Quickstep Coding Standards

This page describes the current Quickstep coding guidelines. This information is targeted towards new and existing **contributors** on this project. All developers of Quickstep should subscribe to the **developer's list**, which is dev@quickstep.incubator.apache.org. To subscribe, send an empty email to dev-subscribe@quickstep.incubator.apache.org.

General Code Style

The Quickstep coding style is closely based on the Google C++ style guide, which you can read at https://google.github.io/styleguide/cppguide.html

The Google style guide is an excellent guide to writing C++ code that is both high-performance and highly readable and maintainable. We do depart from the style guide in a few areas, as listed below:

1. **Code Documentation:** We require documenting the logic in your `.cpp` files (in addition to doxygen comments in the `.hpp` files) so that it is easy for readers to understand the logic that you have implemented. This approach helps maintain the code in the long run, making it easier for other developers to work with your code (if done right, it should also help you down the line if you have to go back to your code at a later time). This approach also makes it easier for another developer to examine your code when merging your pull request as s /he can more easily match your intent (declared in your comments) with your actual action (encapsulated in the code).
2. **File naming**: Header files have the suffix `.hpp`, source files have the suffix `.cpp`. Files should be named after the main class defined in the file, in `CamelCase`, exactly as the class name is in the source. For example, if you write a class named `MyNewClass`, its sources would be `MyNewClass.hpp` and `MyNewClass.cpp`.
3. **Line length**: lines may be up to 120 characters long (up from Google's limit of 80). However, it is preferred to keep lines below 80 characters unless readability/aesthetics would be negatively affected by doing so. It is *strongly* preferred to keep lines below 100 characters unless you can't possibly avoid it. Comments should generally be kept to a limit of 80 character lines. See the Google style guide for rules and advice about how to break up a statement across multiple lines and how to indent continuations (continuations of a line must be indented at least 4 spaces, or they can be indented further to "line up" with related things vertically). Our preference is to put each parameter to a function call on its own line when we have to break up a call statement, and to align "groups" of related things vertically (e.g. parameters separated by commas, or parts of an expression with the same level of precedence).
4. **Method naming**: Instance methods of classes are should have camelcase names starting with a lowercase letter `likeThis()`. Static methods of classes, and constructors, should have capitalized camelcase names `LikeThis()`.
5. **Exceptions**: we do use exceptions to report errors in some cases, but we are trying to cut down on that. If you detect an error which is clearly the result of the caller misusing the API, you should intentionally crash the program with the `LOG(FATAL)` macro in `glog/logging.h`. Where possible, you should report other errors with return values that should be checked by the caller, not exceptions (be sure to note the requirement to check return values in your doxygen comments for the method in question). Exceptions are generally reserved for externally-generated weirdness (like a file loaded from disk which appears to be corrupted). If you really think you need to add a new exception, you should run it by other Quickstep core contributors, and make it inherit from `std::exception`.
6. **Pointer and reference sigils**: the Google style guide does not have an exact recommendation for where to place sigil characters for pointers (*) and references (&), but merely says to be consistent across a project. In Quickstep, we place the sigil next to the variable name rather than the type name, like so: `void *ptr;`. For functions which return a pointer or reference, we place the sigil next to the type name, like so: ` void* someMethod(); `.

Documentation And Comments

All public methods of externally-visible classes you write should have full Doxygen documentation. Private methods and data members can be commented as you see fit, but don't necessarily need comments if it is obvious from their name and context what they do. The same policy applies to private helper classes and functions nested inside other classes or in anonymous namespaces in implementation (.cpp) files: comment them as you see fit for clarity, but full doxygen-style comments aren't necessary. Feel free to intersperse comments with implementation code to clarify what the code is doing, but remember that comments are no substitute for writing clear and understandable code.

Use `TODO` and `FIXME` comments for code you intend to revisit later. `TODO` is for features or improvements you intend to add, while `FIXME` is for known or suspected bugs or unhandled edge cases that you need to fix. Both kinds of comment should include your username in parentheses, e.g. `TODO(chasseur)`.

