C++ Coding Standards for the 2014 FNAL C++ Course

Scientific Software Infrastructure Department/Scientific Computing Division/Fermilab
Revision 2

Contents

1 Introduction **1**
2 Naming Guidelines **1**
3 Coding Guidelines **2**
4 Design Guidelines **5**
Bibliography **5**

1 Introduction

This is a brief summary of C++ coding guidelines for the 2014 edition of the FNAL C++programming course. More substantial suggestions, and explanations, are available in [1] and [2]. Your experiment may have additional or alternate coding guidelines, which should take precedence.

2 Naming Guidelines

1. C++ header files use the suffix `.hh` *e.g.*, `CaloCluster.hh`.
2. C++ source files use the suffix `.cc` *e.g.*, `CaloCluster.cc`.
3. Name each header file after the class it defines.
4. Name each source file after the class it implements.
5. For `class`, `struct`, `typedef`, `template` and `enum` names use upper case first letter and upper case for the initial letter of each "word" in the name ("camel case") *e.g.*, `GeometryBuilder`.
6. For namespaces use lower case *e.g.*, `namespace fc`.
7. Start method names with lower case, use upper case initials for following words *e.g.*, `collisionPoint()`.[1]
8. Start data member names with a leading underscore followed by a lower case letter to distinguish data member from getter method *e.g.*, `_momentum`.
9. Do not use single character names, except for loop indices.
10. Do not use special characters, except for the underscore where allowed.

[1] Allowed exception: implementation of virtual methods inherited from external packages with other conventions, *e.g.*, `Draw()` function required by ROOT.

11. Do not use two leading underscores __.

3 Coding Guidelines

1. Protect each header file from multiple inclusion with code guards:

```
#ifndef PackageName_FileName_hh
#define PackageName_FileName_hh
(body of header file)
#endif
```

2. Each header file should contain only one class declaration.[2]
3. Header files must not contain any implementation except for class templates and code to be inlined.
4. Do not inline virtual functions.
5. The only member functions that should be routinely inlined are simple accessors (getters), *e.g.*,

```
inline double momentum() const { return _momentum; }
```

6. Classes must not have public data members.
7. Use forward declarations if they are sufficient.
8. Do not forward declare an entity from another package; use the forward declaration header provided by the other package.
9. Do not use absolute directory names in `#include` directives.
10. Do not use non-`const` global data (no class `static` data, no function scope `static` data).
11. Use `nullptr`, not `0` and not `NULL`.
12. Use types like `int`, `long`, `size_t`, `ptr_diff_t` consistently and without mixing them (important for 64-bit architectures).
13. Use the `bool` type for logical values.
14. Define constants using `enum` classes, `const` or `constexpr`. Never `#define` or magic numbers.
15. Never use `malloc`, `calloc`, `realloc` or `free`.
16. Use smart pointers to manage memory. Use `std::shared_ptr` only when sharing is intended; use `std::unique_ptr` when no sharing is wanted. Do not use `std::auto_ptr`.
17. Prefer `std::make_shared` to using `new` when creating shared pointers:

```
auto sp = std::make_shared<fc::Hit>(...);
```

18. Have assignment operators return a reference to `*this`.
19. Do not use `goto`.
20. Make `const` all member functions that do not need to be non-`const`.
21. Do not use `mutable` data members.
22. Do not use `const_cast`.
23. Do not use function-like macros.
24. Never use C-style casting.

[2] And possibly any supporting functions and helper classes.

25. Do not use `union` types.
26. Do not write any function that the compiler would correctly generate for you.

```
struct Point
{
  // The compiler-generated default constructor will use
  // these values.
  double x = 0.0;
  double y = 0.0;
}
```

    The compiler will generate a correct destructor, default constructor, copy constructor, and copy assignment; do not write them.
27. To prevent copying of instances of a `class`, use `= delete`.

```
class X {
  X(X const&) = delete; // no copies allowed
};
```

