**CODING STYLE**

SOFTWARE DESIGN & IMPLEMENTATION GROUP 9: RISHI SINGH (PM), MALEESHA DEDIGAMA ARACHCHIGE, DAYEETA DAS & JOSHUA SSENDAGIRE

**Contents**

Background ---------------------------------------------------------------------------------------------- 4

Goals of Style Guide ------------------------------------------------------------------------------------ 4

C++ Version ---------------------------------------------------------------------------------------------- 4

Header Files ---------------------------------------------------------------------------------------------- 5-8

* Self-Contained Headers ----------------------------------------------------------------- 5
* The #define Guard --------------------------------------------------------------------------- 5
* Forward Declarations --------------------------------------------------------------------------- 5-7
* Inline Functions --------------------------------------------------------------------------- 7
* Names and Order of Includes ----------------------------------------------------------------- 7-8

Scoping ------------------------------------------------------------------------------------------ 8-12

* Namespaces ------------------------------------------------------------------------------------ 8
* Unnamed Namespaces & Static Variables --------------------------------------------- 8-9
* Non-member, Static and Global Functions --------------------------------------------- 9
* Local Variables ------------------------------------------------------------------------ 9-10
* Static & Global Variables ------------------------------------------------------------ 10-11
* Decision on Destructors -------------------------------------------------------------- 11
* Decision on Initialization ---------------------------------------------------------------- 11
* thread\_local variables ------------------------------------------------------------------------ 11-12

Classes ------------------------------------------------------------------------------------------- 12-16

* Doing Work in Constructors ------------------------------------------------------------- 12-13
* Implicit Conversions ----------------------------------------------------------------------- 13-14
* Inheritance --------------------------------------------------------------------------------- 14-15
* Access Control --------------------------------------------------------------------------------- 15-16
* Declaration Order ----------------------------------------------------------------------- 16

Functions ------------------------------------------------------------------------------------------- 16-17

* Output Parameters ------------------------------------------------------------------------ 17
* Write Short Functions ------------------------------------------------------------------------ 17

Exceptions -------------------------------------------------------------------------------------------- 17-18

Naming and Naming Style ------------------------------------------------------------------------ 18-20

* General Naming Rules -------------------------------------------------------------- 18
* File Names ---------------------------------------------------------------------------------- 18
* Type Names ---------------------------------------------------------------------------------- 19
* Variable Names ------------------------------------------------------------------------ 19
* Constant Names ------------------------------------------------------------------------ 19
* Function Names
* Namespace Names ------------------------------------------------------------------------- 19-20

Comments Style ----------------------------------------------------------------------------------- 20-21

* Comment Style Adopted for DOXYGEN -------------------------------------------- 20-21
* Comments used before declarations -------------------------------------------- 20
* Descriptive Comments ------------------------------------------------------ 20
* Short Comments ---------------------------------------------------------------- 21

Formatting --------------------------------------------------------------------------------------------- 21-22

* General ------------------------------------------------------------------------------------ 21
* Functions ------------------------------------------------------------------------------------ 21
* Conditionals and Exceptions ---------------------------------------------------------------- 22
* Comments ------------------------------------------------------------------------------------ 22

**Background**

The programming language that we have chosen to use for our project is C++. C++ is a robust, versatile, object oriented programming language. It presents a number of dynamic libraries to choose from that makes the code reliable, reusable and most importantly manageable.

The main goal of this document is to present to the user a description of the constraints used by the programmers for coding the project using C++. These constraints are meant to make the code base manageable. However, the coder is free to use the productive features of the language.

It has been assumed that the reader is familiar with the programming language used here, that is C++.

