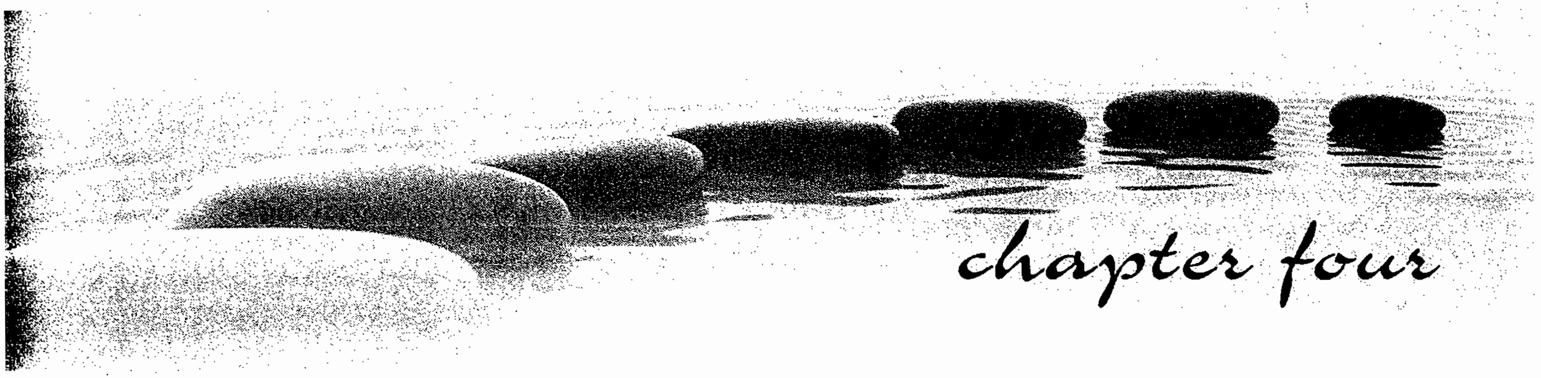


Selection Structures: `if` and `switch` Statements

Chapter Objectives

- To become familiar with the 3 kinds of control structures: sequence, selection, repetition and to understand compound statements
- To learn how to compare numbers, characters, and strings
- To learn how to use the relational, equality, and logical operators to write expressions that are type bool (true or false)
- To learn how to write selection statements with one and two alternatives using the if statement
- To learn how to implement decisions in algorithms using the `if` statement
- To understand how to select among more than two alternatives by nesting `if` statements
- To learn how to use the switch statement as another technique for selecting among multiple alternatives

THIS BOOK TEACHES a disciplined approach that results in programs that are easy to read and less likely to contain errors. An important element of this technique is the use of top-down design to divide an algorithm into individual steps. We can use a small number of control structures to code the individual algorithm steps.

At this point, we've used just one of these control mechanisms, sequence. In this chapter and the next, we'll introduce two other important control structures: selection and repetition.

This chapter will also show how to use the C++ `if` and `switch` statements to select one group of statements for execution from several alternative groups. These control statements will allow a program to evaluate its data and to select a path of execution to follow based on the data. The chapter also discusses conditions and logical expressions, which the `if` statement relies on.

4.1 Control Structures

control structure
A combination of individual instructions into a single logical unit with one entry point and one exit point that regulates the flow of execution in a program.

compound statement
A group of statements bracketed by { and } that are executed sequentially.

Control structures regulate the flow of execution in a program or function, combining individual instructions into a single logical unit with one entry point and one exit point. There are three categories of control structures for controlling execution flow: (1) *sequence,* (2) *selection,* and (3) *repetition.* Until now, we've been using only sequential flow. A **compound statement,** written as a group of statements bracketed by { and }, is used to specify sequential control.

$$\begin{array}{l} \{ \\ \quad \text{statement}_1; \\ \quad \text{statement}_2; \\ \quad \bullet \\ \quad \bullet \\ \quad \bullet \\ \quad \text{statement}_n; \\ \} \end{array}$$

Sequential flow means that each statement is executed in sequence, starting with *statement*$_1$, followed by *statement*$_2$, and so on through *statement*$_n$.

selection control
A control structure that chooses among alternative program statements.

This chapter describes how to write algorithms and programs that use **selection control,** a control structure that enables the program to execute one of several alternative statements or groups of statements. With selection control, flow doesn't necessarily follow from one statement to the next as in sequential control, but may skip some statements and execute others based on selection criteria.

4.2 Logical Expressions

In C++, the primary selection control structure is the `if` statement. We can use an `if` statement to select from several alternative executable statements. For example, the `if` statement

```
if (weight > 100.00)
  shipCost = 10.00;
else
  shipCost = 5.00;
```

selects one of the two assignment statements listed by testing the condition `weight > 100.00`. If the condition is true (`weight` is greater than `100.00`), it selects the statement `shipCost = 10.00;`. If the condition is false (`weight` is not greater than `100.00`), it selects the statement `shipCost = 5.00;`. It is never possible for both assignment statements to execute after testing a particular condition.

In order to write `if` statements, we must first understand how to write the conditions that the `if` statement relies on to make a selection.

Logical Expressions Using Relational and Equality Operators

A program selects among alternative statements by testing the value of key variables. In C++, **logical expressions**, or **conditions**, are used to perform such tests. Each logical expression has two possible values, `true` or `false`. Such an expression is called a condition because it establishes a criterion for either executing or skipping a group of statements.

logical expression (condition)
An expression that is either true or false.

Most conditions that we use to perform comparisons will have one of these forms:

variable	*relational-operator*	*variable*
variable	*relational-operator*	*constant*
variable	*equality-operator*	*variable*
variable	*equality-operator*	*constant*

Table 4.1 lists the relational and equality operators. Be careful not to confuse the assignment operator (single =) with the equality operator (double ==).

Table 4.1 Relational and Equality Operators

Operator	Meaning	Type
<	less than	relational
>	greater than	relational
<=	less than or equal to	relational
>=	greater than or equal to	relational
==	equal to	equality
!=	not equal to	equality

EXAMPLE 4.1

Table 4.2 shows some sample conditions in C++. Each condition is evaluated assuming these variable and constant values:

`x`	`power`	`maxPower`	`y`	`item`	`minItem`	`momOrDad`
`-5`	`1024`	`1024`	`7`	`1.5`	`-999.0`	`'m'`

Table 4.2 Sample Conditions

Operator	Condition	English Meaning	Value
`<=`	`x <= 0`	`x` less than or equal to 0	true
`<`	`power < maxPower`	`power` less than `maxPower`	false
`>=`	`x >= y`	`x` greater than or equal to `y`	false
`>`	`item > minItem`	`item` greater than `minItem`	true
`==`	`momOrDad == 'm'`	`momOrDad` equal to `'m'`	true
`!=`	`num != sentinel`	`num` not equal to `sentinel`	false

Logical Expressions Using Logical Operators

With the three logical operators—`&&` (and), `||` (or), `!` (not)—we can form more complicated conditions or compound logical expressions. Examples of logical expressions formed with the `&&` (and) and `||` (or) operators are

```
(salary < minSalary) || (dependents > 5)
(temperature > 90.0) && (humidity > 0.90)
```

The first logical expression determines whether an employee is eligible for special scholarship funds. It evaluates to `true` if *either* the condition

```
(salary < minSalary)
```

or the condition

```
(dependents > 5)
```

is true. The second logical expression describes an unbearable summer day, with temperature and humidity both in the nineties. The expression evaluates to true only when *both* conditions are true.

The operands of `||` and `&&` are themselves logical expressions and each has a type `bool` value (`true` or `false`). The compound logical expression also has a type `bool` value. None of the parentheses used in these expressions are required; they are used here just for clarity.

The third logical operator, `!` (not), has a single type `bool` operand and yields the **logical complement, or negation,** of its operand (that is, if the variable `positive` is `true`, `!positive` is `false` and vice versa). The logical expression

logical complement (negation)
The logical complement of a condition has the value true when the condition's value is false; the logical complement of a condition has the value false when the condition's value is true.

```
winningRecord && (!probation)
```

manipulates two `bool` variables (`winningRecord, probation`). A college team for which this expression is true has a winning record and is not on probation, so it may be eligible for the postseason tournament. Note that the expression

```
(winningRecord == true) && (probation == false)
```

is logically equivalent to the one above; however, the first one is preferred because it's more concise and more readable.

Simple expressions such as

```
x == y
x < y
```

can be negated by applying the unary operator `!` to the entire expression.

```
!(x == y)
!(x < y)
```

However, using the logical complement of the relational operator often works just as well in these situations:

```
x != y
x >= y
```

The best time to use the `!` operator is when you want to reverse the value of a long or complicated logical expression that involves many operators. For example, expressions such as

```
(salary < minSalary) || (numberDependents > 5)
```

are most easily negated by enclosing the entire expression in parentheses and preceding it with the `!` operator.

```
!((salary < minSalary) || (numberDependents > 5))
```

Table 4.3 shows that the `&&` operator (and) yields a true result only when both its operands are true. Table 4.4 shows that the `||` operator (or) yields a false result only when both its operands are false. The `not` operator `!` has a single operand; Table 4.5 shows that the `not` operator yields the logical complement, or negation, of its operand (that is, if `flag` is `true`, `!flag` is `false` and vice versa). Remember to use logical operators only with logical expressions.

Operator Precedence

The precedence of an operator determines its order of evaluation in an expression. Table 4.6 shows the precedence from highest to lowest of all C++ operators you've seen so far. The unary operators (including `!`) have the highest precedence followed by the arithmetic, relational, equality operators, and then the binary logical operators. To prevent errors and to clarify the meaning of expressions, use parentheses freely.

Table 4.3 `&&` Operator

Operand_1	Operand_2	Operand_1 `&&` Operand_2
true	true	true
true	false	false
false	true	false
false	false	false

Table 4.4 `||` Operator

Operand_1	Operand_2	Operand_1 `||` Operand_2
true	true	true
true	false	true
false	true	true
false	false	false

Table 4.5 `!` Operator

Operand	`!`Operand
true	false
false	true

Table 4.6 Operator Precedence

Operator	Precedence	Description
`!`, `+`, `-`	Highest	Logical not, unary plus, unary minus
`*`, `/`, `%`	↓	Multiplication, division, modulus
`+`, `-`	↓	Addition, subtraction
`<`, `<=`, `>=`, `>`	↓	Relational inequality
`==`, `!=`	↓	Equal, not equal
`&&`	↓	Logical and
`||`	↓	Logical or
`=`	Lowest	Assignment

The expression

```
x < min + max
```

involving the `float` variables `x`, `min`, and `max` is interpreted correctly in C++ as

```
x < (min + max)
```

because + has higher precedence than <. The expression

```
min <= x && x <= max
```

is also correct, but providing the extra parentheses

```
(min <= x) && (x <= max)
```

makes the expression clearer.

