MECH215
Lecture Notes 2

Control Structures

Up until now, all of the programs we have seen were examples of STRAIGHTLINE programming. That is, program execution commenced with the first statement and proceeded statement by statement until the end of the program. Most non-trivial programs require some sort of mechanism to alter the sequence of statement execution. C++ has two conditional constructs and three looping constructs. The conditional constructs determine which statements are to be executed depending upon a certain condition, the looping constructs determine the number of times a block of statements is to be executed. More on this later.

The Boolean Type

In the C programming language a LOGICAL FALSE was represented by the integer value 0, a LOGICAL TRUE was represented by any NON-ZERO integer. Thus in C (and early versions of C++) the following were all legal ways of representing a logical true:

1
5
678*3

Similarly, the following are legal ways of expressing a logical FALSE

0
16 % 16

C++ now has the built-in BOOLEAN logical data type which is named bool. The bool data type can take on one of two possible symbolic constants: true, false.

Relational Operators

C++ has two type of relational operators: equality and ordering. The equality operators are:
== evaluates to true if both of the operands are equal to each other
!= evaluates to true if the two operands are not equal to each other

Some examples of the use of the equality operators are:

```
int i = 1;
int j = 1;
int k = 2;

(i==j); // this evaluates to true
(i==k); // this evaluates to false
(i!=k); // this evaluates to true
```

We are not limited to comparing only integers for equality or inequality. Any built-in C++ data type can be used (when working with floating point values we
should be a bit more careful... more on this later).

```
char letter1 = 'a';
char letter2 = 'b';
char letter3 = 'c';

letter1 == letter2;  // this evaluates to false;
letter2 != letter3;  // this evaluates to true;
```

What do you think the output of the following program which tests two structs for equality will be?

```
// Author: Ted Obuchowicz and John Peach
// structEquality.cpp
// Example program illustrating use of equality operators on struct
data type

#include <iostream>
#include <string>

using namespace std;

int main() {

    struct simpleStruct {
        int  number;
        char letter;
        bool isLogical;
    };

    simpleStruct firstStruct, secondStruct, thirdStruct;

    firstStruct.number      = 1;
    firstStruct.letter      = 'a';
    firstStruct.isLogical   = true;

    secondStruct.number     = 1;
    secondStruct.letter     = 'a';
    secondStruct.isLogical  = true;

    thirdStruct.number      = 5;
    thirdStruct.letter      = 'a';
    thirdStruct.isLogical   = true;
```

```
    if (firstStruct == secondStruct) {
        cout << "first and second structs are equal" << endl;
    } else {
        cout << "first and second structs are not equal" << endl;
    }

    return 0;
}
```

The answer is nothing! The program DOES NOT EVEN COMPILE :

```
#>g++ -o structEquality structEquality.cpp
structEquality.cpp: In function ‘int main()’:
structEquality.cpp:32:24: error: no match for ‘operator==’ in
‘firstStruct == secondStruct’
```

The compiler is complaining about line 24 in the source code which is the if statement:

```
if (firstStruct == secondStruct)
```

If you want to compare structs, then the comparison must be done on a field-by field basis. For example, the above program can be fixed by simply replacing
the if statement with something along the lines of:

```
if (firstStruct.number    == secondStruct.number &&
    firstStruct.letter    == secondStruct.letter &&
    firstStruct.isLogical == secondStruct.isLogical) {
  cout << "first and second structs are equal" << endl;
} else {
  cout << "first and second structs are not equal" << endl;
}
```

Ordering Operators

C++ uses the following 4 ordering operators to determine the relative size of two operands:

< less than
> greater than
<= less than or equal
>= greater than or equal

Examples:

```
( 1 < 2 )  // evaluates to true
( 1 > 2 )  // evaluates to false
```

Boolean Operators

C++ has three Boolean operators: &&, ||, and !. Their meaning are as follows:

&& : used to perform a logical AND of two boolean expressions:

ex. (i < 3) && (k < 56)

|| : used to perform a logical OR of two expressions:

ex. (i < 2) || (k > 45)

! : used to perform a logical NEGATION

ex. ! (i < 2)

Short-Circuit Evaluation

When evaluation the value of a logical expression, C++ requires that first the left operand be evaluated to yield a boolean value, then the right operand be evaluated. Furthermore, if the value of the entire logical expression may be determined from the left operand, then the right hand operand should not be evaluated. This is known as short-circuit evaluation.

For example:

```
( i != 0) && ( (k/i) < 2 )
```

The (i != 0) will be equal to false only in the case of i being equal to 0, since the boolean && operator requires both its operands to be true in order for the entire to be expression to be true, it is not even necessary to evaluate the right-hand side ((k/i) < 2) in the case of i being equal to 0. By virtue of short-circuit evaluation, we can guarantee that division by zero will NEVER occur. If C++ did not provide short-circuit evaluation, then there would be a possibility that a division by zero would be attempted while evaluation the right hand side operand.

Comparing Floating Point Numbers

Round-off errors due to inherent limitations of how real numbers are internally stored in a computer's memory can cause some problems when comparing floating point numbers for equality. For example, consider the following program which adds 0.0000001 ten times to theoretically give the value 0.000001:

```
// Author: Ted Obuchowicz and John Peach
// equalityFloatingPoint.cpp
// example program illustrating use of testing real numbers for
equality

#include <iostream>
#include <string>
```

```
#include <iomanip>

using namespace std;

int main() {

    float number;

    // Sum 0.0000001 (6 zeros after the decimal place) 10 times.
    // This should equal 0.000001 (5 zeros after the decimal place).
    number =   0.0000001 + 0.0000001 + 0.0000001 + 0.0000001
             + 0.0000001 + 0.0000001 + 0.0000001 + 0.0000001
             + 0.0000001 + 0.0000001;

    if (number == 0.000001 ) {
       cout << "number is equal to 0.000001" << endl;
    } else {
       cout << "number is not equal to 0.000001" << endl;
    }

    cout << setprecision(15) << "number is equal to " << number
         << endl;

    return 0;
}
```

