Lab Session 06

Looping Constructs, for loop, while Loop and do while Loop.

Objectives:

1. Illustration of Looping Constructs
2. Illustration of for loop, while Loop, do while Loop.
3. Compile and execution of simple loop constructs program using C++.
4. Compare & Contrast While, Do while and for loop.

Loops:

Loops cause a section of your program to be repeated a certain number of times. The repetition continues while a condition is true. When the condition becomes false, the loop ends and control passes to the statements following the loop. There are three kinds of loops in C++: the for loop, the while loop, and the do loop.

If you worked on the password program practice problem at the end of the last chapter (Jumpung into c++, Chapter No. 4 Page 68-72) that asked you to re-prompt the user on a failed password entry, you probably had to hard-code in a series of if-statements to recheck the password; there was no way to allow a user to re-enter a password until he enters the correct password.

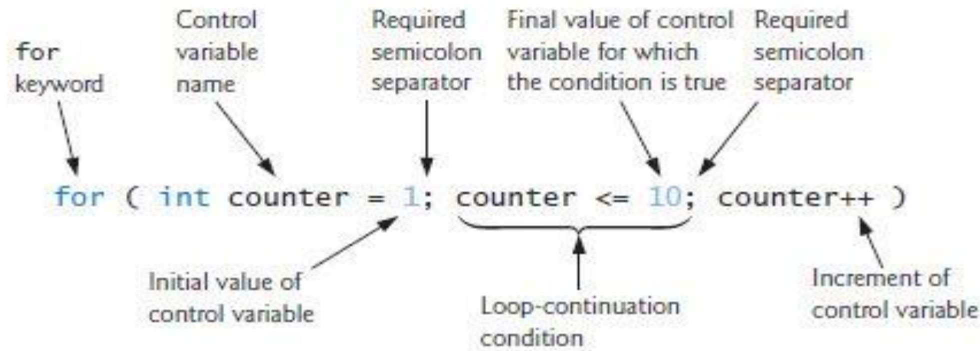
That's what loops are for. Loops repeatedly execute a block of code. Loops are extremely powerful and core parts of most programs. Now, think about what this means: a loop lets you write a very simple statement to produce a significantly greater result simply by repetition. You can prompt a user for a password as many times as the user is willing to try to enter a password.

for Loop:

C++ provides the for repetition statement, which specifies the counter-controlled repetition details in a single line of code.

Syntax:

```
for ( initialization; loopContinuationCondition; increment )
statement
```



To illustrate the power of for, let's write the program of Fig. 1. The result is shown in Fig. 2.

```
// Counter-controlled repetition with the for statement.
#include <iostream>
using namespace std;
int main()
{
        // for statement header includes initialization,
        // loop-continuation condition and increment.
        for ( int counter = 1; counter <= 10; ++counter )
        cout << counter << " ";
        cout << endl; // output a newline
} // end main
```

Output

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

Illustration

When the for statement (lines 10–11) begins executing, the control variable counter is declared and initialized to 1. Then, the loop-continuation condition (line 10 between the semicolons) counter <= 10 is checked. The initial value of counter is 1, so the condition is satisfied and the body statement (line 11) prints the value of counter, namely 1. Then, the expression ++counter increments control variable counter and the loop begins again with the loop-continuation test. The control variable is now equal to 2, so the final value is not exceeded and the program performs the body statement again. This process continues until the loop body has executed 10 times and the control variable counter is incremented to 11—this causes the loop-continuation test to fail and repetition to terminate. The program continues by performing the first statement after the for statement (in this case, the output statement in line 13).

The initialization, loop-continuation condition and increment expressions of a for statement can contain arithmetic expressions. For example, if x = 2 and y = 10, and x and y are not modified in the loop body, the for header:

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

is equivalent to:

```
for ( int j = 2; j <= 80; j += 5 )
```

The "increment" of a for statement can be negative, in which case it's really a *decrement* and the loop actually counts *downward* as shown below:

If the loop-continuation condition is initially false, the body of the for statement is not performed. Instead, execution proceeds with the statement following the for.
Frequently, the control variable is printed or used in calculations in the body of a for statement, but this is not required. It's common to use the control variable for controlling repetition while never mentioning it in the body of the for statement.

