C++ Language Element Summary

The following definitions are intended just to be a short summary of the features of C++ language.

Variables and data types

C and C++ support a wide range of variable types. Variables are nothing more than labelled, reserved memory locations and must be declared before they are used. This means that a specific memory location get given a name so that you can refer to the contents of that location by a human-memorable name (that's the labelling). The reservation refers to the amount of memory you set aside for this content and that is achieved during the declaration.

Different types of data take up different amounts of space and there are different operations you can do to them, so it is important to get this right. For instance, multiplication makes sense to numbers but not to words.

The most important data types you will be using for BNCS are:

Keyword	Explanation
int	Integer – your basic, whole number. Only whole numbers (no fractions) in the range from -2147483648 to 2147483647
bool	Boolean true or false. Only two values available, may be represented as 1 and 0
char	An ASCII character – one letter
float	A real number, for when you need more accuracy than an integer can provide. 4 bytes' worth of precision in the range 1.5×10^{-45} to 3.4×10^{38}, using 7 digits.
double	A proper real number, for when a float just isn't large or precise enough. 8 bytes long, range 5.0×10^{-345} to 1.7×10^{308} and 15 digits are available
enum	Enumerated – basically means a defined (by you, the programmer) set of values. Only those values are allowed – days of the week is a good example.
string	This is an extension of the char data type and is an array of characters

Variable names are case sensitive, and modern convention uses an initial lower case letter, then an upper case letter at the start of each word within the name. This is often called Camel Case to separate word elements within the name. Some example valid names are:

```
commsStatus      rxByteReady      routerSourceName
```

Some examples of declaring a variable are:

```
int rxSignalStrength;     // Normal size integer (eg 32 bits)
```

```
long errorCount;          // Wide integer (eg 64 bits)
byte myChar;              // 8-bit number
float bitErrorRate;       // A floating point value
```

In most cases it is essential to also define the initial value for the variable. If you don't explicitly give it a value it will just contain whatever that memory location held previously which will be totally random and may cause… interesting results to your code. Initialise your variables like so:

```
int commsErrorCount = 0;  // Set initial value to zero
float myPi = 3.1;         // Love the rounding error!
bool txReady = false;     // A Boolean variable
```

Where new classes are defined, we can use the variable declaration syntax to create named instances of the class. The BNCS development team knew that strings of characters are an important aspect of scripted panels, so they created a class that implements useful string processing. The class is called **bncs_string**, is an extension of the string type and we can declare instances such as:

```
bncs_string sourceName;
bncs_string tunedChannel;
```

We can also declare an array (a list) of a simple or complex variable. For example:

```
int destMap[100];         // Declare an array of 100 integers.
```

Note that the index for the array runs from 0, so the above array has elements between destMap[0] to destMap[99]. In BNCS we quite often make the array one larger than the number of stores we require. Index 0 is not used, allowing us to use standard BNCS number ranges that start at an index of 1.

C and C++ strongly encourage the use of *pointers* for some uses. A pointer is literally just a signpost to where the real data is held and may take up much less space than the real data. There is an old computing adage "Never copy data if you can change a pointer to the data". Variables that hold the address of a data element are declared using an asterisk ahead of the variable name.

```
int *status;          // Declares that status holds an pointer
                      // to an integer value
float *frequency;     // Declares that frequency holds a pointer
                      // to a floating point value.

int mystate1, mystate2, mystate3; // Declare 3 integers
status = &mystate2;   // Set integer pointer to the store for
                      // mystate2. The & means compute the
                      // address.

*status = 17;         // Sets mystate2 to a value of 17
```

Pointers are often used in calls to C or C++ functions. These processing blocks can only return a single value. When we need more values returned, we declare the function such that the

addresses at which the results should be stored are provided as part of the function call parameters.

Statements

Like most languages, C++ programs contain statements. A statement may involve a computation, an assignment of a value, a call to a processing routine, a loop control, a logical test, and several other possibilities. A statement must be terminated by a semi-colon **;**

Example:

```
newY = 2 * oldY + OFFSET ;
```

In general, whitespace (space characters, tabs) are ignored when parsing a statement. However whitespace and tabbed indentation is used to make things easier to read. The following statements are identical to the compiler:

```
newY=newY+2*myFunc(13,7,2);
newY = newY + 2*myFunc(13, 7, 2);
```

But the second layout makes it easier to separate the terms by eye.

