Chapter

3

Selection Statements

Objectives

- To declare `bool` type and write Boolean expressions using comparison operators (§3.2).
- To implement selection control using simple `if` statements (§3.3).
- To combine conditions using logical operators (`&&`, `||`, and `!`) (§3.4).
- To implement selection control using `if ... else` statements (§3.5).
- To implement selection control nested `if` statements (§3.6).
- To implement selection control using `switch` statements (§3.9).
- To write expressions using the conditional operator (§3.10).
- To display formatted output using the stream manipulators (§3.11).
- To know the rules governing operand evaluation order, operator precedence, and operator associativity (§3.12).
3.1 Introduction

In Chapter 2, "Primitive Data Types and Operations," if you assigned a negative value for radius in Listing 2.1, ComputeArea.cpp, the program would print an invalid result. If the radius is negative, you don't want the program to compute the area. Like all high-level programming languages, C++ provides selection statements that let you choose actions with two or more alternative courses. You can use selection statements in the following pseudocode (i.e., natural language mixed with programming code) to rewrite Listing 2.1:

```cpp
if the radius is negative
    the program displays a message indicating a wrong input;
else
    the program computes the area and displays the result;
```

Selection statements use conditions. Conditions are Boolean expressions. This chapter first introduces Boolean types, values, operators, and expressions.

3.2 The bool Data Type

Often in a program you need to compare two values, such as whether \( i \) is greater than \( j \). C++ provides six relational operators (also known as comparison operators) in Table 3.1 that can be used to compare two values.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>1 &lt; 2</td>
<td>true</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>1 &lt;= 2</td>
<td>true</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>1 &gt; 2</td>
<td>false</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>1 &gt;= 2</td>
<td>false</td>
</tr>
<tr>
<td>==</td>
<td>equal to</td>
<td>1 == 2</td>
<td>false</td>
</tr>
<tr>
<td>!=</td>
<td>not equal to</td>
<td>1 != 2</td>
<td>true</td>
</tr>
</tbody>
</table>

NOTE

You can also compare characters. Comparing characters is the same as comparing the codes of the characters. For example, 'a' is larger than 'A' because the code of 'a' is larger than the code of 'A'.

CAUTION

The equality comparison operator is two equal signs (==), not a single equal sign (=). The latter symbol is for assignment.
The result of the comparison is a Boolean value: true or false. A variable that holds a Boolean value is known as a Boolean variable. The bool data type is used to declare Boolean variables. For example, the following statement assigns true to the variable lightsOn:

```cpp
bool lightsOn = true;
```

C++ represents the value using a data type called `bool`. Internally, C++ uses 1 to represent true and 0 for false.

For example,

```cpp
cout << (1 < 2);
```

displays 1, because 1 < 2 is true.

```cpp
cout << (1 > 2);
```

displays 0, because 1 > 2 is false.

**NOTE**

In C++, you can assign a numeric value to a `bool` variable. Any nonzero value evaluates true and zero value evaluates false. For example, after the following assignment statements, b1 and b3 become true, and b2 become false.

```cpp
bool b1 = -1.5;
bool b2 = 0;
bool b3 = 1.5;
```

**END NOTE**

### 3.3 if Statements

The programs that you have written so far execute in sequence. Often, however, you are faced with situations in which you must provide alternative paths. C++ provides the if statements, which can be used to control the execution path.

A simple if statement executes an action if and only if the condition is true. The syntax for a simple if statement is shown below:

```cpp
if (booleanExpression)  
{  
    statement(s);
}
```

The execution flow chart is shown in Figure 3.4(a).
An if statement executes statements if the booleanExpression evaluates to true.

If the booleanExpression evaluates to true, the statements in the block are executed. As an example, see the following code:

```cpp
if (radius >= 0)
    area = radius * radius * PI;
    cout << "The area for the circle of " << "radius " << radius << " is " << area;
```

The flow chart of the preceding statement is shown in Figure 3.4(b).

If the value of radius is greater than or equal to 0, then the area is computed and the result is displayed; otherwise, the two statements in the block will not be executed.

**NOTE**

The booleanExpression is enclosed in parentheses for all forms of the if statement. Thus, for example, the outer parentheses in the following if statements are required.

```cpp
if (i > 0) && (i < 10)
    cout << "i is an integer between 0 and 10";
```

Outer parentheses required

```cpp
if ((i > 0) && (i < 10))
    cout << "i is an integer between 0 and 10";
```

Braces can be omitted if the block contains a single statement

The braces can be omitted if they enclose a single statement.

***End of NOTE***
Forgetting the braces when they are needed for grouping multiple statements is a common programming error. If you modify the code by adding new statements in an if statement without braces, you will have to insert the braces if they are not already in place.

***End of CAUTION

Listing 3.1 gives a program that checks whether a number is even or odd. The program prompts the user to enter an integer (line 9) and displays “number is even” if it is even (lines 11-12) and “number is odd” if it is odd (lines 14-15).

```cpp
#include <iostream>
using namespace std;

int main()
{
    // Prompt the user to enter an integer
    int number;
    cout << "Enter an integer: ";
    cin >> number;

    if (number % 2 == 0) // check even
        cout << number << " is even."
    if (number % 2 != 0) // check odd
        cout << number << " is odd."
    return(0);
}
```

Enter an integer: 4
4 is even.
Enter an integer: 5
5 is odd.

CAUTION
Adding a semicolon at the end of an if clause, as shown in (a) in the following code, is a common mistake.
This mistake is hard to find, because it is neither a compilation error nor a runtime error, it is a logic error. The code in (a) is equivalent to (b) with an empty body.

***End of CAUTION (Editor: Please leave this line here to alert the layout person. I will use “End of CAUTION”, “End of NOTE, and End of TIP, for CAUTION, NOTE, and TIP with multiple paragraphs. A)***

### 3.4 Logical Operators

Sometimes, the execution path is determined by a combination of several conditions. You can use logical operators to combine these conditions. Logical operators, also known as Boolean operators, operate on Boolean values to create a new Boolean value. Table 3.2 contains a list of Boolean operators. Table 3.3 defines the not (!) operator. The not (!) operator negates true to false and false to true. Table 3.4 defines the and (&&) operator. The and (&&) of two Boolean operands is true if and only if both operands are true. Table 3.5 defines the or (||) operator. The or (||) of two Boolean operands is true if at least one of the operands is true.

