C++ Coding Rules

April 19, 2013

Table of Contents

Each one may modify each other code (even more if you have a central repository, with almost all the source code), so it is a good idea to define a minimal set of coding rules. Here are our strongly recommanded suggestions, and the coding rules we want to enforce.

First of all, a remark: write in English and in English only.
Comments, functions names, variables, ... must all be in **English**. It is the de facto programming language.

1 Naming conventions

1.1 Identifiers

Use clear and informative names for identifiers (classes, members, parameters, variables).
Clear and informative names are one of the best tools for creating easily understandable code. The name of any identifier should succinctly describe the purpose of that identifier.

Identifiers names must be a mixture of upper and lowercase letters to delineate individual words.

- Each word in a class name must begin with a capital letter, e.g. **class RandomProcess;**
- In a function name, use upper case letters as word separators, and lower case for the rest of a word, e.g. **void getDescription();**
- For macros, enumeration constants and global constants, words are capitalized and separated by underscores, e.g. **DATA_VALID**, **const int MIN_WIDTH = 1**.
- Typedef names use the same naming policy as for a class.

Class names will generally consist of nouns or noun combinations.
Function names will generally begin with a verb as they describe actions.

Method argument names follow the same rules as function names.
Class members should be prefixed with **m_** (letter *m* followed by an undescore). The rest of the name is similar to method argument names (including the beginning lower case letter).

1.2 Class organization

- In general there will be one class declaration per header file. In some cases, smaller related classes may be grouped into one header file.
- public, protected and private sections in the class declaration should be ordered so that the public section comes first, then the protected section, and lastly the private section.
- Within each section, member functions and member data should not be mixed.

1.3 Files

A file must have the same name as the class it contains.

- C++ header file names should have the extension ".hpp".
- C++ source file names should have the extension ".cpp".

Use special ".hxx" extension for files containing the definition of template classes or inline functions (see sections 2.6 and 2.7 for further details).

2 Layout

2.1 Declaration - definition

You must separate declaration from definition of your classes and functions.
Declaration should be in a header file, définition should be in a source file (see section 1.3).

2.2 Header files

Include guards. Use include guards in header files to prevent multiple inclusion:

```
#ifndef NAMESPACE_CLASSNAME_HPP
#define NAMESPACE_CLASSNAME_HPP
  ...
#endif /* ! NAMESPACE_CLASSNAME_HPP */
```

Important: Use also include guards for ".hxx" files.

Self-sufficient header files. Make your header files self-sufficient. That is to say they include all needed dependencies and can be compiled standalone using the following simple test:

```
#include <header.hpp>

int
main {
  return 0;
}
```

Avoid global variables. Class data members must always be private (or protected).
If access to them is required then this must be provided through public or protected member functions.

2.3 Source files

Member order. Public class members must be declared first, then protected members, and lastly private members. Moreover member functions must be separated from member variables.

For example:

```
class A {

public:
  /* public member functions */

protected:
  /* protected member functions */

private:
  /* private member functions */

public:
  /* public member variables */

protected:
  /* protected member variables */

private:
  /* private member variables */

};
```

Important note: Public member variables should be strongly avoided!

No magic numbers. Do not hard-code values inside your program. For example, do not hard-code image width, or number of P frames between two I frames, or anything else.
If you have to hard-code something, use a global const variable (preferable to a define) so that you only have to change it once.

For example, replace:

```
Matrix m(100, 100);
```

by:

```
static const DEFAULT_MATRIX_NBROWS = 100;
static const DEFAULT_MATRIX_NBCOLS = 100;
Matrix m(DEFAULT_MATRIX_NBROWS, DEFAULT_MATRIX_NBCOLS);
```

Use short functions. Functions should be at most one page of the text editor. A function that spreads on more than one page should be subdivided in smaller functions.

2.4 Required members

Default constructor and the rule of three. The **default constructor**, **copy constructor**, **assignment operator**, and (virtual) **destructor** should be either explicitly declared or made inaccessible and undefined rather than relying on the compiler-generated defaults.
The flag **-Weffc++** for g++ can help you to find such cases.

For example, replace:

```
class MyClass {

public:
  MyClass() {
    m_mat = new Matrix;
  }

protected:
  Matrix* m_mat;

};
```

by:

```
class MyClass {

public:
  MyClass() {
    m_mat = new Matrix;
  }

   MyClass() {
    delete m_mat;
  }

private:
  MyClass(const MyClass&);
  MyClass& operator=(const MyClass&);

protected:
  Matrix* m_mat;

};
```

Here, there is no need to implement copy constructor and assignment operator. If using C++0x, you should even write;

```
private:
  MyClass(const MyClass&) = delete;
  MyClass& operator=(const MyClass&) = delete;
```

Virtual destructor. A destructor of a class that will be inherited must be declared *v*irtual.

