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 guidlines include the new C++11 standard. We have made heavy use of the C++ Google Style guide at http://google-styleguide.googlecode.com.

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 guidlines that address specifically new C++11 features.

START EDITING HERE

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

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.

Hooray! Now you know you can expand points to get more details. Alternatively, there are an "expand all summaries" and an "expand all summaries and extra details" at the top of this document.

Header Files

In general, every `.cxx` file should have an associated `.h` file. There are some common exceptions, such as unittests and small `.cxx` 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.

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 `<PROJECT>_<PATH>_<FILE>_H_`.

To guarantee uniqueness, they should be based on the full path in a project's source tree. For example, the file `foo/src/bar/myFile.h` in project `foo` should have the following guard:

```
#ifndef FOO_BAR_MYFILE_H_
#define FOO_BAR_MYFILE_H_
```

```
...

#endif  // FOO_BAR_MYFILE_H_
```

Forward Declarations

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

Definition:

A "forward declaration" is a declaration of a class, function, or template without an associated definition. `#include` lines can often be replaced with forward declarations of whatever symbols are actually used by the client code.

Pros:

- Unnecessary `#include`s force the compiler to open more files and process more input.
- They can also force your code to be recompiled more often, due to changes in the header.

Cons:

- It can be difficult to determine the correct form of a forward declaration in the presence of features like templates, typedefs, default parameters, and using declarations.
- Forward declaring multiple symbols from a header can be more verbose than simply including the header.
- Forward declarations of functions and templates can prevent the header owners from making otherwise-compatible changes to their APIs; for example, widening a parameter type, or adding a template parameter with a default value.
- Forward declaring symbols from namespace `std::` usually yields undefined behavior.

Decision:

- When using a function declared in a header file, always `#include` that header.
- When using a class template, prefer to `#include` its header file.
- When using an ordinary class, relying on a forward declaration is OK, but be wary of situations where a forward declaration may be insufficient or incorrect; when in doubt, just `#include` the appropriate header.
- Do not replace data members with pointers just to avoid an `#include`.

Always `#include` the file that actually provides the declarations/definitions you need; do not rely on the symbol being brought in transitively via headers not directly included. One exception is that `myFile.cxx` may rely on `#include`s and forward declarations from its corresponding header file `myFile.h`.

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).

Definition:

The inline keyword indicates that inline substitution of the function body at the point of call is to be preferred to the usual function call mechanism. But a compiler is not required to perform this inline substitution at the point of call.

Functions that are defined within a class definition are implicitly inline.

An inline function must be defined in every translation unit from where it is called. It is undefined behavior if the definition of the inline function is not the same for all translation units. Note that this implies that the function is defined in a header file. This can have an impact on compile time and lead to longer (= less efficient) development cycles.

Note that the inline keyword has no effect on the linkage of a function. Linkage can be changed via unnamed namespaces or the static keyword.

Decision:

If you add a new function, put it into the `.cxx` file per default. Small functions, like accessors and mutators may be placed into the `.h` file instead (`inline`). Also, most template function implementations need to go into the `.h` file. If you later determine that a function should be moved from the `.cxx` file into the `.h` file, please make sure that it helps the compiler in optimizing the code. Otherwise you're just increasing compile time.

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 <QtCore/QDate>
#include <zlib.h>
```

- Include headers from your own project using double quotes.

```
#include "MyClass.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.cpp 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.

General guideline

Use a namespace for the project, a second nested namespace for code inside a sub-project. Use one or two more nested namespaces to logically group types and functions together.

Example of a typical class definition within a project and sub-project namespace:

```
namespace Project
{
namespace SubProject
{

class MyClass
{
...
};

} // namespace SubProject
} // namespace Project
```

A nonmember function that is logically tied to a specific type should be in the same namespace as that type.

Using declarations and directives

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

```
# include "Bar.h"
// OK in .cxx after include statements
using namespace Foo;

// sometimes a using declaration is preferable, to be precise
// about the symbols that get imported
using Foo::Type;
```

```
// Forbidden in .h -- This pollutes the namespace.
using namespace Foo;
```

▽ **Extra details and exceptions to the rules:**

The using directive can sometimes be useful in header files to import one namespace into another one. This can effectively hide a namespace from the public interface, similar to what an inline namespace does.

Unnamed namespaces

Unnamed namespaces are allowed and even encouraged in `.cxx` files, they are not allowed in `.h` files.

Definition:

All symbols inside an unnamed namespace (sometimes also called anonymous namespace) will have internal linkage. Thus, it is impossible to access these objects from other translation units (.cxx files).

Pros:

Because of the internal linkage it is easier to reason about the code. This can lead to better optimization from compilers and clearer encapsulation.

Decision:

Use unnamed namespaces in `.cxx` files whenever possible.

```
namespace Project {
namespace {                          // This is in a .cxx file.
int someFunction(int value) { return value + 1; }
}  // unnamed namespace

void someOtherFunction() {
int value = 0;
value = someFunction(value);
...
}
}
```

For the above code the compiler can see that `someFunction` is called from only one place. It can thus inline the code and drop `someFunction`.

▽ **Extra details and exceptions to the rules:**

In some special cases an unnamed namespace can be useful in a header file. Leave such decisions to architecture designers.