**Table 3.2**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>not</td>
<td>logical negation</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
<td>logical conjunction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 3.3**

<table>
<thead>
<tr>
<th>p</th>
<th>!p</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
<td>!(1 &gt; 2) is true, because (1 &gt; 2) is false.</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>!(1 &gt; 0) is false, because (1 &gt; 0) is true.</td>
</tr>
</tbody>
</table>

**Table 3.4**

**Truth Table for Operator &&**
Table 3.5
Truth Table for Operator ||

| p1    | p2    | p1 || p2 |
|-------|-------|--------|
| false | false | false  |
| false | true  | true   |
| true  | false | true   |
| true  | true  | true   |

Example

(2 > 3) && (5 > 5) is false, because either (2 > 3) or (5 > 5) is false.

(3 > 2) && (5 > 5) is false, because (5 > 5) is false.

(3 > 2) && (5 >= 5) is true, because (3 > 2) and (5 >= 5) are both true.

Listing 3.2 gives a program that checks whether a number is divisible by 2 and 3, whether a number is divisible by 2 or 3, and whether a number is divisible by 2 or 3 but not both:

Listing 3.2 TestBooleanOperators.cpp (Using Boolean Operators)

```cpp
#include <iostream>
using namespace std;

int main()
{
    int number;
    cout << "Enter an integer: ";
    cin >> number;

    if (number % 2 == 0 && number % 3 == 0)
        cout << number << " is divisible by 2 and 3. " << endl;

    if (number % 2 == 0 || number % 3 == 0)
        cout << number << " is divisible by 2 or 3. " << endl;

    if ((number % 2 == 0 || number % 3 == 0) &&
        !(number % 2 == 0 && number % 3 == 0))
        cout << number << " divisible by 2 or 3, but not both." << endl;

    return(0);
}
```

<Output>

94
Enter an integer: 4
4 is divisible by 2 or 3.
4 divisible by 2 or 3, but not both.

Enter an integer: 18
18 is divisible by 2 and 3.
18 is divisible by 2 or 3.

<End Output>

(number % 2 == 0 && number % 3 == 0) (line 10) checks whether the number is divisible by 2 and 3. (number % 2 == 0 || number % 3 == 0) (line 13) checks whether the number is divisible by 2 or 3. So, lines 16-17

!(number % 2 == 0 && number % 3 == 0)

checks whether the number is divisible by 2 or 3, but not both.

If one of the operands of an && operator is false, the expression is false; if one of the operands of an || operands is true, the expression is true. C++ uses these properties to improve the performance of these operators.

<Side Remark: short-circuit operator>
When evaluating p1 && p2, C++ first evaluates p1 and then evaluates p2 if p1 is true; if p1 is false, it does not evaluate p2. When evaluating p1 || p2, C++ first evaluates p1 and then evaluates p2 if p1 is false; if p1 is true, it does not evaluate p2. Therefore, && is referred to as the conditional or short-circuit AND operator, and || is referred to as the conditional or short-circuit OR operator.

CAUTION

<side remark: x < y < z?>
In Mathematics, you can write the condition

x < y < z

But, In C++ you have to write

(x < y && (y < z)

If you write (x < y < z) in C++, C++ first evaluates (x < y) to a Boolean value and then this Boolean value is compared with z, which would lead a logic error.

NOTE

<side remark: bitwise operations>
C++ also supports the bitwise & and | operators. See Appendix G, “Bit Manipulations,” for details.

3.5 if . . . else Statements
A simple if statement takes an action if the specified condition is true. If the condition is false, nothing is done. But what if you want to take alternative actions when the condition is false? You can use an if . . . else statement. The actions that an if . . . else statement specifies differ based on whether the condition is true or false.

Here is the syntax for this type of statement:

```plaintext
if (booleanExpression)
    statement(s)-for-the-true-case;
else
    statement(s)-for-the-false-case;
```

The flow chart of the statement is shown in Figure 3.1.

![Figure 3.1](image)

Figure 3.1
An if . . . else statement executes statements for the true case if the boolean expression evaluates to true; otherwise, statements for the false case are executed.

If the booleanExpression evaluates to true, the statement(s) for the true case is executed; otherwise, the statement(s) for the false case is executed. For example, consider the following code:

```plaintext
<Side Remark line 1: if-else statement>
    if (radius >= 0)
        area = radius * radius * PI;
        cout << "The area for the circle of radius " << radius << " is " << area;
    else
        cout << "Negative input";
</Side Remark line 1: if-else statement>
```

If `radius >= 0` is true, `area` is computed and displayed; if it is false, the message "Negative input" is printed.

As usual, the braces can be omitted if there is only one statement within them. The braces enclosing the `cout << "Negative input"` statement can therefore be omitted in the preceding example.
Using the if ... else statement, you can rewrite the code for determining whether a number is even or odd, as follows:

```c++
if (number % 2 == 0)
    cout << number << " is even."
else
    cout << number << " is odd."
```

This is more efficient because whether `number % 2` is 0 is tested only once.

Listing 3.3 presents a program that lets the user enter a year and checks whether it is a leap year.

```c++
#include <iostream>
using namespace std;

int main()
{
    cout << "Enter a year: ";
    int year;
    cin >> year;

    // Check if the year is a leap year
    bool isLeapYear =
        (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);

    // Display the result
    if (isLeapYear)
        cout << year << " is a leap year."
    else
        cout << year << " is a not leap year.";

    return(0);
}
```

**Output**

Enter a year: 2008
2008 is a leap year.

Enter a year: 2002
2002 is a not leap year.

**End Output**

3.6 Nested if Statements

The statement in an if or if . . . else statement can be any legal C++ statement, including another if or if . . . else statement. The inner if statement is said to be nested inside the outer if statement. The inner if statement can contain another if statement; in fact, there is
no limit to the depth of the nesting. For example, the following is a nested if statement:

```cpp
<Side Remark line 2: nested if statement>
if (i > k)
    if (j > k)
        cout << "i and j are greater than k";
    else
        cout << "i is less than or equal to k";
</Side Remark line 2: nested if statement>
```

The if (j > k) statement is nested inside the if (i > k) statement.

The nested if statement can be used to implement multiple alternatives. The statement given in Figure 3.2(a), for instance, assigns a letter grade to the variable grade according to the score, with multiple alternatives.

```cpp
if (score >= 90.0)
    grade = 'A';
else if (score >= 80.0)
    grade = 'B';
else if (score >= 70.0)
    grade = 'C';
else if (score >= 60.0)
    grade = 'D';
else
    grade = 'F';
```

(a)

This is better

```cpp
if (score >= 90.0)
    grade = 'A';
else if (score >= 80.0)
    grade = 'B';
else if (score >= 70.0)
    grade = 'C';
else if (score >= 60.0)
    grade = 'D';
else
    grade = 'F';
```

(b)

Figure 3.2
A preferred format for multiple alternative if statements is shown in (b).