Because `!` has higher precedence than ==, C++ incorrectly interprets the expression

```
!x == y
```

as

```
(!x) == y
```

To avoid this error, insert the parentheses where needed:

```
!(x == y)
```

EXAMPLE 4.2

Expressions 1 to 4 below contain different operands and operators. Each expression's value is given in the corresponding comment, assuming `x`, `y`, and `z` are type `float`, `flag` is type `bool`, and the variables have the values:

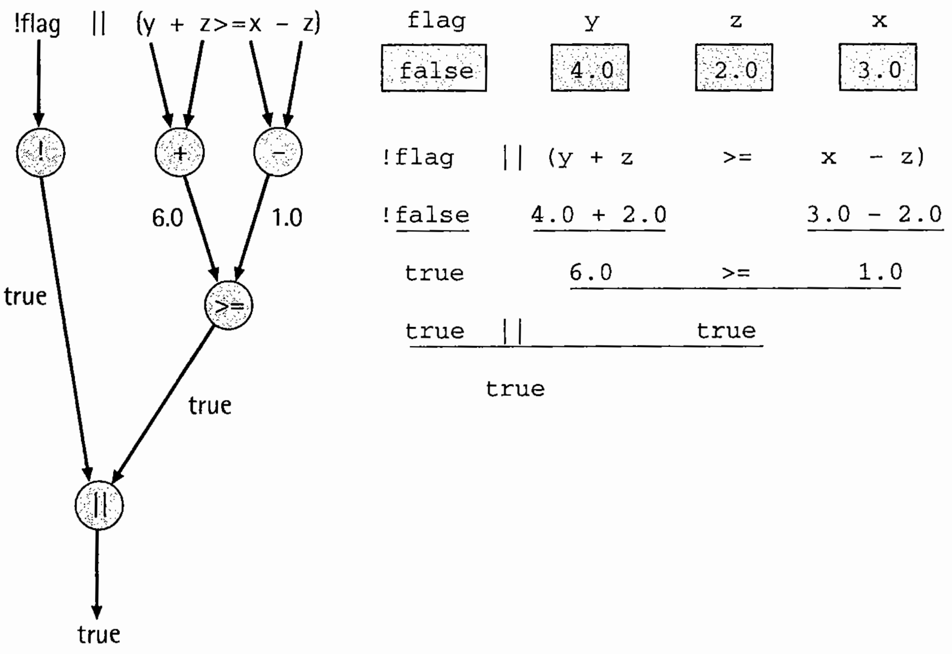
x	y	z	flag
3.0	4.0	2.0	false

```
1. !flag                          // !false is true
2. x + y / z <= 3.5               // 5.0 <= 3.5 is false
3. !flag || (y + z >= x - z)      // true || true is true
4. !(flag || (y + z >= x - z))    // !(false || true) is false
```

Figure 4.1 shows the evaluation tree and step-by-step evaluation for expression 3.

Writing Conditions in C++

To solve programming problems, you must convert conditions expressed in English to C++. Many algorithm steps require testing to see if a variable's value is within a specified range of values. For example, if `min` represents the lower bound of a range of values and `max` represents the upper bound (`min` is less than `max`), the expression



```
!flag    || (y + z      >=    x   - z)
!false      4.0 + 2.0         3.0 - 2.0
 true          6.0      >=       1.0
 true   ||              true
             true
```

Figure 4.1 Evaluation tree and step-by-step evaluation of `!flag || (y + z >= x - z)`

```
(min <= x) && (x <= max)
```

tests whether x lies within the range min through max, inclusive. In Figure 4.2, this range is shaded. The expression is true if x lies within this range and false if x is outside the range.

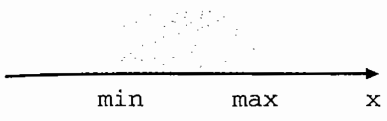


Figure 4.2 Range of true values for `(min <= x) && (x <= max)`

EXAMPLE 4.3

Table 4.7 shows some English conditions and their corresponding C++ expressions. Each expression is evaluated assuming x is `3.0`, y is `4.0`, and z is `2.0`.

The first logical expression shows the C++ code for the English condition "x and y are greater than z." You may be tempted to write this as

```
x && y > z        // invalid logical expression
```

However, if you apply the precedence rules to this expression, you quickly see that it doesn't have the intended meaning. Also, the type `float` variable x is an invalid operand for the logical operator `&&`.

The third logical expression shows the C++ code for the mathematical relationship $z \le x \le y$. The boundary values, `2.0` and `4.0`, are included in the range of x values that yield a true result.

The last table entry shows a pair of logical expressions that are true when x is outside the range bounded by z and y. We get the first expression in the pair by complementing the expression just above it. The second expression states that x is outside the range if z is larger than x or x is larger than y. In

Table 4.7 English Conditions as C++ Expressions

English Condition	Logical Expression	Evaluation
x and y are greater than z	`(x > z) && (y > z)`	`true && true` is `true`
x is equal to 1.0 or 3.0	`(x == 1.0) || (x == 3.0)`	`false || true` is `true`
x is in the range z to y, inclusive	`(z <= x) && (x <= y)`	`true && true` is `true`
x is outside the range z to y	`!(z <= x && x <= y)`	`!(true && true)` is `false`
	`(z > x) || (x > y)`	`false || false` is `false`

Figure 4.3, the shaded areas represent the values of x that yield a true result. Both y and z are excluded from the set of values that yield a true result.

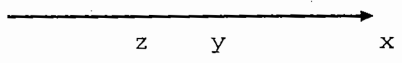


Figure 4.3 Range of true values for `(z > x) || (x > y)`

Comparing Characters and Strings

In addition to comparing numbers, it's also possible to compare characters and strings of characters using the relational and equality operators. Several examples of such comparisons are shown in Table 4.8.

In writing such comparisons, you can assume that both the uppercase and lowercase letters appear in alphabetical order, and that the *digit characters* are ordered as expected: `'0' < '1' < '2' < . . . < '9'`. For the relationships among other characters (such as `'+'`, `'<'`, `'!'`, etc.) or between two characters not in the same group (for example, `'a'` and `'A'`, or `'B'` and `'b'`, or `'c'` and `'A'`), see Appendix A.

If you include library `string`, you can compare `string` objects (see the last four examples in Table 4.8). C++ compares two strings by comparing corresponding pairs of characters in each string, starting with the leftmost pair. The result is based on the relationship between the first pair of different characters (for example, for `"acts"` and `"aces"`, the result depends on characters `t` and `e`). The last comparison involves a string (`"aces"`) that begins with a shorter *substring* (`"ace"`). The substring `"ace"` is considered less than the longer string `"aces"`. We'll say more about character comparisons in Chapter 7.

Table 4.8 Examples of Comparisons

Expression	Value
`'a' < 'c'`	true
`'X' <= 'A'`	false
`'3' > '4'`	false
`('A' <= ch) && (ch <= 'Z')`	true if `ch` contains an uppercase letter; otherwise false
`"XYZ" <= "ABC"`	false (`X <= A`)
`"acts" > "aces"`	true (`t > e`)
`"ace" != "aces"`	true (strings are different)
`"ace" < "aces"`	true (shorter string is smaller)

Boolean Assignment

Assignment statements can be written to assign a type `bool` value to a `bool` variable. If `same` is type `bool`, the statement

```
same = true;
```

assigns the value `true` to `same`. Since assignment statements have the general form

```
variable = expression;
```

you can use the statement

```
same = (x == y);
```

to assign the value of the logical expression `(x == y)` to `same`. The value of `same` is `true` when `x` and `y` are equal; otherwise, `same` is `false`.

EXAMPLE 4.4

The following assignment statements assign values to two type `bool` variables, `inRange` and `isLetter`. Variable `inRange` gets `true` if the value of `n` is between −10 and 10; variable `isLetter` gets `true` if `ch` is an uppercase or a lowercase letter.

```
inRange =  (n > -10) && (n < 10);
isLetter = (('A' <= ch) && (ch <= 'Z')) ||
           (('a' <= ch) && (ch <= 'z'));
```

The expression in the first assignment statement is `true` if `n` satisfies both conditions listed (`n` is greater than 210 and `n` is less than 10); otherwise, the expression is `false`. The expression in the second assignment statement uses the logical operators `&&`, `||`. The first subexpression (before `||`) is `true` if `ch` is an uppercase letter; the second subexpression (after `||`) is true if `ch` is a lowercase letter. Consequently, `isLetter` gets `true` if `ch` is either an uppercase or lowercase letter; otherwise, `isLetter` gets `false`.

EXAMPLE 4.5

The statement below assigns the value `true` to `even` if `n` is an even number:

```
even = (n % 2 == 0);
```

Because all even numbers are divisible by 2, the remainder of `n` divided by `2` (the C++ expression `n % 2`) is `0` when `n` is an even number. The expression in parentheses above compares the remainder to `0`, so its value is `true` when the remainder is `0` and its value is `false` when the remainder is nonzero.

Writing bool Values

Most logical expressions appear in control structures, where they determine the sequence in which C++ statements execute. You'll rarely have a need to read bool values as input data or display bool values as program results. If necessary, however, you can display the value of a bool variable using the output operator <<. If flag is false, the statement

```
cout << "The value of flag is " << flag;
```

displays the line

```
The value of flag is 0
```

If you need to read a data value into a type bool variable, you can represent the data as an integer; use 0 for false and 1 for true.

Using Integers to Represent Logical Values

Earlier C++ compilers did not implement type bool. Instead they used the int data type to model type bool data. Internally C++ uses the integer value 0 to represent false and any nonzero integer value (usually 1) to represent true. A logical expression that evaluates to 0 is considered false; a logical expression that evaluates to nonzero is considered true.

EXERCISES FOR SECTION 4.2

Self-Check

1. Assuming x is 15 and y is 10, what are the values of the following conditions?

   a. `x != y` **b.** `x < y` **c.** `x >= (y - x)` **d.** `x == (y + x - y)`

2. Evaluate each expression below if a is 10, b is 12, c is 8, and flag is false.

   a. `(c == (a * b)) || !flag`

   b. `(a != 7) && flag || ((a + c) <= 20)`

   c. `!(b <= 12) && (a % 2 == 0)`

   d. `!((a < 5) || (c < (a + b)))`

3. Evaluate the following logical expressions. Assume x and y are type float and p, q, and r are type bool. The value of q is false. Why is it not necessary to know the value of x, y, p, q, or r?

   a. `x < 5.1 || x >= 5.1 || y != 0`

   b. `p && q || q && r`

4. Draw evaluation trees for the following:

   a. `a == (b + a - c)`

   b. `(c == (a + b)) || !flag`

   c. `(a <> 7) && (c >=6) || flag`

   d. `!flag && a > b || c < d`

Programming

1. Write a logical expression that is true for each of the following conditions.

   a. `gender` is `'m'` or `'M'` and `age` is greater than or equal to 62 or `gender` is `'f'` or `'F'` and `age` is greater than or equal to 65.

   b. `water` is greater than 0.1 and also less than 1.5.

   c. `year` is divisible by 4 but not divisible by 100. (Hint: use %.)

   d. `speed` is not greater than 55.

   e. `temp` below freezing and `windSpeed` over 20 knots

2. Write logical assignment statements for the following:

   a. Assign `true` to `between` if `n` is in the range -k and +k, inclusive; otherwise, assign a value of `false`.

   b. Assign `true` to `lowercase` if `ch` is a lowercase letter; otherwise, assign a value of `false`.

   c. Assign `true` to `isDigit` if `next` is a digit character ('0' through '9', inclusive).

   d. Assign `true` to `divisor` if `m` is a divisor of `n`; otherwise assign a value of `false`.

   e. Assign `true` to `isLeapYear` if year is divisible by 400 or if year is divisible by 4 but not by 100. For example, 2000 is a leap year, 2004 is a leap year, 2008 is a leap year . . . but 2100 is not a leap year. Every fourth century is a leap year (2000, 2400, 2800, and so on).

   f. Assign `true` to `isOp` if `ch` is one of the characters `+, -, *, /, %`.

4.3 Introduction to the `if` Control Statement

You now know how to write a C++ expression that is the equivalent of a question such as "is x an even number?" Next, we need to investigate a way to use the value of the expression to select a course of action. In C++, the `if` statement is the primary selection control structure.

`if` Statement with Two Alternatives

The `if` statement

```
if (gross > 100.00)
   net = gross - tax;
else
   net = gross;
```

selects one of the two assignment statements listed. It selects the statement immediately following the condition `(gross > 100.00)` if the condition is true (that is, `gross` is greater than `100.00`); it selects the statement following the reserved word `else` if the logical expression is false (that is, `gross` is not greater than `100.00`). It is never possible for both statements to execute after testing a particular condition. You must always enclose the condition in parentheses, but must not place a semicolon after the condition.

flowchart
A diagram that shows the step-by-step execution of a control structure

Figure 4.4 provides a **flowchart**—a diagram that uses boxes and arrows to show the step-by-step execution of a control structure—for the `if` control statement above. A diamond-shaped box in a flowchart represents a decision. There's always one path into a decision and there are two paths out (labeled true and false).

