Chapter
4

**Loops**

Objectives

- To use *while*, *do-while*, and *for* loop statements to control the repetition of statements (§§4.2-4.4).
- To understand the flow of control in loop statements (§§4.2-4.4).
- To use Boolean expressions to control loop statements (§§4.2-4.4).
- To write nested loops ($4.5$).
- To know the similarities and differences of three types of loops ($4.6$).
- (Optional) To implement program control with *break* and *continue* ($4.7$).
4.1 Introduction

*Side Remark: why loop?>

Suppose that you need to print a string (e.g., "Welcome to C++!") a hundred times. It would be tedious to have to write the following statement a hundred times.

```cpp
cout << "Welcome to C++!\n";
```

C++ provides a powerful control structure called a loop, which controls how many times an operation or a sequence of operations is performed in succession. Using a loop statement, you simply tell the computer to print a string a hundred times without having to code the print statement a hundred times.

Loops are structures that control repeated executions of a block of statements. The concept of looping is fundamental to programming. C++ provides three types of loop statements: while loops, do-while loops, and for loops.

4.2 The while Loop

The syntax for the while loop is as follows:

```cpp
while (loop-continuation-condition)
{
    // Loop body
    Statement(s);
}
```

*Side Remark: loop body*

*Side Remark: iteration*

The while loop flow chart is shown in Figure 4.1(a). The part of the loop that contains the statements to be repeated is called the loop body. A one-time execution of a loop body is referred to as an iteration of the loop. Each loop contains a loop-continuation-condition, a Boolean expression that controls the execution of the body. It is always evaluated before the loop body is executed. If its evaluation is true, the loop body is executed; if its evaluation is false, the entire loop terminates and the program control turns to the statement that follows the while loop. For example, the following while loop prints Welcome to C++! a hundred times.

```cpp
int count = 0;
while (count < 100)
{
    cout << "Welcome to C++!\n";
    count++;
}
```

The flow chart of the preceding statement is shown in Figure 4.1(b). The variable count is initially 0. The loop checks whether (count < 100) is true. If so, it executes the loop body to print the message Welcome to C++! and increments count by 1. It repeatedly executes the loop body until (count < 100) becomes false. When (count < 100) is false (i.e., when count reaches 100), the loop terminates and the next statement after the loop statement is executed.

***Same as Fig 3.6 in intro5e p91***
Figure 4.1
The \texttt{while} loop repeatedly executes the statements in the loop body when the \texttt{loop-continuation-condition} evaluates to \texttt{true}.

\textbf{NOTE}

The \texttt{loop-continuation-condition} must always appear inside the parentheses. The braces enclosing the loop body can be omitted only if the loop body contains one or no statement.

\textbf{CAUTION}

\textit{subtitle: infinite loop}

Make sure that the \texttt{loop-continuation-condition} eventually becomes \texttt{false} so that the program will terminate. A common programming error involves infinite loops. That is, the program cannot terminate because of a mistake in the \texttt{loop-continuation-condition}. For instance, if you forgot to increase \texttt{count} (\texttt{count++}) in the code, the program would not stop. To terminate the program, press CTRL+C.

\textbf{TIP}

\textit{side remark: debugging in IDE}

If you use an IDE such as C++Builder and Visual C++.NET, please refer to \textit{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 loop statement.

4.2.1 Example: An Advanced Math Learning Tool

The Math subtraction tutor program in Listing 3.5, SubtractionTutor.cpp, generates just one question for each run. You can use a loop to generate questions repeatedly. Listing 4.1 gives a program that generates ten questions and reports the number of the
correct answers after a student answers all ten questions. The program also displays the time spent on the test and lists all the questions, as shown in sample output.

Listing 4.1 SubtractionTutorLoop.cpp (Repeating Subtractions)

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

***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, A.***

<Side Remark line 8: correct count>
<Side Remark line 9: total count>
<Side Remark line 10: get start time>
<Side Remark line 12: loop>
<Side Remark line 28: display a question>
<Side Remark line 33: grade an answer>
<Side Remark line 35: increase correct count>
<Side Remark line 42: increase control variable>
<Side Remark line 45: get end time>
<Side Remark line 46: test time>
<Side Remark line 48: display result>

#include <iostream>
#include <ctime> // for time function
#include <cmath> // for the srand and rand functions
using namespace std;

int main()
{
    int correctCount = 0; // Count the number of correct answers
    int count = 0; // Count the number of questions
    long startTime = time(0);

    while (count < 10)
    {
        // 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) {
```
The program uses the control variable `count` to control the execution of the loop. `count` is initially 0 (line 9) and is increased by 1 in each iteration (line 42). A subtraction question is displayed and processed
in each iteration. The program obtains the time before the test starts in line 10 and the time after the test ends in line 45, and computes the test time in line 46.

4.2.2 Controlling a Loop with User Confirmation

**<Side Remark: confirmation>**
The preceding example executes the loop ten times. Suppose you want the user to decide whether to take another question, you can let the user control the loop with a user confirmation. The template of the program can be coded as follows:

```cpp
char continueLoop = 'Y';
while (continueLoop == 'Y')
{
    // Execute body once

    // Prompt the user for confirmation
    cout << "Enter Y to continue and N to quit: ";
    cin >> continueLoop;
}
```

You can rewrite Listing 4.1 with user confirmation to let the user decide whether to continue the next question.

4.2.3 Controlling a Loop with a Sentinel Value

**<Side Remark: sentinel value>**

Another common technique for controlling a loop is to designate a special value when reading and processing a set of values. This special input value, known as a sentinel value, signifies the end of the loop.

Listing 4.2 writes a program that reads and calculates the sum of an unspecified number of integers. The input 0 signifies the end of the input. Do you need to declare a new variable for each input value? No. Just use one variable named data (line 9) to store the input value and use a variable named sum (line 12) to store the total. Whenever a value is read, assign it to data and added to sum (line 14) if it is not zero.

**<Side Remark line 8: input data>**

**<Side Remark line 12: loop>**

**<Side Remark line 21: output result>**

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

int main()
{
    // Prompt the user for input
    cout << "Enter an int value (the program exits if the input is 0): ";
    int data;

Listing 4.2 SentinelValue.cpp (Using while Loop)
```
cin >> data;

// Keep reading data until the input is 0
int sum = 0;
while (data != 0)
{
    sum += data;

    // Read the next data
    cout << "Enter an int value (the program exits if the input is 0): ";
    cin >> data;
}

cout << "The sum is " << sum << endl;
return 0;
}

<output>
Enter an int value (the program exits if the input is 0): 2
Enter an int value (the program exits if the input is 0): 3
Enter an int value (the program exits if the input is 0): 4
Enter an int value (the program exits if the input is 0): 0
The sum is 9

<end of output>

If data is not 0, it is added to the sum (line 14) and the next items of input data are read (lines 17–18). If data is 0, the loop body is no longer executed and the while loop terminates. The input value 0 is the sentinel value for this loop. Note that if the first input read is 0, the loop body never executes, and the resulting sum is 0.

