with less effort if your coding style is clear and consistent. This translates into fewer coding errors.

I recommend that as a beginner you mimic the style you see in this book. You can change it later when you've gained some experience of your own.

When working on a program with several programmers, it's just as important that you all use the same style to avoid a Tower of Babel effect with conflicting and confusing styles. Every project that I've ever worked on had a coding manual that articulated sometimes in excruciating detail exactly how an `if` statement was to be laid out, how far to indent for `case`, and whether to put a blank line after the `break` statements, to name just a few examples.

Fortunately, Code::Blocks can help. The Code::Blocks editor understands C++. It will automatically indent the proper number of spaces for you after an open brace, and it will outdent when you type in the closed brace to align statements properly.

You can run the 'Source code formatter' plug-in that comes with Code::Blocks. With the file you are working on open and the project active, select Plugins➪ Source Code Formatter (AStyle). This will reformat the current file using the standard indention rules.

C++ doesn't care about indention. All whitespace is the same to it. Indention is there to make the resulting program easier to read and understand.

Establishing variable naming conventions

There is more debate about the naming of variables than about how many angels would fit on the head of a pin. I use the following rules when naming variables:

- **The first letter is lowercase and indicates the type of the variable.** n for `int`, c for `char`, b for `bool`. You'll see others in later chapters. This is very helpful when using the variable because you immediately know its type.
- **Names of variables are descriptive.** No variables names like `x` or `y`. I'm too old — I need something that I can recognize when I try to read my own program tomorrow or next week or next year.
- **Multiple word names use uppercase at the beginning of each word with no underscores between words.** I save underscores for a particular application, which I describe in Chapter 12.

I expand on these rules in chapters involving other types of C++ objects (such as functions in Chapter 11 and classes in Chapter 19).

Finding the First Error with a Little Help

My first version of the Conversion program appeared as follows (it appears on the enclosed CD-ROM as ConversionError1):

```
//
//  Conversion - Program to convert temperature from
//               Celsius degrees into Fahrenheit:
//               Fahrenheit = Celsius  * (212 - 32)/100 + 32
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // enter the temperature in Celsius
    int nCelsius;
    cout << "Enter the temperature in Celsius: ";

    // convert Celsius into Fahrenheit values
    int nFahrenheit;
    nFahrenheit = 9/5 * nCelsius + 32;

    // output the results (followed by a NewLine)
    cout << "Fahrenheit value is: ";
    cout << nFahrenheit << endl;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

During the build step, I get my first indication that there's a problem — Code::Blocks generates the following warning message:

```
In function 'int main(int char**)':
warning: 'nCelsius' is used uninitialized in this function
=== Build finished: 0 errors, 1 warnings ===
```

How bad can this be? After all, it's just a warning, right? So I decide to push forward and execute the program anyway.

Sure enough, I get the following meaningless output without giving me a chance to enter the Celsius temperature:

```
Enter the temperature in Celsius:
Fahrenheit value is:110
Press any key to continue . . .
```

Referring to the prompt, I can see that I have forgotten to input a value for `nCelsius`. The program proceeded forward calculating a Fahrenheit temperature based upon whatever garbage happened to be in `nCelsius` when it was declared.

Adding the following line immediately after the prompt gets rid of the warning and solves the first problem:

```
cin >> nCelsius;
```

The moral to this story is "Pay attention to warnings!" A warning almost always indicates a problem in your program. You shouldn't even start to test your programs until you get a clean build: no errors and no warnings. If that's not possible, at least convince yourself that you understand the reason for every warning generated.

Finding the Run-Time Error

Once all the warnings are gone, it's time to start testing. Good testing requires an organized approach. First, you decide the test data that you're going to use. Next, you determine what output you expect for each of the given test inputs. Then you run the program and compare the actual results with the expected results. What could be so hard?

Formulating test data

Determining what test data to use is part engineering and part black art. The engineering part is that you want to select data such that every statement in your program gets executed at least once. That means every branch of every `if` statement and every case of every `switch` statement gets executed at least once.

Having every statement execute at least once is called *full statement coverage* and is considered the minimum acceptable testing criteria. The chance of programming mistakes making it into the field is just too high without executing every statement at least once under test conditions.

This simple program has only one path and contains no branches.

The black art is looking at the program and determining where errors might lie in the calculation. For some reason, I just assume that every test should include the key values of 0 and 100 degrees Celsius. To that, I will add one negative value and one value in the middle between 0 and 100. Before I start, I use a handy-dandy conversion program to look up the equivalent temperature in Fahrenheit, as shown in Table 8-1.

Table 8-1 Test Data for the Conversion Program

Input Celsius	*Resulting Fahrenheit*
0	32
100	212
-40	-40
50	122

Executing the test cases

Running the tests is simply a matter of executing the program and supplying the input values from Table 8-1. The first case generates the following results:

```
Enter the temperature in Celsius: 0
Fahrenheit value is: 32
Press any key to continue . . .
```

So far, so good. The second data case generates the following output:

```
Enter the temperature in Celsius: 100
Fahrenheit value is: 132
Press any key to continue . . .
```

This doesn't match the expected value. Houston, we have a problem.

The value of 132 degrees is not completely unreasonable. That's why it's important to decide what the expected results are before you start. Otherwise, reasonable but incorrect results can slip by undetected.

Seeing what's going on in your program

What could be wrong? I check over the calculations and everything looks fine. I need to get a peek at what's going on in the calculation. A way to get at the internals of your program is to add output statements. I want to print the values going into each of the calculations. I also need to see the intermediate results. To do so, I break the calculation into its parts that I can print. Keep the original expression as a comment so you don't forget where you came from.

This version of the program is included on the enclosed CD-ROM as ConversionError2.

This version of the program includes the following changes:

```
// nFahrenheit = 9/5 * nCelsius + 32;
cout << "nCelsius = " << nCelsius << endl;
int nFactor = 9 / 5;
cout << "nFactor = " << nFactor << endl;
int nIntermediate = nFactor * nCelsius;
cout << "nIntermediate = " << nIntermediate << endl;
nFahrenheit = nIntermediate + 32;
cout << "nFahrenheit = " << nFahrenheit << endl;
```

I display the value of `nCelsius` to make sure that it got read properly from the user input. Next, I try to display the intermediate results of the conversion calculation in the same order that C++ will. First to go is the calculation `9 / 5`, which I save into a variable I name `nFactor` (the name isn't important). This value is multiplied by `nCelsius`, the results of which I save into `nIntermediate`. Finally, this value will get added to 32 to generate the result, which is stored into `nFahrenheit`.

By displaying each of these intermediate values, I can see what's going on in my calculation. Repeating the error case, I get the following results:

```
Enter the temperature in Celsius: 100
nCelsius = 100
nFactor = 1
nIntermediate = 100
nFahrenheit = 132
Fahrenheit value is: 132
Press any key to continue . . .
```

Right away I see a problem: `nFactor` is equal to 1 and not 9 / 5. Then the problem occurs to me; integer division rounds down to the nearest integer value. Integer 9 divided by integer 5 *is* 1.

I can avoid this problem by performing the multiply before the divide. There will still be a small amount of integer round-off, but it will only amount to a single degree.

Another solution would be to use decimal variables that can retain fractional values. You'll see that solution in Chapter 14.

The resulting formula appears as follows:

```
nFahrenheit = nCelsius * 9/5 + 32;
```

This is the version of the calculation that appears on the CD-ROM in the original Conversion program.

Now rerunning the tests, I get the following:

```
Enter the temperature in Celsius: 0
Fahrenheit value is: 32
Press any key to continue . . .
```

```
Enter the temperature in Celsius: 100
Fahrenheit value is: 212
Press any key to continue . . .
```

```
Enter the temperature in Celsius: -40
Fahrenheit value is: -40
Press any key to continue . . .
```

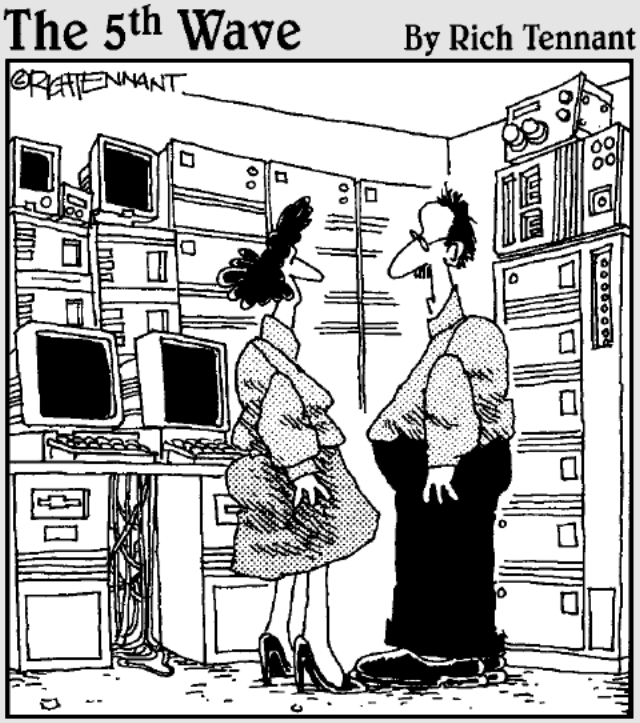
```
Enter the temperature in Celsius: 50
Fahrenheit value is: 122
Press any key to continue . . .
```

This matches the expected values from Table 8-1.

Notice that, after making the change, I started over from the beginning, supplying all four test cases — not just the values that didn't work properly the first time. Any changes to the calculation invalidate all previous tests.

Part III

Becoming a Functional Programmer



"We're researching molecular/digital technology that moves massive amounts of information across binary pathways that interact with free-agent programs capable of making decisions and performing logical tasks. We see applications in really high-end doorbells."

In this part . . .

Now that you've mastered the basics of simple expressions, it's time for you to learn about loops, how to get into them, and, even more importantly, how to get out of them. You'll also see how to break a large program into smaller components that are easier to program. In the last chapter of this part, you'll see some more techniques for debugging your programs.

Chapter 9

while Running in Circles

In This Chapter

- Looping using the `while` statement
- Breaking out of the middle of a loop
- Avoiding the deadly infinite loop
- Nesting loops within loops

Decision making is a fundamental part of almost every program you write, which I initially emphasize in Chapter 1. However, another fundamental feature that is clear — even in the simple Lug Nut Removal algorithm — is the ability to loop. That program turned the wrench in a loop until the lug nut fell off, and it looped from one lug nut to the other until the entire wheel came off. This chapter introduces you to two of the three looping constructs in C++.

Creating a while Loop

The `while` loop has the following format:

```
while (expression)
{
    // stuff to do in a loop
}

// continue here once expression is false
```

When a program comes upon a `while` loop, it first evaluates the expression in the parentheses. If this expression is `true`, then control passes to the first line inside the `{`. When control reaches the `}`, the program returns back to the expression and starts over. Control continues to cycle through the code in the braces until `expression` evaluates to `false` (or until something else breaks the loop — more on that a little later in this chapter).

The following Factorial program demonstrates the `while` loop:

```
Factorial(N) = N * (N-1) * (N-2) * ... * 1
```

```
//
//  Factorial - calculate factorial using the while

//              construct.
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // enter the number to calculate the factorial of
    int nTarget;
    cout << "This program calculates factorial.\n"
         << "Enter a number to take factorial of: ";
    cin  >> nTarget;

    // start with an accumulator that's initialized to 1
    int nAccumulator = 1;
    int nValue = 1;
    while (nValue <= nTarget)
    {
        cout << nAccumulator << " * "
             << nValue << " equals ";
        nAccumulator = nAccumulator * nValue;
        cout << nAccumulator << endl;
        nValue++;
    }

    // display the result
    cout << nTarget << " factorial is "
         << nAccumulator << endl;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The program starts by prompting the user for a target value. The program reads this value into `nTarget`. The program then initializes both `nAccumulator` and `nValue` to 1 before entering the loop.

(Pay attention — this is the interesting part.) The program compares `nValue` to `nTarget`. Assume that the user had entered a target value of 5. On the first loop, the question becomes, "Is 1 less than or equal to 5?" The answer is

obviously `true`, so control flows into the loop. The program outputs the value of `nAccumulator` (1) and `nValue` (also 1) before multiplying `nAccumulator` by `nValue` and storing the result back into `nAccumulator`.

The last statement in the loop increments `nValue` from 1 to 2.

That done, control passes back up to the `while` statement where `nValue` (now 2) is compared to `nTarget` (still 5). "Is 2 less than or equal to 5?" Clearly, yes; so control flows back into the loop. `nAccumulator` is now set to the result of `nAccumulator` (1) times `nValue` (2). The last statement increments `nValue` to 3.

This cycle of fun continues until `nValue` reaches the value 6, which is no longer less than or equal to 5. At that point, control passes to the first statement beyond the closed brace `}`. This is shown graphically in Figure 9-1.

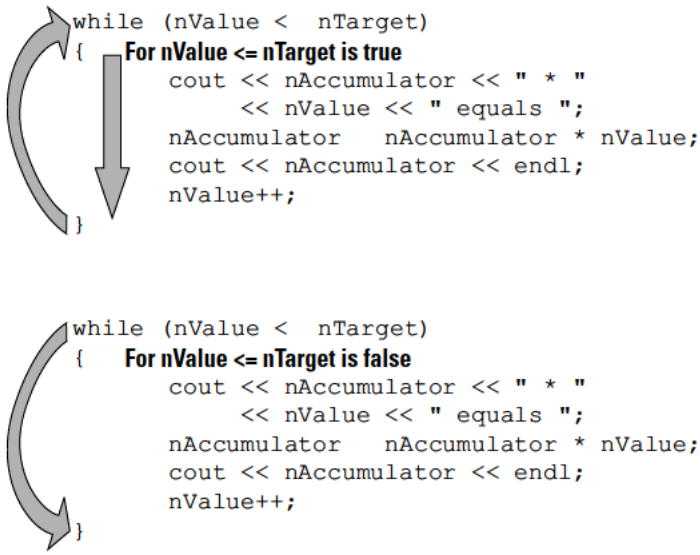


Figure 9-1: Control continues to cycle through the body of a `while` loop until the conditional expression evaluates to `false`.

The actual output from the program appears as follows for an input value of 5:

```
This program calculates factorial.
Enter a number to take factorial of: 5
1 * 1 equals 1
1 * 2 equals 2
2 * 3 equals 6
6 * 4 equals 24
24 * 5 equals 120
5 factorial is 120
Press any key to continue . . .
```

You are not guaranteed that the code within the braces of a `while` loop is executed at all: If the conditional expression is false the first time it's evaluated, control passes around the braces without ever diving in. Consider, for example, the output from the Factorial program when the user enters a target value of 0:

```
This program calculates factorial.
Enter a number to take factorial of: 0
0 factorial is 1
Press any key to continue . . .
```

No lines of output are generated from within the loop because the condition "Is `nValue` less than or equal to 0" was false even for the initial value of 1. The body of the `while` loop was never executed.

Breaking out of the Middle of a Loop

Sometimes the condition that causes you to terminate a loop doesn't occur until somewhere in the middle of the loop. This is especially true when testing user input for some termination character. C++ provides these two control commands to handle this case:

- `break` exits the inner most loop immediately.
- `continue` passes control back to the top of the loop.

The following Product program demonstrates both `break` and `continue`. This program multiplies positive values entered by the user until the user enters a negative number. The program ignores zero.

```
//
//  Product - demonstrate the use of break and continue.
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // enter the number to calculate the factorial of
    cout << "This program multiplies the numbers\n"
         << "entered by the user. Enter a negative\n"
         << "number to exit. Zeroes are ignored.\n"
         << endl;

    int nProduct = 1;
    while (true)
    {
```

```
        int nValue;
        cout << "Enter a number to multiply: ";
        cin  >> nValue;
        if (nValue < 0)
        {
            cout << "Exiting." << endl;
            break;
        }
        if (nValue == 0)
        {
            cout << "Ignoring zero." << endl;
            continue;
        }

        // multiply accumulator by this value and
        // output the result
        cout << nProduct << " * " << nValue;
        nProduct *= nValue;
        cout << " is " << nProduct << endl;
    }

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The program starts out with an initial value of `nProduct` of 1. The program then evaluates the logical expression `true` to see if it's true. It is.

There aren't too many rules that hold in C++ without exception, but here's one: `true` is always true.

The program then enters the loop to prompt the user for another value to multiply times `nProduct`, the accumulated product of all numbers entered so far. If the value entered is negative, then the program outputs the phrase `"Exiting."` before executing the `break`, which passes control out of the loop.

If the value entered is not negative, control passes to the second `if` statement. If `nValue` is equal to zero, then the program outputs the messages `"Ignoring zero."` before executing the `continue` statement which passes control back to the top of the loop to allow the user to enter another value.

If `nValue` is neither less than zero nor zero, then control flows down to where `nValue` is multiplied by `nProduct` using the special assignment operator (see Chapter 4 if you don't remember this one):

```
nProduct *= nValue;
```

Why is "break" necessary?

You might be tempted to wonder why `break` is really necessary. What if I had coded the loop in the Product example program as

```
int nProduct = 1;
int nValue = 1;
while (nValue > 0)
{
    cout << "Enter a number to multiply: ";
    cin  >> nValue;

    cout << nProduct << " * " << nValue;
    nProduct *= nValue;
    cout << " is " << nProduct << endl;
}
```

You might think that as soon as the user enters a negative value for `nValue`, the expression `nValue > 0` is no longer true and control immediately exits the loop — unfortunately, this is not the case.

The problem is that the logical expression is only evaluated at the beginning of each pass through the loop. Control doesn't immediately fly out of the body of the loop as soon as the condition ceases to be true. An `if` statement followed by a `break` allows me to move the conditional expression into the body of the loop where the value of `nValue` is assigned.

This expression is the same as:

```
nProduct = nProduct * nValue;
```

The output from a sample run from this program appears as follows:

```
This program multiplies the numbers
entered by the user. Enter a negative
number to exit. Zeroes are ignored.

