Iteration

In this lecture we will be primarily looking at controlling the flow of the program using loops and also at I/O (input/output).
Last week I mentioned the 3 methods of controlling the flow of a program:

- statements may only be obeyed under certain conditions (choice),
- statements may be obeyed repeatedly (loops),
- a group of remote statements may be obeyed (subroutines).

We looked at choice control, now we are looking at loop control or the ability to execute a series of statements repeatedly.

The Lingo

Iteration = causing a set of statements (the body) to be executed repeatedly.

Block (Compound Statement)
We have covered statements before, and we briefly mentioned statement blocks (in lecture 2 if you don't remember).
Last week, and especially this week, we see code that is in between two parentheses (brackets). This type of bracket { }, it is known as block (or compound statement).

A block is a sequence of zero or more statements enclosed by a pair of curly braces { }

SYNTAX

```
{
     Statement (optional)
     .
     .
     .
}
```

Loops

What is a loop?
When we talk about loops we're talking about a code structure that executes a piece of code either none, one or multiple times depending on a condition.

A loop is a repetition control structure.
It causes a single statement or block to be executed repeatedly.

Two Types of Loops

- Count controlled loops repeat a specified number of times.
- Event-controlled loops some condition within the loop body changes and this causes the repeating to stop.

C++ provides three control structures to support iteration (or looping): **while**, **do-while** and **for**.

Before considering specifics we define some general terms that apply to any iteration construct:

The Body - statement or block of statements we wish to be executed under certain conditions
Pass (or iteration) - one complete execution of the body
Loop entry - the point where flow of control passes to the body
Loop test - the point at which the decision is made to (re)enter the body, or not
Loop exit - the point at which the iteration ends and control passes to the next statement after the loop
Termination condition - the condition that causes the iteration to stop

When designing a loop, it is important to clearly identify each of these. Understanding the termination condition and what guarantees it will eventually occur, are particularly vital.

While loops

While loops cause your program to repeat a sequence of statements as long as the starting conditions remains true.

SYNTAX

```
while ( Expression )
{

    // loop body

}
```

NOTE: Loop body can be a single statement, a null statement, or a block.
NOTE: The expression is Boolean, it must return true or false

The most versatile loop construct in C++ is the while statement.
The Boolean **expression** must be enclosed in parentheses ( ), and the **loop body** can be either a single statement or a compound statement.

The Boolean expression is examined before each pass through the body of the loop. If the Boolean expression is true, the loop body is executed; if it is false, execution proceeds to the first statement after the loop body.

The expression tested by a while loop can be as complex as any legal C++ expression and like the if-then-else statements can contain relational and logical (&& || !) operators to increase the complexity:

```
while(countx < 7 || county > 11) // test expression
{
      // body of loop
}
```

Examples

```
int counter = 0;  // Initialise condition variable

while(counter <  5)     // Test whether condition still true
{     // Body of loop
      counter++;
      cout << “Counter: “ << counter << endl;
}
```

This is a count-controlled loop.
The above example says that whilst counter (which starts at zero) is less than 5, add one to the counter and print out the result.

It demonstrates the fundamentals of a while loop. A condition is tested, if it is true the body of the while loop executes. When the condition is no longer true, when counter is no longer less than 5 the body of the while loop is skipped and the program moves on.

```
int numAst;
int numPrinted = 0;

cout << "How many asterisks do you want? ";
cin >> numAst;

while (numPrinted < numAst)
{
      cout << '*';
      numPrinted++;
}
```

This is a count-controlled loop. The loop control variable (LCV) is a variable whose value determines whether the loop body is executed.

In this example the LCV is the variable `numAst`. This number inputted by the user determines how many asterisks, if any, will be printed.

Some observations:

- It is possible that the body of the loop will never be executed.
- It is possible that the loop test condition will become false during iteration. Even so, the remainder of the loop body will be executed before the loop test is performed and iteration stops.
- The loop test must involve at least one variable whose value is (potentially) changed by at least one statement within the loop body.

Event-controlled Loops

There are 3 types of event controlled loops:

- Sentinel controlled keep processing data until a special value which is not a possible data value is entered to indicate that processing should stop
- End-of-file controlled keep processing data as long as there is more data in the file
- Flag controlled keep processing data until the value of a flag changes in the loop body (when we use flag, remember we mean bool – true or false)

Examples:

- Count controlled read in 25 values from a user
- Sentinel controlled read in values until a impossible value (or character is entered) such as -1 is selected when reading in pulse rates
- End-of-file controlled read in a file until the end-of-file is found
- Flag controlled read in values until the users chooses to quit (by pressing q)

Count-controlled Loop

A loop that terminates when a counter reaches some limiting value

The LCV (loop control variable) will be:

- An integer variable used in the Boolean expression
- Initialized before the loop,
- Incremented or decremented within the loop body

```
int count ;
count = 4;  // initialize loop variable

while (count > 0) // test expression
{
      cout << count << endl ; // repeated action
      count -- ; // update loop variable
}
```

This loop is terminated when variable `count` becomes 0

Event-controlled Loop

A loop that executes until a specified situation arises to signal the end of the iteration.

```
bool bRunning = true;   // initialise loop flag
char cQuit;

while(bRunning)   // test expression
{
      cout << "Want do it again? press q to quit\n";
      cin >> cQuit;     // get user input

      if(cQuit == 'q' || cQuit == 'Q')
      {
            bRunning = false;
      }
}
```

This loop is an example of a flag controlled event loop, it is terminated when the user presses q

Sentinel-controlled Loop

A loop that terminates when a specified marker (dummy data value) is read, signalling the end of the iteration.

```
const string SENTINEL = "Nameless"; // the specified sentinel

string name;

while (name != SENTINEL) // test expression
{
      cout << "what is your name?\n";
      cin >> name;      // get user input
}
```

Infinite Loops

One thing to be wary about when coding loops, especially while and do-while, is making sure any conditions will be met or that the expression will be met after an finite amount of time.

Infinite loops are examples of logical errors and can cause your program to hang.

Last week we looked at `if` statements and the assignment operator (=). Using = instead of == in a while loop will cause an infinite loop to occur and your program to hang.

An example of an infinite while loop is:

```
while(1)    // 1 is always true
{
        // print out some stuff
}
```

Loop Design Considerations

Questions that one should consider carefully when coding a loop:

- What is the condition that terminates the loop?
- How should the condition be initialized?
- How should the condition be updated?
- What guarantees this condition will eventually occur?
- What is the process to be repeated?
- How should the process be initialized?
- How should the process be updated?
- What is the state of the program on exiting the loop?
- Are "boundary conditions" handled correctly?

Do-While loops

So we know that while loops check the condition before executing any of the statements, if the condition evaluates false the body of the while loop is skipped.
So what does a do while loop do?
The do-while loops execute the body of the loop before its condition is evaluated. This ensures that the body of loop is executed at least once before the loop exits.

SYNTAX

```
do
{
        // loop body

} while ( Expression ) ;
```

Loop body statement can be a single statement or a block.

Do-While

- POST-TEST loop (exit-condition)

- The looping condition is tested after executing the loop body.
- Loop body is always executed at least once.

While

- PRE-TEST loop (entry-condition)
- The looping condition is tested before executing the loop Body.
- Loop body may not be executed at all.

Example:

```
int   counter;          // Initialise counter variable

/* Get user input */
cout << “How many times?” << endl;

cin >> counter;

do
{     // execute statements at least once.
      cout << counter << “ time \n”;
      counter--;
} while(counter > 0);
```

The program asks the user how many times he wants to print out time. It will execute at least once even if the input is zero.

For loops

Is basically a count controlled loop.

The `for` loop combines the three steps of initialisation, test and increment into one initial statement.
A `for` loop consists of the keyword for followed by a set of parentheses that contain three statements separated by semi-colons.

SYNTAX

```
for ( initialization ; test expression ; update )
{
      // loop body
}
```

The for loop contains

- An initialization

- An expression to test for continuing
- An update to execute after each iteration of the body

Note that in a `for` loop:

- The *exit condition* of a `for` loop could be any valid condition - it does not need to be related to or involve the loop counter.
- The loop is exited when the exit condition evaluates to false or zero.
- The terminating condition is evaluated before the loop body is executed.
- In addition, the *increment/decrement statement* (for example, *i*++) in the `for` loop is executed **after** each loop iteration.

The *increment/decrement statement* could be a statement to increment or decrement any valid C++ variable - it does not need to be related to or involve the loop counter.

Example:

```
int x;      // initialise variable for the loop

for(x = 0; x<100; x++)
{
        /* Keep in mind that the loop condition checks
        the conditional statement before it loops again.
        consequently, when x equals 100 the loop breaks */

        cout<<x<<endl;           // Outputting x
}
```

For loops are used whenever the number of times a loops needs to execute is known or can be calculated beforehand.

```
for (initial expression; test expression; update expression)
{
      <statement>
}
```

The **initial** expression is performed just once, prior to loop execution. The initial expression may declare variables as well as specify initializations for them.
The **test** expression is evaluated and if false then the next statement after the `for` loop is executed.
If the test expression evaluates to true, then the <statement> of the `for` loop is executed, the update expression is performed, and test expression is evaluated again.
For loops carry out the following process in sequence:

1) Performs the operations in the initialisation
2) Evaluates the condition
3) If the condition is true it executes the loop and then the action or modification statement.

