

STAR C++ Coding Guidelines

Authors:
Mustafa Mustafa
Thomas Ullrich
Anselm Vossen


Introduction

This document is a draft of new C++ coding guidelines compiled for the STAR collaboration by the above mentioned authors. This effort was initiated by the STAR computing coordinator Jerome Lauret on October 31, 2014. The charge can be viewed [here](#). The committee produced two documents, one for the coding guidelines seen here, and one for the naming and formatting guidelines that can be viewed [here](#).

The committee based their work on the existing guidelines, expanded them for clarity, and added new material where it saw fit. The coding guidelines include the new C++11 standard. We have made heavy use of the C++ Google Style guide at <http://google-styleguide.googlecode.com> using their xml and css style sheets.

The goal of this guide is to manage the complexity of C++ (often in conjunction with ROOT) by describing in detail the dos and don'ts of writing C++ code. These rules exist to keep the STAR code base manageable while still allowing coders to use C++ language features productively. In some cases we constrain, or even ban, the use of certain C++ and ROOT features. We do this to keep code simple and to avoid the various common errors and problems that these features can cause. We also had to take into account that millions of lines of STAR code exist. For a new experiment the guidelines certainly would look different in places but we have to live with the legacy of existing code and the guidelines under which they were written.

Note that this guide is not a C++ tutorial: we assume that the reader is familiar with the language. We marked parts of the guidelines that address specifically new C++11 features.

Each style point has a summary for which additional information is available by toggling the accompanying arrow button that looks this way: . You may toggle all summaries with the big arrow button:

Toggle all summaries



Toggle all extra details

Table of Contents

Introduction	
Header Files	The #define Guard Forward Declarations Inline Functions Names and Order of Includes
Namespaces	General Guideline Using Declarations and Directives std Namespace
Scoping	Nonmember and Global Functions Local Variables Variables Initialization Brace Initialization Global Variables Global Variables Initialization Static Variables in Functions
Classes	Constructors Initialization Virtual Functions in Constructors and Destructors Copy Constructors and Assignment Operator Copy and Move Delegating and Inheriting Constructors Structs vs. Classes Destructors Inheritance Multiple Inheritance Interfaces Operator Overloading Access Control Keywords Access Control Friend Declaration
ROOT Related Issues	ROOT Types ROOT Mathematical Function
Others	Attributes Exceptions Use of const Use of constexpr Suffix Return Type Syntax Smart Pointers Magic Numbers (Hard Coded Numbers) Preprocessor Macros Write Short Functions Run-Time Type Information (RTTI) Casting Variable-Length Arrays and alloca() Increment and Decrement Operators Loops and Switch Statements Range-for Statement Integer Types Portability 0 and nullptr sizeof auto Non-member begin() and end() static_assert and type traits Rvalue Reference and Move Semantics
Exceptions to the Rules	Existing Non-Conformant Code

Important Note

Displaying Hidden Details in this Guide

This style guide contains many details that are initially hidden from view. They are marked by the triangle icon, which you see here on your left. The first level of hidden information is the subsection *Summary* in each rule and the second level of hidden information is the optional subsection *Extra details and exceptions to the rule*. Click the arrow on the left now, you should see "Hooray" appear below.

Header Files

In general, every `.cxx` file should have an associated `.h` file. Each header file should contain only one or related class declarations for maintainability and for easier retrieval of class definitions.

Correct use of header files can make a huge difference to the readability, size and performance of your code. The following rules will guide you through the various pitfalls of using header files.

The #define Guard

All header files should have `#define` guards to prevent multiple inclusion. The format of the symbol name should be `<FILE>_H`.

Forward Declarations

You may forward declare ordinary classes in order to avoid unnecessary `#includes`.

Inline Functions

As a general rule, put function definitions into the `.cxx` file and let the compiler decide what gets inlined (it can decide anyway, regardless of the `inline` keyword). Use `inline` when you require the implementation of a function in multiple translation units (e.g. template classes/functions).

Names and Order of Includes

Include headers from external libraries using angle brackets. Include headers from your own project/libraries using double quotes.

Do not rely on implicit includes. Make header files self-sufficient.

There are two types of `#include` statements: `#include <myFile.h>` and `#include "myFile.h"`.

- Include headers from external libraries using angle brackets.

```
#include <iostream>
#include <cmath>
#include <TH1D.h>
```

- Include headers from your own project or any STAR related project using double quotes.

```
#include "MyClass.h"
#include "StEnumeration.h"
```

The header files of external libraries are obviously not in the same directory as your source files. So you need to use angle brackets.