Enter a number to multiply: 2
1 * 2 is 2
Enter a number to multiply: 5
2 * 5 is 10
Enter a number to multiply: 0
Ignoring zero.
Enter a number to multiply: 3
10 * 3 is 30
Enter a number to multiply: -1
Exiting.
Press any key to continue . . .
```

Nested Loops

The body of a loop can itself contain a loop in what is known as *nested loops*. The inner loop must execute to completion during each time through the outer loop.

I have created a program that uses nested loops to create a multiplication table of the form:

```
    0    1    2    3    4    5    6    7    8    9
0  0*0  0*1  0*2  0*3  0*4  0*5  0*6  0*7  0*8  0*9
1  1*0  1*1  1*2  1*3  1*4  1*5  1*6  1*7  1*8  1*9
2  2*0  2*1  2*2  2*3  2*4  2*5  2*6  2*7  2*8  2*9
//... and so on...
```

You can see that for row 0, the program will need to iterate from column 0 through column 9. The program will repeat the process for row 1 and again for row 2 and so on right down to row 9. This implies the need for two loops: an inner loop to iterate over the columns and a second outer loop to iterate over the rows. Each position in the table is simply the row number times the column number.

This is exactly how the following NestedLoops program works:

```
//
//  NestedLoops - this program uses a nested loop to
//                calculate the multiplication table.
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // display the column headings
    int nColumn = 0;
    cout << "    ";
    while (nColumn < 10)
    {
        // set the display width to two characters
        // (even for one digit numbers)
        cout.width(2);

        // now display the column number
        cout << nColumn << " ";

        // increment to the next column
        nColumn++;
    }
    cout << endl;

    // now go loop through the rows
    int nRow = 0;
```

```
    while (nRow < 10)
    {
        // start with the row value
        cout << nRow << " - ";

        // now for each row, start with column 0 and
        // go through column 9
        nColumn = 0;
        while(nColumn < 10)
        {
            // display the product of the column*row
            // (use 2 characters even when product is
            // a single digit)
            cout.width(2);
            cout << nRow * nColumn << " ";

            // go to next column
            nColumn++;
        }

        // go to next row
        nRow++;
        cout << endl;
    }

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The first section creates the column headings. This section initializes `nColumn` to 0. It then iterates through `nColumn` printing out its value separated by a space until `nColumn` reaches 10. At this point, the program exits the first loop and then tacks a new line on the end to finish the row. This is shown graphically in Figure 9-2.

Executing just this section alone generates the following output:

```
 0  1  2  3  4  5  6  7  8  9
```

This program demonstrates an unfair advantage that I have. The expression `cout.width(2)` sets the display width to two columns — C++ will pad a space on the left for single-digit numbers. I know it's cheating to make use of a feature that I don't present to the reader until Chapter 31, but it's very difficult to get the columns to line up without resorting to fixed-width output.

The second set of loops, the nested loops, starts at `nRow` equal to 0. The program prints out the row number followed by a dash before launching into a

second loop that starts `nColumn` at 0 again and iterates it back up to 9. For each pass through this inner loop, the program sets the output width to two spaces and then displays `nRow * nColumn` followed by a space.

```
// display the column headings
int nColumn   0;
while (nColumn < 10)
{
    // now display the column number
    cout << nColumn << " ";

    // increment to the next column
    nColumn++;
}
//go to the next row
cout << endl;
```

Output:

Figure 9-2: The first loop outputs the column headings.

The display width resets itself each time you output something, so it's necessary to set it back to two each time before outputting a number.

The program outputs a newline to move output to the next row each time it increments `nRow`. This is shown graphically in Figure 9-3.

The output from this program appears as follows:

```
      0  1  2  3  4  5  6  7  8  9
0 -   0  0  0  0  0  0  0  0  0  0
1 -   0  1  2  3  4  5  6  7  8  9
2 -   0  2  4  6  8 10 12 14 16 18
3 -   0  3  6  9 12 15 18 21 24 27
4 -   0  4  8 12 16 20 24 28 32 36
5 -   0  5 10 15 20 25 30 35 40 45
6 -   0  6 12 18 24 30 36 42 48 54
7 -   0  7 14 21 28 35 42 49 56 63
8 -   0  8 16 24 32 40 48 56 64 72
9 -   0  9 18 27 36 45 54 63 72 81
Press any key to continue . . .
```

There is nothing magic about 0 through 9 in this table. I could just have easily created a 12 x 12 multiplication table (or any other combination) by changing the comparison expression in the three `while` loops. However, for anything larger than 10 x 10, you will need to increase the minimum width to accommodate the three-digit products. Use `cout.width(3)`.

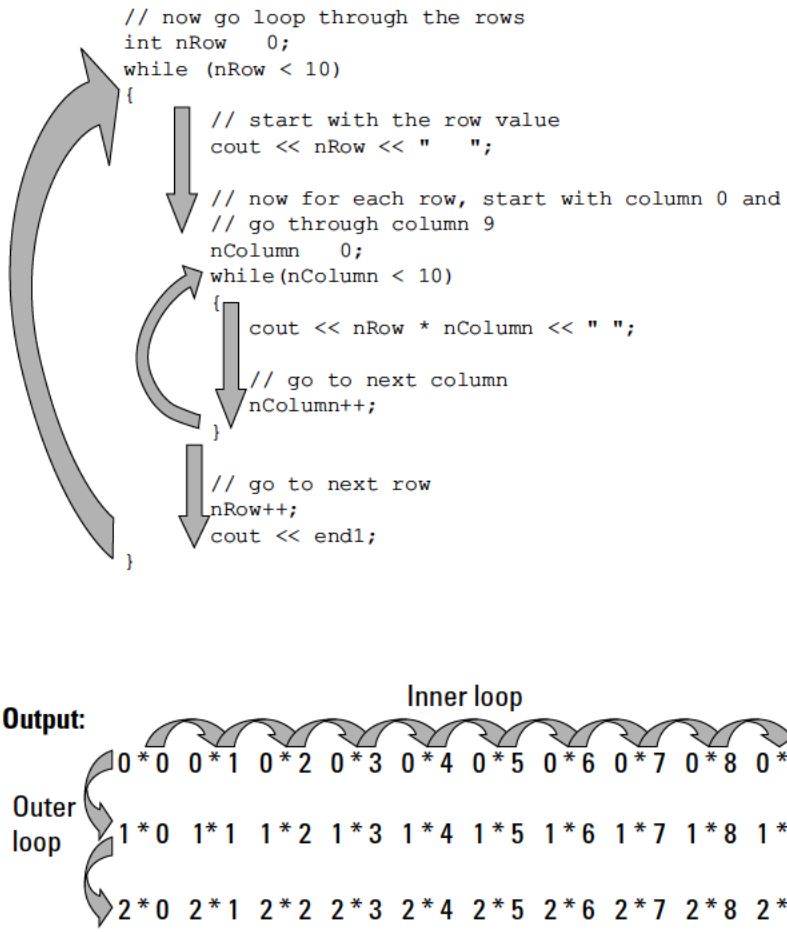


Figure 9-3: The inner loop iterates from left to right across the columns, while the outer loop iterates from top to bottom down the rows.

Chapter 10

Looping for the Fun of It

In This Chapter

- Introducing the `for` loop
- Reviewing an example ForFactorial program
- Using the comma operator to get more done in a single `for` loop

The most basic of all control structures is the `while` loop, which is the topic of Chapter 9. This chapter introduces you its sibling, the `for` loop. Though not quite as flexible, the `for` loop is actually the more popular of the two — it has a certain elegance that is hard to ignore.

The for Parts of Every Loop

If you look again at the examples in Chapter 9, you'll notice that most loops have four essential parts. (This feels like breaking down a golf swing into its constituent parts.)

- **The setup:** Usually the setup involves declaring and initializing an `increment` variable. This generally occurs immediately before the `while`.
- **The test expression:** The expression within the `while` loop that will cause the program to either execute the loop or exit and continue on. This always occurs within the parentheses following the keyword `while`.
- **The body:** This is the code within the braces.
- **The increment:** This is where the increment variable is incremented. This usually occurs at the end of the body.

In the case of the Factorial program, the four parts looked like this:

```
int nValue = 1;              // the setup
while (nValue <= nTarget)    // the test expression
{                            // the body
    cout << nAccumulator << " * "
```

```
        << nValue << " equals ";
    nAccumulator = nAccumulator * nValue;
    cout << nAccumulator << endl;
    nValue++;                // the increment
}
```

The `for` loop incorporates these four parts into a single structure using the keyword `for`:

```
for(setup; test expression; increment)
{
    body;
}
```

The flow is shown graphically in Figure 10-1.

1. As the CPU comes innocently upon the `for` keyword, control is diverted to the `setup` clause.
2. Once the setup has been performed, control moves over to the `test expression`.
3. (a) If the `test expression` is `true`, control passes to the body of the `for` loop.

   (b) If the `test expression` is `false`, control passes to the next statement after the closed brace.
4. Once control has passed through the body of the loop, the CPU is forced to perform a U-turn back up to the `increment` section of the loop.

   That done, control returns to the `test expression` and back to Step 3.

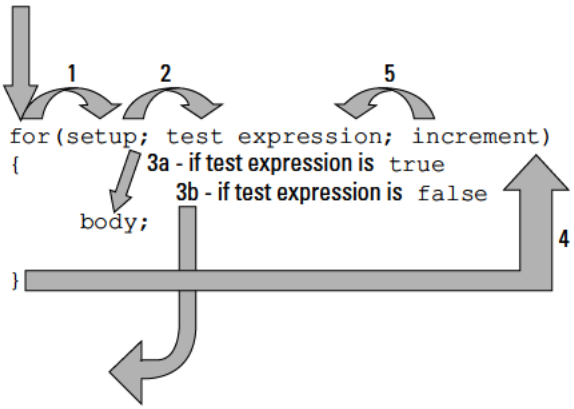


Figure 10-1: The flow in and around the `for` loop.

This `for` loop is completely equivalent to the following `while` loop:

```
setup;
while(test expression)
{
    body;

    increment;
}
```

Looking at an Example

The following example program is the Factorial program written as a `for` loop (this program appears on the enclosed CD-ROM as ForFactorial):

```
//
//  ForFactorial - calculate factorial using the for
//                 construct.
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // enter the number to calculate the factorial of
    int nTarget;
    cout << "This program calculates factorial.\n"
         << "Enter a number to take factorial of: ";
    cin  >> nTarget;

    // start with an accumulator that's initialized to 1
    int nAccumulator = 1;
    for(int nValue = 1; nValue <= nTarget; nValue++)
    {
        cout << nAccumulator << " * "
             << nValue << " equals ";
        nAccumulator = nAccumulator * nValue;
        cout << nAccumulator << endl;
    }

    // display the result
    cout << nTarget << " factorial is "
```

```
            << nAccumulator << endl;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The logic of this ForFactorial program is virtually identical to its older Factorial twin: The program prompts the user to enter a number to take the factorial of. It then initializes `nAccumulator` to 1 before entering the loop that calculates the factorial.

ForFactorial creates an `increment` variable, `nValue`, that it initializes to 1 in the setup clause of the `for` statement. That done, the program compares `nValue` to `nTarget`, the value entered by the user in the `test expression` section of the `for`. If `nValue` is less than or equal to `nTarget`, control enters the body of the loop where `nAccumulator` is multiplied by `nValue`.

That done, control flows back up to the `increment` section of the `for` loop. This expression, `nValue++`, increments `nValue` by 1. Flow then moves to the `test expression`, where `nValue` is compared with `nTarget` and the process repeated until eventually `nValue` exceeds the value of `nTarget`. At that point, control passes to the next statement after the closed brace.

The output from this program appears as follows:

```
This program calculates factorials of user input.
Enter a negative number to exit
Enter number: 5
5 factorial is 120
Enter number: 6
6 factorial is 720
Enter number: -1
Press any key to continue . . .
```

All four sections of the `for` loop are optional. An empty setup, body, or `increment` section has no effect; that is, it does nothing. (That makes sense.) An empty `test expression` is the same as `true`. (This is the only thing that would make sense — if it evaluated to `false`, then the body of the `for` loop would never get executed, and the result would be useless.)

A variable defined within the setup section of a `for` loop is only defined within the `for` loop. It is no longer defined once control exits the loop.

Getting More Done with the Comma Operator

There is a seemingly useless operator that I haven't mentioned (up until now, that is) known as the comma operator. It appears as follows:

```
expression1, expression2;
```

This says execute `expression1` and then execute `expression2`. The resulting value and type of the overall expression is the same as that of `expression2`. Thus, I could say something like the following:

```
int i;
int j;
i = 1, j = 2;
```

Why would I ever want to do such a thing, you ask? Answer: You wouldn't except when writing `for` loops.

The following CommaOperator program demonstrates the comma operator in combat. This program calculates the products of pairs of numbers. If the operator enters N, the program outputs 1 * N, 2 * N-1, 3 * N-2, and so on, all the way up to N * 1. (This program doesn't do anything particularly useful. You'll see the comma operator used to effect when discussing arrays in Chapter 15.)

```
//
//  CommaOperator - demonstrate how the comma operator
//                  is used within a for loop.
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    // enter a target number
    int nTarget;
    cout << "Enter maximum value: ";
    cin  >> nTarget;
```

```
    for(int nLower = 1, nUpper = nTarget;

              nLower <= nTarget; nLower++, nUpper--)
    {
        cout << nLower << " * "
             << nUpper << " equals "
             << nLower * nUpper << endl;
    }

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The program first prompts the operator for a target value, which is read into `nTarget`. It then moves to the `for` loop. However, this time not only do you want to increment a variable from 1 to `nTarget`, you also want to decrement a second variable from `nTarget` down to 1.

Here the setup clause of the `for` loop declares a variable `nLower` that it initializes to 1 and a second variable `nTarget` that gets initialized to `nTarget`. The body of the loop displays `nLower`, `nUpper`, and the product `nLower * nTarget`. The `increment` section increments `nLower` and decrements `nUpper`.

The output from the program appears as follows:

```
Enter maximum value: 15
1 * 15 equals 15
2 * 14 equals 28
3 * 13 equals 39
4 * 12 equals 48
5 * 11 equals 55
6 * 10 equals 60
7 * 9 equals 63
8 * 8 equals 64
9 * 7 equals 63
10 * 6 equals 60
11 * 5 equals 55
12 * 4 equals 48
13 * 3 equals 39
14 * 2 equals 28
15 * 1 equals 15
Press any key to continue . . .
```

In this example run, I entered 15 as the target value. You can see how nLower increments in a straight line from 1 to 15, while nUpper makes its way from 15 down to 1.

Actually, the output from this program is mildly interesting: No matter what you enter, the value of the product increases rapidly at first as nLower increments from 1. Fairly quickly, however, the curve flattens out and asymptotically approaches the maximum value in the middle of the range before heading back down. The maximum value for the product always occurs when nLower and nUpper are equal.

Could I have made the earlier for loop work without using the comma operator? Absolutely. I could have taken either variable, nLower or nUpper, out of the for loop and handled them as separate variables. Consider the following code snippet:

```
nUpper = nTarget;
for(int nLower = 1; nLower <= nTarget; nLower++)
{
    cout << nLower << " * "
         << nUpper << " equals "
         << nLower * nUpper << endl;
    nUpper--;
}
```

This version would have worked just as well.

The for loop can't do anything that a while loop cannot do. In fact, any for loop can be converted into an equivalent while loop. However, because of its compactness, you will see the for loop a lot more often.

Up to and including this chapter, all of the programs have been one monolithic whole stretching from the opening brace after main() to the corresponding closing brace. This is okay for small programs, but it would be really cool if you could divide your program into smaller bites that could be digested separately. That is the goal of the next chapter on functions.

Chapter 11

Functions, I Declare!

In This Chapter

- Breaking programs down into functions
- Writing and using functions
- Returning values from a function
- Passing values to a function
- Providing a function prototype declaration

The programs you see prior to this chapter are small enough and simple enough to write in one sequence of instructions. Sure, there have been branches using `if` statements and looping with `while` and `for` loops, but the entire program was in one place for all to see.

Real-world programs aren't usually that way. Programs that are big enough to deal with the complexities of the real world are generally too large to write in one single block of C++ instructions. Real-world programs are broken into modules called *functions* in C++. This chapter introduces you to the wonderful world of functions.

Breaking Your Problem Down into Functions

Even the Tire Changing Program from Chapter 1 was too big to write in a single block. I only tackled the problem of removing the lug nuts. I didn't even touch the problem of jacking up the car, removing the wheel, getting the spare out, and so on.

