Lesson 1 - Your first program

Friday, 03 March 2017 11:53 AM

General rule:

Always use << with cout.

Include <iostream>:

Special instruction that we always put at the top of the code to tell the compiler to include the iostream header file. Header files basically define a number of additional instructions that we can use in our programs to make them simpler and neater. A particularly useful one is iostream. The io in iostream stands for Input/Output and this header file allows us to use the cout instruction to output messages to the screen.

namespace std;

Some C++ compilers allow you to leave this out.

```
#include <iostream>
int main( )
{
     std::cout << "Hello world";
     return 0;
}
```

cout is defined in the iostream header file. All the names defined in it belong to a namespace called std. If you don't use using namespace std; in your program, then you should prefix every name that belongs to the std namespace with std::.

int main( )

This is the header for the main function of the program.

Semicolons:

Semicolon is called a statement terminator. Each statement must be terminated by a semicolon.

cout <<:

Stands for console output (display). Pronounce cout as “see-out”

Comment statement:

Is a text that we add to a program that has no effect on the running of the program but that is generally used to explain what the programming (or part of the program) does.

//This is a comment statement

Endl:

Endl stands for end line. When used with cout it makes output continue on the next line.

C++ Structure:

```
//DescriptiveComment
#include <StandardHeaderFile>
using namespace std;

int main( )
{
    StatementSequence;
}
```

Lesson 2 - Integers

Friday, 03 March 2017 11:56 AM

Integer rules:

Brackets are calculated 1st.

Multiplication and division is calculated 2nd.

Plus and minus is calculated last.

Integers can have negative numbers

We can write programs that make calculations on integers. Integers are like the whole numbers since they include 0 and the negative numbers. No fractions are allowed with integers.

An integer expression is a number of integer values and integer operators (+, -, * and /). Addition and subtraction are done after multiplication and division. If addition and subtraction (or multiplication and division) are done next to one another, they are performed from left to right. Round brackets can be used to force certain subexpressions to be calculated before others to get around these rules.

The division operator / throws away any remainder

The value of an expression can be calculated and displayed in a cout statement. A string can be displayed with the value of such an expression in the same cout statement by separating them with the << operator. Any integers and/or operators included in a string (i.e. between quote characters) are not evaluated but are displayed as is. Only expressions without quote characters are evaluated.

We can also display a message and the value of an expression on the same line using two separate cout statements. To do this, we omit << endl at the end of the first cout statement. The output of the second cout statement will then be displayed on the same line as the output of the first cout statement.

Lesson 3 - Variables

Friday, 03 March 2017 11:57 AM

Variables

```
//What happens?
#include <iostream>
using namespace std;

int main( )
{
int age;
cout << "Enter your age: ";
cin >> age;
cout << "That's a great age to be!" << endl;
return 0;
}
```

Int age; is called a declaration statement. It declares a variable called age. It sets aside a memory position for a value to be stored and associates a name with that memory position.

A variable has a name so that we can refer to the value stored in the memory position later in the program.

Note that a variable name has no meaning to the computer whatsoever. Use meaningful names so that you and others can read and understand the programs.

The statement cin >> age; is called an input statement. Is used to obtain a value from the keyboard and store it in a variable.

Like cout, cin is defined in the standard file iostream. It stands for console input, and is pronounced as “see-in”.

Variable rules:

- Any combination of numeric characters and letters of the alphabet can be used, except that a name cannot start with a numeric character.
- The only other character that may be used in a name is the underscore character _. Names may also start with _.
- Certain words like int and return can’t be used. However, a variable can be declared with the name cout, cin or endl.
- We can display the value of a variable in a cout statement. Furthermore, it can be displayed together with other strings (and values of expressions) in a single cout statement.
- We must always declare a variable before we use it or refer to it.

Camel notation: When we want a variable name to consist of more than one word, we generally connect all the words together and use uppercase for the first letters of all the subsequent words. E.g. taxBeforeProfits.

% - Remainder

If we have two numbers i and j, the expression i % j gives the remainder of i divided by j. The remainder is always a number from 0 up to 1 less than the divisor. For example, n % 10 will always give a remainder of 0 up to 9, no matter what the value of n is.

Lesson 4 - Assignment statements

Friday, 03 March 2017 11:58 AM

Programming concepts

We can change the value of a variable in two ways: Firstly we can input a value into a variable using a cin statement. Secondly (as we saw in this lesson) we can assign a value to a variable using an *assignment statement*.

The general format of an assignment statement is

Variable = Expression;

The value of *Expression* is calculated first, and then the result is stored in *Variable*. *Expression* may therefore contain (one or more occurrences of) *Variable*. In such a case, the old value of *Variable* is used to calculate *Expression*, and the result is stored back in *Variable*.

Compound assignment operators, namely +=, -=, *= and /=. They provide a short-hand notation for assignment statements for changing the value of a variable. For example:

Variable -= Expression;

Subtracts the value of *Expression* from *Variable*.

The two unary operators ++ and -- add 1 and subtract 1 from the variable they are applied to, respectively. ++ is called the *unary increment operator* and -- is called the *unary decrement operator*. These operators, together with the variable they are operating on, return a value. They can therefore be used where an expression is required or in a longer complicated expression. The value returned by such an expression depends on whether the operator is used before the variable (prefix form) or after it (postfix form). If the operator is used before it, the returned value is the final (changed) value of the variable, whereas if it is used after it, the returned value is the original value of the variable (i.e. before it was changed).

Programming principles

- Use numerical digits in variable names to distinguish different variables that are used to store the same kinds of values.
- Remember to initialise variables (either by assigning a value to them or by inputting a value for them) before using them
- Rather declare additional variables than do calculations in an output statement.
- If you find you are doing exactly the same calculations over and over, declare a variable and store the results of the calculations (once) in it.

Lesson 5 - Variable diagrams

Friday, 03 March 2017 11:59 AM

Programming concepts

The purpose of this lesson was to show how variable diagrams can be used to work out what programs do, particularly as they get more difficult. Variable diagrams can also be a very good way of tracking down bugs which are difficult to find.