The following examples show methods of varying the control variable in a for statement. In each case, we write the appropriate for statement header. Note the change in the relational operator for loops that decrement the control variable.

a) Vary the control variable from 1 to 100 in increments of 1.

```
for ( int i = 1; i <= 100; ++i )
```

b) Vary the control variable from 100 down to 1 in decrements of 1.

```
for ( int i = 100; i >= 1; --i )
```

c) Vary the control variable from 7 to 77 in steps of 7.

```
for ( int i = 7; i <= 77; i += 7 )
```

d) Vary the control variable from 20 down to 2 in steps of -2.

```
for ( int i = 20; i >= 2; i -= 2 )
```

e) Vary the control variable over the following sequence of values: 2, 5, 8, 11, 14, 17.

```
for ( int i = 2; i <= 17; i += 3 )
```

f) Vary the control variable over the following sequence of values: 99, 88, 77, 66, 55.

```
for ( int i = 99; i >= 55; i -= 11 )
```

Example 1:
Program that displays the squares of the numbers from 0 to 14:

```
#include <iostream>
using namespace std;
int main()
{
	int j; //define a loop variable
	for(j=0; j<15; j++) //loop from 0 to 14,
	cout << j * j << " "; //displaying the square of j
	cout << endl;
return 0;
}
```

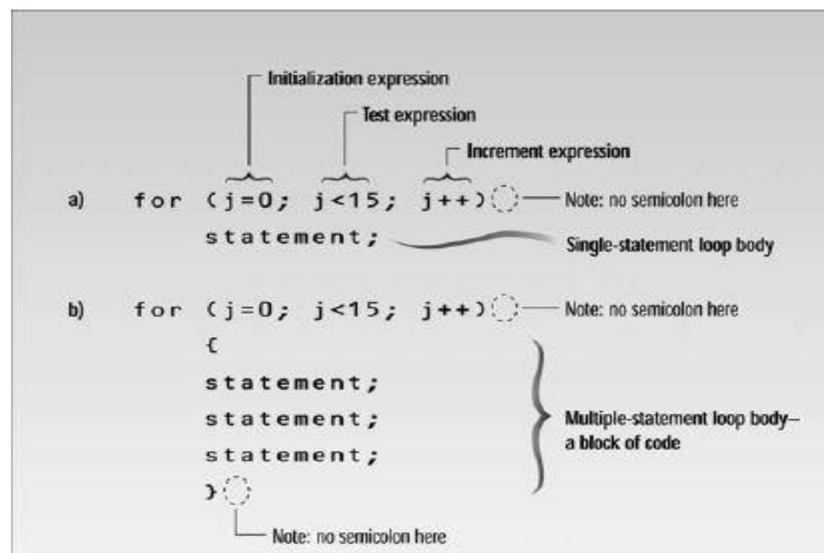
Output

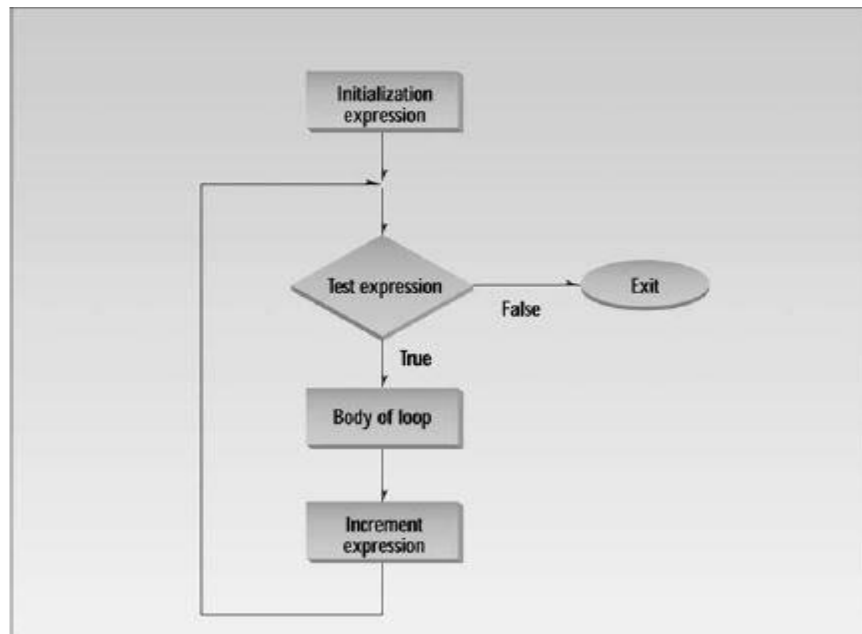
```
0 1 4 9 16 25 36 49 64 81 100 121 144 169 196
```

How does this work? The for statement controls the loop. It consists of the keyword for, followed by parentheses that contain three expressions separated by semicolons:

for(j=0; j<15; j++)

These three expressions are the *initialization expression*, the *test expression*, and the *increment expression*, as shown in Figure 3.1.





