Coding Style Guide

|  |  |
| --- | --- |
| **Convention** | **Description** |
| C++ version & IDE | For this project, it is crucial for the team to use QT creator in order to complete the project. QT Creator is an open source C++ IDE which specialises in creating GUI’s. For this project we will be using version **5.13.2.** We will be using C++11 for the completion of this project. This is the most recent release for the language and it compatible with all the Qt’s features. |
| Header Files | For this project, every header file should be stored within the SRC directory in the repo, the IDE will then separate the header and source files once the project is loaded. All .cpp files must have an associated header files stored within the directory, an exception can be made for the main.cpp file, however this file will not be only be used for building the project and creating the MVC format.  Header files should only contain the necessary libraries for defining the relevant methods and properties of said class. All functional libraries should be included in the associated source file.  Header files should be named identically to the source file they are representing, except with a different extension (.h). When creating a header file for a class, the class and header files should be named in the following way:  **<classname>class.extension**  This way, it is evident which header files represent the OO aspects of the project and any other associated files are kept separately.  Header files should be all lower case and not be camel cased, this is Qt’s standard which we will be upholding. |
| Scoping Conventions | Place a function's variables in the narrowest scope possible and initialize variables in the declaration. It is encouraged to define all key variables at the top of a function and perform all initialisation at definition where possible. Global variables should remain constant and not be able to be changed in various parts of the program. For classes, there should be a getter and setter for any modifiable values. If a global variable is required, it should be in the form of a pointer by reference which negates the possibility of the value being changed unless explicitly specified. |
| Classes / Structs | Class’s should have a header files named identically to the source files which honours the same naming convention as stated in the header file section. Class name definitions should be Capitalised and each subsequent word in the class if needed should also be capitalised, e.g. **class ItemClass**. Each class has to have a constructor and should have a destructor where relevant.  Structs should follow the same naming convention as classes having the first letter of each word being capitalised. Structs do not require a header file and can be defined in any section of the project. Structs can have a constructor method, however C++ lets the programmer define a new instance of the struct with the parameters being the properties in the same order as defined. |
| Functions | Each function in the project should complete one main goal and be focused on solely performing that task. This means that no unnecessary parameters should be included in order to save memory space. Function names should be in CamelCase and should prioritise returning a value over modifying an existing variable. Where references to an external object is required, a pointer should be used. Default values for parameters are allowed on non-vital functions when the default is guaranteed to always gave the same value. On higher priority functions, this should be avoided in order to maintain data consistency. When writing names for paramters, if the data inputted is being stored as a property inside of the function, use an \_ to distinguish that the value is simply being copied. For example: void Annotation::**setCaption**(string \_caption){ this->caption = \_caption; } |
| Naming | Variable names should be clear and represent exactly what the values stored means. Single letter variables such as ‘x’ or ‘i’ should be avoided expect for in for loops where they can be used to represent an iterating value. Variable names should aim to be concise and not trail on in order to make the code look cleaner. |
| Comment Styling | Comments should be written using ‘//’ for single like comments and ‘/\* \*/’ for multi-line comments. For each function, a comment should be added next to the definition to describe what the function is trying to achieve where the name of the function does not make this explicitly obvious. For classes, a comment describing the class and how it should be used should be present in the header file. This description should include examples of how the class would be used and what the purpose of this class is. This includes describing what the data stored is used for. When defining a constructor for a class, a comment should follow to definition in order to show the order of parameters and which values are associated with which property of the class. In the case where multiple constructors are used, the comment should describe how the remaining data is obtained for the class. Variable name should be descriptive enough to not require a comment to explain what is stored and how the variable is used. |
| Code Formatting | Code formatting must remain consistent in order to help other group members or anyone refactoring the code. When indenting a line of code, 1 tab should be used. This is especially important for code within loops, all code within loops should be indented from the loop definition to show which section is being iterated over. |