See also: <http://www.boost.org/doc/libs/1_64_0/doc/html/index.html>

TODO: relocate the above instances.

HEADER FILES

Self-contained Headers

Header files should be self-contained (compile on their own) and end in .h. Non-header files that are meant for inclusion should end in .inc and be used sparingly.

All header files should be self-contained. Users and refactoring tools should not have to adhere to special conditions to include the header. Specifically, a header should have header guards and include all other headers it needs.

Prefer placing the definitions for template and inline functions in the same file as their declarations. The definitions of these constructs must be included into every .cc file that uses them, or the program may fail to link in some build configurations. If declarations and definitions are in different files, including the former should transitively include the latter. Do not move these definitions to separately included header files (-inl.h); this practice was common in the past, but is no longer allowed.

As an exception, a template that is explicitly instantiated for all relevant sets of template arguments, or that is a private implementation detail of a class, is allowed to be defined in the one and only .cc file that instantiates the template.

There are rare cases where a file designed to be included is not self-contained. These are typically intended to be included at unusual locations, such as the middle of another file. They might not use header guards, and might not include their prerequisites. Name such files with the .inc extension. Use sparingly, and prefer self-contained headers when possible.

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/baz.h in project foo should have the following guard:

#ifndef FOO\_BAR\_BAZ\_H\_

#define FOO\_BAR\_BAZ\_H\_

...

#endif // FOO\_BAR\_BAZ\_H\_

Forward Declarations

Try to avoid forward declarations of entities defined in another project.

When using a function declared in a header file, always #include that header.

When using a class template, prefer to #include its header file.

Inline Functions

Define functions inline only when they are small, say, 10 lines or fewer.

You can declare functions in a way that allows the compiler to expand them inline rather than calling them through the usual function call mechanism.

Inlining a function can generate more efficient object code, as long as the inlined function is small. Feel free to inline accessors and mutators, and other short, performance-critical functions.

Overuse of inlining can actually make programs slower. Depending on a function's size, inlining it can cause the code size to increase or decrease. Inlining a very small accessor function will usually decrease code size while inlining a very large function can dramatically increase code size. On modern processors smaller code usually runs faster due to better use of the instruction cache.

A decent rule of thumb is to not inline a function if it is more than 10 lines long. Beware of destructors, which are often longer than they appear because of implicit member- and base-destructor calls!

Another useful rule of thumb: it's typically not cost effective to inline functions with loops or switch statements (unless, in the common case, the loop or switch statement is never executed).

It is important to know that functions are not always inlined even if they are declared as such; for example, virtual and recursive functions are not normally inlined. Usually recursive functions should not be inline. The main reason for making a virtual function inline is to place its definition in the class, either for convenience or to document its behavior, e.g., for accessors and mutators.

See also: <https://en.wikipedia.org/wiki/Mutator_method>

Names and Order of Includes

Use standard order for readability and to avoid hidden dependencies: Related header, C library, C++ library, other libraries' .h, your project's .h.

SCOPING

Namespaces

Unnamed Namespaces and Static Variables

All declarations can be given internal linkage by placing them in unnamed namespaces, and functions and variables can be given internal linkage by declaring them static. This means that anything you're declaring can't be accessed from another file. If a different file declares something with the same name, then the two entities are completely independent.

Use of internal linkage in .cc files is encouraged for all code that does not need to be referenced elsewhere. Do not use internal linkage in .h files.

Format unnamed namespaces like named namespaces. In the terminating comment, leave the namespace name empty:

namespace {

...

} // namespace

Also read: <https://stackoverflow.com/questions/1358400/what-is-external-linkage-and-internal-linkage>

<https://en.wikipedia.org/wiki/Translation_unit_(programming)>

<https://en.wikipedia.org/wiki/Linkage_(software)>

<https://en.wikipedia.org/wiki/Object_file>

Nonmember, Static Member, and Global Functions

Prefer placing nonmember functions in a namespace; use completely global functions rarely. Prefer grouping functions with a namespace instead of using a class as if it were a namespace. Static methods of a class should generally be closely related to instances of the class or the class's static data.

Sometimes it is useful to define a function not bound to a class instance. Such a function can be either a static member or a nonmember function. Nonmember functions should not depend on external variables, and should nearly always exist in a namespace. Rather than creating classes only to group static member functions which do not share static data, use namespaces instead.

See Also: <http://en.cppreference.com/w/cpp/language/static>

<http://en.cppreference.com/w/cpp/language/storage_duration>

<http://en.cppreference.com/w/cpp/language/class#Local_classes>

<http://en.cppreference.com/w/cpp/language/friend>

<http://en.cppreference.com/w/cpp/language/enum>

<https://stackoverflow.com/questions/19736281/what-are-the-differences-between-overriding-virtual-functions-and-hiding-non-vir>

<http://en.cppreference.com/w/cpp/thread/thread>

<http://en.cppreference.com/w/cpp/thread/lock_guard>

Local Variables

Place a function's variables in the narrowest scope possible, and initialize variables in the declaration.

C++ allows you to declare variables anywhere in a function. We encourage you to declare them in as local a scope as possible, and as close to the first use as possible. This makes it easier for the reader to find the declaration and see what type the variable is and what it was initialized to. In particular, initialization should be used instead of declaration and assignment.

Static and Global Variables

CLASSES

Doing Work in Constructors

Avoid virtual method calls in constructors, and avoid initialization that can fail if you can't signal an error.

It is possible to perform arbitrary initialization in the body of the constructor.

No need to worry about whether the class has been initialized or not.

Objects that are fully initialized by constructor call can be const and may also be easier to use with standard containers or algorithms.

If the work calls virtual functions, these calls will not get dispatched to the subclass implementations. Future modification to your class can quietly introduce this problem even if your class is not currently subclassed, causing much confusion.

There is no easy way for constructors to signal errors, short of crashing the program (not always appropriate) or using exceptions (which are forbidden).

If the work fails, we now have an object whose initialization code failed, so it may be an unusual state requiring a bool IsValid() state checking mechanism (or similar) which is easy to forget to call.

You cannot take the address of a constructor, so whatever work is done in the constructor cannot easily be handed off to, for example, another thread.

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

Implicit Conversions

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

See also: <http://en.cppreference.com/w/cpp/language/implicit_conversion>

Copyable and Movable Types