Chapter Two

3. Control Statements 1

3.1. INTRODUCTION 1

3.2. CONDITIONAL STATEMENTS 1

3.2.1. The if Statement 1

3.2.2. The switch Statement 4

3.3. LOOPING STATEMENTS 5

3.3.1. The ‘for’ Statement 7

3.3.2. The ‘while’ Statement 5

3.3.3. The ‘do…while’ Statement 8

3.4. OTHER STATEMENTS 9

3.4.1. The ‘continue’ Statement 9

3.4.2. The ‘break’ Statement 11

3.4.3. The ‘goto’ Statement 12

3.4.4. The ‘return’ Statement 12

3. Control Statements

3.1. Introduction

A running program spends all of its time executing statements. The order in which statements are executed is called **flow control** (or control flow). This term reflect the fact that the currently executing statement has the *control* of the CPU, which when completed will be handed over (*flow*) to another statement. Flow control in a program is typically sequential, from one statement to the next, but may be diverted to other paths by branch statements. Flow control is an important consideration because it determines what is executed during a run and what is not, therefore affecting the overall outcome of the program.

Like many other procedural languages, C++ provides different forms of statements for different purposes. Declaration statements are used for defining variables. Assignment-like statements are used for simple, algebraic computations. Branching statements are used for specifying alternate paths of execution, depending on the outcome of a logical condition. Loop statements are used for specifying computations, which need to be repeated until a certain logical condition is satisfied. Flow control statements are used to divert the execution path to another part of the program. We will discuss these in turn.

3.2. Conditional Statements

3.2.1. The if Statement

It is sometimes desirable to make the execution of a statement dependent upon a condition being satisfied. The if statement provides a way of expressing this, the general form of which is:

```
if (expression)
    statement;
```

First *expression* is evaluated. If the outcome is nonzero (true) then *statement* is executed. Otherwise, nothing happens.

For example, when dividing two values, we may want to check that the denominator is nonzero:

```
#include<iostream.h>
int main()
{
int a=5;
int b=10;
If (a<b)
cout<<"five is less than ten";
Return 0;
}
```

To make multiple statements dependent on the same condition, we can use a compound statement:

```
if (balance > 0) {
    interest = balance * creditRate;
    balance += interest;
}
```

A variant form of the if statement allows us to specify two alternative statements: one which is executed if a condition is satisfied and one which is executed if the condition is not satisfied. This is called the if-else statement and has the general form:

```
if (expression)
     statement1;
else
     statement2;
```

First expression is evaluated. If the outcome is nonzero (true) then statement1 is executed. Otherwise, *statement*$_2$ is executed.

For example:

```
#include <iostream.h>

int main ()

{

   int a = 100;

   if( a < 20 )

   {

     cout << "a is less than 20;" << endl;

   }

   else

   {

 cout << "a is not less than 20;" << endl;

   }

   cout << "value of a is : " << a << endl;

   Return 0;

}

output: a is not less than 20

         value of a is :100
```

Given the similarity between the two alternative parts, the whole statement can be simplified to:

```
if (balance > 0)
    interest = balance * creditRate;
else
    interest = balance * debitRate;
balance += interest;
```

Or simplified even further using a conditional expression:

```
interest = balance * (balance > 0 ? creditRate : debitRate);
balance += interest;
```

Or just:

```
balance += balance * (balance > 0 ? creditRate : debitRate);
```

If statements may be nested by having an if statement appear inside another if statement. For example:

```
if (callHour > 6) {
    if (callDuration <= 5)
        charge = callDuration * tarrif1;
    else
        charge = 5 * tarrif1 + (callDuration - 5) * tarrif2;
} else
    charge = flatFee;
```

A frequently-used form of nested if statements involves the else part consisting of another if-else statement.

For example:

```
#include<iostream.h>
int main()
{

int age, height;
cout<<"Enter your age=:"<<endl;
cin >> age;
cout<<"Enter your height=:"<<endl;
cin >> height;
if(height = = age)
{
             cout << "Your height is equal to your age!";
}
else if(height < age)
{
             cout << "Your height is less than your age!";
}
else if(height > age)
{
             cout << "Your height is greater than your age!";
}
}
```

3.2.2. The switch Statement

The switch statement provides a way of choosing between a set of alternatives, based on the value of an expression. The general form of the switch statement is:

```
switch (expression) {
    case constant1:
        statements;
    ...
    case constantn:
        statements;
    default:
        statements;
}
```

First *expression* (called the switch **tag**) is evaluated, and the outcome is compared to each of the numeric *constant*s (called case **labels**), in the order they appear, until a match is found. The *statements* following the matching case are then executed. Note the plural: each case may be followed by zero or more statements (not just one statement). Execution continues until either a break statement is encountered or all intervening statements until the end of the switch statement are

executed. The final default case is optional and is exercised if none of the earlier cases provide a match.

