Flow of Control

So far all off the programs we have seen have been linear – each statement executes in order, from top to bottom. To make interesting games we need to write programs that skip or execute sections of code based on some condition.
In this lecture we will look in more detail at Boolean variables, relational operators and controlling the flow of the program using if, if-else and switch statements. Together these represent the basics in introducing controlling outcomes and control of the program.

Boolean Variables and Expressions

Last we mentioned and covered very briefly Boolean variables. We talked about how standard C++ supports a data type for representing logical values.

This data type is called Boolean and is represented in the C++ language by the keyword `bool`.

Boolean variables can have either of two values `true` or `false`.

In C++, in order to ask a question, a program makes a logical assertion which is evaluated to either true or false at run-time.

Assertion = In computer programming, an assertion is a programming language construct which immediately aborts program execution if a certain condition or expression is false (an "assertion failure")
Logical Assertion = A statement that asserts that a certain premise is true, and is useful for statements in proof

In order to assert that the student's age is greater-than or equal to 18 we can do the following:

```
const int DRINKING_AGE = 18;
bool      isDrinkingAge;
int       userAge;

cin >> userAge;

isDrinkingAge = (userAge >= DRINKING_AGE);
```

The value of `isDrinkingAge` will now be true is the `userAge` was greater-than or equal to the value of `DRINKING_AGE` (18).
It is then possible to test the Boolean variable in our program, a check to see if it is either true or false.

Relational Operators

We have just seen from the above C++ code something else than is reminiscent of maths.

This line:

```
isDrinkingAge = (userAge >= DRINKING_AGE);
```

Holds the following operator >=

This is an example of a relational operator. Relational operators are used in relational expressions.

C++ has the following six standard relational operators:

Relational Operator	**Meaning**
==	Equals
!=	Does not equal
>	Is greater-than
>=	Is greater-than or equal to
<	Is less-than
<=	Is less-than or equal to

You should have all seen either these, or certainly most of these in maths.

They are called relational because they test the relationship, generally in terms of value, between two things.

For instance the following expression:

```
int x = 7, y = 8;
x == y;              // test to see if x is equal to y
```

Tests to see if the variable `x` is equal to `y`. If they are equal then true is returned if they are not equal then false is returned.
In this the example expression above `x` is not equal to `y`, so the expression `x` is equal to `y` (`x == y`) is false. Therefore false is returned.

It is possible to include relational operators in larger expressions.
Example:

```
(a * b - c * d) > 0
```

Relational operators can be used to compare any of the C++ types covered so far.
Mixing comparisons between data types is allowed but may make no sense.

You can for example compare two `char` variables with relational operators:

```
char firstChar = 'a';
char secondChar = 'b';

firstChar == secondChar;
```

**Note: You can compare char variables as in C++ char variables have an associated value*

Comparing two character variables makes sense, but comparing a character variable and an integer makes not much sense at all:

```
char firstChar = 'a';
int firstInt = 27;

firstInt == firstChar;
```

For examples of relational operators and expressions see the activehelix website.

Logical Operators

Logical operators are used to combine multiple relational or logical tests. They allow compound conditions to be evaluated.

They are sometimes called Boolean operators as they allow you to logically combine Boolean variables (that is, variables of type `bool`, with true or false values).

C++ has three logical (or Boolean operators):

Operator	Meaning
!	NOT
&&	AND
||	OR

The NOT operator is unary requiring one operand whilst AND and OR operators are binary as they require two operands.

If that still seems a little vague lets examine them one at a time:

NOT
The logical NOT operator ! is a unary operator—that is, it takes only one operand.
The effect of the ! is that the logical value of its operand is reversed: If something is true, ! makes it false; if it is false, ! makes it true.

For example, `(x==7)` is true if x is equal to 7, but `!(x==7)` is true if x is not equal to 7. (In this situation you could use the relational not equals operator, `x != 7`, to achieve the same effect.)

You can also perform the following `!x` – this checks to see if x is equal to zero.
This is because if you remember from last week a true statement is one that evaluates to a non-zero value.

The expression x is true whenever x is not 0, and false when x is 0. Applying the ! operator to this situation, we can see that the !x is true whenever x is 0, since it reverses the truth value of x.