Namespace aliases

Namespace aliases are allowed anywhere in a `.cxx` file, inside the named namespace that wraps an entire `.h` file and in functions and methods.

```
// Shorten access to some commonly used names in .cxx files.
namespace Fbz = ::Foo::Bar::Baz;

// Shorten access to some commonly used names (in a .h file).
namespace Librarian {
// The following alias is available to all files including
// this header (in namespace Librarian):
// alias names should therefore be chosen consistently
// within a project.
namespace Pds = ::PipelineDiagnostics::Sidetable;

inline void myInlineFunction() {
// namespace alias local to a function (or method).
namespace Fbz = ::Foo::Bar::Baz;
...
}
}  // namespace Librarian
```

Note that an alias in a .h file is visible to everyone including that file, so public headers (those available outside a project) and headers transitively included by them, should avoid defining aliases, as part of the general goal of keeping public APIs as small as possible.

std namespace

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

Declaring entities in namespace `std` represents undefined behavior, i.e., not portable. To declare entities from the standard library, include the appropriate header file.

Scoping

Nonmember and Global Functions

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

Putting nonmember functions in a namespace avoids polluting the global namespace. Static member functions are an alternative as long as it makes sense to include the function within the class.

```
namespace MyNamespace {
void doGlobalFoo(); // Good -- doGlobalFoo is within a namespace.

class MyClass {
public:
...
// Good -- doGlobalBar is a static member of class MyClass and
// has a reason to be part of this class (not shown here).
static Bar* doGlobalBar();
}
```

Local Variables

Declare variables as locally as possible.

Variables whose lifetime are longer than necessary have several drawbacks:

- They make the code harder to understand and maintain.
- They can't be always sensibly initialized.

▽ **Extra details and exceptions to the rules:**

- It can be sometimes more efficient to declare a variable (usually of an object type) outside a loop.

  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.

  ```
  // Inefficient implementation:
  for (int i = 0; i < bigNumber; ++i) {
  Foo foo;  // My ctor and dtor get called bigNumber times each.
  foo.doSomething(i);
  }
  ```

  It may be more efficient to declare such a variable used in a loop outside that loop:

  ```
  Foo foo;  // My ctor and dtor get called once each.
  for (int i = 0; i < bigNumber; ++i) {
  foo.doSomething(i);
  }
  ```

- This item does not apply to constants, because constants don't add a state.

Variables Initialization

Always initialize variables.

Do not separate initialization from declaration, e.g.

```
int value;
value  =  function();          //  Bad  --  initialization  separate  from
declaration.
```

```
int value = function();  // Good -- declaration has initialization.
```

Use a default initial value or ? to reduce mixing data flow with control flow.

```
int speedupFactor;       // Bad: does not initialize variable
if (condition) {
speedupFactor = NoFactor;
}
else {
speedupFactor = DoubleFactor;
```

```
}
```

```
int speedupFactor = DoubleFactor; // Good: initializes variable
if (condition) {
speedupFactor = NoFactor;
}
```

```
int  speedupFactor  =  condition  ?  NoFactor  :  DoubleFactor;  //  Good:
initializes variable
```

Prefer declaration of loop variables inside a loop, e.g.

```
int i;                      // Bad: does not initialize variable
for (i = 0; i < number; ++i) {
doSomething(i);
}
```

```
for (int i = 0; i < number; ++i) {
doSomething(i); // Good
}
```

since C++11

Brace Initialization

Prefer initialization with braces except for single-argument assignment.

In C++11, the brace initialization syntax for builtin arrays and POD structures has been extended for use with all other datatypes.

Example of brace initialization:

```
std::vector<std::string> myVector{"alpha", "beta", "gamma"};
```

Example of single-argument assignments:

```
int value = 3;                                // preferred style
std::string name = "Some Name";

int value { 3 };                              // also possible
std::string name{ "Some Name" };
std::string name = { "Some Name" };
```

User data types can also define constructors that take `initializer_list`, which is automatically created from *braced-init-list*:

```
#include <initializer_list>
class MyType {
public:
// initializer_list is a reference to the underlying init list,
// so it can be passed by value.
MyType(std::initializer_list<int> initList) {
for (int element : initList) { .. }
}
};
MyType myObject{2, 3, 5, 7};
```

Finally, brace initialization can also call ordinary constructors of data types that do not have `initializer_list` constructors.

```
// Calls ordinary constructor as long as MyOtherType has no
// initializer_list constructor.
class MyOtherType {
public:
explicit MyOtherType(std::string name);
MyOtherType(int value, std::string name);
};
MyOtherType object1 = {1, "b"};
// If the constructor is explicit, you can't use the "= {}" form.
MyOtherType object2{"b"};
```

Never assign a *braced-init-list* to an auto local variable. In the single element case, what this means can be confusing.

```
auto value = {1.23};        // value is an initializer_list<double>
```

```
auto value = double{1.23};   // Good -- value is a double, not an
initializer_list.
```

For clarity of the examples above we use directly explicit values, however following the rule about magic numbers requires to define all such numbers as named constants or `constexpr` first.

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.

Definition:

A global variable is a variable that can be accessed (theoretically) from everywhere in the program. The adjective "global" should rather be understood as concerning its linkage, not whether it is in the global scope.

```
int gBar;           // this is obviously global
class Something
{
private:
static int sId; // but this one too (details at the end of the rule)
};
namespace NotGlobalScope {
Foo fooObject;    // and finally this one as well
}
```

Pros:

Global variables are a simple solution to sharing of data.

Cons:

Global variables make it harder to reason about the code (for humans and compilers): the smaller the number of variables a given region of code reads and writes, the easier. Global variables can be read and written from anywhere. Therefore, global variables pose a challenge to the optimizer.

Decision:

We want to reduce the shared data in our software to the unavoidable minimum. Therefore, global variables should be avoided where other means of communication are possible.

▽ **Extra details and exceptions to the rules:**

Note that a private static class variable sId is global. For example two threads having each an instance of the class could access sId via these instances.

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.