Can initialise a variable while declaring it:

- `Int Variable = Value;`

Can declare constants to improve the readability and maintainability of our code:

- `Const int ConstantName = Value;`

When declaring a variable, telling the computer to set aside a memory position to store a value that will probably change during execution of the program. By contrast, when declaring a constant, giving it a value that will remain unaltered throughout the program. When declaring a variable we *may* provide a value to initialise it, but when declaring a constant, we *must* provide a value.

In many computer languages, a variable only has a value once it has been initialised (i.e once the program actually assigns it a value). Before initialisation, the value of a variable is undefined.

Programming principles

- Remember to initialise variables
- Declare constants whenever you can
- Use uppercase letters for the names of constants (to distinguish them from variables). If the name consists of more than one word, use the underscore character to separate the words.
- If it becomes difficult to work out what a program does, draw variable diagrams!

Lesson 6 - Floating point numbers

Friday, 03 March 2017 11:59 AM

Programming concepts

Cout.setf(ios::fixed); → sets numbers to non-scientific.

Cout.precision(4); → sets decimals to 4

Float(variable1); → changes the integer variable to float.

Programming principles

- Test your programs thoroughly with a number of test cases. If necessary, use additional cout statements at intermediate points in the program to show the values of intermediate calculations.
- Rather use explicit than implicit type conversions. This makes it clearer what the program is actually doing.
- Use comments in your programs to explain what sections of code do. This is so that anyone who reads your program can easily follow what every part of the program does.

Lesson 7 - String and character variables

Friday, 03 March 2017 12:00 PM

The statement cin.get( ); removes the newline character left behind by >> so that getline can input up to the next newline character.

The char type is used to store individual characters, as opposed to the string type, which is used for multiple characters.

Programming concepts

We can declare string variables like any other variables. We just have to include the standard header file <string>. (This is because the string type is not a primitive type like ints, floats and chars.)

The only operator that is available for string values is the + operator. It is used to concatenate two strings together. The compiler decides which version of + to use (for ints, floats or strings) by the operands provided on either side of it.

We can use a cin statement to input a value into a string variable. However, we have to use getline to input a string containing spaces into a single string variable. The format of a getline statement is

```
getline(cin, StringVariable, Delimiter);
```

Where StringVariable is declared as string, and Delimiter is a single character, usually the newline character '\n', to specify that everything up to that character must be stored in StringVariable.

We have seen that the >> operator does not remove the newline character from the input stream when used with cin. (It gets everything up-to-but-not-including the newline character.) This causes a problem when getline is used after >> - getline doesn't input anything because it immediately encounters a newline character. To work around this problem, we use cin.get( ); to remove the newline character left in the input stream after >>, before we use getline.

A string is made up of characters. There is a primitive type called char that we can use to declare a variable to store a single character. We can input a single character into such a variable, and output single characters. We use single quotes to specify a character literal, and double quotes to specify a string literal. If a string literal consists of a single character, we prefer to specify it as a char with single quotes.

\t This represents the tab character. It will cause the cursor to jump to the next tab column if it is inserted in the output stream.

\n This represents the newline character. If it is inserted into the output stream, it will cause the cursor to jump to the next line.

\" This represents the double quote character. It is necessary if you want to include the double quote character inside a string. It is an escape character because without a preceding backslash, a " will be interpreted as the beginning or end of a string.
\\ This represents the backslash character. You always have to use two backslashes if you want one, because a single backslash will be interpreted as signifying an escape character to follow.

Lesson 8 - If statements

Friday, 03 March 2017 12:00 PM

Structure of an If statement:

```
if (Condition)
     Statement1;
else
     Statement2;
```

The Condition is something that can either be True or False. It usually involves a relational operator, i.e. one of the following:

< is less than

<= is less than or equal to

> is greater than

>= is greater than or equal to

== is equal to

!= is not equal to

These relational operators also work with integer and string values and variables.

Statement1 and Statement2 can be single C++ commands, or compound statements. (A compound statement is where we combine a sequence of statements between braces.) For example:

```
if (Condition)
{
     StatementSequence1;
}
else
{
     StatementSequence2;
}
```

The else part can also be left out:

```
if (Condition)
     Statement;
```

or

```
if (Condition)
{
     StatementSequence;
}
```

Programming principles:

Beware of the difference between the assignment operator = and the relational operator ==.

Lesson 9 - While loops

Friday, 03 March 2017 12:01 PM

Structure of a While loop:

```
while (Condition)
     Statement;
```

The Condition is a relational expression that can be either True or False, and Statement can be a single statement or a compound statement. Statement, which we also call the body of the loop, executes repeatedly, as long as Condition is True.

The variables that occur in Condition are called the control variables, and are used to control the number of repetitions.

Programming principles:

There are three important aspects about the control variable(s) that you should always keep in mind when you use a while loop:

1. The control variable (or variables) that appear in the *Condition* should always be initialised correctly before entering the while loop for the first time.
2. The condition of the while loop must test the control variable, specifying the values for which the loop should keep on repeating.
3. The body of the while loop must contain a statement which changes the value of the control variable (or variables) so that *Condition* becomes False sooner or later. If you do not do this, the program will become stuck in a never-ending loop.

Lesson 10 - Program debugging

Friday, 03 March 2017 12:02 PM

Programming errors:

Programming errors fall into one of three categories:

- **Syntax error:** Compiler checks for syntax errors and gives error messages that often make it easier to find the error.
- **Run-time error:** Occurs when a program is running and invariably causes the program to crash and a nasty error message to be displayed.
- **Logical error:** Occurs when program gives the wrong results. The program does not crash.

If statement can be used inside another if statement.

Lesson 11 - Boolean values

Friday, 03 March 2017 12:14 PM

Programming concepts:

C++ provides two Boolean values, namely true and false. They are generally used to determine the value of a condition of an if statement or a loop.

Internally, true and false are stored as 1 and 0 respectively.

One can in fact provide an integer value wherever a Boolean value is required. In this case, all non-zero values are treated as true, and 0 is treated as false.

The result of a relational operation (i.e. an expression involving one of the relational operators ==, <, > etc.) is a Boolean value.

