**Coding Standards**

C++ Version:

Now, the C++ programming language should be at C++17, which means that older functions may be depreciated (code which has been replaced by newer, more efficient code).  
Do not use non-standard programming language extensions.

Header Files:

Every *.cpp* file should have a corresponding *.h* file, for example “*addNumbers.cpp*” and “*addNumbers.h*”. However, there are exceptions of this rule, for example with testing files or initialisation files.   
The use of header files is to improve the readability and size of the program, introducing modularity as many files could access the methods or attributes defined in the header file.

Figure : addNumbers.h

Figure : addNumbers.cpp

#include “addNumbers.h”

int main() {

a = 12;  
 b = 8;

sumOfNumbers(a, b);

}

int sumOfNumbers(int a, int b) {

return a + b;

}

Scoping:

Namespaces divide the scope of the program from global into smaller scopes, referenced with the namespace name before the attribute or method, like how structs are referenced in C++.   
The advantages of using namespaces are that using namespaces prevent the risk of name conflicts when writing large programs and introduce a method to navigate the program easier, because if you know which namespace a function is in, then it would be easy to navigate to the target namespace and look through a smaller section of code as opposed to all the code.  
However, the disadvantages of using namespaces are that using inline namespaces can be confusing to code with if you are a novice programmer as well as the risk of not knowing how to refer to an attribute or method inside more complicated namespace structures.  
When using namespaces, different types of variables are introduced into the program, these being:   
**Global Variables** – Variables which have a program-wide scope, which means that they can be accessed by any function in the program, which can be good for modularity, but can lead to the variable being modified where it should not be.  
**Local Variables** – Variables which have a less wide, more specific scope, usually inside a function or sub-procedure. This type of variable cannot be accessed by other functions in the program other than the on it has been declared inside.  
**Non-member Variables** – Variables which are defined outside of a class but inside a namespace. By declaring non-member functions this way, it stops the variable having a global scope.

One more type of object related to namespaces are static members, which is a type of function found inside namespaces. Static functions are accessed using the operator “::”.

namespace IntegerOperations {

class division {

float divideNumbers(a, b) {

return (a / b);

}

}

}

IntegerOperations::division Divide;  
DivideSum = Divide.divideNumbers(10, 2);  
std::cout << DivideSum << “\n”;

Figure : using namespace for a division calculation, which includes a static member function.

Classes:

Classes are structures which hold different types of related values, such as variables and methods.  
The type of related values that classes can hold are attributes and methods.  
Attributes are variables which define characteristics of the class, for example how much an item costs in a shop or the colour of a car.  
Methods are sub-procedures and functions which describe the actions of a class, for example how to update the stock of an item in a shop or how to make a car drive forward.

A similar structure used in C++ are called Structs, which are used identically but have different applications. Structs are used inside the STL library and should be used for stateless types instead of classes.

Classes are a part of a programming paradigm called Object-Oriented Programming (OOP), which are used to create programs closer to real-life objects. One example of a special feature of OOP is called Inheritance, which means that other classes can inherit the attributes and methods of a broader class, but also have other attributes and methods inside its own scope.

class ShopItem:

public:

std::string Name;  
float Cost;  
std::string ExpirationDate;  
void ChangeExpiryDate(std::string ShopItem.ExpirationDate, std::string NewDate) {

ShopItem.ExpirationDate = NewDate;

}

ShopItem Item;  
Item.Name = “”;  
Item.Cost = 1.99;  
Item.ExpirationDate = “01/03/2021”  
Item.ChangeExpiryDate(Item.ExpirationDate, “08/03/2021”);

Figure : Class to store the attributes and methods of an item in a shop.

Functions:

Functions are sections of code which can be called several times throughout a program and return a variable back to the function which called it, which shows that they are a great way to introduce modularity into a program.  
Functions can have variables and values passed into itself, these are called parameters. There are two types of parameters that can be passed into functions, these are:  
By Value – Data values which are copied and passed into the function. By using By Value parameters, the original value will not be changed after the function is returned from.  
By Reference – Data values have their address included, so when an operation is performed on the parameter, the original value will be changed when the function is returned from.

//By Value parameters in Function

Int SubtractNumbersByValue(int a, int b) {

Return (a-b);

}

//By Reference parameters in Function

Int SubtractNummbersByRef(int &a, int &b) {

&a -= 5;  
 return (&a - &b);

}

//Main Function  
int main() {

a = 10:  
 b = 6;

ByValueFunc = SubtractNumbersByValue(a, b);  
 ByRefFunc = SubtractNumbersByRef(\*a, \*b);

}

Figure : Functions using By Value and By Reference parameters.

Other C++ Features:

There are several more advanced features of C++ which can improve the quality of the program, these being:

Noexcept:

Noexcept is used to define whether a function will throw exceptions or not and crashes the program if a function marked “noexcept” throws an exception.  
An advantage of using noexcept is that it could lead to compiler optimisation as the compiler does not need to make further changes to handle exceptions if no exceptions will be generated from a function.  
However, a disadvantage of using noexcepts is that it may be difficult to disable noexcept due to the modifications it makes to the compiler.

void DisplayNumbers() noexcept;

Figure : Use of noexcept.