AND
The logical AND operator && is a binary operator—that is, it takes two operands.

Today is a weekday has a Boolean value, since it's either true or false. Another Boolean expression is Maria took the car. We can connect these expressions logically: If today is a weekday, AND Maria took the car, then I'll have to take the bus. The logical connection here is the word and, which provides a true or false value to the combination of the two phrases. Only if they are both true will I have to take the bus.

A logical C++ expression involving an AND (&&) operator may look like this:

```
(x==7 && y==11)
```

The test expression will be true only if x is 7 and y is 11. The logical AND operator && joins the two relational expressions to achieve this result. (A relational expression is one that uses a relational operator.)

OR
The logical OR operator || is a binary operator—that is, it takes two operands.

Today is a weekday has a Boolean value, since it's either true or false. Another Boolean expression is Maria took the car. We can connect these expressions logically: If today is a weekday, OR Maria took the car, then I'll have to take the bus. The logical connection here is the word OR, which provides a true or false value to the combination of the two phrases. If either one is true will I have to take the bus.

A logical C++ expression involving an OR (||) operator may look like this:

```
(x<7 || y>11)
```

The test expression will be true only if either x is less-than 7 OR y is 11. The logical OR operator || joins the two relational expressions to achieve this result. (A relational expression is one that uses a relational operator.)

For examples of logical operators and expressions see the activehelix website.

Operator Precedence

We talked about the precedence of operators, the order in which they are evaluated, last week.
Last week though we only talked about the precedence of arithmetic operators.

These new operators add new levels of precedence, as seen in the following table:

Precedence rules:

- Expressions in parenthesis are evaluated first
- Unary operators: (unary) – and NOT !
- Multiplication, Division, Modulus: *, % and /
- Addition and Subtraction: + and –
- Relational operators: <, <=, > and >=
- Logical equal and not equal: == and !=
- Logical AND: &&
- Logical OR: ||
- Assignment statement: =

The Flow of Control

The flow of execution is the order in which the computer executes statements in a program.

The default flow is sequential execution:

```
cin >> x;       // read in value and assign to x
y = x++;        // add one to x and assign to y
cout << y;      // print out y
```

It happens in the order listed starting at the first statement below main to the return statement at the end of main.

Control Flow = In computer science and in computer programming, statements in pseudo-code or in a program are normally obeyed one after the other in the order in which they are written (sequential flow of control). Most programming languages have control flow statements which allow variations in this sequential order:

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

Control Structure = A statement that is used to alter the default sequential flow of control

Selection = A control structure that allows choice among two or more actions

The ability to control the flow of your program, whether or not your program will run a section of code based on specific tests, is valuable to the programmer.

There quite a few ways of ‘controlling the flow’ of your program. In today’s lecture we are going to cover `if`, `if-else` and `switch` statements.

`if` Statements

‘If’ statements are a way of controlling this flow, and are possibly the simplest way of doing so.

They are used all the time in programming, generally to test things – they enable you to control if a program enters a section of code or not based on whether a given condition is true or false.

In a computer game for instance you only what a trap to activate when the play has moved near by then you use a if statement:
If the player is with distance x
Activate trap

Another important function of the ‘if’ statement is that it allows the program to select an action based upon the user's input. For example, by using an ‘if’ statement to check a user entered password, your program can decide whether a user is allowed access to the program.

Basically an if-statement is used to select between performing an action and not performing.

Despite this simplicity, without a conditional statement such as the ‘if’ statement, programs would run almost the exact same way every time. ‘If’ statements allow the flow of the program to be changed and they therefore allow algorithms and more interesting code.

** I have just mentioned the word conditional now – without conditional statements. This is why conditional operators were covered first – as they are usually used in concert with statements that control the flow as they enable a block of code to run based on the evaluation of that condition (if the condition is true)*

The syntax of if statement:

```
if (condition)
{
     // statements to be executed if condition is true
}
```

This if often expressed in the following way:

```
if (TRUE)
{
     do this stuff;
}
```

You may also see ‘if’ statements written like this:

```
if (TRUE)
     do this stuff;
```

They are written that way, no curly brackets, by some programmers if there is only one statement that needs executing.