The following Boolean operators are available: &&, || and !, representing logical AND, OR and NOT respectively. Their operands are Boolean values, and the result of the operation is a Boolean value.

C++ uses short-circuit Boolean evaluation. This means that if && or || are used in a Boolean expression, the whole expression might not be evaluated. In particular, if the first operand of && evaluates to false, the second operand is ignored and the whole expression is evaluated to false. If the first operand of || evaluates to true, the second operand is ignored and the whole expression is evaluated to true.

We can declare Boolean variables (of type bool), and we can assign values to them in assignment statements. We can use Boolean variables in Boolean expressions involving the Boolean operators &&, || and ! together with other Boolean variables or Boolean expressions provided by relational operations.

Programming principles:

It is sometimes useful to introduce a Boolean variable in a program to provide the condition (or part of a compound condition) of an if or while statement.

An if..else statement which simply assigns the value true or false to a Boolean variable can always be rewritten as a single assignment statement. Consider the following examples:

if..else	Equivalent assignment statement

`if (x >= 0)`
`  nonnegative = true;`
`else`
`  nonnegative = false;`
	nonnegative = x >= 0;

`if (rainy || windy)`
`  swimming = false;`
`Else`
`  swimming = true`
	Swimming = !(rainy || windy);

It is better to use a single assignment statement like these examples rather than an if statement with two assignment statements.

It is poor programming practice to use an integer value for the condition of an if or while statement.

Lesson 12 - Nested if statements

Friday, 03 March 2017 12:16 PM

Programming concepts:

Simple if **statements**

The if statement:

```
if (Condition)
     Statement1;
```

The if..else statement:

```
if (Condition)
     Statement1;
else
     Statement2;
```

Condition is tested, and if it is true, Statement1 is executed. If it is false and there is an else part, Statement2 is executed, otherwise the statement following the if statement in the program is executed. Statement1 and Statement2 can also be compound statements (i.e. a number of statements placed between braces). These statements may also be other if statements, in which case we are dealing with nested if statements.

Nested if **statements**

A	B	C

`if (Condition1)`
`  if (Condition2)`
`  Statement1;`
`  else`
`  Statement2;`
`else`
`  Statement3;`
	`if (Condition1)`
	`  if (Condition2)`
	`  Statement1;`
	`  else`
	`  Statement2;`
		`if (Condition1)`
		`{`
		`  if (Condition2)`
		`  Statement1;`
		`}`
		`else`
		`  Statement2;`

The first part of the first if..else statement in A contains another if..else statement. Since there are two if statements and two else parts, it is fairly clear which else part belongs to which if statement. On the other hand, statement B is an if statement with a nested if..else statement in it. It is clear from the way we laid it out that the else belongs to the inner if statement, but how does the C++ compiler determine this? The rule is that an else belongs to the nearest if preceding it, which does not have another else part of its own. If we want the else part in code B to belong to the first if, we must place the nested if statement between braces. Code C shows what the resulting code should look like.

Multiple alternative decisions

Nested if statements can become quite complex when you have to make provision for more than three alternative options. In the situation where all the else parts (except perhaps the last one) are followed by an if..else statement, the nested if can be coded as a multiple alternative decision, with

the following structure:

```
if (Condition1)
     Statement1;
else if (Condition2)
     Statement2;
else if (Condition3)
     Statement3;
:
else if (ConditionM)
     StatementM;
else
     StatementN;
```

An if statement of this form is interpreted as follows: The conditions Condition1, Condition2, ... are evaluated from the top until one of the conditions is true. The statement in the corresponding if statement is then executed and the rest of the multiple alternatives are ignored. If all the conditions are false, the statement in the else part of the last if statement is executed.

Simplification of if **statements**

```
if (Condition1)
     if (Condition2)
          if (Condition3)
                    Statement;
```

Such an if statement can often be simplified by using the boolean operator &&:

```
     if (Condition1 && Condition2 && Condition3)
          Statement;
```

Lesson 13 - Switch statements

Friday, 03 March 2017 12:17 PM

Programming concepts

A switch statement is used when a program must choose between more than two possible routes, and precisely one of the alternative options must be chosen. The structure of a switch statement is as follows:

```
switch (Selector)
{
     case Label1:
          Statements1;
     case Label2:
          Statements2;
     :
     case LabelN:
          StatementsN;
     default:
          StatementsD;
}
```

The *Selector* expression must be of an ordinal data type. In other words, it may not be a floating point expression or a character string.

Label1, *Label2*, ... and *LabelN* may only take values of the same type as *Selector*.

The same value may not occur in more than one of the labels.

Statements1, *Statements2*, ... can be zero or more C++ statements, including if statements or while loops. It is not necessary to place multiple statements in braces.

The value of *Selector* is compared to each of the labels (from the top) until the value of *Selector* matches a *Label*. If the value of *Selector* matches *Label3* for example, *Statements3* will be executed.

All statements in *all* cases under a matching label are executed. The break command must therefore be used to break out of the switch statement if you want to prevent further statements in subsequent cases from being executed.

The default part is optional. If present, it is always executed unless a break has been encountered beforehand.

A switch statement can only be used if all the alternatives depend on the value of the same ordinal variable (or expression). If the boolean expressions on which the choices are made depend on different variables or on a floating point variable, nested if statements must be used. Any switch statement can be replaced by a (nested) if statement, but not always the other way round.

Lesson 14 - More while loops

Friday, 03 March 2017 12:18 PM

When using a while loop, remember to always apply the three rules:

1. The variable(s) that appear in the loop condition (i.e. the loop control variable(s)) must be initialised when the while loop is first encountered.
2. Test the loop control variable(s) in the condition of the loop. The condition must specify the values of the control variable(s) for which the loop must continue repeating, and hence (implicitly) the values for which the loop must terminate.
3. Inside the body of the loop, the value of the loop control variable(s) should be changed to ensure that the loop condition becomes false at some stage.

This is called the ITC principle (for *Initialise-Test-Change*).

Programming concepts:

Block variables are variables that are declared in a block. The body of a loop (or in fact any sequence of statements enclosed in braces) represents a block. Block variables hide variables with the same name that are declared in the enclosing function.