In fact, suppose that I were to take the lug nut removing code and put it into a module that I call something fiendishly clever, like RemoveLugNuts(). (I add the parentheses to follow C++ grammar.) I could bundle up similar modules for the other functions.

The resulting top-level module for changing a tire might look like the following:

```
1. Grab spare tire;
2. RaiseCar();
3. RemoveLugNuts();   // we know what this does
4. ReplaceWheel();
5. AttachLugNuts();   // inverse of RemoveLugNuts()
6. LowerCar();
```

Only the first statement is actually an instruction written in Tire Changing Language. Each of the remaining statements is a reference to a module somewhere. These modules consist of sequences of statements written in Tire Changing Language (including possible references to other, simpler modules).

Imagine how this program is executed: The tire changing processor starts at statement 1. First it sees the simple instruction Grab spare tire, which it executes without complaint (it always does exactly what you tell it to do). It then continues on to statement 2.

Statement 2, however, says, "Remember where you're at and go find the set of instructions called RaiseCar(). Once you've finished there, come back here for further instructions." In similar fashion, Statements 3 through 6 also direct the friendly mechanically inclined processor off to separate sets of instructions.

Understanding How Functions Are Useful

There are several reasons for breaking complex problems up into simpler functions. The original reason that a function mechanism was added to early programming languages was the Holy Grail of reuse. The idea was to create functions that could be reused in multiple programs. For example, factorial is a common mathematical procedure. If I rewrote the Factorial program as a function, I could invoke it from any program in the future that needs to calculate a factorial. This form of reuse allows code to be easily reused from different programs as well as from different areas within the same program.

Once a function mechanism was introduced, however, people discovered that breaking up large problems into simpler, smaller problems brought with it further advantages. The biggest advantage has to do with the number of things that a person can think about at one time. This is often referred to as the "Seven Plus or Minus Two" Rule. That's the number of things that a person can keep active in his mind at one time. Almost everyone can keep at

least five objects in their active memory, but very few can keep more than nine objects active in their consciousness at one time.

You will have no doubt noticed that there are a lot of details to worry about when writing C++ code. A C++ module quickly exceeds the nine-object upper limit as it increases in size. Such functions are hard to understand and therefore to write and to get working properly.

It turns out to be much easier to think of the top-level program in terms of high-level functionality, much as I did in the tire changing example at the beginning of this chapter. This example divided the act of changing a tire into six steps, implemented in five functions.

Of course, I still have to implement each of these functions, but these are much smaller problems than the entire problem of changing a tire. For example, when implementing `RaiseCar()`, I don't have to worry about tires or spares, and I certainly don't have to deal with the intricacies of loosening and tightening lug nuts. All I have to think about in that function is how to get the car off the ground.

In computer nerd-speak, we say that these different functions are written at different *levels of abstraction*. The Tire Changing program is written at a very high level of abstraction; the `RemoveLugNuts()` function in Chapter 1 is written at a low level of abstraction.

Writing and Using a Function

Like so many things, functions are best understood by example. The following code snippet shows the simplest possible example of creating and invoking a function:

```
void someFunction()
{
    // do stuff
    return;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // do something

    // now invoke someFunction()
    someFunction();

    // keep going here once control returns
}
```

This example contains all the critical elements necessary to create and invoke a function:

1. **The declaration:** The first thing is the declaration of the function. This appears as the name of the function with a type in front followed by a set of open and closed parentheses. In this case, the name of the function is `someFunction()`, and its return type is `void`. (I'll explain what that last part means in the "Returning things" section of this chapter.)
2. **The definition:** The declaration of the function is followed by the definition of what it does. This is also called the body of the function. The body of a function always starts with an open brace and ends with a closed brace. The statements inside the body are just like those within a loop or an `if` statement.
3. **The return:** The body of the function contains zero or more `return` statements. A `return` returns control to immediately after the point where the function was invoked. Control returns automatically if it ever reaches the final closed brace of the function body.
4. **The call:** A function is called by invoking the name of the function followed by open and closed parentheses.

The flow of control is shown in Figure 11-1.

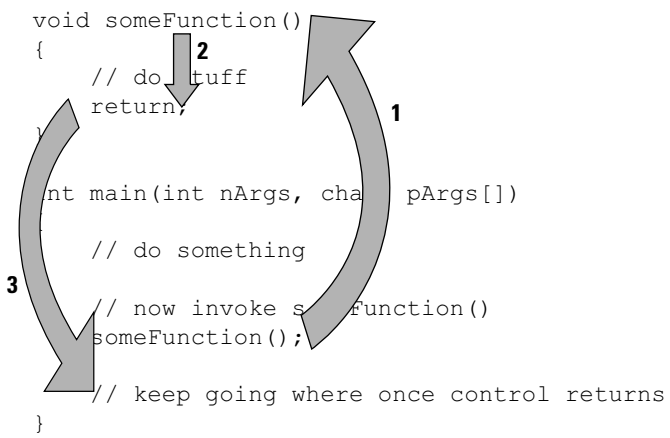


Figure 11-1: Invoking a function passes control to the module. Control returns to immediately after the call.

Returning things

Functions often return a value to the caller. Sometimes this is a calculated value — a function like `factorial()` might return the factorial of a number. Sometimes this value is an indication of how things went — this is usually known as an *error return*. So the function might return a zero if everything went OK, and a non-zero if something went wrong during the execution of the function.

To return a value from a function, you need to make two changes:

1. Replace `void` with the type of value you intend to return.
2. Place the value to return after the keyword `return`. C++ does not allow you to return from a function by running into the final closed brace if the return type is other than `void`.

The keyword `void` is C++-ese for "nothing." Thus a function declared with a return type of `int` returns an integer. A function declared with a return type of `void` returns nothing.

Reviewing an example

The following FunctionDemo program uses the function `sumSequence()` to sum a series of numbers entered by the user at the keyboard. This function is invoked repeatedly until the user enters a zero length sequence.

```
//
//  FunctionDemo - demonstrate how to use a function
//                 to simplify the logic of the program.
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

//
// sumSequence() - return the sum of a series of numbers
//                 entered by the user. Exit the loop
//                 when the user enters a negative
//                 number.
int sumSequence()
{
    // create a variable into which we will add the
    // numbers entered by the user
    int nAccumulator = 0;

    for(;;)
    {
        // read another value from the user
        int nValue;
        cout << "Next: ";
        cin  >> nValue;

        // exit if nValue is negative
        if (nValue < 0)
        {
            break;
        }
```

```
        // add the value entered to the accumulated value
        nAccumulator += nValue;
    }

    // return the accumulated value to the caller
    return nAccumulator;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    cout << "This program sums sequences of numbers.\n"
         << "Enter a series of numbers. Entering a\n"
         << "negative number causes the program to\n"
         << "print the sum and start over with a new\n"
         << "sequence. "
         << "Enter two negatives in a row to end the\n"
         << "program." << endl;

    // stay in a loop getting input from the user
    // until he enters a negative number
    for(;;)
    {
        // accumulate a sequence
        int nSum = sumSequence();

        // if the sum is zero...
        if (nSum == 0)
        {
            // ...then exit the program
            break;
        }

        // display the result
        cout << "Sum = " << nSum << endl;
     }

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

First, concentrate on the `main()` program. After outputting rather verbose instructions to the user, the program enters a `for` loop.

A `for` loop whose conditional expression is empty (as in `for(;;)`) will loop forever unless something within the body of the loop causes control to exit the loop (or until Hell freezes over).

The first non-comment line within this loop is the following:

```
int nSum = sumSequence();
```

This expression passes control to the `sumSequence()` function. Once control returns, the declaration uses the value returned by `sumSequence()` to initialize `nSum`.

The function `sumSequence()` first initializes `nAccumulator` to zero. It then prompts the user for value from the keyboard. If the number entered is not negative, it is added to the value in `nAccumulator`, and the user is prompted for another value in a loop. As soon as the user enters a negative number, the function breaks out of the loop and returns the value accumulated in `nAccumulator` to the caller.

The following is a sample run from the FunctionDemo program:

```
This program sums sequences of numbers.
Enter a series of numbers. Entering a
negative number causes the program to
print the sum and start over with a new
sequence. Enter two negatives in a row to end the
program.
Next: 5
Next: 15
Next: 20
Next: -1
Sum = 40
Next: 1
Next: 2
Next: 3
Next: 4
Next: -1
Sum = 10
Next: -1
Press any key to continue . . .
```

Passing Arguments to Functions

Functions that do nothing but return a value are of limited value because the communication is one-way — from the function to the caller. Two-way communication requires *function arguments*, which I discuss next.

Function with arguments

A function argument is a variable whose value is passed to the function during the call. The following FactorialFunction converts the previous factorial operation into a function:

```
//
//  FactorialFunction - rewrite the factorial code as
//                a separate function.
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

//
// factorial - return the factorial of the argument
//             provided. Returns a 1 for invalid arguments
//             such as negative numbers.
int factorial(int nTarget)
{
    // start with an accumulator that's initialized to 1
    int nAccumulator = 1;
    for (int nValue = 1; nValue <= nTarget; nValue++)
    {
        nAccumulator *= nValue;
    }

    return nAccumulator;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    cout << "This program calculates factorials"
         << " of user input.\n"
         << "Enter a negative number to exit" << endl;

    // stay in a loop getting input from the user
    // until he enters a negative number
    for (;;)
    {
        // enter the number to calculate the factorial of
        int nValue;

        cout << "Enter number: ";
        cin  >> nValue;

        // exit if the number is negative
        if (nValue < 0)
```

```
        {
            break;
        }

        // display the result
        int nFactorial = factorial(nValue);
        cout << nValue << " factorial is "
             << nFactorial << endl;
    }

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The declaration of `factorial()` includes an argument `nTarget` of `int`. Looking ahead, you can see that this is intended to be the value to calculate the factorial of. The return value of the function is the calculated factorial.

In `main()`, the program prompts the user for a value, which it stores in `nValue`. If the value is negative, the program terminates. If not, it calls `factorial()` passing the value of `nValue`. The program stores the returned value in `nFactorial`. It then outputs both values before returning to prompt the user for a new value.

Functions with multiple arguments

A function can have multiple arguments by separating them by commas. Thus, the following function returns the product of two integer arguments:

```
int product(int nValue1, int nValue2)
{
    return nValue1 * nValue2;
}
```

Exposing main()

Now the truth can be told: The "keyword" `main()` from our standard template is nothing more than a function — albeit a function with strange arguments, but a function nonetheless.

Overloading function names

C++ allows the programmer to assign the same name to two or more functions if the functions can be distinguished by either the number or types of arguments. This is called *function overloading*. Consider the following example functions:

```
void someFunction()
{
    // ...perform some function
}
void someFunction(int nValue)
{
    // ...perform some other function
}
void someFunction(char cValue)
{
    // ...perform a function on characters
}

int main(int nNumberofArgs, char* pszArgs[])
{
    someFunction();    // call the first function
    someFunction(10);  // call the second function
    someFunction('a'); // now the third function
    return 0;
}
```

By comparing each of the preceding calls with the declarations, it is clear which function is meant by each call. Because of this, C++ aficionados include the type of arguments with the name of the function in what is called the function's *extended name* or *signature*. Thus, the extended names of the three functions are, in fact, different: `someFunction()`, `someFunction(int)`, and `someFunction(char)`.

Warning: Notice that the return type is not part of the extended name and cannot be used to differentiate functions.

When a program is built, C++ adds some boilerplate code that executes before your program ever gains control. This code sets up the environment in which your program will operate. For example, this boilerplate code opens the default input and output channels and attaches them to `cin` and `cout`.

After the environment has been established, the C++ boilerplate code calls the function `main()`, thereby beginning execution of your code. When your program finishes, it returns from `main()`. This enables the C++ boilerplate to clean up a few things before terminating the program and handing control back over to the operating system.

Defining Function Prototype Declarations

There's a little more to the previous program examples than meets the eye. Consider the second program, FactorialFunction, for example. During the build process, the C++ compiler scanned through the file. As soon as it came upon the `factorial()` function, it made a note in an internal table somewhere in the function's extended name and its return type. This is how the compiler was able to understand what I was talking about when I invoked the `factorial()` function later on in `main()` — it saw that I was trying to call a function, and it said, "Let me look in my table of defined functions for one called `factorial()`. Aha, here's one!"

In this case, the function was defined and the types and number of arguments matched perfectly, but that isn't always the case. What if I had invoked the function not with an integer but with something that could be converted into an integer? Suppose I had called the function as follows:

```
factorial(1.1);
```

That's not a perfect match, 1.1 is not an integer, but C++ knows how to convert 1.1 into an integer. So it *could* make the conversion and use `factorial(int)` to complete the call. The question is, *does* it?

The answer is "Yes." C++ will generate a warning in some cases to let you know what it's doing, but it will generally make the necessary type conversions to the arguments to use the functions that it knows about.

Note: I know that I haven't discussed the different variable types and won't until Chapter 14, but the argument I am making is fairly generic. You will also see in Chapter 14 how to avoid warnings caused by automatic type conversions.

What about a call like the following:

```
factorial(1, 2);
```

There is no conversion that would allow C++ to lop off an argument and use the `factorial(int)` function to satisfy this call, so C++ generates an error in this case.

The only way C++ can sort out this type of thing is if it sees the function declaration before it sees the attempt to invoke the function. This means each function must be declared before it is used.

I know what you're thinking (I think): C++ could be a little less lazy and look ahead for function declarations that occur later on before it gives up and starts generating errors, but the fact is that it doesn't. It's just one of those things, like my crummy car; you learn to live with it.

So does that mean you have to define all of your functions before you can use them? No. C++ allows you to declare a function without a body in what is known as a *prototype declaration*.

A prototype declaration creates an entry for the function in the table I was talking about. It fills in the extended name, including the number and type of the arguments, and the return type. C++ leaves the definition of the function, the function body, empty until later.

In practice, a prototype declaration appears as follows:

```
// the prototype declaration
int factorial(int nTarget);

int main(int nNumberofArgs, char* pszArgs[])
{
    cout << "The factorial of 10 is "
         << factorial(10) << endl;

    return 0;
}

// the definition of the factorial(int) function;
// this satisfies our promise to provide a definition
// for the prototype function declaration above
int factorial(int nTarget)
{
    // start with an accumulator that's initialized to 1
    int nAccumulator = 1;
    for (int nValue = 1; nValue <= nTarget; nValue++)
    {
        nAccumulator *= nValue;
    }

    return nAccumulator;
}
```

The prototype declaration tells the world (or at least that part of the world after the declaration) that `factorial()` takes a single integer argument and returns an integer. That way, C++ can check the call in `main()` against the declaration to see whether any type conversions need to take place or whether the call is even possible.

The prototype declaration also represents a promise to C++ to provide a complete definition of `factorial(int)` somewhere else in the program. In this case, the full definition of `factorial(int)` follows right after `main()`.

It is common practice to provide prototype declarations for all functions defined within a module. That way, you don't have to worry about the order in which they are defined. I'll have more to say about this topic in the next chapter.

Chapter 12

Dividing Programs into Modules

In This Chapter

- Breaking programs down into functions
- Writing and using functions
- Returning values from a function
- Passing values to a function
- Providing a function prototype declaration

In Chapter 11, I show you how to divide a complex problem into a number of separate functions; it is much easier to write and get a number of smaller functions to work than one large, monolithic program. Oftentimes, however, you may want to reuse the functions you create in other applications. For example, I could imagine reusing the `factorial()` function I created in Chapter 11 in the future.

One way to reuse such functions is to copy-and-paste the source code for the `factorial()` function into my new program. However, it would be a lot easier if I could put the function in a separate file that I could then link into future applications. Breaking programs into separate source code modules is the subject of this chapter.

Breaking Programs Apart

The programmer can break a single program into separate source files generally known as *modules*. These modules are compiled into machine code by the C++ compiler separately and then combined during the build process to generate a single program.

The process of combining separately compiled modules into a single program is called *linking*.

Breaking programs into smaller, more manageable pieces has several advantages. First, breaking a program into smaller modules reduces the compile time. Code::Blocks takes only a few seconds to gobble up and digest the programs that appear in this book. Very large programs can take quite a while, however. I have worked on projects that took most of the night to rebuild.

In addition, recompiling all of the source code in the project just because one or two lines change is extremely wasteful. It's much better to recompile just the module containing the change and then relink it into all of the unchanged modules to create a new executable with the change. (The updated module may contain more than just the one changed function but not that many more.)

Second, it's easier to comprehend and, therefore, easier to write and debug a program that consists of a number of well thought out but quasi-independent modules, each of which represents a logical grouping of functions. A large, single source module full of all the functions that a program might use quickly becomes hard to keep straight.

Third is the much vaunted specter of reuse. A module full of reusable functions that can be linked into future programs is easier to document and maintain. A change in the module to fix some bug is quickly incorporated into other executables that use that module.

Finally, there's the issue of working together as a team. Two programmers can't work on the same module (at least not very well). An easier approach is to assign one set of functions contained in one module to a programmer while assigning a different set of functions in a different module to a second programmer. The modules can be linked together when ready for testing.

Breaking Up Isn't That Hard to Do

I can't really include a large program in a book like this . . . well, I could, but there wouldn't be enough left for anything else. I will use the FactorialFunction demo from Chapter 11 as my example large-scale program. In this section, I will create the FactorialModule project that separates the program into several source modules. To do this, I will perform the following steps:

1. Create the FactorialModule project.

   This is no different than creating any of the other project files up to this point in the book.