The execution of this if statement proceeds as follows. The first condition (score >= 90.0) is tested. If it is true, the grade becomes 'A'. If it is false, the second condition (score >= 80.0) is tested. If the second condition is true, the grade becomes 'B'. If that condition is false, the third condition and the rest of the conditions (if necessary) continue to be tested until a condition is met or all of the conditions prove to be false. If all of the conditions are false, the grade becomes 'F'. Note that a condition is tested only when all of the conditions that come before it are false.

The if statement in Figure 3.2(a) is equivalent to the if statement in Figure 3.2(b). In fact, Figure 3.2(b) is the preferred writing style for multiple alternative if statements. This style avoids deep indentation and makes the program easy to read.

NOTE

<Side Remark: matching else with if>
The else clause matches the most recent unmatched if clause in the same block. For example, the following statement in (a) is equivalent to the statement in (b).
int i = 1;
int j = 2;
int k = 3;
if (i > j)
  if (i > k)
    cout << "A";
else
  cout << "B";

Equivalent
This is better with correct indentation

int i = 1;
int j = 2;
int k = 3;
if (i > j)
  if (i > k)
    cout << "A";
else
  cout << "B";

The compiler ignores indentation. Nothing is printed from the statement in (a) and (b). To force the `else` clause to match the first `if` clause, you must add a pair of braces:

```c
int i = 1; int j = 2; int k = 3;
if (i > j)
  if (i > k)
    cout << "A";
else
  cout << "B";
```

This statement prints B.

***End of NOTE

**TIP**

*<Side Remark: assign bool variable>*

Often new programmers write the code that assigns a test condition to a `bool` variable like the code in (a):

```
if (number % 2 == 0)
  even = true;
else
  even = false;
```

**Equivalent**

```
bool even = number % 2 == 0;
```

(a)  (b)

The code can be simplified by assigning the test value directly to the variable, as shown in (b).

***End of TIP

**CAUTION**

*<Side Remark: test Boolean value>*

To test whether a `bool` variable is true or false in a test condition, it is redundant to use the equality comparison operator like the code in (a):

```
if (even == true)
  cout << "It is even.";
```

**Equivalent**

```
if (even)
  cout << "It is even.";
```

(a)  (b)

This is better
Instead, it is better to use the `bool` variable directly, as shown in (b). Another good reason to use the `bool` variable directly is to avoid errors that are difficult to detect. Using the `=` operator instead of the `==` operator to compare equality of two items in a test condition is a common error. It could lead to the following erroneous statement:

```cpp
if (even = true)
    cout << "It is even."
```

This statement does not have syntax errors. It assigns `true` to `even` so that `even` is always true.

***End of CAUTION

TIP

<side remark: debugging in IDE>

If you use an IDE such as C++Builder and Visual C++.NET, please refer to Learning C++ Effectively with C++Builder/Visual C++.NET in the supplements. This supplement shows you how to use a debugger to trace a simple if-else statement.

3.7 Example: Computing Taxes

This section uses nested if statements to write a program to compute personal income taxes. The United States federal personal income tax is calculated based on filing status and taxable income. There are four filing statuses: single filers, married filing jointly, married filing separately, and head of household. The tax rates for 2002 are shown in Table 3.6. If you are, say, single with a taxable income of $10,000, the first $6,000 is taxed at 10% and the other $4,000 is taxed at 15%. So your tax is $1,200.

Table 3.6

2002 U.S. Federal Personal Tax Rates

<table>
<thead>
<tr>
<th>Tax rate</th>
<th>Single filers</th>
<th>Married filing jointly or qualifying widow/widower</th>
<th>Married filing separately</th>
<th>Head of household</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>Up to $6,000</td>
<td>Up to $12,000</td>
<td>Up to $6,000</td>
<td>Up to $10,000</td>
</tr>
<tr>
<td>15%</td>
<td>$6,001 - $27,950</td>
<td>$12,001 - $46,700</td>
<td>$6,001 - $23,350</td>
<td>$10,001 - $37,450</td>
</tr>
<tr>
<td>30%</td>
<td>$67,701 - $141,250</td>
<td>$112,851 - $171,950</td>
<td>$56,426 - $85,975</td>
<td>$96,701 - $156,600</td>
</tr>
<tr>
<td>35%</td>
<td>$141,251 - $307,050</td>
<td>$171,951 - $307,050</td>
<td>$85,976 - $153,528</td>
<td>$156,601 - $307,050</td>
</tr>
<tr>
<td>38.6%</td>
<td>$307,051 or more</td>
<td>$307,051 or more</td>
<td>$153,526 or more</td>
<td>$307,051 or more</td>
</tr>
</tbody>
</table>

Your program should prompt the user to enter the filing status and taxable income and computes the tax for the year 2002. Enter 0 for single filers, 1 for married filing jointly, 2 for married filing...
separately, and 3 for head of household. A sample run of the program is shown as follows:

<Output>
Enter the filing status  
(0-single filer, 1-married jointly, 
2-married separately, 3-head of household): 0 
Enter the taxable income: 400000 
Tax is 130599 

<End Output>

Your program computes the tax for the taxable income based on the filing status. The filing status can be determined using if statements outlined as follows:

```cpp
if (status == 0)
// Compute tax for single filers
else if (status == 1)
// Compute tax for married file jointly
else if (status == 2)
// Compute tax for married file separately
else if (status == 3)
// Compute tax for head of household
else
// Display wrong status
```

For each filing status, there are six tax rates. Each rate is applied to a certain amount of taxable income. For example, of a taxable income of $400,000 for single filers, $6,000 is taxed at 10%, (27950 – 6000) at 15%, (67700 – 27950) at 27%, (141250 – 67700) at 35%, and (400000 – 307050) at 38.6%.

Listing 3.4 gives the solution to compute taxes for single filers. The complete solution is left as exercise.

Listing 3.4 ComputeTaxWithSelectionStatement.cpp (Computing Tax)

```cpp
***PD: Please add line numbers (including space lines) in the following code***

This is true for all source code in the book. Thanks, A.

<Side Remark line 11: enter status>
<Side Remark line 16: enter income>
<Side Remark line 19: compute tax>
<Side Remark line 65: display tax>

#include <iostream>
using namespace std;

int main()
{
    // Prompt the user to enter filing status
    cout << "Enter the filing status\n" 
        << "(0-single filer, 1-married jointly,\n" 
        << "2-married separately, 3-head of household): ";
```
int status;
    cin >> status;

    // Prompt the user to enter taxable income
    cout << "Enter the taxable income: ";
    double income;
    cin >> income;