The output may surprise you:

```
number is not equal to 0.000001
number is equal to 9.99999997475243e-07
```

When testing real number for equality, it is a better practice to check that the two numbers are close enough to one another. The following program makes use of the fabs() function found in the <math> library to check that the absolute value of number - 0.000001 is less than some specified value called epsilon:

```
// Author: Ted Obuchowicz and John Peach
// equalityFloatingPointEpsilon.cpp
// example program illustrating use of testing real numbers for
equality

#include <iostream>
#include <string>
#include <iomanip>
#include <math.h>
```

```
using namespace std;

int main() {

    float number;
    const float EPSILON = 0.0000001;

    // Sum 0.0000001 (6 zeros after the decimal place) 10 times.
    // This should equal 0.000001 (5 zeros after the decimal place).
    number =   0.0000001 + 0.0000001 + 0.0000001 + 0.0000001
             + 0.0000001 + 0.0000001 + 0.0000001 + 0.0000001
             + 0.0000001 + 0.0000001;

    if  (  fabs( (number - 0.000001) <= EPSILON  )  ) {
       cout << "number is equal to 0.000001" << endl;
    } else {
       cout << "number is not equal to 0.000001" << endl;
    }

    cout << setprecision(15) << "number is equal to " << number
         << endl;

    return 0;
}
```

This time the program output is what we expect:

```
number is equal to 0.000001
number is equal to 9.99999997475243e-07
```

The if Statement

The previous programs have already examined the use of the IF statement. The if statement comes in two forms:

Form 1:

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

Form 2:

```
if (expression)
        statement;
else
        another statement;
```

NOTE: If it is required to perform more than one statement in a branch of the if statement, then the

entire block of statements must be enclosed in { } such as:

Form 1:

```
if (expression) {
        statement1;
        statement2;
        ...
        statementn;
};
```

Form 2:

```
if (expression) {
        statement1;
        statement2;
        ...
        statementn;
} else {
        statementx;
        statementy;
        ...
        statementz;
};
```

It is considered poor programming style to omit the { } even if there is only one statement. Therefore, ALWAYS use { }. Be careful to use the delimiting { } properly. Failure to do so may cause strange program behaviour as is the following program:

```
// Author: Ted Obuchowicz and John Peach
// poorlyBlockedIf.cpp
// example program illustrating INCORRECT use of if statement

#include <iostream>
#include <string>

using namespace std;

int main() {

    int userInt;

    // Get an integer from the user
    cout << "Enter an integer: ";
    cin >> userInt;

    // If the user did not enter 1 then print a message
    if (userInt != 1)
```

```
        cout << "i is not equal to 1"    << endl;
        cout << "end of program goodbye" << endl;

    return 0;
}
```

When we run this program and enter 1 for the value of 1, the output is:

```
end of program goodbye
```

What went wrong? The expression (i != 1) is false for i equal to 1 (since 1 is equal to 1). So why was the "end of program goodbye" produced as output? . What the program actually did was:

```
if (i != 1)
     cout << " i is not equal to 1" << endl;

cout << " end of program goodbye" << endl;
```

The second cout is actually independent of the first if since we did not "block" the two statements within { }.

If we change the program to:

```
// Author: Ted Obuchowicz and John Peach
// properlyBlockedIf.cpp
// Example program illustrating INCORRECT use of if statement

#include <iostream>
#include <string>

using namespace std;

int main() {

    int userInt;

    // Get an integer from the user
    cout << "Enter an integer: ";
    cin >> userInt;

    // If the user did not enter 1 then print a message
    if (userInt != 1) {
        cout << "i is not equal to 1"    << endl;
        cout << "end of program goodbye" << endl;
    }
    return 0;
}
```

A sample session with the running program is:

```
#>properlyBlockedIf
1

#>properlyBlockedIf
3456
i is not equal to 1
end of program goodbye
```

Caveat

The following is a VERY COMMON MISTAKE THAT BEGINNER C and C++ programmers make:

```
// Author: Ted Obuchowicz and John Peach
// balancedOwedError.cpp
// example program illustrating use of common programming error with
an if

#include <iostream>
#include <string>

using namespace std;

int main() {

    int balanceOwed;

    cout << "Enter the balanced owed: ";
    cin >> balanceOwed;

    if (balanceOwed = 0) {
        cout << "You owe me nothing" << endl;
    } else {
        cout <<  "You owe me " << balanceOwed << " dollars" << endl;
    }

    return 0;
}
```

This program when run will ALWAYS print out:

```
#>balancedOwedError
Enter the balanced owed: 123
You owe me 0 dollars
```

Irrespective of the value the user inputs for the integer variable balanceOwed!
Why? Look very carefully at the if statement:

```
if (balanceOwed = 0) {
    cout << "You owe me nothing" << endl;
} else {
    cout <<  "You owe me " << balanceOwed << " dollars" << endl;
}
```

the (balanceOwed = 0) is an ASSIGNMENT of 0 to the variable balanceOwed. The result of the assignment is that variable balanceOwed take son the value
0. The condition then reduces to:

```
if (0) {
    cout << "You owe me nothing" << endl;
} else {
    cout <<  "You owe me " << balanceOwed << " dollars" << endl;
}
```

Since a 0 is false in C++, the condition is always false so the else part is always executed with a value of 0 for the amount owed!!!