2. Create the `Factorial.cpp` file to contain the factorial function.
3. Create the `Factorial.h` include file (whatever that is) to be used by all modules that want to call.
4. Update `main.cpp` to use the `factorial()` function.

Creating Factorial.cpp

The initial console application project created by Code::Blocks has only one source file, `main.cpp`. The next step is to create a second source file that will contain the factorial function.

Follow these steps to create `factorial.cpp` containing the `factorial()` function:

1. **Select File➪New➪File.**

   Code::Blocks responds by opening the window shown in Figure 12-1 showing the different types of files you can add.

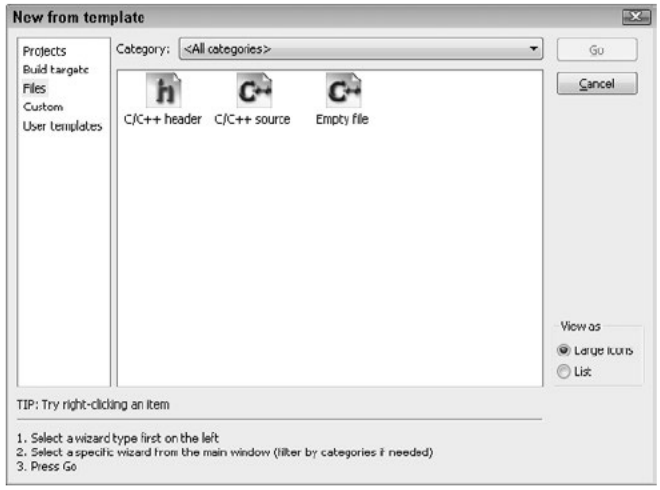


Figure 12-1: The New File wizard provides you help in adding source files to your project.

2. **Select C/C++ Source and then click Go.**

   This opens up a box warning that you are about to enter the mysterious and dangerous Source File Wizard.

3. **Click Next.**

   This will open the Source File Wizard.

4. **Click the ... next to the Filename with Full Path prompt.**

   A File Open dialog box appears, allowing you to navigate to a different folder if you want to keep your source files in different directories. But don't make it any more complicated than it has to be.

5. **Enter `factorial.cpp` as the name of the source file and click Save.**

6. **You want this file added to all executables that you create, so select All for the build targets.**

   When you are finished, the dialog box should look like Figure 12-2.

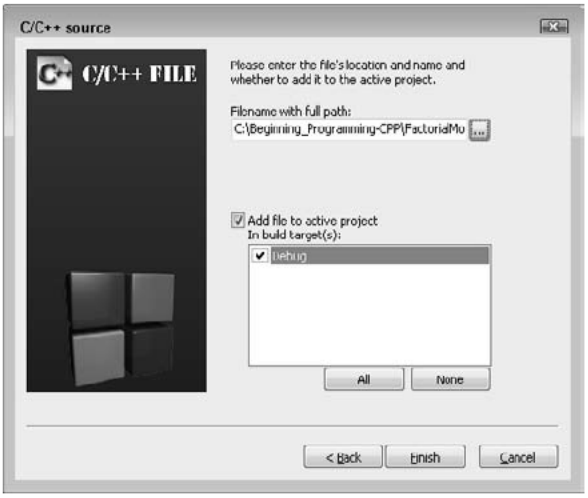


Figure 12-2: The C/C++ Source File dialog box lets you enter the name of the new module, `factorial.cpp`.

7. **Click Finish to create `Factorial.cpp` and add it to the Project.**

   The project file includes the list of all source files that it takes to build your program.

8. **Update `factorial.cpp` as follows:**

```
//
// factorial - this module includes the factorial function
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;
#include "factorial.h"
//
// factorial - return the factorial of the argument
//             provided. Returns a 1 for invalid arguments
//             such as negative numbers.
int factorial(int nTarget)
{
    // start with an accumulator that's initialized to 1
    int nAccumulator = 1;
    for (int nValue = 1; nValue <= nTarget; nValue++)
    {
```

```
        nAccumulator *= nValue;
    }

    return nAccumulator;
}
```

The first four lines are part of the standard template used for all C++ source files in this book. The next line is the `factorial.h` include file, which I discuss further later in this chapter. This is followed by the `factorial()` function much as it appeared in Chapter 11.

Include files don't follow the same grammar rules as C++. For example, unlike other statements in C++, the `#include` must start in column 1 and doesn't require a semicolon at the end.

Don't try to compile `factorial.cpp`, as you haven't created `factorial.h` yet.

Creating an #include file

The next step in the process is to create an include file. Okay, what's an include file?

As I discuss in Chapter 11, the prototype declaration describes the functions to be called by providing the number and types of arguments and the type of the return value. Every function that you invoke must have a prototype declaration somewhere before the call.

It is possible to list out the prototype declarations manually for each function you intend to use, but fortunately that isn't necessary. Instead C++ allows the same dummy who created the function to create an include file that contains the function's prototype declarations. This file can then be included in the source files of the modules where the functions are called.

There are (at least) two ways to include these prototypes. One way is to copy the contents of the include file and paste them into the module where the calls are made. This isn't a very good idea, however. For one thing, it is really laborious. For another, if the prototype declaration for any one of the functions in the include file is changed, the programmer will have to go through every place the include file is used, delete the old one, and repaste in the new file.

Rather than do that, C++ includes a preprocessor that understands very few instructions. Each of these instructions starts with a pound sign (#) in column 1 followed immediately by a command. (Preprocessor commands also end at the end of the line and don't require a semicolon.)

The most common preprocessor command is `#include "filename.h"`. This command copies and pastes the contents of `filename.h` at the point of the `#include` to create what is known as an *intermediate source file.* The preprocessor then passes this intermediate source file on to the C++ compiler for processing. This process is shown graphically in Figure 12-3.

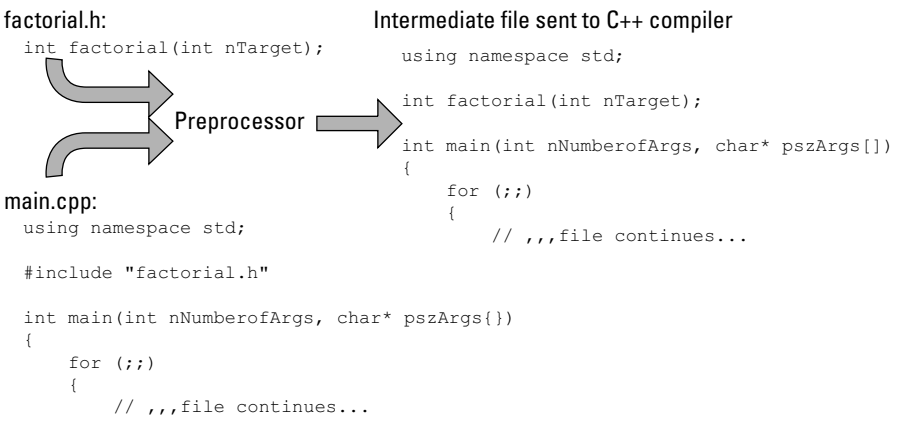


Figure 12-3: The preprocessor inserts the contents of an include file at the point of the `#include` command before passing the results to the C++ compiler.

Including #include files

The Code::Blocks wizard makes creating an include file painless. Just execute the following steps:

1. **Select File➪New➪File.**

   Code::Blocks responds by opening the window shown in Figure 12-1 just as before. This time you're creating an include file.

2. **Select Include File and then click Go.**

3. **In the next window that warns you're about to enter the Include File Wizard, click Next.**

4. **Click the ... next to the Filename with Full Path prompt.**

   A File Open dialog box appears.

5. **Enter `factorial.h` as the name of the include file and click Save.**

6. **You want this file added to all executables that you create, so select All for the build targets.**

   When you are finished, the dialog box should look like Figure 12-4.

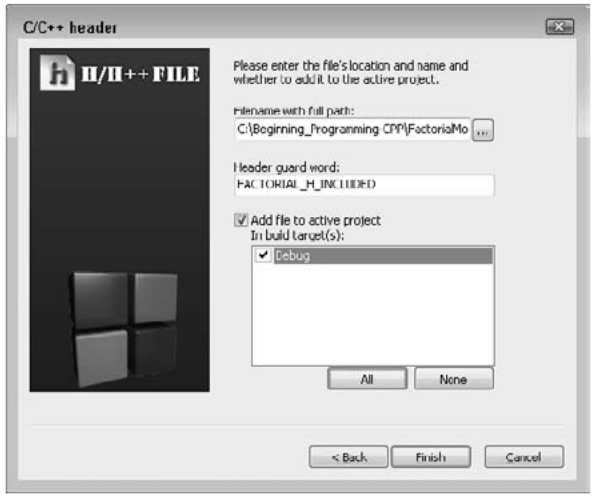


Figure 12-4: The C/C++ Header File dialog box lets you enter the name of the new include file module, `factorial.h`.

7. **Click Finish to create an empty include file that looks like the following:**

```
#ifndef FACTORIAL_H_INCLUDED
#define FACTORIAL_H_INCLUDED



#endif // FACTORIAL_H_INCLUDED
```

8. **Edit the include file by adding the prototype for the `factorial()` function as follows:**

```
#ifndef FACTORIAL_H_INCLUDED
#define FACTORIAL_H_INCLUDED

int factorial(int nTarget);

#endif // FACTORIAL_H_INCLUDED
```

9. **Click File Save.**

   You're done!

Notice that the include file has been added to the project description in the Management tab of Code::Blocks. This indicates that Code::Blocks will automatically rebuild the application if the include file changes.

Why include `factorial.h` in `factorial.cpp`? After all, `factorial()` doesn't require a prototype of itself. You do this as a form of error checking. C++ will generate an error message when compiling `factorial.cpp` if the prototype declaration in `factorial.h` does not match the definition of the function. This ensures that the prototype declaration being used by other source code modules matches the function definition.

Creating main.cpp

You're almost there: Open `main.cpp` and edit it to look like the following:

```
//
//  FactorialModule - rewrite the factorial code as
//                a separate function in its own module.
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

#include "factorial.h"

int main(int nNumberofArgs, char* pszArgs[])
{
    cout << "This program calculates factorials"
         << " of user input.\n"
         << "Enter a negative number to exit" << endl;

    // stay in a loop getting input from the user
    // until he enters a negative number
    for (;;)
    {
        // enter the number to calculate the factorial of
        int nValue;

        cout << "Enter number: ";
        cin  >> nValue;

        // exit if the number is negative
        if (nValue < 0)
        {
            break;
        }

        // display the result
        int nFactorial = factorial(nValue);
        cout << nValue << " factorial is "
             << nFactorial << endl;
    }

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

This version of `main.cpp` looks identical to the FactorialFunction version except that the definition of the `factorial()` function has been removed and the `#include "factorial.h"` added.

Building the result

Now you can build the program (by selecting Build➪Build). Notice in the output messages that the compiler now compiles two files, `main.cpp` and `factorial.cpp`. This is then followed by a single link step.

When executed, the output from this version is indistinguishable from earlier versions as demonstrated by the following test output:

```
This program calculates factorials of user input.
Enter a negative number to exit
Enter number: 5
5 factorial is 120
Enter number: 6
6 factorial is 720
Enter number: -1
Press any key to continue . . .
```

Using the Standard C++ Library

Now you can see why the standard C++ template includes the directives

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
```

These include files contain the prototype declarations for functions provided by C++ as part of its standard library of routines (like `cin >>`, for example).

Notice that the standard C++ library include files are included in angle brackets (<>), while I included my user-defined include file in quotes (""). The only difference between the two is that C++ looks for files contained in quotes starting with the "current" directory (the directory containing the project file), while C++ begins searching for bracketed files in the C++ include file directories.

The online help files (at `www.cppreference.com/wiki/`) are a good source of information about the functions that make up the Standard C++ Library.

Variable Scope

Variables are also assigned a storage type depending upon where and how they are defined, as shown in the following snippet:

```
int nGlobalVariable;
void fn()
{
    int nLocalVariable;
    static int nStaticVariable = 1;

    nStaticVariable = 2;
}
```

Variables defined within a function like `nLocalVariable` don't exist until control passes through the declaration. In addition, `nLocalVariable` is only defined within `fn()` — the variable ceases to exist when control exits the `fn()` function.

By comparison, the variable `nGlobalVariable` is created when the program begins execution and exists as long as the program is running. All functions have access to `nGlobalVariable` all the time.

We say that `nLocalVariable` has *local scope,* and `nGlobalVariable` has *global scope*.

The keyword `static` can be used to create a sort of mishling — something between a global and a local variable. The static variable `nStaticVariable` is created when execution reaches the declaration the first time that function `fn()` is called. Unlike `nLocalVariable`, however, `nStaticVariable` is not destroyed when program execution returns from the function. Instead, it retains its value from one call to the next.

In this example, `nStaticVariable` is initialized to 1 the first time that `fn()` is called. The function changes its value to 2. `nStaticVariable` retains the value 2 on every subsequent call — it is not reinitialized once it has been created. The initialization portion of the declaration is ignored every subsequent time that `fn()` is called after the first time.

However, the scope of `nStaticVariable` is still local to the function. Code outside of `fn()` does not have access to `nStaticVariable`.

Global variables are useful for holding values that you want all functions to have access to. Static variables are most useful for counters — for example, if you want to know how many times a function is called. However, most variables are of the plain ol' local variety.

Chapter 13

Debugging Your Programs, Part 2

In This Chapter

- Debugging a multifunction program
- Performing a unit test
- Using predefined preprocessor commands during debug

This chapter expands upon the debugging techniques introduced in Chapter 8 by showing you how to create debugging functions that allow you to navigate your errors more quickly.

C++ functions represent further opportunities both to excel and to screw up. On the downside are the errors that are possible only when your program is divided into multiple functions. However, dividing your programs into functions allows you to examine, test, and debug each function without regard to how the function is being used in the outside program. This allows you to create a much more solid program.

Debugging a Dys-Functional Program

To demonstrate how dividing a program into functions can make the result easier to read and maintain, I created a version of the SwitchCalculator program in which the calculator operation has been split off as a separate function (which it would have been in the first place if I had only known about functions back then). Unfortunately, I introduced an error during the process that didn't show up until performing testing.

The following listing appears on the enclosed CD-ROM as CalculatorError1:

```
// CalculatorError1 - the SwitchCalculator program
//                    but with a subtle error in it
//
#include <cstdio>
#include <cstdlib>
```

```
#include <iostream>
using namespace std;
// prototype declarations
int calculator(char cOperator, int nOper1, int nOper2);

int main(int nNumberofArgs, char* pszArgs[])
{
    // enter operand1 op operand2
    int  nOper1;
    int  nOper2;
    char cOperator;
    cout << "Enter 'value1 op value2'\n"
         << "where op is +, -, *, / or %:" << endl;
    cin >> nOper1 >> cOperator >> nOper2;

    // echo what the user entered followed by the
    // results of the operation
    cout << nOper1 << " "
         << cOperator << " "
         << nOper2 << " = "
         << calculator(cOperator, nOper1, nOper2)
         << endl;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}

// calculator -return the result of the cOperator
//             operation performed on nOper1 and nOper2
int calculator(char cOperator, int nOper1, int nOper2)
{
    int nResult = 0;
    switch (cOperator)
    {
        case '+':
            nResult = nOper1 + nOper2;
        case '-':
            nResult = nOper1 - nOper2;
            break;
        case '*':
        case 'x':
        case 'X':
            nResult = nOper1 * nOper2;
            break;
        case '/':
            nResult = nOper1 / nOper2;
            break;
```

```
        case '%':
            nResult = nOper1 % nOper2;
            break;
        default:
            // didn't understand the operator
            cout << " is not understood";
    }
    return nResult;
}
```

The beginning of this program starts the same as its SwitchCalculator precursor except for the addition of the prototype declaration for the newly created `calculator()` function. Notice how much cleaner `main()` is here: It prompts the user for input and then echoes the output along with the results from `calculator()`. Very clean.

The `calculator()` function is also simpler than before since all it does is perform the computation specified by `cOperator`. Gone is the irrelevant code that prompts the user for input and displays the results.

All that's left to do is test the results.

Performing unit level testing

Breaking a program down into functions not only allows you to write your program in pieces, but also it allows you to test each function in your program separately. In this functional version of the SwitchCalculator program, I need to test the `calculator()` function by providing all possible inputs (both legal and illegal) to the function.

First, I generate a set of test cases for `calculator()`. Clearly, I need a test for each case in the switch statement. I will also need some boundary conditions, like "how does the function respond when asked to divide by zero?" Table 13-1 outlines some of the cases I need to test.

Table 13-1 Test Cases for calculator() Showing Expected and Actual Results

Operator	*Operand1*	*Operand2*	*Expected Result*	*Actual Result*	*Explanation*
+	10	20	30		Simple case
–	20	10	10		Simple case

(continued)

Table 13-1 ***(continued)***

Operator	*Operand1*	*Operand2*	*Expected Result*	*Actual Result*	*Explanation*
-	10	20	–10		Generate a negative number
*	10	20	200		Simple case
*	10	–5	–50		Try with a negative argument
X	10	20	200		Use the other form of multiply operator
/	20	10	2		Simple case
/	10	0	Don't care as long error generated and program doesn't crash		Try divide by zero
%	23	10	3		Simple case
%	20	10	0		Generate a zero result
%	23	–10	3		Try modulo with a negative number
Y	20	10	Don't care as long as error generated and program doesn't crash		Illegal input

It turns out that I'm lucky in this case — the calling function `main()` allows me to provide any input to the function that I want. I can send each of these test cases to `calculator()` without modifying the program. That isn't usually the case — very often the function is only invoked from the main program in certain ways. In these cases, I must write a special test module that puts the function under test through its paces by passing it the various test cases and recording the results.