**Goals of the Style Guide**

There are a few goals that we want to achieve through this guide. These guide states all the fundamental rules that one must follow while contributing to the particular project and why we have opted these styles. The goals that we have strived to achieve through this guide are as stated below:

* The style guide is large enough to justify asking of all engineers to remember it.
* This coding style is mainly optimized for the reader than the writer. For a person who wants to contribute towards this project, the style guide should make it transparent for him/her to understand the format in which the code has been written and will enable him/her to make sense from the existing code.
* The style guide enables to maintain consistency throughout the code. This will in turn make debugging easier and will also allow automation. Maintaining the same style throughout will enable the developers to get back to working on a particular part of the project easily specially in case of sprints.
* The guide encourages one to avoid surprising or dangerous constructs. There is a high bar for style guide waivers on such restrictions, because waiving such rules often directly risks compromising program correctness.
* The guide encourages one to avoid constructs that average C++ programmers will find hard to maintain.
* The guide enables one to concede to optimization when necessary. Performance optimizations can sometimes be necessary and appropriate, even when they conflict with the other principles of this document.

**C++ Version**

The code should target the current version of C++ which is C++17 at the moment. It should be noted that the targeted version by the guide might advance over time.

**Header Files**

For each .cpp file, there should be a .h file except for the unit tests and for small .cpp files that contains just a main() function. Correct use of header files can enhance the readability, size and overall performance of the code.

**Self-Contained Headers**

Header files should be able to compile on their own and end in .h. A header file should have header guards and all the other headers it needs so that the users or refactoring tools won’t require special conditions to include the header.

The definitions for the templates and functions should be placed in the same file as their declarations. Their definitions should be included into every .cpp file that uses them else the program might fail to link in some build configurations .

**The #define Guard**

All header files should have #define guards to prevent multiple inclusion. The format of the symbol name should be <PROJECT>\_<PATH>\_<FILE>\_H\_. To guarantee uniqueness, they should be based on the full path in a project's source tree.

Example from code snippet:

A screenshot of a cell phone

Description automatically generated

**Forward Declarations**

Try to reduce the use of forward declarations and try using #include to include the headers wherever possible.

Example from code snippet:

A screenshot of text

Description automatically generated

**Definition:**

In computer programming a forward declaration is a declaration of an identifier for which the programmer hasn’t yet given a complete definition.

**Advantages:**

* Forward declarations saves compile time. The complier has to open more files and process more inputs when using header files.
* Forward declarations prevent unnecessary compilations. Header files may force the code to get compiled more than necessary in case any changes are made to them.

**Disadvantages:**

* Forward declarations might prevent the users from making necessary compilations when changes are made in the headers.
* Forward declarations can prevent the header owners from making compatible changes to their APIs.
* Forward declaring symbols from namespace std:: yields undefined behaviour.
* Sometimes replacing a #include with the forward declaration can change the meaning of the piece of code.
* In case multiple symbols need forward declaration, it is best to switch that with an #include header file.
* Structuring code to enable forward declarations can make the code slower and more complex.

**Inline Functions**

Only small functions are defined inline.

**Definition:**

The inline functions are a C++ enhancement feature to increase the execution time of a program. Functions can be instructed to compiler to make them inline so that compiler can replace those function definition wherever those are being called.

**Advantages:**

Inline functions can generate more efficient object code.

**Disadvantages:**

Overuse of inlining can make the program to run slower. Inlining small functions will decrease the code size but inlining a large function will increase the code size. This in turn will make a modern processor to run slower than usual due to the instruction cache.

**Names and Order of Includes**

The headers should be included in the following order:

* Related header
* C++ standard library headers
* Other libraries’ headers ( if necessary)
* The project’s headers

All of a project's header files should be listed as descendants of the project's source directory without use of UNIX directory aliases . (the current directory) or .. (the parent directory).

For each section, the includes must be ordered alphabetically.

Include the headers that define the symbols that are used in the program except in the unusual cases of forward declaration.

Sometimes, system-specific code needs conditional includes. Such code can put conditional includes after other includes. The system-specific code should be small and localized.