Example 2:

Summing the Even Integers from 2 to 20.

```
// Summing integers with the for statement.
 #include <iostream>
using namespace std;


int main()
 {
int total = 0; // initialize total
// total even integers from 2 through 20
        for ( int number = 2; number <= 20; number += 2 )
        total += number;
        cout << "Sum is " << total << endl; // display results
} // end main
```

Output:

```
Sum is 110
```

Example 3:
Program that calculetes Compound Interest Calculations
Consider the following problem statement:

> A person invests $1000.00 in a savings account yielding 5 percent interest. Assuming that all interest is left on deposit in the account, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula for determining these amounts:
>
> $a = p (1 + r)n$
>
> where
>
> p is the original amount invested (i.e., the principal),
> r is the annual interest rate,
> n is the number of years and
> a is the amount on deposit at the end of the nth year.

The for statement (lines 21–28) performs the indicated calculation for each of the 10 years the money remains on deposit, varying a control variable from 1 to 10 in increments of 1. C++ does *not* include an exponentiation operator, so we use the standard library function pow (line 24). The function pow(x, y) calculates the value of x raised to the yth power. In this example, the algebraic expression (1 + *r*)*n* is written as pow(1.0 + rate, year), where variable rate represents *r* and variable year represents *n*. Function pow takes two arguments of type double and returns a double value.

```
 3   #include <iostream>
 4   #include <iomanip>
 5   #include <cmath> // standard C++ math library
 6   using namespace std;
 7
 8   int main()
 9   {
10      double amount; // amount on deposit at end of each year
11      double principal = 1000.0; // initial amount before interest
12      double rate = .05; // interest rate
13
14      // display headers
15      cout << "Year" << setw( 21 ) << "Amount on deposit" << endl;
16
17      // set floating-point number format
18      cout << fixed << setprecision( 2 );
19
20      // calculate amount on deposit for each of ten years
21      for ( int year = 1; year <= 10; ++year )
22      {
23         // calculate new amount for specified year
24         amount = principal * pow( 1.0 + rate, year );
25
26         // display the year and the amount
27         cout << setw( 4 ) << year << setw( 21 ) << amount << endl;
28      } // end for
29   } // end main
```

Output:

```
Year    Amount on deposit
   1              1050.00
   2              1102.50
   3              1157.63
   4              1215.51
   5              1276.28
   6              1340.10
   7              1407.10
   8              1477.46
   9              1551.33
  10              1628.89
```

A Caution about Using Type float or double for Monetary Amounts:

Lines 10–12 declare the double variables amount, principal and rate. We did this for simplicity because we're dealing with fractional parts of dollars, and we need a type that allows decimal points in its values. Unfortunately, this can cause trouble. Here's a simple explanation of what can go wrong when using float or double to represent dollar amounts (assuming setprecision(2) is used to specify two digits of precision when printing):

Two dollar amounts stored in the machine could be 14.234 (which prints as 14.23) and 18.673 (which prints as 18.67). When these amounts are added, they produce the internal sum 32.907, which prints as 32.91. Thus your printout could appear as

```
  14.23
+ 18.67
-------
  32.91
```

But a person adding the individual numbers as printed would expect the sum 32.90! You've been warned! In the exercises, we explore the use of integers to perform monetary calculations. [*Note:* Some third-party vendors sell C++ class libraries that perform precise monetary calculations.]

While loops:

The for loop does something a fixed number of times. What happens if you don't know how many times you want to do something before you start the loop? In this case a different kind of loop may be used: the while loop.

While loops are the simplest kind of loop. The basic structure is

```
while ( <condition> ) { Code to execute while the condition is true }
```

In fact, a while loop is almost exactly like an if statement, except that the while loop causes its body to be repeated. Just like an if statement, the condition is a Boolean expression. For example, here's a while loop with two conditions:

```
while ( i == 2 || i == 3 )
```

Here's a really basic example of a while loop:

```
while ( true )
{
cout << "I am looping\n";
}
```

Warning: if you run this loop, it will never stop! The condition will always evaluate to true. This is called an **infinite loop**. Because an infinite loop never stops, you have to kill your program to stop it (you can do this by either pressing Ctrl-C, Ctrl-Break or closing the console window). To avoid infinite loops, you should be sure your loop condition won't always be true.