Definition:

- This rule additionally applies to file-static variables.
- A global variable is statically initialized if the type has no constructor or a `constexpr` constructor. This is the case for fundamental types (integers, chars, floats, or pointers) and POD (Plain Old Data: structs/unions/arrays of fundamental types or other POD structs/unions).

Pros:

Dynamic initialization of globals can be used to run code before `main()`. Destructors of globals can be used to run code after `main()`. Dynamic initialization of globals in dynamically loaded objects can be used to run code on plugin load.

Cons:

It is very hard to reason about the order of execution of such functions. Especially if dynamically initialized globals are present in shared libraries that are linked into dynamically loaded objects, few people understand the semantics.

As an example do this:

```
struct Pod {
int indexes[5];
float width;
};
struct LiteralType {
int value;
constexpr LiteralType() : value(1) {}
};

Pod gData;
LiteralType gOtherData;
```

But not this :

```
class NotPod {
NotPod();
};
NotPod gBadData;  // dynamic constructor
```

Decision:

Global variables must be initialized statically.
We only allow global variables to contain POD data. This rule completely disallows `std::vector` (use `std::array` instead), or `std::string` (use `const char []`) for global variables.

▽ **Extra details and exceptions to the rules:**

If there is a need for startup or shutdown code, dynamic constructors may be used. But only if:

- The dependencies on other data are minimized.
- It is documented what code depends on this and why there is no issue of incorrect calling order.
- Side-effects are clearly understood and documented.

Example that exhibits the problem of execution order:

```
// Struct.h:
#include <string>
struct Struct {
static std::string sString;
};

// Struct.cxx:
std::string Struct::sString = "Hello World";

// main.cxx:
#include "Struct.h"
#include <iostream>

std::string gAnotherString = Struct::sString;

int main() {
std::cout << gAnotherString << std::endl;
return 0;
}
```

This program will either output "Hello World" or crash, depending on the initialization order (with GCC on Linux it depends on whether you link with `g++ Struct.o main.o` or `g++ main.o Struct.o`).

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.

Definition:

Function-local static variables are initialized on first use and destructed in the reverse order of construction.

Pros:

The variable is lazily initialized. Therefore the order of construction is better under control. Also the cost of initialization is only incurred if it is really needed.

C++11

Since C++11, function-local static variables are guaranteed to be initialized exactly once, even in a multi-threaded environment.

Cons:

Because of the thread-safe lazy initialization, function-local static variables have an extra cost compared to other function variables.

Decision:

Use static class variables rather than function-local static variables. Therefore, do this :

```
class Something
{
public:
int generateId() {
return Something::sId++;
}

private:
static int sId; // initialized in .cxx
};
```

instead of :

```
class Something
{
public:
int generateId() {
static int sId = 0;
return sId++;
}
};
```

▽ **Extra details and exceptions to the rules:**

Function-local static variables can be used to build factory functions for lazily initialized global objects. This makes it possible to safely use dynamically initialized types in the global scope.

In case function-local static variables are nevertheless used, it is best to avoid non-owning references because their destruction happens after return from `main()`, as shown on the following code snippets :

```
std::string &globalMessage()
{
static std::string message = "Hello world.";
return message;
}

int main()
{
std::cout << globalMessage() << '\n';
globalMessage() = "Goodbye cruel world.";
std::cout << globalMessage() << '\n';
return 0;
}
// prints:
// Hello world.
// Goodbye cruel world.
```

```
class BreakTheCode
```

```
{
public:
void setReference(int *value) { mValuePointer = value; }
~BreakTheCode() { *mValuePointer += 2; }

private:
int *mValuePointer = 0;
};

BreakTheCode &globalBreaker()
{
static BreakTheCode breaker;
return breaker;
}

int main()
{
int someValue = 1;
globalBreaker().setReference(&someValue);
return 0;
}
// globalBreaker::breaker accesses main::someValue after it went out of
scope.
// This example should be harmless, but if a more complex type instead
of int
// is involved – and possibly the destructor of another function-local
// static overwrites the stack data from main – this pattern can lead to
a crash.
```

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.

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). Do not perform unmanaged resource acquisition in the constructor.

Definition:

Class member variables are initialized in the order in which they are declared in the class definition. The order in the constructor initializer list has no influence on initialization order and therefore may be misleading if it does not match the order of the declaration. Compilers often issue a warning if this rule is broken, but not always.

If a member variable is not explicitly initialized in the constructor initializer list the default constructor of that variable is called. Therefore, if a member variable is assigned to in the constructor function body, the member variable may get initialized unnecessarily with the default constructor.

in-class member initialization C++11

Since C++11, it is possible to initialize struct/class member variables in the class definition. Thus, if there is no explicit initialization of this member variable in the constructor, this assignment will be used for initialization.

```
class MyClass {
public:
MyClass() = default;
MyClass(int z) : a(z) {}

int data() const {
return a;
}
```

If you do not declare any constructors yourself then the compiler will generate a default constructor for you, which may leave some fields uninitialized or initialized to inappropriate values.

```
private:
int  a  =  1234;    //  in-class
initialization
};

int main() {
MyClass firstClass;
MyClass secondClass{5678};

std::cout << firstClass.data()
<< ' ' << secondClass.data();
return 0;
}
```

The program above prints "1234 5678".

Examples:

In-class member initialization:

```
// MyClass.h
class MyClass {
// ...
private:
int mValue = 1234;
};
```

Initialization list:

```
// MyClass.h
class MyClass : public MyBase  {
// ...
private:
int mValue;
std::vector mVector;
};
// MyClass.cxx
MyClass::MyClass()
: MyBase(),
mValue(0),
mVector()
{}
```

See an example of initialization via `std::initializer_list` in Brace Initialization.

Decision:

Member variables should be declared and initialized in the same order.