The proper version of the program is:

```
// Author: Ted Obuchowicz and John Peach
// balancedOwed.cpp

#include <iostream>
#include <string>

using namespace std;

int main() {

    int balanceOwed;

    cout << "Enter the balanced owed: ";
    cin >> balanceOwed;

    if (balanceOwed == 0) {     // This line has been corrected
        cout << "You owe me nothing" << endl;
    } else {
        cout <<  "You owe me " << balanceOwed << " dollars" << endl;
    }
```

```
    return 0;
}
```

Note that the only difference between the good and the bad versions is the difference between = and ==. Amazing what a single missing = will do in a program! PAY YOUR ATTENTION INSIDE OF IF STATEMENTS AND ALWAYS CHECK FOR EQUALITY WITH THE ==

The proper output is:

```
#>balancedOwed
Enter the balanced owed: 0
You owe me nothing
#>balancedOwed
Enter the balanced owed: 456
You owe me 456 dollars
```

Nested if-else-if Statements

Consider the following if construct:

```
char command;
cin >> command;

if (command == 'u') {
    cout << "Move up command was received" << endl;
} else {
    if (command == 'd') {
        cout << "Move down command was received" << endl;
    } else {
        if (command == 'l') {
            cout << "Move left command was received" << endl;
        } else {
            if (command == 'r') {
                cout << "Move right command was received" << endl;
            } else {
                cout << "Invalid command" << endl;
            }
        }
    }
}
```

The indentation of the if statements is purely cosmetic. The compiler does not care about nesting. We can rewrite the above if statement as:

```
if (command == 'u') {
    cout << "Move up command was received" << endl;
} else if (command == 'd') {
    cout << "Move down command was received" << endl;
} else if (command == 'l') {
    cout << "Move left command was received" << endl;
} else if (command == 'r') {
    cout << "Move right command was received" << endl;
} else {
    cout << "Invalid command" << endl;
}
```

Using this style improves program readability and avoids the problem of running out of space on the right hand side as the nesting level of the if statements increases. Readability is extremely important as the cost of maintaining code very high.

The switch Statement

C++ has a very convenient statement for multi-branch conditional testing as in the above statement. It is called the switch statement. The above if can be rewritten as a switch statement.

```
// Author: Ted Obuchowicz and John Peach
// switch.cpp
// Example program illustrating use of switch statement

#include <iostream>
#include <string>

using namespace std;

int main() {

    char command;
    cout << "Enter a command: ";
    cin >> command;

    switch (command) {
        case 'u':
            cout << "Move up command was received" << endl;
            break;
        case 'd':
            cout << "Move down command was received" << endl;
            break;
        case 'l':
            cout << "Move left command was received" << endl;
```

```
            break;
        case 'r':
            cout << "Move right command was received" << endl;
            break;
        default :
            cout << "Invalid command" << endl;
    }

    return 0;
}
```

When the switch statement is executed, the value of command is compared with the specified cases. If a match is found, the statements corresponding to the matching case are executed. If the value of command does not match any of the specified cases, then the sequence of statements for the specified default case is executed (if there is no default case specified, flow of control resumes with the statement immediately following the switch).

The break Statement

Note that in the above switch, there is a break statement associated with every specified case. The break statement causes a change of flow of control to the statement following the switch. The break can be interpreted as "the switch statement has completed its task and program execution should resume with the statement immediately following the switch statement".

If a switch statement does not contain breaks for the specified cases, then the action for the FIRST matching value is executed, FOLLOWED BY ANY REMAINING groups of action statements (for the remaining values even though they do not match).

For example:

```
// Author: Ted Obuchowicz and John Peach
// switchNoBreak.cpp
// Example program illustrating use of switch statement

#include <iostream>
#include <string>

using namespace std;

int main() {

    char command;
    cout << "Enter a command: ";
    cin >> command;
```

```
    switch (command) {
        case 'u':
            cout << "Move up command was received" << endl;
        case 'd':
            cout << "Move down command was received" << endl;
        case 'l':
            cout << "Move left command was received" << endl;
        case 'r':
            cout << "Move right command was received" << endl;
        default :
            cout << "Invalid command" << endl;
    }

    return 0;
}
```

If the user supplies u for the value of the command, the program produces the following output:

```
#>switchNoBreak
Enter a command: u
Move up command was received
Move down command was received
Move left command was received
Move right command was received
Invalid command
```

If we supply d for the value of the input variable command, the following output is produced:

```
#>switchNoBreak
Enter a command: d
Move down command was received
Move left command was received
Move right command was received
Invalid command
```

Giving l as an input value results in:

```
#>switchNoBreak
Enter a command: l
Move left command was received
Move right command was received
Invalid command
```

if we give r as the value :

```
#>switchNoBreak
Enter a command: r
Move right command was received
Invalid command
```

cases In A switch With No break

In most programming problems, every 'action' statement should have an associated break statement:

```
switch (condition) {
   case value1:
      action1;
      break;
   case value2:
      action2;
      break;
   ...
   case valuen:
      actionn;
      break;
   default:
      default_action;
}
```

Note: This is not C++ code. It represents the form of the statements but not actual C++ statements.

In some programming situations, using the so-called "fall through" behaviour (some actions without an associated break) makes sense. Consider once again the menu switch where we now consider the upper and lower case letters as being valid input characters (i.e. both 'u' and 'U' should be recognized as a legal move up command, 'd' and 'D' as legal move down commands etc). We can rewrite our previous program as:

```
// Author: Ted Obuchowicz and John Peach
// switchFallThrough.cpp
// Example program illustrating use of switch statement

#include <iostream>
#include <string>

using namespace std;

int main() {

    char command;
    cout << "Enter a command: ";
```

```
    cin >> command;

    switch (command) {
        case 'u':
        case 'U':
            cout << "Move up command was received" << endl;
            break;
        case 'd':
        case 'D':
            cout << "Move down command was received" << endl;
            break;
        case 'l':
        case 'L':
            cout << "Move left command was received" << endl;
            break;
        case 'r':
        case 'R':
            cout << "Move right command was received" << endl;
            break;
        default :
            cout << "Invalid command" << endl;
    }

    return 0;

}
```

To reiterate, the switch statement is different from similar statements in languages such as Pascal in a very important way. In C and C++, when a program jumps to a certain line in a switch statement (when it finds the first matching value of the listed cases), the program THEN SEQUENTIALLY EXECUTES all the statements following that line unless you EXPLICITLY TELL OTHERWISE (with the break) . Program execution DOES NOT AUTOMATICALLY stop at the next case. To make execution stop at a particular line, you must use the break statement. The break causes the program to continue exection from the next statement following the switch.

One final note regarding the switch statement. Both the expression which is being tested and the values of the cases should be of an INTEGRAL type. This limitation means that a switch statement cannot be used to determine the flow of program control based on the value of a floating-point variable.