Why do you need to write extra debug code? What do you care if the function doesn't handle a case properly if that case never occurs in the program? You care because you don't know how the function will be used in the future. Once written, a function tends to take on a life of its own beyond the program that it was written for. A useful function might be used in dozens of different programs that invoke the function in all sorts of different ways that you may not have thought of when you first wrote the function.

The following shows the results for the first test case:

```
Enter 'value1 op value2'
where op is +, -, *, / or %:
10 + 20
10 + 20 = -10
Press any key to continue . . .
```

Already something seems to be wrong. What now?

Outfitting a function for testing

Like most functions, `calculator()` doesn't perform any I/O of its own. This makes it impossible to know for sure what's going on within the function. I addressed this problem in Chapter 8 by adding output statements in key places within the program. Of course, in Chapter 8, you didn't know about functions, but now you do.

It turns out that it's easier to create an error function that prints out everything you might want to know. You can then just copy and paste calls to this test function in key spots. This is quicker and less error prone than making up a unique output statement for each different location.

C++ provides some help in creating and calling such debug functions. The preprocessor defines several special symbols shown in Table 13-2.

Table 13-2 Predefined Symbols Useful in Creating Debug Functions

Symbol	*Type*	*Value*
`__LINE__`	`int`	The line number within the current source code module
`__FILE__`	`const char*`	The name of the current module
`__DATE__`	`const char*`	The date that the module was compiled (not the current date)
`__TIME__`	`const char*`	The time that the module was compiled (not the current time)
`__FUNCTION__`	`const char*`	The name of the current function (GCC only)
`__PRETTY_FUNCTION__`	`const char*`	The extended name of the current function (GCC only)

You haven't seen the type `const char*`. You will in Chapter 16. You'll have to take my word for now that this is the type of a character string contained in double quotes like `"Stephen Davis is a great guy"`.

You can see how the predefined preprocessor commands from Table 13-2 are used in the following version of the `calculator()` function outfitted with calls to a newly created debugger function `printErr()` (the following code segment is taken from the program CalculatorError2, which is on the enclosed CD-ROM):

```
void printErr(int nLN, char cOperator, int nOp1, int nOp2)
{
    cout << "On line " << nLN
         << ": \'" << cOperator
         << "\' operand 1 = " << nOp1
         << " and operand 2 = " << nOp2
         << endl;
}

// calculator -return the result of the cOperator
//             operation performed on nOper1 and nOper2
int calculator(char cOperator, int nOper1, int nOper2)
{
```

```
    printErr(__LINE__, cOperator, nOper1, nOper2);
    int nResult = 0;
    switch (cOperator)
    {
        case '+':
            printErr(__LINE__, cOperator, nOper1, nOper2);
            nResult = nOper1 + nOper2;
        case '-':
            printErr(__LINE__, cOperator, nOper1, nOper2);
            nResult = nOper1 - nOper2;
            break;
        case '*':
        case 'x':
        case 'X':
            printErr(__LINE__, cOperator, nOper1, nOper2);
            nResult = nOper1 * nOper2;
            break;
        case '/':
            printErr(__LINE__, cOperator, nOper1, nOper2);
            nResult = nOper1 / nOper2;
            break;
        case '%':
            printErr(__LINE__, cOperator, nOper1, nOper2);
            nResult = nOper1 % nOper2;
            break;
        default:
            // didn't understand the operator
            cout << " is not understood";
    }
    return nResult;
}
```

The `printErr()` function displays the value of the operator and the two operands. It also displays the line number that it was called from. The line number is provided by the C++ preprocessor in the form of the `__LINE__` symbol. Printing the line number with the error messages tells me how to differentiate the debug output from the program's normal output.

You can see how this works in practice by examining the output from this newly outfitted version of the program:

```
Enter 'value1 op value2'
where op is +, -, *, / or %:
10 + 20
On line 50: '+' operand 1 = 10 and operand 2 = 20
On line 55: '+' operand 1 = 10 and operand 2 = 20
On line 58: '+' operand 1 = 10 and operand 2 = 20
10 + 20 = -10
Press any key to continue . . .
```

Figure 13-1 shows the display of the program within the CodeBlocks editor including the line numbers along the left side of the display.

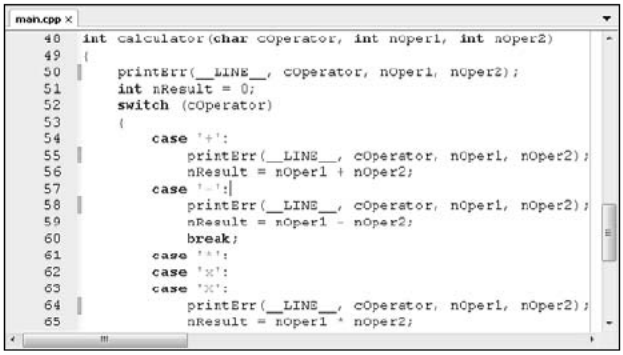


Figure 13-1: The view of the `calculator()` function in the CodeBlocks editor showing the line numbers.

Immediately after I input "10 + 20" followed by the Enter key, the program calls the `printErr()` function from line 50. That's correct since this is the first line of the function. Checking the values, you can see that the input appears to be correct: `cOperator` is '+', `nOper1` is 10, and `nOper2` is 20 just as you expect.

The next call to `printErr()` occurred from line 55, which is the first line of the addition case, again just as expected. The values haven't changed, so everything seems okay.

The next line is completely unexpected. For some reason, `printErr()` is being called from line 58. This is the first line of the subtraction case. For some reason, control is falling through from the addition case directly into the subtraction case.

And then I see it! The break statement is missing at the end of the addition case. The program is calculating the sum correctly but then falling through into the next case and overwriting that value with the difference.

First, I add the missing break statement. I do not remove the calls to `printErr()` — there may be other bugs in the function, and I'll just end up putting them back. There's no point in removing these calls until I am convinced that the function is working properly.

Returning to unit test

The updated program generates the following output for the addition test case:

```
Enter 'value1 op value2'
where op is +, -, *, / or %:
10 + 20
```

```
On line 49: '+' operand 1 = 10 and operand 2 = 20
On line 54: '+' operand 1 = 10 and operand 2 = 20
10 + 20 = 30
Press any key to continue . . .
```

This matches the expected results from Table 13-1. Continuing through the test cases identified in this table, everything matches until I get to the case of `10 / 0` to which I get the output shown in Figure 13-2. The output from the `printErr()` shows that the input is being read properly, but the program crashes soon after line 68.

It's pretty clear that the program is, in fact, dying on line 69 when it performs division by zero. I need to add a test to intercept that case and not perform the division if the value of `nOper2` is zero.

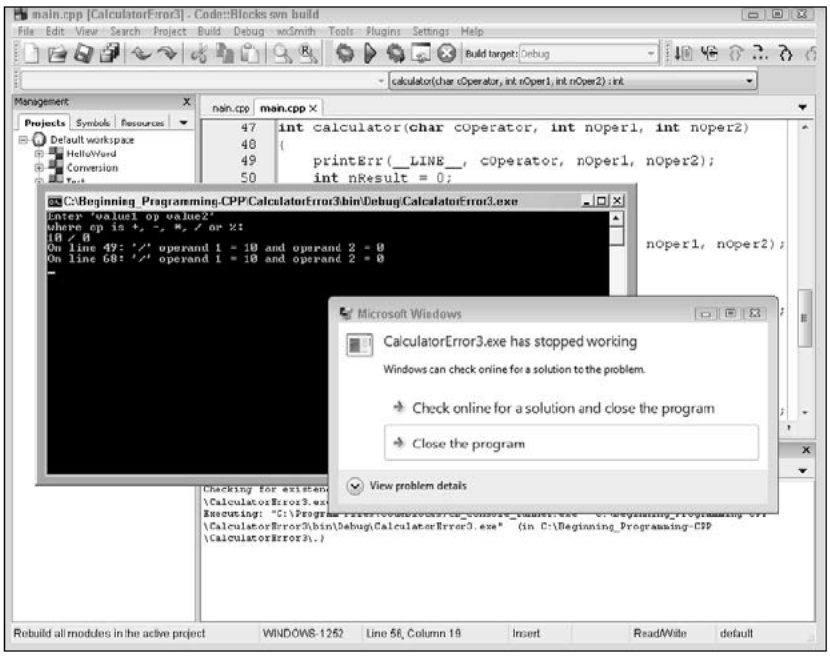


Figure 13-2: The Calculator-Error program terminates with a mysterious error message when I enter '10 / 0'.

Of course, this begs the question: What value should I return from the function if `nOper2` is zero? The "Expected Result" case in Table 13-1 says that we don't care what gets returned when dividing by zero as long as the program doesn't crash. That being the case, I decide to return 0. However, I need to document this case in the comments to the function.

With that addition to the function, I start testing again from the top.

You need to restart back at the beginning of your test cases each time you modify the function.

The function generates the expected results in every case. Now I can remove the `printErr()` functions. The completed `calculator()` function (included in the CalculatorError4 program on the enclosed CD-ROM) appears as follows:

```
// calculator -return the result of the cOperator
//             operation performed on nOper1 and nOper2
//             (In the case of division by zero or if it
//             cannot understand the operator, the
//             function returns a zero.)
int calculator(char cOperator, int nOper1, int nOper2)
{
    int nResult = 0;
    switch (cOperator)
    {
        case '+':
            nResult = nOper1 + nOper2;
            break;
        case '-':
            nResult = nOper1 - nOper2;
            break;
        case '*':
        case 'x':
        case 'X':
            nResult = nOper1 * nOper2;
            break;
        case '/':
            if (nOper2 != 0)
            {
                nResult = nOper1 / nOper2;
            }
            break;
        case '%':
            nResult = nOper1 % nOper2;
            break;
        default:
            // didn't understand the operator
            cout << " is not understood";
    }
    return nResult;
}
```

This version of the `calculator()` function does not suffer from the error that made the original version incapable of adding properly. In addition, this updated version includes a test in the division case: If `nOper2`, the divisor, is zero, the function does not perform a division that would cause the program to crash but leaves the value of `nResult` its initial value of 0.

Part IV
Data Structures

The 5th Wave By Rich Tennant

"This should unstick the keys a little."

In this part . . .

So far you've been limited to just integer and character variables. Fortunately, C++ defines a rich set of variable types, including that most feared of concepts, the C++ pointer. (Don't worry if you don't know what I'm talking about, you will soon.) I wrap up this part with another discussion of debugging.

Chapter 14

Other Numerical Variable Types

In This Chapter

- Reviewing the limitations of integers
- Introducing real numbers to C++
- Examining the limitations of real numbers
- Looking at some variable types in C++
- Overloading function names

The programs so far have limited themselves to variables of type `int` with just a few `char`s thrown in. Integers are great for most calculations — more than 90 percent of all variables in C++ are of type `int`. Unfortunately, int variables aren't adapted to every problem. In this chapter, you will see both variations of the basic `int` as well as other types of intrinsic variables. An *intrinsic type* is one that's built into the language. In Chapter 19, you will see how the programmer can define her own variable types.

Some programming languages allow you to store different types of data in the same variable. These are called *weakly typed* languages. C++, by contrast, is a *strongly typed* language — it requires you to declare the type of data the variable is to store. A variable, once declared, cannot change its type.

The Limitations of Integers in C++

The `int` variable type is the C++ version of an integer. As such, `int` variables suffer the same limitations as their counting integer equivalents in mathematics do.

Integer round-off

It isn't that an integer expression can't result in a fractional value. It's just that an `int` has no way of storing the fractional piece. The processor lops off

the part to the right of the decimal point before storing the result. (This lopping off of the fractional part of a number is called *truncation*.)

Consider the problem of calculating the average of three numbers. Given three `int` variables — `nValue1`, `nValue2`, and `nValue3` — their average is given by the following expression:

```
int nAverage = (nValue1 + nValue2 + nValue3)/3;
```

Suppose that `nValue1` equals 1, `nValue2` equals 2, and `nValue3` equals 2 — the sum of this expression is 5. This means that the average is 5 /3 or either 1 2/3 or 1.666, depending upon your personal preference. But that's not using integer math.

Because all three variables are integers, the sum is assumed to be an integer as well. And because 3 is also an integer, you guessed it, the entire expression is taken to be an integer. Thus, given the same values of 1, 2, and 2, C++ will calculate to the unreasonable but logical result of 1 for the value of `nAverage` (3, 4, and 5 divided by 3 are all 1; 6 divided by 3 is 2).

The problem is much worse in the following mathematically equivalent formulation:

```
int nAverage = nValue1/3 + nValue2/3 + nValue3/3;
```

Plugging in the same values of 1, 2, and 2, the resulting value of `nAverage` is now 0 (talking about logical but unreasonable). To see how this can occur, consider that 1/2 truncates to 0, 2/3 truncates to 0, and 2/3 truncates to 0. The sum of 0, 0, and 0 is (surprise!) 0.

You can see that there are times when integer truncation is completely unacceptable.

Limited range

A second problem with the `int` variable type is its limited range. A normal `int` can store a maximum value of 2,147,483,647 and a minimum value of –2,147,483,648 — that's roughly from positive 2 billion to negative 2 billion for a total range of 4 billion.

That's on a modern PC, Mac, or other common processor. If you have a much older machine, the `int` may not be nearly so expansive in its range. I will have a little more to say about that later in this chapter.

Two billion is a very large number — plenty big enough for most applications. That's why the `int` is useful. But it's not large enough for some applications, including computer technology. In fact, your computer probably executes faster than 2 GHz (gigahertz), depending on how old your computer is (2 GHz is two billion cycles per second). A single strand of fiber cable (the kind that's strung back and forth from one side of the country to the other) can carry way more than 2 billion bits per second. I won't even start on the number of stars in the Milky Way.

A Type That "doubles" as a Real Number

The limitations of the `int` variable are unacceptable in some applications. Fortunately, C++ understands decimal numbers that have a fractional part. (Mathematicians call these *real numbers*.) In C++, decimal numbers are called *floating point numbers* or simply *floats*. This is because the decimal point can float around from left to right to handle fractional values.

Floating point variables come in two basic flavors in C++. The small variety is declared using the keyword `float` as follows:

```
float fValue1;         // declare a floating point
float fValue2 = 1.5;   // initialize it at declaration
```

Oddly enough, the standard floating point variable in C++ is its larger sibling, the double precision floating point or simply `double`. You declare a double precision floating point as follows:

```
double dValue1;
double dValue2 = 1.5;
```

Because the native floating point type for C++ is the `double`, I generally avoid using `float`. The `float` does take up less memory, but this is not an issue for most applications. I will stick with `double` for the remainder of this book. In addition, when I say "floating point variable," you can assume that I'm talking about a variable of type `double`.

Solving the truncation problem

To see how the `double` fixes our truncation problem, consider the average of three floating point variables `dValue1`, `dValue2`, and `dValue3` given by the formula

```
double dAverage = dValue1/3.0 + dValue2/3.0 + dValue3/3.0;
```

Assume, once again, the initial values of 1.0, 2.0, and 2.0. This renders the above expression equivalent to

```
double dAverage = 1.0/3.0 + 2.0/3.0 + 2.0/3.0;
```

which is, in turn, equivalent to

```
double dAverage = 0.333... + 0.6666... + 0.6666...;
```

resulting in a final value of

```
double dAverage = 1.666...;
```

I have written the preceding expressions as though there were an infinite number of sixes after the decimal point. In fact, this isn't the case. The accuracy of a double is limited to about 14 significant digits. The difference between 1.666666666666 and 1 2/3 is small, but not zero. I will have more to say about this a little later in this chapter.

When an integer is not an integer

C++ assumes that a number followed by a decimal point is a floating point constant. Thus, it takes 2.5 to be a floating point. This decimal point rule is true even if the value to the right of the decimal point is zero. Thus, 3.0 is also a floating point. The distinction to you and me between 3 and 3.0 is small, but not to C++.

Actually, you don't have to put anything on the right of the decimal point. Thus 3. is also a `double`. However, it's considered good style to include the 0 after the decimal point for floating point constants.

Computer geeks will be interested to know that the internal representations of 3 and 3.0 are totally different (yawn). More importantly, the constant `int` 3 is subject to `int` rules, whereas 3.0 is subject to the rules of floating point arithmetic.

Thus, you should try to avoid expressions like the following:

```
double dValue = 1.0;
double dOneThird = dValue/3;
```

Technically this is what is known as a *mixed mode* expression because `dValue` is a `double`, while 3 is an `int`. C++ is not a total idiot — it knows what you want in a case like this, so it will convert the 3 to a `double` and perform floating point arithmetic.

We say that C++ *promotes* the `int` 3 to a `double`.

C++ will also allow you to assign a floating point result to an `int` variable:


```
int nValue = dValue / 3.0;
```

Assigning a `double` to an `int` is known as a *demotion*.

