C++ Fundamentals

C++ Words Continued

Last we mentioned the 3 word types in C++: Library, Reserved and user Defined. We also looked in more detail at Library and Reserved words, known collectively as keywords.
Time to look in more detail at user defined words.

Identifiers

Identifier = in computer science, an identifier is a string of bits (or characters) which name an entity, such as a program, device, or system; in order that other entities can "call" that entity. In programming languages, identifiers are lexical units which name a language object, such as a variable, array, record, label, or procedure.
Basically Identifiers are the name of an object (constant, variable, function) used in a program.

Identifiers can be used to name locations in the computers memory where:

- Changeable values (variables) can be stored
- Unchangeable values (constants) can be stored

In C++, identifiers may be composed of letters, digits, and underscores; identifiers may not begin with a digit. Obviously we cannot use keywords, as defined last week, as identifiers.

You name your own identifiers, despite this there are some general rules that should be followed when naming identifiers:

- They should be descriptive i.e. daysInYear or name
- They should generally be more than 1 or 2 characters

Remember that C++ is case sensitive so the variables:

- `int age, int Age and int AGE` are all different.

In this following example the identifier is the name of the integer variable, daysInYear:

- `int daysInYear = 365;`

Simple Data Types

What is a data type?
When we wish to store data in a C++ program, such as a whole number or a character, we have to tell the compiler which type of data we want to store. The data type will have characteristics such as the range of values that can be stored and the operations that can be performed on variables of that type.

Simple data types are classes (or types) of data. There are only four of these in C++. They are often called the fundamental built-in data types.

For each of the fundamental data types the range of values and the operations that can be performed on variables of that data type are determined by the compiler. Each compiler should provide the same operations for a particular data type but the range of values may vary between different compilers.

Integer
- Positive or Negative whole number: `-45689, 365, 0, 3`
- In C++ there are three sub-types: `int, short, long`

Real
- Positive or Negative decimal number: `3.14159, -65.4504, 87.5`
- In C++ there are three sub-types: `float, double, long double`

Character
- Letter, digit, punctuation or special symbol: `'x', ' ', '!', '\n'`
- In C++ there is one sub-type: `char`

Boolean
- logical value (`true` or `false`)
- In C++ there is one sub-type: `bool`

Each of these data types has certain characteristics; they all occupy different numbers of bytes. The number of bytes occupied can depend on the compiler, which is why we use the term typically.

Integer representation:

```
int height = 100;
int balance = -67;
```

`int, long` are typically 4-bytes, a `short` typically 2-bytes
`int` values range from around -2 billion to 2 billion

Decimal representation:

```
float celsius = 37.623;
double fahrenheit = 98.415;
```

`float` is typically 4-bytes, `double` is typically 8-bytes
`double` typically stores more significant figures than `float`
`double` is normally used for increased accuracy
Most decimal types can't be stored fully accurately anyway!

Character representation:

```
char menuSelection = 'q';
char userInput = '3';
```

`char` variable holds a single character at a time
`char` variables occupy a single byte
The stored value is a binary code representing the character, usually in ASCII

Boolean representation:

```
bool isEven = false;
bool keyFound = true;
```

`bool` variables typically occupy 1-byte

**Note: Floating point values - The range of values that can be stored in each of these is defined by your compiler. Typically double will hold a greater range than float and long double will hold a greater range than double but this may not always be true. However, we can be sure that double will be at least as great as float and may be greater, and long double will be at least as great as double and may be greater.*

All of these data types are integral parts of C++ and are therefore reserved words. They will go blue when you type the data type into your editor. You also do not need to include any additional files at the top of your program.

The String Type

Standard C++ also includes the `string` type which can be used to store a sequence of characters.
This is usually known as a character string, surprisingly!

```
string cvg;
cvg = "Computer and Video Games!";
```

The `string type` is declared in the header file `<string>`.

It can hold an arbitrary amount of characters at once.

It is actually a class, something we will explore near the end of the course, whose internal workings are abstracted from us.

The `string` data type has quite a lot of additional functionality that is useful to us.
You can check out the string class in more detail here:
http://www.msoe.edu/eecs/ce/courseinfo/stl/string.htm
If you are really keen, we will be covering it in more detail later on.