2.5 Namespaces

Namespaces must be unique! Make your namespace verbose, just like a absolute path. Prefix with your application name.
Use lower case letters.

For example:

```
namespace myapp {

  namespace core {

    ...
  }

}

myapp::core::Image img;
```

NEVER add a *using namespace* **directive to header files!** If a header file (.hpp) contains a *using namespace* (such as *using namespace std;*) directive, every file that includes this file will use the namespace and so it removes all the interest of the namespace.
It may be a good idea to never write such a directive at all.

You can use namespace shortcuts for convenience.

For example:

```
namespace alias = a::very::long::namespace;

class YourClass : public alias::TheirClass {
  ...
};
```

2.6 Templates

Most of compilers impose that templates must be entirely in a single header file. To presere our declaration/definition separation, you must use the following tip.
Template declaration must be in a header file (".hpp" extension); template definition must be in a specific source file (".hxx" extension). The header file must include the source file; this inclusion must occur *at the end* of the header file. *Use include guards* (section 2.2) in both header and source files!
For example, a Vector.hpp file is the following:

```
#ifndef MYAPP_VECTOR_HPP
#define MYAPP_VECTOR_HPP

namespace myapp {

  template <typename T>
  class Vector {

  public:
    Vector();
    ...
    bool isEmpty() const;

  }; // end class declaration

}

#include "Vector.hxx"

#endif /* ! MYAPP_VECTOR_HPP */
```

and the corresponding Vector.hxx file is:

```
#ifndef MYAPP_VECTOR_HXX
#define MYAPP_VECTOR_HXX

template <typename T>
myapp::Vector<T>::Vector() {
    ...
}

template <typename T>
bool
myapp::Vector<T>::isEmpty() const {
    ...
}

#endif /* ! MYAPP_VECTOR_HXX */
```

2.7 Inline functions

Inline functions must follow the same rules that templates (section 2.6).

3 Formatting

3.1 Indentation, spacing and trailing whitespaces

Indentation must be achieved using spaces; do not use tabs.
Indent using 2 spaces for each level.

You should use one space after each keyword (if, for, while, ...), use one space after a comma.

For example:

```
for (int i=0; i<10; ++i) {
  ...
}

if (a >= 0) {
  ...
} else {
  ...
}
```

Avoid trailing whitespaces in source code, as it may cause unecessary differences for files under version control.

3.2 One statement per line

Each line should contain at most one statement.

For example:

```
++argv; /* Correct */
–argc; /* Correct */
++argv; –argc; /* AVOID! */
```

3.3 Brackets

Opening braces should be on the same line of the statement, except for class and functions/methods declarations.
Closing braces are always on their own line.
It is recommended to add braces even if they contain a single line.
It can be a good idea to add a comment to closing braces identifying the corresponding opening braces if these ones are not in the visible part of your code editor.

Do not put parentheses next to keywords; put a space between.
Do put parentheses next to function names.

For example:

```
MyClass::MyClass(size_t n) : m_vector(n) {
  for (int i=0; i<n; ++i) {
    m_vector[i] = i;
  }
}

void
MyClass::f(int i) const {
  assert(0 <= i);
  assert(m_vector.size() > i);
  if (i == m_vector[i]) {
    std::cerr<<"ok !"<<std::endl;
  } else {
    std::cout<<"ko !!!"<<std::endl;
  }
}
```

3.4 Instructions

If-then-else. Respect brackets rule.
Prefer variable first in condition format.

For example:

```
if (a < 0) {
  std::cerr<<"a is negative!"<<std::endl;
} else if (a == 0){
  std::cerr<<"a is zero!"<<std::endl;
} else {
  std::cerr<<"a is positive!"<<std::endl;
}
```

Switch. Falling through a case statement into the next case statement shall be permitted as long as a comment is included.
The default case should always be present and trigger an error if it should not be reached, yet is reached.
If you need to create variables, put all the code in a block.

