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

Drafted by XXX, XXX, and Haoda Fu

December 15, 2018

Contents

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.

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. **LaTeX A Document Preparation System** Addison-Wesley, California 1986.