The figure shows that the condition `(gross > 100.00)` enclosed in the diamond-shaped box is evaluated first. If the condition is true, program control follows the arrow labeled true, and the assignment statement in the rectangle on the right is executed. If the condition is false, program control

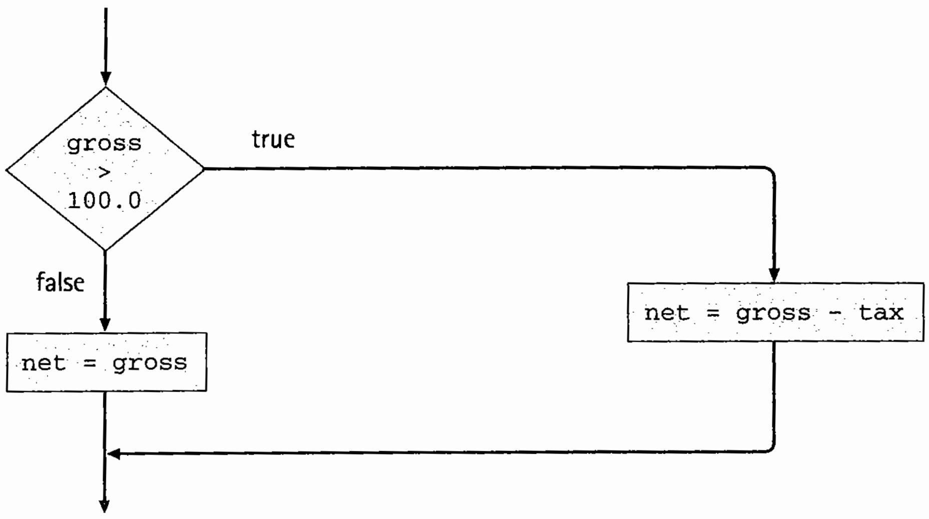


Figure 4.4 Flowchart of `if` statement with two alternatives

follows the arrow labeled false, and the assignment statement in the rectangle on the left is executed.

if Statement with Dependent Statement

The if statement above has two alternatives, but only one will be executed for a given value of gross. Example 4.6 illustrates that an if statement can also have a single alternative that is executed only when the condition is true.

EXAMPLE 4.6

The if statement below has one dependent statement that is executed only when x is not equal to zero. It causes product to be multiplied by x; the new value is saved in product, replacing the old value. If x is equal to zero, the multiplication is not performed. Figure 4.5 shows a flowchart for this if statement.

```
// Multiply product by a non zero x only.
if (x != 0)
   product = product * x;
```

if Statement Conditions with Characters and Strings

The if statements in the next few examples evaluate conditions with character or string operands.

In all our examples so far, the true and false alternatives of an if statement consist of a single C++ statement. In the next section, we'll see how to write true and false tasks consisting of more than one statement.

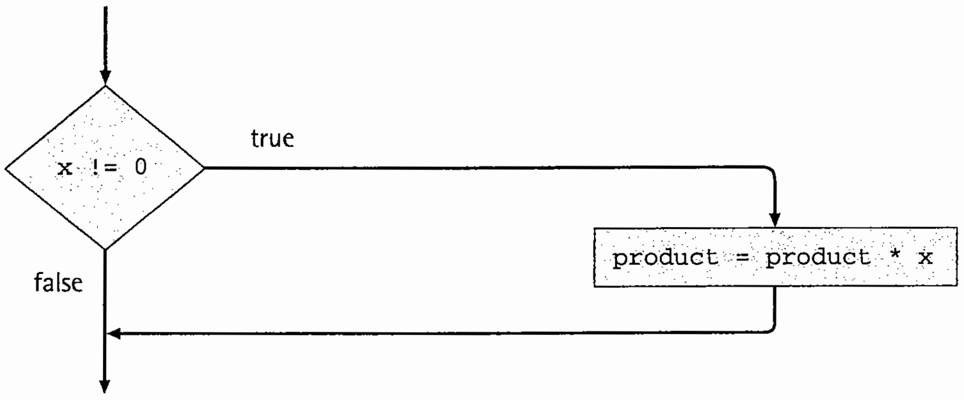


Figure 4.5 Flowchart of if statement with a dependent statement

EXAMPLE 4.7

The `if` statement below has two alternatives. It displays either `"Hi Mom"` or `"Hi Dad"` depending on the character stored in variable `momOrDad` (type `char`).

```
if (momOrDad == 'm')
   cout << "Hi Mom" << endl;
else
   cout << "Hi Dad" << endl;
```

EXAMPLE 4.8

The `if` statement below has one dependent statement; it displays the message `"Hi Mom"` only when `momOrDad` has the value `'m'`. Regardless of whether or not `"Hi Mom"` is displayed, the message `"Hi Dad"` is always displayed.

```
if (momOrDad == 'm')
   cout << "Hi Mom" << endl;
cout << "Hi Dad" << endl;
```

EXAMPLE 4.9

The `if` statement below displays the result of a string search operation. If the search string is found, the position is printed; otherwise, a message is printed indicating that the search string was not found. The call to `string` member function `find` (see Table 3.3) returns to `posTarget` (type `int`) the starting position in `testString` (type `string`) of `target` (type `string`). If `target` is not found, `find` returns a value that is not in the allowable range (0 through *k*–1 where *k* is the length of `testString`). The if statement tests whether the value returned to `posTarget` is in range before reporting the search result. If `testString` is `"This is a string"` and `target` is `"his"`, the `if` statement would display `his` found at `position 1`.

```
posTarget = testString.find(target);
if ((0 <= posTarget) && (posTarget < testString.length()))
    cout << target << " found at position " << posTarget
         << endl;
else
   cout << target << " not found!" << endl;
```

EXAMPLE 4.10

The next if statement replaces one substring (target) with another (newString). If target is found in testString, function find returns the position of target. The condition will be true and the **true task** (following the condition) of the if statement replaces it with newString. Recall that the three arguments of string member function replace (see Table 3.3) are the starting position of the string to be replaced (posTarget), the number of characters to be removed (target.length()), and the replacement string (newString). If testString is "This is a string", target is "his", and newString is "here", testString will become "There is a string". If the condition is false, the **false task** (following the word else) displays a message.

true task
Statement(s) after the condition that execute when the condition is true.

false task
Statement(s) after else that execute when the condition is false.

```
posTarget = testString.find(target);
if ((0 <= posTarget) && (posTarget < testString.length()))
   testString.replace(posTarget, target.length(),
                      newString);
else
   cout << target << " not found - no replacement!" << endl;
```

Format of the if Statement

In all the if statement examples, *statement*$_T$ and *statement*$_F$ are indented. If you use the word else, enter it on a separate line, aligned with the word if. The format of the if statement makes its meaning apparent. Again, this is done solely to improve program readability; the format used makes no difference to the compiler.

if Statement with Dependent Statement

Form:
```
if (condition)
   statementT
```

Example:
```
if (x > 0.0)
   positiveProduct = positiveProduct * x;
```

Interpretation: If the *condition* evaluates to true, then *statement*$_T$ is executed; otherwise, it is skipped.

if Statement with Two Alternatives

Form:

```
if (condition)
   statement_T
else
   statement_F
```

Example:

```
if (x >= 0.0)
      cout << "Positive" << endl;
else
      cout << "Negative" << endl;
```

Interpretation: If the *condition* evaluates to true, then *statement*$_T$ is executed and *statement*$_F$ is skipped; otherwise, *statement*$_T$ is skipped and *statement*$_F$ is executed.

EXERCISES FOR SECTION 4.3

Self-Check

1. What does the following fragment in part a display when x is 10? When x is 20? What does the fragment in part b display?

   a.
   ```
   if (x < 12)
      cout << "less" << endl;
   else
      cout << "done" << endl;
   ```

   b.
   ```
   var1 = 15.0;
   var2 = 25.12;
   if (2 * var1 >= var2)
      cout << "O.K." << endl;
   else
      cout << "Not O.K." << endl;
   ```

2. What value is assigned to x for each segment below when y is 10.0?

   a.
   ```
   x = 20.0;
   if (y != (x - 10.0))
      x = x - 10.0;
   else
      x = x / 2.0;
   ```

   b.
   ```
   if (y < 5.0 && y >= 0.0)
      x = 5 + y;
   else
      x = 2 - y;
   ```

3. Trace the execution of the `if` statement in Example 4.10 for the values of `target` and `newString` below. For each part, assume `testString` is `"Here is the string"` before the `if` statement executes.
   a. `target` is `"the"`, `newString` is `"that"`
   b. `target` is `"Here"`, `newString` is `"There"`
   c. `target` is `"Where"`, `newString` is `"There"`

Programming

1. Write C++ statements to carry out the steps below.
   a. Store the absolute difference of `x` and `y` in `result`, where the absolute difference is `(x - y)` or `(y - x)`, whichever is positive. Don't use the `abs` or `fabs` function in your solution.
   b. If `x` is zero, add 1 to `zeroCount`. If `x` is negative, add `x` to `minusSum`. If `x` is greater than zero, add `x` to `plusSum`.
2. Write your own absolute value function that returns the absolute value of its input argument.
3. Write an `if` statement that stores the value of `a + b` in `result` if the character stored in `op` is `'+'`; otherwise, it stores −1.0 in `result`. Variables `a`, `b`, and `result` are type `float`.
4. Write an `if` statement that displays the value of `a / b` if b is not zero; otherwise, it displays the message quotient not defined.

4.4 if Statements with Compound Alternatives

So far, we've seen `if` statements in which only one statement executes in either the true case or the false case. You may also write `if` statements that execute a series of steps by using compound statements, statements bracketed by `{ }`. When the symbol `{` follows the condition or the keyword `else`, the C++ compiler either executes or skips all statements through the matching `}`. Assignment statements, function call statements, or even other `if` statements may appear in these alternatives.

EXAMPLE 4.11

Suppose you're a biologist studying the growth rate of fruit flies. The `if` statement

```
if (popToday > popYesterday)
{
   growth = popToday - popYesterday;
   growthPct = 100.0 * growth / popYesterday;
   cout << "The growth percentage is " << growthPct;
}
```

computes the population growth from yesterday to today as a percentage of yesterday's population. The compound statement after the condition executes only when today's population is larger than yesterday's. The first assignment computes the increase in the fruit fly population, and the second assignment converts it to a percentage of the original population, which is displayed.

EXAMPLE 4.12

As the manager of a clothing boutique, you want to keep records of your financial transactions, which can be either payments you make for goods received or deposits made to your checking account. Payments are designated by a `transactionType` value of `'c'`. The true task in the `if` statement below processes a payment; the false task processes a deposit. In both cases, an appropriate message is printed and the account balance is updated. Both the true and false statements are compound statements.

```
if (transactionType == 'c')
{  // process check
   cout << "Check for $" << transactionAmount << endl;
   balance = balance - transactionAmount;
}
else
{  // process deposit
   cout << "Deposit of $" << transactionAmount << endl;
   balance = balance + transactionAmount;
}
```

PROGRAM STYLE Writing `if` Statements with Compound True or False Statements

Each `if` statement in this section contains at least one compound statement bracketed by `{ }`. The placement of the braces is a stylistic preference. Your instructor may want the opening `{` at the end of the same line as the condition. The closing `}` of a compound `if` may also appear on the same line as the `else`:

```
} else {
```

Some programmers prefer to use braces even when the statement following `if` or `else` is a single statement. This way all `if` statements look the same, and there will be no danger of forgetting braces when needed. Whichever style you choose, be consistent in its use.

Tracing an `if` Statement

A critical step in program design is to verify that an algorithm or C++ statement is correct before you spend extensive time coding or debugging it.

Often a few extra minutes spent in verifying the correctness of an algorithm saves hours of coding and testing time.

A **hand trace,** or **desk check,** is a careful, step-by-step simulation on paper of how the computer executes the algorithm or statement. The results of this simulation should show the effect of each step's execution using data that are relatively easy to process by hand. We illustrate this next.

hand trace (desk check)
The step-by-step simulation of an algorithm's execution.

EXAMPLE 4.13

In many programming problems you must order a pair of data values in memory so that the smaller value is stored in one variable (say, x) and the larger value in another (say, y). The next `if` statement rearranges any two values stored in `x` and `y` so that the smaller number is in `x` and the larger number is in `y`. If the two numbers are already in the proper order, the compound statement is not executed.

```
if (x > y)
{                    // exchange values in x and y
   temp = x;         // store original value of x in temp
   x = y;            // store original value of y in x
   y = temp;         // store original value of x in y
}
```

Although the values of `x` and `y` are being switched, an additional variable, `temp`, is needed to store a copy of one of these values. Variables `x, y`, and `temp` should all be the same data type.