CAUTION

Side Remark: numeric error>

Don’t use floating-point values for equality checking in a loop control. Since floating-point values are approximations for some values, using them could result in imprecise counter values and inaccurate results. This example uses int value for data. If a floating-point type value is used for data, (data != 0) may be true even though data is exactly 0. For example,

double data = pow(sqrt(2.0), 2) - 2;
if (data == 0)
    cout << "data is zero";
else
    cout << "data is not zero";

Like pow, sqrt is a method in the cmath header file for computing the square root of a number. The
variable data in the above code should be zero, but it is not, because of rounding-off errors.

***End of Caution

4.3 The do-while Loop

The do-while loop is a variation of the while loop. Its syntax is given below:

<Side Remark line 1: do-while loop>
do
   // Loop body;
   Statement(s);
while (loop-continuation-condition);

Its execution flow chart is shown in Figure 4.2.

![Flow chart of the do-while loop](image)

Figure 4.2

The do-while loop executes the loop body first, and then checks the loop-continuation-condition to determine whether to continue or terminate the loop.

The loop body is executed first. Then the loop-continuation-condition is evaluated. If the evaluation is true, the loop body is executed again; if it is false, the do-while loop terminates. The major difference between a while loop and a do-while loop is the order in which the loop-continuation-condition is evaluated and the loop body executed. The while loop and the do-while loop have equal expressive power. Sometimes one is a more convenient choice than the other. For example, you can rewrite the while loop in Listing 4.2 using a do-while loop, as shown in Listing 4.3:

Listing 4.3 TestDo.cpp (Using do-while Loop)

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

int main()
{
    // Loop body;
    Statement(s);
    while (loop-continuation-condition);
}
```

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

<Side Remark line 10: loop>
<Side Remark line 16: input>
```cpp
// Keep reading data until the input is 0
int sum = 0;
int data = 0;

do {
    sum += data;
    // Read the next data
    cout << "Enter an int value (the program exits if the input is 0): ";
    cin >> data;
} while (data != 0);

cout << "The sum is " << sum << endl;
return 0;
}
```

What would happen if `sum` and `data` are not initialized to 0. Would it cause syntax error? No. It would cause a logic error, because `sum` and `data` could be initialized to any value.

**TIP**

Use the do-while loop if you have statements inside the loop that must be executed at least once, as in the case of the do-while loop in the preceding TestDoWhile program. These statements must appear before the loop as well as inside the loop if you use a while loop.

### 4.4 The for Loop

Often you write a loop in the following common form:

```cpp
i = initialValue; // Initialize loop control variable
while (i < endValue) {
    // Loop body
    i++; // Adjust loop control variable
}
```

A for loop can be used to simplify the above loop:

```cpp
for (i = initialValue; i < endValue; i++) {
    // Loop body
    ...
}
```

In general, the syntax of a for loop is as shown below:

**<Side Remark line 1: for loop>**

```cpp
for (initial-action; loop-continuation-condition; action-after-each-iteration) {
    // Loop body;
    Statement(s);
}
```

The flow chart of the for loop is shown in Figure 4.3(a).

***Same as Fig3.9 intro5e p96***
Figure 4.3
A for loop performs an initial action once, then repeatedly executes the statements in the loop body, and performs an action after an iteration when the loop-continuation-condition evaluates to true.

The for loop statement starts with the keyword for, followed by a pair of parentheses enclosing initial-action, loop-continuation-condition, and action-after-each-iteration, and followed by the loop body enclosed inside braces. Initial-action, loop-continuation-condition, and action-after-each-iteration are separated by semicolons.

<Side Remark: control variable>
A for loop generally uses a variable to control how many times the loop body is executed and when the loop terminates. This variable is referred to as a control variable. The initial-action often initializes a control variable, the action-after-each-iteration usually increments or decrements the control variable, and the loop-continuation-condition tests whether the control variable has reached a termination value. For example, the following for loop prints Welcome to C++! a hundred times:

```cpp
int i;
for (i = 0; i < 100; i++)
{
    cout << "Welcome to C++!\n";
}
```

The flow chart of the statement is shown in Figure 4.3(b). The for loop initializes i to 0, then repeatedly executes the statement to display a message and evaluates i++ while i is less than 100.

The initial-action, i = 0, initializes the control variable, i. The loop-continuation-condition, i < 100, is a Boolean expression. The expression is evaluated at the beginning of each iteration. If this condition is true, execute the loop body. If it is false, the loop terminates and the program control turns to the line following the loop.
The action-after-each-iteration, i++, is a statement that adjusts the control variable. This statement is executed after each iteration. It increments the control variable. Eventually, the value of the control variable should force the loop-continuation-condition to become false. Otherwise the loop is infinite.

```cpp
for (int i = 0; i < 100; i++)
  cout << "Welcome to C++!\n";
```

If there is only one statement in the loop body, as in this example, the braces can be omitted.

**TIP**

The control variable must always be declared inside the control structure of the loop or before the loop. If the loop control variable is used only in the loop, and not elsewhere, it is good programming practice to declare it in the initial-action of the for loop. If the variable is declared inside the loop control structure, it cannot be referenced outside the loop. For example, you cannot reference i outside the for loop in the preceding code, because it is declared inside the for loop.

**NOTE**

The initial-action in a for loop can be a list of zero or more comma-separated variable declaration statements or assignment expressions. For example,

```cpp
for (int i = 0, j = 0; (i + j < 10); i++, j++)
  // Do something
```

The action-after-each-iteration in a for loop can be a list of zero or more comma-separated statements. For example,