Use in-class member initialization for simple initializations, especially when a member variable must be initialized the same way in more than one constructor.

If your class defines member variables that aren't initialized in-class, and if it has no other constructors, you must define a default constructor (one that takes no arguments). It should preferably initialize the object in such a way that its internal state is consistent and valid.

If your class inherits from an existing class but you add no new member variables, you are not required to have a default constructor.

Avoid unmanaged resource acquisition in constructors, since the destructor of the class will not be called if an exception is thrown from the constructor. Therefore, always use C++ "resource acquisition is initialization" (RAII) idiom to ensure resources are properly freed.

▽ **Extra details and exceptions to the rules:**

An example demonstrating how to avoid unmanaged resource acquisition in constructor. Instead of

```
// MyClass.h
```

```
class MyClass {
public:
MyClass();
~MyClass();
private:
Something* mSomething;  // raw pointer
};
// MyClass.cxx
MyClass::MyClass()
: mSomething(new Something())
{}
MyClass::~MyClass()
{
delete mSomething;       // mSomething has to be deleted with its owner
}
```

do

```
// MyClass.h
class MyClass {
public:
MyClass();
~MyClass() = delete;   // mSomething is deleted automatically
private:
std::unique_ptr<Something> mSomething; // smart pointer
};
// MyClass.cxx
MyClass::MyClass()
: mSomething(new Something())
{}
```

Virtual functions in constructors and destructors

Do not call virtual functions in constructors and destructors.

Inside constructors and destructors virtual function do not behave "virtually". If the work calls virtual functions, these calls will not get dispatched to the subclass implementations. Calls to an unimplemented pure virtual function result in undefined behavior.

Calling a virtual function non-virtually is fine:

```
class MyClass {
public:
MyClass() { doSomething(); }    // Bad
virtual void doSomething();
};
```

```
class MyClass {
public:
MyClass() { MyClass::doSomething(); }    // Good
virtual void doSomething();
};
```

Decision:

Constructors should never call virtual functions.

Explicit Constructors

Use the C++ keyword `explicit` for constructors with one argument.

Definition:

Normally, if a constructor takes one argument, it can be used as a conversion. For instance, if you define

```
class Foo {
public:
Foo(const std::string &name);
};
```

and then pass a string to a function that expects a `Foo`, the constructor will be called to convert the string into a `Foo` and will pass the `Foo` to your function for you. Declaring a constructor `explicit` prevents an implicit conversion.

```
class MyClass {
public:
explicit MyClass(int number); //allocate number bytes
explicit  MyClass(const  std::string  &name);  //  initialize  with  string
name
};
```

Decision:

We require all single argument constructors to be explicit.

Exceptions are copy constructors and classes that are intended to be transparent wrappers around other classes.

Finally, constructors that take only an `initializer_list` may be non-explicit. This is to permit construction of your type using the assignment form for brace init lists (i.e. `MyType myObject = {1, 2}`).

Copy (and Move)

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

Definition:

The copy constructor and copy assignment operator are used to create copies of objects. The move constructor and move assignment operator are used to move (semantically) objects.

The copy and move constructors are implicitly invoked by the compiler or generic containers in some situations, e.g. passing objects by value or from `std::vector`.

Decision:

Classes that use value semantics normally need to be copyable. If they are not copyable (e.g. unique resource ownership) they should be movable. If the copy constructor is trivial (which

move semantics C++11

Move semantics were introduced with C++11. In essence, a move is just a copy that can be optimized from the knowledge that the source object is at the end of its life (such objects bind to rvalue references). Thus, a move of a `std::vector` does not need to copy all data, but only the pointer to the data. Additionally, the source object must be told that it does not own the data normally implies an empty destructor) it can be useful for performance reasons to use the key word `default`:

```
ClassName(const ClassName &other) = default;
```

instead of

```
ClassName(const ClassName &other)
: data(other.data) {}
```

The former can be optimized much better by the compiler.

anymore, to inhibit the `free` from the destructor. In most cases the move constructor/assignment operator therefore modifies the source object (e.g. setting the data pointer to `nullptr`).

Polymorphic class design implies pointer semantics. These classes should have their copy constructors disabled via `delete`:

```
ClassName(const ClassName &) = delete;
ClassName &operator=(const ClassName &) = delete;
```

default and delete C++11

Since C++11 it is possible to `delete`/`default` copy/move constructors and copy/move assignment operators.

If your polymorphic class needs to be copyable, use a virtual `clone()` method. This way copying can be implemented without slicing and be used more naturally for pointers:

```
void someFunction(SomeInterface *object)
{
SomeInterface *objectCopy = object->clone();
...
}
```

since C++11

Delegating and inheriting constructors

Use delegating and inheriting constructors when they reduce code duplication.

Definition:

Delegating and inheriting constructors are two different features, both introduced in C++11, for reducing code duplication in constructors. Delegating constructors allow the constructor to forward work to another constructor of the same class, using a special variant of the initialization list syntax. For example:

```
Foo::Foo(const string& name) : mName(name) {
...
}

Foo::Foo() : Foo("example") { }
```

A subclass, per default, inherits all functions of the base class. This is not the case for constructors. Since C++11 it is possible to explicitly inherit the constructors of a base class. This can be a significant simplification for subclasses that don't need custom constructor logic.

```
class Base {
public:
Base();
explicit Base(int number);
explicit Base(const string& name);
...
};

class Derived : public Base {
public:
using Base::Base;  // Base's constructors are redeclared here.
};
```

This is especially useful when `Derived`'s constructors don't have to do anything more than calling `Base`'s constructors.

Decision:

Use delegating and inheriting constructors when they reduce code duplication.
Be cautious about inheriting constructors when your derived class has new member variables and use in-class member initialization for the derived class's member variables.