28. If you declare any one of copy assignment, move assignment, copy constructor, move constructor, or destructor, you should declare all five. See http://www.stroustrup.com/C++11FAQ.html#default2.
29. Do not inherit from a class that has no virtual functions.
30. If a class has at least one `virtual` member function it must have a `public virtual` destructor.[3]
31. Always declare `virtual` functions in derived classes using `override`.
32. Pass by value arguments of primitive type, unless the called function is to modify the value.
33. Pass objects of `class` type by (possibly `const`) reference.[4]
34. Declare a reference or pointer argument passed to a function as `const` if the function does not change the object bound to it.
35. The argument to a copy constructor and to an assignment operator must be a `const` reference.
36. Do not let `const` member functions change the state of the object. Do not modify data members. Do not call non-`const` functions on data members.
37. Member functions declared `const` should not return non-`const` references to data members.
38. Member functions declared `const` should not return pointers-to-non-`const`.

```
int const* MyClass::value() const { return _value; } // ok
int* const MyClass::value() const { return _value; } // bad
int* MyClass::value() const { return _value; }       // bad
```

39. A function must never return a reference or pointer to a function-scope variable.
40. The `public`, `protected` and `private` sections of a class should be declared in that order.
41. A function that returns an object whose ownership is transferred to the caller must do so using a smart pointer.

[3] A class derived from one that has a virtual destructor will have a virtual destructor automatically; the compiler-generated destructor will be virtual.

[4] An exception can be made for classes that take advantage of move semantics.

42. Provide argument names in member function declarations in the header file to indicate their usage.
43. Variables should be declared in the smallest scope possible.
44. Variables should be initialized upon declaration. Prefer use of the uniform initialization syntax.

```
std::map<int, std::string> const z { {1, "cow"}, {7, "dog"} };
```

45. Prefer `emplace_back` to `push_back` when possible.

```
std::vector<fc::Helix> tracks;
tracks.emplace_back(0.05, 0.001, 0.002, 1.3, 1.0, alpha);
```

46. All member data should be initialized. This can be done either in the variable declaration or in the initializer list.
47. Prefer delegating constructors to calling of an initialization function in the body of a constructor.

```
class X {
public:
  explicit X(double d) : _value(d) { }
  X() : X(3.14) { }
private:
  double _value;
};
```

48. Single-argument constructors should be declared `explicit`.
49. Data members of a class must not be redefined in derived classes.
50. Prefer the range-for loop to iterator-based loops. Pay attention to when references should be used to avoid copying.

```
std::vector<std::string> names;
...
for ( auto const& name : names) {
  // the const& avoids a copy.
  // use name here ...
}
```

51. Use `auto` to declare a local variable for which the type can be determined by the initializer. This avoids accidental type conversions and makes the code work if the type of the initializer is changed in the future.

```
int foo() { return 3; }
auto i = foo();
```

52. Use Standard Library algorithms when possible. Lambda-expressions make this convenient.

```
std::vector<double> nums {4.4, 1.1., 2.2, 3.3};
// Do an ascending sort, rather than descending sort.
std::sort(begin(nums), end(nums),
          [](double x, double y) { return x > y});
```

4 Design Guidelines

1. Avoid inlining unless you are sure you have a relevant performance problem (except for simple accessors).
2. Make sure each header file `#include`s all the headers necessary to declare the functions and types it uses.
3. Use `std::string`, not `char *`.
4. Do not use C-style arrays, use Standard Library containers (commonly `std::vector` or `std::array`).
5. Do not use exception specifications.
6. Do not use the Singleton pattern.[5]
7. Use namespaces to group collaborating classes that provide a certain functionality e.g. `fc`.
8. Encapsulate functions related to a `class` in the same namespace as the `class`.
9. Do not introduce compilation or link dependencies that are not required. E.g., the `fc::Track` class does not depend upon the algorithms that create instances of `fc::Track`, and the `class` lives in a different library than the algorithms.
10. Avoid nested control structures. If a function has more than one level of control structure, the inner control structure should be replaced by a well-named function.
11. Do not use member data to avoid passing arguments between member functions of a class.
12. Do not write long functions. Rather than a large function with commented blocks of code, write a short function that calls several well-named functions, each of which does one block of work.
13. Each `class` should have one clearly-defined purpose. A well-designed `class` can have its purpose expressed in a short descriptive comment.

References

Bibliography

[1] Sutter H and Alexandrescu A 2005 *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices* (Upper Saddle River, NJ, USA: Pearson Education)

[2] Programming Research L 2013 High integrity c++ coding standard, version 4.0 URL http://www.codingstandard.com

[5] Your experiment's framework probably provides "services" to solve the problem this pattern is often used to solve.