Other Loop Options

Now that you have mastered the looping constructs, you should learn some loop-related statements. This chapter teaches the concepts of *timing loops*, which enable you to slow down your programs. Slowing program execution can be helpful if you want to display a message for a fixed period of time or write computer games with slower speeds so they are at a practical speed for recreational use.

You can use two additional looping commands, the `break` and `continue` statements, to control the loops. These statements work with `while` loops and `for` loops.

This chapter introduces you to the following:

- Timing loops
- The `break` statement with `for` loops
- The `continue` statement with `for` loops

When you master these concepts, you will be well on your way toward writing powerful programs that process large amounts of data.

Timing Loops

Computers are fast, and at times you would probably like them to be even faster. Sometimes, however, you want to slow down the computer. Often, you have to slow the execution of games because the computer's speed makes the game unplayable. Messages that appear on-screen many times clear too fast for the user to read if you don't delay them.

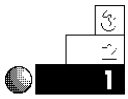
A nested loop is a perfect place for a *timing loop*, which simply cycles through a `for` or `while` loop many times. The larger the end value of the `for` loop, the longer the time in which the loop repeats.

A nested loop is appropriate for displaying error messages to your user. If the user requested a report—but had not entered enough data for your program to print the report—you might print a warning message on-screen for a few seconds, telling users that they cannot request the report yet. After displaying the message for a few seconds, you can clear the message and give the user another chance. (The example program in Appendix F, "The Mailing List Application," uses timing loops to display error messages.)

Timing loops make the computer wait.

There is no way to determine how many iterations a timing loop takes for one second (or minute or hour) of delay because computers run at different speeds. You therefore have to adjust your timing loop's end value to set the delay to your liking.

Examples



1. Timing loops are easy to write—simply put an empty `for` loop inside the program. The following program is a rewritten version of the countdown program (C13CNTD1.CPP) you saw in Chapter 13. Each number in the countdown is delayed so the countdown does not seem to take place instantly. (Adjust the delay value if this program runs too slowly or too quickly on your computer.)

Identify the program and include the input/output header file. You need a counter and a delay, so make `cd` *and* `delay` *integer variables. Start the counter at 10, and start the* `delay` *at 1.*

1. *If the* `delay` *is less than or equal to 30,000, add 1 to its value and repeat step one.*

2. *Print the value of the counter.*
3. *If the counter is greater than or equal to 0, subtract 1 from its value and repeat step one.*

Print a blast-off message.

```
// Filename: C14CNTD1.CPP
// Countdown to the liftoff with a delay.
#include <iostream.h>
main()
{
   int cd, delay;

   for (cd=10; cd>=0; cd--)
   { { for (delay=1; delay <=30000; delay++); }  // Delay
                                                 // program.
     cout << cd << "\n";           // Print countdown value.
   }                                   // End of outer loop
   cout << "Blast off!!! \n";
   return 0;
}
```

2. The following program asks users for their ages. If a user enters an age less than 0, the program beeps (by printing an alarm character, `\a`), then displays an error message for a few seconds by using a nested timing loop. Because an integer does not hold a large enough value (on many computers) for a long timing loop, you must use a nested timing loop. (Depending on the speed of your computer, adjust the numbers in the loop to display the message longer or shorter.)

   The program uses a rarely seen `printf()` conversion character, `\r`, inside the loop. As you might recall from Chapter 7, "Simple Input and Output," `\r` is a carriage-return character. This conversion character moves the cursor to the beginning of the current line, enabling the program to print blanks on that same line. This process overwrites the error message and it appears as though the error disappears from the screen after a brief pause.

```
// Filename: C14TIM.CPP
// Displays an error message for a few seconds.
#include <stdio.h>
main()
{
   int outer, inner, age;

   printf("What is your age? ");
   scanf(" %d", &age);

   while (age <= 0)
     { printf("*** Your age cannot be that small! ***");
       // Timing loop here
       for (outer=1; outer<=30000; outer++)
          { for (inner=1; inner<=500; inner++); }
       // Erase the message
       printf("\r\n\n");
       printf("What is your age? ");
       scanf(" %d", &age);  // Ask again
     }

   printf("\n\nThanks, I did not think you would actually tell");
   printf("me your age!");
   return 0;
}
```

NOTE: Notice the inside loop has a semicolon (;) after the `for` statement—with no loop body. There is no need for a loop body here because the computer is only cycling through the loop to waste some time.

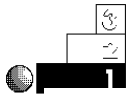
The **break** and **for** Statements

The `for` loop was designed to execute for a specified number of times. On rare occasions, you might want the `for` loop to quit before

the counting variable has reached its final value. As with `while` loops, you use the `break` statement to quit a `for` loop early.