Example from code snippet:

A screenshot of a cell phone

Description automatically generated

**Scoping**

**Namespaces**

Considering a few exceptions, try to place the code in a namespace. Try to give a unique name to the namespace based on the name of the project and its path. Avoid using directives or inline namespaces.

**Definition:**

Namespaces subdivide the global scope into distinct, named scopes, and so are useful for preventing name collisions in the global scope.

**Advantages:**

* Namespaces prevent names conflict in larger programs and allow the code to use reasonable short names.
* Inline namespaces automatically place their names in the enclosing scope.

**Disadvantages:**

* Namespaces can be confusing, because they complicate the mechanics of figuring out what definition a name refers to.
* Inline namespaces, in particular, can be confusing because names aren't actually restricted to the namespace where they are declared. They are only useful as part of some larger versioning policy.
* In some contexts, it's necessary to repeatedly refer to symbols by their fully-qualified names. For deeply-nested namespaces, this can add a lot of clutter.

**Unnamed Namespaces and Static Variables**

If definitions in a C++ file don’t need external referencing then place them in an unnamed namespace or declare them static and do not use them in the header files.

**Definition:**

Static is a keyword in C++ used to give special characteristics to an element. Static elements are allocated storage only once in a program lifetime in static storage area. And they have a scope till the program lifetime. Static Keyword can be used with following, Static variable in functions.

**Nonmember, Static Member, and Global Functions**

Place non member functions in a namespace and avoid using global functions. A class should not be used to simply group static functions. Static methods of a class should generally be closely related to instances of the class or the class's static data.

**Advantages:**

Nonmember and static member functions can be useful in some situations. Putting non member functions in a namespace avoids polluting the global namespace.

**Disadvantages:**

Nonmember and static member functions may make more sense as members of a new class, especially if they access external resources or have significant dependencies.

**Local Variables**

The variables used in a function should be placed in the narrowest scope and they should be initialized with declarations.

C++ enables the programmer to declare the variables anywhere. However, for convenience of the readers, it should be declared closer to where it has been used and within a local scope.

Variables needed for the branch statements and the loops should be declared within them so that they are confined to their scopes.

If the variable is an object, its constructor is invoked every time it enters scope and is created, and its destructor is invoked every time it goes out of scope.

Example from code snippet:

A screenshot of a social media post

Description automatically generated

**Static and Global Variables**

Static variables are only declared when they are absolutely necessary for the program.

**Definition:**

Static is a keyword in C++ used to give special characteristics to an element. Static elements are allocated storage only once in a program lifetime in static storage area. And they have a scope till the program lifetime. Static Keyword can be used with following.

* Static variable in functions
* Static Class Objects
* Static member Variable in class
* Static Methods in class

Static variables when used inside function are initialized only once, and then they hold their value even through function calls. These static variables are stored on static storage area , not in stack.

Static keyword works in the same way for class objects too. Objects declared static are allocated storage in static storage area, and have scope till the end of program.

Static objects are also initialized using constructors like other normal objects. Assignment to zero, on using static keyword is only for primitive datatypes, not for user defined datatypes.

Static data members of class are those members which are shared by all the objects. Static data member has a single piece of storage, and is not available as separate copy with each object, like other non-static data members.

Static member variables (data members) are not initialized using constructor, because these are not dependent on object initialization.

Also, it must be initialized explicitly, always outside the class. If not initialized, Linker will give error.

Static member functions can also be declared which work for a class as whole rather for a particular object of the class.

**Advantages:**

Global and static variables are very useful for a large number of applications: named constants, auxiliary data structures internal to some translation unit, command-line flags, logging, registration mechanisms, background infrastructure, etc.

**Disadvantages:**

Global and static variables having dynamic initialization or having non-trivial destructors create complexity that makes it hard to find bugs.

**Decision on Destructors**

When destructors are trivial, their execution is not subject to ordering at all otherwise we are exposed to the risk of accessing objects after the end of their lifetime. Therefore, we only allow objects with static storage duration if they are trivially destructible. Fundamental types (like pointers and int) are trivially destructible, as are arrays of trivially destructible types.

