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 B. 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

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.

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*.

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)))
```

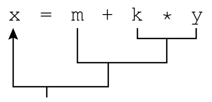


Figure 4.2. Applying precedence.

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.

```
3 / z * 10 % 3
(3 / z) * 10 % 3
((3 / z) * 10) % 3
(((3 / z) * 10) % 3)
```

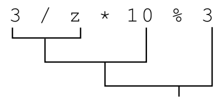


Figure 4.3. Applying associativity.

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.

[3]Complications of evaluation order are covered in Sections 4.3 and 4.5.3 and Appendix D.

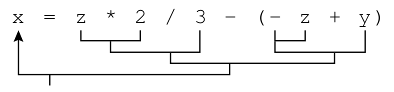


Figure 4.4. Applying the parsing rules.

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.)
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.

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.

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`.

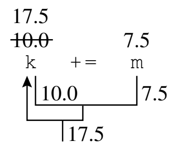


Figure 4.6. Parse tree for an assignment combination.

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 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.

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 k;          /* Loop counter. */
    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);

    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.

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

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.

a process that decreases `n` until the terminal condition, `n` equals 0, occurs. More loop types are discussed in Chapter 6.

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

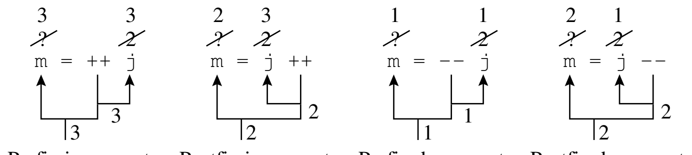


Figure 4.9. Increment and decrement trees.

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 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.

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

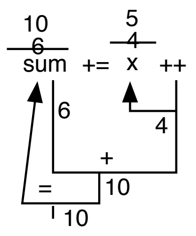


Figure 4.10. Evaluating an increment expression.

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.

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,

[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.

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.

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 Relational Operators

The **relational operators** (`==, !=, <, >, <=`, and `>=`), with their meanings and precedence values, are listed in Figure 4.11. 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.

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

[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.

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 4.11. Relational operators.

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 conitional 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.

4.4.1 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

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 4.12. The size of things.

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 4.12 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 4.12 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`.

4.5 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.

4.5.1 True and False

`C` has no special, different data type for truth values. Truth values are integers, and every integer has a **truth value**. The integers 1 and 0 are the "standard true values"; that is, when `C` generates a true value as the result of a comparison or logical operator, that value is always 1 or 0.

In addition, every other data value can be interpreted as 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.

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. To make this style convenient, the following two definitions can be included at the top of your program:[7]