    // Compute tax
    double tax = 0;

    if (status == 0)
    {
        // Compute tax for single filers
        if (income <= 6000)
        {
            tax = income * 0.10;
        }
        else if (income <= 27950)
        {
            tax = 6000 * 0.10 + (income - 6000) * 0.15;
        }
        else if (income <= 67700)
        {
            tax = 6000 * 0.10 + (27950 - 6000) * 0.15 +
                   (income - 27950) * 0.27;
        }
        else if (income <= 141250)
        {
            tax = 6000 * 0.10 + (27950 - 6000) * 0.15 +
                   (67700 - 27950) * 0.27 +
                   (income - 67700) * 0.30;
        }
        else if (income <= 307050)
        {
            tax = 6000 * 0.10 + (27950 - 6000) * 0.15 +
                   (67700 - 27950) * 0.27 +
                   (141250 - 67700) * 0.30 +
                   (income - 141250) * 0.35;
        }
        else
        {
            tax = 6000 * 0.10 + (27950 - 6000) * 0.15 +
                   (67700 - 27950) * 0.27 +
                   (141250 - 67700) * 0.30 +
                   (307050 - 141250) * 0.35 +
                   (income - 307050) * 0.386;
        }
    }
    else if (status == 1)
    {
        // Compute tax for married file jointly
        // Left as exercise
    }
    else if (status == 2)
    {
        // Compute tax for married separately
        // Left as exercise
    }
    else if (status == 3)
    {
        // Compute tax for head of household
        // Left as exercise
    }
    else
    {
        cout << "Error: invalid status";
        return (0);
    }

    // Display the result
    cout << "Tax is " << static_cast<int>(tax * 100) / 100.0 << endl;
    return (0);
}

3.8 Example: A Math Learning Tool

This example creates a program for a first grader to practice subtractions. The program randomly generates two single-digit integers number1 and number2 with number1 > number2 and displays a question such as "What is 9 − 2?" to the student, as shown in the sample output.
After the student types the answer, the program displays a message to indicate whether the answer is correct. A sample run of the program is shown as follows:

```
<Output>
What is 5 - 2? 3
You are correct!

What is 4 - 2? 1
Your answer is wrong.
4 - 2 should be 2
<End Output>
```

**Side Remark: rand function**

To generate a random number, use the `rand()` function in the `cmath` library. This function returns a random integer. In fact, the numbers produced by `rand()` are pseudorandom, i.e., it produces the same sequence of numbers every time it is executed on the same system. For example, executing these three statements will get the same numbers 130, 10982, and 1090 every time on my machine.

```
cout << rand() << endl;
cout << rand() << endl;
cout << rand() << endl;
```

**Side Remark: srand function**

Why? The reason is that the algorithm used by the `rand()` function uses a seed value to control how to generate the numbers. By default the seed value is 1. If you change the seed to a different value, the sequence of random numbers will be different. To change the seed, use the `srand(seed)` function in the `cmath` library. To ensure that the seed value is different each time you run the program, use `time(0)`. As discussed in §2.12.3, “Example: Displaying the Current Time,” invoking `time(0)` returns the current time in seconds elapsed since the time 00:00:00 on January 1, 1970 GMT. So, the following code will display a random integer every second you run it on any machine.

```
srand(time(0));
cout << rand() << endl;
```

To obtain a random integer between 0 and 9, use `rand() % 10`.

The program may work as follows:

Step 1: Generate two single-digit integers into `number1` and `number2`.
Step 2: If `number1 < number2`, swap `number1` with `number2`.
Step 3: Prompt the student to answer “what is number1 – number2?”
Step 4: Check the student’s answer and display whether the answer is correct.

The complete program is shown in Listing 3.5.

```
Listing 3.5 SubtractionTutor.cpp (Practice Subtraction)
***PD: Please add line numbers (including space lines) in the following code***
```
```cpp
#include <iostream>
#include <ctime> // for time function
#include <cmath> // for rand function
using namespace std;

int main()
{
    // 1. Generate two random single-digit integers
    srand(time(0));
    int number1 = rand() % 10;
    int number2 = rand() % 10;

    // 2. If number1 < number2, swap number1 with number2
    if (number1 < number2)
    {
        int temp = number1;
        number1 = number2;
        number2 = temp;
    }

    // 3. Prompt the student to answer "what is number1 - number2?"
    cout << "What is " << number1 << " - " << number2 << "? ";
    int answer;
    cin >> answer;

    // 4. Grade the answer and display the result
    if (number1 - number2 == answer)
        cout << "You are correct!";
    else
        cout << "Your answer is wrong. \n" << number1 << " - " << number2
              << " should be " << (number1 - number2) << endl;

    return 0;
}
```

To swap two variables `number1` and `number2`, a temporary variable `temp` (line 15) is used to first hold the value in `number1`. The value in `number2` is assigned to `number1` (line 16) and the value in `temp` is assigned to `number2` (line 17).

### 3.9 switch Statements

The `if` statement in Listing 3.4 makes selections based on a single `true` or `false` condition. There are four cases for computing taxes, which
depend on the value of status. To fully account for all the cases, nested if statements were used. Overuse of nested if statements makes a program difficult to read. C++ provides a switch statement to handle multiple conditions efficiently. You could write the following switch statement to replace the nested if statement in Listing 3.4:

```cpp
switch (status)
{
    case 0:  compute taxes for single filers;
             break;
    case 1:  compute taxes for married file jointly;
             break;
    case 2:  compute taxes for married file separately;
             break;
    case 3:  compute taxes for head of household;
             break;
    default: cout << "Errors: invalid status" << endl;
}
```

The flow chart of the preceding switch statement is shown in Figure 3.3.

![Flow chart of switch statement](image)

**Figure 3.3**
The switch statement checks all cases and executes the statements in the matched case.

This statement checks to see whether the status matches the value 0, 1, 2, or 3, in that order. If matched, the corresponding tax is computed; if not matched, a message is displayed. Here is the full syntax for the switch statement:

```cpp
<Side Remark line 1: switch statement>
switch (switch-expression)
{
    case value1: statement(s)1;
                 break;
    case value2: statement(s)2;
                 break;
    …
    case valueN: statement(s)N;
                 break;
    default:     statement(s)-for-default;
}
```

The switch statement observes the following rules:
• The switch-expression must yield an integral value, and must always be enclosed in parentheses.
• The value1, ..., and valueN are integral constant expressions, meaning that they cannot contain variables in the expression, such as \( 1 + x \).
• When the value in a case statement matches the value of the switch-expression, the statements starting from this case are executed until either a break statement or the end of the switch statement is reached.
• The keyword break is optional. The break statement immediately ends the switch statement.
• The default case, which is optional, can be used to perform actions when none of the specified cases matches the switch-expression.
• The case statements are checked in sequential order, but the order of the cases (including the default case) does not matter. However, it is good programming style to follow the logical sequence of the cases and place the default case at the end.

**NOTE**  
*Side Remark: integral value*  
In C++, a char or bool value is treated as integral. So, this type of value can be used in a switch statement as switch expression or case value.