**Decision on initialization**

Constant initialization is always allowed. Any non-local static storage duration variable that is not so marked should be presumed to have dynamic initialization, and reviewed very carefully.

Dynamic initialization of nonlocal variables is discouraged, and in general it is forbidden. However, we do permit it if no aspect of the program depends on the sequencing of this initialization with respect to all other initializations. Under those restrictions, the ordering of the initialization does not make an observable difference.

**thread\_local Variables**

thread**\_**local variables that don’t have a declaration inside a function should be initialized with a true compile-time constant.

**Definition:**

Thread-local storage can be created using the thread\_local keyword. A variable declared with the thread\_local specifier is said to have thread storage duration**.**

* Each thread in a program has its own copy of each thread-local variable.
* A thread-local variable with function (local) scope will be initialized the first time control passes through its definition. Such a variable is implicitly static, unless declared extern.
* A thread-local variable with namespace or class (non-local) scope will be initialized as part of thread start up.
* Thread-local variables are destroyed upon thread termination.
* A member of a class can only be thread-local if it is static. There will therefore be one copy of that variable per thread, rather than one copy per (thread, instance) pair.

**Advantages:**

* Only one thread can access a thread-local data. This makes it useful for concurrent programming.
* thread\_local is the only formal way of creating a thread-local data.

**Disadvantages:**

* Accessing a thread\_local variable may trigger execution of an unpredictable and uncontrollable amount of other code.
* thread\_local variables are effectively global variables, and have all the drawbacks of global variables other than lack of thread-safety.
* The memory consumed by a thread\_local variable scales with the number of running threads (in the worst case), which can be quite large in a program.
* An ordinary class member cannot be thread\_local.
* thread\_local may not be as efficient as certain compiler essentials.

**Classes**

Classes are the basic unit of code in C++ and are used widely throughout the project. This section lists the main constraints one must follow while working with the classes.

**Doing Work in Constructors**

Virtual method calls should be avoided in a constructor and any such initialization must be avoided that may cause a signal error.

**Definition:**

Arbitrary initialization can be performed in the body of the constructor.

**Advantages:**

* One does not have to be worried even if the class has not been initialized.
* Objects that are fully initialized by constructor call can be const and may also be easier to use with standard containers or algorithms.

**Disadvantages:**

* If the work calls virtual functions, these calls will not get dispatched to the subclass implementations. Future modification to your class can quietly introduce this problem even if your class is not currently subclassed, causing much confusion.
* There is no easy way for constructors to signal errors, short of crashing the program (not always appropriate) or using exceptions (which are forbidden).
* If the work fails, we now have an object whose initialization code failed, so it may be an unusual state requiring a bool IsValid() state checking mechanism (or similar) which is easy to forget to call.
* You cannot take the address of a constructor, so whatever work is done in the constructor cannot easily be handed off to, for example, another thread.

Example from code snippet:

A screenshot of a cell phone

Description automatically generated

**Decision:**

Constructors should never call virtual functions. If appropriate for your code , terminating the program may be an appropriate error handling response. Otherwise, consider a factory function or Init() method. Avoid Init() methods on objects with no other states that affect which public methods may be called (semi-constructed objects of this form are particularly hard to work with correctly).

**Implicit Conversions**

Do not define implicit conversions. Use the explicit keyword for conversion operators and single-argument constructors.

**Definition:**

Implicit conversions allow an object of one type (called the source type) to be used where a different type (called the destination type) is expected, such as when passing an int argument to a function that takes a double parameter.

In addition to the implicit conversions defined by the language, users can define their own, by adding appropriate members to the class definition of the source or destination type. An implicit conversion in the source type is defined by a type conversion operator named after the destination type (e.g. operator bool()). An implicit conversion in the destination type is defined by a constructor that can take the source type as its only argument (or only argument with no default value).

