

Advanced Analytics and Data Sciences **C++** Style Guide

Drafted by XXX, XXX, and Haoda Fu ©*2019*

December 16, 2018

Contents

Contents	i
Preface	1
1 Naming Convention	3
1.1 General Naming Rules	3
1.2 File Names	4
1.3 Type Names	5
1.4 Variable Names	5
1.5 Constant Names	6
1.6 Function Names	6
1.7 Namespace Names	6
2 Functions	9
2.1 Reference Arguments	9
Bibliography	11

List of Figures

List of Tables

Preface

`C++` is one of the main development languages used by many of Eli Lilly and Company Advanced Analytics and Data Sciences (AADS) team. As every `C++` programmer knows, the language has many powerful features, but this power brings with it complexity, which in turn can make code more bug-prone and harder to read and maintain. This guidance is primary based on [Google C++ Style Guide](#) and incorporates some Lilly code development styles.

The goal of this guide is to manage this complexity by describing in detail the dos and don'ts of writing `C++` code. These rules exist to keep the code base manageable while still allowing developer to use `C++` language features productively.

Style, also known as readability, is what we call the conventions that govern our `C++` code. The term Style is a bit of a misnomer, since these conventions cover far more than just source file formatting.

Note that this guide is not a `C++` tutorial: we assume that the reader is familiar with the language.

Chapter 1

Naming Convention

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.

1.1 General Naming Rules

Names should be descriptive; avoid abbreviation.

Give as descriptive a name as possible, within reason. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader. Do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do not abbreviate by deleting letters within a word. Abbreviations that would be familiar to someone outside your project with relevant domain knowledge are OK. As a rule of thumb, an abbreviation is probably OK if it's listed in Wikipedia.

```
//Recommended
int price_count_reader; // No abbreviation.
int num_errors; // "num" is a widespread convention.
int num_dns_connections; // Most people know what "DNS" stands for.
```

```

int lstm_size; // "LSTM" is a common machine learning abbreviation.

//Not recommended
int n; // Meaningless.
int nerr; // Ambiguous abbreviation.
int n_comp_conns; // Ambiguous abbreviation.
int wgc_connections; // Only your group knows what this stands for.
int pc_reader; // Lots of things can be abbreviated "pc".
int cstmr_id; // Deletes internal letters.
FooBarRequestInfo fbri; // Not even a word.

```

Note that certain universally-known abbreviations are OK, such as `T` for a template parameter and `i` for an iteration variable in small iteration environment (when iteration contains more than 10 lines, we recommend to use `iter` for iterations).

For some symbols, this style guide recommends names to start with a capital letter and to have a capital letter for each new word (a.k.a. "Camel Case" or "Pascal case"). When abbreviations or acronyms appear in such names, prefer to capitalize the abbreviations or acronyms as single words (i.e `StartRpc()`, not `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.

1.2 File Names

Since some of the operation systems use case in-sensitive file systems, 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 `.cpp` 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.cpp`, defining a class called `FooBar`.

1.3 Type Names

Type names follow UpperCamelCase, i.e. 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 *, string> PropertiesMap;

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

//enums
enum UrlTableErrors { ...
```

1.4 Variable Names

The names of variables (including function parameters) and data members follow lowerCamelCase. Private data members of classes additionally have trailing underscores. For instance: `aLocalVariable`, `aStructPublicDataMember`, `aClassPrivateDataMember_`.

```
string tableName; // common variable name.
```

```
class TableInfo {  
    ...  
    private:  
        string tableName_; // OK - underscore at end.  
        static Pool<TableInfo>* pool_; // OK.  
};  
  
struct UrlTableProperties {  
    string name;  
    int numEntries; //public data member  
    static Pool<UrlTableProperties>* pool;  
};
```

1.5 Constant Names

Constant names use `CONSTANT_CASE`: all uppercase letters, with each word separated from the next by a single underscore. These names are typically nouns or noun phrases.

```
const int DAYS_IN_A_WEEK = 7;
```

1.6 Function Names

Function names are written in `lowerCamelCase`. Function names are typically verbs or verb phrases. For example, `sendMessage` or `searchValue`.

1.7 Namespace Names

Namespace names are all lower-case. 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`.

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`)

Chapter 2

Functions

2.1 Reference Arguments

Definition: In C, if a function needs to modify a variable, the parameter must use a pointer, eg `int foo(int *pval)`. In C++ , the function can alternatively declare a reference parameter: `int foo(int &val)`.

Pros: Defining a parameter as reference avoids ugly code like `(*pval)++`. Necessary for some applications like copy constructors. Makes it clear, unlike with pointers, that a null pointer is not a possible value.

Cons: References can be confusing, as they have value syntax but pointer semantics.

Decision: Within function parameter lists all references must be `const`:

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

In fact it is a very strong convention in Lilly that input arguments are values or `const` references while output arguments are pointers. Input parameters may be `const` pointers, but we never allow non-`const` reference parameters except when required by convention, e.g., `swap()`.

However, there are some instances where using `const T*` is preferable to `const T&` for input parameters. For example:

- You want to pass in a null pointer.
- The function saves a pointer or reference to the input.

Remember that most of the time input parameters are going to be specified as `const T&`. Using `const T*` instead communicates to the reader that the

input is somehow treated differently. So if you choose `const T*` rather than `const T&`, do so for a concrete reason; otherwise it will likely confuse readers by making them look for an explanation that doesn't exist.

Bibliography

- [1] L. Lamport. **L^AT_EX A Document Preparation System** Addison-Wesley, California 1986.