A shorthand way of inputting values in a loop is to use a while loop of the following form:

```
while (cin >> Variable)
    Statement;
```

Variable can be a variable of any type, and Statement can be any statement or block of statements enclosed in braces. The user has to press <Ctrl+D> to terminate input and cause the condition of the loop to become false.

A do..while loop has the following structure:

```
do
    Statement;
while (Condition);
```

Statement can be a single statement or a block of statements enclosed in braces.

After execution of *Statement*, the *Condition* is tested. If it is true, the loop is repeated, otherwise it ends.

A do..while loop is a *posttest* loop, which means the statements inside the loop execute before the loop condition is tested. A while loop, on the other hand, is a *pretest* loop - the loop condition is tested before the statements inside the loop execute.

Programming principles:

Rules of using do..while **loops**

Very often it is not necessary to initialise the control variable(s) of a do..while loop before the loop, since its value can be set in the body of the loop. We therefore only have two rules for a do..while loop:

1. Inside the body of the loop, the value(s) of the loop control variable(s) must be changed to ensure that the loop condition becomes false at some stage.
2. The condition of the do..while loop must test the values of the loop control variable(s).

Counter-driven while **loops**

In a counter-driven loop the number of iterations is determined by a variable that keeps count of the number of times the loop is executed. Before the loop, the counter is given an initial value and is then incremented on each iteration. When this variable reaches a certain value, the loop ends. To use a counter-driven loop, the exact number of times the loop should execute should be known before the loop commences.

Example:

```
count = 1;
while (count <= 10)
{
     cout << "Iteration number " << count << endl;
     count++;
}
```

Accumulation of a sum

We often use while loops to accumulate the sum of a sequence of values. In such a loop some variable acts as an accumulator. This variable is initialised to 0 before the loop and then, inside the loop, the values of which the sum must be calculated, are added to it one-by-one.

Example 1:

```
count = 1;
sum = 0;
while (count <= 10)
{
     cin >> value;
     sum += value;
     count++;
}
```

Similarly, we can calculate the product of a sequence of values using a while loop. In this case, however, the accumulator variable must be initialised to 1 and not to 0.

Example 2:

```
count = 1;
product = 1;
while (count <= 10)
{
     cin >> value;
     product *= value;
     count++;
}
```

Sentinel-driven loops

Sometimes we use a loop to input a sequence of values without knowing how many values appear in the list. One way to handle such a loop is to ask the user to enter some unique value after the last value has been given. This unique value is called the *sentinel*.

Example:

```
product = 1;
cin >> value;
while (value != 0)
{
     product *= value;
     cin >> value;
}
```

The loop executes until the value 0 is input. In other words, 0 is the sentinel.

Situation-driven loops

The number of times a loop executes is not always determined by a counter or a sentinel. A situation-driven loop ends when a specific situation is reached. It also contains one or more loop control variables, but these are not counter variables.

Example:

```
cin >> x;
while (x != 1)
{
     if (x % 2 == 1)
               x = x * 3 + 1;
     else
               x = x / 2;
     cout << x << endl;
}
```

In this example, the loop ends when the value of x becomes 1.

Loops that never execute

It is possible that the statements in a while loop never execute. Assume, for example, the loop control variable x is initialised to 20 before the loop, and the loop condition is x <= 10. The condition is false when the loop is first encountered and therefore the loop statements do not execute.

A boolean loop control variable

A while loop can also be controlled by a boolean variable. The same rules apply here as with the other while loops: The boolean variable must be initialised before the loop, and inside the loop it

should be changed. For example,

Example:

```
again = true;
while (again)
{
     //Do something
     cout << "Do you want to do it again (Y/N)? ";
     cin >> answer;
     if (answer == 'n' || answer == 'N')
              again = false;
     }
```

Initially, again is true. The //Do something comment represents any C++ statement(s). The other statements inside the loop ask the user to type Y or N. Only when the user types N or n will again get the value false. The loop will then end.

Do..while vs while loops

A do..while loop is suitable when we are not certain how many times the loop should execute, but we are sure it must be executed at least once. If there is a possibility that the statements will not be executed at all, use a while loop.

Block variables

There are three disadvantages of using block variables in the body of a loop:

- They are inaccessible after the loop because they are destroyed at the end of the block.
- (ii) They make a program inefficient because every time a block variable declaration is encountered, memory has to be set aside for the variable. And every time the end of the block is reached (i.e. the end of the body of the loop) the variable is destroyed.
- (iii) They hide variables with the same name declared in the enclosing function. They are a common cause of logic errors in a program.

Lesson 15 - For loops

Sunday, 12 March 2017 9:23 AM

Programming concepts

The for loop is a counter-driven loop. The most common structure of a for loop looks as follow:

```
For (int Counter = InitialValue ; Counter <= FinalValue ; Counter++)
    Statement;
```

Counter is the loop control variable that the for statement initialises to *InitialValue*, and increases by 1 until the value reaches *FinalValue*. *Statement* is executed for each value of *Counter*.

In this form, *Counter* is an integer variable, and *InitialValue* and *FinalValue* are variables, literals or expressions of the same type. If *InitialValue* is greater than *FinalValue*, the loop is never executed. If the two values are equal, the loop executes only once.

We can get a for loop to count down by decrementing *Counter* in the last part of the for statement. The condition part of the for loop should be changed accordingly. In other words, the relational operator that tests whether *Counter* has reached *FinalValue*, will need to be changed.

Any variables declared in the body of a for loop are inaccessible after the loop.

If the counter variable is declared in the initialisation part of a for loop, it is inaccessible after the for loop. If it is declared before the for loop, its value will be 1 more than the final value specified in the condition part of the for loop (or 1 less if the control variable is decremented).

The control variable need not be of integer type, and it can be initialised with any (complex) assignment statement. The condition for termination of a for loop can also be a complex condition involving numerous variables. Also, the third part of the for statement need not be a simple increment or decrement statement. It can in fact be any C++ statement. The body of a for loop can contain a break statement. This will cause the loop to terminate prematurely.

Programming principles

Use a for loop when you can determine the number of iterations before the loop commences (as opposed to using a while loop when you can't determine the number of iterations before the loop commences).

