C++ Style Guide

> *"Style, or simply readability, is what we call the conventions that govern our C++ code." – Google C++ Style Guide*
>
> *"Never offend people with style when you can offend them with substance." – Sam Brown*

Maintainers of this style guide: Lars Fröhlich (3857), Olaf Hensler (3372)

Motivation & Generalities

- **Code is read more often than it is written.**
  Code should be easy to read. Not only for colleagues that have to debug or maintain it when you are not around, but also for yourself – ever try to remember how that tricky algorithm works that you wrote three years ago? A clean, consistent style makes code more readable, maintainable, and debuggable. This style guide is a collection of a few core practices to help with this.
- **These are guidelines, not commandments.**
  You can deviate from the practices of this style guide whenever it seems like a good idea. But please, think about it carefully. It does not hurt to document your reasoning in the code as well.
- **Be consistent.**
  When editing someone else's code, you should follow the stylistic conventions used in that package, even if they do not agree with these guidelines. *Unless* you can agree with the original maintainer to retrofit the entire package, of course. You'll be the hero of the day.
- **Starting from scratch?**
  If you are writing a new piece of code, *please* follow these guidelines. It makes life easier for all of us.

Indentation & Formatting

- **Use spaces instead of tabs.**
  Tabs may expand to different widths depending on editor settings, making source code hard to read.
- **The maximum line length is 90 characters.**
  This is highly controversial and somewhat arbitrary, but consistency really helps all of us to avoid the constant resizing of our editor windows. Many people feel that the traditional limit of 80 characters is too tight, so we chose a little more. On the other hand, it should still be possible to fit two editor windows on a single screen with 90 characters per line.
- **Use 4 spaces per indentation level.**
  Consistency matters. Indentation by four spaces is already used in a sizable part of our code base. At the same time, it is compatible with old code using 8-character-wide tabs**.**
- **Let your editor do the work.**
  Configure your editor to use the settings above by default. For example, use this line for *vim*:

  ```
  /* vim:set expandtab softtabstop=4 tabstop=4 shiftwidth=4 textwidth=90 cindent: */
  ```

  Or, if you know about EditorConfig, use a file similar to this one: .editorconfig

Exceptions

- **Derive custom exception classes from std::exception.**
  Many people like to catch all possible exceptions with `catch(std::exception &e)`. We don't recommend that practice, but it is a hell of a lot better than `catch(...)`. More importantly, others can catch std::exceptions even if they do not know anything about your custom class – they do not need to include your header file(s). Plus, if you derive from `std::runtime_error` or `std::logic_error` (which are in turn derived from `std::exception`), you get a customizable string message for free – and implementing that safely in an exception class is not as trivial as it looks.
- **Catch exceptions by const reference.**
  Catching by value leads to *slicing* of the exception object if the exception is of a derived class and you catch the base class. This means for example that you don't get overridden virtual functions of the derived class but those with possibly misleading information from the base class. Catching by reference (preferably const) avoids that error.
  Good:
  `catch(const std::exception &e)`
  Bad:
  ~~`catch(std::exception e)`~~

Header Files

- **Use include guards.**
  Every header file must be protected against multiple inclusion by *one* of the two following mechanisms:
  A pragma once statement (only if you are not targeting Solaris platforms, because the Sun Studio compiler is probably the only C++ compiler out there that does not support it):
  `#pragma once`
  A "traditional" define guard of the form:
  ```
  #ifndef FOO_BAR_H_
  #define FOO_BAR_H_
  ...
  #endif // FOO_BAR_H_
  ```

- **Do not use "using namespace" directives in header files.**
  `using namespace` pollutes the namespace of any piece of code that includes the header file. It has completely unexpected side effects for the user of the header file or even for other headers included afterwards. In fact, using `using namespace` is bad practice even in .cc files, but at least it stays locally confined in this case.

Naming

- **Function names are_in_small_letters_with_underscores.**
  All functions – also member functions – have names made of small letters (and maybe numbers). Underscores are used at word boundaries. A function name should clearly and concisely describe what the function does. This usually means that the name should start with a verb:
  ~~`int do_it();`~~ `// NOT OK - it is unclear what the function does`
  ```
  std::string MyClass::get_description() const; // OK
  bool is_timer_expired(); // OK
  int count_llamas(std::vector<Animal *> animals); // OK
  size_t MyContainer::empty() const; // OK – empty(), size() etc. are idiomatic for C++ containers
  ```
- **Functions in an overload set must perform the same task.**
  Functions of the same name in a namespace or class scope form an overload set. While they may differ in arguments and qualifiers, the user expects them to do (roughly) the same thing.
  ```
  // NOT OK: Functions with the same name as setter and getter. Prefer set_description and
  get_description.
  ~~std::string description();~~
  ~~void description(const std::string &description);~~
  // OK: Accept different parameter types for the same logical operation
  void set_description(const std::string &description);
  void set_description(const char *description);
  ```
- **Variable names consist of small_letters_with_underscores.**
  All variables – local, global, member – have names made of small letters (and maybe numbers). Underscores are used at word boundaries. Variable names should be understandable to someone not familiar with the code; abbreviations are only OK if they are commonly and consistently used:
  ~~`std::string colnm;`~~ `// NOT OK - incomprehensible abbreviation`
  ```
  std::string column_name; // OK
  int num_columns; // OK - num as "number of" is idiomatic
  ```