For example:

```
switch(...) {
case 1:
  {
    ...
  }
  break;
case 2:
  ...
// FALL THROUGH
case 3:
  {
    int a;
    ...
  }
  break;
default:
  {
    assert(!"Should not get here!");
  }
}
```

3.5 Members

Const elements. Use const at the beginning of the expression.
Prefer

```
const char* p;
```

to

```
char const* p;
```

Pointers and references. The **&** and ***** tokens should be adjacent to the type, not the name.

For example:

```
int* iPtr;
int& iRef;
```

4 Code efficiency

4.1 Functions

Use standard functions. Use **STL** facilities (vector, list, string, iostream, stringstream, and algorithms) or **standard functions** (memcpy, memset, ..) whenever possible. Most often, these functions are more optimized than what you could ever write.

For example, replace:

```
int arr[100];
memset(arr, 0, sizeof(int)*100);
f(arr);
```

or

```
int* arr = new int[100];
memset(arr, 0, sizeof(int)*100);
f(arr);
delete [] arr;
```

by

```
const size_t size = 100;
std::vector<int> arr(size, 0);
f(&arr[0]);
```

For example, replace:

```
char str[100];
```

by:

```
std::string str;
```

For example, replace:

```
int a, b;
char line[500];
...
sscanf(line, "%d %d", &a, &b);
```

by

```
int a, b;
std::string line;
...
std::stringstream ss(line);
ss >> a;
ss >> b;
```

Const correctness. Use **const** keyword after each function of a class that does not modify members of the class, for each pointer or reference parameter of a function when applicable, and even for unmodified local variables. It greatly helps to understand the code.

For example:

```
MyClass::MyClass(const Matrix& m) : m_mat(m) {}

const Matrix&
MyClass::getMatrix() const {
  return m_mat;
}
```

Initialization list. Use initialization list for constructor (better performance wise). In particular, inialize pointers to NULL in initialization lists.

For example, replace:

```
MyClass::MyClass(int a, int b, int c) {
  m_a = a;
  m_b = b;
  m_c = c;
  m_d = NULL;
}
```

by:

```
MyClass::MyClass(int a, int b, int c)
  : m_a(a), m_b(b), m_c(c), m_d(NULL) {}
```

4.2 Variables

Reduce variable scope. It is C++, not C, so variables do not need to be declared at the start of the block.
Hence, declare variables near their use, then the code is easier to read, and the compiler can often do a better job.

For example:

```
for (int i=0; i<N; ++i) {
  const int g = fct(i);
  ...
}
```

However, avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block.

For example:

```
int count;
...
int f(...) {

  if (condition) {
    int count; /* AVOID! */
    ...
  }
  ...
}
```

Incrementation. Prefer pre-increment to post-increment when they are equivalent.
Some compiler produce a useless temporary when post-increment is used (on complex types), thus it is better, performance-wise, to use pre-increment.

For example:

```
typedef std::vector<int> VectorInt;
VectorInt v;
...
for (VectorInt::const_iterator it = v.begin(); it != v.end(); ++it) {
  ...
}
```

Dynamic allocations. Avoid dynamic allocations (new/malloc) as much as possible, in particular in performance critical code.
For example, for complex types (requiring an allocation for example) it could be better, performance-wise, to declare them outside of loops (and thus not the nearest to their use).

For example, replace:

```
for (int i=0; i<N; ++i) {
  Matrix m(nbRows, nbCols);
  ...
}
```

by:

```
Matrix m(nbRows, nbCols);
for (int i=0; i<N; ++i) {
  ...
}
```

4.3 Forward declaration

Forward declarations should be employed when safely applicable to minimize *#include* dependencies and reduce compilation time.
Forward declarations may be applied to avoid including declarations of objects that are used solely by pointer/reference or as function return values; they may not be used if the object serves as a base class, or as a non-pointer/reference class member or parameter.

For example, replace:

```
#include "Matrix.hpp"
#include "Image.hpp"

matrixToImage(const Matrix& m, Image& img);
```

by:

```
class Matrix;
class Image;

matrixToImage(const Matrix& m, Image& img);
```

Here, includes of Matrix.hpp and Image.hpp will only be done in .cpp file.

5 Documentation

5.1 Code comments

Comments must be in English.
You may write a short description of your classes and member functions. Describing arguments is really useful everytime you can raise a critical question, e.g. "Who owns the data? Who is responsible for memory free?". Don't be shy, tell other people.

If an algorithm is described in a paper, it is a good idea to add a reference to a paper, or better an equation number in a paper, or explanations of differences with the paper.
It is recommanded to use Doxygen to document your code.