The body of the for loop can use the counter variable, but changing the value of the counter variable in the body of the loop can give rise to unwanted side-effects. In general it is considered poor programming practice to change the value of the counter variable inside the body of a for loop.

Changing the values of any variables used in the condition (for determining loop termination) in the body of the loop will affect the number of iterations. In general it is considered poor programming practice to change such variables inside the body of a for loop.

Any relational operator can be used to test whether the counter variable has reached the final value, but it is dangerous to use != because if the counter variable manages to jump over the exact final value, the loop will not terminate. Rather use <, <=, > or >= as appropriate.

Although the for loop is very flexible in terms of how the control variable is initialised, tested and changed, we prefer to stick to the simple structure given at the beginning of the **Programming concepts** section above. If you do not need a counting loop, or you need to use two control variables, or you need to terminate a for loop prematurely (i.e. before the counter has reached its final value) rather use a while loop.

Lesson 16 - Nested loops

Sunday, 12 March 2017 9:24 AM

Programming concepts

Like if statements, loops can be nested. Nested loops consist of an outer loop with one or more loops that form part of its body. With each iteration of the outer loop, the inner loop is executed from the beginning and all its iterations are carried out.

Programming principles

Just because a program needs more than one loop, doesn't mean that they should necessarily be nested. Perhaps they should be executed consecutively.

Lesson 17 - Using functions

Sunday, 12 March 2017 9:26 AM

Programming concepts

We have seen that a function is called by using its name with a list of parameters in parentheses. If there are no parameters (as in the rand function) then the parentheses are left empty. If the function takes a parameter (as in time and abs) then a value must be provided in the form of a variable or an expression. When functions (like time, rand and abs) return a value, we must do something with the value. There are many possibilities:

- We can call a function in an assignment statement:
  ```
  VariableName = FunctionName(Parameter);
  ```
  (Of course, VariableName must have been declared as a variable already.)

- The value returned by the function can be used in a more complicated expression, for example
  ```
  VariableName = 20 * FunctionName(Parameter) - 41;
  ```
  as we did with the rand function.

- We can also use a function call to provide a value to be output:
  ```
  cout << FunctionName(Parameter) << endl;
  ```

The abs function returns the absolute value of its integer parameter.

The rand function returns a large random integer value each time it is called. However, the first value it returns is always the same, and the values after that follow the same sequence unless the random number generator is seeded with the srand function. The time function returns a large integer representing the value of the internal clock.

Programming principles

Although we can use a function call to provide a value to be output as explained above, in general we prefer to introduce an additional variable and assign the value returned by the function to the variable before outputting its value.

To generate random numbers in a program, we use the srand, time and rand functions. We firstly use the srand and time functions to seed the random number generator with the statement:

```
srand(time(0));
```

In this statement, the time function returns an integer value (depending on the date and time of day) which is passed to the srand function. This ensures that the rand function (which should only be called after this statement) will not return the same random number each time the program is executed. The above statement only needs to be executed once (somewhere near the beginning of the program) even if the program needs to generate many random numbers with the rand function.

Random numbers are then generally created with a statement like:

```
RandomInteger = Smallest + Interval * (rand( ) % Many);
```

Where

- *Many* is a literal value or variable representing how many possible values (i.e. the range of values that) are required,
- *Smallest* is a literal value or variable representing the smallest value in the required range
- *Interval* is the gap between values in the range.

Say for example you want to generate random even numbers between 20 and 100. Firstly, count how many possible values you want (there are 41 - check this), the smallest value is 20 and the interval is 2. Then the statement

```
randomEven = 2 * (rand( ) % 41) + 20;
```

will give a random value that complies with these restrictions.

Lesson 18 - Writing functions

Sunday, 12 March 2017 11:32 AM

Programming concepts

The general layout of a function is

```
ReturnType FunctionName(ParameterList)
{
    Statements;
}
```

Statements should contain (at least one) return statement that returns a value matching *ReturnType*.

Note how the main functions that we have been writing have all complied with this structure. The return
type is int, the parameter list is empty, and the program consists of C++ statements including a return
statement that returns a value matching the return type of main (i.e. int).

The format of a return statement is

```
return ReturnValue;
```

where *ReturnValue* can be an expression with the same type as the *ReturnType* of the function.

When a return statement is executed, the function terminates immediately, and any other statements in the body of the function are skipped.

Apart from these features, functions must be used in the following ways:

- When calling a function, the parameters must correspond: There must be the same number of actual parameters in the calling statement as there are formal parameters in the subprogram header, and their types must match.
- A function is called by using its name in an expression, much as one uses a variable name except that the name must always be followed by parentheses containing the actual parameters (if any).
- Each function is an independent subprogram unit. This means that local variables of a calling function can only be accessed by that function itself and not by the functions that it calls. If such functions need the values of local variables, they must be sent as parameters.

Programming principles

Like variables, it's good to choose descriptive names for functions. The name of a function should describe the value that is returned, since the function call will generally be used to provide a value in an expression. Also, to avoid confusion between the names of actual and formal parameters, we follow the convention of using different names for the formal parameters of a function from all other variables in a program, especially the actual parameters.

Lesson 19 - Local and global variables

Sunday, 12 March 2017 3:13 PM

Programming concepts

A global variable is a variable that is declared before all function definitions, and can be accessed by all functions in the program. A local variable is declared in the body of a function, and can only be accessed by that function.

Programming principles

In this lesson we have seen some important things about functions and parameters:

- We should use local rather than global variables when variables belong to specific functions. This prevents functions from inadvertently changing the values of variables needed in other functions.
- We should use parameters rather than global variables when more than one function needs access to the same variables. This makes it clear which variables need to be shared or which values need to be transferred from one function to another, and hence increases the readability and reliability of programs.

If the above rules are followed, the formal parameter list serves as a clear indication of what values a function requires to do its work. Even though it may use local variables to do its work, this is no concern of the rest of the program. This makes a program more readable, and allows a function to be re-used in other programs.