The explicit keyword can be applied to a constructor or (since C++11) a conversion operator, to ensure that it can only be used when the destination type is explicit at the point of use, e.g. with a cast.

**Advantages:**

* Implicit conversions can make a type more usable and expressive by eliminating the need to explicitly name a type when it's obvious.
* Implicit conversions can be a simpler alternative to overloading, such as when a single function with a string\_view parameter takes the place of separate overloads for std::string and const char\*.
* List initialization syntax is a concise and expressive way of initializing objects.

**Disadvantages:**

* Implicit conversions can hide type-mismatch bugs, where the destination type does not match the user's expectation, or the user is unaware that any conversion will take place.
* Implicit conversions can make code harder to read, particularly in the presence of overloading, by making it less obvious what code is actually getting called.
* Constructors that take a single argument may accidentally be usable as implicit type conversions, even if they are not intended to do so.
* When a single-argument constructor is not marked explicit, there's no reliable way to tell whether it's intended to define an implicit conversion, or the author simply forgot to mark it.
* It's not always clear which type should provide the conversion, and if they both do, the code becomes ambiguous.
* List initialization can suffer from the same problems if the destination type is implicit, particularly if the list has only a single element.

**Decision:**

Type conversion operators, and constructors that are callable with a single argument, must be marked explicit in the class definition. As an exception, copy and move constructors should not be explicit, since they do not perform type conversion. Implicit conversions can sometimes be necessary and appropriate for types that are designed to transparently wrap other types. In that case, contact your project leads to request a waiver of this rule.

Constructors that cannot be called with a single argument may omit explicit. Constructors that take a single std::initializer\_list parameter should also omit explicit, in order to support copy-initialization.

**Inheritance**

When using inheritance, make it public.

**Definition:**

When a sub-class inherits from a base class, it includes the definitions of all the data and operations that the base class defines. "Interface inheritance" is inheritance from a pure abstract base class (one with no state or defined methods); all other inheritance is "implementation inheritance".

**Advantages:**

* Implementation inheritance reduces code size by re-using the base class code as it specializes an existing type.
* Inheritance is a compile-time declaration, you and the compiler can understand the operation and detect errors.

**Disadvantages:**

* For implementation inheritance, because the code implementing a sub-class is spread between the base and the sub-class, it can be more difficult to understand an implementation.
* The sub-class cannot override functions that are not virtual, so the sub-class cannot change implementation.
* Multiple inheritance is especially problematic, because it often imposes a higher performance overhead.

**Decision:**

All inheritance should be public. If you want to do private inheritance, you should be including an instance of the base class as a member instead.

Do not overuse implementation inheritance. Composition is often more appropriate. Try to restrict use of inheritance to the "is-a" case.

Limit the use of protected to those member functions that might need to be accessed from subclasses.

Explicitly annotate overrides of virtual functions or virtual destructors with exactly one of an override or (less frequently) final specifier. Do not use virtual when declaring an override. Rationale: A function or destructor marked override or final that is not an override of a base class virtual function will not compile, and this helps catch common errors. The specifiers serve as documentation; if no specifier is present, the reader has to check all ancestors of the class in question to determine if the function or destructor is virtual or not.

Multiple inheritance is permitted, but multiple implementation inheritance is strongly discouraged.

**Access Control**

Member variables that won’t be used by any other classes should be declared as private. So, is the case with the slots and functions that restrict to only one particular class.

In case of member variables and functions that needs to be called by other classes should be declared as public to allow them to be accessed or called easily whenever required.

Example from code snippet:

A screenshot of a social media post

Description automatically generated

**Declaration Order**

Similar declarations should be grouped under the same access specifiers. The class definition should usually start with the declaration of the public members followed by the private or protected members. Large method definitions should not be put as inline in the class definitions.

