The `while` Loop

The repetitive capabilities of computers make them good tools for processing large amounts of information. Chapters 12-15 introduce you to C++ constructs, which are the control and looping commands of programming languages. C++ constructs include powerful, but succinct and efficient, looping commands similar to those of other languages you already know.

The `while` loops enable your programs to repeat a series of statements, over and over, as long as a certain condition is always met. Computers do not get "bored" while performing the same tasks repeatedly. This is one reason why they are so important in business data processing.

This chapter teaches you the following:

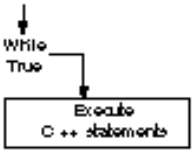
- The `while` loop
- The concept of loops
- The `do-while` loop
- Differences between `if` and `while` loops
- The `exit()` function
- The `break` statement
- Counters and totals

After completing this chapter, you should understand the first of several methods C++ provides for repeating program sections. This chapter's discussion of loops includes one of the most important uses for looping: creating counter and total variables.

The `while` Statement

The `while` statement is one of several C++ *construct statements.* Each construct (from *construction*) is a programming language statement—or a series of statements—that controls looping. The `while`, like other such statements, is a *looping statement* that controls the execution of a series of other statements. Looping statements cause parts of a program to execute repeatedly, as long as a certain condition is being met.

The format of the `while` statement is




```
while (test expression)
   { block of one or more C++ statements; }
```

The parentheses around `test expression` are required. As long as `test expression` is True (nonzero), the *block* of one or more C++ statements executes repeatedly until `test expression` becomes False (evaluates to zero). Braces are required before and after the body of the `while` loop, unless you want to execute only one statement. Each statement in the body of the `while` loop requires an ending semicolon.

The body of a `while` loop executes repeatedly as long as test expression is True.

The placeholder `test expression` usually contains relational, and possibly logical, operators. These operators provide the True-False condition checked in `test expression`. If `test expression` is False when the program reaches the `while` loop for the first time, the body of the `while` loop does not execute at all. Regardless of whether the body of the `while` loop executes no times, one time, or many times, the statements following the `while` loop's closing brace execute if `test expression` becomes False.

Because `test expression` determines when the loop finishes, the body of the `while` loop must change the variables used in `test expression`. Otherwise, `test expression` never changes and the `while` loop repeats forever. This is known as an *infinite loop,* and you should avoid it.

TIP: If the body of the `while` loop contains only one statement, the braces surrounding it are not required. It is a good habit to enclose all `while` loop statements in braces, however, because if you have to add statements to the body of the `while` loop later, your braces are already there.

The Concept of Loops

You use the loop concept in everyday life. Any time you have to repeat the same procedure, you are performing a loop—just as your computer does with the `while` statement. Suppose you are wrapping holiday gifts. The following statements represent the looping steps (in `while` format) that you follow while gift-wrapping.

```
while (there are still unwrapped gifts)
  { Get the next gift;
      Cut the wrapping paper;
      Wrap the gift;
      Put a bow on the gift;
      Fill out a name card for the gift;
      Put the wrapped gift with the others; }
```

Whether you have 3, 15, or 100 gifts to wrap, you use this procedure (loop) repeatedly until every gift is wrapped. For an example that is more easily computerized, suppose you want to total all the checks you wrote in the previous month. You could perform the following loop.

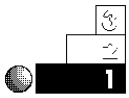
```
while (there are still checks from the last month to be totaled)
  { Add the amount of the next check to the total; }
```

The body of this pseudocode `while` loop has only one statement, but that single statement must be performed until you have added each one of the previous month's checks. When this loop ends (when no more checks from the previous month remain to be totaled), you have the result.

The body of a `while` loop can contain one or more C++ statements, including additional `while` loops. Your programs will be

more readable if you indent the body of a `while` loop a few spaces to the right. The following examples illustrate this.

Examples



1. Some programs presented earlier in the book require user input with `cin`. If users do not enter appropriate values, these programs display an error message and ask the user to enter another value, which is an acceptable procedure.

