Copperbelt University
School of Information and Communication Technology
Computer Science Department

Introduction to Programming in C++

Extracted from Sam's Teach yourself C++ in 21 Days

More Program Flow

Programs accomplish most of their work by branching and looping.

Looping

Many programming problems are solved by repeatedly acting on the same data. There are two ways to do this: recursion and iteration. Iteration means doing the same thing again and again. The principal method of iteration is the loop.

The Roots of Looping goto

In the primitive days of early computer science, programs were nasty, brutish, and short. Loops consisted of a label, some statements, and a jump. In C++, a label is just a name followed by a colon (`:`). The label is placed to the left of a legal C++ statement, and a jump is accomplished by writing `goto` followed by the label name. The program below illustrates this.

```
1:    //
2:    // Looping with goto
3:
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:           int counter = 0;      // initialize counter
9:    loop:  counter ++;           // top of the loop
10:            cout << "counter: " << counter << "\n";
11:           if (counter < 5)     // test the value
12:               goto loop;       // jump to the top
13:
14:           cout << "Complete. Counter: " << counter << ".\n";
15:       return 0;
16: }
```

On line 8, `counter` is initialized to `0`. The label `loop` is on line 9, marking the top of the loop. `Counter` is incremented and its new value is printed. The value of `counter` is tested on line 11. If it is less than 5, the `if` statement is true and the `goto` statement is executed. This causes program execution to jump back to line 9. The program continues looping until `counter` is equal to 5, at which time it "falls through" the loop and the final output is printed.

while Loops

A `while` loop causes your program to repeat a sequence of statements as long as the starting condition remains true. In the example of `goto`, in the previous program, the counter was incremented until it was equal to 5. The following program shows the same program rewritten to take advantage of a `while` loop.

```
1:    //
2:    // Looping with while
3:
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:      int counter = 0;               // initialize the condition
9:
10:      while(counter < 5)     // test condition still true
11:       {
12:          counter++;              // body of the loop
13:          cout << "counter: " << counter << "\n";
14:       }
15:
16:       cout << "Complete. Counter: " << counter << ".\n";
17:        return 0;
18: }
```

This simple program demonstrates the fundamentals of the *`while`* loop. A condition is tested, and if it is true, the body of the `while` loop is executed. In this case, the condition tested on line 10 is whether `counter` is less than 5. If the condition is true, the body of the loop is executed; on line 12 the counter is incremented, and on line 13 the value is printed. When the conditional statement on line 10 fails (when `counter` is no longer less than 5), the entire body of the `while` loop (lines 11-14) is skipped. Program execution falls through to line 15.

The while Statement

The syntax for the `while` statement is as follows:

```
while ( condition )
statement;
```

condition is any C++ expression, and statement is any valid C++ statement or block of statements. When condition evaluates to `TRUE` (`1`), statement is executed, and then condition is tested again. This continues until condition tests `FALSE`, at which time the `while` loop terminates and execution continues on the first line below statement. Consider the example below:

```
// count to 10
int x = 0;
while (x < 10)
cout << "X: " << x++;
```

The condition tested by a `while` loop can be as complex as any legal C++ expression. This can include expressions produced using the logical `&&` (AND), `||` (OR), and `!` (NOT) operators.

Continue and Break

At times you'll want to return to the top of a `while` loop before the entire set of statements in the `while` loop is executed. The *`continue`* statement jumps back to the top of the loop. At other times, you may want to exit the loop before the exit conditions are met. The *`break`* statement immediately exits the `while` loop, and program execution resumes after the closing brace. The program below demonstrates the use of these statements on a small game. The user is invited to enter a `small` number and a `large` number, a `skip` number, and a `target` number. The `small` number will be incremented by one, and the `large` number will be decremented by 2. The decrement will be skipped each time the `small` number is a multiple of the `skip`. The game ends if `small` becomes larger than `large`. If the `large` number reaches the `target` exactly, a statement is printed and the game stops. The user's goal is to put in a target number for the `large` number that will stop the game.

```
1:    // Program to
2:    // Demonstrates break and continue
3:
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:      unsigned short small;
9:      unsigned long  large;
10:      unsigned long  skip;
11:      unsigned long target;
12:      const unsigned short MAXSMALL=65535;
13:
14:      cout << "Enter a small number: ";
15:      cin >> small;
16:      cout << "Enter a large number: ";
17:      cin >> large;
18:      cout << "Enter a skip number: ";
19:      cin >> skip;
20:      cout << "Enter a target number: ";
21:      cin >> target;
22:
23:     cout << "\n";
24:
25:     // set up 3 stop conditions for the loop
26:      while (small < large && large > 0 && small < 65535)
27:
28:      {
29:
30:        small++;
31:
32:         if (small % skip == 0)  // skip the decrement?
33:         {
34:           cout << "skipping on " << small << endl;
35:           continue;
36:         }
37:
38:         if (large == target)    // exact match for the target?
39:         {
40:           cout << "Target reached!";
41:           break;
```

```
42:          }
43:
44:          large-=2;
45:       }                    // end of while loop
46:
47:       cout << "\nSmall: " << small << " Large: " << large << endl;
48:     return 0;
49: }
```

