

Personal coding style guide regarding the C/C++ languages

Vladislav Valentinov

v1.0

November 1, 2018

1 Introduction

There is a large number of coding standarts, developed both by organization and individuals. When talking about C/C++ coding conventions, some adopt the Java naming style, others try a more traditional scheme.

I personally prefer to write in lower case, separating words with underscores. Although it is hard to think of any objective reasons for my choice, others, including Bjarne Stroustrup^[2] have given arguments defending this style of writting. Separating words with underscores is argued to improve readability and it generally looks more consistent when combined with library code (stl, boost code). The use of underscores to separate parts of a name is the original C and C++ style and used in the C++ standard library.

This document describes a coding style convention, which I try to follow in my personal projects. At its current version it is not very specific, and applies only to the C and C++ languages. When I start writting in Java and Python, then I might develop a separate style convention for those languages. My main source for the naming convention and the rules are [1] and [3].

2 General rules

This section presents some rules and suggestions, taken from [2], that I found usefull to adopt.

- Avoid names that are easy to mistype, misread, or confuse. For example: name, names, nameS, foo, f00, fl, f1, fI, fi. The characters 0, o, 0, 1, l, I, are particularly prone to cause trouble.

- Put “`const`” before. This will confuse fewer programmers (“is more idiomatic”). Ofcourse, this is a matter of taste, because both “`const T`” and “`T const`” are allowed and equivalent:

```
const int a = 1; // correct
int const b = 2; // also correct
```

Note that in const pointers, “`const`” always comes after the “*”. For example:

```
int *const p1 = q; // constant pointer to int variable
int const* p2 = q; // pointer to constant int
const int* p3 = q; // pointer to constant int
```

2.1 Naming files

To easier distinguish between C and C++, code header files should be named “*.hpp” and source files “*.cpp”.

2.2 Declaration layout in C++

The C-style layout emphasizes use in expressions and grammar, whereas the C++-style emphasizes types:

```
T& operator[](size_t); // C++ style
T &operator[](size_t); // C style
T & operator[](size_t); // undecided
```

These layout styles are to be followed.

2.3 Class member declaration order

Use a conventional class member declaration order. A conventional order of members improves readability^[3]. When declaring a class use the following order:

1. types: classes, enums;
2. constructors, assignments, destructor;
3. functions;
4. data;

Use the “`public`” before “`protected`” before “`private`” order. For an example:

```
class X {
public:
    // interface
protected:
    // unchecked function for use by derived class implementations
```

```
private:
    // implementation details
};
```

Sometimes, the default order of members conflicts with a desire to separate the public interface from implementation details. In such cases, private types and functions can be placed with private data.

2.4 Casting in C++

Casts are generally best avoided^[2]. Especially C-style casts. With the exception of “`dynamic_cast`”, their use implies the possibility of a type error or the truncation of a numeric value. Even an innocent-looking cast can become a serious problem if, during development or maintenance, one of the types involved is changed. For example, what does this mean?:

```
x = (T)y;
```

It is not explicitly clear, which leads to complications. Because the C-style cast `(T)` can be used to express many logically different operations, the compiler has only the barest chance to catch misuses.

The “new-style casts” were introduced to give programmers a chance to state their intentions more clearly and for the compiler to catch more errors. Further, it is argued, that they are easier to spot, than C-style casts and are more consistent with the “`template`” notation. Here is a brief description of the c++ style casts:

const_cast Adds or removes the “`const`” modifier.

dynamic_cast Casts an object from a derived type, to one of its base types. If the reverse is performed, the operation will return a “NULL” pointer.

reinterpret_cast Tries to convert from one type to another, even if the two types are not related.

static_cast Similar to the C-style cast, but more restrictive. For example, if the C-style cast would allow an “`int`” pointer to be converted to a “`char`” pointer, the “`static_cast`” will not.

Next, an example, of the c++ style casts:

```
int a = 7;
double* p1 = (double*) &a; // ok (but a is not a double)
double* p2 = static_cast<double*>(&a); // error
double* p2 = reinterpret_cast<double*>(&a); // ok: I really mean it

const int c = 7;
int* q1 = &c; // error
int* q2 = (int*)&c; // ok (but *q2=2; is still invalid code and may fail)
```

```
int* q3 = static_cast<int*>(&c); // error: static_cast doesn't cast
      away const
int* q4 = const_cast<int*>(&c); // I really mean it
```

The idea here is that conversions allowed by “`static_cast`” are somewhat less likely to lead to errors than those that require “`reinterpret_cast`”. In principle, it is possible to use the result of a `static_cast` without casting it back to its original type, whereas you should always cast the result of a `reinterpret_cast` back to its original type before using it to ensure portability^[2].

2.5 Layout style

Use K&R-derived layout. This is the original C and C++ layout^[3]. It preserves vertical space well, and also distinguishes different language constructs (such as functions and classes) well. Example:

```
class C : public B {
public:
    ...
};

void f(int* p, int max)
{
    if (p) {
        ...
    }

    for (int i = 0; i<max; ++i) {
        ...
    }
}
```

Note the space between “`if`” and “`(`”. Use separate lines for each statement, the branches of an “`if`”, and the body of a “`for`”. The “`{`” for a class and a struct is not on a separate line, but the “`{`” for a function is.

3 Naming convention

3.1 Preprocessor identifiers

All preprocessor identifiers should be named using capital letters and underscores.

```
#define CORE_VERSION 0.2
```

Header include guards The identifiers used in include guards, should be named after the file name, ending with the file extension:

```
#ifndef HTTP_SERVER_HPP
#define HTTP_SERVER_HPP
...
#endif
```

3.2 Variables and function parameters

Local variables and function parameter names use “camel” notation, which means that names start with a lowercase letter, and all subsequent words in the name, if any, begin with an uppercase letter, as in:

```
int numOfSamples;
double dx;
```

Variable names should be as short as possible, but remain descriptive.

Constants Constants should be named using lowercase letters and underscores:

```
namespace physics
{
    namespace constants
    {
        const double eps_0 = 8.8542E-12; //vacuum permittivity
    }
}
```

Indices Variables, used as indices, like in loops, should have names with minimal length, like: *i*, *j*, *k*, ...

3.3 C Functions

C functions, or free functions, use the GNU convention of all lowercase letters with underscores.

```
double min(const double a, const double b);
int save_bmp(const char* fileName, bitmap* bmp, const RGB* pal);
```

3.4 Namespaces

```
namespace log
{
    ...
}
```

3.5 Classes, Enums

Class and enumeration names consist of all lower case letters and words are separated with an underscore.

```
class http_server {
    ...
};

enum http_methods {
    GET, POST, UPDATE, PUT, PATCH, DELETE, HEAD, CONNECT, OPTIONS,
    TRACE
};
```

Enumeration values should be named with uppercase letters.

3.6 Class fields, methods

Public members They start with lower case letters and each word is separated with underscore:

```
class http_server {
public:
    void set_uri_handler(...);
    ...
};
```

Setter/getter methods They are named after the variable they set. Setter accepts parameter to set. Getter method does not accept any parameters.

```
class http_server {
public:
    void port(unsigned int port_);
    unsigned int port() const;

private:
    unsigned int port_;
    ...
};
```

Private members Private class fields end with underscore:

```
class http_server {
private:
    unsigned int port_;
};
```

References

- [1] <https://style-guides.readthedocs.io/en/latest/cpp.html>.
- [2] Bjarne Stroustrup. C++ style and technique faq. http://www.stroustrup.com/bs_faq2.html#Hungarian, September 2017.
- [3] Bjarne Stroustrup. C++ core guidelines. <https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md>, August 2018.