   Now that you understand the `while` loop construct, however, you should put the error message inside a loop. In this way, users see the message continually until they type proper input values, rather than once.

   The following program is short, but it demonstrates a `while` loop that ensures valid keyboard input. It asks users whether they want to continue. You can incorporate this program into a larger one that requires user permission to continue. Put a prompt, such as the one presented here, at the bottom of a text screen. The text remains on-screen until the user tells the program to continue executing.

Identify the file and include the necessary header file. In this program, you want to ensure the user enters Y or N. You have to store the user's answer, so declare the `ans` *variable as a character. Ask the users whether they want to continue, and get the response. If the user doesn't type Y or N, ask the user for another response.*


```
// Filename: C12WHIL1.CPP
// Input routine to ensure user types a
// correct response. This routine can be part
// of a larger program.
#include <iostream.h>
main()
{
   char ans;

   cout << "Do you want to continue (Y/N)? ";
   cin >> ans;                  // Get user's answer
```

```
   while ((ans != 'Y') && (ans != 'N'))
      { cout << "\nYou must type a Y or an N\n";  // Warn
                                                  // and ask
        cout << "Do you want to continue (Y/N)?"; // again.
        cin >> ans;
      }          // Body of while loop ends here.

   return 0;
}
```

Notice that the two `cin` functions do the same thing. You must use an initial `cin`, outside the `while` loop, to provide an answer for the `while` loop to check. If users type something other than Y or N, the program prints an error message, asks for another answer, then checks the new answer. This validation method is preferred over one where the reader only has one additional chance to succeed.

The `while` loop tests the test expression at the top of the loop. This is why the loop might never execute. If the test is initially False, the loop does not execute even once. The output from this program is shown as follows. The program repeats indefinitely, until the relational test is True (as soon as the user types either Y or N).

```
Do you want to continue (Y/N)? k

You must type a Y or an N
Do you want to continue (Y/N)? c

You must type a Y or an N
Do you want to continue (Y/N)? s

You must type a Y or an N
Do you want to continue (Y/N)? 5

You must type a Y or an N
Do you want to continue (Y/N)? Y
```

2. The following program is an example of an *invalid* `while` loop. See if you can find the problem.

```
// Filename: C12WHBAD.CPP
// Bad use of a while loop.
#include <iostream.h>
main()
{
   int a=10, b=20;
   while (a > 5)
      { cout << "a is " << a << ", and b is " << b << "\n";
        b = 20 + a; }
   return 0;
}
```

This `while` loop is an example of an infinite loop. It is vital that at least one statement inside the `while` changes a variable in the test expression (in this example, the variable `a`); otherwise, the condition is always True. Because the variable `a` does not change inside the `while` loop, this program will never end.

TIP: If you inadvertently write an infinite loop, you must stop the program yourself. If you use a PC, this typically means pressing Ctrl-Break. If you are using a UNIX-based system, your system administrator might have to stop your program's execution.

3. The following program asks users for a first name, then uses a `while` loop to count the number of characters in the name. This is a *string length program;* it counts characters until it reaches the null zero. Remember that the length of a string equals the number of characters in the string, not including the null zero.

```
// Filename: C12WHIL2.CPP
// Counts the number of letters in the user's first name.
#include <iostream.h>
main()
{
   char name[15];           // Will hold user's first name
```

```
   int count=0;       // Will hold total characters in name

   // Get the user's first name
   cout << "What is your first name? ";
   cin >> name;

   while (name[count] > 0) // Loop until null zero reached.
      {  count++; }                  // Add 1 to the count.

   cout << "Your name has " << count << " characters";
   return 0;
}
```

The loop continues as long as the value of the next character in the `name` array is greater than zero. Because the last character in the array is a null zero, the test is False on the name's last character and the statement following the body of the loop continues.

