University of Puerto Rico
Mayagüez Campus
College of Engineering
Department of Computer Science and Engineering

UPR
Recinto Universitario de Mayagüez

Selected GOOGLE coding styles

This file specifies which of the GOOGLE recommendations about coding style for C/C++ will be required for the laboratories and programming projects of the Operating systems course of the University of Puerto Rico at Mayaguez.

Source: https://google.github.io/styleguide/cppguide.html

Header Files

In general, every .cc file should have an associated .h file. There are some common exceptions, such as unit tests and small .cc 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/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_
```

Naming

File Names

Filenames should be all lowercase and can include underscores (`_`) or dashes (`-`). Follow the convention that your project uses. If there is no consistent local pattern to follow, prefer "`_`".

Examples of acceptable file names:

- `my_useful_class.cc`
- `my-useful-class.cc`
- `myusefulclass.cc`
- `myusefulclass_test.cc // _unittest and _regtest are deprecated.`

C++ files should end in `.cc` and header files should end in `.h`.

In general, make your filenames very specific. For example, use `http_server_logs.h` rather than `logs.h`. A very common case is to have a pair of files called, e.g., `foo_bar.h` and `foo_bar.cc`, defining a class called `FooBar`.

Type Names

Type names start with a capital letter and have a capital letter for each new word, with no underscores: `MyExcitingClass`, `MyExcitingEnum`.

The names of all types — classes, structs, type aliases, enums, and type template parameters — have the same naming convention. Type names should start with a capital letter and have a capital letter for each new word. No underscores. For example:

```
// classes and structs
class UrlTable { ...
class UrlTableTester { ...
struct UrlTableProperties { ...

// typedefs
typedef hash_map<UrlTableProperties *, std::string> PropertiesMap;

// using aliases
using PropertiesMap = hash_map<UrlTableProperties *, std::string>;

// enums
enum class UrlTableError { ...
```

Concept Names

Concept names follow the same rules as type names.

Variable Names

The names of variables (including function parameters) and data members are `snake_case` (all lowercase, with underscores between words). Data members of classes (but not structs) additionally have trailing underscores. For instance: `a_local_variable`, `a_struct_data_member`, `a_class_data_member_`.

Common Variable names

For example:

```
std::string table_name;  // OK - snake_case.
```

```
std::string tableName;   // Bad - mixed case.
```

Struct Data Members

Data members of structs, both static and non-static, are named like ordinary nonmember variables. They do not have the trailing underscores that data members in classes have.

```
struct UrlTableProperties {
  std::string name;
  int num_entries;
  static Pool<UrlTableProperties>* pool;
};
```

Function Names

Regular functions have mixed case; accessors and mutators may be named like variables.

Ordinarily, functions should start with a capital letter and have a capital letter for each new word.

```
AddTableEntry()
DeleteUrl()
OpenFileOrDie()
```

Until January 2009, the style was to name enum values like macros. This caused problems with name collisions between enum values and macros. Hence, the change to prefer constant-style naming was put in place. New code should use constant-style naming.

Macro Names

You're not really going to define a macro, are you? If you do, they're like this:

`MY_MACRO_THAT_SCARES_SMALL_CHILDREN_AND_ADULTS_ALIKE`.

Please see the description of macros; in general macros should *not* be used. However, if they are absolutely needed, then they should be named with all capitals and underscores, and with a project-specific prefix.

```
#define MYPROJECT_ROUND(x) ...
```

Formatting

Coding style and formatting are pretty arbitrary, but a project is much easier to follow if everyone uses the same style. Individuals may not agree with every aspect of the formatting rules, and some of the rules may take some getting used to, but it is important that all project contributors follow the style rules so that they can all read and understand everyone's code easily.

Line Length

Each line of text in your code should be at most 80 characters long.

We recognize that this rule is controversial, but so much existing code already adheres to it, and we feel that consistency is important.

Spaces vs. Tabs

Use only spaces, and indent 2 spaces at a time.

We use spaces for indentation. Do not use tabs in your code. You should set your editor to emit spaces when you hit the tab key.

Function Declarations and Definitions

Return type on the same line as function name, parameters on the same line if they fit. Wrap parameter lists which do not fit on a single line as you would wrap arguments in a function call.

Functions look like this:

```
ReturnType ClassName::FunctionName(Type par_name1, Type par_name2) {
  DoSomething();
  ...
}
```

If you have too much text to fit on one line:

```
ReturnType ClassName::ReallyLongFunctionName(Type par_name1, Type par_name2,
                                             Type par_name3) {
  DoSomething();
  ...
}
```

or if you cannot fit even the first parameter:

```
ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
    Type par_name1,  // 4 space indent
    Type par_name2,
    Type par_name3) {
  DoSomething();  // 2 space indent
  ...
}
```

Some points to note:

- Choose good parameter names.
- A parameter name may be omitted only if the parameter is not used in the function's definition.
- If you cannot fit the return type and the function name on a single line, break between them.
- If you break after the return type of a function declaration or definition, do not indent.
- The open parenthesis is always on the same line as the function name.
- There is never a space between the function name and the open parenthesis.
- There is never a space between the parentheses and the parameters.
- The open curly brace is always on the end of the last line of the function declaration, not the start of the next line.
- The close curly brace is either on the last line by itself or on the same line as the open curly brace.
- There should be a space between the close parenthesis and the open curly brace.
- All parameters should be aligned if possible.
- Default indentation is 2 spaces.
- Wrapped parameters have a 4 space indent.

Pointer and Reference Expressions

No spaces around period or arrow. Pointer operators do not have trailing spaces.

The following are examples of correctly-formatted pointer and reference expressions:

```
x = *p;
p = &x;
x = r.y;
x = r->y;
```

Note that:

- There are no spaces around the period or arrow when accessing a member.
- Pointer operators have no space after the `*` or `&`.