**CAUTION**  
*Side Remark: without break*  
*Side Remark: fall-through behavior*  
Do not forget to use a break statement when one is needed. Once a case is matched, the statements starting from the matched case are executed until a break statement or the end of the switch statement is reached. This phenomenon is referred to as the fall-through behavior. For example, the following code prints character a three times if \( ch \) is 'a':
**End of CAUTION**

**TIP**

To avoid programming errors and improve code maintainability, it is a good idea to put a comment in a case clause if `break` is purposely omitted.

### 3.10 Conditional Expressions

You might want to assign a value to a variable that is restricted by certain conditions. For example, the following statement assigns 1 to `y` if `x` is greater than 0, and -1 to `y` if `x` is less than or equal to 0.

```cpp
if (x > 0)
    y = 1;
else
    y = -1;
```

Alternatively, as in this example, you can use a conditional expression to achieve the same result.

```cpp
y = (x > 0) ? 1 : -1;
```

Conditional expressions are in a completely different style, with no explicit `if` in the statement. The syntax is shown below:

```cpp
<Side Remark line 1: conditional expression>
booleanExpression ? expression1 : expression2;
```

The result of this conditional expression is `expression1` if `booleanExpression` is true; otherwise the result is `expression2`.

Suppose you want to assign the larger number between variable `num1` and `num2` to `max`. You can simply write a statement using the conditional expression:

```cpp
max = (num1 > num2) ? num1 : num2;
```

For another example, the following statement displays the message “num is even” if `num` is even, and otherwise displays “num is odd.”

```cpp
cout << ((num % 2 == 0) ? "num is even" : "num is odd");
```

**NOTE**

The symbols `?` and `:` appear together in a conditional expression. They form a conditional
operator. This operator is called a ternary operator because it uses three operands. It is the only ternary operator in C++.

3.11 Formatting Output

You already know how to display console output using the `std::cout` object. C++ provides additional functions for formatting how a value is printed. These functions are called stream manipulators and are included in the `<iomanip>` header file. Table 3.7 summarizes several useful stream manipulators.

**Frequently used Stream Manipulator**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setw(width)</code></td>
<td>specifies the width of a print field</td>
</tr>
<tr>
<td><code>setprecision(n)</code></td>
<td>sets the precision of a floating-point number</td>
</tr>
<tr>
<td><code>fixed</code></td>
<td>displays floating-point numbers in fixed point notation</td>
</tr>
<tr>
<td><code>showpoint</code></td>
<td>causes a floating-point number to be displayed with a decimal point with trailing zeros even if it has no fractional part</td>
</tr>
<tr>
<td><code>left</code></td>
<td>justifies the output to the left</td>
</tr>
<tr>
<td><code>right</code></td>
<td>justifies the output to the right</td>
</tr>
</tbody>
</table>

**Side Remark: `setw(width)` manipulator**

By default, `cout` uses just the number of the positions needed for an output. You can use `setw(width)` to specify the minimum number of positions for an output. For example, the statements

```cpp
cout << setw(8) << "C++" << setw(6) << 101 << endl;
cout << setw(8) << "Java" << setw(6) << 101 << endl;
cout << setw(8) << "HTML" << setw(6) << 101 << endl;
```

display

<table>
<thead>
<tr>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
</tr>
<tr>
<td>Java</td>
</tr>
<tr>
<td>HTML</td>
</tr>
</tbody>
</table>

**End Output**

In line 1, `setw(8)` specifies that "C++" is printed in eight position. So, there are five spaces before C++. `setw(6)` specifies that 101 is printed in six position. So, there are three spaces before 101.

**Side Remark: `setprecision(n)` manipulator**

You can specify the total number of digits printed for a floating-point number using the `setprecision(n)` manipulator, where `n` is the total number of digits that appear before and after the decimal point. If a number to be displayed has more digits than the specified precision, the number will be rounded. For example, the statement

```cpp
double number = 12.34567;
```
cout << setw(10) << setprecision(5) << number;
cout << setw(10) << setprecision(4) << number;
cout << setw(10) << setprecision(3) << number;
cout << setw(10) << setprecision(8) << number;

<Output>
  12.346    12.35    12.3    12.34567
<End Output>

The first value is displayed with the precision 5. So, 12.34567 is rounded to 12.346. The second value is displayed with the precision 4. So, 12.34567 is rounded to 12.35. The second value is displayed with the precision 3. So, 12.34567 is rounded to 12.3. The precision for the last value is set to 8, which is more than the number of the digits in 12.3457. In this case, the precision manipulator has no effect.

NOTE:

Unlike the setw manipulator, the setprecision manipulator remains in effect until it is changed to a new precision. So,

double number = 12.34567;
cout << setw(10) << setprecision(3) << number;
cout << setw(10) << number;
cout << setw(10) << number;

<Output>
  12.3    12.3    12.3
<End Output>

The precision is set to 3 for the first value, and it remains effective for the subsequent two values, because it has not been changed.

<Side Remark: fixed manipulator>

Sometimes, the computer automatically displays a large floating-point number in scientific notation. For example, on my Windows machine, the statement

cout << fixed << 232123434.357;

displays

<Output>
  2.32123e+08
<End Output>

You can use the fixed manipulator to force the number to be displayed in non-scientific notation with fixed number of digits after the decimal point. For example, the statement

cout << fixed << 232123434.357;

109
By default, the fixed number of digits after the decimal point is 6. You can change it using the fixed manipulator along with the setprecision manipulator. When the setprecision manipulator is used after the fixed manipulator, the setprecision manipulator specifies the number of digits after the decimal point. For example, the following statements

```cpp
double monthlyPayment = 345.4567;
double totalPayment = 78676.887234;
cout << fixed << setprecision(2);
cout << setw(10) << monthlyPayment << endl;
cout << setw(10) << totalPayment << endl;
```

display

```
345.46
78676.89
```

The `setprecision(2)` manipulator after fixed manipulator specifies that the precision after the decimal point is 2. So, `monthlyPayment` is displayed as 345.46 and `totalPayment` is displayed as 78676.90. Once the fixed manipulator is used, it applied to all subsequent floating-point numbers to be displayed.

**Side Remark: showpoint manipulator**

By default, floating-point numbers that do not have a fractional part are not displayed with a decimal point. You can use the fixed manipulator to force the floating-point numbers to be displayed with a decimal point and a fixed number of digits after the decimal point. Alternatively, you can use the showpoint manipulator together with the setprecision manipulator.

For example, the following statements

```cpp
cout << setprecision(6);
cout << 1.23 << endl;
cout << showpoint << 1.23 << endl;
cout << showpoint << 123.0 << endl;
```

display

```
1.23
1.23000
123.000
```
The `setprecision(6)` method sets the precision to 6. So, the first number 1.23 is displayed as 1.23. Because the `showpoint` manipulator forces the floating-point number to be displayed with a decimal point and trailing zeros if necessary to fill in the positions, the second number 1.23 is displayed as 1.23000 with trailing zero, and the third number 123.0 is displayed as 123.000 with a decimal point and trailing zero.