NOTE: A built-in string function called `strlen()` determines the length of strings. You learn about this function in Chapter 22, "Character, String, and Numeric Functions."

4. The previous string-length program's `while` loop is not as efficient as it could be. Because a `while` loop fails when its test expression is zero, there is no need for the greater-than test. By changing the test expression as the following program shows, you can improve the efficiency of the string length count.

```
// Filename: C12WHIL3.CPP
// Counts the number of letters in the user's first name.
#include <iostream.h>
main()
{
   char name[15];           // Will hold user's first name
   int count=0;       // Will hold total characters in name

   // Get the user's first name
```

```
   cout << "What is your first name? ";
   cin >> name;

   while (name[count])  // Loop until null zero is reached.
      {  count++; }                  // Add 1 to the count.

   cout << "Your name has " << count << " characters";
   return 0;
}
```

The do-while Loop

The body of the do-while loop executes at least once.

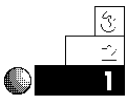
The do-while statement controls the do-while loop, which is similar to the while loop except the relational test occurs at the end (rather than beginning) of the loop. This ensures the body of the loop executes at least once. The do-while tests for a *positive relational test;* as long as the test is True, the body of the loop continues to execute.

The format of the do-while is

```
do
   { block of one or more C++ statements; }
while (test expression)
```

test expression must be enclosed in parentheses, just as it must in a while statement.

Examples



1. The following program is just like the first one you saw with the while loop (C12WHIL1.CPP), except the do-while is used. Notice the placement of test expression. Because this expression concludes the loop, user input does not have to appear before the loop and again in the body of the loop.

```
// Filename: C12WHIL4.CPP
// Input routine to ensure user types a
// correct response. This routine might be part
// of a larger program.
```

```
#include <iostream.h>
main()
{
   char ans;

   do
    { cout << "\nYou must type a Y or an N\n";    // Warn
                                                  // and ask
      cout << "Do you want to continue (Y/N) ?"; // again.
      cin >> ans; }            // Body of while loop
                               //    ends here.
   while ((ans != 'Y') && (ans != 'N'));

   return 0;
}
```

2. Suppose you are entering sales amounts into the computer to calculate extended totals. You want the computer to print the quantity sold, part number, and extended total (quantity times the price per unit), as the following program does.


```
// Filename: C12INV1.CPP
// Gets inventory information from user and prints
// an inventory detail listing with extended totals.
#include <iostream.h>
#include <iomanip.h>
main()
{
   int part_no, quantity;
   float cost, ext_cost;

   cout << "*** Inventory Computation ***\n\n";   // Title

   // Get inventory information.
   do
    { cout << "What is the next part number (-999 to end)? ";
      cin >> part_no;
      if (part_no != -999)
         { cout << "How many were bought? ";
           cin >> quantity;
           cout << "What is the unit price of this item? ";
```

```
            cin >> cost;
            ext_cost = cost * quantity;
            cout << "\n" << quantity << " of # " << part_no <<
                    " will cost " <<  setprecision(2) <<
                    ext_cost;
            cout << "\n\n\n";      // Print two blank lines.
           }
      } while (part_no != -999);       // Loop only if part
                                       //  number is not -999.

   cout << "End of inventory computation\n";
   return 0;
}
```

Here is the output from this program:

```
*** Inventory Computation ***

What is the next part number (-999 to end)? 213
How many were bought? 12
What is the unit price of this item? 5.66

12 of # 213 will cost 67.92


What is the next part number (-999 to end)? 92
How many were bought? 53
What is the unit price of this item? .23

53 of # 92 will cost 12.19


What is the next part number (-999 to end)? -999
End of inventory computation
```

The `do-while` loop controls the entry of the customer sales information. Notice the "trigger" that ends the loop. If the user enters –999 for the part number, the `do-while` loop quits because no part numbered –999 exists in the inventory.