Table 4.9 traces the execution of this `if` statement when `x` is `12.5` and `y` is `5.0`. The table shows that `temp` is initially undefined (indicated by `?`). Each line of the table shows the part of the `if` statement that is being executed, followed by its effect. If any variable gets a new value, its new value is shown on that line. If no new value is shown, the variable retains its previous value. The last value stored in `x` is `5.0`, and the last value stored in `y` is `12.5`.

The trace in Table 4.9 shows that `5.0` and `12.5` are correctly stored in `x` and `y` when the condition is `true`. To verify that the `if` statement is correct, you should select other data that cause the condition to evaluate to `false`. Also, you should verify that the statement is correct for special situations. For example, what would happen if `x` were equal to `y`? Would the statement still provide the correct result? To complete the hand trace, you would need to show that the algorithm handles this special situation properly.

In tracing each case, you must be careful to execute the statement step-by-step exactly as the computer would execute it. Often programmers assume how a particular step will be executed and don't explicitly test each condition and trace each step. A trace performed in this way is of little value.

Table 4.9 Step-by-Step Hand Trace of `if` Statement

Statement Part	x	y	temp	Effect
	12.5	5.0	?	
`if (x > y)`				`12.5 > 5.0` is true
`{`				
`temp = x;`			12.5	Store x in temp
`x = y;`	5.0			Store original y in x
`y = temp;`		12.5		Store original x in y
`}`				

EXERCISES FOR SECTION 4.4

Self-Check

1. Insert braces where needed below to avoid syntax or logic errors. Indent as needed to improve readability. The last statement should execute regardless of the value of x or y.

```
if (x > y)

x = x + 10.0;
cout << "x bigger than y" << endl;

else

  y = y + 10.0;
cout << "x smaller than y" << endl;
cout << "x is " << x
     << "y is " << y << endl;
```

2. What would be the syntax error in Self-Check Exercise 1?

3. What would be the effect of placing braces around the last three statements in Self-Check Exercise 1?

4. Correct the following `if` statement:

```
if (num1 < 0)
{
   product = num1 * num2 * num3;
   cout << "Product is " << product << endl;
else
   sum = num1 + num2 + num3;
   cout << "Sum is " << sum << endl;
}
```

Programming

1. Write an `if` statement that assigns to `ave` the average of a set of `n` numbers when `n` is greater than 0. If `n` is not greater than 0, assign 0 to `ave` and display an error message. The average should be computed by dividing `total` by `n`.

2. Write an interactive program that contains a compound `if` statement and may be used to compute the area of a square (area = side^2) or triangle (area = $1/2 \times$ base $\times$ height) after prompting the user to type the first character of the figure name (`t` or `s`).

4.5 Decision Steps in Algorithms

Algorithm steps that select from a choice of actions are called **decision steps.** The algorithm in the following case contains decision steps to compute an employee's gross and net pay after deductions. The decision steps are coded as `if` statements.

decision step
An algorithm step that selects one of several alternatives.

case study Payroll Problem with Functions

STATEMENT

Your company pays its hourly workers once a week. An employee's pay is based upon the number of hours worked (to the nearest half hour) and the employee's hourly pay rate. Weekly hours exceeding 40 are paid at a rate of time and a half. Employees who earn over $100 a week must pay union dues of $15 per week. Write a payroll program that will determine the gross pay and net pay for an employee.

ANALYSIS

The problem data include the input data for hours worked and hourly pay, and two required outputs: gross pay and net pay. There are also several constants: the union dues ($15), the minimum weekly earnings before dues must be paid ($100), the maximum hours before overtime must be paid (40), and the overtime rate (1.5 times the usual hourly rate). With this information, we can begin to write the data requirements for this problem. We can model all data using the `float` data types.

DATA REQUIREMENTS

Problem Constants

```
MAX_NO_DUES = 100.00      // maximum earnings (dollars) without
                          // paying union dues
```

```
DUES = 15.00                  // union dues (dollars) to be paid
MAX_NO_OVERTIME = 40.0        // maximum hours without overtime pay
OVERTIME_RATE = 1.5           // time and a half for overtime
```

Problem Input

```
float hours      // hours worked
float rate       // hourly rate
```

Problem Output

```
float gross      // gross pay
float net        // net pay
```

PROGRAM DESIGN

The problem solution requires that the program read the hours worked and the hourly rate before performing any computations. After reading these data, we need to compute and then display the gross pay and net pay.

The structure chart for this problem (Figure 4.6) shows the decomposition of the original problem into five subproblems. We'll write three of the subproblems as functions. For these three subproblems, the corresponding function name appears under its box in the structure chart.

We added to the structure chart *data flow* information that shows the input and output of each program step. The structure chart shows that the step *enter data* provides values for `hours` and `rate` as its output (data flow arrow points up). Similarly, the step *compute gross pay* uses `hours` and `rate` as input to the function (data flow arrow points down) and provides `gross` as the function output.

We can now write the initial algorithm.

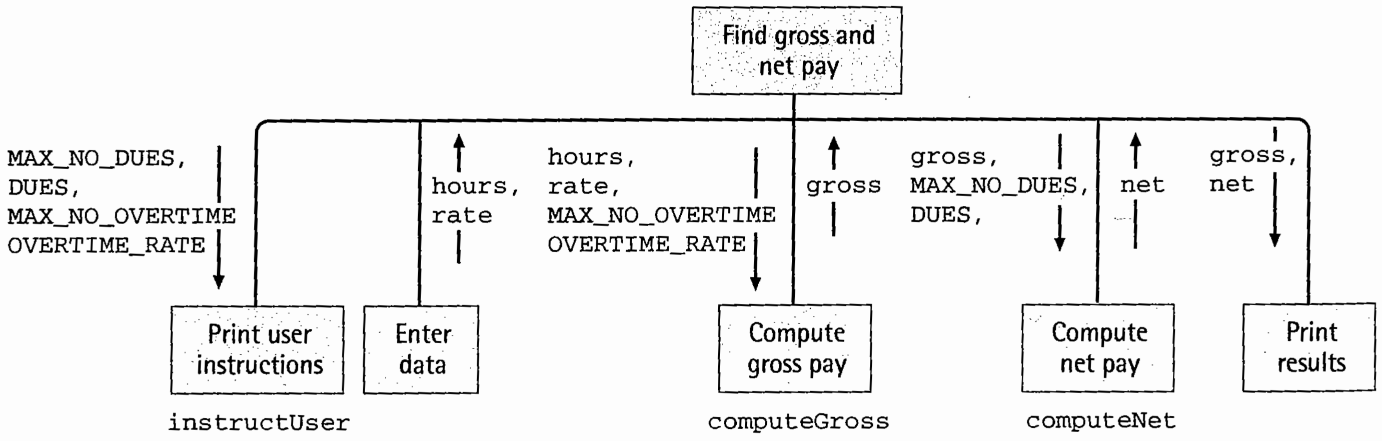


Figure 4.6 Structure chart for payroll problem

INITIAL ALGORITHM

1. Display user instructions (function `instructUser`).
2. Enter hours worked and hourly rate.
3. Compute gross pay (function `computeGross`).
4. Compute net pay (function `computeNet`).
5. Display gross pay and net pay.

We now focus on the three lower-level functions listed in parentheses in the algorithm. We begin by defining the interfaces between these functions and the main function.

ANALYSIS AND DESIGN FOR `INSTRUCTUSER` (ALGORITHM STEP 1)

In our analysis of the lower-level functions, we'll assume that the constant identifiers `MAX_NO_DUES`, `DUES`, `MAX_NO_OVERTIME`, and `OVERTIME_RATE` are declared before function `main`. This makes them **global constants**, which means that their values can be referenced in all functions without the need to pass them as arguments.

global constant
A constant declared before function `main` that can be referenced in all functions.

INTERFACE FOR **`instructUser`**

Input Arguments

(none)

Function Return Value

(none—a void function)
This function simply displays a short list of instructions and information about the program for the user. The information includes the values of the four program constants and other information to ensure that the user has an overview of what the program does and how the user should enter the input data.

ANALYSIS AND DESIGN FOR `COMPUTEGROSS` (ALGORITHM STEP 3)

The function `computeGross` needs to know the hours worked and the hourly rate for the employee. It also needs the values of the constants (`MAX_NO_OVERTIME` and `OVERTIME_RATE`) involved in determining the employee's gross pay.

INTERFACE FOR **`computeGross`**

Input Arguments

```
float hours        // number of hours worked
float rate         // hourly pay rate (dollars)
```

Function Return Value

```
float gross
```

FORMULA

Gross pay = Hours worked × Hourly pay

To compute gross pay, we first need to determine whether any overtime pay is due. If there is, we should compute regular pay and overtime pay and return their sum. If not, we simply compute gross pay as the product of hours and rate. We list the local data needed and refine the algorithm step (Step 3) as a *decision step*.

Local Data for `computeGross`

```
float gross             // gross pay (dollars)
float regularPay        // pay for first 40 hours of work
float overtimePay       // pay for hours in excess of 40 (dollars)
```

Algorithm for `computeGross`

```
3.1. If the hours worked exceeds 40.0 (max hours before overtime)
        3.1.1. Compute regularPay.
        3.1.2. Compute overtimePay.
        3.1.3. Add regularPay to overtimePay to get gross.
     Else
        3.1.4. Compute gross as hours * rate.
```

pseudocode
A mixture of English and C++ reserved words that is used to describe an algorithm step.

The decision step above is expressed in **pseudocode**—a mixture of English and C++ reserved words that is used to describe algorithm steps. In the pseudocode above, we use indentation and the reserved words `if` and `else` to show the flow of control.

ANALYSIS AND DESIGN FOR `COMPUTENET` (STEP 4)

Once we have the gross pay (returned by `computeGross`), we can compute net pay. The constants representing the union dues cutoff value ($100) and the dues ($15) are required as well as the value of gross salary.

INTERFACE FOR **`computeNet`**

Input Arguments

```
float gross        // gross pay (dollars)
```

Function Return Value

```
float net          // net pay (dollars)
```

FORMULA

```
net pay = gross pay − deductions
```

Next, we list the local data needed for `computeNet` and refine the algorithm step (Step 4).

Local Data for computeNet

```
float net           // net pay (dollars)
```

Algorithm for computeNet

```
4.1. If the gross pay is larger than $100.00
        4.1.1.  Deduct the dues of $15 from gross pay.
     Else
        4.1.2.  Deduct no dues.
```

IMPLEMENTATION

We now write the main function along with the prototypes for the lower-level functions (see Listing 4.1).

TESTING

To test this program, we need to be sure that all possible alternatives work properly. For example, to test function `computeNet`, we need three sets of data, one for the `if` (gross salary greater than $100), one for the `else` (gross salary less than $100), and one for the pivotal point (gross salary exactly $100). Listing 4.2 shows program output from a sample run.