```cpp
for (int i = 1; i < 100; cout << i, i++);
```

This example is correct, but it is not a good example, because it makes the code difficult to read. Normally, you declare and initialize a control variable as initial action, and increment or decrement the control variable as an action after each iteration.

***End of NOTE***
NOTE
If the loop-continuation-condition in a for loop is omitted, it is implicitly true. Thus the statement given below in (a), which is an infinite loop, is correct. Nevertheless, it is better to use the equivalent loop in (b) to avoid confusion:

***Same as intro5e p97

<table>
<thead>
<tr>
<th>for { ; ; }</th>
<th>Equivalent</th>
<th>while (true)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ // Do something }</td>
<td>This is better</td>
<td>{ // Do something }</td>
</tr>
</tbody>
</table>

***End of NOTE

4.5 Which Loop to Use?
<Side Remark: pre-test loop>
<Side Remark: post-test loop>
The while loop and for loop are called pre-test loops because the continuation condition is checked before the loop body is executed. The do-while loop is called post-test loop because the condition is checked after the loop body is executed. The three forms of loop statements, while, do-while, and for, are expressively equivalent; that is, you can write a loop in any of these three forms. For example, a while loop in (a) in the following figure can always be converted into the for loop in (b):

<table>
<thead>
<tr>
<th>while (loop-continuation-condition)</th>
<th>Equivalent</th>
<th>for ( ; loop-continuation-condition; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ // Loop body }</td>
<td></td>
<td>{ // Loop body }</td>
</tr>
</tbody>
</table>

A for loop in (a) in the next figure can generally be converted into the while loop in (b) except in certain special cases (see Review Question 4.12 for such a case):

<table>
<thead>
<tr>
<th>for (initial-action; loop-continuation-condition; action-after-each-iteration)</th>
<th>Equivalent</th>
<th>initial-action; while (loop-continuation-condition)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ // Loop body; }</td>
<td></td>
<td>{ // Loop body; action-after-each-iteration; }</td>
</tr>
</tbody>
</table>

Use the loop statement that is most intuitive and comfortable for you. In general, a for loop may be used if the number of repetitions is known, as, for example, when you need to print a message a hundred times. A while loop may be used if the number of repetitions is not known, as in the case of reading the numbers until the input is 0. A do-while loop can be used to replace a while loop if the loop body has to be executed before the continuation condition is tested.

CAUTION
Adding a semicolon at the end of the for clause before the loop body is a common mistake, as
shown below in (a). In (a), the semicolon signifies the end of the loop prematurely. The loop body is actually empty, as shown in (b). (a) and (b) are equivalent.

Similarly, the loop in (c) is also wrong. (c) is equivalent to (d).

In the case of the do-while loop, the semicolon is needed to end the loop.

4.6 Nested Loops
Nested loops consist of an outer loop and one or more inner loops. Each time the outer loop is repeated, the inner loops are reentered, and started anew.

Listing 4.4 presents a program that uses nested for loops to print a multiplication table.

Listing 4.4 TestMultiplicationTable.cpp (Using Nested for Loop)

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

int main()
{
    cout << "       Multiplication Table\n";
    cout << "--------------------------------\n";
```

***End of CAUTION***
```cpp
  // Display the number title
  cout << "  | ";
  for (int j = 1; j <= 9; j++)
    cout << setw(3) << j;
  cout << "\n";

  // Print table body
  for (int i = 1; i <= 9; i++) {
    cout << i << " | ";
    for (int j = 1; j <= 9; j++) {
      // Display the product and align properly
      cout << setw(3) << i * j;
    }
    cout << "\n";
  }
  return 0;
```

The program displays a title (line 7) on the first line and dashes (--) (line 8) on the second line. The first for loop (lines 12-13) displays the numbers 1 through 9 on the third line.

The next loop (lines 18-25) is a nested for loop with the control variable i in the outer loop and j in the inner loop. For each i, the product i * j is displayed on a line in the inner loop, with j being 1, 2, 3, . . . , 9. The `setw(3)` manipulator specifies the width for each number to be displayed.

### 4.7 Case Studies
Control statements are fundamental in programming. The ability to write control statements is essential in learning programming. If you can write programs using loops, you know how to program! For this reason, this section presents three additional examples of how to solve problems using loops.

#### 4.7.1 Example: Finding the Greatest Common Divisor
This section presents a program that prompts the user to enter two positive integers and finds their greatest common divisor.
<Side Remark: GCD>
The greatest common divisor of two integers 4 and 2 is 2. the greatest common divisor of two integers 16 and 24 is 8. How do you find the greatest common divisor? Let the two input integers be n1 and n2. You know that number 1 is a common divisor, but it may not be the greatest common divisor. So you can check whether k (for k = 2, 3, 4, and so on) is a common divisor for n1 and n2, until k is greater than n1 or n2. Store the common divisor in a variable named gcd. Initially, gcd is 1. Whenever a new common divisor is found, it becomes the new gcd. When you have checked all the possible common divisors from 2 up to n1 or n2, the value in variable gcd is the greatest common divisor. The idea can be translated into the following loop:

```cpp
int gcd = 1;
int k = 1;
while (k <= n1 && k <= n2)
{
    if (n1 % k == 0 && n2 % k == 0)
        gcd = k;
    k++;
}
// After the loop, gcd is the greatest common divisor for n1 and n2
```

The complete program is given in Listing 4.5.

Listing 4.5 GreatestCommonDivisor.cpp (Finding GCD)

***PD: Please add line numbers in the following code***

<Side Remark line 9: input>
<Side Remark line 13: input>
<Side Remark line 15: gcd>
<Side Remark line 24: output>

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

int main()
{
    // Prompt the user to enter two integers
    cout << "Enter first integer: ";
    int n1;
    cin >> n1;

    cout << "Enter second integer: ";
    int n2;
    cin >> n2;

    int gcd = 1;
    int k = 1;
    while (k <= n1 && k <= n2)
    {
        if (n1 % k == 0 && n2 % k == 0)
            gcd = k;
        k++;
    }

    cout << "The greatest common divisor for " << n1 << " and 
```
<< n2 << " is " << gcd;

return 0;
}

<Output>

Enter first integer: 125
Enter second integer: 2525
The greatest common divisor for 125 and 2525 is 25

<End output>

<Side Remark: think before you type>
How did you write this program? Did you immediately begin to write the code? No. It is important to think before you type. Thinking enables you to generate a logical solution for the problem without concern about how to write the code. Once you have a logical solution, type the code to translate the solution into a program. The translation is not unique. For example, you could use a for loop to rewrite the code as follows:

```c
for (int k = 1; k <= n1 && k <= n2; k++)
    if (n1 % k == 0 && n2 % k == 0)
        gcd = k;
```

<Side Remark: multiple solutions>
A problem often has multiple solutions. The GCD problem can be solved in many ways. Exercise 4.15 suggests another solution. A more efficient solution is to use the classic Euclidean algorithm. See http://www.cut-the-knot.org/blue/Euclid.shtml for more information.

<Side Remark: erroneous solutions>
You might think that a divisor for a number n1 cannot be greater than n1 / 2. So you would attempt to improve the program using the following loop:

```c
for (int k = 1; k <= n1 / 2 && k <= n2 / 2; k++)
    if (n1 % k == 0 && n2 % k == 0)
        gcd = k;
```

This revision is wrong. Can you find the reason? See Review Question 4.9 for the answer.

4.7.2 Example: Finding the Sales Amount
You have just started a sales job in a department store. Your pay consists of a base salary and a commission. The base salary is $5,000. The scheme shown below is used to determine the commission rate.

<table>
<thead>
<tr>
<th>Sales Amount</th>
<th>Commission Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0.01–$5,000</td>
<td>8 percent</td>
</tr>
<tr>
<td>$5,000.01–$10,000</td>
<td>10 percent</td>
</tr>
<tr>
<td>$10,000.01 and above</td>
<td>12 percent</td>
</tr>
</tbody>
</table>
Your goal is to earn $30,000 a year. This section writes a program that finds out the minimum amount of sales you have to generate in order to make $30,000.

Since your base salary is $5,000, you have to make $25,000 in commissions to earn $30,000 a year. What is the sales amount for a $25,000 commission? If you know the sales amount, the commission can be computed as follows:

\[
\text{if } (\text{salesAmount} \geq 10000.01) \\
\text{ commission } = 5000 \times 0.08 + 5000 \times 0.1 + (\text{salesAmount} - 10000) \times 0.12; \\
\text{else if } (\text{salesAmount} \geq 5000.01) \\
\text{ commission } = 5000 \times 0.08 + (\text{salesAmount} - 5000) \times 0.10; \\
\text{else} \\
\text{ commission } = \text{salesAmount} \times 0.08; \\
\]

This suggests that you can try to find the salesAmount to match a given commission through incremental approximation. For salesAmount of $0.01 (1 cent), find commission. If commission is less than $25,000, increment salesAmount by 0.01 and find commission again. If commission is still less than $25,000, repeat the process until the commission is greater than or equal to $25,000. This is a tedious job for humans, but it is exactly what a computer is good for. You can write a loop and let a computer execute it painlessly. The idea can be translated into the following loop:

\[
\text{Set COMMISSION_SOUGHT as a constant;}
\text{Set an initial salesAmount;}
\text{do} \\
\text{ Increase salesAmount by 1 cent;}
\text{ Compute the commission from the current salesAmount;}
\text{ while } (\text{commission} < \text{COMMISSION_SOUGHT});
\]

The complete program is given in Listing 4.6.

Listing 4.6 FindSalesAmount.cpp (Finding Sales Amount)

***PD: Please add line numbers in the following code***

<Side Remark line 8: constants>
<Side Remark line 13: loop>
<Side Remark line 30: fixed precision>

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

int main()
{
    // The commission sought
    const double COMMISSION_SOUGHT = 25000;
    const double INITIAL_SALES_AMOUNT = 0.01;
    double commission = 0;
    double salesAmount = INITIAL_SALES_AMOUNT;

    do
    {
        // Increase salesAmount by 1 cent
        salesAmount += 0.01;
    }
    while (commission < COMMISSION_SOUGHT);
```

140
// Compute the commission from the current salesAmount:
if (salesAmount >= 10000.01)
    commission =
        5000 * 0.08 + 5000 * 0.1 + (salesAmount - 10000) * 0.12;
else if (salesAmount >= 5000.01)
    commission = 5000 * 0.08 + (salesAmount - 5000) * 0.10;
else
    commission = salesAmount * 0.08;
}
while (commission < COMMISSION_SOUGHT);

// Display the sales amount
cout << "The sales amount $" << fixed << setprecision(2) << 
    (salesAmount * 100) / 100.0 << 
    "\nis needed to make a commission of $" << COMMISSION_SOUGHT;

return 0;
}

<Output>
The sales amount $210833.34
is needed to make a commission of $25000.00

<End output>

The do-while loop (lines 13–27) is used to repeatedly compute commission for an incremental salesAmount. The loop terminates when commission is greater than or equal to a constant COMMISSION_SOUGHT.

In Exercise 4.17, you will rewrite this program to let the user enter COMMISSION_SOUGHT dynamically.

You can improve the performance of this program by estimating a higher INITIAL_SALES_AMOUNT (e.g., 25000).

What is wrong if saleAmount is incremented after the commission is computed as follows?

do
  // Compute the commission from the current salesAmount:
  if (salesAmount >= 10000.01)
      commission =
          5000 * 0.08 + 5000 * 0.1 + (salesAmount - 10000) * 0.12;
  else if (salesAmount >= 5000.01)
      commission = 5000 * 0.08 + (salesAmount - 5000) * 0.10;
  else
      commission = salesAmount * 0.08;
  // Increase salesAmount by 1 cent
  salesAmount += 0.01;
} while (commission < COMMISSION_SOUGHT);

<Side Remark: off-by-one error>
The change is erroneous because saleAmount is 1 cent more than is needed for the commission when the loop ends. This is a common error in loops, known as the off-by-one error.
TIP

**Side Remark: constants**

This example uses constants COMMISSION_SOUGHT and INITIAL_SALES_AMOUNT. Using constants makes programs easy to read and maintain.

4.8.3 Example: Displaying a Pyramid of Numbers

This section presents a program that prompts the user to enter an integer from 1 to 15 and displays a pyramid. If the input integer is 12, for example, the output is shown as follows:

**Output**

```
Enter the number of lines: 12

     1
    2  1  2
   3  2  1  2  3
  4  3  2  1  2  3  4
 5  4  3  2  1  2  3  4  5
6  5  4  3  2  1  2  3  4  5  6
7  6  5  4  3  2  1  2  3  4  5  6  7
8  7  6  5  4  3  2  1  2  3  4  5  6  7  8
9  8  7  6  5  4  3  2  1  2  3  4  5  6  7  8  9
10 9  8  7  6  5  4  3  2  1  2  3  4  5  6  7  8  9 10
11 10 9  8  7  6  5  4  3  2  1  2  3  4  5  6  7  8  9 10 11
12 11 10 9  8  7  6  5  4  3  2  1  2  3  4  5  6  7  8  9 10 11 12
```

**End output**

Your program receives the input for an integer (numberOfLines) that represents the total number of lines. It displays all the lines one by one. Each line has three parts. The first part comprises the spaces before the numbers; the second part, the leading numbers, such as 3 2 1 in line 3; and the last part, the ending numbers, such as 2 3 in line 3.