**Functions**

Functions are used throughout the code in order to represent and implement a modular and object oriented approach. Different types of functions are used based on the requirements of the project.

The coder must follow certain rules while using functions in the code:

1. The function name should match their purpose.
2. Functions should be declared in the headers before being used in the source file.
3. It is better to return values rather than using output parameters.
4. Functions should not be too lengthy.
5. Try to keep the functions intact to only a particular functionality.
6. Try to use different functions for implementing functionality of different parts of the UI.
7. In case one function has to implement several operations for a particular task, use function overloading.

Using small functions helps in easy debugging of the program.

Example from code snippet:

A screenshot of a cell phone

Description automatically generated

**Output Parameters**

The output of a C++ function is naturally provided via a return value and sometimes via output parameters.

Prefer using return values over output parameters: they improve readability, and often provide the same or better performance.

Parameters are either input to the function, output from the function, or both.

**Write Short Functions**

Prefer small and focused functions. We recognize that long functions are sometimes appropriate, so no hard limit is placed on functions length. Even if your long function works perfectly now, someone modifying it in a few months may add new behaviour. This could result in bugs that are hard to find. Keeping your functions short and simple makes it easier for other people to read and modify your code. Small functions are also easier to test.

You could find long and complicated functions when working with some code. Do not be intimidated by modifying existing code: if working with such a function proves to be difficult, you find that errors are hard to debug, or you want to use a piece of it in several different contexts, consider breaking up the function into smaller and more manageable pieces.

**Exceptions**

We have used try-catch block to implement exception handling in our code. We have also used if statements in order to put checks on certain aspect of our project.

The try-catch block is used to handle any generic exceptions arising in a particular slot in any class.

The if statements are used to put checks on specific aspect of that particular slot. This means that the if statement could put a check within a try catch block itself.

**Advantages:**

* Exceptions allow higher levels of an application to decide how to handle "can't happen" failures in deeply nested functions, without the obscuring and error-prone bookkeeping of error codes.
* Exceptions are used by most other modern languages. Using them in C++ would make it more consistent.
* Exceptions are really handy in testing frameworks.

**Disadvantages:**

* When you add a throw statement to an existing function, you must examine all of its transitive callers. Either they must make at least the basic exception safety guarantee, or they must never catch the exception and be happy with the program terminating as a result.
* More generally, exceptions make the control flow of programs difficult to evaluate by looking at code: functions may return in places you don't expect. This causes maintainability and debugging difficulties.
* Turning on exceptions adds data to each binary produced, increasing compile time (probably slightly) and possibly increasing address space pressure.
* The availability of exceptions may encourage developers to throw them when they are not appropriate or recover from them when it's not safe to do so.

**Decision:**

On their face, the benefits of using exceptions outweigh the costs, especially in new projects. However, for existing code, the introduction of exceptions has implications on all dependent code. If exceptions can be propagated beyond a new project, it also becomes problematic to integrate the new project into existing exception-free code.

**Naming and Naming Style**

**General Naming Rules**

The names used for the objects in the code should be relevant to their purpose. This enables the coder as well as a contributor to easily identify an object based on their need. IN the long run when the code eventually gets more complex and lengthier, it helps in easy identification of the objects that are required to make any necessary changes. Naming should be given priority by all the contributors and coders. The name should reflect the purpose of an object used.

**File Names**

Filenames should be all lowercase and can include underscores (\_) or dashes (-). Follow the convention that your project uses. If there is no consistent local pattern to follow, prefer "\_".

**Type Names**

Type names start with a capital letter and have a capital letter for each new word.

The names of all types — classes, structs, type aliases, and type template parameters — have the same naming convention. Type names should start with a capital letter and have a capital letter for each new word.

**Variable Names**

The names of variables (including function parameters) and data members are all lowercase, with or without underscores between words. Data members of classes (but not structs) additionally have trailing underscores.

