Chapter 4: Flow Control

Notes

- Four types of scope: function, file, block, and function-prototype. (See 5.12 of C How To Program)
- Boolean constants: true and false.
- Nested loops

Introduction

In this chapter you will have your first introduction to splitting things up and changing the flow. It sounds real groovy and it can be. So hold on tight as I take you through the wild, wild wavy course of blocks and flow control. This intermittently requires the appearance of such terms as variable scope which will be covered with minty freshness.

Statement Blocks

Once upon a time I started writing a tutorial on loops. I discovered early on that I unwittingly introduced a new thing and did not explain it. You were actually introduced to it with the first program you wrote. I am speaking of *statement blocks*. This is, as its name suggests, a "block" of statements or multiple statements. The start and end of a statement block are defined by the opening and closing curly braces. Sound familiar? Let's look at our 'Hello World' program once more:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      cout << "Hello World" << endl;
06      return 0;
07  }
```

In this program there is only one statement block; it is the *body* of '`main()`' that begins with line 4 and ends with line 6. The body of a function, such as '`main()`', is the function's root statement block. A statement block can actually contain *no* or more statements. Yes, this means it is possible to have an *empty* statement block. The following source has an empty statement block:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      return 0;
```

```
06  }
```

The only reason we need the first line (‘`#include <iostream.h>`’) there is for ‘`cin`’ and ‘`cout`’. Also it is possible to omit both the ‘`int`’ before ‘`main()`’ and the ‘`return 0;`’ before the closing curly brace. This will likely make the compiler complain, but it will still work. Bearing these things in mind we could shorten this program even further:

```
01  main()
02  {
03  }
```

This is the smallest program you can write in C++ and even C (well, except that we could still put everything on one line and scrunch it all together). It does absolutely nothing. We’ll come back to this later because it’s interesting that even though it does nothing, it still has a fair file size.

A statement block then is a container for statements or no statements at all. It is a container for a single list of logic. It’s like the paper that directions are written on. The instructions in a statement block can be decisively executed, avoided, or executed multiple times when used with certain language constructs for *flow control*. But, also importantly, variables in a statement block are limited to a specific *scope* unless otherwise specified.

Scope

Variable lifetime is not the only thing which limits where a variable can be used. A variable’s identifier also has something known as scope. Scope is the *visibility of an identifier*, not a minty mouthwash. An identifier can only be used validly when it is *in scope* or “can be seen”. Imagine that everything within the curly braces of ‘main()’ is in a room. You can only use things within that room. Thus the things declared in that room have a scope that is *limited to that room*.

If a variable is a person then you cannot smack that person unless they are in the room. A statement block is *like* a room. Things declared within it can be used, but things declared outside of it *may* not. Scope is related to a variable’s identifier not necessarily the variable itself. Usually when a variable goes *out of scope* (i.e. it can no longer “be seen”) its life also ends.

Nesting

When something is in another room, you must go to that room to use it. Such is the same with variables in statement blocks. All variables have a scope, which is in what places they are usable in. All of the variables we’ve worked with so far have been declared in the body of ‘`main()`’. And so we’ve been able to use them without much thought to

scope because all of the logic of using them is with them in '`main()`'. What I need to do to really explain scope is give some examples. Before I can do that you must learn about *nesting*.

Nesting is containing a statement block within another statement block. The inner statement block is known as the *nested* statement block. It has access to all of the outer block's variables, but the outer block does not have access to its variables. First an example of nesting:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      cout << "In outer block" << endl;
06      {
07          cout << "In inner block" << endl;
08      }
09      return 0;
10  }
```

If we were to declare variables inside the outer block, the body of main, they would be accessible from the inner block:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int outer = 1;
06      cout << "In outer block" << endl;
07      {
08          cout << "'outer' from inner is " << x " << endl;
09      }
10      return 0;
11  }
```