Comment lines should be kept to a limit of 80 characters for readability (although you may make an exception for a short comment following some code on the same line).

An example doxygen comment is given below:

```
/**
 * @brief This line is to give a quick overview of the add_five function.
 * @note Sometimes functions are so simple (for example getters) that they only
 * need a '@brief' or '@return' comment to explain them.
 * @warning This is an optional warning tag which might state side effects of add_five.
 * @details Details gives a more in-depth explaination of the commented function.
 * Note that to give a detailed explaination, you can also omit the
 * '@details' tag and simply write on a new line line the following:
 * Subsequent lines which go into detail about the add_five functions should
 * follow the brief explaination and do not need to be indented to match the
 * brief text. You can also use the '@details' tag like above.
 *
 * @param arg This line explains what arg is, and if it is a pointer, ownership
 * transfer details. Notive how this comment text is indented with
 * the start of this comment's text and not the word 'arg'.
 * @return Any function with a return should state what is return. Also, transfer
 * of pointer ownership should also be noted.
 **/
int add_five(char *arg) {
 return static_cast<int>(*arg) + 5;
}
```

Unit Tests

Unit tests are great: they expose bugs, make sure changes don't break expectations, and can even help in design by forcing you to think clearly and precisely about the interfaces and expected behavior of your code. New classes and features in Quickstep should generally have unit tests, and the test coverage should be as complete as possible (i.e. test out all public methods with inputs that will go through all the different paths in your code, including error cases). We use the googletest testing framework (formerly known as gunit or gtest) https://code.google.com/p/googletest/ . Googletest is a great xUnit-style testing framework for C++ which automates a lot of the hard and boring stuff about testing.

Sometimes, you want to measure the performance impact of a change, and not just verify correctness. We use the Google micro-benchmarking library https://github.com/google/benchmark to implement small micro-benchmarks for such purposes. Don't forget to run the benchmarks in a RELEASE build.

Portability and Compatibility

We want Quickstep to run on as many systems as possible, with as few external dependencies as possible. That means writing standard C++ without reliance on platform-specific features or extensions. Use the standard library liberally, but avoid features which aren't portable (like POSIX C features which aren't part of the ANSI standard). Your code should compile with -Wall turned on with no warnings with recent versions of GCC and clang (we also try to support Microsoft Visual C++ and Intel C++, and have thus far succeeded).

In some cases where you *really* need OS-specific support for a feature, write CMake platform checks for the feature (don't simply assume it will be present). There should either be multiple platform-specific implementations of the feature for common platforms (e.g. `FileManagerLocal` has a version for POSIX and a version for Windows), or a fallback for the feature that is portable vanilla C++ (e.g. memory allocation in `StorageManager` uses POSIX `mmap()` where available, but has a fallback using regular `malloc()` where it isn't).

We are very selective about adding external library dependencies to Quickstep. Currently there are 4 hard dependencies on external libraries, and 2 soft dependencies on external libraries that are used for optional features. The hard dependencies are on **farmhash** (a fast and high-quality suite of hash functions for strings and other byte arrays), **glog** (a flexible logging library from Google), **googletest** (our unit-testing framework), and **protocol buffers** (a portable and high performance data serialization framework). The soft dependencies are on **linenoise**, an interactive command-line editing library (like GNU readline, but much smaller & simpler and released under a more permissive BSD license), and on **tcmalloc**, a very fast memory allocator which is especially well-suited to multi-threaded programs (part of Google perftools).

Adding a new library dependency is a big deal which affects the portability of Quickstep, and is something that should always be discussed on the developer's list before it happens. We prefer to use libraries with BSD/MIT/X11/Apache-style licenses, which are very permissive in what they allow us to do with the code. We also prefer libraries which are lightweight and provide a limited, specific set of features that we need, as big libraries lead to long build times and overly-large statically linked executables. Finally, we want to use libraries which are highly portable across different systems so that Quickstep can be portable too (for example, all of our hard dependencies work fine with many different compilers across different flavors of UNIX and Windows). We also have a preference for libraries that can be built with CMake (either using CMake build files provided by the original project or written by us), since that makes it much easier to integrate with our build system.