After the initial test only steps 2 and 3 are repeated on subsequent loops.

Because of these three independent statements for loops are powerful and flexible.

Multiple conditionals can be declared at the same time, we don't however have to use the logical operators to include these in the `for` loop.
Example:

```
for( i = 0, j = 0; i < 3; i++, j++)
```

`for` loops can also contain NULL statements.

First what are NULL statements: The C++ Null Statement
The null statement is an expression statement with the expression missing. It is useful when the syntax of the language calls for a statement but no expression evaluation. It consists of a semicolon.

Null statements are commonly used as placeholders in iteration statements or as statements on which to place labels at the end of compound statements or functions.

Example:

```
for( ; counter < 5 ; )
{
      counter++
      cout << "Looping \n";
}
```

NOTE: In the above example the for loop acts as a while loop.

******The best way to learn about loops in by coding them and running examples, so see the tutorial for code examples and exercises.***

Nested Loops

We covered nested if-else statements last week, this week we are looking at loops and so we cover nested loops.

Just like if-else statements loops can contain valid statements in their body, this includes if-else statements or other loops.
Loops can be nested, with one loop sitting in the body of the other.

The inner loop will be executed for every execution of the outer loop.

Example:

```
int rows, columns;
char theChar;

cout << "How many rows? ";
cin >> rows;
cout << "How many cloumns? ";
cin >> columns;
cout << "What character? ";
cin >> theChar;

for(int i = 0; i < rows; i++)
{
      for(int j = 0; j < columns; j++)
      {
            cout << theChar;
      }
      cout << '\n';
}
```

This prints out `theChar` a row at a time.
For each pass of the initial `for` statement the nested `for` statement is run to completion, it is set back to zero and then counts up.

Goto Statements

The goto statement was once widely used. It can cause a jump to any location within your source code which can make for difficult to read code. Unfortunately code that used goto extensively was poorly structured and could easily become unmanageable, and is known as "spaghetti code".

Today, programming languages provide other ways to structure code so goto is rarely needed. In C++ it is always possible to use something other than goto to achieve what you need. So I am going to briefly cover goto's for the sake of completion, but you shouldn't really need to use them.

Note that the use of goto is almost a religious issue and has provoked a great deal of debate. In C++ there is always a better alternative for writing loops, so you needn't use goto at all.

The goto statement performs an unconditional transfer of control to the named label. The label must be in the current function.

Syntax

```
goto label:
...
label: statement
```

A statement label is meaningful only to a goto statement; in any other context, a labelled statement is executed without regard to the label. What this means is that the label is ignored in regard to executing code, but the statement is still run when execution passes in to in the normal way.

This example demonstrates the goto statement:

```
#include <stdio.h>
void main()
{
    int i, j;

    for ( i = 0; i < 10; i++ )
    {
        printf( "Outer loop executing. i = %d\n", i );
        for ( j = 0; j < 2; j++ )
        {
            printf( " Inner loop executing. j = %d\n", j );
            if ( i == 3 )
                goto stop;
        }
    }
    /* This message does not print: */
    printf( "Loop exited. i = %d\n", i );
    stop: printf( "Jumped to stop. i = %d\n", i );
}
```

In this example, a goto statement transfers control to the point labelled stop when i equals 3.

Coding Practice and Programming Style

This is just a brief mention, online there is a larger document that you can refer to. The notes are there to provide guidelines for internal documentation and style. Although they are intended for student programmers, style skills carry over into professional life after university.

The point of a style guide is to greater uniformity in the appearance of source code. The benefit is enhanced readability and hence maintainability for the code.

The essence of good programming style is communication. Good style in programming is roughly as difficult to learn as good style in English. In both cases, the document has no value if it does not convey its meaning to the reader. Any program that will be used must be maintained by somebody - and that somebody must be able to understand the code by reading it. Any program that needs debugging will be easier to debug if the creator carefully explains what's going on.

Within the program text, programmers have three primary tools for communicating their intentions:

- comments (explanation for the program);
- clear variable names, constants, expressions and subroutines (the words of the program itself);
- and white space and alignment (the organization of the words in the program).

Each of these aspects aid communication between the program writer and the program reader, which includes the program writer at debug time - so you as a program writer have a stake in good style too.

General Guidelines

The most useful things to know about program documentation are ”what'' and “when''. In general, you should include comments explaining what every subroutine does, what every variable does, and an explanation for every tricky expression or section of code. But, there is much too good style beyond comments, as we shall see. “When'' is easy - comment it before you write it. Whenever you declare a variable, include a comment. Whenever you start to write a subroutine, first write a comment explaining it.

The most important rule of style is: be consistent! If you adopt some method for variable naming or indenting, stick to it throughout your program.

Note

That commenting and programming style does earn marks in regards to the assessments