Some C++ compilers generate a warning when promoting a variable, but Code::Blocks/gcc does not. All C++ compilers generate a warning (or error) when demoting a result due to the loss of precision.

You should get in the habit of avoiding mixed mode arithmetic. If you have to change the type of an expression, do it explicitly using a caste as in the following example:

```
void fn(int nArg)
{
    // calculate one third of nArg; use a caste to
    // promote it to a floating point
    double dOneThird = (double)nArg / 3.0;

    // ...function continues on
```

I am using the naming convention of starting double precision double variables with the letter d. That is merely a convention. You can name your variables any way you like — C++ doesn't care.

Discovering the limits of double

Floating point variables come with their own limitations. They cannot be used to count things, they take longer to process, they consume more memory, and they also suffer from round-off error (though not nearly as bad as `int`). Now, consider each one of these problems in turn.

Counting

You can't use a floating point variable in an application where counting is important. In C++, you can't say that there are 7.0 characters in my first name. Operators involved in counting don't work on floating point variables. In particular, the auto-increment (++) and auto-decrement (–) operators are strictly verboten on `double`.

Calculation speed

Computers can perform integer arithmetic faster than floating point arithmetic. Historically, this difference was significant. In the 1980s, a CPU without a floating point processor to help it along took about 1,000 times longer to perform a floating point division than it did to perform an integer division.

Fortunately, floating point processors have been built into CPUs for many years now, so the difference in performance is not nearly so significant. I wrote the following loop just as a simple example, first using integer arithmetic:

```
int nValue1 = 1, nValue2 = 2, nValue3 = 2;
for (int i = 0; i < 1000000000; i++)
{
    int nAverage = (nValue1 + nValue2 + nValue3) / 3;
}
```

This loop took about 5 seconds to execute on my laptop. I then executed the same loop in floating point:

```
double dValue1 = 1, dValue2 = 2, dValue3 = 2;
for (int i = 0; i < 1000000000; i++)
{
    double dAverage = (dValue1 + dValue2 + dValue3) / 3.0;
}
```

This look took about 21 seconds to execute on the same laptop. Calculating an average 1 billion times in a little over 20 seconds ain't shabby, but it's still four times slower than its integer equivalent.

Consume more memory

Table 14-2 shows the amount of memory consumed by a single variable of each type. On a PC or Macintosh, an `int` consumes 4 bytes, whereas a `double` takes up 8 bytes. That doesn't sound like much and, in fact, it isn't; but if you had a few million of these things you needed to keep in memory . . . well, it still would be a great number. But if you had a few hundred million, then the difference would be considerable.

This is another way of saying, unless you need to store a heck of a lot of objects, don't worry about the difference in memory taken by one type versus another. Instead, pick the variable type based upon your needs.

If you do just happen to be programming an application that needs to manipulate the age of every human on the planet at one time, then you may want to lean toward the smaller `int` (or one of the other integer types I discuss in this chapter) based upon the amount of memory it consumes.

Loss of accuracy

A double variable has about 16 significant digits of accuracy. Consider that a mathematician would express the number 1/3 as 0.333..., where the ellipses indicate that the threes go on forever. The concept of an infinite series makes sense in mathematics, but not in computing. A computer only has a finite amount of memory and a finite amount of accuracy.

C++ can correct for round-off error in a lot of cases. For example, on output if a variable is 0.99999999999999, C++ will just assume that it's really 1.0 and display it accordingly. However, C++ can't correct for all floating point round-off errors, so you need to be careful. For example, you can't be sure that 1/3 + 1/3 + 1/3 is equal to 1.0:

```
double d1 = 23.0;
double d2 = d1 / 7.0;
if (d1 == (d2 * 7.0))
{
    cout << "Did we get here?" << endl;
}
```

You might think that this code snippet would always display the `"Did we get here?"` string, but surprisingly it does not. The problem is that 23 / 7 cannot be expressed exactly in a floating point number. Something is lost. Thus, `d2 * 7` is very close to 23, but is not exactly equal.

Rather than looking for exact equality between two floating point numbers, you should be asking, "Is d2 * 7 vanishingly close to d1 in value?" You can do that as follows:

```
double d1 = 23.0;
double d2 = d1 / 7.0;

// Is d2 * 7.0 within delta of d1?
double difference = (d2 * 7.0) - d1;
double delta = 0.00001;
if (difference < delta && difference > -delta)
{
    cout << "Did we get here?" << endl;
}
```

This code snippet calculates the difference between `d1` and `d2 * 7.0`. If this difference is less than some small delta, the code calls it a day and says that `d1` and `d2 * 7` are essentially equal.

Not so limited range

The largest number that a double can store is roughly 10 to the 38th power. That's a 1 with 38 zeroes after it; that eats the puny 2 billion maximum size for an `int` for breakfast. That's even more than the national debt (at least, at the time of this writing). I'm almost embarrassed to call this a limit, but I suppose there are applications where 38 zeroes aren't enough.

Remember that only the first 16 digits are significant. The remaining 22 digits are noise having already succumbed to standard floating point round-off error.

Variable Size — the "long" and "short" of It

C++ allows you to expand on integer variable types by adding the following descriptors on the front: `const`, `unsigned`, `short`, or else `long`. Thus, you could declare something like the following:

```
unsigned long int ulnVariable;
```

A `const` variable cannot be modified. All numbers are implicitly `const`. Thus, 3 is of type `const int`, while 3.0 is a `const double`, and '3' is a `const char`.

An `unsigned` variable can take on non-negative values only; however, it can handle a number roughly twice as large as its signed sibling. Thus, an `unsigned int` has a range of 0 to 4 billion (as opposed to the regular `signed int`'s range of –2 billion to 2 billion).

C++ allows you to declare a `short int` and a `long int`. For example, a `short int` takes less space but has a more limited range than a regular `int`, whereas a `long int` takes more storage and has a significantly larger range.

The `int` is assumed. Thus, the following two declarations are both accepted and completely equivalent:

```
long int lnVar1;  // declare a long int
long lnVar2;      // also a long int; int is assumed
```

The C++ 2009 Standard even defines a `long long int` and a `long double`. The Code::Blocks/gcc that comes on the enclosed CD-ROM understands what these are, but not all compilers do. These are just like `long int` and `double`, respectively, only more so — more accuracy and larger range.

Not all combinations are allowed. For example, `unsigned` can be applied only to the counting types `int` and `char`. Table 14-1 shows the legal combinations and their meaning along with how to declare a constant of that type.

Table 14-1 The Common C++ Variable Types

Type	*Declaring a Constant*	*What It Is*
`int`	1	A simple counting number, either positive or negative.
`unsigned int`	1U	A non-negative counting number.

Type	Declaring a Constant	What It Is
`short int`	---	A potentially smaller version of the int. It uses less memory but has a more limited range.
`long int`	1L	A potentially larger version of the `int`. It may use more memory but has a larger range. There is no difference between `long` and `int` on the Code::Blocks/gcc compiler.
`long long int`	1LL	A potentially even larger version of the `int`.
`float`	1.0F	A single precision real number.
`double`	1.0	A double precision real number.
`long double`	---	A potentially larger floating point number. On the PC, `long double` is the native size for numbers internally to the numeric processor.
`char`	'c'	A single char variable stores a single character. Not suitable for arithmetic.
`wchar_t`	L'c'	A wide character. Used to store larger character sets such as Chinese kanji symbols. Also known as UTF or Unicode.

How far do numbers range?

It may seem odd, but the C++ standard doesn't actually say exactly how big a number each of the data types can accommodate. The standard addresses only the relative size of each variable type. For example, it says that the maximum `long int` is at least as large as the maximum `int`.

The authors of C++ weren't trying to be mysterious. They wanted to allow the compiler to implement the absolute fastest code possible for the base machine. The standard was designed to work for all different types of processors running different operating systems.

In fact, the standard size of an `int` has changed over the past decades. Before 2000, the standard `int` on most PCs was 2 bytes and had a range of plus or minus 64,000. Some time around 2000, the basic word size on the Intel processors changed to 32 bits. Most compilers changed to the default `int` of today that's 4 bytes and has a range of plus or minus 2 billion.

Table 14-2 provides the size and range of each variable type on the Code::Blocks/gcc compiler provided on the enclosed CD-ROM (and most other compilers meant for an Intel processor running on a 32-bit operating system).

Table 14-2	Range of Numeric Types in Code::Blocks/gcc		
Type	*Size [bytes]*	*Accuracy*	*Range*
`short int`	2	exact	–32,768 to 32,767
`int`	4	exact	–2,147,483,648 to 2,147,483,647
`long int`	4	exact	–2,147,483,648 to 2,147,483,647
`long long int`	8	exact	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	4	7 digits	$+/- 3.4028 * 10^{+/-38}$
`double`	8	16 digits	$+/- 1.7977 * 10^{+/-308}$
`long double`	12	19 digits	$+/-1.1897 * 10^{+/-4932}$

Attempting to calculate a number that is beyond the range of a variable's type is known as an *overflow*. The C++ standard generally leaves the results of an overflow undefined. That's another way that the inventors of C++ wanted to leave the language flexible so that the machine code generated would be as fast as possible.

On the PC, a floating point overflow generates an exception that, if not handled, will cause your program to crash. (I don't discuss exception handling until Chapter 32.) As bad as that sounds, an integer overflow is even worse — C++ generates an incorrect result without complaint.

Types of Constants

I mentioned the `const` declaration earlier in this chapter and again in Table 14-1, but I would like to take a minute to expand upon constants now.

A *constant value* is an explicit number or character such as 1 or 0.5 or 'c'. Constant values cannot be changed, that is, they cannot appear on the left-hand side of an assignment statement. Every constant value has a type. The type of 1 is `const int`. The type of 0.5 is `const double`. Table 14-1 explains how to declare constant values with different types. For example, 1L is of type `const long`.

A variable can be declared constant using the `const` keyword:

```
const double PI = 3.14159;  // declare a constant variable
```

A const variable must be initialized when it is declared since you will not get another chance in the future — just like a constant value, a const variable cannot appear on the left-hand side of an assignment statement.

It is common practice to declare const variables using all capitals. Multiple words within a variable name are divided by an underscore as in TWO_PI. As always, this is just convention — C++ doesn't care.

It may seem odd to declare a variable and then say that it can't be changed. Why bother? Largely because a carefully named constant can make a program a lot easier to understand. Consider the following two equivalent expressions:

```
double dC = 6.28318 * dR;  // what does this mean?
double dCircumference = TWO_PI * dRadius; // this is a lot
                              // easier to understand
```

It should be a lot clearer to the reader of this code that the second expression is multiplying the radius by 2π to calculate the circumference.

Passing Different Types to Functions

Floating point variables and variables of different size are passed to function in the same way that int variables are as demonstrated in the following code snippet. This example snippet passes the value of the variable dArg along with the const double 0.0 to the function maximumFloat().

```
// maximumFloat - return the larger of two floating
//                point arguments
double maximumFloat(double d1, double d2)
{
    if (d1 > d2)
    {
        return d1;
    }
    return d2;
}

void otherFunction()
{
    double dArg = 1.0;
    double dNonNegative = maximumFloat(dArg, 0.0);
    // ...function continues...
```

I discuss functions in Chapter 11.

Overloading function names

The type of the arguments are part of the extended name of the function. Thus, the full name of the earlier example function is `maximumFloat (double, double)`. In Chapter 11, you see how to differentiate between two functions by the number of arguments. You can also differentiate between two functions by the type of the arguments, as shown in the following example:

```
double maximum(double d1, double2);
int    maximum(int n1, int n2);
```

When declared this way, it's clear that the call `maximum(1, 2)` refers to `maximum(int, int)`, while the call `maximum(3.0, 4.0)` refers to `maximum(double, double)`.

Defining functions that have the same name but different arguments is called *function overloading*.

Sometimes the programmer's intentions start to get a little obscure, but you can even differentiate by the signedness and length as well:

```
int maximum(int n1, int n2);
long maximum(long l1, long l2);
unsigned maximum(unsigned un1, unsigned un2);
```

Fortunately, this is rarely necessary in practice.

Mixed mode overloading

The rules can get really weird when the arguments don't line up exactly. Consider the following example code snippet:

```
double maximum(double d1, double d2);
int    maximum(int n1, int n2);

void otherFunction()
{
    // which function is invoked by the following?
    double dNonNegative = maximum(dArg, 0);
    // ...function continues...
```

const arguments are a constant problem

Since C++ passes the value of the argument, you cannot differentiate by `const`-ness. Consider the following call to see why:

```
double maximum(double d1, double d2);

void otherFunction()
{
    double dArg = 2.0;
    double dNonNegative = maximum(dArg, 0.0);
```

What actually gets passed to `maximum()` are the values 2.0 and 0.0. The `maximum()` function can't tell whether these values came from a variable like `dArg` or a constant like 0.0.

You *can* declare the arguments of a function to be `const`. Such a declaration means that you cannot change the argument's value within the function. This is demonstrated in the following implementation of `maximum(double, double)`:

```
double maximum(const double d1, const double d2)
{
    double dResult = d1;
    if (d2 > dResult)
    {
        dResult = d2;
    }

    // the following would be illegal
    d1 = 0.0; d2 = 0.0

    return dResult;
}
```

The assignment to `d1` and `d2` is not allowed because both have been declared `const` and therefore are not changeable.

What is not legal is the following:

```
// these two functions are not different enough to be
   distinguished
double maximum(double d1, double d2);
double maximum(const double d1, const double d2);

void otherFunction()
{
    double dArg = 2.0;

    // C++ doesn't know which one of the above functions to call
    double dNonNegative = maximum(dArg, 0.0);
```

C++ has no way of differentiating between the two when you make the call. I have more to say about `const` arguments in Chapter 17.

Here, the arguments don't line up exactly with either declaration. There is no `maximum(double, int)`. C++ could reasonably take any one of the following three options:

- Promote the 0 to a `double` and call `maximum(double, double)`.
- Demote the double to an `int` and invoke `maximum(int, int)`.
- Throw up its electronic hands and report a compiler error.

The general rule is that C++ will promote arguments in order to find a match but will not automatically demote an argument. However, you can't always count on this rule. In this case, Code::Blocks generates an error that the call is ambiguous. That is, the third option wins.

My advice is to not rely on C++ to figure out what you mean by making the necessary conversions explicit:

```
void otherFunction(int nArg1, double dArg2)
{
    // use an explicit cast to make sure that the
    // proper function is called
    double dNonNegative = maximum((double)nArg1, dArg2);
```

Now it is clear that I mean to call `maximum(double, double)`.

Chapter 15

Arrays

In This Chapter

- Expanding simple variables into an array
- Comparing the array to a rental car lot
- Indexing into an array
- Initializing an array

The variables declared so far have been of different types with different sizes and capabilities. Even so, each variable has been capable of holding only a single value at a time. If I wanted to hold three numbers, I had to declare three different variables. The problem is that there are times when I want to hold a set of numbers that are somehow closely related. Storing them in variables with names that bear some similarity of spelling like `nArg1`, `nArg2`, and so on may create associations in my mind but not for poor, ignorant C++.

There is another class of variable known as the *array* that can hold a series of values. Arrays are the subject of this chapter and the next chapter. (Here I present arrays in general. In the next chapter, I look at the particular case of the character array.)

What Is an Array?

If you are mathematically inclined and were introduced to the concept of the array in high school or college, you may want to skim this section.

You may think of a variable as a truck. There are small trucks, like a `short int`, capable of holding only a small value; and there are larger trucks, like a `long double`, capable of holding astoundingly large numbers. However, each of these trucks can hold only a single value.

Each truck has a unique designator. Perhaps you give your vehicles names, but even if you don't, each has a license plate that uniquely describes each of your vehicles, at least within a given state.

This works fine for a single family. Even the largest families don't have so many cars that this arrangement gets confusing. But think about an auto rental agency. What if they referred to their cars solely by a license plate number or some other ID? (Boy, just thinking about that Hertz!)

After filling out the myriad forms — including deciding whether I want the full insurance coverage and whether I'm too lazy to fill it up with gas before I bring it back — the guy behind the counter says, "Your car is QZX123." Upon leaving the building and walking to the parking lot, I look over a sea of cars that rival a Wal-Mart parking lot. Exactly where is QZX123?

That's why the guy behind the counter actually says something quite different. He says something to the effect, "Your car is in slot B11." This means that I am to skip past row A directly to row B and then start scanning down the line for the eleventh car from the end. The numbers are generally painted on the pavement to help me out, but even if they weren't, I could probably figure out which car he meant.

Several things have to be true in order for this system to work:

- The slots have to be numbered in order (B2 follows B1 and comes immediately before B3), ideally with no breaks or jumps in the sequence.
- Each slot is designed to hold a car (a given parking slot may be empty, but the point is that I would never find a house in a parking slot).
- The slots are equally spaced (being equally spaced means that I can jump ahead and guess about where B50 is without walking along from B1 through B49, checking each one).

That's pretty much the way arrays work. I can give a series of numbers a single name. I refer to individual numbers within the series by index. So the variable *x* may refer to a whole series of whole numbers, x(1) would be the first number in the series, x(2) the second, and so on, just like the cars at the rental agency.

Declaring an Array

To declare an array in C++, you must provide the name, type, and number of elements in the array. The syntax is as follows:

```
int nScores[100];
```

This declares an array of 100 integers and gives them the name `nScores`.