Looping Constructs

Suppose we wish to find the sum , and average value of a list of 4 numbers. Our first (brute-force) program might look like:

```
// Author: Ted Obuchowicz and John Peach
// averageBruteForce.cpp
// example program illustrating use of brute force approach to
finding the sum and average of 5 floating
// point values

#include <iostream>
#include <string>

using namespace std;

int main() {

    float sum;          // sum of all the entered numbers
    float average;      // average of all the entered numbers
    float number1;      // User defined value
    float number2;      // User defined value
    float number3;      // User defined value
    float number4;      // User defined value
    float number5;      // User defined value

    // Get the numbers from the user
    cout << "Please enter first number ";
    cin >> number1;

    cout << "Please enter second number ";
    cin >> number2;

    cout << "Please enter third number ";
    cin >> number3;

    cout << "Please enter fourth number ";
    cin >> number4;

    cout << "Please enter fifth number ";
    cin >> number5;

    // Calculate the sum
    sum = number1 + number2 + number3 + number4 + number5;

    // Find the average
```

```
    average = sum / 5.0;

    // Tell the user the results
    cout << "The sum of the 5 numbers is : " << sum << endl;
    cout << "The average value is : " << average << endl;

    return 0;
}
```

Suppose we had to find the sum and average of a list of 10 000 values? Clearly, the above brute force approach is not very convenient to use.

We can use a while loop to control the number of times a portion of program code is to be executed. The following is an ALGORITHM which repeatedly gets the next input value and adds it to the sum of the previous input values and then calculates the average:

Step 1: Set the value of the running sum to 0.0
Step 2: Set the value of the number of processed values to 0.
Step 3: while (number of processed items < sixe of the list)
    Step 3a: read in a value;
    Step 3b: sum = sum + value;
    Step 3c; add 1 to the number of items processed so far
Step 4: calculate the average as sum / list size
Step 5: display the values of the sum and the average

A complete C++ program which corresponds to the above pseudo-code is:

```
// Author: Ted Obuchowicz and John Peach
// averageWhile.cpp
// Example program illustrating use of while loop to find sum and
// average of 5 floating point values read in from keyboard

#include <iostream>
#include <string>

using namespace std;

int main() {

    float sum = 0.0;            // sum of all the entered numbers
    float average;              // average of all the entered numbers
    const int NUM_OF_VALUES = 5;// The number of values that are
                                // going to be read
    float number;               // Current value entered by the user
    int itemsProcessed = 0;     // The number of values read in
```

```
    while (itemsProcessed++ < NUM_OF_VALUES) {
        cout << "Please enter a number ";
        cin >> number;          // get the number from the user
        sum += number;          // update running sum
    }

    average = sum / NUM_OF_VALUES;

    cout << "The sum is " << sum << endl;
    cout << "The average is "  << average << endl;

    return 0;
}
```

Improving the above program:

The above program is fine for lists whose sizes is known ahead of time. What if we wanted to find the sum and average of an unknown number of values? We can modify the above program to test the number which is input to see if it is a special "flag" (in computer programming a flag variable is one which the programmer places "special meaning" to). For example, we can reserve the number -9999 to indicate that the list has no more values. The modified program is :

```
// Author: Ted Obuchowicz and John Peach
// averageUnlimited.cpp
// Example program illustrating use of while loop to find sum and
// average of an unknown number of values read in from the keyboard

#include <iostream>
#include <string>

using namespace std;

int main() {

    float sum           = 0.0;      // sum of all the entered numbers
    float average;                  // average of all the numbers
    const float FLAG    = -9999.0;  // sentential value to stop
    const float EPSILON = 0.00001;  // error tolerance
    float number;                   // Current value entered
    int itemsProcessed  = 0;        // The number of values read in

    // get the first value
    cout << "Please enter a number ";
    cin >> number;
```

```
    // Loop until the user enters the value of flag
    while ( ! ( (number - FLAG) < EPSILON ) ) { // test to see if
                                                // FLAG was entered

        sum += number;              // update running sum
        itemsProcessed++;           // increment number of items
        // Get the next value
        cout << "Please enter a number ";
        cin >> number;

    }

    // Check to make sure that some values were entered
    if (itemsProcessed) {
        average = sum / itemsProcessed;
        cout << "The sum is " << sum << endl;
        cout << "The average is " << average << endl;
    } else {
        cout << " No numbers were entered " << endl;
        cout << " Sum is " << sum << endl;
        cout << " Impossible to compute average... sorry " << endl;
    }

    return 0;
}
```

Some comments on the above program:

1) Note that before we enter the while loop, we first read in a number... this is necessary so that the variable called number has an intial value when it is being tested inside the conditional expression of the while loop:

```
cout << "Please enter a number ";
cin >> number;

while ( ! ( (number - FLAG) < EPSILON ) )
             -----
               |
                -----> this must have some value before it can be tested
```

2) The test in the while loop does not test for equality with -9999.0, rather it tests that the difference between the number read in and the constant flag of -9999 is less than some small number (actually we should really test for the absolute value of the difference between number and flag being less than some value...)

3) In the while loop , we read in a new value for number:

```
cin >> number;
```

This is necessary, since we eventually want the while loop to terminate.

4) we test that some numbers have been actually entered to avoid a potential division by zero in our source code:

```
if (itemsProcessed) {
    average = sum / itemsProcessed;
    cout << "The sum is " << sum << endl;
    cout << "The average is " << average << endl;
} else {
    cout << " No numbers were entered " << endl;
    cout << " Sum is " << sum << endl;
    cout << " Impossible to compute average... sorry " << endl;
}
```

We could change the if statement to:
`if (itemsProcessed != 0)`
however, with C++ definition of false being 0 and anything else is true, the addition of != 0 is redundant. Best practices say to avoid this and write it the way that it is in the programme.

Alternative Solution

While flags are useful programming tricks, they are not totally without problems of their own. What if the flag represents a valid possible number to be entered? Fortunately, C++ has a mechanism which overcomes the problems associated with flags.