Lesson 20 - Void functions

Sunday, 19 March 2017 11:44 AM

Programming concepts

We can design a void function to receive one or more values from a calling statement, perform some processing on them and then display some output on the screen. We do this by specifying the types and names of the parameters in a list of formal parameters in the function header. The general layout of the header of a void function is

void *FunctionName(FormalParameterList)*

Or in more detail

void *FunctionName(Type1 FormalParameter1,*
Type2 FormalParameter2,
:
TypeN FormalParameterN)

To cause the function to be executed, the values to be sent to the function need to be specified as actual parameters via a function call which acts as a statement on its own:

FunctionName(ActualParameterList);

Or

FunctionName(ActualParameter1, ActualParameter2, ...,
ActualParameterN);

When the function is called, the value of *ActualParameter1* is copied to *FormalParameter1*, the value of *ActualParameter2* to *FormalParameter2*, etc. C++ is very precise about this. The transfer of values between the actual and formal parameters always takes the order into account. In other words, the first actual parameter always goes to the first formal parameter, and so on. As a result, there must always be the same number of actual parameters in the calling statement as the number of formal parameters in the function header, and their types must match, otherwise C++ will give an error message.

We said above that the body of a void function does not contain a `return` statement. This is not strictly true. A void function may contain an `empty return` statement, namely `return;`. This will cause immediate termination of the function. We, however, do not use it.

Programming principles

We use void functions when we have some code that is repeated in our program, or when a group of statements belong together to perform a subtask, or when input or output has to be done.

Lesson 21 - Reference parameters, part1

Sunday, 19 March 2017 11:59 AM

Programming concepts

This lesson introduced reference parameters. They are used to allow a called function to change the values of variables declared in the main function. When a function changes the value of one of its reference parameters, it actually changes the value of a variable declared in the main function that is used as the corresponding actual parameter. (As you will remember from the previous lesson, value parameters contain copies of values sent from the main function to the called function. So any changes made to value parameters within a function have no effect on any variables in the main function.)

To specify a reference parameter, we use the & symbol between the type and the name of the parameter in the formal parameter list in the function header.

```
void FunctionName(ParameterType & ParameterName)
```

A function can have more than one reference parameter, or a mixture of value and reference parameters.

Programming principles

The name of a void function should describe what it does, so the name is generally a verb or contains a verb (e.g. `displayRow` or `calcAverage`). The name of a value-returning function should describe the value that it returns, so the name is generally a noun (e.g. `average`) or an adjective (e.g. `smallest`).

Sometimes it is difficult to decide whether a function should be a value-returning function or a void function with a reference parameter. A general rule of thumb is to ask yourself what is the most important: what the function does or what it returns? If what the function returns is more important than what it does, it should probably be a value-returning function. On the other hand, if the function contains lot of processing or does any input or output, a void function is preferable.

Also, if a function needs to return more than one value, make it a void function and use two or more reference parameters. Don't use reference parameters with a value returning function.

Don't use a reference parameter if a value parameter would work. In other words, if the function does not change the value of a parameter, don't make it a reference parameter.

Lesson 22 - Reference parameters, part 2

Sunday, 19 March 2017 5:22 PM

Programming concepts

In the previous lesson we saw that functions can be used to initialise variables declared in the main function by means of reference parameters. In this lesson we saw that a reference parameter can also be used to update a variable, i.e. use its old value to calculate a new value.

Whenever the value of a reference parameter is changed within a function, the value of the corresponding variable in the main function also changes. If the value of a value parameter is changed, no values of variables in the main function change.

Programming principles

If a function should update the value of a variable, use a void function with a reference parameter. If a function should use the value of a variable and calculate a separate value, leaving the variable unchanged, use a value returning function with a value parameter.

Think carefully about what should be local variables, what should be value parameters and what should be reference parameters.

Lesson 23 - Variable diagrams (again!)

Sunday, 26 March 2017 11:30 AM

Programming concepts

This lesson shows us how useful variable diagrams can be in investigating how a program works, and how they help to clarify several things, particularly about using functions with parameters:

- Parameters are transferred on the basis of their order, not by name. For example, the function call `calcSquare1(y, x)` will give a different result to `calcSquare1(x, y)`, even if the formal parameters of `calcSquare1` are called `x` and `y`.
- Value parameters make a copy of the values sent to them. Additional memory is required for this. They act as local variables for the duration of the function.
- Value parameters are destroyed when the function terminates, i.e. their values are lost. If we want a
  function to change the values of variables in the calling function, we must use reference parameters.
- The names of local variables in the calling function are hidden while another function is being called.
  Although their memory positions are still allocated, they are inaccessible.
- The only way to access a local variable of a calling function from the function it calls is to use a reference parameter. This gives access to the memory position via another name, i.e. the name of the (formal) reference parameter.
- The memory position of a reference parameter is not destroyed when a function terminates. Only the name is no longer valid.

Programming principles

It is important always to initialise variables before trying to use their values.

Don't use the same names for actual and formal parameters (and any other variables in the program, for that matter).

Lesson 24 - One-dimensional arrays

Sunday, 26 March 2017 1:31 PM

Programming concepts

In this lesson we saw how a set of values of the same type can be stored in a data structure called an array.

An array is declared as follows:

```
Type ArrayName[NumValues];
```

The square brackets indicates that *ArrayName* is an array. *Type* indicates the type of values to be stored in
the array. *NumValues* indicates the number of values that can be stored in the array.

We can also initialise an array during declaration:

```
Type ArrayName[] = ListOfValues;
```

ListOfValues is a list of *Type* literals, separated by commas. When initialising an array in its declaration, *NumValues* must correspond with the number of values in *ListOfValues*.

We refer to specific elements in the array with, for example, *ArrayName[Index]*, where *Index* may assume any value in the range from 0 to 1 less than the number of values in the array.

As an alternative to initialising an array in its declaration, we can use a for loop to input values into the array. For example, if array a's size is 10, we can use the following for loop to input its values:

```
for (int i = 0; i < 10; i++)
{
cout << "Enter a value: ";
cin >> a[i];
}
```

Similarly we can use a for loop to display the values of an array:

```
for (int i = 0; i < 10; i++)
     cout << a[i] << endl;
```

Note that it is not possible to input the contents of an entire array from the keyboard with a single statement like `cin >> a;`. Neither can we display all the elements of an array with a statement like `cout << a;`. To input or output the values of an array, we have to do so one element at a time. It is also not possible to assign the contents of an entire array to another similar array, e.g. `a = b;`. This has to be done one element at a time.