The string data type is not an integral part of C++ hence why we need to include the header file and why when we type string into our text editor it doesn't turn blue.

Variables

What are Variables?

Definition: A location in memory, referenced by a name, where a data value that can be changed is stored

Variables are blocks of memory that hold values. The values can be numbers or they can be characters. The values can change or, this is clever; 'vary'.

They are basically a place to store a piece of information. Just as you might store a friend's phone number in your own memory, you can store this information in a computer's memory. Variables are your way of accessing your computer's memory.

Declaring variables

Variables are declared by first stating what type of variable it is and then by declaring the variable name (the identifier).

Identifier declaration:

- C++ indicates that all identifiers are declared before they are used
- Declaration specifies identifier name and type
- Multiple identifiers of the same type may be declared together

Basic declaration syntax:

type identifier1, identifier2, … identifierN;

**Don't forget to end declaration with a semi-colon*

Examples:

```
int  Weight,
     Height,
```

```
     Length;
string name;
char firstInitial;
double groupAverage, GNP;
```

When declaring variables it is considered good form to give them identifiers that are indicative of the information they hold, for instance in the example above the variable is a number who' value is specified by the user, hence the identifier `usernumber`.
As seen multiple variables of the same type can be declared at once, e.g.
`unsigned int myAge, myWeight;`
The variable names have to be separated by commas after the variable type (`unsigned int` in this case) has been declared.

Variables and Memory
To understand variables exactly you need to know how computer memory works. Computer memory is a series of sequentially numbered locations. These numbered locations are known as memory addresses.

When you define variables in programming you assign a name to make it easy to find without knowing its address. E.g. for `int usernumber` usernumber is the variable name.

The initial part of the declared variable, `int usernumber`, sets out what *type* of information the variable will store and therefore how much memory needs to be allocated (set aside) by the computer.

Each memory address is one byte. Variables often comprise multiple bytes, although the exact number of bytes can vary based on the OS and compiler used.

Initialisation
Declaring a variable does not (usually) automatically provide it with a specific starting value.

Memory in its self cannot be empty….

The variable examples initialise variables whose values are random garbage, and using them will generate errors. The random number corresponds to whatever happens to be in memory at the time.

Using these un-initialised variables is one of the most common sources of logically errors.

To initialise the variable you assign it a value:

```
int  Weight = 12,
```

```
Height = 140,
Length = 30;
string name = "mark";
char firstInitial = 'D';
double groupAverage = 0.0, GNP = 0.0;
```

It is often good practice to assign an initial value to avoid errors, even if you just set everything to zero before using it.
You can always assign another value in the future.

Signed and Unsigned
Variables also come in two additional types, just to complicate things –
signed and unsigned.

Variables have a bit reserved in the memory to indicate whether they have negative or positive value.

Signed is the default selection when you declare a variable, this allows the variable to take both positive and negative values.
Unsigned variables, which have to be declared, can only take positive values.

You do not have to declare whether a variable is signed, although you can, it being the default selection. To initialise an unsigned variable we use the keyword `unsigned`.

```
signed char myChar = 100;
signed char newChar = -43;
unsigned char yourChar = 200;
char myChar = 100;  // this is signed by default

signed int index = 41982;
signed int temperature = -32;
unsigned int count = 0;
int index = 41982;  // this is signed by default
```

Assigning Values to Variables
As you have seen in the above examples we assign values to variables by using the assignment (=) operator (see below for assignment statements).

Constants

Like variables constants are storage locations. Unlike variables constants remain constant, there values not changing.
A constant is any expression that has a fixed value. They can be divided in Integer Numbers, Floating-Point Numbers, Characters and Strings.

There are two types of constants literal and symbolic

Literal Constants

Literal constants are explicit numbers or characters, such as:

```
'z'
"Hello world"
16
-273
```

**Note: Single quotes are used to indicate a character value, and double quotes are used to indicate a string value.*

Explicit = fully revealed or expressed without vagueness, implication, or ambiguity: leaving no question as to meaning or intent
In Mathematics: defined by an expression containing only independent variables

Literal constants may be of any of the built-in C++ types.