The `break` statement is nested in the body of the `for` loop. Programmers rarely put `break` on a line by itself, and it almost always comes after an `if` test. If the `break` were on a line by itself, the loop would always quit early, defeating the purpose of the `for` loop.

Examples



1. The following program shows what can happen when C++ encounters an *unconditional* `break` statement (one not preceeded by an `if` statement).

Identify the program and include the input/output header files. You need a variable to hold the current number, so make `num` *an integer variable. Print a "Here are the numbers" message.*

1. *Make* `num` *equal to 1. If* `num` *is less than or equal to 20, add one to it each time through the loop.*
2. *Print the value of* `num`*.*
3. *Break out of the loop.*

Print a goodbye message.

```
// Filename: C14BRAK1.CPP
// A for loop defeated by the break statement.
#include <iostream.h>
main()
{
   int num;

   cout << "Here are the numbers from 1 to 20\n";
   for(num=1; num<=20; num++)
     { cout << num << "\n";
       break;  } // This line exits the for loop immediately.

   cout << "That's all, folks!";
   return 0;
}
```

The following shows you the result of running this program. Notice the `break` immediately terminates the `for` loop. The `for` loop might as well not be in this program.

```
Here are the numbers from 1 to 20
1
That's all, folks!
```

2. The following program is an improved version of the preceding example. It asks users if they want to see another number. If they do, the `for` loop continues its next iteration. If they don't, the `break` statement terminates the `for` loop.

```
// Filename: C14BRAK2.CPP
// A for loop running at the user's request.
#include <iostream.h>
main()
{
   int num;   // Loop counter variable
   char ans;

   cout << "Here are the numbers from 1 to 20\n";

   for (num=1; num<=20; num++)
     { cout << num << "\n";
       cout << "Do you want to see another (Y/N)? ";
       cin >> ans;
       if ((ans == 'N') || (ans == 'n'))
         { break; }    // Will exit the for loop
                       // if user wants to.
     }

   cout << "\nThat's all, folks!\n";
   return 0;
}
```

The following display shows a sample run of this program. The `for` loop prints 20 numbers, as long as the user does not answer `N` to the prompt. Otherwise, the `break` terminates the `for` loop early. The statement after the body of the loop always executes next if the `break` occurs.

```
Here are the numbers from 1 to 20
1
Do you want to see another (Y/N)? Y
2
Do you want to see another (Y/N)? Y
3
Do you want to see another (Y/N)? Y
4
Do you want to see another (Y/N)? Y
5
Do you want to see another (Y/N)? Y
6
Do you want to see another (Y/N)? Y
7
Do you want to see another (Y/N)? Y
8
Do you want to see another (Y/N)? Y
9
Do you want to see another (Y/N)? Y
10
Do you want to see another (Y/N)? N

That's all, folks!
```

If you nest one loop inside another, the `break` terminates the "most active" loop (the innermost loop in which the `break` statement resides).

3. Use the *conditional* `break` (an `if` statement followed by a `break`) when you are missing data. For example, when you process data files or large amounts of user data-entry, you might expect 100 input numbers and receive only 95. You can use a `break` to terminate the `for` loop before it iterates the 96th time.

   Suppose the teacher that used the grade-averaging program in the preceding chapter (C13FOR4.CPP) entered an incorrect total number of students. Maybe she typed 16, but there are only 14 students. The previous `for` loop looped 16 times, no matter how many students there are, because it relies on the teacher's count.

The following grade averaging program is more sophisticated than the last one. It asks the teacher for the total number of students, but if the teacher wants, she can enter –99 as a student's score. The –99 is not averaged; it is used as a trigger value to break out of the `for` loop before its normal conclusion.

```
// Filename: C14BRAK3.CPP
// Computes a grade average with a for loop,
// allowing an early exit with a break statement.
#include <iostream.h>
#include <iomanip.h>
main()
{
   float grade, avg;
   float total=0.0;
   int num, count=0; // Total number of grades and counter
   int loopvar;               // Used to control for loop

   cout << "\n*** Grade Calculation ***\n\n";     // Title
   cout << "How many students are there? ";
   cin >> num;               // Get total number to enter.

   for (loopvar=1; loopvar<=num; loopvar++)
      { cout << "\nWhat is the next student's " <<
                "grade? (-99 to quit) ";
        cin >> grade;
        if (grade < 0.0)             // A negative number
                                     //    triggers break.
          { break; }              // Leave the loop early.
        count++;
        total += grade;  }         // Keep a running total.

   avg = total / count;
   cout << "\n\nThe average of this class is "<<
           setprecision(1) << avg;
   return 0;
}
```