However, this program can be improved in several ways. The invoice can be printed to the printer rather than the

screen. You learn how to direct your output to a printer in Chapter 21, “Device and Character Input/Output.” Also, the inventory total (the total amount of the entire order) can be computed. You learn how to total such data in the “Counters and Totals” section later in this chapter.

The `if` Loop Versus the `while` Loop

Some beginning programmers confuse the `if` statement with loop constructs. The `while` and `do-while` loops repeat a section of code multiple times, depending on the condition being tested. The `if` statement may or may not execute a section of code; if it does, it executes that section only once.

Use an `if` statement when you want to conditionally execute a section of code *once*, and use a `while` or `do-while` loop if you want to execute a section *more than once*. Figure 12.1 shows differences between the `if` statement and the two `while` loops.

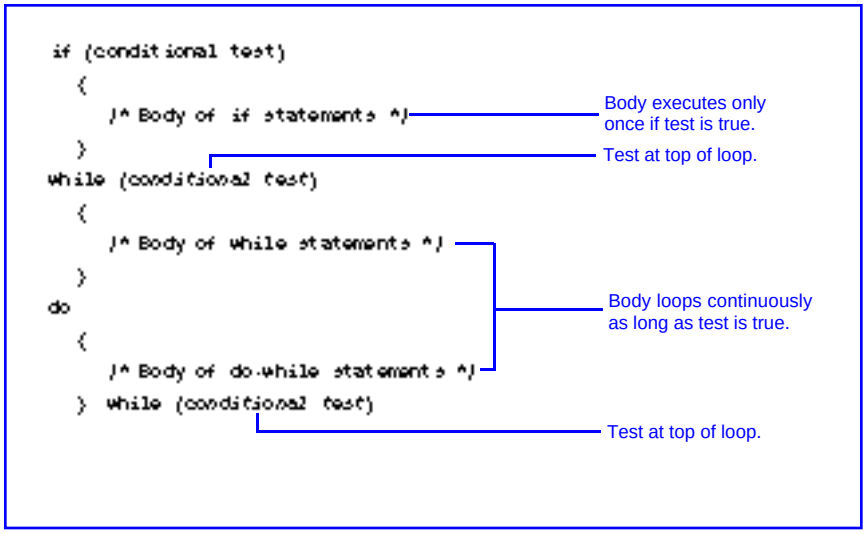


Figure 12.1. Differences between the `if` statement and the two `while` loops.

The exit() Function and break Statement

C++ provides the exit() function as a way to leave a program early (before its natural finish). The format of exit() is

```
exit(status);
```

where status is an optional integer variable or literal. If you are familiar with your operating system's return codes, status enables you to test the results of C++ programs. In DOS, status is sent to the operating system's errorlevel *environment variable*, where it can be tested by batch files.

The exit() function provides an early exit from your program.

Many times, something happens in a program that requires the program's termination. It might be a major problem, such as a disk drive error. Perhaps users indicate that they want to quit the program—you can tell this by giving your users a special value to type with cin or scanf(). You can isolate the exit() function on a line by itself, or anywhere else that a C++ statement or function can appear. Typically, exit() is placed in the body of an if statement to end the program early, depending on the result of some relational test.

Always include the stdlib.h header file when you use exit(). This file describes the operation of exit() to your program. Whenever you use a function in a program, you should know its corresponding #include header file, which is usually listed in the compiler's reference manual.

The break statement ends the current loop.

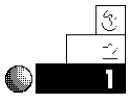
Instead of exiting an entire program, however, you can use the break statement to exit the current loop. The format of break is


```
break;
```

The break statement can go anywhere in a C++ program that any other statement can go, but it typically appears in the body of a while or do-while loop, used to leave the loop early. The following examples illustrate the exit() function and the break statement.

NOTE: The break statement exits only the most current loop. If you have a while loop in another while loop, break exits only the internal loop.