5.2 Gotchas

Embed special keywords in your code comments so that an automatic process can establish a maintenance report. There is a list of these gotchas:

- :**TODO**: **topic**
  Means there is more to do there, don't forget.
- :**BUG**: [**bugid**] **topic**
  Means there is a *known* bug here; explain it and optionnaly give a bug id.
- :**GLITCH**:
  Means there is a *suspected* bug here, need more debug to confirm.
- :**COMMENT**:
  Add a specific note you want to be reported to others, e.g. to raise a warning.

- :**TRICKY**:
  Tells somebody that the following code is very tricky so don't go changing it without thinking.
- :**KLUDGE**:
  When you've done something ugly (Do you? Really?), say so and explain how you would do it differently next time if you had more time.
- :**OPTIM**:
  An optimization may be possible or needed here.
- :**PARALLEL**:
  Comment relative to possible parallelization.

Gotchas comment may consist of several lines, but the first line should be a self-containing, meaningful summary, respecting the following line format:

:**GOTCHA**: **author date**: **summary**

Date must be in the form *yymmdd*.

For example:

```
// :TODO: tom 091127: implement case of missing data.
// We should read incomplete datasets and fill missing parts
// with a predefined default value.
```

5.3 Project documentation

It is mandatory to have a **README.txt** file at the root of your project that explains how to compile your code (and thus on which other libraries/programs it depends), how to execute your code and what your code does. You may also add the publications where algorithms are described.
It is very useful to have an explanation of all the parameters and the range of validity for their value. It is also very interesting to know which parameters must be modified first to improve the results.
It is a good practice to have good default values for all parameters, and a simple help that may be displayed when the program is run (a **-h** option for example).

6 Compilation

6.1 Choice of the compiler

You program in **C++**, and the primary compiler you use is **gcc**. Try to test your code with different compilers. If you use g++, try to test different versions (4.2, 4.5, 4.6). A good way to do that is to compile on various linux distributions (you can use virtualbox for example to easily install several distributions on your machine). Try also to compile with different compilers if possible: llvm-g++, clang, icpc, ...

6.2 Compiler flags

You should all use **-Wall -Wextra** (and even **-pedantic**) to compile our code, and correct the code to avoid most warnings.
G++ also provides flags **-Wshadow** and **-Weffc++**, very interesting for C++.

You should provide both a debug and an optimized version.
A variable in the Makefile could be enough to decide if you compile a debug or optimized version.
The optimized version should be the fastest possible: use **-O3** (or -O2) and **-DNDEBUG** during compilation.
Compile with **-g** for the debug version.

Intermediary results are often interesting for publications or even for debugging. It should be possible to extract meaningful intermediary results from your program without too much effort. You can for example add a command line parameter or define a specific compilation flag.

7 Debugging and testing

7.1 Assertions

Assert can be used to check pre and post conditions for example.
It ensures that code is correct and also documents code.
Assert has absolutely no impact on performances when code is compiled with -DNDEBUG.

For example:

```
#include <cassert>

int f(int n) {
  assert(n>3);
  const int result = g(n);

  assert(result>=6);
  return result;
}
```

7.2 Memory leaks

Valgrind is a very helpful tool suite to debug programs (to check memory leaks, to assess performance, ...).
It is highly recommended to check memory accesses of your program using valgrind **memcheck** tool.
See page on Valgrind page [1] for more information.

[1] http://www.cprogramming.com/debugging/valgrind.html

8 TODO

- Naming conventions
  - variables: stack, pointer, ref, global, static, const
- Layout
  - methods, accessors
  - friend class
  - typedef inside a class
- Formatting
  - for/while
  - continue/break
  - conditional (?:)
  - c++0x syntax
- Code efficiency
  - prefer inline functions to macros
- Documentation
  - licence
- Compilation
  - cmake
  - using c++0x
- Debugging and testing
  - unit tests
  - automated code checking: Abraxis code check
-
  - write a "design rules" document
  - write a "howto compile" document
  - write a "howto debug" document

8.1 Intentionnaly omitted

Not pertinent enough or to give developer a little freedom...

References

1. C++ FAQ. http://www.parashift.com/c++-faq-lite/.
2. Henri Garreta. Le langage C++. http://henri.garreta.perso.luminy.univmed.fr/Polys/PolyCpp.pdf.
3. Todd Hoff. C++ Coding Standard. http://www.possibility.com/Cpp/CppCodingStandard.html.
4. Scott Meyers. Effective C++, Effective STL. http://www.aristeia.com/books.html.
5. Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley, 2004.

[]