However, a variable declared inside the nested, or inner, block is *not* available to the outer block:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int outer = 1;
06      cout << "In outer block" << endl;
07      {
08          int inner = 2;
09          cout << "'outer' from inner is " << x " << endl;
10      }
11      cout << "won't compile " << inner << endl;
12      return 0;
13  }
```

This program won't compile because the variable '`inner`' is only usable from within the nested statement block. Thus the scope of the variable '`inner`' in the inner block is limited to the inner block only. It would be available to any blocks nested within it, but since there are none; the scope of '`inner`' is limited to a single statement block.

There are two real reasons why the variable 'inner' cannot be used once its statement block ends. The first is that 'inner' *goes out of scope*. In other words the visibility of 'inner' is limited to the statement block it was declared in, and so when it ends it is no longer visible. The second reason is because it gets destroyed. A variable declared within a statement block is usually destroyed when it goes out of scope.

Think of scope like a cat in a tree that will only go up. A statement block can only *see* and *use* variables that are declared in the current block or an outer one. Likewise a cat can only get to branches that are above it. If there is a mouse on a branch above the cat, it can get it. However, if the mouse is below the cat, or even on the ground (god forbid), it is inaccessible. A statement block cannot use variables within one of its nested blocks because they are "below currently visibility/accessibility" (like the mice being out of reach of the cat).

<cat on tree analogy picture>

A statement block that is *not* nested within another is known as a *disjointed statement block*. Thus the curly braces for 'main()' represent a disjointed statement block.

Locals

Variables declared within a block are known as *local variable* or *locals*. A constant can usually exist in the same context of a variable, and does in this case, so the same rules apply to constants. However, the term "*local constants*" is explicitly used, rather than "locals" (which implies a variable).

Locals can only be used within the block they are declared, as you have seen. It is possible for two variables to have the same name as long as they have different scope:

```
{
    int x = 0;
}
{
    int x = 0;
}
```

This is legal because the scope of the first 'x' is limited to the first statement block and the second to the second. Think of them as two guys with the same name, but locals of different towns: "Billy Bob of Austin and Billy Bob of Onalaska".

Globals

Variables that are declared outside of any block are known as *global variables* or *globals*. These are variables whose scope is not limited to a single block, but the *entire* C++ program, thus the word global. Their lifetime is tied to the entire program. A global variable is created *before* 'main' is executed and destroyed *after* it is finished. Thus a global variable lives for the entire course of a program. Like most variables a global is destroyed when it goes out of scope; but its scope does not end until the program is finished. Thus a global's identifier has global scope (visibility) and global lifetime.

To create a global variable, you simply declare it outside of any block. Obviously, it should also be *above* where you intend to use it or the compiler will flip out. The following creates a global variable and uses it:

```
01  #include <iostream.h>
02
03  int myglobal;
04
05  int main()
06  {
07      myglobal = 5;
08      cout << "In outer block (global = " << myglobal
09           << endl;
10      {
11          cout << "In inner block (global = " << myglobal
12               << endl;
13      }
14      return 0;
15  }
```

Author's Preference: Use globals very *sparingly*. I have found that it is possible to write *any* C++ program without using them at all. Using a lot of global variables is known as "*globals abound*".

You may think that scope and lifetime are inevitably tied together, but later on you will find out how to create variables with a global lifetime and local scope: thus they exist for the entire execution of the program, but can only be used within the statement block they are declared.

Namespaces

A namespace is a space in which names, or identifiers, are contained. No two identifiers may be the same in a single namespace. When two or more identifiers have the same name in the same namespace it is known as a *name conflict*. When there is a great, perhaps largely unnecessary, amount of identifiers in a namespace it is known as *namespace clutter*. Namespace clutter is usually considered in the context of the global namespace.

There are two namespaces that you have dealt with thus far: global and local. The global namespace is the space in which all global identifiers, or globals, are stored. The local namespace is partially unique to each statement block. That is to say that each statement block contains its own local namespace. Each statement block has access to the identifiers in its own namespace as well as the namespace of the statement block it is nested in, if any.

No two globals, or any two locals within the same statement block, may have the same name. But because identifiers can be the same as long as they exist in different namespaces, a nested statement block may contain the same name as the statement block it is contained in:

```
{
    int x;
    {
        int x;
    }
}
```

This causes complications because you will not be able to access the identifier in the outer statement block when inside the inner statement block. Each time you use the identifier, you will be accessing the current local one. Because of this danger, many compilers may allow you to do this, but at the same time warn you.

If you have a local that has the same identifier as a global, you cannot use the global implicitly. The situation is okay by the compiler because the two identifiers exist in separate namespaces: global and local. The global can be accessed by using the scope operator, which is made of two colons (`::`).

The scope operator allows you to specify the scope of the variable you are trying to use. A variable with global scope exists in the global namespace, and one with a local scope exists in the local namespace. So, the scope operator might also be considered the "namespace specification operator".

The syntax of the scope operator is the name of the namespace, followed by the operator, and ended with the identifier you are looking for:

```
<namespace>::<identifier>
```

The name of the global namespace is blank, i.e. non-existant. To specify the global namespace you don't put down a name, just the double-colon followed by the identifier:

```
01  #include <iostream.h>
02
03  int myglobal = 7;
04
05  int main()
06  {
07      int myglobal = 5;
```

```
08      cout << "myglobal = " << myglobal << endl;
09      cout << "::myglobal = " << ::myglobal << endl;
10      return 0;
11  }
```

The output of this program would be:

```
myglobal = 5
::myglobal = 7
```

Local namespaces have no names and cannot be specified by the scope operator. This is fine because the scope operator can only ever be used from within a statement block. The only time you'd want (remember you can't) to specify a local namespace is in the case of my example earlier; whereby a statement block has a local variable with the same identifier as a local variable in a nested statement block. In that situation you are S.O.L. and better off renaming one of the variables.

Flow Control

If a program is a river then all the ones we've written so far go one way, never change course, and disappear when they are done. The concept of flow control is performing logic decisively or multiple times; that's it. We already do these little ourselves.

If

First what is decisively? Simply put, it's using an 'if' clause. If the water is boiling we turn off the stove. If our stomach is full we stop eating, or should. And if the cat farts, we bolt like lightning. All of these actions are only performed on a condition.

In C++ programming we have the '`if`' statement. It is a *control statement* and doesn't end with a semi-colon, but is still considered a statement. A control statement defines or participates in a flow-control construct. I like to think of many control statements as *header* statements because they typically sit before a single statement or statement block. The syntax of an '`if`' statement is as follows:

```
if (condition)
```

That's it! But remember, immediately following this must be a single statement *or* statement block. That single statement or statement block will be executed only *if* the conditional expression (represented by '*condition*') results in a non-zero value. What is this conditional expression? Think back to a couple chapters ago when I explained what an expression is. An expression is one or more operations. In this case the expression must result in a value otherwise it will not compile. Luckily for us we have not dealt with operations that don't have results. By conditional I mean that the evaluation of the expression causes a condition to happen or no. Let's look at a single statement 'if' first:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int num = 0;
06      cout << "Enter a number (1-10):" << endl;
07      cin >> num;
08      if (5 == num)
09          cout << "Special message for 5." << endl;
10      return 0;
11  }
```

This program asks the user to input a number between one (1) and ten (10). If the user inputs five (5), it outputs a special message. This particular program uses a single statement '`if`'; that is, an '`if`' statement that is followed by only a single statement and not a statement block. The following is the same, but is followed by a statement block containing multiple statements:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int num = 0;
06      cout << "Enter a number (1-10):" << endl;
07      cin >> num;
08      if (5 == num)
09      {
10          cout << "Special message for 5." << endl;
11          cout << "How did we get here?" << endl;
12      }
13      cout << "You entered " << num << endl;
14      return 0;
15  }
```

In this program, no matter what is entered, the last output line is executed. Meaning if you enter '5' you'll see:

```
Enter a number (1-10):5
Special message for 5.
How did we get here?
You entered 5
```

But if you enter 6, you'll see:

```
Enter a number (1-10):6
You entered 6
```

The statement block below the '`if`' is the only one avoided if the condition of that '`if`' is *false*. A false condition is a conditional expression that results in zero (0). On the opposite side, a true condition is a conditional expression that results in *anything except zero*. Yes, this is a fine line. ☺ If you were to simply say:

```
if (0)
```

Then the statement or statement block associated with the '`if`' statement would never be executed. Likewise in the following:

```
if (1)
```

The statement or statement block associated with this 'if' would *always* be executed. As you can see, we can put any expression within the parenthesis of the '`if`' and, as long as it evaluates to a number, it will compile. For example:

```
if (6 * 2 / 4 - 3)
```

Whatever statement or statement block that followed this would never be executed. This is because the expression evaluates to zero. Most of the time, however, you'll be using *relational* and *equality* operators within the conditional expression. In fact you've already seen the first: '==' which is equality.

Relational and Equality

In the operators you've used so far, the result has been of the same type as the primary operand. It is not the case with relational and equality. The result of these operations is always an integer (type '`int`') that is either one (1) or zero (0). They are practically built to be used with 'if' statements. All of these are binary operators whose left operand is compared to the right and the result is, as I said either true (1) or false (0). Here is a list of these operators:

op	condition	name
==	equal to	equality
!=	not equal to	non-equality
<	less than	less than
<=	less than or equal to	less than or equal to
>	greater than	greater than
>=	greater than or equal to	greater than or equal to

All of these work in the basically the same way. The left operand is compared to the right using the specified condition. In the case of the equality operator, the result is true if the left operand is equal to the right operand. With the greater-than operator, the result is true if the left operand is *greater than* the right operand. I think you get the idea.

We use these operators like we use any other operator as you've seen above with equality ('=='). Let's say for instance we wanted to modify the "Special 5" program to print a special message if the user entered a number *greater* than ten (rather than within the range). Here's what it would look like:

```
01  #include <iostream.h>
```

```
02
03  int main()
04  {
05      int num = 0;
06      cout << "Enter a number (1-10):" << endl;
07      cin >> num;
08      if (num > 10)
09          cout << "Special message." << endl;
10      return 0;
11  }
```

Rather than outputting simply "Special message" it probably would have been more beneficial to print something like, "The number you entered is too large". But then the user could also enter a number *smaller* than one as well. We could handle that with two 'if' statements:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int num = 0;
06      cout << "Enter a number (1-10):" << endl;
07      cin >> num;
08      if (num > 10)
09          cout << "Number too large." << endl;
10      if (num < 1)
11          cout << "Number too small." << endl;
12      return 0;
13  }
```

This is pretty inconvenient though. Wouldn't it be better if there was a way to check for both conditions at the same time? Well, now there is actually, as demonstrated in the following:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int num = 0;
06      cout << "Enter a number (1-10):" << endl;
07      cin >> num;
08      if (num < 1 || num > 10)
09          cout << "Number out of range." << endl;
10      return 0;
11  }
```

The new thing here is the '`||`' which is a logical operator known as '*or*'.

Logical Operators

These work much the same as relational and equality operators in that their result is always of type '`int`' and either one (1) or zero (0); in other words: true or false. Luckily there isn't a lot to learn here because there are only *two* logical operators: *and* ('`&&`') and *or* ('`||`'). The bar character, if you were wondering, is usually located above the backspace key on keyboards.

These operators are used to facilitate compound or complex conditions which are made up of single conditions. For example, if you were running you'd want to stop if you reached your destination *or* your left arm went numb and you got a pain in your chest. Likewise you wouldn't want to go a birthday party unless you knew where it was *and* you were bringing a present.

The 'or' operator will have a true result (1) if either of its operands are non-zero. On the other side of the spoon the 'and' operator will have a true result (1) if both of its operands are non-zero. In the case above were we used 'or', the number entered would have to be smaller than one or greater than zero. If the number is less than one the expression would break down as follows:

```
num < 1 || num > 10
1 || num > 10
1 || 0
1
```

If we had used an 'and' operator in place of 'or' this condition would *never* be true because the number can't be less than one and greater than ten at the same time. Notice in the above that the 'or' operation is the last to be completed. Logical operators have some of the lowest precedence of all operators. This allows all the mini conditions to be evaluated first before they are all compared.

These logical operators are special, however, because the conditions on either side are not necessarily evaluated. With the 'or' operator, the right-hand condition is only evaluated if the first condition is false. This is because if the left-hand condition is true then the entire 'or' operation must be true, since either can be true for an 'or' expression to be true. Take the following for example:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int num = 1;
06      if (num || num++)
07          cout << "Some message." << endl;
08      cout << num << endl;
09      return 0;
10  }
```

Think the '`num++`' operation will be executed? Think again! Since the result of the left-hand condition is true, the second (right-hand) will never be evaluated. The output of this program would be:

```
Some message.
1
```

Pretty weird, huh? This is not the same case with the 'and' operator. In the 'and' operator the right-hand condition is only evaluated if the left-hand condition is false. This is because an 'and' expression is only true if *both* conditions result in true. So if the first (left-hand) is false then we know the expression will be false regardless of the second (right-hand) condition. Can you guess the output of the following?

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int num = 0;
06      if (num && num++)
07          cout << "Some message." << endl;
08      cout << num << endl;
09      return 0;
10  }
```

If you said just '`0`' then you are correct. Since the left-hand condition resulted in false, the right-hand condition is not evaluated. As such the '`num++`' is not performed.

Complex Conditions

Just like any operator you can string along multiple and's and or's. But in what order are they executed? The 'and' operation has a higher precedence than 'or' so it is always performed first. Take the following:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int num = 0;
06      if (num && num < 1 || num > 10)
07          cout << "Some message." << endl;
08      cout << num << endl;
09      return 0;
10  }
```

Else

Sometimes you want one of two things to be done depending on the condition, but not both. You may even want one of three, four, or more. But at the moment let's stick with one of two. Let's take a real world example: if you have a piece of fruit then eat it, otherwise go find one:

```
if (have fruit)
      eat fruit;
else
      get fruit;
```

Programmatically we don't use the term 'otherwise', we use *else* as you can see above. In C++ 'else' is a keyword and language construct. It must be matched with an 'if' statement and therefore *cannot* be used by itself. This is easy to understand, we have no reason to do otherwise if there's nothing to do otherwise from (substitute 'otherwise' with 'else'☺).

Let's look at a real example though using our little number "range" program. The following will test to see if the number is invalid. If it is (number is less than one or greater than ten) then it will print a message saying such, otherwise it will print a message saying the number is valid:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int num;
06      cout << "Enter a number (1-10):";
07      cin >> num;
08      if (num < 1 || num > 10)
09          cout << "Invalid number." << endl;
10      else
11          cout << "Valid number, thank you." << endl;
12      return 0;
13  }
```

Like the 'if' statement, else can be followed by a single statement or a statement block, like in the following:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int num;
06      cout << "Enter a number (1-10):";
07      cin >> num;
08      if (num < 1 || num > 10)
09          cout << "Invalid number." << endl;
10      else
11      {
12          cout << "Valid number." << endl;
13          cout << "Thank you." << endl;
14      }
15      return 0;
16  }
```

It is bad etiquette to use a statement block with an 'else', but not with its predecessor 'if', or vice versa. But I think you can deal with it for now and it shows you something you

may not have realized. An ‘if’ statement can be followed by a statement block while its ‘else’ statement is not, and vice versa. The statement block is simply a compounding of statements. Now, *back to the show!* What is the above doing? Let us run the program and enter the number ‘0’. Here is what the output would be:

```
Enter a number (1-10):0
Invalid number.
```

Now, if we entered ‘5’ the output would be:

```
Enter a number (1-10):5
Valid number.
Thank you.
```

Now you see the control in flow control. Our ‘if’ statement acts as a railroad switch, stemming the run of the train from one track to another:

<railroad analogy picture>

It stems the flow from one logical code path to another. A code path is the path of instruction execution. In the case where we enter the number ‘5’, the code path goes through the statement block of the ‘else’ statement.