Each number occupies three spaces. Display an empty space before a double-digit number, and display two empty spaces before a single-digit number.

You can use an outer loop to control the lines. At the n<sup>th</sup> row, there are (numberOfLines - n) * 3 leading spaces, the leading numbers are n, n-1, . . ., 1, and the ending numbers are 2, . . ., n. You can use three separate inner loops to print each part.

Here is the algorithm for the problem:

```
Input numberOfLines;

for (int row = 1; row <= numberOfLines; row++) {
    Print (numberOfLines - row) * 3 leading spaces;
    Print leading numbers row, row - 1, ..., 1;
    Print ending numbers 2, 3, ..., row - 1, row;
    Start a new line;
}
```
The complete program is given in Listing 4.7.

Listing 4.7 PrintPyramid.cpp (Printing Pyramid)

```cpp
***PD: Please add line numbers in the following code***
<Side Remark line 9: input line number>
<Side Remark line 11: check input>
<Side Remark line 18: print lines>
<Side Remark line 21: print spaces>
<Side Remark line 25: print leading numbers>
<Side Remark line 29: print ending numbers>
<Side Remark line 33: a new line>

#include <iostream>
using namespace std;

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

    if (numberOfLines < 1 || numberOfLines > 15)
    {
        cout << "You must enter a number from 1 to 15";
        return 0;
    }

    // Print lines
    for (int row = 1; row <= numberOfLines; row++)
    {
        // Print NUMBER_OF_LINES - row) leading spaces
        for (int column = 1; column <= numberOfLines - row; column++)
            cout << "   ";

        // Print leading numbers row, row - 1, ..., 1
        for (int num = row; num >= 1; num--)
            cout << ((num >= 10) ? " " : "  ") << num;

        // Print ending numbers 2, 3, ..., row - 1, row
        for (int num = 2; num <= row; num++)
            cout << ((num >= 10) ? " " : "  ") << num;

        // Start a new line
        cout << endl;
    }

    return 0;
}
```

The conditional expression (num >= 10) ? " " : "  " in lines 26 and 30 displays a single empty space before the number if the number is
greater than or equal to 10, and otherwise displays with two empty spaces before the number.

Printing patterns like this one and the ones in Exercises 4.18 and 4.19 are good exercise for practicing loop control statements. The key is to understand the pattern and to describe it using loop control variables.

4.9 (Optional) Keywords **break** and **continue**

Two statements, break and continue, can be used in loop statements to provide the loop with additional control.

<Side Remark: break>

[BL] **break** immediately ends the innermost loop that contains it. It is generally used with an if statement.

<Side Remark: continue>

[BL] **continue** only ends the current iteration. Program control goes to the end of the loop body. This keyword is generally used with an if statement.

You have already used the keyword break in a switch statement. You can also use break and continue in a loop. Listings 4.8 and 4.9 present two programs to demonstrate the effect of the break and continue keywords in a loop.

The program in Listing 4.8 adds the integers from 1 to 20 in this order to sum until sum is greater than or equal to 100. Without the if statement (line 10), this program calculates the sum of the numbers from 1 to 20.

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

int main()
{
    int sum = 0;
    int number = 0;

    while (number < 20)
    {
        number++;
        sum += number;
        if (sum >= 100) break;
    }  

    cout << "The number is " << number << endl;
    cout << "The sum is " << sum << endl;

    return 0;
}
```

**Output**

144
The number is 14
The sum is 105

The program in Listing 4.9 adds all the integers from 1 to 20 except 10 and 11 to sum. With the if statement in the program (line 11), the continue statement is executed when number becomes 10 or 11. The continue statement ends the current iteration so that the rest of the statement in the loop body is not executed; therefore, number is not added to sum when it is 10 or 11.

Listing 4.9 TestContinue.cpp (Skipping Iteration)

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

int main()
{
    int sum = 0;
    int number = 0;

    while (number < 20) {
        number++;
        if (number == 10 || number == 11) continue;
        sum += number;
    }

    cout << "The sum is " << sum;

    return 0;
}
```

The sum is 189

NOTE
The continue statement is always inside a loop. In the while and do-while loops, the loop-continuation-condition is evaluated immediately after the continue statement. In the for loop, the action-after-each-iteration is performed, then the loop-continuation-condition is evaluated, immediately after the continue statement.

TIP
You can always write a program without using break or continue in a loop. See Review Question 4.13. In general, it is appropriate to use break and continue if their use simplifies coding and makes programs easier to read.
4.10 Example: Displaying Prime Numbers

This section presents a program that displays the first fifty prime numbers in five lines, each of which contains ten numbers. An integer greater than 1 is prime if its only positive divisor is 1 or itself. For example, 2, 3, 5, and 7 are prime numbers, but 4, 6, 8, and 9 are not.

The problem can be broken into the following tasks:
- Determine whether a given number is prime.
- For number = 2, 3, 4, 5, 6, ..., test whether the number is prime.
- Count the prime numbers.
- Print each prime number, and print ten numbers per line.

Obviously, you need to write a loop and repeatedly test whether a new number is prime. If the number is prime, increase the count by 1. The count is 0 initially. When it reaches 50, the loop terminates.

Here is the algorithm for the problem:

Set the number of prime numbers to be printed as a constant NUMBER_OF_PRIMES;
Use count to track the number of prime numbers and set an initial count to 0;
Set an initial number to 2;

while (count < NUMBER_OF_PRIMES)
{
    Test if number is prime;
    if number is prime {
        Print the prime number and increase the count;
    }
    Increment number by 1;
}

To test whether a number is prime, check whether the number is divisible by 2, 3, 4, up to number/2. If a divisor is found, the number is not a prime. The algorithm can be described as follows:

Use a boolean variable isPrime to denote whether the number is prime; Set isPrime to true initially;

for (int divisor = 2; divisor <= number / 2; divisor++)
{
    if (number % divisor == 0) {
        Set isPrime to false
        Exit the loop;
    }
}

The complete program is given in Listing 4.10.
#include <iostream>
using namespace std;

int main()
{
    const int NUMBER_OF_PRIMES = 50; // Number of primes to display
    const int NUMBER_OF_PRIMES_PER_LINE = 10; // Display 10 per line
    int count = 0; // Count the number of prime numbers
    int number = 2; // A number to be tested for primeness

    cout << "The first 50 prime numbers are \n";

    // Repeatedly find prime numbers
    while (count < NUMBER_OF_PRIMES)
    {
        // Assume the number is prime
        bool isPrime = true; // Is the current number prime?