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++. We add our own semantic meanings to each keyword, so you should use the appropriate keyword for the data-type you're defining.

`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.

You can use `struct` instead of `class` for functors and traits.

Note that member variables in structs and classes have different naming rules.

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.

In polymorphic design a special care is needed in implementing base class destructors. If deletion through a pointer to a base `Base` should be allowed, then the `Base` destructor must be public and virtual. Otherwise, it should be protected and can be non-virtual.

Decision:

Always write a destructor for a base class, because the implicitly generated one is public and nonvirtual.

▽ **Extra details and exceptions to the rules:**

In some class designs the destructors (of all classes in the inheritance tree) do nothing (implying that the classes and their members never allocate any resources). Typically, such designs do not have any virtual functions at all, and the virtual destructor would be the only reason for the existence of a vtable. Then a virtual destructor may be unnecessary and may be omitted.

Inheritance

When using inheritance, make it `public` and declare overriden 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.

Definition:

When a sub-class inherits from a base class, it includes the definitions of all the data and operations that the parent base class defines. In practice, inheritance is used in two major ways in C++: implementation inheritance, in which actual code is inherited by the child, and interface inheritance, in which only method names are inherited.

Pros:

Implementation inheritance reduces code size by re-using the base class code as it specializes an existing type. Because inheritance is a compile-time declaration, you and the compiler can understand the operation and detect errors. Interface inheritance can be used to programmatically enforce that a class expose a particular API. Again, the compiler can detect errors, in this case, when a class does not define a necessary method of the API.

override and final C++11

Since C++11 it is possible to mark virtual functions as overriding a virtual function from the base class. This is useful to state the intent and get a compile error if this intent is not fulfilled for some reason (e.g. typo in the function name, mismatching function signature, virtual keyword forgotten in the base class).

The `final` keyword tells the compiler that subclasses may not override the virtual function anymore. This is a special case, but useful to limit abuse of your classes by users.

Cons:

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. The base class may also define some data members, so that specifies physical layout of the base class.

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

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.

Definition:

Multiple inheritance allows a sub-class to have more than one base class. However this functionality can bring to the so-called Diamond problem unless base classes are pure interfaces.

Decision:

Multiple inheritance is allowed only when all superclasses, with the possible exception of the first one, are pure interfaces.

Interfaces

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

Definition:

A class is a pure interface if it meets the following requirements:

- It has only public pure virtual ("`= 0`") methods and static methods (see Destructors).
- It does not have data members.
- It does not have any constructors defined. If a constructor is provided, it must take no arguments and it must be protected.
- If it is a subclass, it may only be derived from classes that satisfy these conditions.

Decision:

When writing a pure interface, apply the corresponding naming rule and make sure there is no implementation in it. Make sure not to add implementation to an existing pure interface.

Operator Overloading

When overloading operators keep the same semantics.

Definition:

Operator overloading is a specific case of function overloading in which some or all operators like +, = or == have different behaviors depending on the types of their arguments. It can easily be emulated using function calls.

For example: `a << 1;` shifts the bits of the variable left by one bit if a is an integer, but if a is an output stream instead this will write "1" to it.

Decision:

The semantics of the operator overloading should be kept the same. Because operator overloading allows the programmer to change the usual semantics of an operator, it should be used with care.

Access Control

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

Information hiding protects the code from uncontrollable modifying state of your object by clients and it also help to minimize dependencies between calling and called codes.

A class consisting mostly of gets/sets is probably poorly designed. Consider providing an abstraction or changing it in `struct`.

Decision:

Make data members `private`, except in `structs`. If there is no better way how to hide the class internals, provide the access through protected or public accessor and, if really needed, modifier functions.

See also Inheritance, Structs vs. Classes and Function Names.

Others

Exceptions

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

Exceptions should be used for error handling.
Exception classes typically derive from `std::exception` (or one of its subclasses like `std::runtime_error`).
Exceptions should be scoped inside the class that throws them.
By default, catch exceptions by reference.

```
int computePedestals()
{
...
if (somethingWrong) {
throw BadComputation();
}
...
}
...
try {
computePedestals();
}
catch (BadComputation& e) {  // catch exception by reference
// code that handles error
...
}
```

```
int computePedestals()
{
...
if (somethingWrong) {
return -1;
}
...
}
...
if (computePedestals() == -1) {
// code that handles error
...
}
```

Use of const

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

Definition:

Variables and parameters can be declared as `const` to indicate that the variables are logically immutable. (Because of `const_cast` and `mutable` member variables, and global variables, `const` is no hard guarantee for immutability.) Member functions can be declared `const` to allow calls with `const this` pointer. Note that overloading member functions on `const` is possible.

Decision:

`const` variables, data members, methods and arguments add a level of compile-time type checking; it is better to detect errors as soon as possible. Therefore we strongly recommend that you use `const` whenever it makes sense to do so.

Use `const`:

- for an argument, if the function does not modify it when passed by reference or by pointer.
- For accessors.
- For methods, if they:
  - do not modify any non-local data;
  - can be safely (no data race) called from multiple threads;
  - do not call any non-`const` methods;
  - do not return a non-`const` pointer or non-`const` reference to a data member.
- For data members, whenever they do not need to be modified after construction.

C++11

The Standard Library […] in simple words says that it expects operations on const objects to be thread-safe. This means that the Standard Library won't introduce a data race as long as operations on const objects of your own types either

- Consist entirely of reads –that is, there are no writes–; or
- Internally synchronizes writes.

[Source: Stack Overflow]

This is a great example of how C++11 is a simpler language: we can stop the Cold War-era waffling about subtleties about what 20th-century C++ const means, and proudly declare modern C++ const has the simple and natural and “obvious” meaning that most people expected all along anyway.