Both `continue` and `break` should be used with caution. They are the next most dangerous commands after `goto`, for much the same reason. Programs that suddenly change direction are harder to understand, and liberal use of `continue` and `break` can render even a small `while` loop unreadable. `Continue` causes a `while` or `for` loop to begin again at the top of the loop. `Break` causes the immediate end of a `while` or `for` loop. Execution jumps to the closing brace. Example

While (1) Loops

The condition tested in a `while` loop can be any valid C++ expression. As long as that condition remains true, the `while` loop will continue. You can create a loop that will never end by using the number 1 for the condition to be tested. Since 1 is always true, the loop will never end, unless a `break` statement is reached. The program below demonstrates counting to 10 using this construct.

```
1:    //
2:    // Demonstrates a while true loop
3:
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:      int counter = 0;
9:
10:      while (1)
11:      {
12:         counter ++;
13:         if (counter > 10)
14:             break;
15:      }
16:      cout << "Counter: " << counter << "\n";
17:       return 0;
18:
```

Eternal loops such as `while(1)` can cause your computer to hang if the exit condition is never reached. Use these with caution and test them thoroughly.

Do…while Loops

The `do...while` loop executes the body of the loop before its condition is tested and ensures that the body always executes at least one time. The following program demonstrates this.

```
1:      //
```

```
2:      // Demonstrates do while
3:
4:      #include <iostream.h>
5:
6:      int main()
7:      {
8:         int counter;
9:         cout << "How many hellos? ";
10:        cin >> counter;
11:        do
12:        {
13:           cout << "Hello\n";
14:           counter--;
15:        }  while (counter >0 );
16:        cout << "Counter is: " << counter << endl;
17:         return 0;
18: }
```

The user is prompted for a starting value on line 9, which is stored in the integer variable `counter`. In the `do...while` loop, the body of the loop is entered before the condition is tested, and therefore the body of the loop is guaranteed to run at least once. On line 13 the message is printed, on line 14 the counter is decremented, and on line 15 the condition is tested. If the condition evaluates `TRUE`, execution jumps to the top of the loop on line 13; otherwise, it falls through to line 16. The `continue` and `break` statements work in the `do...while` loop exactly as they do in the `while` loop. The only difference between a `while` loop and a `do...while` loop is when the `condition` is tested.

The do...while Statement

The syntax for the `do...while` statement is as follows:

```
do
statement
while (condition);
```

> statement is executed, and then condition is evaluated. If condition is `TRUE`, the loop is repeated; otherwise, the loop ends. The statements and conditions are otherwise identical to the `while` loop.

The For Loops

A `for` loop combines three steps into one statement. The three steps are initialization, test, and increment. A `for` statement consists of the keyword `for` followed by a pair of parentheses. Within the parentheses are three statements separated by semicolons. The first statement is the initialization. Any legal C++ statement can be put here, but typically this is used to create and initialize a counting variable. Statement 2 is the test, and any legal C++ expression can be used here. This serves the same role as the condition in the `while` loop. Statement 3 is the action. Typically a value is incremented or decremented, though any legal C++ statement can be put here. Note that statements 1 and 3 can be any legal C++ statement, but statement 2 must be an expression, i.e., a C++ statement that returns a value. The following program demonstrates a `for` loop.

```
1:      //
2:      // Looping with for
3:
4:      #include <iostream.h>
5:
6:      int main()
7:      {
8:        int counter;
9:        for (counter = 0; counter < 5; counter++)
10:          cout << "Looping! ";
11:
12:        cout << "\nCounter: " << counter << ".\n";
13:         return 0;
14: }
```

The `for` statement on line 8 combines the initialization of `counter`, the test that `counter` is less than 5, and the increment of `counter` all into one line. The body of the `for` statement is on line10. Of course, a block could be used here as well.