Examples



1. Here is a simple program that shows you how the `exit()` function works. This program looks as though it prints several messages on-screen, but it doesn't. Because `exit()` appears early in the code, this program quits immediately after `main()`'s opening brace.

```
// C12EXIT1.CPP
// Quits early due to exit() function.
#include <iostream.h>
#include <stdlib.h>              // Required for exit().
main()
{
   exit(0);                 // Forces program to end here.

   cout << "C++ programming is fun.\n";
   cout << "I like learning C++ by example!\n";
   cout << "C++ is a powerful language that is " <<
           "not difficult to learn.";

   return 0;
}
```

2. The `break` statement is not intended to be as strong a program exit as the `exit()` function. Whereas `exit()` ends the entire program, `break` quits only the loop that is currently active. In other words, `break` is usually placed inside a `while` or `do-while` loop to "simulate" a finished loop. The statement following the loop executes after a `break` occurs, but the program does not quit as it does with `exit()`.

   The following program appears to print `C++ is fun!` until the user enters `N` to stop it. The message prints only once, however, because the `break` statement forces an early exit from the loop.

```
// Filename: C12BRK.CPP
// Demonstrates the break statement.
#include <iostream.h>
main()
```

```
{
   char user_ans;

   do
     { cout << "C++ is fun! \n";
       break;                      // Causes early exit.
       cout << "Do you want to see the message again (N/Y)? ";
       cin >> user_ans;
     } while (user_ans == 'Y');

   cout << "That's all for now\n";
   return 0;
}
```

This program always produces the following output:

```
C++ is fun!
That's all for now
```

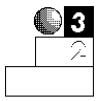
You can tell from this program's output that the `break` statement does not allow the `do-while` loop to reach its natural conclusion, but causes it to finish early. The final `cout` prints because only the current loop—and not the entire program—exits with the `break` statement.

3. Unlike the previous program, `break` usually appears after an `if` statement. This makes it a *conditional* `break`, which occurs only if the relational test of the `if` statement is True.

   A good illustration of this is the inventory program you saw earlier (C12INV1.CPP). Even though the users enter –999 when they want to quit the program, an additional `if` test is needed inside the `do-while`. The –999 ends the `do-while` loop, but the body of the `do-while` still needs an `if` test, so the remaining quantity and cost prompts are not given.

   If you insert a `break` after testing for the end of the user's input, as shown in the following program, the `do-while` will not need the `if` test. The `break` quits the `do-while` as soon as the user signals the end of the inventory by entering –999 as the part number.

```
// Filename: C12INV2.CPP
// Gets inventory information from user and prints
// an inventory detail listing with extended totals.
#include <iostream.h>
#include <iomanip.h>
main()
{
   int part_no, quantity;
   float cost, ext_cost;

   cout << "*** Inventory Computation ***\n\n";   // Title

   // Get inventory information
   do
    { cout << "What is the next part number (-999 to end)? ";
      cin >> part_no;
      if (part_no == -999)
        { break; }                  // Exit the loop if
                                    // no more part numbers.
      cout << "How many were bought? ";
      cin >> quantity;
      cout << "What is the unit price of this item? ";
      cin >> cost;
      cout << "\n" << quantity << " of # " << part_no <<
           " will cost " << setprecision(2) << cost*quantity;
      cout << "\n\n\n";           // Print two blank lines.
    } while (part_no != -999);        // Loop only if part
                                      // number is not -999.

   cout << "End of inventory computation\n";
   return 0;
}
```



4. You can use the following program to control the two other programs. This program illustrates how C++ can pass information to DOS with `exit()`. This is your first example of a menu program. Similar to a restaurant menu, a C++ menu program lists possible user choices. The users decide what they want the computer to do from the menu's available options. The mailing list application in Appendix F, "The Mailing List Application," uses a menu for its user options.