[…] Bjarne Stroustrup writes: “I do point out that const means immutable and absence of race conditions in the last Tour chapter. […]”

[Source: isocpp.org]

▽ **Extra details and exceptions to the rules:**

`mutable` can be used to make objects that are already threadsafe (such as `std::mutex`) mutable in `const` methods. Thus, it is possible to make `const` methods thread-safe, through internal synchronization.

since C++11

Use of constexpr

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

Definition:

Some variables can be declared `constexpr` to indicate the variables are true constants, i.e. fixed at compilation/link time. Some functions and constructors can be declared `constexpr` which enables them to be used in defining a `constexpr` variable.

Pros:

Use of `constexpr` enables definition of constants with floating-point expressions rather than just literals; definition of constants of user-defined types; and definition of constants with function calls.

Cons:

Prematurely marking something as constexpr may cause migration problems if later on it has to be downgraded. Current restrictions on what is allowed in constexpr functions and constructors may invite obscure workarounds in these definitions.

Decision:

`constexpr` definitions enable a more robust specification of the constant parts of an interface. Use `constexpr` to specify true constants and the functions that support their definitions. Avoid complexifying function definitions to enable their use with `constexpr`. Do not use `constexpr` to force inlining.

▽ **Extra details and exceptions to the rules:**

While `constexpr` variables are constant expressions, they can still have an address. Thus, using a `constexpr` variable as argument for a const-ref function parameter requires the `constexpr` variable to have a symbol. Consider the following header file:

```
constexpr int GlobalScopeValue = 0;

namespace Namespace {
constexpr int ScopeValue = 1;
}

struct Struct {
static constexpr int ScopeValue = 1;
};

template<typename T> struct TemplateStruct {
static constexpr int ScopeValue = 1;
};

void function(const int &value);
```

And the following test code:

```
function(GlobalScopeValue);      // fine
function(Namespace::ScopeValue); // fine
function(Struct::ScopeValue);    // link error
function(TemplateStruct<int>::ScopeValue); // link error
```

To provide the missing symbols you have to add

```
template<typename T> constexpr int TemplateStruct<T>::ScopeValue;
```

to the header file and

```
constexpr int Struct::ScopeValue;
```

to one `.cxx` file.

Smart Pointers

`std::unique_ptr` and `std::shared_ptr` should be used consistently instead of non-owning raw pointers. Never use owning raw pointers, and thus never use `delete`. The use of raw pointers may need an explanation in form of a comment.

Definition:

Smart pointers are objects that act like pointers, but automate ownership. There are two main semantics for ownership: unique and shared ownership.

Unique ownership ensures that there can be only one smart pointer to the object. If that smart pointer goes out of scope it will free the pointee.

Shared ownership allows to have multiple pointers to an object without deciding who is the exclusive owner. Thus the owners can be freed in any order and the pointer will stay valid until the last one is freed, in which case the pointee is also freed. Note that `shared_ptr<T>` is thread-safe and thus enables sharing ownership over multiple threads.

smart pointers C++11

Smart pointers have existed long before C++11. But since C++11 the standard library contains the classes `unique_ptr<T>`, `shared_ptr<T>`, and `weak_ptr<T>`. Also, the standard library provides `make_shared<T>` and starting with C++14 also `make_unique<T>`.

Example:

```
{
std::shared_ptr<int> first;
{
std::unique_ptr<int> second(new int);
auto third = std::make_shared<int>();
first = third;
}
// only second is freed automatically here
}
// first and third are automatically freed here
```

When exiting the inner scope, only `second` is freed automatically, because the last reference to it went out of scope. But even though `third` went out of scope here, no free occurred because `first` still has a reference. Only when `first` went out of scope and as it is the last reference, `third` is automatically freed.

A weak pointer (`weak_ptr<T>`) can be used to break cyclic ownership.

Pros:

Smart pointers are extremely useful for preventing memory leaks, and are essential for writing exception-safe code. They also formalize and document the ownership of dynamically allocated memory.

Cons:

Smart pointers enable sharing or transfer of ownership and can thus act as a tempting alternative to careful design of ownership semantics. This could lead to confusing code and even bugs in which memory is never deleted.

Decision:

`std::unique_ptr`
: Straightforward and risk-free. This should be your default for any pointer.

`std::shared_ptr`
: If you really need to share ownership, use a `shared_ptr`. You should avoid designs that require shared ownership, though, as it incurs an overhead. `unique_ptr` on the other hand is without overhead.

▽ **Extra details and exceptions to the rules:**

The following code appears to require shared ownership:

```
void function(shared_ptr<int> value)
{
*value = 0;
}

void otherFunction()
{
auto ptr = make_shared<int>();
function(ptr);
...
}
```

Instead the code really has unique ownership in `otherFunction()`. There is no reason that `function()` and `otherFunction()` need to share ownership:

```
void function(int *value) // caller retains ownership
{
*value = 0;
}

void otherFunction()
{
unique_ptr<int> ptr{new int};
function(ptr.get());
...
}
```

There would be a reason to share ownership if the owner of the object is a different thread.

Magic numbers

Avoid magic numbers.

Avoid spelling literal constants like `42` or `3.141592` in code. Use symbolic names and expressions instead. Names add information and introduce a single point of maintenance.

Example of constants at namespace level:

```
static constexpr double Millimeter  = 1.;
static constexpr double Centimeter  = 10.*Millimeter;
```

Example of class-specific constants:

```
// File Widget.h
class Widget {
private:
static const int sDefaultWidth;                    // value provided in
definition
static constexpr int DefaultHeight = 600; // value provided in
declaration
};
```

```
// File Widget.cxx
const int Widget::sDefaultWidth = 800; // value provided in definition
constexpr int Widget::DefaultHeight;    // definition required only if
```

```
reference/pointer to
// DefaultHeight is needed
```

Preprocessor Macros

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

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.