`<Side Remark: left manipulator>`

You can use the `left` manipulator to left-justify the output and use the `right` manipulator to right-justify the output. On most systems, the default justification is right. For example,

```
cout << right;
cout << setw(8) << 1.23 << endl;
cout << setw(8) << 351.34 << endl;
```

display

```
1.23
351.34
```

`<End Output>`

`<Side Remark: right manipulator>`

```
cout << left;
cout << setw(8) << 1.23 << endl;
cout << setw(8) << 351.34 << endl;
```

display

```
1.23
351.34
```

`<End Output>`

3.12 Operator Precedence and Associativity

Operator precedence and associativity determine the order in which operators are evaluated. Suppose that you have this expression:

```
3 + 4 * 4 > 5 * (4 + 3) - 1
```

`<Side Remark: precedence>`

What is its value? How does the compiler know the execution order of the operators? The expression in the parentheses is evaluated first. (Parentheses can be nested, in which case the expression in the inner parentheses is executed first.) When evaluating an expression without parentheses, the operators are applied according to the precedence rule and the associativity rule. The precedence rule defines precedence for operators, as shown in Table 3.8, which contains the operators you have learned so far. Operators are listed in decreasing order of precedence from top to bottom. Operators with the same precedence appear in the same group. (See Appendix C, "Operator Precedence Chart," for a complete list of C++ operators and their precedence.)

Table 3.8
Operator Precedence Chart

Precedence Operator
---
1. var++, var-- (postfix), static_cast<type>() (cast)
2. +, - (Unary plus and minus), ++var and --var (prefix)
3. ! (Not)
4. *, /, % (Multiplication, division, and modulus)
5. <, <=, >, >= (Comparison)
6. =, +=, -=, *=, /=, %= (Assignment operator)

**Side Remark: associativity**
If operators with the same precedence are next to each other, their associativity determines the order of evaluation. All binary operators except assignment operators are left-associative. For example, since + and - are of the same precedence and are left-associative, the expression

\[ a - b + c - d \]

is equivalent to

\[ ((a - b) + c) - d \]

Assignment operators are right-associative. Therefore, the expression

\[ a = b += c = 5 \]

is equivalent to

\[ a = (b += (c = 5)) \]

Suppose \( a, b, \) and \( c \) are 1 before the assignment; after the whole expression is evaluated, \( a \) becomes 6, \( b \) becomes 6, and \( c \) becomes 5. Note that left associativity for the assignment operator would not make sense.

Applying the operator precedence and associativity rule, the expression

\[ 3 + 4 * 4 > 5 * (4 + 3) - 1 \]

is evaluated as follows:

1. \[ 3 + 4 \]
2. \[ 3 + 16 \]
3. \[ 19 > 35 \]
4. \[ 19 > 34 \]
5. \[ false \]

**TIP**
You can use parentheses to force an evaluation order as well as to make a program easy to read. Use of redundant parentheses does not slow down the execution of the expression.

***End of NOTE***

3.13 Enumerated Types
You have used numeric type, char type, and bool type to declare variables. C++ enables you to declare your own type, known as enumerated type, using the enum keyword. For example,
<Side Remark: define enumerated type>

```cpp
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };  
```

declares an enumerated type named `Day` with possible values `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, and `FRIDAY` in this order.

An enumerated type defines a list of enumerated values. Each value is an identifier, not a string. The identifiers are known to the program once they are declared in the type.

<Side Remark: naming convention>

By convention, an enumerated type is named with first letter of each word capitalized and a value of an enumerated type is named like a constant with all uppercase letters.

Once a type is defined, you can declare a variable of that type:

```cpp
Day day;  
```

The variable `day` can hold one of the values defined in the enumerated type. For example, the following statement assigns enumerated value `MONDAY` to variable `day`:

```cpp
day = MONDAY;  
```

As with any other type, you can declare and initialize a variable in one statement. For example,

```cpp
Day day = MONDAY;  
```

Furthermore, C++ allows you to declare an enumerated type and variable in one statement. For example,

```cpp
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY } day = MONDAY;  
```

**CAUTION**

<side remark: declare only once>

An enumerated value can not be redeclared. For example, the following code would cause syntax error.

```cpp
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };  
const int MONDAY = 0; // Error: MONDAY already declared.  
```

***END CAUTION***

An enumerated variable holds a value. Often your program needs to perform a specific action depending on the value. For example, if the value is `MONDAY`, play soccer; if the value is `TUESDAY`, take piano lesson, and so on. You can use an `if` statement or a `switch` statement to test the value in the variable, as shown in (a) and (b)
if (day == MONDAY) {
    // process Monday
} else if (day == TUESDAY) {
    // process Tuesday
} else 
...

Equivalent

switch (day) {
    case MONDAY:
        // process Monday
        break;
    case TUESDAY:
        // process Tuesday
        break;
    ...
}

<Side Remark: integers>
Enumerated values are stored as integers in memory. By default, the values correspond to 0, 1, 2, ..., in the order of their appearance in the list. So, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, and FRIDAY correspond to the integer values 0, 1, 2, 3, and 4. You can explicitly assign an enumerated value with any integer value. For example,

```cpp
enum Color {RED = 20, GREEN = 30, BLUE = 40};
```

RED has an integer value 20, GREEN 30, and BLUE 40.

If you assign integer values for some values in the enumerated type declaration, the other values will receive default values. For example,

```cpp
enum City {PARIS, LONDON, DALLAS = 30, HOUSTON};
```

PARIS will be assigned 0, LONDON 1, DALLAS 30, and HOUSTON 31.

You can assign an enumerated value to an integer variable. For example,

```cpp
int i = PARIS;
```

This assigns 0 to i.

<Side Remark: comparing values>
Enumerated values can be compared using the six comparison operators. For example, (PARIS < LONDON) yields true.

Listing 3.6 gives an example of using enumerated types.