This program returns either a `1` or a `2` to its operating system, depending on the user's selection. It is then up to the operating system to test the `exit` value and run the proper program.

```
// Filename: C12EXIT2.CPP
// Asks user for his or her selection and returns
// that selection to the operating system with exit().
#include <iostream.h>
#include <stdlib.h>
main()
{
   int ans;

   do
     { cout << "Do you want to:\n\n";
       cout << "\t1. Run the word processor \n\n";
       cout << "\t2. Run the database program \n\n";
       cout << "What is your selection? ";
       cin >> ans;
     } while ((ans != 1) && (ans != 2));  // Ensures user
                                          // enters 1 or 2.
   exit(ans);  // Return value to operating system.
   return 0;   // Return does not ever execute due to exit().
}
```

Counters and Totals

Counting is important for many applications. You might have to know how many customers you have or how many people scored over a certain average in your class. You might want to count how many checks you wrote in the previous month with your computerized checkbook system.

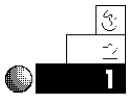
Before you develop C++ routines to count occurrences, think of how you count in your own mind. If you were adding a total number of something, such as the stamps in your stamp collection or the

number of wedding invitations you sent out, you would probably do the following:

Start at 0, and add 1 for each item being counted. When you are finished, you should have the total number (or the total count).

This is all you do when you count with C++: Assign 0 to a variable and add 1 to it every time you process another data value. The increment operator (++) is especially useful for counting.

Examples



1. To illustrate using a counter, the following program prints "Computers are fun!" on-screen 10 times. You can write a program that has 10 cout statements, but that would not be efficient. It would also be too cumbersome to have 5000 cout statements, if you wanted to print that same message 5000 times.

   By adding a while loop and a counter that stops after a certain total is reached, you can control this printing, as the following program shows.

```
// Filename: C12CNT1.CPP
// Program to print a message 10 times.
#include <iostream.h>
main()
{
   int ctr = 0;    // Holds the number of times printed.

   do
     { cout << "Computers are fun!\n";
       ctr++;                    // Add one to the count,
                                 // after each cout.
     } while (ctr < 10);         // Print again if fewer
                                 // than 10 times.
   return 0;
}
```

The output from this program is shown as follows. Notice that the message prints exactly 10 times.

```
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
Computers are fun!
```

The heart of the counting process in this program is the statement that follows.

```
ctr++;
```

You learned earlier that the increment operator adds 1 to a variable. In this program, the counter variable is incremented each time the `do-while` loops. Because the only operation performed on this line is the increment of `ctr`, the prefix increment (`++ctr`) produces the same results.

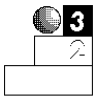
2. The previous program not only added to the counter variable, but also performed the loop a specific number of times. This is a common method of conditionally executing parts of a program for a fixed number of times.

   The following program is a password program. A password is stored in an integer variable. The user must correctly enter the matching password in three attempts. If the user does not type the correct password in that time, the program ends. This is a common method that dial-up computers use. They enable a caller to try the password a fixed number of times, then hang up the phone if that limit is exceeded. This helps deter people from trying hundreds of different passwords at any one sitting.

   If users guess the correct password in three tries, they see the secret message.

```
// Filename: C12PASS1.CPP
// Program to prompt for a password and
// check it against an internal one.
#include <iostream.h>
#include <stdlib.h>
main()
{
   int stored_pass = 11862;
   int num_tries = 0;     // Counter for password attempts.
   int user_pass;

   while (num_tries < 3)              // Loop only three
                                      // times.
      { cout << "What is the password (You get 3 tries...)? ";
        cin >> user_pass;
        num_tries++;                   // Add 1 to counter.
        if (user_pass == stored_pass)
           { cout << "You entered the correct password.\n";
             cout << "The cash safe is behind the picture " <<
                     "of the ship.\n";
             exit(0);
           }
        else
           { cout << "You entered the wrong password.\n";
          if (num_tries == 3)
          { cout << "Sorry, you get no more chances"; }
          else
          { cout << "You get " << (3-num_tries) <<
                  " more tries...\n";}
        }
      }                               // End of while loop.
   exit(0);
   return 0;
}
```

This program gives users three chances in case they type some mistakes. After three unsuccessful attempts, the program quits without displaying the secret message.



