C++ coding guidelines

Jari Häkkinen and Markus Ringnér

$Revision: 46 $
$Date: 2008-08-19 15:33:30 +0200 (Tue, 19 Aug 2008) $

Contents

1 Introduction

Inspired by the guidelines in [1] our main idea with this document is that it should facilitate the communications among developers in our team. Therefore, it will be focused on how to name variables, functions, classes, etc. and how to structure files, modules, classes, etc. However, communication is not all we're after with this document. Even though good design can't be legislated we do provide a set of recommendations for usage of language features. With them we hope that our developers will be able to faster understand each others code.

2 Guidelines for Names and Structuring

2.1 Names

Today all C++ compilers support namespaces, and so do we. We use `theplu` as our major namespace, and use nested project namspaces within the `theplu` namespace. In section 3.1 we explain our view on namespaces usage.

All names should be descriptive unless they are better left as a single letter or word. Words in names should be separated by underscores, except for class names and enum keywords. In class names and enum keywords the first letter in each word should be capitalized and underscores should be avoided. Private member variables should end with an underscore. Functions, variables and enum tags should preferably be all in lower-case. Global variables should be prefixed with upper-case GLOBAL_ and static variables (but not static member variables) with upper-case STATIC_.

```
class MyClass
{
public:
  enum State { active, inactive };
  double calculate_number(void) const;

private:
  double value_;
};

double MyClass::calculate_number(void) const
{
  extern int GLOBAL_factor;
  return value_*GLOBAL_factor;
}
```

2.2 Header File Structure

The complete body of header files should be protected against multiple-inclusion, as follows:

```
#ifndef symbol
#define symbol
<the complete body of the header file>
#endif
```

Note, no lines of code or comments should be outside the #ifndef ... #endif construct. This allows compilers to optimize the reading of header files.

The symbol used in the conditional compilation statement should be made up from the project name and module name. If the module name is MyClass.h and the module is in project myproject then the symbol name should be `_theplu_myproject_myclass_`. The idea of this symbol is that it should be unique in the environment where it is used.

The multiple-inclusion protection should be followed by `#include` statements that describe which interfaces the module depends on. Typically a module depends only on the interfaces of base classes, classes that are contained by value or classes whose member functions are called within inline functions.

Next comes the forward declarations of the classes that the module uses and contains. Forward declarations should always be preferred to `#include` statements, which only should be used when necessary. This will minimize code dependencies, and make the programmer more conscious about what is used and what is not used in their code.

The `#include` statements and forward declarations should both be structured in blocks (separated by empty lines). Each block should contain modules of similar origin and each block should be sorted alphabetically. If needed, a block should begin with a comment explaining it.

All header files must include `config.h` first (if it's needed), followed by the primary header. So for example, `File.cc` should include its primary header `File.h` first (`config.h` omitted here), before other header files. This guarantees that the completeness of each header file is tested during compilation. After the primary header follows inclusion of header files located in the same directory followed by header files belonging to the same project but located in other directories. Then 3rd party project header files are included and last standard header files are included.

```
#include "config.h"

#include "File.h"

// Classes from this project
#include "OtherClass.h"

#include "path/to/other/dir/in/project/SomeClass.h"

// Classes from project X
#include <MyOtherProjectClass.h>

#include <iostream>
#include <vector>
```

Next comes the class specifications, followed by additional operators such as `operator<<` and finally comes the implementation of inline member functions if these are not defined already in the class declaration. We do not support the idea of collecting all inline functions in a separate file since this will only give another file to keep track of.

2.2.1 Class Declaration Structure

All public declarations come first, then all protected declarations followed by all friend declarations and finally all private declarations. The indentation and spacing should be consistent and readable. Generally, we prefer not to indent `public:`, `protected:` and `private:`, while we indent all the declarations.

```
class MyClass
{
public:
  double public_function(void);

protected:
  int protected_function(void);

private:
  string name_;
};
```

The member functions should in each section start off with the constructors, followed by the destructors, then follows the member functions in alphabetical order, and finally the operators are declared in the order defined in Appendix B.

2.2.2 One Class Definition per Module

We generally keep the rule of having one class definition per module. Each class definition is in its own header file. Sometimes, if classes are tightly coupled, we make an exception.

2.2.3 Comments in Header Files

We generate our documentation from our C++ files using Doxygen (see [2]). This reduces the visibility of the structure. We think this is outweighed by not having to duplicate the protoypes of functions in a separate documentation. Before each declaration or definition we want to document we place the following[1]:

```
/**
   Some text documenting the class MyClass
*/
class MyClass
```

[1]By default, Doxygen associates a documentation block with the next declaration or definition. This means that one can have empty lines after the documentation block to enhance the visibility of the structure.

Try to keep comment line lengths within the terminal character limit, i.e less than 80 characters per line. This will make the comments more readable.

2.3 Implementation file structure

We generally keep the rule of placing all the non-inline definitions of the member functions for a class in one module, and that module will not contain definitions of member functions for any other class. Sometimes, if we have made an exception in a header file, we make the same exception in the corresponding implementation file and include the definitions of member functions from several classes in one module.