        // Test if number is prime
        for (int divisor = 2; divisor <= number / 2; divisor++)
        {
            if (number % divisor == 0)
                // If true, the number is not prime
                isPrime = false; // Set isPrime to false
                break; // Exit the for loop
        }

        // Print the prime number and increase the count
        if (isPrime)
        {
            count++; // Increase the count
            if (count % NUMBER_OF_PRIMES_PER_LINE == 0)
                // Print the number and advance to the new line
                cout << number << endl;
            else
                cout << number << " ";
        }

        // Check if the next number is prime
        number++;
    }

    return 0;
}
The first 50 prime numbers are
2 3 5 7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
73 79 83 89 97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229

<Side Remark line 13: subproblem>
This is a complex example for novice programmers. The key to developing
a programmatic solution to this problem, and to many other problems, is
to break it into subproblems and develop solutions for each of them in
turn. Do not attempt to develop a complete solution in the first trial.
Instead, begin by writing the code to determine whether a given number
is prime, then expand the program to test whether other numbers are
prime in a loop.

To determine whether a number is prime, check whether it is divisible
by a number between 2 and number/2 inclusive. If so, it is not a prime
number; otherwise, it is a prime number. For a prime number, display
it. If the count is divisible by 10, advance to a new line. The program
ends when the count reaches 50.

NOTE
The program uses the break statement in line 26 to
exit the for loop as soon as the number is found to
be a nonprime. You can rewrite the loop (lines 20-28)
without using the break statement, as follows:

```java
for (int divisor = 2; divisor <= number / 2 && isPrime;
     divisor++)
{
    // If true, the number is not prime
    if (number % divisor == 0)
    {
        // Set isPrime to false, if the number is not prime
        isPrime = false;
    }
}
```

However, using the break statement makes the
program simpler and easier to read in this case.

***End of NOTE

Key Terms

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

- break statement 89, 101
- continue statement 101
- infinite loop 92
- iteration 91
- loop 91
- loop-continuation-condition 91
- loop body 91
- nested loop 99
Chapter Summary

- Program control specifies the order in which statements are executed in a program. There are three types of control statements: sequence, selection, and repetition. The preceding chapters introduced sequence and selection statements. This chapter introduced the repetition statements.

- There are three types of repetition statements: the while loop, the do-while loop, and the for loop. In designing loops, you need to consider both the loop control structure and the loop body.

- The while loop checks the loop-continuation-condition first. If the condition is true, the loop body is executed; if it is false, the loop terminates. The do-while loop is similar to the while loop, except that the do-while loop executes the loop body first and then checks the loop-continuation-condition to decide whether to continue or to terminate.

- Since the while loop and the do-while loop contain the loop-continuation-condition, which is dependent on the loop body, the number of repetitions is determined by the loop body. The while loop and the do-while loop are often used when the number of repetitions is unspecified.

- The for loop is generally used to execute a loop body a predictable number of times; this number is not determined by the loop body. The loop control has three parts. The first part is an initial action that often initializes a control variable. The second part, the loop-continuation-condition, determines whether the loop body is to be executed. The third part is executed after each iteration and is often used to adjust the control variable. Usually, the loop control variables are initialized and changed in the control structure.

- Two keywords, break and continue, can be used in a loop. The break keyword immediately ends the innermost loop, which contains the break. The continue keyword only ends the current iteration.

Review Questions
Sections 4.2-4.6
4.1
How many times is the following loop body repeated? What is the printout of the loop?
4.2
What are the differences between a while loop and a do-while loop?

4.3
Do the following two loops result in the same value in sum?

4.4
What are the three parts of a for loop control? Write a for loop that prints the numbers from 1 to 100.

4.5
What does the following statement do?

```
for ( ; ; )
{
    do something;
}
```

4.6
If a variable is declared in the for loop control, can it be used after the loop exits?

4.7
Can you convert a for loop to a while loop? List the advantages of using for loops.

4.8
Convert the following for loop statement to a while loop and to a do-while loop:

```
long sum = 0;
for (int i = 0; i <= 1000; i++)
    sum = sum + i;
```

4.9
Will the program work if n1 and n2 are replaced by n1 / 2 and n2 / 2 in line 15 in Listing 4.5?

**Section 4.9 Keywords break and continue**

4.10
What is the keyword break for? What is the keyword continue for? Will the following program terminate? If so, give the output.
int balance = 1000;
while (true)
{
    if (balance < 9)
        break;
    balance = balance - 9;
}
cout << "Balance is " << balance << endl;

int balance = 1000;
while (true)
{
    if (balance < 9)
        continue;
    balance = balance - 9;
}
cout << "Balance is " << balance << endl;

(a)                                (b)

4.11
Can you always convert a while loop into a for loop?
Convert the following while loop into a for loop.

```cpp
int i = 1;
int sum = 0;
while (sum < 10000)
{
    sum = sum + i;
    i++;
}
```

4.12
The for loop on the left is converted into the while loop on the right. What is wrong? Correct it.

```cpp
for (int i = 0; i < 4; i++)
{
    if (i % 3 == 0) continue;
    sum += i;
}
```

4.13
Rewrite the programs TestBreak and TestContinue in Listings 4.9 and 4.10 without using break and continue.

Comprehensive

4.14
Identify and fix the errors in the following code:

```cpp
***PD: Please add line numbers in the following code***
for (int i = 0; i < 10; i++)
{
    sum += i;
    if (i < j);
    cout << i;
    else
    cout << j;
    while (j < 10);
    { j++; };
    do
    { j++;
    } while (j < 10)
}
```
4.16
Show the output of the following programs:

(a) for (int i = 1; i < 5; i++)
    { int j = 0;
        while (j < i)
        { cout << j << " ", j++;
        }
    }

(b) int i = 0;
    while (i < 5)
    { for (int j = i; j > 1; j--)
        { cout << j << " ";
            cout << "****" << endl;
            i++;
        }
    }

(c) int i = 5;
    while (i >= 1)
    { int num = 1;
        for (int j = i; j <= 1; j++)
        { cout << num << "xxx",
            num *= 2;
        }
        cout << endl;
        i--;
    }

(d) int i = 1;
    do
    { int num = 1;
        for (int j = 1; j <= i; j++)
        { cout << num << "G",
            num += 2;
        }
        cout << endl;
        i++;
    } while (i <= 5);

4.17
Reformat the following programs according to the programming style and documentation guidelines proposed in §2.14.

(a) #include <iostream>
    using namespace std;
    int main()
    { int i = 0;
        if (i>0)
            i++;
        else
            i--;
        char grade;
        if (i >= 90)
            grade = 'A';
        else
            if (i >= 80)
                grade = 'B';
    }

(b) #include <iostream>
    using namespace std;
    int main()
    { for (int i = 0; i<10; i++)
        { if (i>0)
            i++;
        else
            i--;
    }

Programming Exercises

Pedagogical NOTE
<side remark: explore solutions>
A problem often can be solved in many different ways. Students are encouraged to explore various solutions.
Sections 4.2-4.7

4.1*
(Repeating additions) Listing 4.1, SubtractionTutorLoop.cpp, generates ten random subtraction questions. Revise the program to generate ten random addition questions for two integers between 1 and 15. Display the correct count and test time.