3. The following program is a letter-guessing game. It includes a message telling users how many tries they made before guessing the correct letter. A counter counts the number of these tries.

```
// Filename: C12GUES.CPP
// Letter-guessing game.
#include <iostream.h>
main()
{
   int tries = 0;
   char comp_ans, user_guess;

   // Save the computer's letter
   comp_ans = 'T';                // Change to a different
                                  //     letter if desired.

   cout << "I am thinking of a letter...";
   do
      { cout << "What is your guess? ";
        cin >> user_guess;
        tries++;   // Add 1 to the guess-counting variable.
        if (user_guess > comp_ans)
           { cout << "Your guess was too high\n";
             cout << "\nTry again...";
           }
        if (user_guess < comp_ans)
           { cout << "Your guess was too low\n";
             cout << "\nTry again...";
           }
       } while (user_guess != comp_ans);  // Quit when a
                                          // match is found.

   // They got it right, let them know.
   cout << "*** Congratulations!  You got it right! \n";
   cout << "It took you only " << tries <<
           " tries to guess.";
   return 0;
}
```

Here is the output of this program:

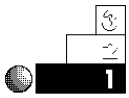
```
I am thinking of a letter...What is your guess? E
Your guess was too low

Try again...What is your guess? X
Your guess was too high

Try again...What is your guess? H
Your guess was too low

Try again...What is your guess? O
Your guess was too low

Try again...What is your guess? U
Your guess was too high

Try again...What is your guess? Y
Your guess was too high

Try again...What is your guess? T
*** Congratulations!  You got it right!
It took you only 7 tries to guess.
```

Producing Totals

Writing a routine to add values is as easy as counting. Instead of adding 1 to the counter variable, you add a value to the total variable. For instance, if you want to find the total dollar amount of checks you wrote during December, you can start at nothing (0) and add the amount of every check written in December. Instead of building a count, you are building a total.

When you want C++ to add values, just initialize a `total` variable to zero, then add each value to the total until you have included all the values.

Examples



1. Suppose you want to write a program that adds your grades for a class you are taking. The teacher has informed you that you earn an *A* if you can accumulate over 450 points.

   The following program keeps asking you for values until you type –1. The –1 is a signal that you are finished entering grades and now want to see the total. This program also prints a congratulatory message if you have enough points for an *A*.

```
// Filename: C12GRAD1.CPP
// Adds grades and determines whether you earned an A.
#include <iostream.h>
include <iomanip.h>
main()
{
   float total_grade=0.0;
   float grade;              // Holds individual grades.

   do
   { cout << "What is your grade? (-1 to end) ";
     cin >> grade;
     if (grade >= 0.0)
      { total_grade += grade; }        // Add to total.
   } while (grade >= 0.0);     // Quit when -1 entered.

   // Control begins here if no more grades.
   cout << "\n\nYou made a total of " << setprecision(1) <<
            total_grade << " points\n";
   if (total_grade >= 450.00)
      { cout << "** You made an A!!"; }

   return 0;
}
```

Notice that the `-1` response is not added to the total number of points. This program checks for the `-1` before adding to `total_grade`. Here is the output from this program:

```
What is your grade? (-1 to end) 87.6
What is your grade? (-1 to end) 92.4
What is your grade? (-1 to end) 78.7
What is your grade? (-1 to end) -1

You made a total of  258.7 points
```

2. The following program is an extension of the grade-calculating program. It not only totals the points, but also computes their average.