C++11

We do use C++11 and require a C++11 compiler and standard library to build Quickstep. Be judicious about using C++11 features, though, and don't use them "just because" (for instance, don't use auto unless it's really needed in a template context, don't implement a move constructor for a class unless move semantics are actually needed, etc.). Feel free to use all the approved C++11 features listed in the Google style guide, plus the chrono library, and in circumstances where it's actually needed, move semantics. You should not use C++11 threads directly, but instead use

our own threads module (which uses a platform-appropriate choice of C++11 threads, POSIX threads, or Windows threads under the covers). If you want to use any other C++11 features, you should email the developer's list first so that the active developers all aware of the rationale and familiar enough with the features in question that the current developers won't be confused by your code.

Namespaces

All Quickstep code lives in the C++ namespace `quickstep`. When using the standard library, always use fully qualified names starting with `std::` in headers, and NEVER put a `using` statement in a header. `using` statements are fine in implementation (`.cpp`) files, but you should have separate using statements for each identifier you use (for example, write `using std::vector;` rather than `using namespace std;`). If you need to write helper functions or classes in an implementation file that should not be accessible to the outside world, it is a good idea to put them in an anonymous namespace in the `.cpp` file (an anonymous namespace is a namespace with no name `namespace { ... }` which is only visible inside the file in which it is defined).

You should always prefer to use the C++ version of C library headers, since the C++ version will define everything (except macros) in the `std` namespace and help avoid namespace pollution. For example, you should `#include <cstdlib>` rather than `#include <stdlib.h>`.

Quickstep is coded using C++-11. If you are new to C++, you will need to get familiar with the language. The language is powerful, but also vast. We try to use C++ “correctly” as per the guidelines at: https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md That page has a LOT of information in there, so it is best to browse it once and then use it as a reference to go to specific sections when you have questions.

A few other helpful items that you may want to pick up (pulled from an email to new members of the Quickstep group in early 2015):

#1: Move Semantics
— Start here: http://stackoverflow.com/questions/3106110/what-are-move-semantics/ and read the first reply.
— Don’t overlook the second reply at: http://stackoverflow.com/questions/3106110/what-are-move-semantics/11540204#11540204
— Another related intersting reading is at: https://en.wikipedia.org/wiki/Rule_of_three_%28C%2B%2B_programming%29

#2: Templates
— A gentle introduction is at: https://en.wikipedia.org/wiki/Template_%28C%2B%2B%29.
— Don’t miss the related topic of Variadic templates at: https://en.wikipedia.org/wiki/Variadic_template
— The more technical guide is at: http://en.cppreference.com/w/cpp/language/templates This is probably more digestible after the first two.

#3: Dependent Names
Ok: I only said two issues above, to not scare you too much, but you probably want to see this too:
— Sooner or later (probably sooner if you are in the guts of the execution engine code) you will hit the issue of dependent names. Here is a gentle starting point:
http://aszt.inf.elte.hu/~gsd/halado_cpp/ch06s03.html#Dependent-name (see the top part and read on if you are ambitious).
— Then see http://stackoverflow.com/questions/610245/where-and-why-do-i-have-to-put-the-template-and-typename-keywords
— The mother load is at http://en.cppreference.com/w/cpp/language/dependent_name.

WARNING: Templates are a powerful component in C++ and key way to do metaprogramming. But it takes a while to get comfortable with it. You can go overboard with it and make the code completely unreadable (even to you a few months after you have written it). So, use it carefully, and when you can, please leave ample comments in the code to help others — with templates you will be very unlikely to be too verbose in your comments.

Of course as you go through the code and see spots that could benefit from additional comments, please add them and open a PR. Comments-only PR on existing code are welcome!

Content by label

There is no content with the specified labels