It is common practice to use the same naming convention for arrays as for non-arrays but to use the plural form. That makes sense because `nScores` refers to 100 integer values.

Indexing into an Array

You must provide an index to access a specific element within the array. An index must be a counting type (like `int`), as demonstrated here:

```
nScores[11] = 10;
```

This is akin to the way that rental cars are numbered. However, unlike humans, C++ numbers its arrays starting with 0. Thus, the first score in the array `nScores` is `nScores[0]`.

So how does this work exactly? I will return to the rental car lot one more time (for the last time, I promise). Figure 15-1 shows how rental cars typically number their parking lots. The first car in row B carries the designation B1. To find B11, I simply move my gaze ten cars to the right.

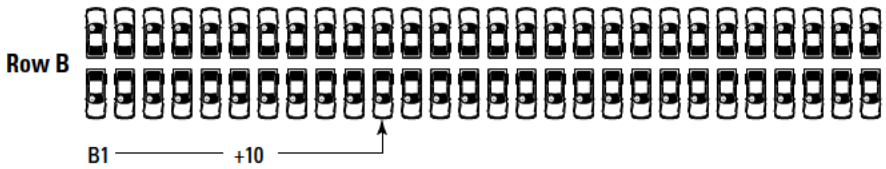


Figure 15-1: Cars in a rental car lot are typically numbered sequentially starting with 1 to make them easier to find.

C++ does a similar thing. To execute the statement `nScores[11] = 10`, C++ starts with the address of the first element in `nScores`. It then moves to the right 11 spaces and stores a 10 at that location. This is shown graphically in Figure 15-2. (I say a lot more about what it means to "take the address of the first element" in the next three chapters. Please just accept the statement for now.)

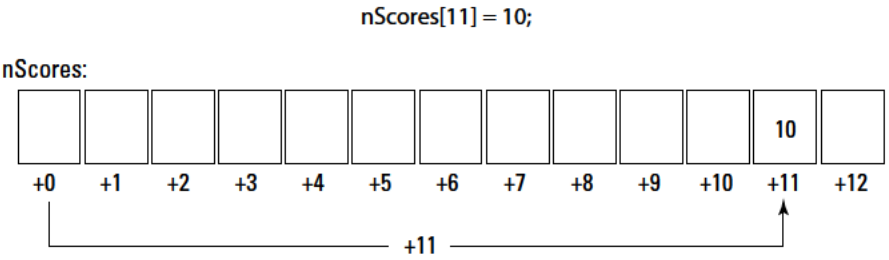


Figure 15-2: C++ calculates the location of `nScores[11]` by moving over 11 `int` slots from the beginning of the `nScores` array.

The fact that C++ starts counting at zero leads to a point that always confuses beginners. The statement

```
int nScores[100];
```

declares 100 scores, which are numbered from 0 to 99. The expression

```
nScores[100] = 0;  // this is an error
```

zeroes out the first element *beyond* the end of the array. The last element in the array is `nScores[99]`. The C++ compiler will not catch this error and will happily access this non-element, which very often leads to the program accessing some other variable by mistake. This type of error is very hard to find because the results are so unpredictable.

Looking at an Example

The following example averages a set of scores and then displays that average. However, unlike earlier demonstrations, this program retains the scores' input in an array that it can then output along with the average.

```
//
//  ArrayDemo - demonstrate the use of an array
//              to accumulate a sequence of numbers
//
#include <cstdio>
#include <cstdlib>
```

```
#include <iostream>

using namespace std;

// displayArray - displays the contents of the array
//                of values of length nCount
void displayArray(int nValues[100], int nCount)
{
    for(int i = 0; i < nCount; i++)
    {
        cout.width(3);
        cout << i << " - " << nValues[i] << endl;
    }
}

// averageArray - averages the contents of an array
//                of values of length nCount
int averageArray(int nValues[100], int nCount)
{
    int nSum = 0;
    for(int i = 0; i < nCount; i++)
    {
        nSum += nValues[i];
    }
    return nSum / nCount;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    int nScores[100];
    int nCount;

    // prompt the user for input
    cout << "This program averages a set of scores\n"
         << "Enter scores to average\n"
         << "(enter a negative value to terminate input"
         << endl;
    for(nCount = 0; nCount < 100; nCount++)
    {
        cout << "Next: ";
        cin >> nScores[nCount];
        if (nScores[nCount] < 0)
        {
            break;
        }
    }

    // now output the results
    cout << "Input terminated." << endl;
    cout << "Input data:" << endl;
```

```
    displayArray(nScores, nCount);
    cout << "The average is "
         << averageArray(nScores, nCount)
         << endl;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

This program starts at the beginning of `main()` by prompting the user for a series of integer values. The program saves each of the numbers that the user inputs into the array `nScores` in a loop. The program exits the loop as soon as the user enters a negative number.

Notice that this program keeps track of the number of values entered in the variable `nCount`. The program will exit the loop after 100 entries whether or not the user enters a negative number — because that's all the room the program has for storing values. You should always make sure that you don't overrun the end of an array.

Once the user has either entered a negative value or 100 values in a row, the program exits the loop. Now the `nScores` array contains all of the numbers entered, and `nCount` contains a count of the number of values that are stored in the array.

The program then calls the function `displayArray()` to echo to the user the values entered. Finally, the function `averageArray()` returns the integer average of the numbers entered.

The `displayAverage()` function iterates through the values in the array passed it, displaying each value in turn. The `averageArray()` function works by also iterating through the array `nValues`, accumulating the sum of each element in a local variable `nSum`. The function returns `nSum / nCount`, which is the average of the values in `nValues`.

In practice, the output of the program appears as follows:

```
This program averages a set of scores
Enter scores to average
(enter a negative value to terminate input
Next: 10
Next: 20
Next: 30
Next: 40
Next: 50
```

```
Next: -1
Input terminated.
Input data:
  0 - 10
  1 - 20
  2 - 30
  3 - 40
  4 - 50
The average is 30
Press any key to continue . . .
```

Initializing an Array

Like any other variable, an array starts out with an indeterminate value if you don't initialize it. The only difference is that unlike a simple variable, which contains only one undetermined value, an array starts out with a whole lot of unknown values:

```
int nScores[100];  // none of the values in nScores
                   // known until you initialize them
```

You can initialize the elements of an array with a loop as follows:

```
int nScores[100];  // declare the array and then...
for (int i = 0; i < 100; i++) // ...initialize it
{
    nScores[i] = 0;
}
```

You can also initialize an array when you declare it by including the initial values in braces after the declaration. For a small array, this is easy:

```
int nCount[5] = {0, 1, 2, 3, 4};
```

Here I initialized the value of `nCount[0]` to 0, `nCount[1]` to 1, `nCount[2]` to 2, and so on. If there are more elements than numbers in the list, C++ pads the list with zeros. Thus, in the following case:

```
int nCount[5] = {1};
```

the first element (`nCount[0]`) is set to 1. Every other element gets initialized to zero. You can use this to initialize a large array to zero as well:

```
int nScores[100] = {0};
```

This not only declares the array but initializes every element in the array to zero.

By the same token, you don't have to provide an array size if you have an initializer list — C++ will just count the number of elements in the list and make the array that size:

```
int nCount[] = {1, 2, 3, 4, 5};
```

This declares `nCount` to be 5 elements large because that's how many values there are in the initializer list.

Arrays are useful for holding small to moderate amounts of data. (Really large amounts of data require a database of some sort.) By far, the most common type of array is the character array, which is the subject of the next chapter.

Chapter 16

Arrays with Character

In This Chapter

- Introducing the `null` terminated character array
- Creating an ASCIIZ array variable
- Examining two example ASCIIZ manipulation programs
- Reviewing some of the most common built-in ASCIIZ library functions

Chapter 15 introduced the concept of arrays. The example program collected values into an integer array, which was then passed to a function to display and a separate function to average. However, as useful as an array of integers might be, far and away the most common type of array is the character array. Specifically something known as the *ASCIIZ character array,* which is the subject of this chapter.

The ASCII-Zero Character Array

Arrays have an inherent problem: You can never know by just looking at the array how many values are actually stored in it. Knowing the size of an array is not enough. That tells you how many values the array *can* hold, not how many it actually *does* hold. The difference is like the difference between how much gas your car's tank can hold and how much gas it actually has. Even if your tank holds 20 gallons, you still need a gas gauge to tell you how much is in it.

For a specific example, the ArrayDemo program in Chapter 15 allocated enough room in `nScores` for 100 integers, but that doesn't mean the user actually entered that many. He might have entered a lot fewer.

There are essentially two ways of keeping track of the amount of data in an array:

- **Keep a count of the number of values in a separate `int` variable.** This is the technique used by the ArrayDemo program. The code that read the user input kept track of the number of entries in `nCount`. The only problem is that the program had to pass `nCount` along to every function to which it passed the `nScores` array. The array was not useful without knowing how many values it stored.
- **Use a special value in the array as an indicator of the last element used.** By convention, this is the technique used for character arrays in C++.

Look back at the table of legal ASCII characters in Chapter 5. You'll notice that one character in particular is not a legal character: '\0'. This character is also known as the `null` character. It is the character with a numerical value of zero. A program can use the `null` character as the end of a string of characters since it can never be entered by the user. This means that you don't have to pass a separate count variable around — you can always tell the end of the string by looking for a `null`.

The designers of C and C++ liked this feature so well that they settled on it as the standard for character strings. They even gave it a name: the *ASCII-zero* array or *ASCIIZ* for short.

The `null` character has another advantageous property. It is the only character whose value is considered `false` in a comparison expression (such as in a loop or an `if` statement).

Remember from Chapter 9 that 0 or `null` is considered `false`. All other values evaluate to `true`.

This makes writing loops that manipulate ASCIIZ strings even easier, as you will see in the following examples.

Declaring and Initializing an ASCIIZ Array

I could declare an ASCIIZ character array containing my first name as follows:

```
char szMyName[8] = {'S', 't', 'e', 'p',
                    'h', 'e', 'n', '\0'};
```

Actually, the 8 is redundant. C++ is smart enough to count the number of characters in the initialization string and just make the array that big. Thus, the following is completely equivalent to the previous example:

```
char szMyName[] = {'S', 't', 'e', 'p',
                   'h', 'e', 'n', '\0'};
```

The only problem with this is that it's awfully clumsy. I have to type a lot more than just the seven characters that make up my name. (I had to type about five keystrokes for every character in my name — that's a lot of overhead.) ASCIIZ strings are so common in C++ that the language provides a shorthand option:

```
char szMyName[] = "Stephen";
```

These two initialization statements are completely equivalent. In fact, a string contained in double quotes is nothing more than an array of constant characters terminated with a `null`.

The string `"Stephen"` consists of eight characters — don't forget to count the terminating `null`.

Looking at an Example

Let's take the simple case of displaying a string. You know by now that C++ understands how to display ASCIIZ strings just fine, but suppose it didn't. What would a function designed to display a string look like? The following DisplayASCIIZ program shows one example:

```
//
//  DisplayASCIIZ - display an ASCIIZ string one character
//                  at a time as an example of ASCIIZ
//                  manipulation
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;


// displayString - display an ASCIIZ string one character
//                 at a time
void displayString(char szString[])
{
```

```
    for(int index = 0; szString[index] != '\0'; index++)
    {
        cout << szString[index];
    }
}

int main(int nNumberofArgs, char* pszArgs[])
{
    char szName1[] = {'S', 't', 'e', 'p',
                      'h', 'e', 'n', '\0'};
    char szName2[] = "Stephen";

    cout << "Output szName1: ";
    displayString(szName1);
    cout << endl;

    cout << "Output szName2: ";
    displayString(szName2);
    cout << endl;


    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The `displayString()` function is the key to this demonstration program. This function iterates through the array of characters passed to it using the variable index. However, rather than rely on a separate variable containing the number of characters in the array, this function loops until the character at `szString[index]` is the `null` character, `'\0'`. As long as the current character is not a `null` character, the loop outputs it to the display.

The `main()` function creates two versions of my name, first using discrete characters for `szName1` and then a second time using the shortcut `"Stephen"` for `szName2`. The function then displays both strings using the `displayString()` function both to show that the function works and to demonstrate the equivalence of the two strings.

The output from the program appears as follows:

```
Output szName1: Stephen
Output szName2: Stephen
Press any key to continue . . .
```

Notice that `szName1` and `szName2` display identically (since they are the same).

Constant character problems

Technically `"Stephen"` is not of type `char[]`, that is, "array of characters" — it's of type `const char[]`, that is "array of const characters." The difference is that you cannot modify the characters in an array of constant characters. Thus, you could do the following:

```
char cT = "Stephen"[1];   // fetch the second character, the 't'
```

But you could not modify it by putting it on the left-hand side of an equal sign:

```
"Stephen"[1] = 'x';       // replace the 't' with an 'x'
```

This pickiness about `const` doesn't normally make a difference, but it can cause C++ consternation when declaring arguments to a function. For example, in the DisplayASCIIZ demo program, I could not say `displayString("Stephen")` because `displayString()` is declared to accept an array of characters (`char[]`), where `"Stephen"` is an array of `const` characters (`const char[]`).

I can solve this problem by simply declaring `displayString()` as follows:

```
void displayString(const char szString[]);
```

The function works because `displayString()` never tries to modify the `szString` array passed to it.

Don't worry if this discussion of `const` versus non-`const` variables leaves you confused — you'll get another chance to see this in action in Chapter 18.

Looking at a More Detailed Example

Displaying a string of characters is fairly simple. What about a little bit tougher example? The following program concatenates two strings that it reads from the keyboard.

To concatenate two strings means to tack one onto the end of the other. For example, the result of concatenating `"abc"` with `"DEF"` is `"abcDEF"`.

Before you examine the program, think about how you could go about concatenating a string, call it `szSource`, onto the end of another one called `szTarget`.

First, you need to find the end of the `szTarget` string (see the top of Figure 16-1). Once you've done that, you copy characters from `szSource` one at a time into `szTarget` until you reach the end of the `szSource` string (as demonstrated at the bottom of Figure 16-1). Make sure that the result has a final `null` on the end, and you're done.

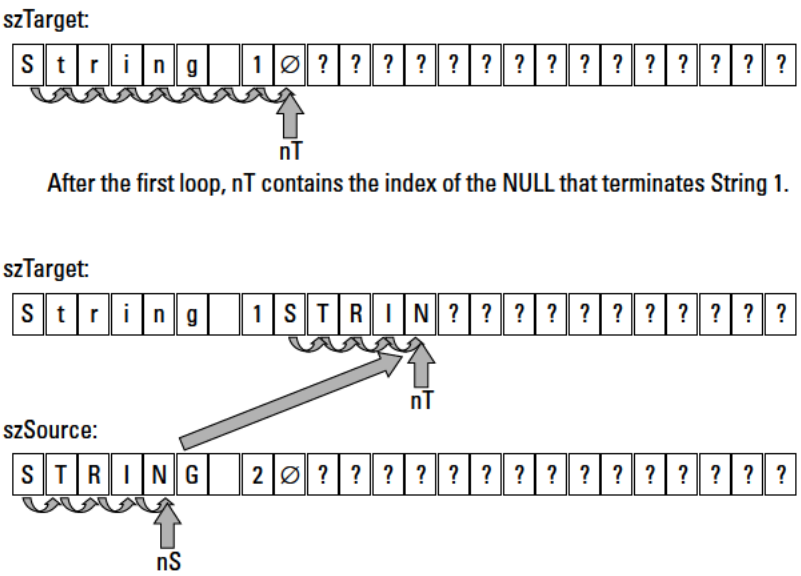


Figure 16-1: To concatenate, the function must do the following: (a) First, find the terminating `null` of the target string; (b) Then copy characters from the source to the target until the terminating `null` on the source is encountered.

That's exactly how the `concatenateString()` function works in the ConcatenateString example program.

```
//
//  ConcatenateString - demonstrate the manipulation of
//                ASCIIZ strings by implementing a
//                concatenate function
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;


// concatenateString - concatenate one string onto the
//                     end of another
void concatenateString(char szTarget[],
                  const char szSource[])
{
    // first find the end of the target string
    int nT;
    for(nT = 0; szTarget[nT] != '\0'; nT++)
    {
    }
```

```
    // now copy the contents of the source string into
    // the target string, beginning at 'nT'
    for(int nS = 0; szSource[nS] != '\0'; nT++, nS++)
    {
        szTarget[nT] = szSource[nS];
    }

    // add the terminator to szTarget
    szTarget[nT] = '\0';
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // Prompt user
    cout << "This program accepts two strings\n"
         << "from the keyboard and outputs them\n"
         << "concatenated together.\n" << endl;

    // input two strings
    cout << "Enter first string: ";
    char szString1[256];
    cin.getline(szString1, 256);

    cout << "Enter the second string: ";
    char szString2[256];
    cin.getline(szString2, 256);

    // now concatenate one onto the end of the other
    cout << "Concatenate first string onto the second"
         << endl;
    concatenateString(szString1, szString2);

    // and display the result
    cout << "Result: <"
         << szString1
         << ">" << endl;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The `concatenateString()` function accepts two strings, `szTarget` and `szSource`. Its goal is to tack `szSource` onto the end of `szTarget`.

The function assumes that the `szTarget` array is large enough to hold both strings tacked together. It has no way of checking to make sure that there is enough room. More on that a little later in this chapter

Notice that the target argument is passed first and the source second. This may seem backwards, but it really doesn't matter — either argument can be the source or the target. It's just a C++ convention that the target goes first.

In the first `for` loop, the function iterates through `szTarget` by incrementing the index `nT` until `szTarget[nT] == '\0'`, that is, until `nT` points to the terminating `null` character. This corresponds to the situation at the top of Figure 16-1.

The function then enters a second loop in which it copies each character from `szSource` into `szTarget` starting at `nT` and moving forward. This corresponds to the bottom of Figure 16-1.

This example shows a situation when using the comma operator in a `for` loop is justified.

Since the `for` loop terminates before it copies the terminating `null` from `szSource`, the function must add the terminating `null` onto the result before returning.

The `main()` program prompts the user to enter two strings, each terminated with a newline. The program then concatenates the two strings by calling the new `concatenateString()` function and displays the results.

The expression `cin >> string;` stops inputting at the first white space. The `getline()` function used in the example program reads input from the keyboard just like `cin >> string;`, but it reads an entire line up to the newline at the end. It does not include the newline in the character string that it returns. Don't worry about the strange syntax of the call to `getline()` — I cover that in Chapter 23.

The results of a sample run of the program appear as follows:

```
This program accepts two strings
from the keyboard and outputs them
concatenated together.