A common mistake

Now is a good time to point that a common cause of infinite loops is using a single equal sign instead of two equal signs in a loop condition:

BAD CODE

```
int i = 1;
while ( i = 1 )
{
cin >> i;
}
```

This loop attempts to read inputs from the user until the user enters something other than 1. Unfortunately, the loop condition is

```
i = 1
```

Rather than

```
i == 1
```

The expression i = 1 will just assign the value of 1 to i. As it turns out, an assignment expression acts as if it also returns the value assigned—in this case, 1. Since 1 is not zero, it is true, so this loop will go on forever.

Example 1:

Let’s look at a loop that actually works well! Here’s a full program demonstrating while loops by displaying the numbers from 0 to 9:

```
#include <iostream>
using namespace std;
int main ()
{
int i = 0; // Don't forget to declare variables
while ( i < 10 ) // While i is less than 10
{
cout << i << '\n';
i++; // Update i so the condition can be met
eventually
}
}
```

Output:

The next example, ENDON0, asks the user to enter a series of numbers. When the number entered is 0, the loop terminates. Notice that there's no way for the program to know in advance how many numbers will be typed before the 0 appears; that's up to the user.

Example 2:

```
// demonstrates WHILE loop
#include <iostream>
using namespace std;
int main()
{
int n = 99; // make sure n isn't initialized to 0
while( n != 0 ) // loop until n is 0
cin >> n; // read a number into n
cout << endl;
return 0;
}
```

Essentials of Counter-Controlled Repetition:

This section uses the while repetition statement to formalize the elements required to perform counter-controlled repetition. Counter-controlled repetition requires

1. The name of a control variable (or loop counter)
2. The initial value of the control variable
3. The loop-continuation condition that tests for the final value of the control variable
4. (i.e., whether looping should continue)
5. The increment (or decrement) by which the control variable is modified each time through the loop.

Controlling the flow of loops:

While you normally decide to exit a loop by checking the loop condition, sometimes you want to exit out of the loop early. C++ has just the keyword for you: **break**. A break statement will immediately terminate whatever loop you are in the middle of.

Here's an example that uses break to end what would otherwise an infinite loop, a basic rewrite be of the password example code:

Example 3:

```
#include <string>
#include <iostream>
using namespace std;
int main ()
```

```
{
string password;
while ( 1 )
	{
	cout << "Please enter your password: ";
	cin >> password;
	if ( password == "foobar" )
		{
			break;
		}
	}
cout << "Welcome, you got the password right";
}
```

A break statement immediately ends the loop, jumping to the closing brace. In this example, once the correct password is entered, the loop terminates. Because the break statement can appear anywhere in the loop, including at the very end, you can use infinite loops as an alternative way of writing a dowhile loop, as we did here.

Break statements are useful when you need an escape route from within a large loop, but too many break statements can make your code hard to read. A second way of controlling loops is to skip a single iteration by using continue. When the continue statement is hit, the current loop iteration ends early, but the loop is not exited. For example, you could write a loop that skips printing out the number 10:

```
int i = 0;
while ( true )
{
	i++;
	if ( i == 10 )
		{
		continue;
		}
	cout << i << "\n";
}
```

Here, the infinite loop will never end, but when i reaches 10, the continue statement will cause it to jump back to the starting line of the loop, skipping the call to cout. The loop condition will still be tested, though. When using continue with a for loop, the update step occurs immediately after the continue.

The continue statement is most useful when you want to skip some code in the middle of the body of a loop. For example, you might do some checks on a user's input, and if they enter something wrong, you can skip processing that input with a loop structure that looks like this:

```
while ( true )
{
cin >> input;
	if ( ! isValid( input ) )
	{
```

```
            continue;
            }
    // go on to process the input as normal
    }
```

Assignment:

1. **Write C++ statements to create truth table for && (logical AND) operator.**
2. **Write C++ statements to check the entered interger is Prime Number or not.**
3. **Write C++ statements to calculate factorial of entered numbers.**
4. **Write C++ statements that prints access granted if username and password is correct, otherwise ask user input again and again till the correct username and password entered.**