4.2*
(Counting positive and negative numbers and computing the average of numbers) Write a program that reads an unspecified number of integers, determines how many positive and negative values have been read, and computes the total and average of the input values, not counting zeros. Your program ends with the input 0. Display the average as a floating-point number. (For example, if you entered 1, 2, and 0, the average should be 1.5.)

4.3
(Conversion from kilograms to pounds) Write a program that displays the following table (note that 1 kilogram is 2.2 pounds):

<table>
<thead>
<tr>
<th>Kilograms</th>
<th>Pounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.2</td>
</tr>
<tr>
<td>3</td>
<td>6.6</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>197</td>
<td>433.4</td>
</tr>
<tr>
<td>199</td>
<td>437.8</td>
</tr>
</tbody>
</table>

4.4
(Conversion from miles to kilometers) Write a program that displays the following table (note that 1 mile is 1.609 kilometers):

<table>
<thead>
<tr>
<th>Miles</th>
<th>Kilometers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.609</td>
</tr>
<tr>
<td>2</td>
<td>3.218</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>14.481</td>
</tr>
<tr>
<td>10</td>
<td>16.09</td>
</tr>
</tbody>
</table>

4.5
(Conversion from kilograms to pounds) Write a program that displays the following two tables side-by-side (note that 1 kilogram is 2.2 pounds):

<table>
<thead>
<tr>
<th>Kilograms</th>
<th>Pounds</th>
<th>Kilograms</th>
<th>Pounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.2</td>
<td>9.08</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>6.6</td>
<td>11.36</td>
<td>510</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>197</td>
<td>433.4</td>
<td>231.82</td>
<td></td>
</tr>
<tr>
<td>199</td>
<td>437.8</td>
<td>234.09</td>
<td></td>
</tr>
</tbody>
</table>

153
4.6 
(Conversion from miles to kilometers) Write a program that displays the following two tables side-by-side (note that 1 mile is 1.609 kilometers):

<table>
<thead>
<tr>
<th>Miles</th>
<th>Kilometers</th>
<th>Kilometers</th>
<th>Miles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.609</td>
<td>20</td>
<td>12.430</td>
</tr>
<tr>
<td>2</td>
<td>3.218</td>
<td>25</td>
<td>15.538</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>14.481</td>
<td>60</td>
<td>37.290</td>
</tr>
<tr>
<td>10</td>
<td>16.09</td>
<td>65</td>
<td>40.398</td>
</tr>
</tbody>
</table>

4.7** 
(Computing future tuition) Suppose that the tuition for a university is $10,000 this year and tuition increases 5% every year. Write a program that uses a loop to compute the tuition in ten years. Write another program that computes the total cost of four year worth of tuition starting ten years from now.

4.8 
(Finding the highest score) Write a program that prompts the user to enter the number of students and each student’s name and score, and finally displays the student with the highest score.

4.9* 
(Finding the two highest scores) Write a program that prompts the user to enter the number of students and each student’s name and score, and finally displays the student with the highest score and the student with the second-highest score.

4.10 
(Finding numbers divisible by 5 and 6) Write a program that displays all the numbers from 100 to 1000, ten per line, that are divisible by 5 and 6.

4.11 
(Finding numbers divisible by 5 or 6, but not both) Write a program that displays all the numbers from 100 to 200, ten per line, that are divisible by 5 or 6, but not both.

4.12 
(Finding the smallest \( n \) such that \( n^2 > 12000 \)) Use a while loop to find the smallest integer \( n \) such that \( n^2 \) is greater than 12,000.

4.13
(Finding the largest \( n \) such that \( n^2 < 12000 \)) Use a while loop to find the largest integer \( n \) such that \( n^2 \) is less than 12,000.

4.14*
(Displaying the ACSII character table) Write a program that prints the characters in the ACSII character table from ‘!’ to ‘~’. Print ten characters per line.

Section 4.8 Case Studies

4.15*
(Computing the greatest common divisor) Another solution for Listing 4.5 to find the greatest common divisor of two integers \( n_1 \) and \( n_2 \) is as follows: First find \( d \) to be the minimum of \( n_1 \) and \( n_2 \), then check whether \( d \), \( d-1 \), \( d-2 \), ..., 2, or 1 is a divisor for both \( n_1 \) and \( n_2 \) in this order. The first such common divisor is the greatest common divisor for \( n_1 \) and \( n_2 \).

4.16**
(Finding the factors of an integer) Write a program that reads an integer and displays all its smallest factors. For example, if the input integer is 120, the output should be as follows: 2, 2, 2, 3, 5.

4.17*
(Finding the sales amount) Rewrite Listing 4.6, FindSalesAmount.cpp, as follows:
- Use a for loop instead of a do-while loop.
- Let the user enter COMMISSION_Sought instead of fixing it as a constant.