The following usage pattern will avoid many problems with macros; if you use macros, follow it whenever possible:

- Don't define macros in a `.h` file.
- Define macros (via `#define`) right before you use them, and undefine them (via `#undef`) right after.
- Do not just undefine an existing macro (via `#undef`) before replacing it with your own; instead, pick a name that's likely to be unique.
- Try not to use macros that expand to unbalanced C++ constructs, or at least document that behavior well.
- Prefer not using `##` to generate function/class/variable names.
- Follow the naming convention as described here.

Write Short Functions

Prefer small and focused functions.

Long functions are hard to debug and makes readability difficult. Short functions allow code reuse. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.
Giving the function a name that describes what it does might help splitting it into smaller pieces. Functions should represent logical grouping, therefore it should be easy to assign them meaningful names.
Please note that nesting is not the same as splitting long functions into short ones. In addition, it does not improve readability and ease of debug.

Run-Time Type Information (RTTI)

Avoid using Run Time Type Information (RTTI).

Definition:

RTTI allows a programmer to query the C++ class of an object at run time. This is done by use of `typeid` or `dynamic_cast`.

Cons:

Querying the type of an object at run-time frequently means a design problem. Needing to know the type of an object at runtime is often an indication that the design of your class hierarchy is flawed.

Undisciplined use of RTTI makes code hard to maintain. It can lead to type-based decision trees or switch statements scattered throughout the code, all of which must be examined when making further changes.

Decision trees based on type are a strong indication that your code is on the wrong track.

```
if (typeid(*data) == typeid(Data1)) {
...
} else if (typeid(*data) == typeid(Data2)) {
...
} else if (typeid(*data) == typeid(Data3)) {
...
```

Code such as this usually breaks when additional subclasses are added to the class hierarchy. Moreover, when properties of a subclass change, it is difficult to find and modify all the affected code segments.

Pros:

The standard alternatives to RTTI (described below) require modification or redesign of the class hierarchy in question. Sometimes such modifications are infeasible or undesirable, particularly in widely-used or mature code.

RTTI can be useful in some unit tests. For example, it is useful in tests of factory classes where the test has to verify that a newly created object has the expected dynamic type. It is also useful in managing the relationship between objects and their mocks.

```
// Example of a unit test
Geo::Factory geoFactory;
Geo::Object* circle = geoFactory.CreateCircle();
if ( ! dynamic_cast<Geo::Circle>(circle) ) {
std::cerr << "Unit test failed."  << std::endl;
}
```

Decision:

RTTI has legitimate uses but is prone to abuse, so you must be careful when using it. You may use it freely in unittests, but avoid it when possible in other code. In particular, think twice before using RTTI in new code. If you find yourself needing to write code that behaves differently based on the class of an object, consider one of the following alternatives to querying the type:

- Virtual methods are the preferred way of executing different code paths depending on a specific subclass type. This puts the work within the object itself.
- If the work belongs outside the object and instead in some processing code, consider a double-dispatch solution, such as the Visitor design pattern. This allows a facility outside the object itself to determine the type of class using the built-in type system.

When the logic of a program guarantees that a given instance of a base class is in fact an instance of a particular derived class, then use of a `dynamic_cast` or `static_cast` as an alternative may be also justified in such situations.

▽ **Extra details and exceptions to the rules:**

An example of code based on `dynamic_cast`:

```
void foo(Bar* bar) {
// ... some code where x, y, z are defined ...
// ...
if (Data1 data1 = dynamic_cast<Data1*>(bar)) {
doSomething(data1, x, y);
}
```

```
else if (Data2 data2 = dynamic_cast<Data2*>(bar)) {
doSomething(data2, z)
}
```

which can be defined using the Visitor pattern:

```
void foo(Bar* bar) {
// ... some code where x, y, z are defined ...
// ...
DoSomethingVisitor visitor(x, y, z);
bar.accept(visitor);
}
```

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.

Decision:

- Try to avoid casts. The need for casts may be a hint that too much type information was lost somewhere.
- Use `static_cast` to explicitly convert values between different types. `static_cast`s are useful for up-casting pointers in an inheritance hierarchy.
- Avoid `const_cast`. (Possibly use `mutable` member variables instead.) `const_cast` may be used to adapt to const-incorrect interfaces that you cannot (get) fix(ed).
- `reinterpret_cast`s are powerful but dangerous. Rather try to avoid them. Code that requires a `reinterpret_cast` should document the aliasing implications. (reinterpret_cast on cppreference.com)

See the RTTI section for guidance on the use of `dynamic_cast`.

▽ **Extra details and exceptions to the rules:**

For the dangers of `reinterpret_cast` consider:

```
std::uint32_t fun()
{
std::uint32_t binary = 0;
reinterpret_cast<float &>(binary) = 1.f;
return binary; // the return value is undefined, according to the C++
standard
}
```

The following is better, but still undefined behavior according to the type aliasing rules:

```
std::uint32_t fun()
{
float value = 1.f;
return reinterpret_cast<std::uint32_t &>(value); // this returns
0x3f800000 on x86
}
```

In case of doubt, prefer not to use `reinterpret_cast` in order to avoid mistakes.

Variable-Length Arrays and alloca()

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

Pros:

Stack-allocated objects avoid the overhead of heap allocation. Variable-length arrays and `alloca` allow variably-sized stack allocations, whereas all other stack allocations in C++ only allow fixed-size objects on the stack.