**Constant Names**

Variables declared constexpr or const, and whose value is fixed for the duration of the program, are named with mixed case. Underscores can be used as separators in the rare cases where capitalization cannot be used for separation. All such variables with static storage duration. This convention is optional for variables of other storage classes, e.g. automatic variables, otherwise the usual variable naming rules apply.

**Function Names**

Regular functions have mixed case; accessors and mutators may be named like variables. Ordinarily, functions should start with a capital letter and have a capital letter for each new word with or without an underscore between them.

Accessors and mutators (get and set functions) may be named like variables. These often correspond to actual member variables, but this is not required.

Example from code snippet:

A screenshot of a cell phone

Description automatically generated

**Namespace Names**

Namespace names are all lower-case. Top-level namespace names are based on the project name . Avoid collisions between nested namespaces and well-known top-level namespaces.

The name of a top-level namespace should usually be the name of the project or team whose code is contained in that namespace. The code in that namespace should usually be in a directory whose base name matches the namespace name (or in subdirectories thereof).

**Comments Style**

Just like naming, comments are also very essential towards making our code look more readable. Comments especially help a person who is looking at our code for the first time or one who will be contributing to our project.

/\* \*/ This is used in the code when our comment exceeds multiple lines or for describing the purpose of a particular piece of code. While // this is used when we are commenting out a small operation or pointing out an identifier.

**Comments Style adopted for DOXYGEN**

**Comments Used Before Declarations:**

/// statements are used for commenting before class or function declarations in the header or source files throughout the project.

Example from code snippet:

A screenshot of a social media post

Description automatically generated

**Descriptive Comments:**

/\*\* … \*/ statements are used for describing the purpose of a function or a class or for comments that give a detailed description of a particular snippet of code and might range upto multiple lines.

A screenshot of a cell phone

Description automatically generated

**Short Comments:**

/// … are used for stating a brief description of a particular statement within the program.

A picture containing bird

Description automatically generated

**Formatting**

**General**

* Each line in the code should not be too long.
* Whitespaces must be used wherever necessary in order to make the code neater and easily readable.
* Whitespaces should be used before an operator and after an operator to prevent a statement from looking clumsy.

**Functions**

* Choose good parameter names.
* A parameter name may be omitted only if the parameter is not used in the function's definition.
* If you cannot fit the return type and the function name on a single line, break between them.
* If you break after the return type of a function declaration or definition, do not indent.
* The open parenthesis is always on the same line as the function name.
* There is never a space between the function name and the open parenthesis.
* There is never a space between the parentheses and the parameters.
* The open curly brace is always on the end of the last line of the function declaration, not the start of the next line.
* The close curly brace is either on the last line by itself or on the same line as the open curly brace.
* There should be a space between the close parenthesis and the open curly brace.
* All parameters should be aligned if possible.
* Default indentation is 2 spaces.
* Wrapped parameters have a 4 space indent.

**Conditionals and Exceptions**

* In case of conditionals like if-else or switch-case, the code must be written within curly braces in case it extends upto multiple lines. This makes the code cleaner and easier to look at for a new reader.
* In case of try-catch or exceptions, the code must be written within curly braces because it represents functionality over a block of code. Even if the block is just a line, the usage of curly braces is encouraged for maintaining consistency.

**Comments**

* Declarative comments /\* \*/ should be placed before the start of a piece of code.
* Identifying comments // should be placed beside the particular line of code.
* This helps to maintain the readability of the code.
* There should be space between a literal and an operator.

Though these rules of formatting might be hampered in the test file. It is best to try abiding these rules to maintain a consistency in the format of the entire piece of code. This enhances the readability of the code.

**Parting words**

I would encourage anyone who is contributing to this project to take sometime to go through this Coding Style guide or examine the code around the files in the projects and maintain consistency while making changes to the code. In this way, the readability of the code will not be hampered. The discontinuity in the format of the code can cause distraction to other contributors or the developers when they are making changes to the code.