Style Guide for CSCI 111

PART 1

The following is a list of recommended C++ coding style guidelines. While style is often a matter of personal preference, consistency of style is very important. The purpose of using consistent style when coding is to make your code easier to read and understand by both you and a third party.
Using a good style will also help you to think in a more organized way while programming. Therefore, I suggest you adopt the style guidelines here to aid your learning and to avoid unnecessary confusion and ease the task of reading your code. Read through this document and if you have any questions discuss them with me or the course TA.

FILE HEADER:

Start every file with a header. Headers are essential in coding as they contain information about the purpose of the file, its author and when it was last modified. The header information, while extremely useful to the reader of the file, and will be ignored by the compiler as it is treated as a comment (see below).
For example:

```
/*****************************************************************

* Filename:

* Author:                    -    Date Last Modified:

* Assignment No:

* File Description:

* Academic integrity statement:

*
*****************************************************************/
```

```
/*****************************************
*   Assignment1.cpp
*   Sherine      - August 1 2016
*  Assignment 1
*  Calculates the larger of 5 given numbers
*  Academic integrity as per syllabus
*****************************************/
```

COMMENTING:

Commenting (documenting) your code is an integral part of good programming. The main aim of commenting code is to assist in the readability of the code. It should, however, be used moderately as too much commenting and unnecessary commenting can make reading the code more difficult.
The compiler will ignore all comments.
Comment should begin at the header to your files (see above) and also used to add a header to your functions (to be covered later in the course) and throughout your file to help explain complex code. There are two forms of commenting:

LINE COMMENTING:

// The two forward slashes indicate the beginning of line commenting. Everything
// from this point till the end of the line is deemed a comment and will be ignored //
by the compiler. Notice that each line begins with "//"

BLOCK COMMENTING:

/* Block commenting can cover many lines. Everything from this point on is treated as a comment till the closing asterisk-slash. It doesn't matter how many lines I use – it's all treated as one comment. This comment block finishes on the next line. */
NOTE: Avoid commenting variable declarations. This is generally unnecessary, as the selection of meaningful identifiers will satisfactorily explain their use.

WHITE SPACE:

Effective use of white space (spaces, tabs, blank lines) is vital in coding. Its purpose is to maximise readability of code. Without it, programs become very difficult to understand.

SPACES:

All keywords, identifiers, operators and operands should be separated by 1 space. Too much white space can have the opposite effect, so do not over do it.

Examples:

```
int myVariable = 10;
```

Note the space before and after the operator '='

```
myVariable = 10 + 5;
```

Note the space between the operator '+' and the operands '10', '5'

```
cout << " ←There are spaces here also → ?" << endl;
```

Note the space before and after the operator '<<'

BLANK LINES:

Keep all code relating to a specific task grouped together and separated from other code segments by a blank line.

For example:

```
// variable declarations
        int someVariable = 0;
         int anotherVariable = 10;
// get input
        cout << "Enter a value: ";
        cin >> someVariable;
// output
         cout << "someVariable = " << someVariable << endl;
        cout << "anotherVariable = " << anotherVariable << endl;
```

INDENTATION:

Indentation is vital to readability of code by facilitating the user's ability to follow the flow of the code.

Use tabs (set your tabs to 4 spaces) not spaces to indent code. (See note below).

A rule of thumb is that after an opening '{' you indent until the closing '}'.

For example, in the case of functions:
if statements, etc. (see example at the end of this document).
Indent your code AS YOU GO (Do not write your code and then indent after). This is to help YOU when writing your programs.

Be careful when using automatic indenting programs. Always check if the indentation is done properly. Note: never mix up spaces and tabs for indenting. Often when a program looks indented correctly in some editors it will not print out correctly for marking. This is because it contains a mixture of spaces and tabs. You may have tabs in your editor set to a different value to the tabs used for printing. This hinders readability.

An example of the correct use of white space is provided at the end of this document.

IDENTIFIERS:

An identifier is a name given to a memory location used to hold data during program execution. An identifier must be meaningful (name relates to its purpose) and reflect the purpose of the data held. It is much easier to write code using meaningful names, as you will easily remember what each identifier is for. Moreover, your programs will be easier to read and understand by a third party.

There are two types of identifiers: variables and constants.

VARIABLES:

Variables, as the name suggests, contain data, which can be varied during program execution. The style recommended in this course is to use all lower case letters for variable names. For example:

```
int count = 0; (See note below)
```

If your variables have multiple words then start each subsequent word with a capital letter. For example:

```
int currentRecord = 0;

int numberOfRecords = 0;
```

- Keep your identifiers as short as possible while avoiding abbreviation. If you choose to abbreviate, be careful that meaning does not become lost or ambiguous.

- When using boolean types it is important to choose identifiers, which are

positives. If not, you may end up using double negatives that are very confusing. For example:

```
bool valid = true; // is a good choice

bool invalid = false; // is a bad choice

  if ( valid == true ) // clear

if ( !(invalid == true) ) // not clear – means same as above
```

NOTE:

It is strongly recommended to assign your variables an initial value. This most often is '0' for numeric types or 'true' for boolean types, i.e. int count = 0;

CONSTANTS:

- Constants, unlike variables, contain values that remain constant throughout program execution, which once declared, cannot be altered.
- Constants should be UPPER_CASE with multiple words separated by an underscore. This makes constants easily distinguishable from variables. For example:

```
const int MAX_SIZE = 100;
```

- Constants should be declared at the beginning of your main function. This makes them easily locatable.

NOTE: Identifiers such as a, b, c, etc. are NOT acceptable as they are NOT meaningful

CODE EXAMPLE:

/* Filename: sample.cpp

Author: student - Date Modified: July 22, 2020

File Description: The following sample program shows the correct use of white space. The commenting is included to aid understanding only. I recommend you practice commenting in all your programs. */

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[])

{

      // Opening brace encountered above - indent all statements

      int myAge = 0, yourAge= 0;

      // note the space before and after the '=' operator and after the ','

      // notice a blank line is inserted above separating //code segments

      if (myAge == yourAge)

      { //Indent for new code block 1 tab from preceding '{ up to the closing '}'

            cout << myAge << " is equal " << yourAge << "!" << endl;
            cout << " we are the same age !" << endl;


       }//note close brace align to open brace

      //the following "if" statement doesn't contain braces as there is only 1 line of
      //code, however, indentation is still required.

      if (myAge > yourAge)

            cout << myAge << " is greater than " << yourAge << "!" << endl;


      return 0;

}//Closing brace aligns with Opening brace
```