For example, #include<iostream.h>

```
int main()
{

switch (4)
{
  case 1:
     cout <<"The correct choice is one" << endl;
     break;
  case 2:
     cout <<"The correct choice is two" << endl;
     break;
  case 3:
     cout << "The correct choice is three"  << endl;
     break;
  case 4:
     cout << "The correct choice is four"  << endl;
     break;
  default:
     cout <<"The correct choice is five" << endl;
     break;
}
}
```

3.3.Looping Statements

3.3.1. The ‘while’ Statement

The while statement (also called **while loop**) provides a way of repeating a statement while a condition holds. It is one of the three flavors of **iteration** in C++. The general form of the while statement is:

```
while (expression)
    statement;
```

First *expression* (called the **loop condition**) is evaluated. If the outcome is nonzero then *statement* (called the **loop body**) is executed and the whole process is repeated. Otherwise, the loop is terminated.

For example, suppose we wish to calculate the sum of all numbers from 1 to some integer denoted by *5*.

```
#include <iostream.h>
int main ()
{
int i=1;
int sum = 0;
while (i<=5)
{
sum =sum+i;
i++;
}
cout<<"The sum=:"<<sum;
 return 0;
}
```

For n set to 5, Table 2.9 provides a trace of the loop by listing the values of the variables involved and the loop condition.

Table 5.1 While loop trace

Iteration	i	n	i <= n	sum += i++
First	1	5	1	1
Second	2	5	1	3
Third	3	5	1	6
Fourth	4	5	1	10
Fifth	5	5	1	15
Sixth	6	5	0	

It is not unusual for a while loop to have an empty body (i.e., a null statement). The following loop, for example, sets n to its greatest odd factor.

```
while (n % 2 == 0 && n /= 2)
    ;
```

Here the loop condition provides all the necessary computation, so there is no real need for a body. The loop condition not only tests that n is even, it also divides n by two and ensures that the loop will terminate should n be zero.

Example:

```
#include <iostream.h>
int main ()
{
  int a = 10;
  while( a < 20 )
  {
     cout << "value of a: " << a << endl;
     a++;
  }
  Return 0;
}
```

3.3.2. The 'for' Statement

The for statement (also called **for loop**) is similar to the while statement, but has two additional components: an expression which is evaluated only once before everything else, and an expression which is evaluated once at the end of each iteration. The general form of the for statement is:

```
for (expression1; expression2; expression3)
    statement;
```

First $expression_1$ is evaluated. Each time round the loop, $expression_2$ is evaluated. If the outcome is nonzero then statement is executed and $expression_3$ is evaluated. Otherwise, the loop is terminated. The general for loop is equivalent to the following while loop:

```
expression1;
```

```
while (expression2) {
    statement;
    expression3;
}
```

```
Example
#include <iostream.h>
int main ()
{
   for( int a = 10; a < 20; a = a ++ )
   {
       cout << "value of a: " << a << endl;
   }

   return 0;
}
```

3.3.3. The 'do…while' Statement

The do statement (also called **do loop**) is similar to the while statement, except that its body is executed first and then the loop condition is examined. The general form of the do statement is:

```
do
    statement;
while (expression);
```

First *statement* is executed and then *expression* is evaluated. If the outcome of the latter is nonzero then the whole process is repeated. Otherwise, the loop is terminated.
The do loop is less frequently used than the while loop. It is useful for situations where we need the loop body to be executed at least once, regardless of the loop condition. For example, suppose we wish to repeatedly read a value and print its square, and stop when the value is zero. This can be expressed as the following loop:

```
#include <iostream.h>
int main ()
{
   int a = 10;
   do
```

```
   {
   cout << "value of a: " << a << endl;
       a = a + 1;
   }while( a < 20 );

   return 0;
}
```

Unlike the while loop, the do loop is never used in situations where it would have a null body. Although a do loop with a null body would be equivalent to a similar while loop, the latter is always preferred for its superior readability.

3.4. Other Statements

3.4.1. The 'continue' Statement

The `continue` statement terminates the current iteration of a loop and instead jumps to the next iteration. It applies to the loop immediately enclosing the continue statement. It is an error to use the continue statement outside a loop.

