

  1. Initialize a loop variable just before the beginning of the loop.
  2. Do the input just before the conditional that tests it.

- Use the `for` loop effectively, putting all initializations and increment steps in the part of the loop.

- When one control structure is placed within the body of another, we call it *nested logic*. For example, a `switch` statement can be nested inside a loop, and a loop can be nested inside one clause of an `if` statement. Most real applications require the use of nested logic. Some generally accepted guidelines are these:

  1. Keep it simple. An `if` statement nested inside another `if` statement inside a loop has excessive complexity. Many times, the `if` statement can be moved outside the loop or the second `if` statement can be eliminated by using a logical operator.
  2. Establish a regular indenting style and stick to it without exception.
  3. Limit the number of levels of nesting to two or three.
  4. If your application seems to require deeper nesting, break up the nesting by defining a separate function that contains the innermost one or two levels of logic.

- To improve efficiency, move everything possible outside of a loop. For example, when you must compute the average of some numbers, use the loop to count the numbers and add them to a total. Do not do the division within the loop. Moving actions out of the loop frequently shortens the code; it always simplifies it and makes it more efficient. In poorly written programs, "fat loops" account for many of the extra lines. Take advantage of any special properties of the data, such as data that may be sorted, may have a sentinel value on the end, or may be validated by a prior program step and not need validation again when processed.
- During the primary debugging phase of an application, every loop should contain some screen output. This allows the programmer to monitor the program's progress and detect an infinite loop or a loop that is executed too many or too few times. Debugging output lines should print the loop variable, any input read within the loop, and any totals or counters changed in the loop.

6.5.3 Sticky Points and Common Errors

Semicolons in the wrong places. The header of a `for` loop or the condition clause of a `while` loop or an `if` statement is, normally, *not* followed by a semicolon.

Off-by-one errors. The programmer must take care with every loop to make sure it repeats the correct number of times. A small error can cause one too many or one too few iterations. Every loop should be tested carefully to ensure that it executes the proper number of times. In some counted loops, the loop counter is used as a subscript or as a part of a computation. In these loops, the results may be wrong even when the loop is repeated the correct number of times. This kind of error happens when the value of the loop counter is **off by one** because it is not incremented at the correct time in relation to the expression or expressions that use the counter's value. Consider these two counted loops:

```
for (sumj = j = 0; j < 10; ++j)
    sumj += j;                  // Sum from 0 to 9

// ------------------------------------------------
sumk = k = 0;
while (k < 10) {                // Sum from 1 to 10
    ++k;
    sumk += k;
}
```

These loops are very similar, but `k` is incremented before adding it to `sumk` and `j` is incremented after adding it to the sum. A programmer must decide which timing pattern is correct and be careful to write the correct form.

Infinite loops. The `for` loop with no loop test sometimes is called an *infinite loop*, although most such loops contain an `if...break` statement that stops execution under appropriate conditions. Such loops are useful tools. However, a real infinite loop is not useful and should be avoided. Such loops are the result of forgetting to include an update step in the loop body. (In a counted loop, the update step increments the loop variable. In an input-controlled loop, it reads a new value for the loop variable.) During the construction and debugging phases of a program, it is a good idea to put a `puts()` statement in every loop so that you can easily identify an infinite loop when it occurs.

Nested logic. When using nested logic, the programmer must know how control will flow into, through, and out of the unit. Statements such as `break` and `continue` affect the flow of control in ways that are simple when single units are considered but become complex when control statements are nested. Be sure you understand how your control statements interact.

6.5.4 Where to Find More Information

- Strings will be introduced in Chapter 12; a sentinel loop that processes a string is shown in Figure 12.12.
- Chapter 19 presents the code for quicksort, one of the best sorting algorithms. Arrays are used with sentinel loops are used in that program.
- Chapter 22 presents two kinds of linked lists; both are typically processed using a sentinel loop.
- Chapter 8, Figure ex-char-work, we show how to use a query loop with character inputs (y/n), which makes a better human interface than numeric responses (1/0).
- Figure 6.31 implements a simple version of a familiar guessing game. We revisit and elaborate on this game in the Random Numbers program on the website.
- The `switch` statement is used with an enumerated type in Figure 13.5. Figures 12.24 and 15.21 show a very typical use of `switch` to process single-character selections from a menu-interface.
- Appendix **??** describes the properties and usage of the conditional operator, `? :`, which is the only operator in `C` that has three operands. The first operand (before the `?` is a condition. The second operand gives an expression to evaluate if the condition is true. If it is false, the expression after the `:` is evaluated. The result of the conditional operator is the result of whichever expression was evaluated.