Headers of your own application have a defined relative location to the source files of your application. Using double quotes, you have to specify the correct relative path to the include file.

Include order

Another important aspect of include management is the include order. Typically, you have a class named Foo, a file Foo.h and a file Foo.cxx . The rule is : In your file Foo.cxx, you should include Foo.h as the first include, before the system includes.

The rationale behind that is to make your header standalone.

Let's imagine that your Foo.h looks like this:

```
class Foo
{
public:
    Bar getBar();
};
```

And your Foo.cxx looks like this:

```
#include "Bar.h"
#include "Foo.h"
```

Your Foo.cxx file will compile, but it will not compile for other people using Foo.h without including Bar.h. Including Foo.h first makes sure that your Foo.h header works for others.

```
// Foo.h
#include "Bar.h"
class Foo
{
public:
    Bar getBar();
};
```

```
// Foo.cxx
#include "Foo.h"
```

For more details: Getting #includes right.

Namespaces

Namespaces subdivide the global scope into distinct, named scopes, and thus are useful for logically grouping related types and functions and preventing name collisions. In C++ it is in general very good practice to use namespaces, especially in libraries. However, historically STAR software makes little to no use of namespaces but rather uses a specific naming scheme (prefixes) to indicate the scope (e.g. `StEmc...`, `StTpc...` etc). While certain tools in STAR can handle namespaces (such as `cons`) others would be very cumbersome to adapt.

General Guideline

Namespaces are for legacy reasons depreciated in STAR. As with every guideline there might be exceptions, especially in end user code. However, care should be taken to check for possible side effects. Namespaces should be entirely avoided in the context of `StEvent`.

Using Declarations and Directives

Don't write namespace using declarations or using directives in a header file or before an `#include`.

std Namespace

Do not declare anything in namespace `std`, not even forward declarations of standard library classes.

Scoping

Nonmember and Global Functions

Nonmember functions (also known as global functions) should be within a namespace.

Local Variables

Declare variables as locally as possible.

Variables Initialization

Always initialize variables.

since
C++11

Brace Initialization

Prefer initialization with braces except for single-argument assignment.

Global Variables

Variables declared in the global scope are not allowed. Other global variables, including static class variables and variables in namespace scope, should be avoided where other means of communication are possible.

Global Variables Initialization

In the rare and justified cases where you use global variables, including file-static variables, static member variables and variables in namespace scope, initialize them statically.

Static Variables in Functions

Static variables in functions (called "function-local static variables" in the C++ terminology) are expensive and need care on destruction. Prefer to use static class variables where possible.

Classes

Classes are the fundamental unit of code in C++. Naturally, we use them extensively. This section lists the main dos and don'ts you should follow when writing a class.

Constructors

Every class should have at least one constructor. All uninitialized variables should be initialized in the constructor.

Initialization

Declare and initialize members variables in the same order. Prefer initialization (in the constructor initializer list or in-class) to assignment (in the constructor function body).

Virtual Functions in Constructors and Destructors

Do not call virtual functions in constructors and destructors.

Copy Constructors and Assignment Operator

Each class should have an assignment operator and a copy constructor when they allocate subsidiary data structures on the heap or consume any other kind of shared resources. Be aware of data slicing for polymorphic classes.

Copy and Move

Explicitly enable or delete the copy constructor/assignment operator. Only implement move constructors/assignment operators if your class needs optimizations for move semantics.

since
C++11

Delegating and Inheriting Constructors

Use delegating and inheriting constructors when they reduce code duplication. Be aware of self delegation.

Structs vs. Classes

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

Destructors

Every class must free resources (objects, IO handlers, etc.) it allocated during its lifetime. The base class destructors must be declared virtual if they are public.

Inheritance

When using inheritance, make it `public` and declare overridden methods as `override` or `final`. However, composition is often more appropriate than inheritance especially if a class is not designed to be a base class.

Multiple Inheritance

Use multiple inheritance implementation only when at most one of the base classes has an implementation; all other base classes must be pure interface classes.

Interfaces

If a class was designed as a pure interface, keep it as a pure interface.

Operator Overloading

When overloading operators keep the same semantics.

Access Control Keywords

The `public`, `protected` and `private` keywords must be used explicitly in the class declaration in order to make code more readable. It is recommended to list the public data member and methods first since they define the global interface and are most important for the user/reader.

Access Control

Hide internals. Avoid returning handles to internal data managed by your class.

Friend Declaration

The use of friend declarations should be avoided where possible.

ROOT Related Issues

ROOT Types

Prefer the use of fundamental types built-in C++ over ROOT types, except where absolutely required.