Programming principles

An array is used when a group of related data elements must be stored for later use in the program. Sometimes, even when a program needs to work with a lot of different data values, an array is necessary. For example, say a program must calculate the sum of 20 values. It is not necessary to store the values in an array; the sum can be calculated as the values are input. There is no need to have the values available later on. On the other hand, if all twenty values are required later on, it will be necessary to have access to all the values that were input, and an array will be needed.

Rather declare a constant representing the number of values to be store in the array than specify it

with a literal value.

```
const int NUM_VALS = NumValues;
Type ArrayName[NUM_VALS];
```

This makes it easier to change the number of values to be stored in the array if needed, and makes it easier to see what is being done when the number of values is used in other parts of the program, e.g. in controlling the number of repetitions of a loop.

If the values that need to be stored in an array are known before the program starts, rather initialise the array in its declaration than input them via the keyboard.

If the exact number of values that will be stored in an array cannot be determined, declare the array to make provision for the maximum number of elements. Then use a `while` or `do..while` loop to input and store the values in the array. You'll also have to decide how to determine when all the values have been input. One way is to ask the user repeatedly whether there are more values. Another way is to include a sentinel in the input values to signal the end of the input.

If certain elements of an array are unused (i.e. uninitialised), you (i.e. the programmer) must make sure that these elements are never referred to in the program.

Also, beware of allowing an index to go out of range. The program will work with memory positions with unpredictable values, and which can change unpredictably during execution of the program.

Lesson 25 - Arrays as parameters

Friday, 14 April 2017 12:03 PM

Programming concepts

This lesson focussed on the use of arrays as parameters in function. The general form of a function that takes an array as parameter is

```
ReturnType FunctionName(Type ArrayName[])
{
    Statements;
}
```

There is no point in specifying the size of the array parameter in the square brackets, C++ simply ignores it. If the function requires the size of the array, it should either be specified as a global constant before the function definition, e.g.

```
const int NUM_ELEMS = NumElems;
```

or the size can be sent as an additional parameter:

```
ReturnType FunctionName(Type ArrayName[], int NumElemsP)
{
    Statements;
}
```

In C++, all array parameters are passed by reference. This is to save copying the entire contents of an array as would have to be done if it were passed by value. The & operator is not required (and should not be used) with an array parameter.

To prevent a function from changing the values of an array parameter, the `const` reserved word has to be used before *Type* in the parameter list. This is called *pass by constant reference*.

Functions cannot return an array as the return type. In other words, we cannot use a value-returning function to return a whole array to a calling function.

Programming principles

The decision of whether the size of the array should be specified as a global constant or as a separate integer parameter depends on whether you want to allow the function to deal with arrays of different sizes, or restrict the number of elements that should be accessed.

Lesson 26 - Two-dimensional arrays

Sunday, 16 April 2017 2:43 PM

Programming concepts

Data is table form can be stored in a two-dimensional array. A two-dimensional array can be viewed as a one-dimensional array of which the elements are also one-dimensional arrays.

The declaration of a two-dimensional array is like a one-dimensional array. The difference is that two subscripts must be used for a two-dimensional array. When declaring a two-dimensional array, the size of the row and column subscripts must be specified, e.g.

```
Type ArrayName [NUM_ROWS][NUM_COLS];
```

NUM_ROWS and *NUM_COLS* can be integer literals, constants or variables.

Two-dimensional arrays with the same number of rows and columns are often referred to as *matrices*. An NxN matrix is a two-dimensional array with N rows and N columns.

A two-dimensional array can be initialised in its declaration statement. Since a two-dimensional array is really a one-dimensional array where each element is another one-dimensional array, each row of the array is listen in braces (curly brackets) and they are all put in braces. For example, the following will declare and initialise a 2x3 array of integers:

```
int mat2x3[][3] = {{1, 2, 3}, {4, 5, 6}};
```

The number of rows need to be specified in the declaration (as above) but the number of values in each row (i.e. the number of columns) must be specified, even if it is obvious from the initialisation list.

When passing a two-dimensional array as a parameter to a function, you need not specify the size of the first index (the number of rows) but you MUST specify the size of the second index (the number of columns) between the relevant square brackets of the formal parameter. The standard technique is to declare global constants specifying the number of rows and columns and using these in the function. The format therefore looks like this:

```
const int NUM_ROWS = NumRows;
const int NUM_COLS = NumCols;
:
void FunctionName(Type ArrayName[][NUM_COLS])
     :
```

The function can then use *NUM_ROWS* and *NUM_COLS* to ensure that it only accesses values in valid indices of the array.

If you want to allow the function to deal with arrays of different sizes, you can send an additional integer parameter specifying the number of rows:

```
const int NUM_COLS = NumCols;
:
void FunctionName(Type ArrayName[][NUM_COLS], int NumRows)
     :
```

Note that this only allows the function to deal with arrays with different numbers of rows. You CANNOT get a function to deal with arrays of different numbers of columns, since the number of columns MUST be specified in the square brackets for the seconds index as a constant or literal.

The above trick (of sending an additional parameter) is also useful for getting a function to only look at a restriction number of rows of an array. In other words, by providing a value for *NumRows* which is less than *NUM_ROWS*, we can get the function to only access some of the rows of the array.

Programming principles

We generally specify the size of the indices (i.e. the number of rows and columns) of a two-dimensional matrix as constants (like *NUM_ROWS* and *NUM_COLS*). This makes it much easier to change the size of the array if we ever need to in the future.