The for Statement

The syntax for the `for` statement is as follows:

```
for (initialization; test; action )
statement;
```

The initialization statement is used to initialize the state of a `counter`, or to otherwise prepare for the loop. test is any C++ expression and is evaluated each time through the loop. If test is `TRUE`, the action in the header is executed (typically the counter is incremented) and then the body of the `for` loop is executed. `for` statements are powerful and flexible. The three independent statements (initialization, test, and action) lend themselves to a number of variations. A `for` loop works in the following sequence:

Multiple Initialization and Increments

It is not uncommon to initialize more than one variable, to test a compound logical expression, and to execute more than one statement. The initialization and the action may be replaced by multiple C++ statements, each separated by a comma. The program below demonstrates the initialization and increment of two variables.

```
1:  //
2:  // demonstrates multiple statements in
3:  // for loops
4:
5: #include <iostream.h>
6:
7:  int main()
8:  {
9:      for (int i=0, j=0; i<3; i++, j++)
10:          cout << "i: " << i << " j: " << j << endl;
11:     return 0;
12: }
```

On line 9, two variables, `i` and `j`, are each initialized with the value `0`. The test (`i<3`) is evaluated, and because it is true, the body of the `for` statement is executed, and the values are printed. Finally, the third clause in the `for` statement is executed, and `i` and `j` are incremented. Once line 10 completes, the condition is evaluated again, and if it remains true the actions are repeated (`i` and `j` are again incremented), and the body of the loop is executed again. This continues until the test fails, in which case the action statement is not executed, and control falls out of the loop.

Null Statements in for Loops

Any or all of the statements in a `for` loop can be null. To accomplish this, use the semicolon to mark where the statement would have been. To create a `for` loop that acts exactly like a `while` loop, leave out the first and third statements. The following program illustrates this idea.

```
1:    //
2:    // For loops with null statements
3:
4:    #include <iostream.h>
5:
6:    int main()
7:    {
8:        int counter = 0;
9:
10:        for( ; counter < 5; )
11:        {
12:           counter++;
13:           cout << "Looping!  ";
14:        }
15:
16:        cout << "\nCounter: " << counter << ".\n";
17:       return 0;
18: }
```

On line 8, the counter variable is initialized. The `for` statement on line 10 does not initialize any values, but it does include a test for `counter < 5`. There is no increment statement, so this loop behaves exactly as if it had been written: while (counter < 5)

Once again, C++ gives you a number of ways to accomplish the same thing. No experienced C++ programmer would use a `for` loop in this way, but it does illustrate the flexibility of the `for` statement. In fact, it is possible, using `break` and `continue`, to create a `for` loop with none of the three statements. The following program illustrates how this is done.

```
1:     //Illustrating
2:     //empty for loop statement
3:
4:     #include <iostream.h>
5:
6:     int main()
7:     {
8:         int counter=0;       // initialization
9:         int max;
10:         cout << "How many hellos?";
11:         cin >> max;
```

```
12:          for (;;)          // a for loop that doesn't end
13:          {
14:             if (counter < max)       // test
15:             {
16:               cout << "Hello!\n";
17:               counter++;          // increment
18:             }
19:             else
20:                 break;
21:          }
22:         return 0;
23: }
```

The `for` loop has now been pushed to its absolute limit. Initialization, test, and action have all been taken out of the `for` statement. The initialization is done on line 8, before the `for` loop begins. The test is done in a separate `if` statement on line 14, and if the test succeeds, the action, an increment to `counter`, is performed on line 17. If the test fails, breaking out of the loop occurs on line 20. While this particular program is somewhat absurd, there are times when a `for(;;)` loop or a `while (1)` loop is just what you'll want.