Listing 3.6 TestEnumeratedType.cpp (Using Enumerated Type)

```cpp
#include <iostream>
using namespace std;

int main() {
    enum Day {MONDAY = 1, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY} day;
    cout << "Enter a day (1 for Monday, 2 for Tuesday, etc): ";
    int dayNumber;
    cin >> dayNumber;
    switch (dayNumber) {
        case MONDAY:
            // process Monday
            break;
        case TUESDAY:
            // process Tuesday
            break;
        ...
    }
}
```

<Side Remark line 6: declare enumerated type>
<Side Remark line 10: enter an integer>
<Side Remark line 12: check values>
cout << "Play soccer" << endl;
break;
case TUESDAY:
    cout << "Piano lesson" << endl;
    break;
case WEDNESDAY:
    cout << "Math team" << endl;
    break;
default:
    cout << "Go home" << endl;
}
return 0;

<Output>
Enter a day (1 for Monday, 2 for Tuesday, etc): 1
Play soccer

Enter a day (1 for Monday, 2 for Tuesday, etc): 4
Go home
<End Output>

Line 6 declares an enumerated type Day and declares a variable named day in one statement. Line 10 reads an int value from the keyboard. The switch statement in lines 12–24 checks whether day is MONDAY, TUESDAY, WEDNESDAY, or others to display a message accordingly.

Key Terms
***PD: Please place terms in two columns same as in intro5e.

- Boolean expression 47
- Boolean value 47
- bool type 47
- break statement 89, 101
- conditional operator 91
- enumerated type
- fall-through behavior 90
- operand evaluation order 51
- operator associativity 50
- operator precedence 50
- selection statement 81
- short-circuit evaluation 49

Chapter Summary

- The Boolean operators &&, ||, and ! operate with Boolean values and variables. The relational operators (<, <=, ==, !=, >, >=) work with numbers and characters, and yield a Boolean value.
- When evaluating p1 && p2, C++ first evaluates p1 and then evaluates p2 if p1 is true; if p1 is false, it does not evaluate p2. When evaluating p1 || p2, C++ first evaluates p1 and then evaluates p2 if p1 is false; if p1 is true, it does not evaluate p2.
• Selection statements are used for building selection steps into programs. There are several types of selection statements: if statements, if . . . else statements, nested if statements, switch statements, and conditional expressions.

• The various if statements all make control decisions based on a Boolean expression. Based on the true or false evaluation of that expression, these statements take one of two possible courses.

• The switch statement makes control decisions based on a switch expression of type char, byte, short, or int.

• The keyword break is optional in a switch statement, but it is normally used at the end of each case in order to terminate the remainder of the switch statement. If the break statement is not present, the next case statement will be executed.

• The operands of a binary operator are evaluated from left to right. No part of the right-hand operand is evaluated until all the operands before the binary operator are evaluated.

• The operators in arithmetic expressions are evaluated in the order determined by the rules of parentheses, operator precedence, and associativity.

• Parentheses can be used to force the order of evaluation to occur in any sequence. Operators with higher precedence are evaluated earlier. The associativity of the operators determines the order of evaluation for operators of the same precedence.

• All binary operators except assignment operators are left-associative, and assignment operators are right-associative.

Review Questions

Section 3.2 The bool Data Type

3.1 List six comparison operators.

3.2 Assume that \( x \) is 1, show the result of the following Boolean expressions.

\[
\begin{align*}
(x > 0) \\
(x < 0) \\
(x != 0) \\
(x >= 0) \\
(x != 1)
\end{align*}
\]

Section 3.3 if Statements
3.3
What is wrong in the following code?

```cpp
if radius >= 0
    area = radius * radius * PI;
    cout << "The area for the circle of " << radius << " is " << area;
```

3.4 Logical Operators

Assume that \( x \) is 1, show the result of the following Boolean expressions.

- \((true) \&\& (3 > 4)\)
- \((x > 0) \&\& (x < 0)\)
- \((x != 0) || (x == 0)\)
- \((x > 0) || (x < 0)\)
- \((x != 1) == !(x == 1)\)

3.5
Write a Boolean expression that evaluates to \textit{true} if a number stored in variable \texttt{num} is between 1 and 100.

3.6
Write a Boolean expression that evaluates to \textit{true} if a number stored in variable \texttt{num} is between 1 and 100 or the number is negative.

3.7
Assume that \( x \) and \( y \) are \texttt{int} type. Which of the following are correct expressions?

- \(x > y > 0\)
- \(x = y && y\)
- \(x /= y\)
- \(x or y\)
- \(x and y\)
- \((x != 0) || (x = 0)\)

3.8
Can the following conversions involving casting be allowed? If so, find the converted result.

```cpp
bool b = true;
int i = b;
i = 1;
b = i;
```

3.9
Suppose that \( x \) is 1. What is \( x \) after the evaluation of the following expression?

\[(x > 1) \land \land (x++ > 1)\]

3.10
Show the output of the following program:

```cpp
#include <iostream>
using namespace std;

int main()
{
    char x = 'a';
    char y = 'c';

    cout << ++x << endl;
    cout << y++ << endl;
    cout << (x > y) << endl;
    cout << (x - y) << endl;

    return 0;
}
```

Section 3.6 Nested if Statements

3.11
Suppose \( x = 3 \) and \( y = 2 \), show the output, if any, of the following code. What is the output if \( x = 3 \) and \( y = 4 \)? What is the output if \( x = 2 \) and \( y = 2 \)? Draw a flowchart of the following code.

```cpp
if (x > 2)
{
    if (y > 2)
    {
        int z = x + y;
        cout << "z is " << z << endl;
    }
    else
    {
        cout << "y is " << y << endl;
    }
}
else
    cout << "x is " << x << endl;
```

3.12
Which of the following statements are equivalent? Which ones are correctly indented?

(a) `if (i > 0) if (j > 0) x = 0; else if (k > 0) y = 0; else z = 0;`

(b) `if (i > 0) {
    if (j > 0) x = 0;
    else if (k > 0) y = 0;
} else
    z = 0;`

(c) `if (i > 0) if (j > 0) x = 0;
else if (k > 0) y = 0;
else
    z = 0;`

(d) `if (i > 0)
    if (j > 0)
        x = 0;
    else if (k > 0)
        y = 0;
else
    z = 0;`

3.13
Suppose \( x = 2 \) and \( y = 3 \), show the output, if any, of the following code. What is the output if \( x = 3 \) and \( y = 2 \)? What
is the output if \( x = 3 \) and \( y = 3 \)? (Hint: please indent the statement correctly first.)

```c
if (x > 2)
    if (y > 2)
        int z = x + y;
        cout << "z is " << z << endl;
    else
        cout << "x is " << x << endl;
```

3.14
Are the following two statements equivalent?

```c
if (income <= 10000)
    tax = income * 0.1;
else if (income <= 20000)
    tax = 1000 +
        (income - 10000) * 0.15;
```

3.15
Which of the following is a possible output from invoking `rand()`?

- 323.4, 5, 34, 1, 0.5, 0.234

3.16
How do you generate a random integer \( i \) such that \( 0 \leq i < 20 \)?
How do you generate a random integer \( i \) such that \( 10 \leq i < 50 \)?