6.5.5 New and Revisited Vocabulary

These terms and concepts have been defined or expanded in the chapter:

for loop loop header	trip count	search loop
loop variable	input validation loop	input-controlled loop
sentinel loop	nested loops	premature exit
sentinel value	delay loop	avoiding `break`
repeat query	busy wait	constant expression
counted loop	flexible-exit loop	off-by-one error
	infinite `for` loop	

The following C keywords were discussed in this chapter:

`for` loop	`break` statement	`switch` statement
`,` (comma operator)	`if...break` statement	`case` statement
`do...while` loop	`continue` statement	`default` clause

6.6 Exercises

6.6.1 Self-Test Exercises

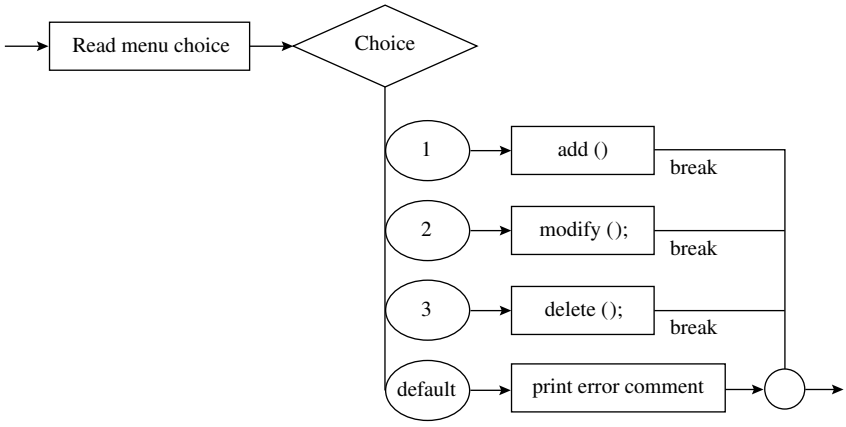
1. The following program contains a loop. What is its output? Rewrite the loop using `for` instead of `while`. Make sure the output does not change.

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

    sum = k = 0;
    while (k < 10) {
        sum += k;
        ++k;
    }
    printf( "A. %i %i \n", k, sum );
}
```

2. The following program contains a loop. What is its output? Rewrite the loop using `while` instead of `do...while`. Make sure the output does not change.

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

    printf( " Please enter an exponent >= 0: " );
    scanf( "%i", &k );
    sum = 1;
    do {
        if (k > 0) sum = 2*sum;
        --k;
    } while (k > 0);
    printf( "B. %i \n", sum );
}
```

3. Explain the fundamental differences between a series of `if...else` statements and a `switch` statement. Under what conditions would you use a `switch` statement? Give an example of a problem for which you could not use a `switch` statement.
4. Explain the fundamental differences between a `while` loop and a `do...while` loop. In what situation would you use each?
5. Given the following fragment of a flow diagram, write the code that corresponds to it and define any necessary variables:



6. Draw a flow diagram for the cash register program in Figure 6.9.
7. Rewrite the following `switch` statement as an `if...else` sequence. (Write code, not a flowchart.)

```
switch (k) {
    case 2:
    case 12:  puts( "You lose" ); break;
    case 7:
    case 11:  puts( "You win" ); break;
    default:  puts( "Try again" );
    }
```

8. Analyze the following loop and draw a flow diagram for it. Then trace the execution of the loop using the initial values shown. Use a storage diagram to show the succession of values stored in each variable. Finally, show the output, clearly labeled and in one place.

```
for (k = 0,j = 1; j < 3; j++)  {
    k += j;
    printf( "\t %i\t %i\n", j, k );
}
printf( "\t %i\t %i\n", j,k );
```

9. What does each of the following loops print? They are supposed to print the numbers from 1 to 3. What is wrong with them?