Enter first string: String 1
Enter the second string: STRING 2
Concatenate first string onto the second
Result: <String 1STRING 2>
Press any key to continue . . .
```

Note that the second argument to `concatenateString()` is actually declared to be a `const char[]` (pronounced "array of constant characters"). That's because the function does not modify the source string. Declaring it to be an array of constant characters allows you to call the function passing it a constant string as in the following call:

```
concatenateString(szString, "The End");
```

Foiling hackers

How does the `concatenateString()` function in the earlier example program know whether there is enough room in `szTarget` to hold both the source and target strings concatenated together? The answer is that it doesn't.

This is a serious bug. If a user entered enough characters before pressing Enter, he could overwrite large sections of data or even code. In fact, this type of fixed buffer overwrite bug is one of the ways that hackers gain control of PCs through a browser to plant virus code.

In the following corrected version, `concatenateString()` accepts an additional argument: the size of the `szTarget` array. The function checks the index `nT` against this number to make sure that it does not write beyond the end of the target array.

The program appears as ConcatenateNString on the enclosed CD-ROM:

```
// concatenateString - concatenate one string onto the
//                     end of another (don't write beyond
//                     nTargetSize)
void concatenateString(char szTarget[],
                       int  nTargetSize,
                  const char szSource[])
{
    // first find the end of the target string
    int nT;
    for(nT = 0; szTarget[nT] != '\0'; nT++)
    {
    }

    // now copy the contents of the source string into
    // the target string, beginning at 'nT' but don't
    // write beyond the nTargetSize'th element (- 1 to
    // leave room for the terminating null)
    for(int nS = 0;
           nT < (nTargetSize - 1) && szSource[nS] != '\0';
              nT++, nS++)
    {
        szTarget[nT] = szSource[nS];
    }

    // add the terminator to szTarget
    szTarget[nT] = '\0';
}
```

The first part of the function starts out exactly the same, incrementing through `szTarget` looking for the terminating `null`. The difference is in the second loop. This `for` loop includes two terminating conditions. Control exits the loop if either of the following is true:

- `szSource[nS]` is the `null` character, meaning that you've gotten to the final character in `szSource`.
- `nT` is greater than or equal to `nTargetSize - 1` meaning that you've exhausted the space available in `szTarget` (`- 1` because you have to leave room for the terminating `null` at the end).

This extra check is irritating but necessary to avoid overrunning the array and producing a program that can crash in strange and mysterious ways.

Do I Really Have to Do All That Work?

C++ doesn't provide much help with manipulating strings in the language itself. Fortunately, the standard library includes a number of functions for manipulating these strings that save you the trouble of writing them yourself. Table 16-1 shows the most common of these functions.

Table 16-1 Common ASCIIZ String Manipulation Functions

Function	*Description*
`isalpha(char c)`	Returns a `true` if the character is alphabetic ('A' through 'Z' or 'a' through 'z').
`isdigit(char c)`	Returns a `true` if the character is a digit ('0' through '9').
`isupper(char c)`	Returns a `true` if the character is an uppercase alphabetic.
`islower(char c)`	Returns a `true` if the character is a lowercase alphabetic.
`isprint(char c)`	Returns a `true` if the character is printable.
`isspace(char c)`	Returns a `true` if the character is a form of white space (space, tab, newline, and so on).
`strlen(char s[])`	Returns the number of characters in a string (not including the terminating `null`).
`strcmp(char s1[], char s2[])`	Compares two strings. Returns 0 if the strings are identical. Returns a 1 if the first string occurs later in the dictionary than the second. Returns a –1 otherwise.
`strncpy(char target[], char source[], int size)`	Copies the source string into the target string but not more than 'size' characters.

Function	Description
`strncat(char target[], char source[], int size)`	Concatenates the source string onto the end of the target string for a total of not more than 'size' characters.
`tolower(char c)`	Returns the lowercase version of the character passed to it. Returns the current character if it is already lowercase or has no uppercase equivalent (such as a digit).
`toupper(char c)`	Returns the uppercase version of the character passed to it.

The following example program uses the `toupper()` function to convert a string entered by the user into all caps:

```
//
//  ToUpper - convert a string input by the user to all
//            upper case.
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;


// toUpper - convert every character in an ASCIIZ string
//           to uppercase
void toUpper(char szTarget[], int  nTargetSize)
{
    for(int nT = 0;
           nT < (nTargetSize - 1) && szTarget[nT] != '\0';
              nT++)
    {
        szTarget[nT] = toupper(szTarget[nT]);
    }
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // Prompt user
    cout << "This program accepts a string\n"
         << "from the keyboard and echoes the\n"
         << "string in all caps.\n" << endl;

    // input two strings
    cout << "Enter string: ";
    char szString[256];
    cin.getline(szString, 256);
```

```
    // now convert the string to all uppercase
    toUpper(szString, 256);

    // and display the result
    cout << "All caps version: <"
         << szString
         << ">" << endl;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The `toUpper()` function follows a pattern that will quickly become old hat for you: It loops through each element in the ASCIIZ string using a `for` loop. The loop terminates if either the size of the string is exhausted or the program reaches the terminating `null` character.

The function passes each character in the string to the standard C++ library `toupper()` function. It stores the character returned by the function back into the character array.

It is not necessary to first test to make sure that the character is lowercase using `islower()` — both the `tolower()` and the `toupper()` functions return the character passed to them if the character has no lower- or uppercase equivalent.

The `main()` function simply prompts the user to enter a string. The program reads the input string by calling `getline()`. It then converts whatever it reads to uppercase by calling `toUpper()` and then displays the results.

The following shows the results of a sample run:

```
This program accepts a string
from the keyboard and echoes the
string in all caps.

Enter string: This is a string 123!@#.
All caps version: <THIS IS A STRING 123!@#.>
Press any key to continue . . .
```

Notice that the input string includes uppercase characters, lowercase characters, digits, and symbols. The lowercase characters are converted to uppercase in the output string, but the uppercase characters, digits, and symbols are unchanged.

In this chapter, you've seen how to handle ASCIIZ strings as a special case of character arrays. In practice, many of the standard functions rely on something known as a *pointer*. In the next two chapters, you'll see how pointers work. I will then return to these same example functions and implement them using pointers to demonstrate the elegance of the pointer solution.

Chapter 17

Pointing the Way to C++ Pointers

In This Chapter

- Introducing the concept of pointer variables
- Declaring and initializing a pointer
- Using pointers to pass arguments by reference
- Allocating variable-sized arrays from the heap

This chapter introduces the powerful concept of *pointers*. By that I don't mean specially trained dogs that point at birds but rather variables that point at other variables in memory. I start with an explanation of computer addressing before getting into the details of declaring and using pointer variables. This chapter wraps up with a discussion of something known as the heap and how we can use it to solve a problem that I slyly introduced in the last chapter.

But don't think the fun is over when this chapter ends. The next chapter takes the concept of pointers one step further. In fact, in one way or another, pointers will reappear in almost every remaining chapter of this book.

It may take you a while before you get comfortable with the concept of pointer variables. Don't get discouraged. You may have to read through this chapter and the next a few times before you grasp all of the subtleties.

What's a Pointer?

A *pointer* is a variable that contains the address of another variable in the computer's internal memory. Before you can get a handle on that statement, you need to understand how computers address memory.

The details of computer addressing on the Intel processor in your PC or Macintosh are quite complicated and much more involved than you need to worry about in this book. I will use a very simple memory model in these discussions.

Every piece of random access memory (RAM) has its own, unique address. For most computers, including Macintoshes and PCs, the smallest addressable piece of memory is a byte.

A byte is 8 bits and corresponds to a variable of type `char`.

An address in memory is exactly like an address of a house, or would be if the following conditions were true:

- Every house is numbered in order.
- There are no skipped or duplicated numbers.
- The entire city consists of one long street.

So, for example, the address of a particular byte of memory might be 0x1000. The next byte after that would have an address of 0x1001. The byte before would be at 0x0FFF.

I don't know why, but, by convention, memory addresses are always expressed in hexadecimal. Maybe it's so that non-programmers will think that computer addressing is really complicated.

Declaring a Pointer

A `char` variable is designed to hold an ASCII character, an `int` an integer number, and a `double` a floating point number. Similarly, a pointer variable is designed to hold a memory address. You declare a pointer variable by adding an asterisk (`*`) to the end of the type of the object that the pointer points at, as in the following example:

```
char* pChar;    // pointer to a character
int*  pInt;     // pointer to an int
```

A pointer variable that has not otherwise been initialized contains an unknown value. You can initialize a pointer variable with the address of a variable of the same type using the ampersand (`&`) operator:

```
char cSomeChar = 'a';
char* pChar;
pChar = &cSomeChar;
```

In this snippet, the variable `cSomeChar` has some address. For argument's sake, let's say that C++ assigned it the address 0x1000. (C++ also initialized that location with the character `'a'`.) The variable `pChar` also has a location of

its own, perhaps 0x1004. The value of the expression `&cSomeChar` is 0x1000, and its type is `char*` (read "pointer to char"). So the assignment on the third line of the snippet example stores the value 0x1000 in the variable `pChar`. This is shown graphically in Figure 17-1.

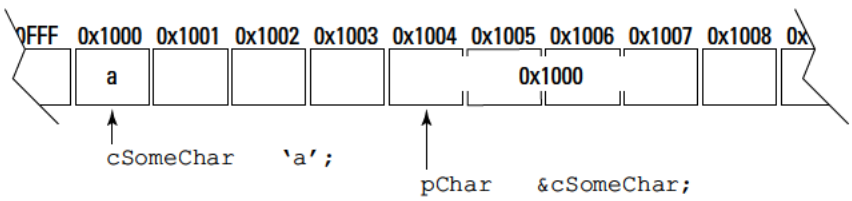


Figure 17-1: The layout of `cSomeChar` and `pChar` in memory after their declaration and initialization, as described in the text.

Take a minute to really understand the relationship between the figure and the three lines of C++ code in the snippet. The first declaration says, "go out and find a 1-byte location in memory, assign it the name `cSomeChar`, and initialize it to `'a'`." In this example, C++ picked the location 0x1000.

The next line says, "go out and find a location large enough to hold the address of a `char` variable and assign it the name `pChar`." In this example, C++ assigned `pChar` to the location 0x1004.

In Code::Blocks, all addresses are 4 bytes in length irrespective of the size of the object being pointed at — a pointer to a `char` is the same size as a pointer to a `double`. The real world is similar — the address of a house looks the same no matter how large the house is.

The third line says, "assign the address of `cSomeChar` (0x1000) to the variable `pChar`." Figure 17-1 represents the state of the program after these three statements.

"So what?" you say. Here comes the really cool part demonstrated in the following expression:

```
*pChar = 'b';
```

This line says, "store a `'b'` at the `char` location pointed at by `pChar`." This is demonstrated in Figure 17-2. To execute this expression, C++ first retrieves the value stored in `pChar` (that would be 0x1000). It then stores the character `'b'` at that location.

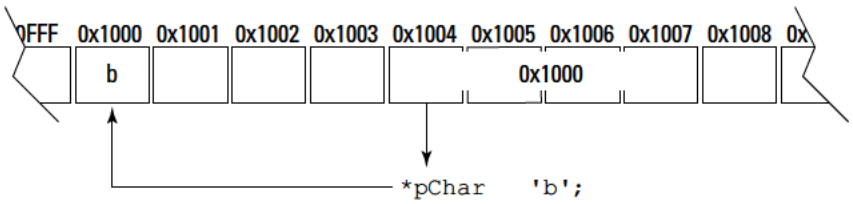


Figure 17-2: The steps involved in executing `*pChar = 'b'`.

The `*` when used as a binary operator means "multiply"; when used as a unary operator, `*` means "find the thing pointed at by." Similarly `&` has a meaning as a binary operator (though I didn't discuss it), but as a unary operator, it means "take the address of."

So what's so exciting about that? After all, I could achieve the same effect by simply assigning a `'b'` to `cSomeChar` directly:

```
cSomeChar = 'b';
```

Why go through the intermediate step of retrieving its address in memory? Because there are several problems that can be solved only with pointers. I discuss two common ones in this chapter. I'll describe a number of problems that are most easily solved with pointers in subsequent chapters.

Passing Arguments to a Function

There are two ways to pass arguments to a function: either by value or by reference. Now, consider both in turn.

Passing arguments by value

In Chapter 11, I write that arguments are passed to functions by value, meaning that it is the value of the variable that gets passed to the function and not the variable itself.

The implications of this become clear in the following snippet (taken from the PassByReference example program on the enclosed CD-ROM):

```
void fn(int nArg1, int nArg2)
{
    // modify the value of the arguments
    nArg1 = 10;
    nArg2 = 20;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // initialize two variables and display their values
    int nValue1 = 1;
    int nValue2 = 2;

    // now try to modify them by calling a function
    fn(nValue1, nValue2);

    // what is the value of nValue1 and nValue2 now?
    cout << "nValue1 = " << nValue1 << endl;
    cout << "nValue2 = " << nValue2 << endl;

    return 0;
}
```

This program declares two variables, `nValue1` and `nValue2`, initializes them to some known value, and then passes their value to a function `fn()`. This function changes the value of its arguments and simply returns.

> **Question:** What is the value of `nValue1` and `nValue2` in `main()` after the control returns from `fn()`?
>
> **Answer:** The value of `nValue1` and `nValue2` remain unchanged at 1 and 2, respectively.

To understand why, examine carefully how C++ handles memory in the call to `fn()`. C++ stores local variables (like `nValue1` and `nValue2`) in a special area of memory known as the *stack*. Upon entry into the function, C++ figures out how much stack memory the function will require and then reserves that amount. Say, for argument's sake, that in this example, the stack memory carved out for `main()` starts at location 0x1000 and extends to 0x101F. In this case, `nValue1` might be at location 0x1000 and `nValue2` at location 0x1004.

An `int` takes up 4 bytes in Code::Blocks. See Chapter 14 for details.

As part of making the call to `fn()`, C++ first stores the values of each argument on the stack starting at the rightmost argument and working its way to the left.

The last thing that C++ stores as part of making the call is the return address so that the function knows where to return to after it is complete.

For reasons that have more to do with the internal workings of the CPU, the stack "grows downward," meaning that the memory used by `fn()` will have addresses smaller than 0x1000. Figure 17-3 shows the state of memory at the point that the computer processor reaches the first statement in `fn()`. C++ stored the second argument to the function at location 0x0FF4 and the first argument at 0x0FF0.

Remember that this is just a possible layout of memory. I don't know (or care) that any of these are in fact the actual addresses used by C++ in this or any other function call.

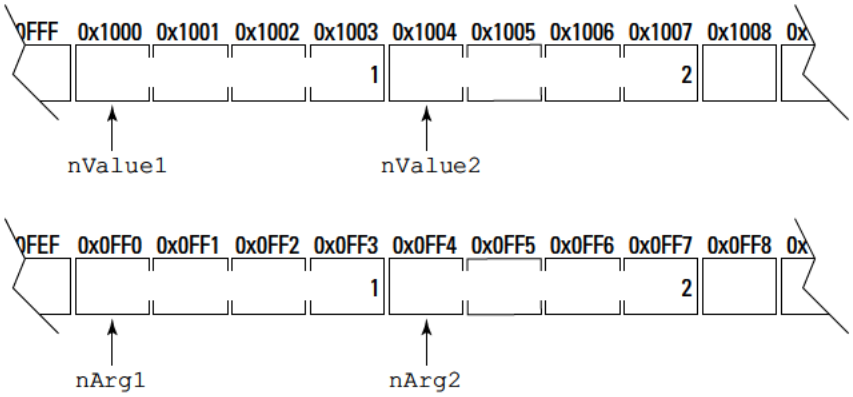


Figure 17-3: A possible layout of memory immediately after entering the function `fn(int, int)`.

Layout in memory immediately after making the call:
`fn(nValue1, nValue2)`

The function `fn(int, int)` contains two statements:

```
nArg1 = 10;
nArg2 = 20;
```