Listing 4.1 Payroll problem with functions

```
// File: payrollFunctions.cpp
// Computes and displays gross pay and net pay given an hourly
//    rate and number of hours worked. Deducts union dues of $15
//    if gross salary exceeds $100; otherwise, deducts no dues.

#include <iostream>
using namespace std;

// Functions used   . . .
void instructUser();
float computeGross(float, float);
float computeNet(money);

const float MAX_NO_DUES = 100.00;     // max earnings before dues (dollars)
const float dues = 15.00;             // dues amount (dollars)
const float MAX_NO_OVERTIME = 40.0;   // max hours before overtime
const float OVERTIME_RATE = 1.5;      // overtime rate
```

(continued)

Listing 4.1 Payroll problem with functions (continued)

```
int main ()
{
   float hours;                   // input: hours worked
   float rate;                    // input: hourly pay rate (dollars)
   float gross;                   // output: gross pay (dollars)
   float net;                     // output: net pay (dollars)

   // Display user instructions.
   instructUser();
   // Enter hours and rate.
   cout << "Hours worked: ";
   cin >> hours;
   cout << "Hourly rate: $";
   cin >> rate;

   // Compute gross salary.
   gross = computeGross(hours, rate);

   // Compute net salary.
   net = computeNet(gross);

   // Print gross and net.
   cout << "Gross salary is $" << gross << endl;
   cout << "Net salary is $" << net << endl;

   return 0;
}

// Displays user instructions
void instructUser()
{
   cout << "This program computes gross and net salary." << endl;
   cout << "A dues amount of " << DUES << " is deducted for" << endl;
   cout << "an employee who earns more than " << MAX_NO_DUES << endl
        << endl;
   cout << "Overtime is paid at the rate of " << OVERTIME_RATE << endl;
   cout << "times the regular rate for hours worked over "
        << MAX_NO_OVERTIME endl << endl;
   cout << "Enter hours worked and hourly rate" << endl;
   cout << "on separate lines after the prompts." << endl;
```

(continued)

Listing 4.1 Payroll problem with functions (continued)

```
      cout << "Press <return> after typing each number." << endl << endl;
   }  // end instructUser

   // Find the gross pay
   float computeGross
      (float hours,                 // IN: number of hours worked
      float rate)                   // IN: hourly pay rate (dollars)
   {

      // Local data . . .
      float gross;                // RESULT: gross pay (dollars)
      float regularPay;           // pay for first 40 hours
      float overtimePay;          // pay for hours in excess of 40

      // Compute gross pay.
      if (hours > MAX_NO_OVERTIME)
      {
         regularPay = MAX_NO_OVERTIME * rate;
         overtimePay = (hours - MAX_NO_OVERTIME) * OVERTIME_RATE * rate;
         gross = regularPay + overtimePay;
      }
      else
         gross = hours * rate;

      return gross;
   }  // end computeGross

   // Find the net pay
   float computeNet
      (float gross)             // IN: gross salary (dollars)
   {
      // Local data . . .
      float net;                // RESULT: net pay (dollars)

      // Compute net pay.
      if (gross > MAX_NO_DUES)
         net = gross - DUES;    // deduct dues amount
      else
         net = gross;           // no deductions

      return net;
   } // end computeNet
```

Listing 4.2 Sample run of payroll program with functions

```
This program computes gross and net salary.
A dues amount of $15.00 is deducted for
an employee who earns more than $100.00

Overtime is paid at the rate of 1.5
times the regular rate on hours worked over 40

Enter hours worked and hourly rate
on separate lines after the prompts.
Press <return> after typing each number.

Hours worked: 50
Hourly rate: $6
Gross salary is $330
Net salary is $315
```

PROGRAM STYLE Global Constants Enhance Readability and Maintenance

The four program constants used in this case are not essential. We could just as easily have placed the constant values (`40.0`, `1.5`, `100.00`, and `15.00`) directly in the code. For example, we could have written the decision in `computeNet` as

```
if (gross > 100.00)
   net = gross - 15.00;  // deduct amount dues
else
   net = gross;          // no deductions
```

However, the use of constant identifiers rather than constant values has two advantages. First, the original `if` statement is easier to understand because it uses descriptive names such as `MAX_NO_DUES` rather than numbers, which have no intrinsic meaning. Second, a program written with constant identifiers is much easier to maintain than one written with constant values. For example, if we want to use different constant values in the payroll program in Listing 4.1, we need to change only the constant declarations. However, if we had inserted constant values directly in the `if` statement, we'd have to change the `if` statement and any other statements that manipulate or display the constant values.

The constants have global scope, which means that any function can reference them. Although you must not give variables global scope, you can

give constants global scope because the value of a constant, unlike a variable, cannot be changed in a lower-level function. If a constant is used by more than one function, you should declare it as a global constant. The alternatives would be to pass it as an argument to each function or to duplicate its definition in each function. Giving it global scope shortens argument lists and ensures that all functions have the same value for the constant.

A Reminder About Identifier Scope

Notice that identifier `gross` is declared as a local variable in functions `main` and `computeGross`, and as a formal argument in function `computeNet`. The scope of each declaration is the function that declares it. To establish a connection between these independent declarations of the identifier `gross`, we use argument list correspondence or the function return mechanism. The variable `gross` in function `main` is passed as an actual argument (corresponding to formal argument `gross`) in the call to `computeNet`. Function `computeGross` returns as its result the value of its local variable `gross`; the statement in function `main` that calls `computeGross`

```
gross = computeGross(hours, rate );
```

assigns the function result to `main's` local variable `gross`.

Adding Data Flow Information to Structure Charts

In Figure 4.6, we added data flow information to the structure chart showing the input and output of each of the top-level problem solution steps. The data flow information is an important part of the program documentation. It shows what program variables are processed by each step and the manner in which these variables are processed. If a step gives a new value to a variable, then the variable is considered an *output of the step.* If a step displays a variable's value or uses it in a computation without changing its value, the variable is considered an *input of the step.* For example, the step "compute net pay" consists of a function that processes variables `gross` and `net`. This step uses the value of the variable `gross`, as well as the constants `MAX_NO_DUES` and `DUES` (its inputs) to compute `net` (its output).

Figure 4.6 also shows that a variable may have different roles in different subproblems of the structure chart. When considered in the context of the original problem statement, `hours` and `rate` are problem inputs (data supplied by the program user). However, when considered in the context of the subproblem "enter data," the subproblem's task is to deliver values for `hours` and `rate` to the main function, so they're considered outputs from this step.

```
            3.1.3. Add regularPay to overtimePay to get gross.
        Else
            3.1.4. Compute gross as hours * rate.
4. Compute net pay.
    4.1. If gross is larger than $100.00
            4.1.1. Deduct the dues of $15.00 from gross pay.
        Else
            4.1.2. Deduct no dues.
5. Display gross and net pay.
```

In Table 4.10, each step is listed at the left in the order of its execution. The last column shows the effect of each step. If a step changes the value of a variable, then the table shows the new value. If no new value is shown, the variable retains its previous value. For example, the table shows that step 2 stores the data values `30.0` and `10.00` in the variables `hours` and `rate`; `gross` and `net` are still undefined (indicated by `?` in the first table row).

The trace in Table 4.10 shows that `300.0` and `285.0` are stored in `gross` and `net` and then displayed. To verify that the algorithm is correct, you'd need to select other data that cause the two conditions to evaluate to different combinations of their values. Since there are two conditions and each has two possible values (`true` or `false`), there are two times two, or four, different combinations that should be tried. An exhaustive hand trace of the algorithm would show that it works for all combinations.

Besides the four cases discussed above, you should verify that the algorithm works correctly for unusual data. For example, what would happen if `hours` were `40.0` (value of `MAX_NO_OVERTIME`) or if `gross` were `100.0` (value of `MAX_NO_DUES`)? Would the algorithm still provide the correct

Table 4.10 Trace of Algorithm for Payroll Problem

Algorithm Step	hours	rate	gross	net	Effect
	?	?	?	?	
2. Enter `hours` and `rate`	`30.0`	`10.00`			Reads the data
3.1. If `hours` > 40.0					`hours > 40` is false
3.1.4. `gross` gets `hours` * `rate`			`300.0`		Compute `gross` as `hours * rate`
4.1. If `gross` > $100.00					`gross > 100.00` is true
4.1.1. Deduct the dues of $15				`285.0`	Deduct the dues of `$15.00`
5. Display gross and net					Displays 300.00 and 285.00

result? To complete the hand trace, you'd need to show that the algorithm handles these special situations properly.

In tracing each case, you must be careful to execute the algorithm exactly as the computer would execute it. Don't assume how a particular step will execute.

EXERCISES FOR SECTION 4.6

Self-Check

1. Provide sample data that cause both conditions in the payroll problem to be true and trace the execution for these data.
2. If `hours` is greater than `MAX_NO_OVERTIME` and `gross` is less than `MAX_NO_DUES`, which assignment steps in the algorithm would be performed? Provide a trace.

4.7 Nested `if` Statements and Multiple-Alternative Decisions

Until now, we used `if` statements to implement decisions involving up to two alternatives. In this section, we use **nested `if` statements** (one `if` statement inside another) to code decisions with several alternatives.

nested `if` statement
An `if` statement with another `if` statement as its true task or its false task.

EXAMPLE 4.14

The following nested `if` statement has three alternatives. It increases one of three variables (`numPos`, `numNeg`, or `numZero`) by one depending on whether `x` is greater than zero, less than zero, or equal to zero, respectively. The boxes show the logical structure of the nested `if` statement: the second `if` statement is the `false` task following the `else` of the first `if` statement.

```
if (x > 0)
     numPos = numPos + 1;
else
     if (x < 0)
          numNeg = numNeg + 1;
     else // x equals 0
          numZero = numZero + 1;
```

The execution of the nested `if` statement proceeds as follows: the first condition `(x > 0)` is tested; if it is true, `numPos` is incremented and the rest of the `if` statement is skipped. If the first condition is false, the second condition `(x < 0)` is tested; if it is true, `numNeg` is incremented; otherwise, `numzero` is incremented. It's important to realize that the second condition is tested only when the first condition is false.

Table 4.11 traces the execution of this statement when `x` is −7. Because `x > 0` is false, the second condition is tested.

Table 4.11 Trace of `if` Statement in Example 4.14 for `x = -7`

Statement Part	Effect
`if (x > 0)`	−7 > 0 is false
`else`	
`   if (x < 0)`	−7 < 0 is true
`   numNeg = numNeg + 1;`	add 1 to `numNeg`

Comparison of Nested `if` Statements and a Sequence of `if` Statements

Beginning programmers sometimes prefer to use a sequence of `if` statements rather than a single nested `if` statement. For example, the previous `if` statement is rewritten as the following sequence of `if` statements.

```
// Less efficient sequence of if statements
if (x > 0)
   numPos = numPos + 1;
if (x < 0)
   numNeg = numNeg + 1;
if (x == 0)
   numZero = numZero + 1;
```

Although this sequence is logically equivalent to the original, it's neither as readable nor as efficient. Unlike the nested `if`, the sequence doesn't show clearly that exactly one of the three assignment statements is executed for a particular `x`. It's less efficient because all three of the conditions are always tested. In the nested `if` statement, only the first condition is tested when `x` is positive.

Writing a Nested `if` as a Multiple-Alternative Decision

Nested `if` statements can become quite complex. If there are more than two alternatives and indentation is not consistent, it may be difficult to determine the logical structure of the `if` statement. For example, in the following nested `if` statement

```
if (condition₁)
   // statement₁
      . . .
   if (condition₂)
      // statement₂
      . . .
else
   // statement₃
      . . .
```

does the `else` match up with the first `if` as shown by the identation or with the second `if`? The answer is the second `if` because an `else` is always associated with the closest `if` without an `else`. In the situation discussed in Example 4.14 in which each false task (except the last) is followed by an `if` and matching `else`, you can code the nested `if` as the *multiple-alternative decision* described next.

Multiple-Alternative Decision Form

Form:

```
if (condition₁)
  statement₁;
else if (condition₂)
  statement₂;
  .
  .
  .
else if (conditionₙ)
  statementₙ;
else
  statementₑ;
```

Example:

```
// Increment numPos, numNeg, or numZero
// depending on x.
if (x > 0)
   numPos = numPos + 1;
else if (x < 0)
   numNeg = numNeg + 1;
else // x == 0
   numZero = numZero + 1;
```

Interpretation: The conditions in a multiple-alternative decision are evaluated in sequence until a true condition is reached. If a condition is true, the statement following it is executed and the rest of the multiple-alternative decision is skipped. If a condition is false, the statement following it is skipped and the next condition is tested. If all conditions are false, then *statement*$_e$ following the last `else` is executed.

EXAMPLE 4.15

Suppose you want to match exam scores to letter grades for a large class of students. The following table describes the assignment of grades based on each exam score.

Exam Score	Grade Assigned
90 and above	A
80 to 89	B
70 to 79	C
60 to 69	D
Below 60	F

The multiple-alternative decision in function `displayGrade` (Listing 4.3) displays the letter grade assigned according to this table. If a student has an exam score of 85, the last three conditions would be true; however, a grade of B would be assigned because the first true condition is `(score >= 80)`.