We can test for the end of input in the following manner:

```
while ( cin >> number ) {
   // do some more processing;
}
```

The value of an extraction operator is true provided the extraction is successful (i.e. a number is typed in). When the input stream has been exhausted (that is there are no more values to read in) the value of the extraction operation is false and the while loop will terminate. The expression cin >> number is true if and only if a number has been read in. To indicate that there are no more values to be entered, the user types in an escape sequence which is specific to the operating system. In UNIX, the end-of-file escape sequence is the Control-D sequence (press the Control key then the D key).

The program is as follows:

```
// Author: Ted Obuchowicz and John Peach
// averageUnlimitedEscape.cpp
// Example program illustrating use of while loop to find sum and
// average of an unknown number of values read in from the keyboard
// using control-d to indicate end of file

#include <iostream>
#include <string>

using namespace std;

int main() {

    float sum           = 0.0;      // sum of all the entered numbers
    float average;                  // average of all numbers
    float number;                   // The current value entered
    int itemsProcessed  = 0;        // The number of values read in

    // get the first value
    cout << "Please enter a number ";

    // Loop until the user
    while ( cin >> number ) {

        sum += number;              // update running sum
        itemsProcessed++;           // increment number of items

        // Get the next value
        cout << "Please enter a number ";

    }

    // Check to make sure that some values were entered
    if (itemsProcessed) {
        average = sum / itemsProcessed;
        cout << "The sum is " << sum << endl;
        cout << "The average is " << average << endl;
    } else {
        cout << " No numbers were entered " << endl;
        cout << " Sum is " << sum << endl;
        cout << " Impossible to compute average... sorry " << endl;
    }
    return 0;
}
```

Here is another example of a while loop. The program uses the simple "keep subtracting one number from another until you cannot subtract anymore" algorithm to perform simple integer division which yields an integer quotient and remainder:

```
// Author: Ted Obuchowicz and John Peach
// quotientRemainder.cpp
// Example program illustrating use of while loop to perform simple
// integer division which yields an integer quotient and an integer
// remainder using the grade school division algorithm

#include <iostream>
#include <string>

using namespace std;

int main() {

    int dividend;       // The dividend from the user
    int divisor;        // The divisor from the user
    int quotient = 0;   // Initialize the quotient
    int remainder;      // remainder from the division

    // read in the two values
    cout << "Enter a dividend and divisor: ";
    cin >> dividend >> divisor;

    cout << "Dividend = " << dividend <<  " Divisor = " << divisor
         << endl;

    // check for a zero divisor
    if (!divisor) {
        cout << "Cannot divide by zero you bonehead!!!!" << endl;
    } else {

        // loop until the ration would be <= 1
        while (dividend >= divisor) {
            quotient++;
            dividend -= divisor;
        }
        remainder = dividend;
        cout << "Quotient = " << quotient <<  " Remainder = "
             << remainder << endl;
    }
    return 0;
}
```

The for Loop

C++ has another looping construct which is frequently used. We will illustrate its use with a simple program which performs integer multiplication by repeated addition. The program is based on the fact that a multiplication of the form:

product = a * b

is equivalent to adding a to itself a total of b times (or adding b to itself a total of a times). In the early times of computing (when there were no built-in multiply instructions, multiplication was actually carried out in such a manner).

```
// Author: Ted Obuchowicz and John Peach
// forMultiplication.cpp
// example program illustrating use of for loop in a multiplication
// by repeated addition program we will keep it simple and assume
// only positive operands

#include <iostream>
#include <string>

using namespace std;

int main() {

    int product = 0;        // initialize to 0
    int multiplicand;
    int multiplier;

    // Get the values that you want to multiple
    cout << "Enter the multiplicand: ";
    cin >> multiplicand;

    cout << "Enter the multiplier: ";
    cin >> multiplier;

    // Perform multiplication by repeated adding
    for (int i = 0; i < multiplier; i++ ) { // Note: i is declared
        product += multiplicand;
    }

    cout << multiplicand << " * " << multiplier << " = " << product
         << endl;
    return 0;
}
```

The general form of a for loop is the following:

```
for ( initialization; test_expression; postexpression )
   // action;
```

Again as in the case of an if, if more than one statement constitutes the action portion, then they must be enclosed in the { } :

```
for ( initialization; test_expression; postexpression ) {
  action1;
  action2;
  ...
   actionn;
}
```

When a for loop is encountered in a program the following steps are performed:

1) the initialization is performed first
2) next, the test_expression is evaluated, if the expression is true then the step(s) in the Action portion are executed and then postexpression is executed
3) the test_expression is reevaluated, if true, the action statements are again executed, if false the for loop TERMINATES and control continues with the next statement in the program.

Another Example

We can re-write our while loop program to use a for loop:

```
// Author: Ted Obuchowicz and John Peach
// forAverage.cpp
// Example program illustrating use of a for loop to find sum and
// average of 5 floating point values read in from keyboard

#include <iostream>
#include <string>

using namespace std;

int main() {

    float sum               = 0.0;  // sum of all the entered numbers
    float average;                  // average of all the numbers
    float number;                   // The current value entered
    int itemsProcessed      = 0;    // The number of values read in
    const int NUM_OF_VALUES = 5;    // The number of values to read
```

```
    // Get the values from the user and add them up
    for (int i = 0; i < NUM_OF_VALUES; i++) {
        cout << "Please enter a number: ";
        cin >> number;
        sum += number;  // update running sum
    }

    average = sum / NUM_OF_VALUES;

    cout << "The sum is " << sum << endl;
    cout << "The average is "  << average << endl;

    return 0;
}
```

In a for loop, any of the

initialization
test_expression
postexpression

may be omitted. This may appear strange , but consider the following program

```
// Author: Ted Obuchowicz and John Peach
// forMissingOptions.cpp
// Example program illustrating use of an INFINITE for loop

#include <iostream>
#include <string>

using namespace std;

int main() {

    for(;;) {
        cout << "This is an infinite for loop " << endl;
    }

    return 0;  // The program will never end until you kill it with
control-c
}
```

