Coding Style C++

Performances

Overall, use the most efficient algorithms.

- Avoid losing performance over laziness.
- Be careful about function call duplication.
- Always think about the memory vs performance ratio.

Always avoid code duplication. Not only does it affect performance a bit but most importantly it makes the code ugly and harder to debug.

Line Length

Each line of text in your code should be at most 80 characters long.

Write Short Functions

Prefer small and focused functions.

Long functions are **sometimes** appropriate, so no hard limit is placed on functions length. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

Brackets

With functions:

If they are in a .cpp file :

```
void Obj::foo()
{
  // code
}
```

If they are in a .hpp file :

```
void foo(){
  // code
}
```

For if / loops and switch statements:

```
if () {
  // code
}
while() {
  // code
}
switch () {
  // code
}
```

Static variables and functions

There should not be any static variable inside methods (unless rare exceptions). Instead, they should be added as attributes to the class.

Static functions should only be used in singletons.

Code division

Implementation must be separated from declaration.

The only exception is with the implementation of `templates` in which case the implementation and the declaration may both be in an `.hpp` file.

Header files

In general, every `.cpp` file should have an associated `.hpp` file. There are some common exceptions, such as unit tests and small `.cpp` files containing just a `main()` function.

Correct use of header files can make a huge difference to the readability, size and performance of your code.

All header files should have `#pragma once` guard to prevent multiple inclusion.

Forward Declarations

Avoid using forward declarations where possible. Just `#include` the headers you need.

A "forward declaration" is a declaration of a class, function, or template without an associated definition.

- Forward declarations can save compile time, as `#include`s force the compiler to open more files and process more input.
- Forward declarations can save on unnecessary recompilation. `#include`s can force your code to be recompiled more often, due to unrelated changes in the header.

- Forward declarations can hide a dependency, allowing user code to skip necessary recompilation when headers change.
- A forward declaration may be broken by subsequent changes to the library. Forward declarations of functions and templates can prevent the header owners from making otherwise-compatible changes to their APIs, such as widening a parameter type, adding a template parameter with a default value, or migrating to a new namespace.
- Forward declaring symbols from namespace `std::` yields undefined behavior.
- It can be difficult to determine whether a forward declaration or a full `#include` is needed. Replacing an `#include` with a forward declaration can silently change the meaning of code.

Namespaces

With few exceptions, place code in a namespace. Namespaces should have unique names based on the project name, and possibly its path. Do not use *using-directives* (e.g. `using namespace foo`). Do not use inline namespaces.

```
namespace outer {
  void foo();
  class Rani;
}  // namespace outer

outer::Rani {
   Rani();
  ~Rani();
}
```

You may not use a *using-directive* to make all names from a namespace available.

```
// Forbidden -- This pollutes the namespace.
using namespace foo;
```

Structs vs. Classes

Use a `struct` only for passive objects that carry data; everything else is a `class`.

The `struct` and `class` keywords behave almost identically in C++.

`structs` should be used for passive objects that carry data, and may have associated constants, but lack any functionality other than access/setting the data members. The accessing/setting of fields is done by directly accessing the fields rather than through method invocations. Methods should not provide behavior but should only be used to set up the data members, e.g., constructor, destructor, `Initialize()`, `Reset()`, `Validate()`.

If more functionality is required, a `class` is more appropriate. If in doubt, make it a `class`.

Reference Arguments

All parameters passed by lvalue reference must be labeled `const`.

In C, if a function needs to modify a variable, the parameter must use a pointer, eg `int foo(int *pval)`. In C++, the function can alternatively declare a reference parameter: `int foo(int &val)`.

Defining a parameter as reference avoids ugly code like `(*pval)++`. Necessary for some applications like copy constructors. Makes it clear, unlike with pointers, that a null pointer is not a possible value.

Within function parameter lists all references must be `const`:

```
void Foo(const string &in, string *out);
```

However, there are some instances where using `const T*` is preferable to `const T&` for input parameters. For example:

- You want to pass in a null pointer.
- The function saves a pointer or reference to the input.

Remember that most of the time input parameters are going to be specified as `const T&`. Using `const T*` instead communicates to the reader that the input is somehow treated differently. So, if you choose `const T*` rather than `const T&`, do so for a concrete reason; otherwise it will likely confuse readers by making them look for an explanation that doesn't exist.