4.18*
(Printing four patterns using loops) Use nested loops that print the following patterns in four separate programs: ***Please layout the following numbers exactly like this.***

<table>
<thead>
<tr>
<th>Pattern I</th>
<th>Pattern II</th>
<th>Pattern III</th>
<th>Pattern IV</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6</td>
<td>1 2 3 4 5 6</td>
<td></td>
</tr>
<tr>
<td>1 2</td>
<td>1 2 3 4 5</td>
<td>2 1 1 2 3 4</td>
<td></td>
</tr>
<tr>
<td>1 2 3</td>
<td>1 2 3 4</td>
<td>3 2 1 1 2 3</td>
<td></td>
</tr>
<tr>
<td>1 2 3 4</td>
<td>1 2 3 4 5</td>
<td>4 3 2 1 2 3</td>
<td></td>
</tr>
<tr>
<td>1 2 3 4 5</td>
<td>1 2 5 4 3 2</td>
<td>1 2 1 2 3 4</td>
<td></td>
</tr>
<tr>
<td>1 2 3 4 5 6</td>
<td>1      6 5 4 3 2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

4.19**
(Printing numbers in a pyramid pattern) Write a nested for loop that prints the following output: ***Please layout the following pyramid exactly like this.***

```
1
1 5
1 5 9
1 5 9 13
1 5 9 13 17
1 5 9 13 17 21
```
HINT
Here is the pseudocode solution:
```cpp
for the row from 0 to 7 {
    for the column from 1 to 7-row
        cout << "    ";
    Print left half of the row for numbers 1, 2, 4, up to
    2^P row P P P row P P P using a loop like this:
        cout << "    " << pow(2, column);
    Print the right half of the row for numbers
    2^P row-1 P P P row-2 P P P, ..., 1 using a loop like this:
        for (int column = row - 1; column >= 0; col--)
            cout << "    " << pow(2, column);
    Start a new line
    cout << endl;
}
```
You need to figure out how many spaces to print before the number. This is dependent on the number. If a number is a single digit, print four spaces. If a number has two digits, print three spaces. If a number has three digits, print two spaces.

The Math.pow() method was introduced in Listing 2.9, ComputeLoan.cpp, Can you write this program without using it?

4.20*  
(Printing prime numbers between 2 and 1000) Modify Listing 4.11 to print all the prime numbers between 2 and 1000, inclusively. Display eight prime numbers per line.

Comprehensive
4.21**  
(Comparing loans with various interest rates) Write a program that lets the user enter the loan amount and loan period in number of years and displays the monthly and total payments for each interest rate starting from 5% to 8%, with an increment of 1/8. Suppose you enter the loan amount 10,000 for five years, display a table as follows:

<table>
<thead>
<tr>
<th>Loan Amount: 10000</th>
<th>Number of Years: 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interest Rate</td>
<td>Monthly Payment</td>
</tr>
<tr>
<td>5%</td>
<td>188.71</td>
</tr>
<tr>
<td>5.125%</td>
<td>189.28</td>
</tr>
<tr>
<td>5.25%</td>
<td>189.85</td>
</tr>
<tr>
<td>...</td>
<td>202.16</td>
</tr>
<tr>
<td>7.85%</td>
<td>202.76</td>
</tr>
<tr>
<td>8.0%</td>
<td></td>
</tr>
</tbody>
</table>
4.22**  
(Displaying the loan amortization schedule) The monthly payment for a given loan pays the principal and the interest. The monthly interest is computed by multiplying the monthly interest rate and the balance (the remaining principal). The principal paid for the month is therefore the monthly payment minus the monthly interest. Write a program that lets the user enter the loan amount, number of years, and interest rate, and displays the amortization schedule for the loan. Suppose you enter the loan amount 10,000 for one year with an interest rate of 7%, display a table as follows:

<table>
<thead>
<tr>
<th>Loan Amount: 10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Years: 1</td>
</tr>
<tr>
<td>Annual Interest Rate: 7%</td>
</tr>
</tbody>
</table>

| Monthly Payment: 865.26 |
| Total Payment: 10383.21 |

<table>
<thead>
<tr>
<th>Payment#</th>
<th>Interest</th>
<th>Principal</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>58.33</td>
<td>806.93</td>
<td>9193.07</td>
</tr>
<tr>
<td>2</td>
<td>53.62</td>
<td>811.64</td>
<td>8381.43</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>5.01</td>
<td>860.25</td>
<td>0.01</td>
</tr>
</tbody>
</table>

**NOTE**
The balance after the last payment may not be zero. If so, the last payment should be the normal monthly payment plus the final balance.

**HINT**
Write a loop to print the table. Since monthly payment is the same for each month, it should be computed before the loop. The balance is initially the loan amount. For each iteration in the loop, compute the interest and principal, and update the balance. The loop may look like this:

```c++
for (i = 1; i <= numberOfYears * 12; i++) {
    interest = monthlyInterestRate * balance;
    principal = monthlyPayment - interest;
    balance = balance - principal;
    cout << i << "  " << interest << "  " << principal << "  " << balance << endl;
}
```

4.23*  
(Demonstrating cancellation errors) A cancellation error occurs when you are manipulating a very large number with a very small number. The large number may cancel out the smaller number. For example, the result of 100000000.0 + 0.000000001 is equal to 100000000.0. To avoid cancellation errors and obtain more accurate results, carefully select the order of computation. For example, in computing the following series, you will obtain more accurate results by computing from right to left rather than from left to right:
Write a program that compares the results of the summation of the preceding series, computing from left to right and from right to left with \( n = 50000 \).

4.24*
(Summing a series) Write a program to sum the following series:

\[
\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}
\]

4.25**
(Computing \( \pi \)) You can approximate \( \pi \) by using the following series:

\[
\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots - \frac{1}{2i-1} + \frac{1}{2i+1}\right)
\]

Write a program that displays the \( \pi \) value for \( i = 10000, 20000, \ldots, \) and 100000.

4.26**
(Computing \( e \)) You can approximate \( e \) by using the following series:

\[
e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots + \frac{1}{i!}
\]

Write a program that displays the \( e \) value for \( i = 10000, 20000, \ldots, \) and 100000. (Hint: Since \( i! = i \times (i-1) \times \ldots \times 2 \times 1 \), \( \frac{1}{i!} \) is \( \frac{1}{i(i-1) \ldots 2} \). Initialize \( e \) and \( \text{item} \) to be 1 and keep adding a new \( \text{item} \) to \( e \). The new item is the previous item divided by \( i! \) for \( i = 2, 3, 4, \ldots \).)

4.27**
(Displaying leap years) Write a program that displays all the leap years, ten per line, in the twenty-first century (from 2001 to 2100).

4.28**
(Displaying first days of each month) Write a program that prompts the user to enter the year and first day of the year, and displays the first day of each month in the year on the console. For example, if the user entered year 2005, and 6 for Saturday, January 1, 2005, your program should display the following output:
4.29**

(Displaying calendars) Write a program that prompts the user to enter the year and first day of the year, and displays the calendar table for the year on the console. For example, if the user entered year 2005, and 6 for Saturday, January 1, 2005, your program should display the calendar for each month in the year, as follows:

```
January 2005
----------------------------------
Sun Mon Tue Wed Thu Fri Sat
 1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
...
December 2005
----------------------------------
Sun Mon Tue Wed Thu Fri Sat
 1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```