Notice that `grade` is tested for less than 0, not –99.0. You cannot reliably use floating-point values to compare for

equality (due to their bit-level representations). Because no grade is negative, *any* negative number triggers the `break` statement. The following shows how this program works.

```
*** Grade Calculation ***

How many students are there? 10

What is the next student's grade? (-99 to quit) 87

What is the next student's grade? (-99 to quit) 97

What is the next student's grade? (-99 to quit) 67

What is the next student's grade? (-99 to quit) 89

What is the next student's grade? (-99 to quit) 94

What is the next student's grade? (-99 to quit) -99


The average of this class is: 86.8
```

The **continue** Statement

The `continue` statement causes C++ to skip all remaining statements in a loop.

The `break` statement exits a loop early, but the `continue` statement forces the computer to perform another iteration of the loop. If you put a `continue` statement in the body of a `for` or a `while` loop, the computer ignores any statement in the loop that follows `continue`.

The format of `continue` is

```
continue;
```

You use the `continue` statement when data in the body of the loop is bad, out of bounds, or unexpected. Instead of acting on the bad data, you might want to go back to the top of the loop and try another data value. The following examples help illustrate the use of the `continue` statement.

TIP: The `continue` statement forces a new iteration of any of the three loop constructs: the `for` loop, the `while` loop, and the `do-while` loop.

Figure 14.1 shows the difference between the `break` and `continue` statements.

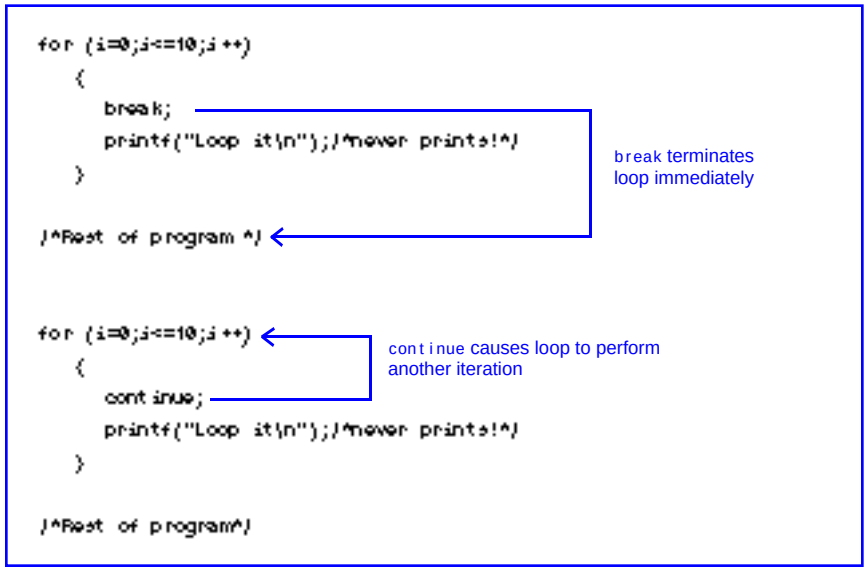
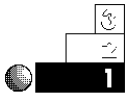


Figure 14.1. The difference between `break` and `continue`.

Examples



1. Although the following program seems to print the numbers 1 through 10, each followed by "C++ Programming," it does not. The `continue` in the body of the `for` loop causes an early finish to the loop. The first `cout` in the `for` loop executes, but the second does not—due to the `continue`.

```
// Filename: C14CON1.CPP
// Demonstrates the use of the continue statement.
#include <iostream.h>
main()
{
   int ctr;

   for (ctr=1; ctr<=10; ctr++)      // Loop 10 times.
     {  cout << ctr << " ";
        continue;         // Causes body to end early.
        cout << "C++ Programming\n";
     }
   return 0;
}
```

This program produces the following output:

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

On some compilers, you receive a warning message when you compile this type of program. The compiler recognizes that the second `cout` is *unreachable* code—it never executes due to the `continue` statement.

Because of this fact, most programs do not use a `continue`, except after an `if` statement. This makes it a conditional `continue` statement, which is more useful. The following two examples demonstrate the conditional use of `continue`.