To step through the values of a two-dimensional array, we use two loops with one nested inside the other. To step through them row-by-row, the outer loop determines the row subscript and the inner loop determines the column subscript. For example:

```
for (int i = 0; i < NUM_ROWS; i++)
{
     for (int j = 0; j < NUM_COLS; j++)
          cout << ArrayName[i][j] << '\t';
     cout << endl;
}
```

To step through the values column-by-column, the loops will change as follows:

```
for (int j 00; j < NUM_COLS; i++)
{
     for (int i = 0; i < NUM_ROWS; j++)
          cout << ArrayName[i][j] << '\t';
     cout << endl;
}
```

Lesson 27 - String manipulation

Sunday, 16 April 2017 4:04 PM

Programming concepts

We can refer to the individual characters of a string by using a subscript in square brackets after the name of the `string` object. For example, `oneWord[4]` refers to the fifth character in the string `oneWord`. This is because the subscript of a string starts at 0.

The `string` class has a number of member functions (i.e. functions associated with the class) that we can use to manipulate string values. To call a `string` member function, we use the dot operator between the `string` object and the member function name:

`StringObject.MemberFunctionName(Parameters)`

Some member functions change the value of the `string` object on which they operate and others do not. We call member functions that change an object *mutators* and member functions that do not, *accessors*. For example, `size`, `substr` and `find` are accessors and `insert`, `erase` and `replace` are mutators.

The `size` member function

Format:	*`StringObject`*`.size()`
Operation:	This function returns an integer representing the number of characters in the string *`StringObject`*.

Example:	`sentence = "How many?";`
	`length = sentence.size( );`
	will assign the value 9 to `length`.

The `substr` member function

Format:	StringObject.substr(StartPos, Length)
Operation:	This function starts at position *`StartPos`* and returns a substring, *`Length`* characters long, from *`StringObject`*. *`StartPos`* and *`Length`* are integers. If *`Length`* specifies more characters than there are left from *`StartPos`* onwards, all the remaining characters are returned. An alternative version of `substr` only takes a single parameter, namely *`StartPos`*, and returns all characters in the string from *`StartPos`* to the end.

Example:	`sentence = "This is copied";`
	`word = sentence.substr(0, 4);`
	will assign the value `"This"` to `word`.

The `find` member function

Format:	*`StringObject`*`.find(`*`Substring`*`)`
Operation:	This function searches from left to right for the first occurrence of *`Substring`* in *`StringObject`*. If found, the function returns the position of the first character of the first occurrence. If not found, the function returns -1. An alternative version of find has two parameters. The optional second parameter is an integer representing the starting position for the search.

Example:	`sentence = "This is it";`
	`position = sentence.find("is");`
	will assign the value 2 to `position`, and
	`position = sentence.find("is", 6);`
	will assign the value -1 to `position`.

The `insert` member function

Format:	*`StringObject`*`.insert(`*`InsertPos, Substring`*`);`
Operation:	This void function inserts *`Substring`* at *`InsertPos`* in *`StringObject`*.

Example:	`sentence = "No sense";`
	`sentence.insert(2, "w it makes");`
	will change the value of sentence to `"Now it makes sense".`

The `erase` member function

Format:	*`StringObject`*`.erase(`*`StartPos, Length`*`);`
Operation	This void function starts at position *`StartPos`* and erases *`Length`* characters from *`StringObject`*.

Example:	`sentence = "Toffee apples";`
	`sentence.erase(2, 9);`
	will change the value of sentence to `"Toes".`

The `replace` member function

Format:	*`StringObject`*`.replace(`*`StartPos, Length, Substring`*`);`
Operation:	This void function starts at *`StartPos`* and replaces *`Length`* characters of *`StringObject`* with *`Substring`*.

Example:	`sentence = "Data structures";`
	`sentence.replace(1, 9, "en");`
	will change the value of sentence to `"Dentures".`

Programming principles

It's possible to write your own functions to manipulate strings. For example, you could write your own function to determine the position of one string in another string. But why do it? Rather use the member functions that are provided for the string class. You can save yourself a lot of time and effort by getting to know the available member functions, to recognise when they can be used.

Lesson 28 - Structs

Monday, 17 April 2017 11:06 AM

Programming concepts

The general layout of a struct definition is as follow:

```
struct  StructName
{
     Type1 FieldName1;
     Type2 FieldName2;
     :
     TypeN FieldNameN;
};
```

Some of the fields may have the same data types, so it's is more accurate to describe the layout as follows:

```
struct StructName
{
     Type1 FieldList1;
     Type2 FieldList2;
     :
     TypeM FieldListM;
};
```

where every *FieldList* contains the names of one or more fields of the same type, separated by commas.

A struct definition does not allocate any memory. This has to be done with declaration statement, for example:

```
StructName VariableName;
```

We refer to a specific field of a struct with *VariableName.FieldName*.

We input and output the fields of a struct individually (unless you know how to overload the stream insertion and extraction operators).

We can assign all the values of one struct (say `struct1`) to another struct of the same type (say `struct2`), with a single assignment statement such as:

```
Struct2 = struct1;
```

This saves having to copy the fields one by one.

The type of a field may be any valid C++ data type - even an array or another struct.

One characteristic of structs that was not mentioned in the lesson is that the same field name may be used in definitions of two different struct types in a program. For example:

```
Struct                          struct Trouser
{                               {
     int size;                       int legSize, waistSize;
     char style;                     char style;
     float price;                    float price;
};                              };
```

The field name is then relative to the type of the variable being used, i.e. `shirt1.price` is different from `trouser1.price`.

Programming principles

Use a struct if you have related data (probably not of the same type), especially if you would otherwise have to pass a lot of separate data values to a function.

Our convention is to use an uppercase letter for the first letter of the name of a struct type, but a lowercase letter for the first letters of the names of the fields and for the names of any struct variables.

Lesson 29 - Arrays of structs

Tuesday, 18 April 2017 7:01 PM

Programming concepts

In this lesson we looked at the use of arrays of which the elements are structs. An array of structs is declared as follows:

```
StructName ArrayName[NUM_ELEMS];
```

Each element of array *ArrayName* (i.e., *ArrayName*[0], *ArrayName*[1], etc.) is a struct of type *StructName*. To access the fields of the individual structs, we use the dot operator, e.g. *ArrayName*[*Index*].*FieldName1*, where *Index* is a subscript between 0 and *NUM_ELEMS* - 1.

Programming principles

When you find you are having to define parallel arrays (i.e. arrays with the same number of elements) think whether you can't use a struct to store the data of corresponding elements together.