Assignment

The assignment operator is the single equal character **=** .

```
int x = 7;
int sum = (3 * x) + 7;
```

The operator works for all types of data. If a C++ class has been defined and the equals operator has been defined (over-ridden) we can also assign complex data to storage.

```
bncs_string  mySource = "BBC ONE DTT";
```

It is not the same as the double equal character **==** which is a test, not assignment:

```
x=7;     //you assign the value 7 to the variable x
x==7     //this is a test, you are asking IF x is equal to 7
```

Operators

There are a large range of operators used in C and C++. These are the same as used in many other computer languages. Some examples are given below.

```
+    add two numbers: a + b;
-    subtract two numbers: a- b;
*    multiply two numbers: a * b;
/    divide two numbers: a / b;
%    modular division: a % b;
|    bitwise OR function.
&    bitwise AND function.
^    bitwise exclusive OR function.
```

The biggest difference in C and C++ are the options for using the symbols. Let us use the addition operator to add 3 to the initial value:

```
newSum = newSum + 3;  // Used in many languages.
newSum += 3;          // C shorthand mode – same result as above.
```

The shorthand version is something new programmers must learn to read, even if they prefer to use the clearer format of the full expression.

Increment and decrement

C and C++ also have some shorthand expressions where values are to be incremented or decremented by one unit.

```
int alpha = 6;
int result;
alpha++;          // alpha now equals 7

// Using the same definitions as above
++alpha;          // adds one to alpha
--alpha;          // subtracts one from alpha
alpha--;          // Subtract one from alpha.
```

The increment and decrement operators are often used as part of another expression. Suppose we want to calculate the value for result, then add one to alpha. Conventional programming would express this as:

```
result = 2*alpha + 7;
alpha = alpha + 1;
```

Using the statement combine capability of C and C++ many programmers would use the syntax below that produces an identical result.

```
result = 2*alpha++ + 7;   // Computes as alpha=7, result=19
```

The placement of the increment operator before or after the variable defines when it is applied. Thus the expression:

```
result = 2 * ++alpha + 7;
```

is equivalent to:

```
alpha = alpha + 1;        // alpha = 7 after this is executed
result = 2 * alpha + 7;   // result = 21.
```

New programmers should use the formulation that they find easiest to understand.

Curly Brackets {}

C and C++, like some other languages, only support a single statement in each program and function (subroutine). Clearly this is not viable – so the trick is to enclose a set of statements in brackets to make a syntactic single statement. The brackets used are Curly brackets **{** and **}**.

```
{
     int myXposition=0;
     myXposition = whereX() + 1;
     newX = myXposition + deltaX;
     newY = newY + deltaY;
}
```

Comments

You may think you have a good memory – but experience shows that all programmers do forget what they did some months ago. It makes sense to add some documentation to the code as you create it. There are two types of comment structure in C++ - line comments and block comments.

Line comments extend for a maximum of a single line (to the next carriage return, line feed). Line comments start with two forward slash characters:

```
// The content of this entire line is a comment.
myStatus = getSerialCommsState();  // Read status code from com1:
```

Block comments run over several lines, and the comment block is delimited by a /* and */ combination.

```
/*
     void function setPosition(int x, int y) is called to move the
     camera position servo to a new position. The new position is
     passed to the function in two integer parameters x and y.

     The function does not return a value
*/
```

Block comments are sometimes used during program debugging to temporarily make the code enclosed by the comment delimiters invisible to the compiler.

Pre-processor

Before the file content is handed to the compiler, the source file is run through a pre-processor function. The pre-processor allows a textual replacement to be done, and supports a processing language with conditional tests.

The text replacement function allows us to define a name that is then replaced by the defined content during the pre-processor run. The name to be defined is preceded by a hash (#) symbol. For example we can use:

```
#define SDIROUTER     400
```

Later in the code we can then use the definition in a statement:

```
routerCrosspoint(SDIROUTER, mySource, myDestination);
```

After the pre-processor has run the above statement becomes:

```
routerCrosspoint(400, mySource, myDestination);
```

If, at a later time, we need to use a different ID for the BNCS router device, we just change the #define statement value, and re-compile the code. #define statements are often used to declare status values. For example:

```
#define RX_UNKNOWN_STATUS 0
#define RX_RECEPTION_OK   1
#define RX_INPUT_LOW      2
#define RX_LOST_COMMS     3
```

By convention, the names that are defined are upper case, and may contain underscores to make the name easier to read. Using #define makes no difference to the final code sent to the compiler, it is simply used to make the programmer's life a little easier. Reading code with lots of seemingly random numbers in it is really hard and makes debugging a nightmare.

Symbols may be defined, without giving them a value. We can then test if the value is defined as part of the pre-processor syntax. This is used to prevent include header files including themselves a second time, or to enable conditional compilation.

```
#define USEDEBUG
```

then later in the program code we can use:

```
#ifdef USEDEBUG
    showDebug("Comms status is %1", CStat);
    showDebug("Input BER = %1", cBER);
#endif
```

When program debugging is complete, we change the #define line to"

```
#undefine USEDEBUG
```

Then recompile the file to remove the now unneeded code.