Nested Loops

Loops may be nested, with one loop sitting in the body of another. The inner loop will be executed in full for every execution of the outer loop. The following program illustrates writing marks into a matrix using nested `for` loops.

```
1:  //Program to
2:  //illustrate nested for loops
3:
4:  #include <iostream.h>
5:
6:  int main()
7:  {
8:     int rows, columns;
9:     char theChar;
10:     cout << "How many rows? ";
11:     cin >> rows;
12:     cout << "How many columns? ";
13:     cin >> columns;
14:     cout << "What character? ";
15:     cin >> theChar;
16:     for (int i = 0; i<rows; i++)
17:     {
18:       for (int j = 0; j<columns; j++)
19:          cout << theChar;
20:       cout << "\n";
21:     }
22:    return 0;
23: }
```

The user is prompted for the number of rows and columns and for a character to print. The first for loop, on line 16, initializes a counter (i) to 0, and then the body of the outer for loop is run. On line 18, the first line of the body of the outer for loop, another for loop is established. A second counter

(j) is also initialized to 0, and the body of the inner for loop is executed. On line 19, the chosen character is printed, and control returns to the header of the inner for loop. Note that the inner for loop is only one statement (the printing of the character). The condition is tested (j < columns) and if it evaluates true, j is incremented and the next character is printed. This continues until j equals the number of columns.

Once the inner `for` loop fails its test, in this case after 12 `X`s are printed, execution falls through to line 20, and a new line is printed. The outer `for` loop now returns to its header, where its condition (`i < rows`) is tested. If this evaluates `true`, `i` is incremented and the body of the loop is executed. In the second iteration of the outer `for` loop, the inner `for` loop is started over. Thus, `j` is reinitialized to `0` and the entire inner loop is run again. The important idea here is that by using a nested loop, the inner loop is executed for each iteration of the outer loop. Thus the character is printed `columns` times for each row.

The Switch Statements

We have already seen how to write `if` and `if/else` statements. These can become quite confusing when nested too deeply, and C++ offers an alternative. Unlike `if`, which evaluates one value, `switch` statements allow you to branch on any of a number of different values. The general form of the `switch` statement is:

```
switch (expression)
{
case valueOne: statement;
                   break;
case valueTwo: statement;
                   break;
....
case valueN:   statement;
                   break;
default:       statement;
}
```

expression is any legal C++ expression, and the statements are any legal C++ statements or block of statements. `switch` evaluates `expression` and compares the result to each of the `case` values. Note, however, that the evaluation is only for equality; relational operators may not be used here, nor can Boolean operations. If one of the `case` values matches the expression, execution jumps to those statements and continues to the end of the `switch` block, unless a `break` statement is encountered. If nothing matches, execution branches to the optional `default` statement. If there is no default and there is no matching value, execution falls through the `switch` statement and the statement ends. It is almost always a good idea to have a `default case` in `switch` statements. If you have no other need for the `default`, use it to test for the supposedly impossible case, and print out an error message; this can be a tremendous aid in debugging. It is important to note that if there is no `break` statement at the end of a `case` statement, execution will fall through to the next `case` statement. This is sometimes necessary, but usually is an error. If you decide to let execution fall through, be sure to put a comment, indicating that you didn't just forget the `break`. The program below illustrates use of the `switch` statement.

```
1:  //Program to
2:  // demonstrate switch statement
3:
4:  #include <iostream.h>
5:
```

```
6:  int main()
7:  {
8:    unsigned short int number;
9:    cout << "Enter a number between 1 and 5: ";
10:    cin >> number;
11:    switch (number)
12:    {
13:       case 0:   cout << "Too small, sorry!";
14:                 break;
15:       case 5:  cout << "Good job!\n";  // fall through
16:       case 4:  cout << "Nice Pick!\n"; // fall through
17:       case 3:  cout << "Excellent!\n"; // fall through
18:       case 2:  cout << "Masterful!\n"; // fall through
19:       case 1:  cout << "Incredible!\n";
20:                break;
21:       default: cout << "Too large!\n";
22:                break;
23:    }
24:    cout << "\n\n";
25:     return 0;
26: }
```

The user is prompted for a number. That number is given to the `switch` statement. If the number is 0, the `case` statement on line 13 matches, the message `Too small, sorry!` is printed, and the `break` statement ends the switch. If the value is `5`, execution switches to line 15 where a message is printed, and then falls through to line 16, another message is printed, and so forth until hitting the `break` on line 20. The net effect of these statements is that for a number between 1 and 5, that many messages are printed. If the value of number is not 0-5, it is assumed to be too large, and the `default` statement is invoked on line 21.