### Section 3.9 switch Statements

3.17
What data types are required for a `switch` variable? If the keyword `break` is not used after a case is processed, what is the next statement to be executed? Can you convert a `switch` statement to an equivalent `if` statement, or vice versa? What are the advantages of using a `switch` statement?

3.18
What is \( y \) after the following `switch` statement is executed?

```c
x = 3; y = 3;
switch (x + 3)
{
    case 6: y = 1;
    default: y += 1;
}
```

3.19
Use a `switch` statement to rewrite the following `if` statement and draw the flowchart for the `switch` statement:

```c
if (a == 1)
    x += 5;
else if (a == 2)
    x += 10;
else if (a == 3)
    x += 16;
else if (a == 4)
    x += 34;
```
Section 3.10 Conditional Expressions

3.20 Rewrite the following if statement using the conditional operator:

```cpp
if (count % 10 == 0)
    cout << count << "\n";
else
    cout << count << " ";
```

Section 3.11 Formatting Output

3.21 To use stream manipulators, which header file must be included?

3.22 Show the output of the following statements.

```cpp
cout << setw(10) << "C++" << setw(6) << 101 << endl;
cout << setw(8) << "Java" << setw(5) << 101 << endl;
cout << setw(6) << "HTML" << setw(4) << 101 << endl;
```

3.23 Show the output of the following statements.

```cpp
double number = 93123.1234567;
cout << setw(10) << setprecision(5) << number;
cout << setw(10) << setprecision(4) << number;
cout << setw(10) << setprecision(3) << number;
cout << setw(10) << setprecision(8) << number;
```

3.24 Show the output of the following statements.

```cpp
double x = 1345.4567;
double y = 866.887234;
cout << fixed << setprecision(2);
cout << setw(8) << monthlyPayment << endl;
cout << setw(8) << totalPayment << endl;
```

3.25 Show the output of the following statements.

```cpp
cout << right;
cout << setw(6) << 21.23 << endl;
cout << setw(6) << 51.34 << endl;
```

3.26 Show the output of the following statements.

```cpp
cout << left;
cout << setw(6) << 21.23 << endl;
cout << setw(6) << 51.34 << endl;
```
Section 12 Operator Precedence and Associativity

3.27
List the precedence order of the Boolean operators.
Evaluate the following expressions:

```
true || true && false
true && true || false
```

3.28
Show and explain the output of the following code:

a.
```cpp
int i = 0;
ocout << (--i + i + i++) << endl;
ocout << (i + ++i) << endl;
```

b.
```cpp
int i = 0;
i = i + (i - i);
ocout << i << endl;
```

c.
```cpp
int i = 0;
i = (1 - i) + i;
ocout << i << endl;
```

3.29
Assume that int a = 1 and double d = 1.0, and that each expression is independent. What are the results of the following expressions?

```
a = (a - 3) + a;
a = a + (a = 3);
a += a + (a = 3);
a = 5 + 5 * 2 % a--;
a = 4 + 1 + 4 * 2 % (++a + 1);
d += 1.5 * 3 + (++d);
d -= 1.5 * 3 + d++;
```

Programming Exercises

Pedagogical NOTE

<side remark: think before coding>
For each exercise, students should carefully analyze the problem requirements and design strategies for solving the problem before coding.

Pedagogical NOTE

<side remark: document analysis and design>
Instructors may ask students to document analysis and design for selected exercises. Students should use their own words to analyze the problem including the input, output, and what needs to be computed and describe how to solve the problem using English or pseudo code.

Debugging TIP

<side remark: learn from mistakes>
Before you ask for help, read and explain the program to yourself, and trace it using several representative input by hand or using an IDE debugger. You learn how to program from debugging your own mistakes.

**Sections 3.2-3.11**

3.1* *(Validating triangles)* Write a program that reads three edges for a triangle and determines whether the input is valid. The input is valid if the sum of any two edges is greater than the third edge. For example, if your input for three edges is 1, 2, 1, the output should be:

> Can edges 1, 2, and 1 form a triangle? false

If your input for three edges is 2, 2, 1, the output should be:

> Can edges 2, 2, and 1 form a triangle? true

3.2 *(Checking whether a number is even)* Write a program that reads an integer and checks whether it is even. For example, if your input is 25, the output should be:

> Is 25 an even number? false

If your input is 2000, the output should be:

> Is 2000 an even number? true

3.3* *(Using the `&&`, `||` and `^` operators)* Write a program that prompts the user to enter an integer and determines whether it is divisible by 5 and 6, whether it is divisible by 5 or 6, and whether it is divisible by 5 or 6, but not both. For example, if your input is 10, the output should be

> Is 10 divisible by 5 and 6? false
> Is 10 divisible by 5 or 6? true
> Is 10 divisible by 5 or 6, but not both? true

**Section 3.3 Selection Statements**

3.4 *(Monetary units)* Modify Listing 2.10, ComputeChange.cpp, to display the non-zero denominations only, using singular words for single units like 1 dollar and 1 penny, and plural words for more than one unit like 2 dollars and 3 pennies. (Use 23.67 to test your program.)

3.5* *(Sorting three integers)* Write a program that sorts three integers. The integers are entered from the console and
stored in variables $\text{num1}$, $\text{num2}$, and $\text{num3}$, respectively. The program sorts the numbers so that $\text{num1} \leq \text{num2} \leq \text{num3}$.

3.6  
(Computing the perimeter of a triangle) Write a program that reads three edges for a triangle and computes the perimeter if the input is valid. Otherwise, display that the input is invalid. The input is valid if the sum of any two edges is greater than the third edge (also see Exercise 3.1).

3.7  
(Computing taxes) Listing 3.4 gives the source code to compute taxes for single filers. Complete Listing 3.4 to give the complete source code.

3.8  
(Finding the number of days in a month) Write a program that prompts the user to enter the month and year, and displays the number of days in the month. For example, if the user entered month 2 and year 2000, the program should display that February 2000 has 29 days. If the user entered month 3 and year 2005, the program should display that March 2005 has 31 days.

3.9  
(Checking a number) Write a program that prompts the user to enter an integer and checks whether the number is divisible by both 5 and 6, or neither of them, or just one of them. Here are some sample output for input 10, 30, and 23.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10 is divisible by 5 or 6, but not both</td>
</tr>
<tr>
<td>30</td>
<td>30 is divisible by both 5 and 6</td>
</tr>
<tr>
<td>23</td>
<td>23 is not divisible by either 5 or 6</td>
</tr>
</tbody>
</table>

3.10*  
(An addition tutor) Listing 3.5, SubtractionTutor.cpp, randomly generates a subtraction question. Revise the program to randomly generate an addition question with two integers less than 100.

3.11*  
(Addition for three numbers) Listing 3.5, SubtractionTutor.cpp, randomly generates a subtraction question. Revise the program to randomly generate an addition question with three integers less than 100.