As another example of the versatility of the for loop let us first rewrite our general purpose sum and average program:

```
// Author: Ted Obuchowicz and John Peach
// forAverageUnlimitedEscapeError.cpp
// Example program illustrating use of while loop to find sum and
// average of an unknown number of values read in from the keyboard
// using control-d to indicate end of file

#include <iostream>
#include <string>

using namespace std;

int main() {

    float sum           = 0.0;      // sum of all the entered numbers
    float average;                  // average of all the numbers
    float number;                   // The current value entered

    // get the first value
    cout << "Please enter a number ";

    // Loop until the user
    for ( int itemsProcessed = 0; cin >> number; itemsProcessed++ )

        sum += number;              // update running sum

        // Get the next value
        cout << "Please enter a number ";

    }

    // Check to make sure that some values were entered
    if (itemsProcessed) {
        average = sum / itemsProcessed;
        cout << "The sum is " << sum << endl;
        cout << "The average is " << average << endl;
    } else {
        cout << " No numbers were entered " << endl;
        cout << " Sum is " << sum << endl;
        cout << " Impossible to compute average... sorry " << endl;
    }

    return 0;
}
```

While at first glance, this program appears both syntactically and semantically correct, it DOES NOT EVEN COMPILE!!!! :

Here are the compilation error messages as reported by the g++ compiler:

```
#>g++ -o forAverageUnlimitedEscapeError forAverageUnlimitedEscapeError.cpp;
./forAverageUnlimitedEscapeError.cpp
forAverageUnlimitedEscape.cpp: In function ‘int main()’:
forAverageUnlimitedEscape.cpp:32:5: error: expected unqualified-id before ‘if’
forAverageUnlimitedEscape.cpp:36:7: error: expected unqualified-id before ‘else’
forAverageUnlimitedEscape.cpp:42:5: error: expected unqualified-id before ‘return’
forAverageUnlimitedEscape.cpp:43:1: error: expected declaration before ‘}’ token
```

The problem is related to RULES OF SCOPE. Basically, the problem is that the loop index variable itemsProcessed is known only within the body of the for loop. Outside the for loop, this index variable cannot be accessed.

The solution is to remove this variable declaration from the for loop initialization statement and put it as part of the main program:

```
// Author: Ted Obuchowicz and John Peach
// forAverageUnlimitedEscape.cpp
// Example program illustrating use of while loop to find sum and
average of an unknown
// number of values read in from the keyboard using control-d to
indicate end of file

#include <iostream>
#include <string>

using namespace std;

int main() {

    float sum           = 0.0;      // sum of all the entered numbers
    float average;                  // average of all the numbers
    float number;                   // The current value entered
    int itemsProcessed;             // Number of items processed

    // get the first value
    cout << "Please enter a number ";

    // Loop until the user
    for ( itemsProcessed = 0; cin >> number; itemsProcessed++ )

        sum += number;              // update running sum
```

```
        // Get the next value
        cout << "Please enter a number ";

    }

    // Check to make sure that some values were entered
    if (itemsProcessed) {
        average = sum / itemsProcessed;
        cout << "The sum is " << sum << endl;
        cout << "The average is " << average << endl;
    } else {
        cout << " No numbers were entered " << endl;
        cout << " Sum is " << sum << endl;
        cout << " Impossible to compute average... sorry " << endl;
    }

    return 0;
}
```

Note: The for-loop should generally not be used in this way. The for-loop is used when you know in advance the number of times that you are going to run the loop. Use a while loop when you do not know in advance how many times it can be iterated.

Nested Loops

Loops may be nested. For example;

```
// Author: Ted Obuchowicz and John Peach
// forNested.cpp

#include <iostream>
#include <string>
using namespace std;

int main() {

    for(int out = 1; out <= 5; out++) {
        for (int in = 1; in <= 5; in++) {
            cout << "out = " << out << " in = " << in << "  ";
            if (in % 5 == 0 ){
                cout << endl;
            }
        }
    }
    return 0;
}
```

The program output is:

```
out = 1 in = 1  out = 1 in = 2  out = 1 in = 3  out = 1 in = 4  out = 1 in = 5
out = 2 in = 1  out = 2 in = 2  out = 2 in = 3  out = 2 in = 4  out = 2 in = 5
out = 3 in = 1  out = 3 in = 2  out = 3 in = 3  out = 3 in = 4  out = 3 in = 5
out = 4 in = 1  out = 4 in = 2  out = 4 in = 3  out = 4 in = 4  out = 4 in = 5
out = 5 in = 1  out = 5 in = 2  out = 5 in = 3  out = 5 in = 4  out = 5 in = 5
```

In the above example, the statements in the body of the innermost for loop are repeated for every value of out in the outer for loop:

First the value of out is set to 1 and then the inner loop is executed with values of in ranging from 1 to 5, then the outer loop index out is incremented to 2 and again the inner loop repeats with values of in ranging from 1 to 5. The process repeats until the outer loop terminated when out becomes equal to 6.

The break Statement

A break statement can be used inside a loop construct to prematurely terminate the loop. Program execution then resumes with the statement which follows the looping statement. The break can be used in any of the C++ looping constructs.