Comparison

We often need to compare values. C and C++ support logical tests in several ways, the most common being:

```
if (conditional test) { /* code */ }

if (conditional test) { /* code */ } else { /* code */ }

switch (myVal) { // myVal is an integer (Cardinal) value
    case 1:
        /* code for case 1 */
        break;   // End of code when myVal equals 1;
```

```
        case 2:
            /* code for case 2 */
            break;

        case 7:
            /* code for case 7 */
            break;

    }
```

The switch statement is a sort of cascaded if ... elseif … elseif structure that is easier to read. An important point to note about the switch statement is that you can ONLY switch on integers, not on strings or characters (as you can in some other languages). If the parameter you want to select for isn't an integer you'll have to think of some type of substitution.

To test if two values of the same type have identical values we would code the test as:

```
if (myStatus == 0x14) {
    /* code to execute if the test is true */
}
```

Be careful when writing comparison test – it is very easy to fall into the trap of

```
if (myStatus = 0x14) {
    /* This code is always executed. myStatus is set to hex 14.
       0x14 is not zero and thus the test evaluates as true */
}
```

Some of the comparison operators are:

```
==   test for equality
!=   test if the values are not equal (different)
<    less than
<=   less than or equal to
>=   greater than or equal to
>    greater than
```

Multi-part tests can be implemented by using logical statements. For example:

```
if ((myStatus == 0x14) && (myWait == 0x01)) { /* code */ }
```

The double AND operator (&&) tells us and the C compiler that the results of test 1 and test 2 are to be combined with a logical AND.

Loop Control

We often need to execute some code a defined number of times, other times we need to do something until a test becomes true. The control structure to repeat a code block a defined number of times is:

```
for (int i=0; i <= 25; i++;)
```

```
{
    /* code to execute */
}
```

The first part of the bracket defines a variable called i that exists only inside the loop. The second part of the brackets provides the completion test. The final segment of the brackets is the code to execute at the end of the **if** {…} **loop**. So this specific example in English – first time you enter the loop you set **i** to 0. Every time round the loop you test whether your control variable **i** is less than or equal to 25, and at the end of every loop you increment **i** by 1. Because the control loop variable is altered here it is considered very bad practice (and heavily frowned upon) to alter this inside the body of the code. If you find you have to do that you've probably chosen the wrong loop control!

Where the code must be executed at least once the **do {….} while (test_condition)** is used.

There is also a **while (test_condition) do {….}** structure. In this structure it is possible that the code block is never executed if the test condition is immediately met. The do-while (and while-do) loops are used where you need to repeat the code a number of times and you don't at time of writing the code know how many times that is – for instance 'until the temperature reaches 47°C', which is something not controlled by the code itself. The for-loop is ideal when you know how many times you need to repeat the code.

Functions

The name used for callable routines in C and C++ is a **function**. A function is declared by a code sequence of the form:

```
int formSum(int a, int b, int c)
{
    int sum = 0;
    sum = a + b + c;
    return sum ;
}
```

This tells the compiler that the function returns an integer, and requires three input values when the function is called. The a, b and c labels are local to the function. The function is invoked by a statement of the form:

```
int myResult;
int z = 17;
myResult = formSum(z, 4, 7);  // myResult = 28
```

We often need to declare a form of template for the compiler to use when the function is not part of the current compilation unit. The structure of the function is placed in a header (.h) file with the same name as the compilation (.c or .cpp) file. For the example function above, the entry in the header file would be :

```
int formSum(int p, int q, int r);
```

When the compiler sees a call to formSum in another compilation module it knows that the function returns an integer and requires three integer inputs. If the programmer sends two integers and a floating point number, the compiler can alert the user to a mismatch of types.

Functions that return no value are declared as of type void:

```
void checkStatus(int a, float s)
{
    /* code to execute */
}
```

…and you can also have functions that don't require any inputs:

```
void changeState(void)
{
    /* code to execute */
}
```

Whether or not the function require any inputs, you still need to use the brackets when calling the function – the bracket is the signpost to the code that Here Cometh A Function. The last two example functions would be called like this:

```
checkStatus(anInteger,aFloat);
changeState();
```

Scope

The final topic to touch on is scope which basically means how 'visible' your code is.

When you are declaring your variables and your methods in the header-file, you have the option of two different sections where you can put the declarations, private or global. If it's in the global section, it is accessible by other pieces of code – if it's in the private section only your code can use it. ALL the code you will be writing yourself should use the private section.

A variable which you have declared in the header file will be accessible to all of your code, so one function can set the value and another function can read it (or indeed also set it to something else). If you declare a variable INSIDE a function however, the variable only exists inside this function so you can reuse variable names without them interfering with each other. It is here said to have local scope.