Cons:

Variable-length arrays are part of C but not Standard C++. `alloca` is part of POSIX, but not part of Standard C++.

Decision:

Use STL containers instead. If you really need to improve the performance consider using a custom allocator for the containers.

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`.

If not conditional on an enumerated value, switch statements should always have a `default` case (in the case of an enumerated value, the compiler will warn you if any values are not handled). If the default case should never execute, simply `assert`:

```
switch (value) {
case 0: {  // 2 space indent
...      // 4 space indent
break;
}
case 1: {
...
break;
}
default: {
assert(false);
}
}
```

Empty loop bodies should use `{}` or `continue`, but not a single semicolon.

```
while (condition) {
// Repeat test until it returns false.
}
for (int i = 0; i < someNumber; ++i) {}  // Good – empty body.
while (condition) continue;  // Good – continue indicates no logic.
```

```
while (condition);  // Bad – looks like part of do/while loop.
```

Integer Types

Per default, use `int` if you need an integer type. Prefer signed integers over unsigned integers and thus `std::int64_t` over `unsigned int` if you need more bits.

The C++ standard only loosely specifies the sizes of its built-in integer types.

If you need something other than `int`, consider one of the integer types in `<cstdint>`.

On Unsigned Integers

Using unsigned types to represent numbers that are never negative may be a source of problems as demonstrated here:

```
for (unsigned int i = foo.getLength() - 1; i >= 0; --i) ...
```

This code will never terminate! A compiler might notice the issue and warn you, but do not count on it. Equally bad bugs can occur when comparing signed and unsigned variables.

To avoid such situation we recommend to consistently use `int`:

```
for (int i = static_cast<int>(foo.getLength()) - 1; i >= 0; --i) ...
```

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 .

Below we give a list (incomplete) of possible portability issues:

- `printf()` specifiers for some types are not cleanly portable between 32-bit and 64-bit systems.
- Remember that `sizeof(void *)` != `sizeof(int)`. Use `intptr_t` if you need a pointer-sized integer.
- You may need to be careful with structure alignments, particularly for structures being stored on disk.
- The data memory representation is computer specific and not defined by C++. The terms endian and endianness, refer to how bytes of a data word are ordered within memory. Big endian store bytes from the highest to the lowest, Little endian from the lowest to the highest.

0 and nullptr

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

`nullptr` is a pointer literal of type `std::nullptr_t`. On the other hand, `NULL` is a macro equivalent the integer `0`. Using `NULL` could bring to unexpected problems. For example imagine you have the following two function declarations:

```
void function(int number);
void function(char *name);

function( NULL );
```

Because `NULL` is `0`, and `0` is an integer, the first version of func will be called instead. In

C++11, `nullptr` is a new keyword that can (and should!) be used to represent `NULL` pointers.

sizeof

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

Use `sizeof(varname)` when you take the size of a particular variable. `sizeof(varname)` will update appropriately if someone changes the variable type either now or later. You may use `sizeof(type)` for code unrelated to any particular variable, such as code that manages an external or internal data format where a variable of an appropriate C++ type is not convenient.

```
Struct data;
memset(&data, 0, sizeof(data));
```

```
memset(&data, 0, sizeof(Struct));
```

```
if (rawSize < sizeof(int)) {
logMessage << "compressed record not big enough for count: " << rawSize;
}
```

since C++11

auto

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.

Definition:

In C++11, a variable whose type is given as `auto` will be given a type that matches that of the expression used to initialize it. You can use `auto` either to initialize a variable by copying, or to bind a reference.

```
vector<string> names;
...
auto name1 = names[0];  // Makes a copy of names[0].
const auto& name2 = names[0];  // name2 is a reference to names[0].
```

Pros:

C++ type names can sometimes be long and cumbersome, especially when they involve templates or namespaces. In a statement like

```
sparse_hash_map<string, int>::iterator iter = myMap.find(val);
```

the return type is hard to read, and obscures the primary purpose of the statement. Changing it to

```
auto iter = myMap.find(val);
```

makes it more readable.

Without `auto` we are sometimes forced to write a type name twice in the same expression, adding no value for the reader, as in

```
diagnostics::ErrorStatus* status = new diagnostics::ErrorStatus("xyz");
```

Using `auto` makes it easier to use intermediate variables when appropriate, by reducing the burden of writing their types explicitly.

```
auto status = new diagnostics::ErrorStatus("xyz");
```

Cons:

Sometimes code is clearer when types are manifest, especially when the initialization of a variable depends on functions/variables that were declared far away. In an expression like

```
auto i = xValue.Lookup(key);
```

it may not be obvious what `i`'s type is, if `x` was declared hundreds of lines earlier.

Programmers have to understand the difference between `auto` and `const auto&` or they'll get copies when they didn't mean to.

The interaction between `auto` and C++11 brace-initialization can be confusing. The declarations

```
auto xValue(3);  // Note: parentheses.
auto yValue{3};  // Note: curly braces.
```

mean different things — `xValue` is an `int`, while `yValue` is an `initializer_list`. The same applies to other normally-invisible proxy types.

If an `auto` variable is used as part of an interface, e.g. as a constant in a header, then a programmer might change its type while only intending to change its value, leading to a more radical API change than intended.

Decision:

`auto` is permitted for local variables only. Do not use `auto` for file-scope or namespace-scope variables, or for class members. Never assign a braced initializer list to an `auto`-typed variable.

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.

To modify code that was written to specifications other than those presented by this guide, it may be necessary to deviate from these rules in order to stay consistent with the local conventions in that code. In case of doubt the original author or the person currently responsible for the code should be consulted. Remember that *consistency* also includes local consistency.

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]