Here is an example of a for loop with a break statement which forces an exit from the loop when the user enters the value -999:

```
// Author: Ted Obuchowicz and John Peach
// break.cpp
// Example program illustrating use of break statement inside a for
// loop to cause premature exit from the loop

#include <iostream>
#include <string>

using namespace std;

int main() {

    int const EXIT_VALUE = -999;    // flag to stop input
    int inputValue;                 // user entered value

    while ( cin >> inputValue ) {
        if ( inputValue != EXIT_VALUE) {
            cout << "You entered " << inputValue << endl;
        } else {
            cout << "EXIT_VALUE encountered in input stream... "
                 << "breaking from loop!" << endl;
```

```
            break;  // exit from the loop
        }
    }

    return 0;
}
```

The use of break statements inside of loops is considered VERY poor style. It makes program behaviour difficult to understand. It is always possible to rewrite the loop without using a break. It is shown here so that you are aware of it, but never use it outside of a switch-statement. The above program would be better written as follows:

```
// Author: Ted Obuchowicz and John Peach
// noBreak.cpp
// Example program illustrating use of break statement inside a for
// loop to cause premature exit from the loop

#include <iostream>
#include <string>

using namespace std;

int main() {

    int const EXIT_VALUE = -999;    // flag to stop input
    int inputValue;                 // user entered value


    while ( cin >> inputValue && inputValue != EXIT_VALUE) {
            cout << "You entered " << inputValue << endl;
    }

    if ( inputValue == EXIT_VALUE ) {
        cout << "EXIT_VALUE encountered in input stream... "
             << “breaking from loop!" << endl;
    }

    return 0;
}
```

A BREAK STATEMENT MAY ONLY APPEAR IN A SWITCH STATEMENT OR IN ANY OF THE LOOP STATEMENTS. IT MAY NOT APPEAR ANYWHERE ELSE!

The following program does not compile (since it contains a break in an if statement which is not allowed in C++):

```
// Author: Ted Obuchowicz and John Peach
// breakIllegal.cpp
// Example program illustrating an illegal use of a break statement

#include <iostream>
#include <string>

using namespace std;

int main() {

    if ( 4 < 5 ) {
        cout << "4 less than 5" << endl;
        break; // this break is illegal
    }

    return 0;
}
```

```
#>g++ -o breakIllegal breakIllegal.cpp; ./breakIllegal
breakIllegal.cpp: In function ‘int main()’:
breakIllegal.cpp:14:9: error: break statement not within loop or switch
```

The continue Statement

The continue statement is used ONLY IN LOOPS. When a continue statement is encountered in the body of a loop, the REMAINING statements in the body of the loop are SKIPPED OVER, and a new LOOP ITERATION commences. Note that that while a continue statement causes the program to skip any remaining body statements, it does not skip the loop test_expression. In a for loop, the continue makes the program skip to the postexpression part and then it tests the condition. In a while loop, the continue causes the program to go directly to the test expression part of the while (condition).

Here is a program which uses a continue statement within a for loop which reads 5 characters and counts the number of 't' characters which have been entered:

```
// Author: Ted Obuchowicz and John Peach
// continue.cpp
// Example program illustrating use of a continue statement within a
for loop

#include <iostream>
#include <string>

using namespace std;
```

```
int main() {

    char letter;
    int  numT = 0;

    for(int i = 0; i < 5; i++) {
        cout << "Please enter a letter: ";
        cin >> letter;
        if (  letter != 't' ) {
            continue;
        }
        cout << "Ahhhh... the t was entered!" << endl;
        numT++;
    }

    cout << "The total number of t's entered was " << numT << endl;

    return 0;
}
```

If the letter entered is not equal to 't', then the program goes to the i++ part of the for (int i = 0; i < 5; i++) part and evaluates if the condition i < 5 is true or not. Based on this evaluation either the loop continues or terminates. If the letter entered is equal to 't', then the two statements following the continue are executed and loop execution carries on in the normal manner (when it gets to the } the i++ is executed and the condition is evaluated).

Like the break statement, its use is considered poor style and it should not be used. A loop can always be re-structured to avoid its use. It is shown here for completeness.

More Examples Of Programs With Loops

Example 1: Computing the factorial of a positive integer using a for loop.

The factorial of an integer n (designated as n!) is defined as the product of the integers 1 through n:

n! = 1 if n = 0
= n x (n-1) x ... x 1 if n >=1

We can use a for loop to compute the product of the numbers 1 through n as in the following program:

```
// Author: Ted Obuchowicz and John Peach
// factorialInt.cpp
// Example program illustrating use of a for loop to compute the
// factorial

#include <iostream>
#include <string>

using namespace std;

int main() {

    unsigned int n;                 // the value to determine the
                                    // factorial for
    unsigned int factorial = 1;     // The calculated factorial of n

    cout << "Please enter a positive integer: ";
    cin >> n;

    for (unsigned int i = 1; i < n + 1; ++i){
        factorial *= i;
    }

    cout << n << "! is " <<  factorial << endl;

    return 0;
}
```

This program exhibits interesting behaviour:

```
10! is 3628800
11! is 39916800
12! is 479001600
13! is 1932053504
```

The last value is incorrect as the true value of 13! = 6 227 020 000. This value exceeds the maximum value of an unsigned long int (which on our UNIX workstations is 4 294 967 295) so an integer OVERFLOW arises during the calculation of 13! and an incorrect answer is given.

How do we fix the problem?

Simple.... change the type of the variable factorial from unsigned long int to a long:

```
// Author: Ted Obuchowicz and John Peach
// factoriallong.cpp
// Example program illustrating use of a for loop to compute the
factorial

#include <iostream>
#include <string>

using namespace std;

int main() {

    unsigned int n;                  // the value to determine the
                                     // factorial for
    unsigned long int factorial = 1; // The calculated factorial of n

    cout << "Please enter a positive integer: ";
    cin >> n;

    for (unsigned int i = 1; i < n + 1; ++i){
        factorial *= i;
    }

    cout << n << "! is " <<  factorial << endl;

    return 0;
}
```

Now, it gives correct results (to a certain point):

```
17! is 355687428096000
18! is 6402373705728000
19! is 121645100408832000
20! is 2432902008176640000
21! is 14197454024290336768
```

The reported value for 21! is incorrect for the same reason that we saw with the int version. The correct value for 21! is 51 090 942 171 709 440 000 but the upper limit an unsigned long in according it ULONG_MAX in limits.h is 18 446 744 073 709 551 615 or 2^16

Example 2: A program to compute the first 15 Fibonacci Numbers:

The sequence of numbers:

1 1 2 3 5 8 13 21 , ...

The first two numbers in the sequence are equal to 1, and every successive term is the sum of the previous two terms is known as the Fibonacci sequence. It was developed by an Italian mathematician by the name of Leonardo Pisano in 1202. He stumbled upon his famous sequence while pondering the problem of determining the number of rabbits which a pair of fertile rabbits may produce in one year...

