Inhaltsverzeichnis

Style Guide

Written to find an consistent programming code style.
Based on the "Google Style Guide".[1]

Rules:

1. Never break rules
2. Study this document
3. If you have studied this document and you have any issues, ready this document again - better read, this document twice or thrice.
4. If you have further any issues, please contact the author of this document.
5. If the author does not answer your query: GL & HF!

In all the other cases: Happy coding! (-;

[1] http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

0.1 Header files

0.1.1 Define Guard

```
#ifndef SAMPLE_H
#define SAMPLE_H
// some code here
#endif // SAMPLE_H
```

0.1.2 Forward Declarations

#include lines can ofter be replaced with forward declarations.
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.

0.1.3 Inline Functions

Include functions as inline functions only when thay are small. 10 lines or less.

0.1.4 Function Parameter Ordering

Parameter order: First inputs, the outputs.

0.1.5 Name and Order of Includes

1. C system files
2. C++ system files
3. Other libraries .h files
4. Your project .h files

Sample:

```
// Preferred location.
#include "foo/public/foosvr.h"

#include <sys/types.h>
#include <unistd.h>
#include <hash_map>
#include <vector>

#include "base/si.h"
#include "base/ne.h"
#include "foo/public/p.h"
```

0.2 Scoping

0.2.1 Namespaces

Access to function via X::Y::function()

```
namespace nX
{
  inline namespace nY
  {
    void tFunction();
  }
}
```

0.2.2 Local Variables

Declare local variables in as local a scope as possible and as close to the first use as possible.
Bad. Initialization separate from declaration and use brace initialization.

```
int tI;
tI = tF();
```

```
vector<int> tV;
tV.push_back(1);
tV.push_back(2);
```

Good. Initialization separate from declaration and brace initialization.

```
int tI = tF();
```

```
vector<int> tV = {1, 2};
```

Avoid inefficient implementation for loops like this:

```
for (int tI = 0; tI <
        1000000; ++tI)
{
  Foo tF; // called 1kk times
  tF.DoSomething(tI);
}
```

Efficient:

```
Foo tF; //called once
for (int tI = 0; tI <
        1000000; ++tI)
{
  tF.DoSomething(tI);
}
```

0.2.3 Static and Global Variables

Global variables are discouraged. Code with global variables make the code less reusable. Avoid global variables as much as possible and try to declare variables and functions inside classes. The remainder should be declared in namespaces.

Static variables are also discouraged. They prevent multiple instantiations of a pice of code and make multi-threded programming a nightmare.

0.3 Classes

0.3.1 Constructors

Avoid to call virtual functions in constructors and avoid doing complex initializations in constructors. For complex initializations consider using a factory function or an Ïnit-Method.

0.3.2 Explicit Constructors

Use the keyword ëxplicit"for constructors with only on argument, like this:

```
explicit fFunc(string pName);
```

0.3.3 Copy Constructors

Provide a copy constructor and assignment operator only when necessary.

0.3.4 Initialization

If your class defines some member variables, you must provide an in-class initializer for every(!) member variable or write an constructor! Try always to initialize variables in a (default-)constructor, also for variables they aren't initialized in-class.

0.4 C++ Features

0.4.1 Reference Arguments

All parameters passed by reference must be labeled "const":

```
void Function(const string &in
        , string *out);
```

Input arguments are values or const references and output arguments are pointers.

0.4.2 Function Overloading

Use overloaded functions (including constructors) only if a reader looking at a call site can get a good idea of what is happening without having to first figure out exactly which overload is being called. If you want to overload a function, consider qualifying the name with some information about the argeuments, e.g. "AppendString()", "AppendInt()" rather than just "Append()".

0.4.3 Default Arguments

Default Arguments are not allowed, except if a function is a static function or if default arguments are used to simulate variable-length argument lists. Simulate them with function overloading instead, if appropriate.

0.4.4 Variable-Length Arrays and alloca()

Not allowed.

0.4.5 Friends

Friends are only allowed if they are defined in the same file so the reader does not have to look in antother file to find uses of the private members of a class.

0.4.6 Exceptions

Not usefull.

0.4.7 Casting

Use c++ casts like "static_cast<>()". Do not use other cast formats like:

```
int y = (int)x;
//or
int y = int(x);
```

Only use:

- static_cast as the equivalent of a C-style cast that does value convension
- const_cast to remove the const qualifier
- reinterpret_cast to du unsafe conversions of pointer types to and from integer and other pointer types

0.4.8 Streams

Use streams only for logging.