2. This program asks users for five lowercase letters, one at a time, and prints their uppercase equivalents. It uses the ASCII table (see Appendix C, "ASCII Table") to ensure that users type lowercase letters. (These are the letters whose ASCII numbers range from 97 to 122.) If users do not type a lowercase letter, the program ignores the mistake with the `continue` statement.

```
// Filename: C14CON2.CPP
// Prints uppercase equivalents of five lowercase letters.
#include <iostream.h>
main()
```

```
{
   char letter;
   int ctr;

   for (ctr=1; ctr<=5; ctr++)
     { cout << "Please enter a lowercase letter ";
       cin >> letter;
       if ((letter < 97) || (letter > 122))  // See if
                                             // out-of-range.
          { continue; }                 // Go get another
       letter -= 32;   // Subtract 32 from ASCII value.
                                        // to get uppercase.
       cout << "The uppercase equivalent is " <<
               letter << "\n";
     }
   return 0;
}
```

Due to the `continue` statement, only lowercase letters are converted to uppercase.

3. Suppose you want to average the salaries of employees in your company who make over $10,000 a year, but you have only their monthly gross pay figures. The following program might be useful. It prompts for each monthly employee salary, annualizes it (multiplying by 12), and computes an average. The `continue` statement ensures that salaries less than or equal to $10,000 are ignored in the average calculation. It enables the other salaries to "fall through."

   If you enter `-1` as a monthly salary, the program quits and prints the result of the average.

```
// Filename: C14CON3.CPP
// Average salaries over $10,000
#include <iostream.h>
#include <iomanip.h>
main()
{
   float month, year;      // Monthly and yearly salaries
   float avg=0.0, total=0.0;
   int count=0;
```

```
   do
     { cout << "What is the next monthly salary (-1) " <<
              "to quit)? ";
       cin >> month;
       if ((year=month*12.00) <= 10000.00)  // Do not add
          { continue; }                     // low salaries.
       if (month < 0.0)
          { break; }           // Quit if user entered -1.
       count++;                 // Add 1 to valid counter.
       total += year;       // Add yearly salary to total.
     } while (month > 0.0);

   avg = total / (float)count;          // Compute average.
   cout << "\n\nThe average of high salaries " <<
           "is $" << setprecision(2) << avg;
   return 0;
}
```

Notice this program uses both a `continue` and a `break` statement. The program does one of three things, depending on each user's input. It adds to the total, continues another iteration if the salary is too low, or exits the `while` loop (and the average calculation) if the user types a `-1`.

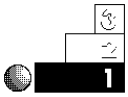
The following display is the output from this program:

```
What is the next monthly salary (-1 to quit)? 500.00
What is the next monthly salary (-1 to quit)? 2000.00
What is the next monthly salary (-1 to quit)? 750.00
What is the next monthly salary (-1 to quit)? 4000.00
What is the next monthly salary (-1 to quit)? 5000.00
What is the next monthly salary (-1 to quit)? 1200.00
What is the next monthly salary (-1 to quit)? -1


The average of high salaries is $36600.00
```

Review Questions

The answers to the review questions are in Appendix B.



1. For what do you use timing loops?
2. Why do timing loop ranges have to be adjusted for different types of computers?
3. Why do `continue` and `break` statements rarely appear without an `if` statement controlling them?

4. What is the output from the following section of code?

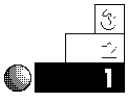
```
for (i=1; i<=10; i++)
   { continue;
     cout << "***** \n";
   }
```

5. What is the output from the following section of code?

```
for (i=1; i<=10; i++)
   { cout << "***** \n";
     break;
   }
```

6. To perform a long timing loop, why do you generally have to use a nested loop?

Review Exercises



1. Write a program that prints `C++ is fun` on-screen for ten seconds. (*Hint:* You might have to adjust the timing loop.)
2. Make the program in Exercise 1 flash the message `C++ is fun` for ten seconds. (*Hint:* You might have to use several timing loops.)
3. Write a grade averaging program for a class of 20 students. Ignore any grade less than 0 and continue until all 20 student grades are entered, or until the user types –99 to end the program early.

4. Write a program that prints the numerals from 1 to 14 in one column. To the right of the even numbers, print each number's square. To the right of the odd numbers, print each number's cube (the number raised to its third power).

Summary

In this chapter, you learned several additional ways to use and modify your program's loops. By adding timing loops, `continue` statements, and `break` statements, you can better control how each loop behaves. Being able to exit early (with the `break` statement) or continue the next loop iteration early (with the `continue` statement) gives you more freedom when processing different types of data.

The next chapter (Chapter 15, "The `switch` and `goto` Statements") shows you a construct of C++ that does not loop, but relies on the `break` statement to work properly. This is the `switch` statement, and it makes your program choices much easier to write.