Streams:

Streams are used for input and output of variables, which is accessible inside a program using the “<iostream>” header.  
The “>>” and “<<” operators are used for assigning data to a variable and reading data back into a stream.  
The advantages of using streams in a program are that \_\_.  
The disadvantages of using streams in a program are that it is difficult to control stream output, which may lead to unwanted outputs during the execution of the program. Also, it is an advanced concept, so programmers would need experience of C++ to properly use this object.

#include <iostream>  
#include <sstream>

std::stringstream Number;  
Number << 100;  
int numOutput;

Number >> numOutput;

Std::cout << numOutput;

Figure : Use of stringstream to store values in variables.

Pre-increment and Pre-decrement:

Pre-incrementing and Pre-decrementing are features of the C++ language which are commonly used inside iterative loops to iterate through a data structure or through a range of numbers. The way pre-incrementing differs from post-incrementing is that the loop will change the value before the section of code inside the loop has been run.

For (int Index = 0; Index < 5; ++Index) {

Std::cout << Index << “\n”;

}

Figure : Use of Pre-incrementation and Pre-decrementation in a For Loop.

Post-increment and Post-decrement:

Pre-incrementing and Pre-decrementing are features of the C++ language which are commonly used inside iterative loops to iterate through a data structure or through a range of numbers. While using post-incrementing, the loop will change the value of a variable after the section of code inside the loop has been run.

For (int Index = 0; Index < 5; Index++) {

Std::cout << Index << “\n”;

}

Figure : Use of Post-incrementation and Post-decrementation in a For Loop.

Data Types:

There are many different data types which can be used while using C++, a few examples being:

String – A collection of characters and integers which represents words, examples being “England” and “John24”.  
Char – A single character, examples being ‘R’ and ‘X’.  
Integer – A number, you can have signed integers or unsigned integers, both have the ability to hold different amounts of values, examples being 34 or 100.  
Float – Represents a decimal value with 32 bit precision  
Double – Represents a decimal value, like the Float data type, but this differs by a Double data type being more precise.   
Boolean – A true or false value represented by a bit of data, either 1 for true or 0 for false.   
Constant – A variable which cannot be changed throughout the program. Constants are good for code quality, as if a change is needed to be made then a programmer would only need to make one change and the updates will apply to the entire program. An example of where a Const would be used is with defining the percentage at which tax is calculated, as if this were to change, then the constant variable holding the decimal value (for example 0.2 for 20%) of percentage can be changed and applied in one action.

sizeof function:

The sizeof function is an in-built function which returns how many items are inside the data structure passed in as a parameter. This function is widely used with array data structures, as vectors have a different in-built function which returns the size of the vector, called “.size”.

Int[5] array = {1,2,3,4,5};

int ArraySize = sizeof(array);

Figure : Use of the sizeof function.

Boost Library:

The Boost library is a collection of open-source libraries which can help improve the code quality of the program. One example of a library which is included in the Boost library is the Boost Test Library. This library is used to create unit tests to check the durability of the program and to flag if there are any errors in the code. The Boost Test Library is accessed by including “ #include <boost/test/unit\_test.hpp>” into the program.

#include “addNumbers.cpp”  
#define BOOST\_TEST\_MODULE Tests  
#include <boost/test/unit\_test.hpp>

BOOST\_AUTO\_TEST\_CASE(CorrectSum) {

BOOST\_CHECK\_EQUAL(sumOfNumbers(5,5), 10);

}

Figure : Boost Test Library example.

Naming:

It is advised that a constant naming convention is used throughout the program, which is even more important if the source code is being continuously added to by several programmers.  
For this project, you should use Camel Case naming convention.

int divideNumbers(int a, int b);  
bool completedOperation;  
float itemCost;

Figure : Examples of the naming rules.

Furthermore, with different object types used in a program, you should add different discernible features between the objects:

* When programming structs, always start with a capital letter.
* When programming constants, always include the letter ‘k’ at the beginning of the constant.
* When programming functions, always start the function name with a capital letter and every word in the name should start with a capital letter. This rule does not apply to the “main” function.
* When programming namespaces, always name the namespace in lower case letters.

Comments:

Use either the “//” syntax for single-line comments and the “/\* \*/” syntax for multiple-line comments while commenting the code, just ensure you are consistent.

At the beginning of every function, describe what the purpose of the function is and if the function returns any variable.

When commenting, try to avoid stating the obvious, for example, only saying that a for loop will loop through an array.  
Ensure that all punctuation, spelling, and grammar are correct throughout the code.

void NumberSquared(int x) {  
/\*Display the square of a number  
No Return \*/

x = x\*\*2;

}

Figure : Example of commenting a function.

Formatting:

When formatting code, use tabs instead of spaces to illustrate indents.  
If appropriate, use lambda functions for line efficiency, and keep the lambda function on one line.  
Each line of code should be 80 characters long maximum, as this would make each line easier to read.

When using conditionals, ensure that there are no spaces inside the brackets.  
When using curly brackets, keep the brackets on the same line as the condition, loop, function etc.   
When using functions and namespaces, do not leave any blank spaces as the first line of the function.

Exception Rules:

You may only diverge from these rules if you are dealing with pre-existing code which does not conform to these styling rules.