Listing 4.3 Function `displayGrade`

```
// Displays the letter grade corresponding to an exam
// score assuming a normal scale.
void displayGrade(int score)        // IN: exam score
{
   if (score >= 90)
      cout << "Grade is A" << endl;
   else if (score >= 80)
      cout << "Grade is B" << endl;
   else if (score >= 70)
      cout << "Grade is C" << endl;
   else if (score >= 60)
      cout << "Grade is D" << endl;
   else
      cout << "Grade is F" << endl;
}
```

Order of Conditions

When more than one condition in a multiple-alternative decision is true, only the task following the first true condition is executed. Therefore, the order of the conditions can affect the outcome.

Writing the decision as follows would be incorrect. All passing scores (60 or above) would be incorrectly categorized as a grade of D because the first condition would be true and the rest would be skipped.

```
// incorrect grade assignment
if (score >= 60)
   cout << "Grade is D" << endl;
else if (score >= 70)
   cout << "Grade is C" << endl;
else if (score >= 80)
   cout << "Grade is B" << endl;
else if (score >= 90)
   cout << "Grade is A" << endl;
else
   cout << "Grade is F" << endl;
```

The order of conditions can also have an effect on program efficiency. If we know that low exam scores are much more likely than high scores, it would be more efficient to test first for scores below 60, next for scores below 70, and so on (see Programming Exercise 3 at the end of this section).

EXAMPLE 4.16

You could use a multiple-alternative `if` statement to implement a decision table that describes several alternatives. Let's assume that you're an accountant setting up a payroll system for a small firm having five different ranges for salaries up to $150,000, as shown in Table 4.12. Each table line shows the base tax amount (column 2) and tax percentage (column 3) for a particular salary range (column 1). Given a salary, you can calculate the tax by adding the base tax for that salary range to the product of the percentage of excess and the amount of salary over the minimum salary for that range.

For example, the second line of the table specifies that the tax due on a salary of $20,000.00 is $2,250.00 plus 16 percent of the excess salary over $15,000.00 (that is, 16 percent of $5,000.00, or $800.00). Therefore, the total tax due is $2,250.00 plus $800.00, or $3,050.00.

The `if` statement in function `computeTax` (Listing 4.4) implements the tax table. If the value of `salary` is within the table range (0.00 to 150,000.00), exactly one of the statements assigning a value to `tax` will be executed. Table 4.13 shows a hand trace of the `if` statement when `salary` is $20,000.00. You can see that the value assigned to `tax`, $3050.00, is correct. Remember not to include symbols such as the dollar sign or commas in a `float` literal.

Table 4.12 Decision Table for Example 4.16

Range	Salary	Base Tax	Percentage of Excess
1	0.00 to 14,999.99	0.00	15
2	15,000.00 to 29,999.99	2250.00	16
3	30,000.00 to 49,999.99	4650.00	18
4	50,000.00 to 79,999.99	8250.00	20
5	80,000.00 to 150,000.00	14,250.00	25

Listing 4.4 Function computeTax

```
// Compute the tax due based on a tax table
float computeTax(float salary)   // IN: salary amount
{
   if (salary < 0.00)
      tax = -1;
   else if (salary < 15000.00)          // first range
      tax = 0.15 * salary;
   else if (salary < 30000.00)          // second range
      tax = (salary - 15000.00) * 0.16 + 2250.00;
   else if (salary < 50000.00)          // third range
      tax = (salary - 30000.00) * 0.18 + 4650.00;
   else if (salary < 80000.00)          // fourth range
      tax = (salary - 50000.00) * 0.20 + 8250.00;
   else if (salary <= 150000.00)        // fifth range
      tax = (salary - 80000.00) * 0.25 + 14250.00;
   else
      tax = -1;

   return tax;
}
```

Table 4.13 Trace of `if` Statement in Listing 4.4 for salary equals $20000.00

Statement Part	salary	tax	Effect
	20000.00	?	
`if (salary < 0.00)`			20000.00 < 0.00 is false
`else if (salary < 15000.00)`			20000.00 < 15000.00 is false
`else if (salary < 30000.00)`			20000.00 < 30000.00 is true
`   tax = (salary - 15000.00)`			Evaluates to 5000.00
`      * 0.16`			Evaluates to 800.00
`      + 2250.00`		3050.00	Evaluates to 3050.00

PROGRAM STYLE Validating the Value of Variables

If you validate the value of a variable before using it in computations, you can avoid processing invalid or meaningless data. Instead of computing an incorrect tax amount, function `computeTax` returns `-1.0` (an impossible tax amount) if the value of salary is outside the range covered by the table (`0.0` to `150,000.00`). The first condition sets tax to `-1.0` if `salary` is negative. All conditions evaluate to false if `salary` is greater than `150000.00`, so the task following `else` sets `tax` to `-1.0`. The function calling `computeTax` should display an error message if the value returned to it is `-1.0`.

Short-Circuit Evaluation of Logical Expressions

When evaluating logical expressions, C++ uses a technique called **short-circuit evaluation.** This means that evaluation of a logical expression stops as soon as its value can be determined. For example, if the value of `(single == 'y')` is `false`, then the logical expression

short-circuit evaluation
Stopping evaluation of a logical expression as soon as its value can be determined.

```
(single == 'y') && (gender == 'm') && (age >= 18)
```

must be false regardless of the value of the other conditions (that is, `false && ( . . . )` must always be false). Consequently, there's no need to continue to evaluate the other conditions when `(single == 'y')` evaluates to `false`.

EXAMPLE 4.17

If `x` is zero, the `if` condition

```
if ((x != 0.0) && (y / x > 5.0))
```

is `false` because `(x != 0.0)` is `false`, and false `&& ( . . . )` must always be `false`. Thus there's no need to evaluate the subexpression `(y / x > 5.0)` when `x` is zero. In this case, the first subexpression guards the second and prevents the second from being evaluated when `x` is equal to `0`. However, if the subexpressions were reversed, the expression

```
if ((y / x > 5.0) && (x != 0.0))
```

would cause a "division by zero" run-time error when the divisor `x` is zero. Therefore, the order of the subexpressions in this condition is critical.

EXERCISES FOR SECTION 4.7

Self-Check

1. Trace the execution of the nested `if` statements in Listing 4.4 for a salary of $135,000.
2. What would be the effect of reversing the order of the first two `if` statements in Listing 4.4?
3. Evaluate the expressions below, with and without short-circuit evaluation, if `x` equals 6 and `y` equals 7.

   a. `((x > 10) && (y / x <= 10))`
   b. `((x <= 10) || (x / (y - 7) > 3))`

4. Write pseudo code for the following: if the transaction is a deposit, add the transaction amount to the balance and display the new balance; else, if the transaction is a withdrawal and the transaction amount is less than or equal to the balance, deduct the transaction amount from the balance and display the new balance; else, if the transaction is a withdrawal but the amount is greater than the balance, deduct a PENALTY amount (a constant) from the balance and display a "withdrawal exceeds balance" warning message; else display an "illegal transaction" message.
5. Write pseudo code for the following: if `ch1` is one of the arithmetic operators, set `isOp` to `true`; otherwise, if `ch1` and `ch2` are both `|` or both `&`, set `isOp` to `true`; otherwise, set `isOp` to `false`.

Programming

1. Implement the decision table below using a nested `if` statement. Assume that the grade point average is within the range 0.0 through 4.0.

Grade Point Average	Transcript Message
0.0 to 0.99	Failed semester—registration suspended
1.0 to 1.99	On probation for next semester
2.0 to 2.99	(no message)
3.0 to 3.49	Dean's list for semester
3.5 to 4.0	Highest honors for semester

2. Implement the solution to Self-Check Exercise 4 using a string to represent the transaction type.
3. Rewrite the `if` statement for Example 4.15 using only the relational operator < in all conditions.
4. The Air Force has asked you to write a program to label supersonic aircraft as military or civilian. Your program is to be given the plane's observed speed in kilometers per hour (km/h) and its estimated length in meters.

For planes traveling in excess of 1100 km/h, you'll label those longer than 52 meters "civilian" and those shorter as "military". For planes traveling at slower speeds, you'll issue an "aircraft type unknown" message.

5. Write a type bool function that returns the value of isOp based on Self-Check Exercise 5.
6. Write a function that returns the value of a + b if op is +, a − b if op is −, and so on for '*', '/', and '%'. Parameters a, b are type int and op is type char.

4.8 The switch Control Statement

The switch control statement may also be used in C++ to select one of several alternatives. The switch statement is especially useful when the selection is based on the value of a single variable or a simple expression (called the switch *selector*). The value of this expression may be of type int, char, or bool, but not of type float or string.

EXAMPLE 4.18

The switch statement

```
switch (momOrDad)
{
   case 'M': case 'm':
      cout << "Hello Mom - Happy Mother's Day" << endl;
      break;
   case 'D': case 'd':
      cout << "Hello Dad - Happy Father's Day" << endl;
      break;
}
```

behaves the same way as the if statement below when the character stored in momOrDad is one of the four letters listed (M, m, D, or d).

```
if ((momOrDad == 'M') || (momOrDad == 'm'))
   cout << "Hello Mom - Happy Mother's Day" << endl;
else if ((momOrDad == 'D') || (momOrDad == 'd'))
   cout << "Hello Dad - Happy Father's Day" << endl;
```

The message displayed depends on the value of the switch selector momOrDad (type char). If the switch selector value is 'M' or 'm', the first message is displayed. If the switch selector value is 'D' or 'd', the second message is displayed. The character constants 'M', 'm' and 'D', 'd' are called *case labels*.

Once a particular case label statement has been executed, the reserved word break causes control to be passed to the first statement following the switch control statement.

switch Statement

Form:

```
switch (selector)
{
 case label_1 : statements_1;
                break;
 case label_2 : statements_2;
                break;
        .
        .
        .
 case label_n : statements_n;
                break;
 default: statements_d; // optional
}
```

Example:

```
// Display a musical note
switch (musicalNote)
{
 case 'c':
    cout << "do";
    break;
 case 'd':
    cout << "re";
    break;
 case 'e':
    cout << "mi";
    break;
 case 'f':
    cout << "fa";
    break;
 case 'g':
    cout << "sol";
    break;
 case 'a':
    cout << "la";
    break;
 case 'b':
    cout << "ti";
    break;
 default:
    cout << "An invalid note was read."
         << endl;
 }
```

Interpretation: The *selector* expression is evaluated and compared to each of the **case** labels until a match is found. The selector expression should be an integral type (for example, **int**, **char**, **bool**, but not **float** or **string**). Each *label* is a single, constant value, and each one must have a different value from the others. When a match between the value of the *selector* expression and a **case** *label* is found, the statements following the case *label* are executed until a **break** statement is encountered. Then the rest of the **switch** statement is skipped.

If no case *label* matches the *selector*, the entire switch body is skipped unless it contains a **default** label. If there is a **default** label, the statements following the **default** are executed. The statements following a **case** *label* may be one or more C++ statements, which means that you don't need to make multiple statements into a single compound statement using braces.

EXAMPLE 4.19

The **switch** statement in Listing 4.5 finds the average life expectancy of a standard lightbulb based on the bulb's wattage. Since the value of the variable **watts** controls the execution of the **switch** statement, **watts** must have a value before the statement executes. Case labels 40 and 60 have the same effect (also, case labels 75 and 100).

Listing 4.5 switch statement to determine life expectancy of a lightbulb

```
// Determine average life expectancy of a standard
// lightbulb.
switch (watts)
{
   case 25:
      life = 2500;
      break;
   case 40:
   case 60:
      life = 1000;
      break;
   case 75:
   case 100:
      life = 750;
      break;
   default:
      life = 0;
}
```

Proper Use of break

Don't forget to use the reserved word `break` at the end of a statement sequence in a `switch` statement. If the `break` is omitted, then execution will continue, or *fall through,* to the next statement sequence. This is usually an error in logic.

Comparison of Nested if Statements and the switch Statement

You can use a nested `if` statement, which is more general than the `switch` statement, to implement any multiple-alternative decision. The `switch` statement is more readable in many contexts and should be used whenever practical. Case labels that contain type `float` values or strings are not permitted.