Examples of expressions that can be evaluated are:

```
if (5 > 4)
{
     cout << “of course it is!”;
}
```

In this example the integer 5 is compared to 4 to see if it is bigger, if it is (TRUE) then `of course it is!` is printed to the screen.

You can apply your brain and work out that there are many possible comparisons, due to the amount of data types plus logical and conditional operators.

Notice as well that the ‘if’ statements do not have semi-colons after them. Putting semi-colons after them creates empty statements.
If we had the following in our code:

```
if(false)
     cout << “this will never be displayed\n”;
```

Then we will never print the literal string - “this will never be displayed\n”
If our statement looked like this:

```
if(false);
     cout << “this will never be displayed\n”;
```

Then we would print out the literal string - “this will never be displayed\n”
This is because the semi-colon is associated with the if(false) statement, effectively making it empty and ending it, the cout statement is therefore evaluated as a not associated with the ‘if’ statement.

`if-else` Statements

The flow of control can be moved on from there by adding an ‘else’ statement. This enables you to take one branch if your condition is TRUE and another if it is FALSE.
You effectively make a choice between two alternatives.

The syntax of if-else statement:

```
if (condition)
{
     // statements to be executed if condition is true
}
else
{
     // statements to be executed if condition is false
}
```

This if often expressed in the following way:

```
if (FALSE)
{
     wont do this
}
else
{
     but will do this
}
```

Or

```
if (FALSE)
     wont do this
else
     will do this
```

Again they are written that way, no curly brackets, by some programmers if there is only one statement that needs executing.

An example of an 'if-else' statement in C++:

```
if (Grade == 'A')
{
     cout << "Well Done!";
}
else
{
     cout << "Nevermind!";
}
```

In this example `Grade` (a variable of type `char`) is evaluated to see if it is equal to 'A' if it is (TRUE) then `Well Done!` is printed to the screen. If it isn't (FALSE) then `Nevermind!` is printed to the screen.

`if-else-if` statement

Increasing levels of complexity can be added, such as an if-else-if statement.

Statements such as these add increasing levels of complexity and enable a greater control of the program.

One use for else is if there are two conditional statements that may both evaluate to true, yet you wish only one of the two to have the code block following it to be executed. You can use an else if after the if statement; that way, if the first statement is true, the else if will be ignored, but if the if statement is false, it will then check the condition for the else if statement. If the if statement was true the else statement will not be checked. It is possible to use numerous else if statements.

The syntax of if-else statement:

```
if (condition)
{
     // statements to be executed if condition is true
}
else if (condition)
{
     // statements to be executed if condition is true
}
else
{
     // statements to be executed if condition is false
}
```

An example of an 'if-else-if' statement in C++:

```
if(age < 21)
{ // If age is less than 21
      cout << "You are pretty young!";
}
else if(age > 21 && age < 100)
{ // If age is greater than 21 and less than 100
      cout << "You are old";
}
else
{ // Executed if no other statement is executed
    cout << "You are really old \n\n";
}
```

The number of additional 'else if' clauses is determined by the programmer. You don't want it to get to silly though by having lots of else if statements, C++ provides an additional alternative to lots of 'else ifs'. It is called a `switch` statement; we will come on to that shortly.

Compound Conditions

In an `if` test, the overall condition must evaluate to true or false.
Compound Conditions are conditionals (the evaluation that occurs at the beginning) consisting of multiple conditions in a single if test, can be constructed using the following logical and relational operators : and (&&), or

(||), not (!), logical equals (==), less than (<), greater than (>), or combinations of these.
You use compound conditionals in more complex programs relatively frequently.
An example is noughts-and-crosses (or tic-tac-toe) where you would use compound conditionals to assess if the player(s) have got 3 in a row.

An example of a compound statement (consisting of multiple conditions) is as follows:

```
if ( ((condition 1) || (condition 2)) && (condition 3) )
{
     // statements to be executed if condition is true
}
```

This says that if either condition 1 OR condition 2 AND condition 3 are true then execute the statements contained.
This means that the combinations that are correct condition 1 AND condition 3 OR condition 2 AND condition 3.