   (a)
   ```
   for(k = 0; k < 3; ++k) printf( "k = %i", k );
   ```

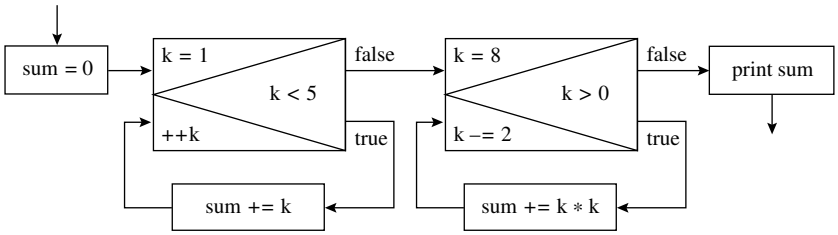
   (b)
   ```
   k = 1;
   do {
       printf( "k = %i", k );
       k++;
   } while (k < 3);
   ```

6.6.2 Using Pencil and Paper

1. Explain the fundamental differences between a counted loop and a sentinel loop. What C statement would you use to implement each?
2. Explain the fundamental similarity between a `while` loop and a `for` loop. In what situation would you use each?
3. Rewrite the following `if...else` sequence as a `switch` statement. Write code, not a flowchart.

   ```
   if (i == 0) puts( "bad" );
   else if (i >= 1 && i < 3) puts( "better" );
   else if (i == 4 || i == 5) puts( "good" );
   else puts( "sorry" );
   puts("------"):
   ```

4. Draw a flow diagram for the cash register program in Figure 6.23.
5. Given the following fragment of a flow diagram, write the code that corresponds to it and define any necessary variables. What number will be printed by the last box?



6. The following program contains a loop. What is its output? Rewrite the loop using `do...while` instead of `for`. Keep the output the same.

   ```
   #include <stdio.h>
   int main ( void )
   {   int k, sum;
       for ( sum = k = 0; k < 5; ++k ) sum += k;
       printf( "C. %i %i \n", k, sum );
   }
   ```

7. The following program contains a loop. What is its output? Rewrite the loop without using `break`. Keep the output the same.

```
#include <stdio.h>
int main ( void )
{   int k, sum;
    for (sum = k = 1 ; k < 10; k++) {
        sum *= k;
        if (sum > 10*k) break;
    }
    printf( "D. %i %i \n", k, sum );
}
```

8. What does the following loop print? It is supposed to print out the numbers from 1 to 3. What is wrong with it?

```
for(k=1; k<=3; ++k);
    printf( "k=%i", k );
```

9. Analyze each loop that follows and draw a flow diagram for it. Then trace execution of the loop using the initial values shown. Use a storage diagram to show the succession of values stored in each variable. Finally, show the output from each exercise, clearly labeled and in one place.

   (a)
   ```
   k=2;
   do { printf("\t %i\n", k); --k; } while (k>=0);
   ```

   (b)
   ```
   j=k=0;
   while (j<3){ k+=j; printf("\t %i\t %i\n", j, k); ++j; }
   ```