You should use the `switch` statement when each label set contains a reasonable number of case labels (a maximum of ten). However, if the number of values is large, use a nested `if`. You should use a `default` label in a `switch` statement wherever possible. The discipline of trying to define a default will help you to consider what will happen if the value of your `switch` statement's selector expression falls outside the range of your set of case labels.

Using a switch Statement to Select Alternative Functions

Instead of placing lengthy statement alternatives in the body of a `switch`, you should consider moving this code to a function. This leads to a very common use for the `switch` statement: selecting a particular function from a group of possible choices.

EXAMPLE 4.20

Assume the type `char` variable `editOp` contains a data value (`D`, `I`, or `R`) that indicates the kind of text edit operation to be performed on `string` object `textString`. We use the `switch` statement in function `edit` (Listing 4.6) to call a function that performs the selected operation, passing `textString` as a function argument. Each function returns the modified string as its result. The modified string is assigned to `textString` and is returned by function `edit`.

Using this structure, we place the code for each edit operation in a separate function. These functions might prompt the user for specific information, such as a substring to insert, replace, or delete, and perform all necessary operations on this data (by calling member functions from the `string` class).

Listing 4.6 String editing function edit

```
// Perform string editing function.
string edit(string textString,       // IN: string to edit
            char editOp)             // IN: edit operation
{
   switch (editOp)
   {
     case 'D': case 'd':             // Delete a substring
        textString = doDelete(textString);
        break;
     case 'I': case 'i':             // Insert a substring
        textString = doInsert(textString);
        break;
     case 'R': case 'r':             // Replace a substring
        textString = doReplace(textString);
        break;
     case 'Q': case 'q' :            // Quit editing
        cout << "All done!" << endl;
        break;
     default:
        cout << "An invalid edit code was entered." << endl;
   }
   return textString;

} //end edit
```

Program decomposition into separate components is the key here. The details of how our string is processed can be written without affecting the simple *control code* in the `switch` statement.

EXERCISES FOR SECTION 4.8

Self-Check

1. What will be printed by the following carelessly constructed `switch` statement if the value of `color` is `'R'`? Fix the error in this statement.

```
switch (color)
{
case 'R': case 'r':
   cout << "red" << endl;
case 'B': case 'b':
   cout << "blue" << endl;
case 'Y': case 'y':
   cout << "yellow" << endl;
}
```

If color were type string, could you use "Red" and "red" as case labels for the first selection? Explain your answer.

2. Write an if statement that corresponds to the switch statement below.

```
switch (x > y)
{
   case 1 :
      cout << "x greater" << endl;
      break;
   case 0 :
      cout << "y greater or equal" << endl;
      break;
}
```

3. Why can't we rewrite our nested if statement examples from Section 4.7 using switch statements?
4. Why is it preferable to include a default label in a switch statement?
5. What would be the effect of omitting the first break statement in Listing 4.5? Would this be a syntax error, run-time error, or logic error?

Programming

1. Write a switch statement that prints a message indicating whether nextCh (type char) is an operator symbol (+, –, *, /, %), a punctuation symbol (comma, semicolon, parenthesis, brace, bracket), or a digit. Your statement should print the category selected. Include a default label.
2. Write a nested if statement equivalent to the switch statement described in Programming Exercise 1.
3. Implement the solution to Self-Check Exercise 4 of Section 4.7 using a type char variable for the transaction type. Assume 'd' or 'D' indicates a deposit and 'w' or 'W' indicates a withdrawal. Use a switch statement.
4. Use a switch statement in a function that returns the value of a + b if op is '+', a – b if op is '–', and so on for '*', '/', and '%'. Parameters a and b are type int and op is type char.

4.9 Common Programming Errors

Common errors using if statements involve writing the condition incorrectly and failure to enclose compound statements in braces. A common error for switch statements is omitting a break statement.

- *Parentheses:* For the most part, the defined precedence levels of C++ will prevent you from making a syntax error when writing conditions. But they will also allow you to do things you may not intend. The rule of thumb is, when in doubt, use parentheses.

- *Operators:* You can use the logical operators, `&&`, `||`, and `!`, only with logical expressions. Remember that the C++ operator for equality is the double symbol ==. Don't mistakenly use the assignment operator (single =) in its place. Your logical expression will probably compile, but the logic will be incorrect.
- *Compound statements:* Don't forget to bracket a compound statement used as a true or false task in an `if` control statement with braces. If the `{ }` bracket is missing, only the first statement will be considered part of the task. This could lead to a syntax error—or, worse, a logic error that could be very difficult to find. In the following example, the `{ }` bracket around the true task is missing. The compiler assumes that only the statement

```
sum = sum + x;
```

  is part of the true task of the `if` control statement. This creates a "misplaced else" syntax error. Of course, the correct thing to do is to enclose the compound statement in braces.

```
if (x > 0)                       // missing { }
   sum = sum + x;
   cout << "Greater than zero" << endl;
else
   cout << "Less than zero" << endl;
```

- *Nested* `if` *statement:* When writing a nested `if` statement, try to select the conditions so that the multiple-alternative form shown in Section 4.7 can be used. If the conditions are not mutually exclusive (that is, if more than one condition may be true), the most restrictive condition should come first.
- `switch` *statements:* When using a `switch` statement, make sure the `switch` selector and `case` labels are of the same type (`int`, `char`, or `bool` but not `float`). If the selector evaluates to a value not listed in any of the `case` labels, the `switch` statement will fall through without any action being taken. For this reason, it's often wise to guard the `switch` with a `default` label. This way you can ensure that the `switch` executes properly in all situations. Be very careful in your placement of the `break` statements. Missing `break` statements will allow control to fall through to the next `case` label.

Chapter Review

1. Control structures are used to control the flow of statement execution in a program. The compound statement is a control structure for sequential execution.
2. Selection control structures are used to represent decisions in an algorithm and pseudocode is used to write them in algorithms. Use the `if` statement or `switch` statement to code decision steps in C++.

3. Expressions whose values indicate whether certain conditions are true can be written
   - using the relational operators (`<`, `<=`, `>`, `>=`) and equality operators (`==`, `!=`) to compare variables and constants.
   - using the logical operators (`&&` (and), `||` (or), `!` (not)) to form more complex conditions.
4. Data flow information in a structure chart indicates whether a variable processed by a subproblem is used as an input or an output, or as both. An input provides data that are manipulated by the subproblem, and an output returns a value copied from an input device or computed by the subproblem. The same variable may be an input to one subproblem and an output from another.
5. A hand trace of an algorithm verifies whether it is correct. You can discover errors in logic by carefully hand tracing an algorithm. Hand tracing an algorithm before coding it as a program will save you time in the long run.
6. Nested `if` statements are common in C++ and are used to represent decisions with multiple alternatives. Programmers use indentation and the multiple-alternative decision form when applicable to enhance readability of nested `if` statements.
7. The `switch` statement implements decisions with several alternatives, where the alternative selected depends on the value of a variable or expression (the *selector* expression). The *selector* expression can be type `int`, `bool`, or `char`, but not type `double` or `string`.

Summary of New C++ Constructs

Construct	Effect
`if` Control Statement	
With Dependent Statement	

`if (y != 0)`
`    result = x / y;`
	Divides x by y only if y is nonzero.

Two Alternatives	

`if (x >= 0)`
`    cout << x << " is positive"`
`         << endl;`
`else`
`    cout << x << " is negative"`
`         << endl;`
	If x is greater than or equal to 0, display the message `"is positive"`; otherwise, display the message `"is negative"`.

Summary of New C++ Constructs (continued)

Construct | Effect

Several Alternatives

```
if (x < 0)
{
   cout << "Negative" << endl;
   absX = -x;
}
else if (x == 0)
{
   cout << "Zero" << endl;
   absX = 0;
}
else
{
   cout << "Positive" << endl;
   absX = x;
}
```

One of three messages is printed, depending on whether x is negative, positive, or zero. absX is set to represent the absolute value or magnitude of x.

Prints one of five messages based on the value of nextCh (type char). If nextDh is 'D', 'd' or 'F', 'f', the student is put on probation.

switch Statement

```
switch (nextCh)
{
   case 'A': case 'a':
      cout << "Excellent" << endl;
      break;
   case 'B': case 'b':
      cout << "Good" << endl;
      break;
   case 'C': case 'c':
      cout << "Fair" << endl;
      break;
   case 'D': case 'd':
   case 'F': case 'f':
      cout << "Poor, student is"
           << " on probation"
           << endl;
      break;
   default :
      cout << "Invalid grade."
           << endl;
}
```

Prints one of five messages based on the value of nextCh (type char). If nextDh is 'D', 'd' or 'F', 'f', the student is put on probation.

Quick-Check Exercises

1. An `if` statement implements a(n) ________ in an algorithm or program.
2. What is a compound statement?
3. A `switch` statement is often used instead of ________.
4. What values can a logical expression have?
5. Are the conditions `(x != y)` and `!(x == y)` logically equivalent or different? Explain your answer.
6. A hand trace is used to verify that a(n) ________ is correct.
7. Correct the syntax errors below.

```
if x > 25.0
   y = x
   cout << x << y << endl;
else
   y = z;
   cout << y << z << endl;
```

8. What value is assigned to `fee` by the `if` statement below (a) when `speed` is `50`? (b) When speed is `51`? (c) When speed is `15`? (d) When speed is `130`?

```
if (speed > 35)
   fee = 20;
else if (speed > 50)
   fee = 40;
else if (speed > 75)
   fee = 60;
```

9. Can you implement the `if` statement for question 8 as a `switch` statement? If yes, what are the advantages or disadvantage of doing it?
10. Answer Exercise 8 for the `if` statement below. Which `if` statement is correct?

```
if (speed > 75)
   fee = 60;
else if (speed > 50)
   fee = 40;
else if (speed > 35)
   fee = 20;
else
   fee = 0;
```

11. What output line(s) are displayed by the statements below (a) when `grade` is `'W'`? (b) When `grade` is `'C'`? (c) When `grade` is `'c'`? (d) When `grade` is `'P'`?

```
switch (grade)
{
   case 'A':
      points = 4;
      break;
   case 'B':
      points = 3;
      break;
   case 'C':
      points = 2;
      break;
   case 'D':
      points = 1;
      break;
   case 'F':
   case 'I':
   case 'W':
      points = 0;
      break;
}
if (('A' <= grade) && (grade <= 'D'))
   cout << "Passed, points earned = " << points << "."
        << endl;
else
   cout << "Failed, no points earned." << endl;
```

12. Explain the difference between the statements on the left and the statements on the right below. For each of them, what is the final value of `x` if the initial value of `x` is `0`?

```
if (x >= 0)               if (x >= 0)
   x = x + 1;                x = x + 1;
else if (x >= 1)          if (x >= 1)
   x = x + 2;                x = x + 2;
```

13. a. Evaluate the expression below when `x` is 25:

```
true  &&  (x % 10 >= 0)  &&  (x / 10 <= 3)
```

b. Is either pair of parentheses required? Is the constant `true` required?

c. Write the complement of the expression.

Review Questions

1. Making a decision between two alternatives is usually implemented with a(n) ________ statement in C++.

2. How does a relational operator differ from a logical operator?

3. What is short-circuit logical evaluation? What are its benefits?

4. Trace the following program fragment and indicate which function will be called if a data value of `35.5` is entered.

```
cout << "Enter a temperature: ";
cin >> temp;
if (temp > 32.0)
   notFreezing();
else
   iceForming();
```

5. Write a nested `if` statement to display a message indicating the educational level of a student based on his or her number of years of schooling (0, none; 1 through 6, elementary school; 7 through 8, middle school; 9 through 12, high school; 13 through 16, college; greater than 16, graduate school). Print a message to indicate bad data as well.

6. Write a `switch` statement to select an operation based on the value of `inventory`. Increment `totalPaper` by `paperOrder` if `inventory` is `'b'` or `'c'`; increment `totalRibbon` by `1` if `inventory` is `'e'`, `'f'`, or `'d'`; increment `totalLabel` by `labelOrder` if `inventory` is `'a'` or `'x'`. Do nothing if `inventory` is `'m'`. Display an error message if inventory is any other value.