Looping

Sometimes it’s necessary to do things multiple times. If you’ve ever done anything except stare at the ceiling (without counting those tiles); then you’ve had to do at least *something* more than once in succession. One example, you’re making cookies from scratch. Stir. Stir again. Stir again. You stir a *lot* until the pukey mixture becomes a tasty goop. Notice, that like a decision, multiple actions, or *iterations*, only continue until a condition is met. You don’t *just do* something over and over; it always has an end point, even if it’s not specific at the time.

In C++ a statement or multiple statements will be executed over and over until a conditional expression (like we’ve used with ‘if’ statements) is true. This is known as a looping; continually executing a set of instructions until a condition is met. There are three types of loop statements in C++: ‘while’, ‘do-while’, and ‘for’. Since the ‘while’ statement is most like the ‘if’ statement in structure; we’ll begin with it.

While

A while statement is simply set up as so:

```
while (condition)
```

Notice the similarity! Just like an 'if' it is followed by a single statement or statement block. Unlike an 'if' statement, however, the statement or statement block following is executed *until* the expression becomes false. Let's put this all to work in an example. If you were to count to ten, you'd say each number until you got there. Here's a program that does much the same:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int num = 1;
06      While (num <= 10)
07      {
08          cout << num << endl;
09          num++;
10      }
11      return 0;
12  }
```

This works as so: '`num`' is set to '1'. Next '`num`' is compared via 'less than or equal to' operator to '10'. If this conditional expression is true (non-zero) the statement block is executed, otherwise it is skipped entirely. To start it is true so the loop block is executed. In the loop block '`num`' is printed and then incremented (by one). The condition is then checked again. If still true the loop is executed again and this repeats until the condition is false. If you run this program you'll see the following output:

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

Yes, we are at the pinnacle of technology when a computer can count to ten faster than a human. ☺ Using our knowledge of operations we can combine the numbering printing and incrementing into a single statement, if we so desired:

```
cout << num++ << endl;
```

If we had forgotten to increment the number within the looping block, it would become endless because the condition would never be false. Creating endless loops is a common programming error even for the experts. Always make sure your loop will end. Going back to our number input program we could use a while loop to insure that we get a number in the range we wanted:

```
01  #include <iostream.h>
02
```

```
03  int main()
04  {
05      int num;
06      cout << "Enter a number (1-10):";
07      cin >> num;
08      while (num < 1 || num > 10)
09      {
10         cout << "Invalid number!  Enter again:";
11         cin >> num;
12      }
13      cout << "You entered " << num << endl;
14      return 0;
15  }
```

Hooray! This is a good program because it demonstrates a lot of what you should have learned by this point: output of text and variables, input of variables, complex relational expressions, and a while loop to boot. So let's run it and try it on some input. Try entering the number '0', then '11', then '7'. The output would be as follows:

```
Enter a number (1-10):0
Invalid number!  Enter again:11
Invalid number!  Enter again:7
You entered 7
```

Remember that *all* of the statements within a statement block are executed as part of a loop.

Do While

The next looping statement that I'll cover is the 'do-while'. This is an interesting breed of looping or just statements in general. The condition is compared effectively *after* the associated statement *block* is executed. Let me show you the syntax:

```
do … while (condition);
```

A 'do-while' requires two separate statements, 'do' and 'while', as well as a statement or statement block between them. Remember to take note of the semi-colon following the trailing 'while'. So how does this fangled thing work? First, its associated statement or statement block, which follows 'do', is executed. Next the condition is tested and if true the statement block is executed again. This continues until the condition is false and execution resumes at past the 'while' statement. Here's an example:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int num;
06      do
07      {
08         cout << "Enter a number (1-10):";
09         cin >> num;
10      } while (num < 1 || num > 10);
```

```
11      cout << "You entered " << num << endl;
12      return 0;
13  }
```

In this program the user must enter a number in the correct range or they will be endlessly bombarded with the 'Enter a number (1-10)' message. Try the input of '0', then '11', and then '7' again. You will see the following output;

```
Enter a number (1-10):0
Enter a number (1-10):11
Enter a number (1-10):7
You entered 7
```

The 'do-while' loop type is obviously useful when you want the statement(s) executed at least once. Even if the condition would have been false to start with, the statement block will be executed because it is not checked until *afterwards*.

