This is an excerpt from **Google C++ Style Guide**

Naming

The most important consistency rules are those that govern naming. The style of a name immediately informs us what sort of thing the named entity is: a type, a variable, a function, a constant, a macro, etc., without requiring us to search for the declaration of that entity. The pattern-matching engine in our brains relies a great deal on these naming rules.

Naming rules are pretty arbitrary, but we feel that consistency is more important than individual preferences in this area, so regardless of whether you find them sensible or not, the rules are the rules.

General Naming Rules

Optimize for readability using names that would be clear even to people on a different team.

Use names that describe the purpose or intent of the object. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader. Minimize the use of abbreviations that would likely be unknown to someone outside your project (especially acronyms and initialisms). Do not abbreviate by deleting letters within a word. As a rule of thumb, an abbreviation is probably OK if it's listed in Wikipedia. Generally speaking, descriptiveness should be proportional to the name's scope of visibility. For example, n may be a fine name within a 5-line function, but within the scope of a class, it's likely too vague.

```
class MyClass {
 public:
  int CountFooErrors(const std::vector<Foo>& foos) {
    int n = 0;  // Clear meaning given limited scope and context
    for (const auto& foo : foos) {
      ...
      ++n;
    }
    return n;
  }
  void DoSomethingImportant() {
    std::string fqdn = ...;  // Well-known abbreviation for Fully Qualified Domain Name
  }
```

```
 private:
  const int kMaxAllowedConnections = ...;  // Clear meaning within context
};

class MyClass {
 public:
  int CountFooErrors(const std::vector<Foo>& foos) {
    int total_number_of_foo_errors = 0;  // Overly verbose given limited scope and context
    for (int foo_index = 0; foo_index < foos.size(); ++foo_index) {  // Use idiomatic `i`
      ...
      ++total_number_of_foo_errors;
    }
    return total_number_of_foo_errors;
  }
  void DoSomethingImportant() {
    int cstmr_id = ...;  // Deletes internal letters
  }
 private:
  const int kNum = ...;  // Unclear meaning within broad scope
};
```

Note that certain universally-known abbreviations are OK, such as i for an iteration variable and T for a template parameter.

For the purposes of the naming rules below, a "word" is anything that you would write in English without internal spaces. This includes abbreviations, such as acronyms and initialisms. For names written in mixed case (also sometimes referred to as "camel case" or "Pascal case"), in which the first letter of each word is capitalized, prefer to capitalize abbreviations as single words, e.g., StartRpc() rather than StartRPC().

Template parameters should follow the naming style for their category: type template parameters should follow the rules for type names, and non-type template parameters should follow the rules for variable names.

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. Files that rely on being textually included at specific points should end in .inc (see also the section on self-contained headers).

Do not use filenames that already exist in /usr/include, such as db.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 { ...
```

Variable Names

The names of variables (including function parameters) and data members are 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 - lowercase with underscore.

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

Class Data Members

Data members of classes, both static and non-static, are named like ordinary nonmember variables, but with a trailing underscore.

```
class TableInfo {
  ...
 private:
  std::string table_name_;  // OK - underscore at end.
  static Pool<TableInfo>* pool_;  // OK.
};
```

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;
};
```

See Structs vs. Classes for a discussion of when to use a struct versus a class.

Constant Names

Variables declared constexpr or const, and whose value is fixed for the duration of the program, are named with a leading "k" followed by mixed case. Underscores can be used as separators in the rare cases where capitalization cannot be used for separation. For example:

```
const int kDaysInAWeek = 7;
const int kAndroid8_0_0 = 24;  // Android 8.0.0
```

All such variables with static storage duration (i.e., statics and globals, see Storage Duration for details) should be named this way. This convention is optional for variables of other storage classes, e.g., automatic variables, otherwise the usual variable naming rules apply.

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()

(The same naming rule applies to class- and namespace-scope constants that are exposed as part of an API and that are intended to look like functions, because the fact that they're objects rather than functions is an unimportant implementation detail.)

Accessors and mutators (get and set functions) may be named like variables. These often correspond to actual member variables, but this is not required. For example, int count() and void set_count(int count).

Namespace Names

Namespace names are all lower-case, with words separated by underscores. Top-level namespace names are based on the project name . Avoid collisions between nested namespaces and well-known top-level namespaces.

The name of a top-level namespace should usually be the name of the project or team whose code is contained in that namespace. The code in that namespace should usually be in a directory whose basename matches the namespace name (or in subdirectories thereof).

Keep in mind that the rule against abbreviated names applies to namespaces just as much as variable names. Code inside the namespace seldom needs to mention the namespace name, so there's usually no particular need for abbreviation anyway.

Avoid nested namespaces that match well-known top-level namespaces. Collisions between namespace names can lead to surprising build breaks because of name lookup rules. In particular, do not create any nested std namespaces. Prefer unique project identifiers (websearch::index, websearch::index_util) over collision-prone names like websearch::util. Also avoid overly deep nesting namespaces (TotW #130).

For internal namespaces, be wary of other code being added to the same internal namespace causing a collision (internal helpers within a team tend to be related and may lead to collisions). In such a situation, using the filename to make a unique internal name is helpful (websearch::index::frobber_internal for use in frobber.h).

Enumerator Names

Enumerators (for both scoped and unscoped enums) should be named like constants, not like macros. That is, use kEnumName not ENUM_NAME.

```
enum class UrlTableError {
  kOk = 0,
  kOutOfMemory,
  kMalformedInput,
};

enum class AlternateUrlTableError {
  OK = 0,
  OUT_OF_MEMORY = 1,
  MALFORMED_INPUT = 2,
};
```

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.

```
#define ROUND(x) ...
#define PI_ROUNDED 3.0
```

Exceptions to Naming Rules

If you are naming something that is analogous to an existing C or C++ entity then you can follow the existing naming convention scheme.

```
bigopen()
function name, follows form of open()
uint
typedef
bigpos
struct or class, follows form of pos
sparse_hash_map
```

STL-like entity; follows STL naming conventions
LONGLONG_MAX
a constant, as in INT_MAX