Here is a program which uses a for loop to display the first 15 Fibonacci numbers:

```
// Author: Ted Obuchowicz and John Peach
// fibonacci.cpp
// Example program illustrating use of for loop to display the first
15 Fibonacci numbers

#include <iostream>
#include <string>

using namespace std;

int main() {

    int previous = 1;       // the previous calculate fibonacci value
    int current  = 1;       // the current fibonacci value
    int next;               // the next fibonacci value

    // Write the first two numbers
    cout << previous << endl;
    cout << current << endl;

    for(int i = 3; i < 16; i++) {
        // Calculate the next value in the sequence and display it
        next = previous + current;
        cout << sum << endl;

        // swap the values as we move forward in the sequence
        previous = current;
        current = sum;
    }

    return 0;
}
```

The program output is:

```
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
```

The do-while Loop

C++ has another loop constuct called the do-while loop which is used in situations where a certain action is to be performed
AT LEAST ONCE. The general form is:

```
do {
   statement 1;
   statement 2;
   ...
   last_statement_to_do;
} while (some_condition);
```

Here is a simple program which uses the do while statement:

```
// Author: Ted Obuchowicz and John Peach
// 42.cpp
// Example program illustrating use of do while construct with a
// guessing game which continues prompting the user for an "Answer to
// the Ultimate Question of Life, The Universe, and Everything"
// The program also keeps track of the number of guesses until a
// correct guess is entered

#include <iostream>
#include <string>
```

```
using namespace std;

int main() {

    const int ANSWER = 42;      // The answer to the question
    int guessCount = 0;         // The number of guesses made
    int userGuess;              // The current guess from the user

    do {
        cout << "Answer to the Ultimate Question of Life, The "
             << "Universe, and Everything: " << endl;
        cin >> userGuess;
        guessCount++;
    } while (userGuess != ANSWER);

    cout << "Congratulations. It took you " << guessCount
         << " to determine the \"Answer to the Ultimate Question of "
         << "Life, The Universe, and Everything.\"  It took Deep "
         << "Thought 7.5 million years" << endl;

    return 0;
}
```

Some More Examples Of Loops

Here is a program which uses while loops to "decompose" an integer value (which is less than 1 000 000) into its components consisting of the number of hundreds of thousands, number of tens of thousands, number of thousands, etc.

A typical interaction with the program is:

```
Please  enter a number less than a million: 567890
hundreds of thousands = 5
tens of thousands = 6
thousands = 7
hundreds = 8
tens = 9
ones = 0
```

Here is the program which uses a sequence of while loops:

```
// Author: Ted Obuchowicz and John Peach
// numberBreakdown.cpp
// Example program illustrating use of while loops

#include <iostream>
#include <string>
```

```
using namespace std;
int main() {

    int number;                          // User entered number
    int hundredThousandsCounter = 0;     // The number hundreds of thousands
    int tenThousandsCounter     = 0;     // The number of tens of thousands
    int thousandsCounter        = 0;     // The number of thousands
    int hundredsCounter         = 0;     // The number of hundreds
    int tensCounter             = 0;     // The number of tens
    int onesCounter             = 0;     // The number of ones

    cout << "Please  enter a number less than a million: ";
    cin >> number;

    while (number >= 100000) {
        hundredThousandsCounter++;
        number = number - 100000;
    }

    while (number >= 10000) {
        tenThousandsCounter++;
        number = number - 10000;
    }

    while (number >= 1000) {
        thousandsCounter++;
        number = number - 1000;
    }

    while (number >= 100) {
        hundredsCounter++;
        number = number - 100;
    }

    while (number >= 10) {
        tensCounter++;
        number = number - 10;
    }

    onesCounter = number;

    cout << "hundreds of thousands = " << hundredThousandsCounter << endl;
    cout << "tens of thousands = " <<  tenThousandsCounter << endl;
    cout << "thousands = " <<  thousandsCounter << endl;
    cout << "hundreds = " <<  hundredsCounter << endl;
    cout << "tens = " << tensCounter << endl;
    cout << "ones = " << onesCounter << endl;

    return 0;
}
```

We start off with the first while loop which will be entered if the number >= 100000, the loop is entered and we keep on subtracting 100000 from the number and increment a counter. Eventually, the number becomes less than 100000 and we continue with the next loop. This loop is responsible for subtracting 10000 from the resulting number and incrementing a certain other counter. We do this for 1000, 100, 10 ... whatever is left over is the number of 'ones' in the number.

Here is a more cryptic version of a similar program. It simply displays line by line the number of hundreds of thousands, number of thousands, etc that make up a certain number:

The output is:

```
Please  enter a number less than a million: 345678
3
4
5
6
7
8
```

The actual program is:

```
// Author: Ted Obuchowicz and John Peach
// numberBreakdownFor.cpp
// Example program illustrating use of for-loop

#include <iostream>
#include <string>
#include <math.h>

using namespace std;

int main() {

    int number;
    cout << "Please enter a number less than a million: ";
    cin >> number;

    for(int i = 6; i >= 1; i--) {
        cout <<  (number %  (int)( pow(10, i) )) / ( (int) pow(10, i-1))
             << endl;
    }

    return 0;
}
```

Which version is easier to understand?

The pow(x,n) function returns the value of x^n by the way. It is found in the library <math.h>. It makes use of the following facts about the mod operator:

i = 1 ones = 345 % 10 = 5

i = 2 tens = 345 % 100 = 45, then do 45/ 10^1, where the division is performed as INTEGER DIVISION (4.5, we truncate the 0.5 to obtain 4)

i = 3 hundreds = 345 % 1000 = 345, then do 345 / 10^2 to obtain 3 (again as integer division)

In each loop iteration we are doing number % 10^i , then take this result and do result % 10^{i-1}

Difficult to understand from the code, but it was possible to code the program using a single clever for loop. What comments should have been added to the code so that someone looking at the code could understand what is going on.