0.4.9 Preincrement and Predecrement

Use prefix form (++i) of the increment and decrement operators with iterators and other template objects.

0.4.10 Const

Use "const" whenever it makes sense.

0.4.11 Preprocessor Macros

Be very cautious with macros. Prefer inline functions, enums, and const variables to macros. 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 right before you use them, and #undef them right after.
- Do not just #undef an existing macro 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.

0.4.12 Nullptr, NULL and 0

Use 0 for integers, 0.0 for reals, nullptr (or NULL) for pointers, and '\0' for chars.

0.4.13 sizeof

Prefer sizeof(varname) to sizeof(type).

0.4.14 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.

0.4.15 Brace Initialization

You may use brace initialization:

```
struct Point {int x; int y;};
Point p = {1, 2};
```

In C++11:

```
// Vector takes lists of
// elements
vector<string> v{"fo", "ba"};

// If constructor is explicit
vector<string> v =
        {"fo", "ba"};

// Maps take lists of pairs
// Nested braced-init-lists
// work
map<int, string> m = {{1,
        "one"}, {2, "2"}};

// braced-init-lists can be
// implicitly converted to
// return types.
vector<int> test_function()
{
  return {1, 2, 3};
}

// Iterate over a braced-init-
// list.
for (int i : {-1, -2, -3}) {}

// Call a function using a
// braced-init-list.
void test_function2(
        vector<int> v) {}
test_function2({1, 2, 3});
```

0.4.16 Lambda Expressions

Do not use lambda expressions, or the related "std::function" or "std::bind" utilities.

0.5 Naming

0.5.1 General Naming Rules

Function names, variable names, and filenames should be descriptive; eschew abbreviation. Good.

```
// No abbreviation
int price_count_reader;

// "num" is a widespread
// convention
int num_errors;

/// Most people know what
// "DNS" stands for
int num_dns_connections;
```

Bad.

```
// Meaningless
int n;

// Ambiguous abbreviation
int nerr;

// Ambiguous abbreviation
int n_comp_conns;

// Only your group knows
// what this stands for
int wgc_connections;

// Lots of things can be
// abbreviated "pc"
int pc_reader;

// Deletes internal letters
int cstmr_id;
```

0.5.2 Files 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:

```
my_useful_class.cc
my-useful-class.cc
myusefulclass.cc

// unittest and _regtest
// are deprecated
myusefulclass_test.cc
```

C++ files should end with ".cc" and header files with ".h". Do not use filenames that already exist. Make files much mor specific:

```
// class declaration
url_table.h
// class definition
url_table.cc
// inline functions that
// include lots of code
url_table-inl.h
```

0.5.3 Type Names

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

```
class UrlTable { ...
class UrlTableTester { ...
struct UrlTableProperties { ...

typedef hash_map<
  UrlTableProperties *,
  string> PropertiesMap;

enum UrlTableErrors { ...
```

0.5.4 Variable Names

Variable names are all lowercase, with underscores between words. Class member variables have trailing underscores. For instance: my_exciting_local_variable, my_exciting_member_variable_. Common variables:

```
string table_name; // OK
string tablename;  // OK
string tableName;  // Bad
```

Class Data Members (instance variables or member variables) are lowercase with optional underscores like regular variable names, but always end with a trailing underscore:

```
string table_name_;  // OK
string tablename_;   // OK
```

Struct Variables should be named like regular variables without the trailing underscores that data members in classes have:

```
struct UrlTableProperties
{
  string name;
  int num_entries;
}
```

Global Variables should have a prefix like "g_".

0.5.5 Constant Names

Use a k followed by mixed case: "kDaysInAWeek".

0.5.6 Function Names

Regular functions have mixed case; accessors and mutators match the name of the variable: "MyExcitingFunction()", "MyExcitingMethod()", "my_exciting_member_variable()", "set_my_exciting_member_variable()".

0.5.7 Namespace Names

Namespace names are all lower-case, and based on project names and possibly their directory structure: "google_awesome_project".

0.5.8 Enumerator Names

Enumerators should be named either like constants or like macros: either kEnumName or ENUM_NAME.

0.5.9 Macro Names

You're not really going to define a macro, are you? All letters in uppercase like this: "MY_MACRO_THAT_SCARES_SMALL_CHILDREN".