Nesting Statements

An if-else statement may contain valid C++ statements; this includes other if or if-else statements.

The syntax of a very basic nested if statement:

```
if (condition)
{
     if (condition)
     {
          // statements to execute if condition is true
     }
}
```

A C++ example:

```
if (age >= 65)
{
    cout << "a senior." << endl;
}
else
{
    if (age >= 19)
    {
        cout << "an adult." << endl;
    }
    else
    {
         if (age >= 13)
         {
              cout << "a teenager." << endl;
```

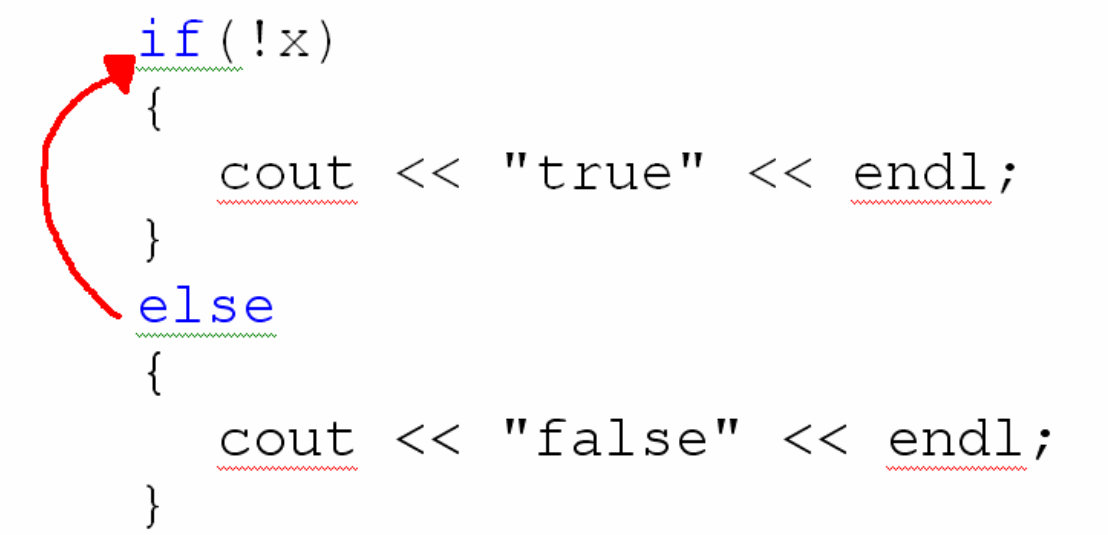
```
            }
            else
            {
                cout << "a child." << endl;
            }
    }
}
```

The expression age >= 65 is evaluated. If the result is true, the characters "a senior." are sent to the output stream. If the result is false, the expression age >= 19 is evaluated. If the result is true, the characters "an adult." are sent to the output stream. If the result is false, the expression age >= 13 is evaluated. If the result is true, the characters "a teenager." are sent to the output stream. If the result is false, the expression "a child." is sent to the output stream.
Notice: once age has a value, only one statement is selected to be executed.

If possible use if-else statements, especially if you would require a large number of nested layers, as they are easier to read

Conditions that are 'mutually exclusive'. One condition being true excludes all else from being true should be tested for with nested if statements for efficiency.

If nesting is carried out to too deep a level and indenting is not consistent then deeply nested if or if-else statements can be confusing to read and interpret. Using a nested if or if-else statement does raise the question of which `if` an `else` is paired to.
The syntax rule is simple: An else is paired with the closest previous uncompleted `if`



`switch` Statements

Some problems require making simple choices among a large number of alternatives, such as encrypting numbers or letters.

One possible solution would be to use an else-if statement:

```
if (nextCharacter == 'a')
      cout << 'a';
else if (nextCharacter == 'b')
      cout << 'b';
else if (nextCharacter == 'c')
      cout << 'c';
else if (nextCharacter == 'd')
      cout << 'd';
else if (nextCharacter == 'e')
      cout << 'e';
else if (nextCharacter == 'f')
      cout << 'f';
else if (nextCharacter == 'g')
      cout << 'g';
else if (nextCharacter == 'h')
      cout << 'h';
```

This looks ugly and repetitive. The alternative is a `switch` statement.