7. Implement a decision structure for the following computation:
   - If your taxable income is less than or equal to $15,000, pay no taxes.
   - If your taxable income is greater than $15,000 but less than or equal to $44,000, pay taxes at a rate of 20 percent on the amount in excess of $15,000.
   - If your taxable income is greater than $44,000 but less than or equal to $68,150, pay taxes at the rate of 22 percent on the first $44,000 and 28 percent on the amount in excess of $44,000.
   - If your taxable income is greater than $68,150 but less than or equal to $121,300, pay taxes at the rate of 28 percent on the first $58,150 and 31 percent on the amount in excess of $58,150.
   - If your taxable income is greater than $121,300 but less than or equal to $263,750, pay taxes at the rate of 31 percent on the first $121,300 and 36 percent on the amount in excess of $121,300.
   - If your taxable income is greater than $263,750, pay taxes at a rate of 36 percent on the first $263,750 and 39.6 percent on the amount in excess of $263,750.

8. Write a type `bool` function that returns `true` if its argument (a year) is a leap year. A leap year occurs every fourth year. Not every century is a leap year; every *fourth* century is a leap year. This means that 1900 is not a leap year, 2000 is a leap year, and 2100 is not a leap year.

Programming Projects

1. Write functions to draw a circle, square, and triangle. Write a program that reads a letter `C`, `S`, or `T` and, depending on the letter chosen, draws either a circle, square, or triangle.
2. Write a program that reads in four words (as character strings) and displays them in increasing as well as in decreasing alphabetic sequence.
3. Write a program that reads in a room number, its capacity, and the size of the class enrolled so far and prints an output line showing the classroom number, capacity, number of seats filled, number of seats available, and a message indicating whether the class is filled. Call a function to display the following heading before the first output line:

```
Room    Capacity    Enrollment    Empty Seats    Filled/
                                                Not Filled
```

   Display each part of the output line under the appropriate heading. Test your program with the following classroom data:

Room	Capacity	Enrollment
426	25	25
327	18	14
420	20	15
317	100	90

4. Write a program that displays a "message" consisting of three block letters where each letter is an X or an O. The program user's data determines whether a particular letter will be an X or O. For example, if the user enters the three letters XOX, the block letters X, O, and X will be displayed.
5. Write a program to simulate a state police radar gun. The program should read an automobile speed and print the message `"speeding"` if the speed exceeds 55 mph.
6. While spending the summer as a surveyor's assistant, you decide to write a program that transforms compass headings in degrees (0 to 360) to compass bearings. A compass bearing consists of three

items: the direction you face (north or south), an angle between 0 and 90 degrees, and the direction you turn before walking (east or west). For example, to get the bearing for a compass heading of `110.0` degrees, you would first face due south (`180` degrees) and then turn `70.0` degrees east (`180.0 - 110.0`). Be sure to check the input for invalid compass headings.

7. Write a program that interacts with the user like this:

```
(1) Carbon monoxide
(2) Hydrocarbons
(3) Nitrogen oxides
(4) Nonmethane hydrocarbons
Enter pollutant number>> 2
Enter number of grams emitted per mile>> 0.35
Enter odometer reading>> 40112
Emissions exceed permitted level of 0.31 grams/mile.
```

Use the table of emissions limits below to determine the appropriate message.

	First Five Years or 50,000 Miles	Second Five Years or Second 50,000 Miles
carbon monoxide	3.4 grams/mile	4.2 grams/mile
hydrocarbons	0.31 grams/mile	0.39 grams/mile
nitrogen oxides	0.4 grams/mile	0.5 grams/mile
nonmethane hydrocarbons	0.25 grams/mile	0.31 grams/mile

8. The New Telephone Company has the following rate structure for long-distance calls:

- The regular rate for a call is $0.10 per minute.
- Any call started at or after 6:00 P.M. (1800 hours) but before 8:00 A.M. (0800 hours) is discounted 50 percent.
- Any call longer than 60 minutes receives a 15 percent discount on its cost (after any other discount is subtracted).
- All calls are subject to a 4 percent federal tax on their final cost.

Write a program that reads the start time for a call based on a 24-hour clock and the length of the call. The gross cost (before any discounts or tax) should be printed, followed by the net cost (after discounts are deducted and tax is added). Use separate functions to print instructions to the program user and to compute the net cost.

9. Write a program that will calculate and print out bills for the city water company. The water rates vary, depending on whether the bill is for home use, commercial use, or industrial use. A code of `h` means home

use, a code of `c` means commercial use, and a code of `i` means industrial use. Any other code should be treated as an error. The water rates are computed as follows:

- Code `h`: $5.00 plus $0.0005 per gallon used
- Code `c`: $1000.00 for the first 4 million gallons used and $0.00025 for each additional gallon
- Code `i`: $1000.00 if usage does not exceed 4 million gallons; $2000.00 if usage is more than 4 million gallons but does not exceed 10 million gallons; and $3000.00 if usage exceeds 10 million gallons

Your program should prompt the user to enter an account number (type `int`), the code (type `char`), and the gallons of water used (type `float`). Your program should echo the input data and print the amount due from the user.

10. Write a program to control a bread machine. Allow the user to input the type of bread as `W` for White and `S` for Sweet. Ask the user if the loaf size is double and if the baking is manual. The program should fail if the user inputs are invalid. The following table is a time chart for the machine used for each bread type. Print a statement for each step. If the loaf size is double, increase the baking time by 50 percent. If baking is manual, stop after the loaf-shaping cycle and instruct the user to remove the dough for manual baking. Use a function to print program instructions.

Time Chart for Making Bread

Operation	White Bread	Sweet Bread
Primary kneading	15 mins	20 mins
Primary rising	60 mins	60 mins
Secondary kneading	18 mins	33 mins
Secondary rising	20 mins	30 mins
Loaf shaping	2 seconds	2 seconds
Final rising	75 mins	75 mins
Baking	45 mins	35 mins
Cooling	30 mins	30 mins

11. Write a function that returns the digit value (an integer) corresponding to the letter passed to it as an argument based on the encoding on your telephone handset. For example, if the argument is the letter `a`, `b`, or `c` (uppercase or lowercase), your function should return the digit 2. If the argument is not one of the letters of the alphabet, return a value of –1. Write a program that tests your function. Implement two versions of the function: one using a `switch` statement and one using a nested `if` statement. Write a program that tests your functions.

12. Write a function `dayNumber` that returns the day number (1 to 366) in a year for a date that is provided as input data. Your function should accept the month (1 through 12), day, and year as integers. As an example, January 1, 1994 is day 1. December 31, 1993 is day 365. December 31, 1996 is day 366 since 1996 is a leap year. A year is a leap year if it's divisible by four, except that any year divisible by 100 is a leap year only if it's also divisible by 400. Write and use a second function that returns true if its argument, a year, is a leap year.

13. Write a program that reads in two dates (using three integers for each) and displays a message showing the date that comes first. Use the following algorithm: `if` the first date's year is smaller, it comes first; `else if` the second date's year is smaller, it comes first; `else` check the month to determine which date comes first in a similar way. If the months are the same, check the day.

14. Write a function that uses Euclid's algorithm to calculate the greatest common divisor of its two integer arguments where both arguments are positive and the first argument should be greater than or equal to the second. For example, `euclid(10, 2)` is `2`, `euclid(20, 12)` is `4`. Euclid's algorithm `(int euclid(int m, int n))` for finding the greatest common divisor of `m` and `n` follows:

```
if (m is less than n)    // the arguments are in the
                         // wrong order
return the result of euclid(n, m)  // reverse the
                                   // arguments
else if (n divides m)  // n is a divisor of m
   return n
else
   return euclid(n, remainder of m divided by n)
```

The above algorithm is called a recursive algorithm because function `euclid` calls itself (a legal operation in C++). If the first condition is true, the arguments need to be reversed before the greatest common divisor can be found by recalling function `euclid`. If the second condition is true, the smaller argument, `n`, must be the divisor we're seeking. Otherwise, the result can be determined by recalling function `euclid` to find the greatest common divisor of `n` and the remainder of `m` divided by `n`. Verify that this algorithm works for the examples shown in the first paragraph. Then write this function and test it out.

15. Keith's Sheet Music needs a program to implement its music teacher's discount policy. The program is to prompt the user to enter the purchase total and to indicate whether the purchaser is a teacher. The store plans to give each customer a printed receipt, so your

program is to create a nicely formatted file called `receipt.txt`. Music teachers receive a 10% discount on their sheet music purchases unless the purchase total is $100 or higher. In that case, the discount is 12%. The discount calculation occurs before addition of the 5% sales tax. Here are two sample output files—one for a teacher and the other for a nonteacher.

```
Total purchases                    $122.00
Teacher's discount (12%)             14.64
Discounted total                    107.36
Sales tax (5%)                        5.37
Total                              $112.73

Total purchases                    $ 24.70
Sales tax (5%)                        1.25
Total                              $ 26.15
```

16. Write a program that calculates the user's body mass index (BMI) and categorizes it as underweight, normal, overweight, or obese, based on the following table from the United States Centers for Disease Control:

BMI	Weight Status
Below 18.5	Underweight
18.5–24.9	Normal
25.0–29.9	Overweight
30.0 and above	Obese

To calculate BMI based on weight in pounds (*wt_lb*) and height in inches (*ht_in*), use this formula (rounded to tenths):

$$\frac{703 \times wt\_lb}{ht\_in^2}$$

Prompt the user to enter weight in pounds and height in inches.

17. Chatmore Wireless offers customers 600 weekday minutes for a flat rate of 39.99. Night (8 P.M. to 7 A.M.) and weekend minutes are free, but additional weekday minutes cost 0.40 each. There are taxes of 5.25% on all charges. Write a program that prompts the user to enter the number of weekday minutes, night minutes, and weekend minutes used and calculates the monthly bill and average cost of a minute before taxes. The program should display with labels all the input data, the pretax bill and average minute cost, the taxes, and the total bill. Store all monetary values as whole cents (rounding the taxes and average minute cost), and divide by 100 for display of results.

18. The table below shows the normal boiling points of several substances. Write a program that prompts the user for the observed boiling point of a substance in °C and identifies the substance if the observed boiling point is within 5% of the expected boiling point. If the data input is more than 5% higher or lower than any of the boiling points in the table, the program should output the message `Substance unknown`.

Substance	Normal Boiling Point (°C)
Water	100
Mercury	357
Copper	1187
Silver	2193
Gold	2660

Your program should define and call a function `inXperCent` that takes as parameters a reference value `ref`, a data value `data`, and a percentage value `x` and returns 1 meaning true if `data` is within `x % of ref`—that is, `(ref – x% * ref) ≤ data ≤ (ref + x% * ref)`. Otherwise `inXperCent` would return zero, meaning false. For example, the call `inXperCent` (357, 323, 10) would return true, since 10% of 357 is 35.7, and 323 falls between 321.3 and 392.7.

Answers to Quick-Check Exercises

1. decision
2. a block that combines one or more statements into a single statement
3. nested `if` statements or a multiple-alternative `if` statement
4. true or false
5. They are logically equivalent because they always have the same logical value.
6. algorithm
7.
```
if (x > 25.0)
{
   y = x;
   cout << x << y << endl;
}
else
{
   y = z;
   cout << y << z << endl;
}
```

8. 20, 20, undefined, 20
9. You could use speed as a *selector* expression, but it would be impractical because the list of `case` labels would be very long.
10. 20, 40, 0, 60; this one
11. **a.** when `grade` is `'W'`: `Failed, no points earned.`
    b. when `grade` is `'C'`: `Passed, points earned = 2.`
    c. when `grade` is `'c'`: `Failed, no points earned.`
    d. when `grade` is `'P'`: `Failed, no points earned.`
12. A nested `if` statement is on the left; a sequence of `if` statements is on the right. `x` becomes `1` on the left; `x` becomes 3 on the right.
13. **a.** true, remainder is 5 and quotient is 2.
    b. no, no
    c. Removing false and unnecessary parentheses: `!(x % 10 > 0 && x % 10 <= 3)`