ROOT Mathematical Function

Prefer the use of mathematical function available in the C++ standard (`<cmath>`) over those provided by ROOT.

Others

since
C++11

Attributes

"Attributes" is a new standard syntax aimed at providing some order in the mess of facilities for adding optional and/or vendor specific information (GNU, IBM, ...) into source code. The use of attributes is discouraged in STAR.

Exceptions

Use C++ exceptions instead of return codes for error handling. Do not use exceptions to return values.

Use of `const`

Declare objects that are logically constant as `const`. Design const-correct interfaces. Consider `constexpr` for some uses of `const`.

since
C++11

Use of `constexpr`

In C++11, use `constexpr` to define true constants or to ensure constant initialization.

since
C++11

Suffix Return Type Syntax

C++11 new suffix return value syntax (or extended function declaration syntax) represents another use for `auto`. It is useful mostly in templates and in methods where the return type is the class itself. The new return syntax, however, is not as easy to read as the standard method and should only be used where necessary. It should not be regarded as an alternative way of defining a simple function.

since
C++11

Smart Pointers

It is a modern C++ idiom to get rid of naked pointers whenever possible. However, it is currently difficult to devise an error free scheme where smart pointers can live in harmony with ROOT object ownership and management rules. Avoid using smart pointers in STAR code. This decision could be revisited in the future if conflict with ROOT is resolved.

Magic Numbers (Hard Coded Numbers)

Avoid magic numbers (hard coded numbers).

Preprocessor Macros

Avoid macros. Use inline functions, constexpr functions, enums, constexpr variables, or templates instead if they can solve the problem.

Write Short Functions

Prefer small and focused functions.

Run-Time Type Information (RTTI)

Use RTTI with caution. If you find yourself overusing `dynamic_cast` consider the design of your code and classes.

Casting

In general, avoid designs that require casting. You may use `static_cast` when necessary, but avoid `const_cast` and `reinterpret_cast`. C-casts are forbidden.

Variable-Length Arrays and `alloca()`

Don't use variable-length arrays or `alloca()`.

Increment and Decrement Operators

Prefer the prefix form of the increment (`++i`) and decrement (`--i`) operators because it has simpler semantics.

Loops and Switch Statements

If not conditional on an enumerated value, switch statements should always have a `default` case. Empty loop bodies should use `{}` or `continue`.

since
C++11

Range-for Statement

Use reference to elements in range-for statements especially when dealing with large objects. Prefer ordinary loops when you need the index information.

since
C++11

Integer Types

Per default, use `int` if you need an integer type. If you need to guarantee a specific size use the new extended integer types defined in `<cstdint>`.

Portability

Take extra care of the code portability. Bear in mind problems of printing, comparisons, and structure alignment related to 32-bit and 64-bit data representations .

0 and nullptr

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

sizeof

Prefer `sizeof(varname)` to `sizeof(type)`.

since
C++11

auto

If the compiler is able to determine the type of a variable from its initialization, you don't need to provide the type. This is achieved by using the `auto` keyword.

Use `auto` to avoid type names that are just clutter. Continue to use manifest type declarations when it helps readability, and never use `auto` for anything but local variables.

Use `auto` against verbosity, not consistency. In cases where the rhs expression is an integer or floating point literal the use of `auto` is strongly discouraged.

since
C++11

Non-member begin() and end()

The non-member `begin()` and `end()` functions are a new addition to the standard library, promoting uniformity, consistency and enabling more generic programming. They work with all STL containers, but more than that they are overloadable, so they can be extended to work with any type. Overloads for C-like arrays are also provided. The use of non-member `begin()` and `end()` is encouraged.

since
C++11

static_assert and type traits

`static_assert()` performs an assertion check at compile-time. Type traits and `static_assert` is mostly for template class developer. Since the use of templates in STAR is minimal, these new C++11 features will be rarely used, if at all. There's no argument against using this feature if needed.

since
C++11

Rvalue Reference and Move Semantics

Classes should have move constructor and assignment operator OR explicitly have them deleted using `= delete` specifier.

Exceptions to the Rules

The coding conventions described above have to be followed. However, like all good rules, these sometimes have exceptions.

Existing Non-Conformant Code

It is permissible to deviate from the rules when dealing with code that does not conform to

these guidelines.

Parting Words

Use common sense and *BE CONSISTENT*.

The point about having style guidelines is to have a common vocabulary of coding so people can concentrate on what the programmer is saying, rather than on how he/she is saying it.

OK, enough writing about writing code; the code itself is much more interesting. Have fun!

References

[1] Herb Sutter on software, hardware, and concurrency blog [<http://herbsutter.com/2013/05/09/gotw-1-solution>]