10. Given the following code,

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

   for (j = 1; j < 3; j++) {
       for (k = 0; k < 5; k += 2) {
           if (k != 4) printf( " %i, %i ", k, j );
           else k--;
       }
       printf( "\n" );
       if (j % 2 == 1) printf( ">>> %i , %i <<<\n", j, k );
   }
}
```

   (a) Draw a flowchart that corresponds to the program.

   (b) Using a table like the one that follows, trace the execution of the program. Show the initial values of `j` and `k`. For each trip through a loop, show how the values of `j` and `k` change and what is displayed on the screen. Draw a vertical line between the columns that correspond to each trip through the inner loop.

j	
k	
Output	

6.6.3 Using the Computer

1. Sum a series.

   Write a program that uses a `for` loop to sum the first 100 terms of the series $1/x^2$. Use Figure 6.14 as a guide. Develop a test plan and carry it out. Turn in the source code and output of your program when run on the test data.

2. Sum a function.

   Write a function with parameter x that will compute the value of $f(x) = (3 \times x + 1)^{\frac{1}{2}}$.

   Write a main program that sums $f(x)$ from $x = 0$ to x =1,000 in increments of 2. Use a `for` loop. Print the result.

3. Find the best.

   Write a program that will allow an instructor to enter a series of exam scores. After the last score, the instructor should enter a negative number as a signal that there is no more input. Print the average of all the scores and the highest score entered.

4. Gas prices.

   The example at the end of Chapter 3 (Figure 3.18) is a program that converts a Canadian gas price to an equivalent U.S. gas price. This program does the calculation for only one price. Modify it so that it can convert a series of prices, as follows:

   (a) Remove the price-per-liter input, computation, and output from the main program and put them in a separate function, named `convert()`.
   (b) In place of this code in the main program, substitute a query loop that will allow you to enter a series of Canadian prices. For each, call `convert()` to do the work.

5. An increasing voltage pattern.

   Write a program that will calculate and print out a series of voltages.

   (a) Prompt the user for a value of $v$max and restrict it to the range `12 <=` $v$max `<= 24`. Let time `t` start at `0` and increase by 1 at each step until the voltage $v > 95\%$ of $v$max; `v` is a function of time according to the following formula. Print the time and voltage at each step.

   $$v = v\text{max} \times \left(1 - \mathrm{e}^{(-0.1 \times t)}\right)$$

   (b) Add a delay loop so that the voltage output is timed more slowly. See Figure 6.18.

6. Sine or cosine?

   (a) Write a double→double function, `f()`, with one `double` parameter, `x`, that will evaluate either $f_1(x) = x^2 \sin(x) + e^{-x}$ if $x \leq 1.0$ or $f_2(x) = x^3 \cos(x) - \log(x)$ if $x > 1.0$, and return the result. Write a prototype that can be included at the top of a program.
   (b) Start with the main program in Figure 6.11. Modify the program title and write a new `work()` function that will input a value for `x`, call `f()` using the value of `x`, and output the result.
   (c) Design a test plan for your program.
   (d) Incorporate all the pieces of your program in one file and compile it. Then carry out the test plan to verify the program's correctness.

7. A voltage signal.

   An experiment will be carried out repeatedly using an ac voltage signal. The signal is to have one of three values, depending on the time since the beginning of the experiment:

   (a) For time $t < 1$, volts$(t) = 0.5 \times \sin(2t)$.
   (b) For time $1.0 \leq t \leq 10.0$, volts$(t) = \sin(t)$.
   (c) For time $t > 10.0$, volts$(t) = \sin(10.0)$.

(a) Write a function named `volts()` with t as a parameter that will calculate and return the correct voltage. Use the function in Figure 5.18 as a guide.

(b) Using Figure 6.11 as a guide, write a main program that starts at time 0. Use a loop to call the `volts()` function and output the result. Increment the time by 0.5 after each repetition until the time reaches 12. Print the results in the form of a neat table.

8. Rolling dice.
Over a large number of trials, a "fair" random number generator will return each of the possible values approximately an equal number of times. Therefore, in each set of 60 trials, if values are generated in the range $1 \ldots 6$, there should be about 10 ones and 10 sixes.

(a) Using the loop in Figure 5.26 as a guide, write a function that will generate 60 random numbers in the range $1 \ldots 6$. Use `if` statements to count the number of times 1 and 6 each turns up. At the end, print out the number of ones and the number of sixes that occurred.

(b) Following the example in Figure 6.11, write a main program that will call the function from part (a) 10 times. The main program should print table headings; the function will print each set of results under those headings. Look at your results; are the numbers of ones and sixes what you expected? Try it again. Are the results similar? Can you draw any conclusions about the quality of the algorithm for generating random numbers?

9. Prime number testing.
A prime number is an integer that has no factors except itself and 1. The first several prime numbers are 2, 3, 5, 7, 11, 13, 17, and 19. Very large prime numbers have become important in the field of cryptography. The original public-key cryptographic algorithm is based on the fact that there is no fast way to find the prime factors of a 200-digit number that is the product of two 100-digit prime numbers. In this program, you will implement a simple but very slow way to test whether a number is prime.

One method of testing a number N for primality, is by calculating `N % x`, where x is equal to every prime number from 2 to $R = \sqrt{N}$. If any of these results equals 0, then N is not a prime. We can stop testing at $\sqrt{N}$, since if N has any factor greater than R, it also must have a factor less than or equal to R. Unfortunately, keeping track of a list of prime numbers requires techniques that have not yet been presented. However, a less efficient method is to calculate `N % x` for $x = 2$ and every odd number between 3 and $\sqrt{N}$. Some of these numbers will be primes, most will not. But if any one value divides N evenly, we know that N is not a prime.

Write a function that enters an integer N to test and prints the word `prime` if it is a prime number or `nonprime` otherwise. Write a main program with a query loop to test many numbers.