For

I have dreaded coming to this particular looping type because it is the hardest to explain. I'll drop you face first into it with its syntax:

```
for (initialization; condition; iteration)
```

Gee, I wonder why it's so hard to explain. Perhaps because it's *very daunting to look at!* A 'for' loop is, basically, a 'while' loop "compressed". Remember our one to ten counting program using a 'while' loop? Here's a program that does the same thing with a 'for' loop:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int num;
06      for (num = 1; num <= 10; num++)
07      {
08          cout << num << endl;
09      }
10      return 0;
11  }
```

So far the language constructs you have dealt with have only used a single statement for their conditions. The 'for' statement, however, is made up of *three* statements and they must be divided using semi-colons. The one you're already familiar with is '*condition*' which is the conditional expression that decides whether or not the loop block should be executed. In the program above the condition is '`num <= 10`'.

The two you're not familiar with are '*initialization*' and '*iteration*'. The first is a statement that is executed *before* the loop is executed. This means it is executed

regardless of the conditional expression. In the program above we set '`num`' to '1' as the initialization statement. This means that even if the condition is false on the first test, '`num`' will still have been set to '1'. This statement can be practically any statement even if doesn't have a result.

It is possible to declare one or more variables as the initialization. This however has a different effect depending on your compiler and its newness. In the latest C++ standard, variables that are declared as part of a 'for' loop's initialization have their scope limited to the 'for' loop itself. In some older compilers these variables remain available after the 'for' loop is finished. For example, the following will compile fine on older compilers with this "feature":

```
01  #include <iostream.h>
02
03  int main()
04  {
05      for (int num = 1; num <= 10; num++)
06      {
07          cout << num << endl;
08      }
09      cout << "Last num is " << num << endl;
10      return 0;
11  }
```

If your compiler gives you an error when trying to build a program of above, then it is tied to the newer C++ standard. Remember, in newer compilers (and possibly some older compilers) any variables declared in a 'for' loop's initialization statement will be discarded after the loop is finished. Because of these possible differences in compilers I will stay away from declaring variables there in this book.

The last unfamiliar statement is '*iteration*' which is executed each time the loop block (or single statement) ends. Hence its name, since it is executed at each iteration. Like the initialization statement this can be practically any statement, even one that doesn't have a result. However, you cannot declare variables in this statement as you can with the initialization. I know; the *horror*. It makes sense if you think about it. If variables could be declared there, what would be their scope?

We could even put our number printing statement as the iteration, leaving the loop block empty:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int num = 1;
06      for (int num = 1; num <= 10; cout << num++ << endl)
07      {
08      }
09      return 0;
10  }
```

Remember that the iteration statement does *not* end with a semi-colon like the previous two. We could actually shrink this program a bit more. A 'for' loop, like a 'while' loop; can loop a single statement as well as a statement block. But what kind of statement would we use in the above? Why, an empty one of course. A lone semi-colon represents an empty statement:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int num = 1;
06      for (int num = 1; num <= 10; cout << num++ << endl)
07          ;
08      return 0;
09  }
```

Luckily for you my loops will not look like this in the rest of this book, I was just presenting some possibilities. I'll try to keep them as readable as possible.

You can omit any or all of the statements in a 'for' statement, but the semi-colons must remain. For example, the following program is perfectly valid:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int num = 1;
06      for ( ; num <= 10 ; )
07      {
08          cout << num << endl;
09          num++;
10      }
11      return 0;
12  }
```

The only statement present in the above is the condition. Since the initialization is empty, nothing is done. And since the iteration is empty, nothing is done each time the loop is executed. Any of the statements can be omitted. In fact, the 'for' loop is the *only* loop type that allows you to omit the condition. This doesn't mean that a loop is endless however, because there are other ways to end a loop than just its condition.