By default decimal constants are of type `double`. Suffixing an `'f'` or `'F'` to a decimal constant will cause the compiler to interpret it as a `float`.

```
3.14159   // type double, 8-bytes storage
3.14159F  // type float, 4-bytes storage
```

Symbolic (or Named) constants

These constants are declared and referenced by identifiers, the same way as variables.

It is generally better to use named constants, rather than literal constants:

- The name carries meaning that makes the code easier to understand
- If the value is used in more than one place and needed to be updated, the value would need to be changed only in one place

There are two main types of named constants

The Keyword `const`

This is the first way of declaring constants. They are declared in much the same way as variables but you use the reserved word `const` before the other declarations.

```
const unsigned int  MAX_NUMBER = 100;
const float PI = 3.1415;
const string NAME = "mark";
```

This is generally the preferred way to define a constant; it takes longer to type than the next method but offers some advantages, the biggest being that it has a type and the compiler can enforce that it is used according to its type – in this example PI has the type float.

#Define
The Old fashion way of defining constants is using the pre-processor directive #define: This is a legacy from C.

So what does define mean and what does it do.

#define is a pre-processor directive: **#define**, that serves to generate what we called *defined constants* or *macros* and whose form is the following:

```
#define name value
```

Its function is to define a macro called *name* that whenever it is found in some point of the code is replaced by *value*.

For example:

```
#define PI = 3.1415;
```

When the pre-processor goes through the code and sees `PI` it replaces it with the value `3.1415`.

These types should be avoided in preference to the reserved word const because this technique has the problem that the replacement is done lexically, without any type checking, without any bound checking and without any scope checking. Every "PI" is just replaced by its value. The technique is outdated, exists to support legacy code and should be avoided.

Directives

What do directives do? They provide instructions to the pre-processor, part of the compilation process where by the code is processed before being converted into machine code, to edit your code file.

Although directives do not generate code, they are commonly used to incorporate code, for instance:
They are commonly used to incorporate standard header files into your program:

```
#include <iostream> // embed the file "iostream" here
```

Directives can also be used to declare identifiers:

```
// replace every occurrence of MAX_NUMBER
// in our code with the value 100
#define MAX_NUMBER  100
```

Statements

A statement is just an instruction; essentially it is an imperative sentence.

Imperative = expressive of a command; having power to restrain, control, and direct

Here are some examples of C++ statements:

```
const float PI = 3.1415;
double    Average;
Average = totalValue / numberOfMembers;
cout << "\nWelcome to Salford Uni "
      << main << ", hope you enjoy it\n"
      << endl;
```

C++ statements may be on a single line or they may be on several lines.

In every English sentence we end with a punctuation mark, usually a full stop. In C++ we end very statement with a semi-colon ‘;’.
Failure to end statements with a semi-colon is another common source of errors.

Assignment Statements

The assignment operator, equals symbol ‘=’, causes the operand on the left to have its value changed to the value on the right.

In C++ the equals symbol ‘=’ is the assignment operator it does not represent equality, as it does in maths, nor does it compare to sides of a statement.

For instance:

```
x = x +1;
```

This statement says calculate the value x + 1 (using the current value of x) and store the result in x.

Examples of assignment include:

```
double Average = totalValue / numberOfMembers;
```

An assignment statement is an executable statement; it gives the variable the value of an expression.

Semantics: The value of the expression on the right hand side is stored in the memory location referenced by the variable on the right hand side.

Using Assignment

It is good practice, as sometimes you will generate errors, to make sure that the "receiving" variable matches the type of the assignment variable.

Example:

```
int X = 2, Y = 7;
X = Y;    // variable types match
```

It may not make logical sense though for instance two integer variables weight and height could have the following applied:

```
height = weight;
```

And although this is not illegal it would make no sense to a human programmer reading the code.

Arithmetic Operators

The arithmetic operators are:

Symbol	Meaning	Example
+	Addition	9 + 7 = 15
-	Subtraction	9 – 7 = 2
*	Multiplication	9 * 7 = 63
/	Division	9 / 7 = 1.28
%	Remainder	9 % 7 = 2
-	Unary minus	-9 = -9

Operands can be of any type for which the operation makes sense.

In mathematics, an operand is one of the inputs of an operator. For instance:

3 + 6 = 9

'+' is the operator and '3' and '6' are the operands.