Casting

Use C++-style casts like `static_cast<float>(double_value)`, or brace initialization for conversion of arithmetic types like `int64 y = int64{1} << 42`. Do not use cast formats like `int y = (int)x` or `int y = int(x)` (but the latter is okay when invoking a constructor of a class type).

C++ introduced a different cast system from C that distinguishes the types of cast operations.

The problem with C casts is the ambiguity of the operation; sometimes you are doing a *conversion* (e.g., `(int)3.5`) and sometimes you are doing a *cast* (e.g., `(int)"hello"`). Brace initialization and C++ casts can often help avoid this ambiguity. Additionally, C++ casts are more visible when searching for them.

The C++-style cast syntax is verbose and cumbersome.

Do not use C-style casts. Instead, use these C++-style casts when explicit type conversion is necessary.

- Use brace initialization to convert arithmetic types (e.g. `int64{x}`). This is the safest approach because code will not compile if conversion can result in information loss. The syntax is also concise.
- Use `static_cast` as the equivalent of a C-style cast that does value conversion, when you need to explicitly up-cast a pointer from a class to its superclass, or when you need to explicitly cast a pointer from a superclass to a subclass. In this last case, you must be sure your object is actually an instance of the subclass.
- Use `reinterpret_cast` to do unsafe conversions of pointer types to and from integer and other pointer types. Use this only if you know what you are doing and you understand the aliasing issues.

Preprocessor Macros

Avoid defining macros, especially in headers; prefer `inline` functions, `enums`, and `const` variables. Name macros with a project-specific prefix.

Macros mean that the code you see is not the same as the code the compiler sees. This can introduce unexpected behavior, especially since macros have global scope.

0 and nullptr/NULL

Use `0` for integers, `0.0` for reals, `nullptr` for pointers, and `'\0'` for chars.

Use `0` for integers and `0.0` for reals.

For pointers (address values), use `nullptr`, as this provides type-safety.

auto

Use `auto` to avoid type names that are noisy, obvious, or unimportant - cases where the type doesn't aid in clarity for the reader. You can either continue to use manifest type declarations when it helps readability or use auto.

- C++ type names can be long and cumbersome, especially when they involve templates or namespaces.
- When a C++ type name is repeated within a single declaration or a small code region, the repetition may not be aiding readability.
- It is sometimes safer to let the type be specified by the type of the initialization expression, since that avoids the possibility of unintended copies or type conversions.

Braced Initializer List

Use braced initializer list whenever possible as this has many advantages. Beware not to use them when initializing vectors or other pitfalls that you may look into.

```
auto i {0}; // OK
```

Aliases

There are several ways to create names that are aliases of other entities:

```
typedef Foo Bar;
using Bar = Foo;
using other_namespace::Foo;
```

U`sing` is preferable to `typedef`, because it provides a more consistent syntax with the rest of C++ and works with templates.

- Aliases can improve readability by simplifying a long or complicated name.
- Aliases can reduce duplication by naming in one place a type used repeatedly in an project, which *might* make it easier to change the type later.

Comments

Though a pain to write, comments are absolutely vital to keeping our code readable. The following rules describe what you should comment and where. But remember while comments are very important, the best code is self-documenting. Giving sensible names to types and variables is much better than using obscure names that you must then explain through comments.

Inheritance

Composition is often more appropriate than inheritance. When using inheritance, make it `public`.

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".

Multiple inheritance is especially problematic, because it often imposes a higher performance overhead (in fact, the performance drop from single inheritance to multiple inheritance can often be greater than the performance drop from ordinary to virtual dispatch), and because it risks leading to "diamond" inheritance patterns, which are prone to ambiguity, confusion, and outright bugs.

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: `Bar` subclasses `Foo` if it can reasonably be said that `Bar` "is a kind of" `Foo`.

Limit the use of `protected` to those member functions that might need to be accessed from subclasses. Note that data members should be private.

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.

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

Enumerator

Prefer `enum classes` to `enums.`

What is the difference between two?

- enum classes - enumerator names are **local** to the enum and their values do *not* implicitly convert to other types (like another enum or int)
- Plain enums - where enumerator names are in the same scope as the enum and their values implicitly convert to integers and other types

Deleted functions

Prefer deleted functions to private undefined ones.

Any function may be deleted, including non-member functions and template instantiations.

Iterators

Prefer `const_iterators` to `iterators.`

In maximally generic code, prefer non-member versions of `begin, end, rbegin` etc.., over their member function counterparts.

Syntactic sugar

If available, syntactic sugar **must** always be used.