   To calculate the average grade, the program must first determine how many grades were entered. This is a subtle problem because the number of grades to be entered is unknown in advance. Therefore, every time the user enters a valid grade (not –1), the program must add 1 to a counter as well as add that grade to the `total` variable. This is a combination counting and totaling routine, which is common in many programs.

```
// Filename: C12GRAD2.CPP
// Adds up grades, computes average,
// and determines whether you earned an A.
#include <iostream.h>
#include <iomanip.h>
main()
{
   float total_grade=0.0;
   float grade_avg = 0.0;
   float grade;
   int grade_ctr = 0;

   do
   { cout << "What is your grade? (-1 to end) ";
     cin >> grade;
     if (grade >= 0.0)
      { total_grade += grade;              // Add to total.
        grade_ctr ++; }                    // Add to count.
   } while (grade >= 0.0);          // Quit when -1 entered.
```

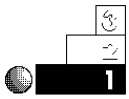
```
    // Control begins here if no more grades.
    grade_avg = (total_grade / grade_ctr);       // Compute
                                                 // average.
    cout << "\nYou made a total of " << setprecision(1) <<
            total_grade << " points.\n";
    cout << "Your average was " << grade_avg << "\n";
    if (total_grade >= 450.0)
       { cout << "** You made an A!!"; }
    return 0;
}
```

Below is the output of this program. Congratulations! You are on your way to becoming a master C++ programmer.

```
What is your grade? (-1 to end) 67.8
What is your grade? (-1 to end) 98.7
What is your grade? (-1 to end) 67.8
What is your grade? (-1 to end) 92.4
What is your grade? (-1 to end) -1

You made a total of 326.68 points.
Your average was 81.7
```

Review Questions

The answers to the review questions are in Appendix B.



1. What is the difference between the `while` loop and the `do-while` loop?
2. What is the difference between a total variable and a counter variable?
3. Which C++ operator is most useful for counting?
4. True or false: Braces are not required around the body of `while` and `do-while` loops.

5. What is wrong with the following code?

```
while (sales > 50)
   cout << "Your sales are very good this month.\n";
   cout << "You will get a bonus for your high sales\n";
```

6. What file must you include as a header file if you use `exit()`?

7. How many times does this `printf()` print?

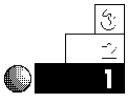
```
int a=0;
do
   { printf("Careful \n");
     a++; }
while (a > 5);
```

8. How can you inform DOS of the program exit status?

9. What is printed to the screen in the following section of code?

```
a = 1;
while (a < 4)
   { cout << "This is the outer loop\n";
     a++;
     while (a <= 25)
        { break;
          cout << "This prints 25 times\n"; }
   }
```

Review Exercises



1. Write a program with a `do-while` loop that prints the numerals from 10 to 20 (inclusive), with a blank line between each number.

2. Write a weather-calculator program that asks for a list of the previous 10 days' temperatures, computes the average, and prints the results. You have to compute the total as the input occurs, then divide that total by 10 to find the average. Use a `while` loop for the 10 repetitions.

3. Rewrite the program in Exercise 2 using a `do-while` loop.

4. Write a program, similar to the weather calculator in Exercise 2, but generalize it so it computes the average of any number of days' temperatures. (*Hint:* You have to count the number of temperatures to compute the final average.)

5. Write a program that produces your own ASCII table on-screen. Don't print the first 31 characters because they are nonprintable. Print the codes numbered 32 through 255 by storing their numbers in integer variables and printing their ASCII values using `printf()` and the `"%c"` format code.

Summary

This chapter showed you two ways to produce a C++ loop: the `while` loop and the `do-while` loop. These two variations of `while` loops differ in where they test their `test condition` statements. The `while` tests at the beginning of its loop, and the `do-while` tests at the end. Therefore, the body of a `do-while` loop always executes at least once. You also learned that the `exit()` function and `break` statement add flexibility to the `while` loops. The `exit()` function terminates the program, and the `break` statement terminates only the current loop.

This chapter explained two of the most important applications of loops: counters and totals. Your computer can be a wonderful tool for adding and counting, due to the repetitive capabilities offered with `while` loops.

The next chapter extends your knowledge of loops by showing you how to create a *determinate* loop, called the `for` loop. This feature is useful when you want a section of code to loop for a specified number of times.