The number of operands of an operator is called its arity. Based on arity, operations are classified as unary, binary etc.

In mathematics, a unary operation is an operation with only one operand. For instance, logical negation is a unary operation on truth values and squaring is a unary operation on the real numbers. A unary operation on the set S is nothing but a function $S \rightarrow S$.

Parenthesis may be used to group terms into more complex expressions:

40 – (3 * 2) * 5 → 40 – 5 * 5 → 40 – 25 → 15

**Note: There is no operator in C++ for exponentiation (X^y)*

**Note: The division operator may not do what you expect if you are dividing integers. When you divide 21/5 the result is 4.2. Integers don't have fractions so the remainder (0.2) is lopped off returning the value 4.*

**Note: The modulus operator (%) returns the remainder value of integer division. So if we do 21%5 the modulus returns the value 2.*

Arithmetic Precedence

Every operator is given a precedence number and an operator with a higher precedence than its adjacent operators is evaluated before them.

For example ***** has higher precedence than **+**, so in **2 + 3 * 4**, ***** is evaluated first. If we want **+** to be evaluated first we must write **(2 + 3) * 4**

Precedence rules:

- Expressions in parenthesis are evaluated first
- Then unary –
- Then *, % and /
- Lastly + and –

Operators with the same precedence level are evaluated in left to right order (unless they are in parenthesis!).

These are the same rules used in mathematic so they shouldn't feel too abnormal….hopefully….

If you are ever in doubt use parenthesis

Examples:

21 / 7 + 5 → 3 + 5 → 8
24 / (7 + 5) → 24 / 12 → 2
1 + 2 – 3 – 3 → 3 – 3 – 3 → 0 – 3 → - 3

Incrementing and Decrementing

This is such a common procedure in programming, generally using a variable as a counter, that C++ has a short hand way of adding or subtracting 1.

`++` Increment operator

- Pre-increment if placed before the variable, post-increment if placed after

`--` Decrement operator

- Pre-decrement if placed before the variable, post-decrement if placed after

Example:

`X++` is the same as `X = X + 1;`
`Y--` is the same as `Y = Y – 1;`

The increment operators and decrement operators come into varieties *postfix* and *prefix*. With postfix the operator is written after the variable (X++), with prefix the operator is written before the variable (++X). In basic statements it doesn't matter which one you use, but in more complex statements it matters. The reason being the prefix operator is evaluated before the assignment operator, the postfix after.

Prefix semantics = Increment the value then fetch it.
Postfix semantics = Return the value then increment the original.

Example of Basic Statement:

`++x` has the same effect as `X++`

Example of Complex Statement:

```
int x = 0, y = 7, z;
z = y * x++;   // z <--- 7 * 0
z = y * ++x;   // z <--- 7 * 1
```

Arithmetic with Variables looks like strange algebra

When we perform arithmetic it often looks like algebra because we often perform arithmetic on variables, which we reference by there names, rather than numbers (or the values the store).

For instance in a game if we want to store where about in the world a player is we will have 2 or 3 variables X,Y,Z. These variables will obviously hold the necessary values. If we then wanted to update where the player was in relation to the world we would use the relevant variable and either add or subtract.

X = X + 10;
Y = Y + 5;

Sometimes though we are performing arithmetic on variables that we have named in a more descriptive way, such as:

```
int PlayerPositionX;
int PlayerPositionY;
```

Because we often use variable names that are more descriptive, for good reason, our arithmetic may of resemble the following:

```
PlayerPositionX = PlayerPositionX * PlayerMovedX;
```

So what are header files?

A header file is a file containing C++ declarations and macro definitions to be shared between several source files. You request the use of a header file in your program by *including* it, with the C++ preprocessing directive #include.

Header files serve two purposes.

- System header files declare the interfaces to parts of the operating system. You include them in your program to supply the definitions and declarations you need to invoke system calls and libraries.
- Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them.

Why use them?
Including a header file produces the same results as copying the header file into each source file that needs it. Such copying would be time-consuming and error-prone. With a header file, the related declarations appear in only one place. If they need to be changed, they can be changed in one place, and programs that include the header file will automatically use the new version when next recompiled. The header file eliminates the labor of finding and changing all the copies as well as the risk that a failure to find one copy will result in inconsistencies within a program.