```
#define false 0
#define true  1
```

You can then use these defined constants directly in comparisons, expressions and assignment statements where the logical values are desired.

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

[7]It is not necessary to do this in recent `C` compilers or in `C++`, because the symbols are defined by the `C++` standard.

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.

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).

4.5.2 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.[8] 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.

4.5.3 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

[8] 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.

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.

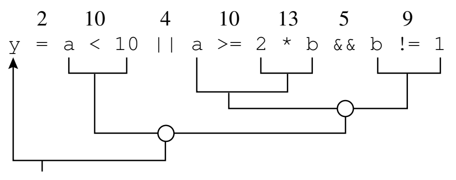


Figure 4.15. Parsing logical operators.

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

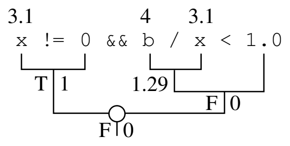
- To evaluate an `&&` operator, first evaluate its *left* operand. This operand might be a simple variable or literal, or it might be a complicated expression.
- Look at the truth value. If it is `false`, return 0 as the answer to the `&&` operation and skip the next step.
- Otherwise, we do not yet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

can be used only for logical operators. Basically, the left operand of the logical operator is evaluated then 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.

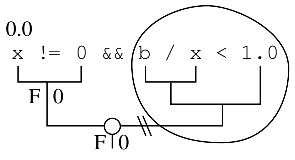
We evaluate a logical expression twice with different operand values.

(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

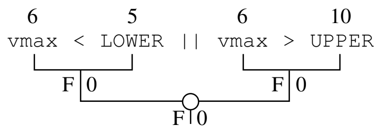
- To evaluate an `||` operator, first evaluate its *left* operand. This operand might be a simple variable or literal or it might be a more complicated expression.
- Look at the truth value. If it is `true`, return 1 as the answer to the `||` operation and skip the next step.
- Otherwise, we do not yet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

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.

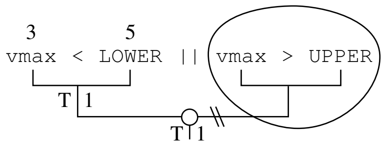
We evaluate a logical expression twice with different operand values. The first time, the entire expression is evaluated. The second time, a shortcut is taken.

(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.

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. 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.

4.6 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.

4.6.1 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.[9] 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, 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.

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

x	y	`x % y`	x	y	`x % y`
10	3	1	3	10	3
9	3	0	3	9	3
8	3	2	7	2873	7
7	3	1	7	7	0
6	3	0	7	1	0
5	3	2	7	0	Undefined

Figure 4.20. The modulus operation is cyclic.

- 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.

4.6.2 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.[10] 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

[10]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.

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:

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

Figure 4.22. Positional notation and base conversion.

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

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.

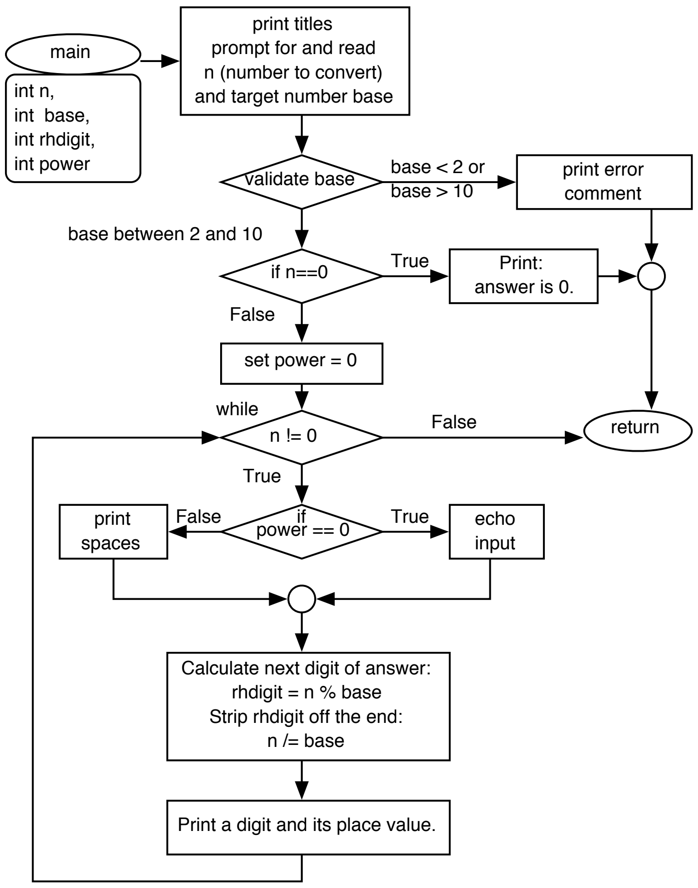


Figure 4.24. A flow diagram for the base conversion.

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

  ```
  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

        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.

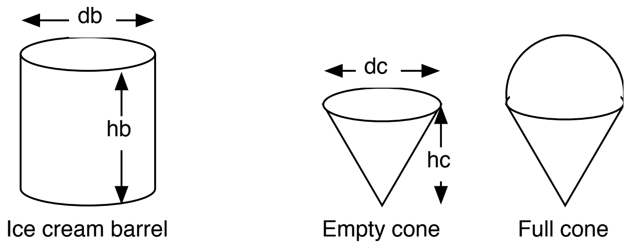
4.7 Techniques for Debugging

4.7.1 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?

```
#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?

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.

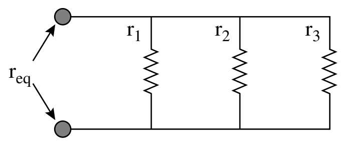
- We implement the separate formulas instead of the combined formula to make both programming and debugging easier and less error prone. It is a little more work for the computer this way, but in almost all circumstances, simplicity is better than complexity.
- We compute each formula and store its result 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.
- 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.

```
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.
```

Sample output:

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:

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



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.

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.

4.7.2 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`.

```
/* ----------------------------------------------------------------
** 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.

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 );
```

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.

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.
```

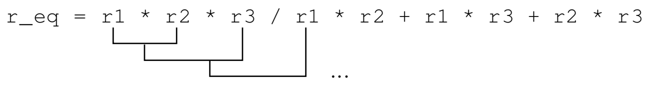


Figure 4.29. Finding an expression error.

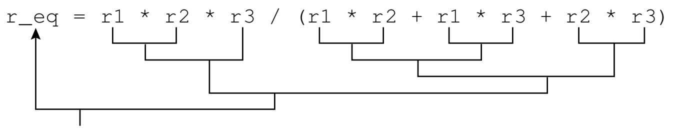


Figure 4.30. Parsing the corrected expression.

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:

```
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.

4.8 What You Should Remember

4.8.1 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.

4.8.2 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.

4.8.3 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:***

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.

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

4.8.4 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.

4.8.5 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 D. 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 D
- 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.

4.9 Exercises

4.9.1 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;`

4.9.2 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 );
}
```

4.9.3 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. 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;
}
```

4. 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.

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.