Breaking and Continuing

Most of us know and possibly fear the word premature as it has a certain scary pretence. However, there are cases when you want to prematurely exit from a loop without letting it *fall out* in the normal way of the conditional expression being false. There are also times when you'll want to stop the execution of the loop statement(s) and jump to the

condition; which effectively skips any statements yet to be executed in that iteration. There are two keywords used for each of these instances: *break* and *continue*.

Let's explore 'break' first. The following is the same number counting program, but it is done without a condition in the 'for' statement:

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int num = 1;
06      for ( ; ; )
07      {
08          cout << num << endl;
09          num++;
10          if (num >= 10)
11              break;
12      }
13      return 0;
14  }
```

All of the logic for each loop iteration and the exiting of the loop is within the statement block associated with the 'for' loop. Let's look at the loop. First it outputs '`num`'. Next '`num`' is incremented (by one). Lastly, and most importantly, there is an 'if' statement. If the condition is true ('`num`' is greater than or equal to '10') the 'break' keyword is executed which causes the looping to "break" (*dur*!!).

You may find in your future programming endeavors that there are many times when you need to break out of a loop before getting to the condition. However, this usually involves much more complicated loops so I'll hold off giving you an example. ☺

Now let us *continue*; ha ha, pun pun! The 'continue' keyword causes execution in a loop to *jump* to the end of the block. The next thing executed in a 'for' loop would be the iteration statement, then the condition is tested, then the loop either goes on and repeats this process, or it exits. With a 'while' or 'do-while' loop the next thing executed would be the conditional expression.

The following program acts the same as the one above, counting to ten, but uses 'continue' to say in the loop as long as necessary rather than 'break' to get out of it when the time has come. Its basically switching from until ten is reached to unless ten is reached.

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int num = 1;
06      for ( ; ; )
07      {
08          cout << num << endl;
```

```
09          num++;
10          if (num < 10)
11              continue;
12      }
13      return 0;
14  }
```

Goto

Extremely raw flow control can be achieved using the 'goto' statement. This "jumps" execution from the point where it is called to the label that is specified. The syntax for this is:

```
goto <label>;
```

The '*<label>*' is a marker you place in the current top-level statement block. The marker is an identifier, following the same rules as all identifiers, followed by a colon. It simply provides an "anchor" that code execution can jump to.

```
01  #include <iostream.h>
02
03  int main()
04  {
05      int num = 1;
06  myloop:
07      cout << num << endl;
08      num++;
09      if (num < 10)
11          goto myloop;
12      return 0;
13  }
```

The above example program creates the label 'myloop' and uses it as an anchor for the line 11, 'goto myloop'. Each time it is encountered, execution jumps up to where 'myloop' is declared on line 6. This simulates the same loop we've done all along with a more explicit method.

Branching

Changing the flow of a program is also known as *branching*. Conceptually there are two types of branching: conditional and unconditional. A conditional branch is one that occurs only if a condition is met. An unconditional branch is one that occurs regardless, usually because it has to. This is the case with 'do', 'goto', and function calls[1].
You have seen this with 'if', 'while', and 'for'.

[1] Don't panic, I haven't explained these yet. Just know that they are considered unconditional branches.

Imagine a giant tree as your program and execution is you climbing the tree beginning at the trunk. At certain points you will have the option of climbing onto a new limb or staying on the one you're at. This decision would be based on a condition: if I set foot on that one I'll die or that one looks sturdy enough. If you suddenly climbed somewhere else on the tree it would be because you had to, like if you reached the end of the current branch.

Alternate Terminology

- compound statement: statement block
- branching: flow control
- iteration statement: loop statement