Switch statements make a good alternative to if-then-else statements in certain circumstances. They enable you to branch on any number of values instead of evaluating just one value.

How ever, a switch statement can only be used in certain circumstances:

- When the test is for equality (==)
- The compared values are characters and integers

The basic layout of a switch statement is the following:

```
switch (selector)       // variable or expression of type char or int
{
      case valueOne:    // If expression is equal to case...
      {
            statement;  // Do this..
            break;      // Then exit the switch statement.
      }
      case valueTwo:
      {
            statement;
            break;
      }
      default:
      {
            statement;
      }
}
```

When a switch statement is executed the `selector` is evaluated and the statement corresponding to the matching constant in the value list is executed.
If no match occurs the default clause, if present, is selected.

It is a good idea for the values to match the selector type or it won't work very well.

The massive else-if statement code above has been re-done using a switch statement:

```
switch (nextCharacter)
{
	case 'a':
		cout << 'a';
		break;
	case 'b':
		cout << 'b';
		break;
	case 'c':
		cout << 'c';
		break;
	case 'd':
		cout << 'd';
		break;
	case 'e':
		cout << 'e';
		break;
	case 'f':
		cout << 'f';
		break;
	case 'g':
		cout << 'g';
		break;
	case 'h':
		cout << 'h';
		break;
	default:
		cout << nextCharacter;
		break;
}
```

The logical effect is the same, but using the switch statement:

- Makes the code easier to read
- Will execute slightly faster
- Easier to modify certain aspects (e.g. input name)

You may have noticed as well that my syntax example had brackets surrounding the statements after the case, but the example does not.
You can use either; sometimes it makes the code more readable adding brackets.

`switch` Details

The break statement is there to stop the flow of the program from falling through into the next succeeding case. It terminates the execution of the nearest enclosing loop or conditional statement in which it appears. Control passes to the statement that follows the terminated statement, if any.
If I missed out for whatever reason the first break statement the second case will also be executed as well.

If the selector does not match the any case label and there is not default clause then execution proceeds to the first statement following the `switch` statement.
Is generally considered bad programming practice to omit a default.

It is legal for a case clause to be empty:

```
switch (nextCharacter)
{
      case 'a':
            cout << 'a';
            break;
      case 'b':
            cout << 'b';
            break;
      case 'c':
            break;      // nothing happens – clause is empty
}
```

`switch` Limitations

A `switch` can only be used in cases involving equality comparison for either a `char` of `int` variable.
It is therefore of no use when checking the value of a `float`, `double` or `string` variable.

Execution Tracing

If you are really keen on computer science methods or are struggling with trying to work out the output of a program whilst looking at code you can use a method known as execution tracing (or Desk-Checking).

It basically involves hand calculating the output of a program with test data by mimicking the actions of a computer.
This process can also be applied to working out an algorithm.

It is very tedious though so be warned!

A Brief Word on Commenting

You have seen code that I have done contain green highlights that describe the code. This is known as commenting.

Commenting is an important part of programming. Commenting is the act of putting little notes to yourself, and other potential users, within your code. It's done so that if you or someone else looks over your code, you don't have to go through several lines to figure out what each does. Notice the difference in the following two sections of code. Which is more easily understood (assuming little or no knowledge of C++, of course).

```
float answer, var1, var2;
answer = var1 / var2;
cout << answer << endl;
```

And another piece of code, with commenting:

```
/* This adds the variables var1 and var2. It then prints
out the result */
int answer, var1 = 6, var2 = 7;
answer = var1 + var2;
cout << answer << endl;
```

Comments are added using two methods:
Forward double slash (//) or forward slash and star followed by star forward slash (/* */)

The /* or // tells the compiler that text is a comment and is not to be compiled.

// Indicates the line after is a comment, /* and */ indicates that everything in between is a comment.

```
/* This can go over multiple lines and everything between
the stars and slashes is classed as a comment */

// This just works on one line
// Everything after the slashes is a classed as a comment
```

There will be a document available that helps with programming style tips available. This includes commenting guidelines. Check to website when it is back online.