- **Member variable names end with an underscore_.**
  Member variables of classes and structs should end with an underscore. This makes it much easier to recognize them in member functions.
  ```
  class A
  {
  public:
     A(int a, double b) : a_{a}, b_{b}
     {}
  private:
     int a_;
     double b_;
  };
  ```

- **Class/type names use CamelCaseStartingWithACapital.**
  All types – classes, structs, typedefs, enums – start with a capital letter and have a capital letter for each new word. Underscores are not allowed.
  Exceptions:
  DOOCS D-functions start with the prefix "D_" and continue with lowercase letters; underscores are allowed (`D_float`).
- **Namespace names consist of small letters.**
  Namespace names should be as concise as possible. If there really needs to be a word boundary, use underscores:
  ```
  namespace hlc {}
  namespace white_rabbit {}
  ```

- **Macro names are written in capital letters: MY_MACRO_DESTROYS_OTHER_PEOPLES_CODE**
  Don't use macros. But, if you really have to, at least make them easy to spot by giving them an all-caps name with underscores at word boundaries. Also, please do not use short or frequently-used names like SYSTEM, USER, or HOST. These names will almost certainly cause unwanted macro expansion in someone else's code or in the most obscure library headers.

- **Use .h as filename extension for header files.**
  For pre-existing projects, .hh and .hpp are acceptable as well.

- **Use .cc as filename extension for implementation files.**
  For pre-existing projects, .cpp is acceptable as well.

- **For a major class MyClass, use a single header and implementation file MyClass.h and MyClass.cc.**
  Using the class name as a filename makes it easier for others to find important classes in your repository. It also minimizes the chance of version control conflicts and merges. Plus, it probably reduces compile time and decreases the need to rebuild other parts of your project.

Strings

- **Use std::string instead of char *.**
  The standard string class saves you from all kinds of problems with memory management and pointer handling. It is not only safer but also easier to use.
  When you have to call legacy code that accepts only C-style char * parameters, use `c_str()`:
  ```
  std::string str = "Hello World!";
  call_nasty_legacy_function(str.c_str());
  ```

- **Pass string arguments by const reference by default.**
  Unless you need to do something special, use `const std::string &` as the type for string input parameters. Among other things, this has the advantage that the function can also be called with C-style strings because there is an implicit conversion:
  ```
  void do_something(const std::string &str) { ... }
  int main()
  {
      std::string std_str = "Test with std::string";
      do_something(std_str); // Works, of course.
      do_something("Test with char *"); // Works, too!
  }
  ```

- **Return strings by value.**
  There are many ways of returning strings. Dealing with pointers (either to char or to std::string) is dangerous because ownership of the memory is unclear – who deletes it and when? The same is true for returning references – how long is the referenced string going to live? A safe but unintuitive way is to modify a string passed by non-const reference. This makes the code verbose and hard to read:
  ~~~~
  ~~void get_name(std::string &str) { str = "Hello World"; }~~
  ~~std::string str;~~
  ~~get_name(str);~~
  ~~~~
  To make your code both safe and easy to understand, just keep it simple – return the string by value:
  ```
  std::string get_name() { return "Hello World"; }
  std::string str = get_name();
  ```
  If you worry about performance, consider that modern compilers use return value optimization so that this version has no higher cost than the first one. If you do not believe it, please, measure – and if this part of the code is not important enough for you to measure, you probably should not think about optimizing it anyhow.

General Advice

- **Read pointer and reference declarations right-to-left.**
  OK, this is not a guideline, but it makes life easier, and surprisingly few C++ programmers seem to know it. Try to read the following declarations:
  ```
  const int a; // a is an int that is const
  int const b; // b is a const int (same as a)
  int * const c; // c is a const pointer to an int
  std::string *&d;  // d is a reference to a pointer to a std::string
  const char * const * const e; // e is a const pointer to a const pointer to a char that is const
  ```
  If there are parentheses in the declaration, just read the innermost parenthesis first, then continue outwards:
  ```
  int * (* (*f)(int) ) [10]; // f is a pointer to a function of int  that returns a pointer to an array of 10 pointers to int
  ```
  And please... the fact that you can somehow read a declaration like this does not mean that you should write it. Ever.