In while and do loops, the next iteration commences from the loop condition. In a for loop, the next iteration commences from the loop's third expression. For example, a loop which repeatedly reads in a number, processes it but ignores negative numbers, and terminates when the number is zero, may be expressed as:

Example:

```
#include <iostream.h>
int main ()
{
   int a = 10;
   do
   {
       if( a == 15)
       {
          a = a + 1;
```

```
            Continue;
        }
        cout << "value of a: " << a << endl;
        a = a + 1;
    } while( a < 20 );

    Return 0;
}
Output:
Value of a:10
Value of a:11
Value of a:12
Value of a:13
Value of a:14
Value of a:15
Value of a:16
Value of a:17
Value of a:18
Value of a:19
```

A variant of this loop which reads in a number exactly *n* times (rather than until the number is zero) may be expressed as:

```
for (i = 0; i < n; ++i) {
    cin >> num;
    if (num < 0) continue;          // causes a jump to: ++i
    // process num here...
}
```

When the continue statement appears inside nested loops, it applies to the loop immediately enclosing it, and not to the outer loops. For example, in the following set of nested loops, the continue applies to the for loop, and not the while loop:

```
while (more) {
    for (i = 0; i < n; ++i) {
        cin >> num;
        if (num < 0) continue;          // causes a jump to: ++i
        // process num here...
    }
    //etc...
}
```

3.4.2. The 'break' Statement

A `break` statement may appear inside a loop (while, do, or for) or a switch statement. It causes a jump out of these constructs, and hence terminates them. Like the continue statement, a break statement only applies to the loop or switch immediately enclosing it. It is an error to use the break statement outside a loop or a switch.

For example, suppose we wish to read in a user password, but would like to allow the user a limited number of attempts:

Example:

```
#include<iostream.h>
int main()
{
for(int y=0; y<3; y++)
{
cout<<" The User is a very good programmer :"<<endl;
if(y==3)
{
cout<<" The User is a very bad programmer :"<<endl;
break;
cout<<" It was a Mistake :"<<endl;
}
}
return 0;
} output:
The User is a very good programmer :
The User is a very good programmer :
The User is a very good programmer :
```

Here we have assumed that there is a function called `Verify` which checks a password and returns true if it is correct, and false otherwise.

Rewriting the loop without a break statement is always possible by using an additional logical variable (verified) and adding it to the loop condition:

```
verified = 0;
for (i = 0; i < attempts && !verified; ++i) {
    cout << "Please enter your password: ";
    cin >> password;
    verified = Verify(password));
    if (!verified)
        cout << "Incorrect!\n";
```

```
}
```

The break version is arguably simpler and therefore preferred.

3.4.3. The 'goto' Statement

The goto statement provides the lowest-level of jumping. It has the general form:

`goto` *label;*

where *label* is an identifier which marks the jump destination of goto. The label should be followed by a colon and appear before a statement within the same function as the goto statement itself. For example, the role of the break statement in the for loop in the previous section can be emulated by a goto:

```
example:
#include<iostream.h>
int main()
{

for(int y=0; y<3; y++)
{
cout<<" The User is a very good programmer :"<<endl;
if(y==3)
{
cout<<" It was a Mistake :"<<endl;
goto a;
}
}
cout<<endl;
a:
{
cout<<" It is a result of goto statement ";
}

return 0;
}
Output: The User is a very good programmer:
The User is a very good programmer:
The User is a very good programmer:
It is a result of go to statement
```

Because goto provides a free and unstructured form of jumping (unlike break and continue), it can be easily misused. Most programmers these days avoid using it altogether in favor of clear programming. Nevertheless, goto does have some legitimate (though rare) uses.

3.4.4. The 'return' Statement

The `return` statement enables a function to return a value to its caller. It has the general form:

`return` *expression;*

where *expression* denotes the value returned by the function. The type of this value should match the return type of the function. For a function whose return type is void, *expression* should be empty:

```
return;
```

The only function we have discussed so far is `main`, whose return type is always `int`. The return value of main is what the program returns to the operating system when it completes its execution. Under UNIX, for example, it its conventional to return 0 from `main` when the program executes without errors. Otherwise, a non-zero error code is returned. For example:

```
int main (void)
{
    cout << "Hello World\n";
    return 0;
}
```

When a function has a non-void return value (as in the above example), failing to return a value will result in a compiler warning. The actual return value will be undefined in this case (i.e., it will be whatever value which happens to be in its corresponding memory location at the time).