We begin the implementation file with the necessary `#include` statements and we structure them in the same way as in the header files (see Section 2.2) with the exception that the class's own header file must be included first.

Next comes the definition of member functions and they are ordered in the same way as a section in the class declaration. At the end we have the definitions of additional operators such as `operator<<`.

2.3.1 Comments in the Implementation Module

Most member functions are fully explained by the names and the comments in the header file. If tricky algorithms, that must be explained, are used in a function then we preface its definition with a comment describing the algorithm. Within the code we try to limit our comments to one line. Doxygen supports documentation in several places, but we prefer to only put comments extracted by Doxygen in the header files (cf. [2]).

2.4 Two Issues About Style

Two issues that always come up when programmers with different backgrounds get together are the placement of braces and indentation. In this section we give you our position on these two eternal questions.

2.4.1 Brace placement

We prefer to put the begin brace on the same line as the controlling statement, while we keep the end brace on its own line and at the same indent level as the controlling statement. There are two exceptions to this rule as outlined below.

In general we use the

```
if (mycondition) {
  // statements inside the if
}
```

everywhere where braces are used. The exceptions are brace usage in function definitions

```
MyClass::public_function(void)
{
  // function body
}
```

and in inline declarations we prefer a more compact style

```
class MyClass
{
public:
  inline double return_zero(void) { return 0; }
  inline double another_public_function_with_a_long_name(void)
  { return some_function_returning_double(); }
}
```

where the second style is used when the line has to be split for readability. If declaration and definition for inline functions are separated then the function definition style is preferred.

2.4.2 Indentation

The question of indentation is really not an issue, every indentation level should strictly be a `tab` character. The problem is to get your favourite text editor to show and treat indentation properly (see Appendix A for information on how to configure a few common editors).

2.5 Summary

So far we have been concerned with guidelines that facilitate the communications among our developers. The important thing with these guidelines is not the guidelines themselves but to stick to a chosen set of guidelines.

3 Recommendations for Usage of Language Features

In this section we leave the guidelines for facilitating communication. Instead we will focus on practices of a particular programming style. This programming style is based on Jaris and Markus collective experience. We believe that adhering to this programming style has a positive aspect on the quality of our code. This is not intendend to be a legislation of any kind, and we do not want it to limit our developers. We present this in hope that it will inspire others or at least start discussions. Minimally, it will at least make it easier to understand our code.

3.1 Namespaces

We use namespaces, and avoid importing all symbols from namespaces we use in our programs. Thus, the use of a file wide `using namespace std;` is deprecated and strictly forbidden in header files. Putting the statement `using namespace std;` into a header file will pollute the namespace for everyone using the header file.

In the rare occasions you must import all symbols from a namespace do it within a local scope as in

```
int main(const int argc,const char* argv[])
{
  using namespace::std;               // import all std symbols!
  cout << "This operator usage works fine here\n";
}
```

since this will only polute the `main` scope. Explicit import of classes is preferred, e.g. use `using theplu::my_project::MyClass;` to import `MyClass`. Similarily, if you are only using `cout`, use the explicit import `using std::cout;` instead of importing the entire `std` namespace.

When we create our own namespaces we do it in a nested fashion. All packages developed in-house should be in `theplu` (lower case characters) namespace, and a specific package is then defined within the `theplu` namespace,

```
namespace theplu {
namepace my_project {

class MyClass
{
};

}} // of namepace my_project and namespace theplu
```

3.2 Inline functions and assert(3)

Inline functions should be used sparingly and only for one line statements. In principle inlining should only be used when a performance increase can be proved.

The assert macro must never be used in header files (*i.e.,* in inline functions) to avoid different debug level in header files and production program libraries, and to avoid changes in library behaviour with `-NDEBUG` usage.

3.3 Constructors

We prefer all our classes to have a copy constructor and we like their implementation to use the assignment operator. This is beacuse eventually one will almost always want to copy an object and then it should be done properly. If no copy constructor is implemented we recommend to declare it private, this will create a compiler diagnostic if the constructor is implicitly used.

Implementing the copy constructor in terms of the assignment operator removes duplication of code. Remember that sometimes data members of the object to be constructed has to be properly initialized before the call to the assignment operator.

3.3.1 One argument constructors

One argument constructors can be used in implicit conversions, which can lead to unexpected behaviour. Therefore, we declare our one argument constructors `explicit`, if we do not specifically want them to work in implicit conversions.

3.4 Destructors

3.4.1 Virtual Destructors

We always make our destructors virtual, unless we are certain that no class will ever be derived from them. This is due to that if one tries to delete a derived class through a base class pointer and the base class lacks a virtual destructor the results are undefined.

3.5 Operators

3.5.1 assignment

We prefer all our classes to have an assignment operator. Assignment operators should return a reference to `this`, check (avoid) for assignment to self, and make sure that all data members are properly assigned. Also, remember to free allocated resources correctly in assignment operators. If no assignment operator is implemented declare it private in order to avoid implicit usage.

3.6 Books

Many of our ideas about C++ are based on the following books:

- Design Patterns [3]
- Effective C++ [4]
- More Effective C++ [5]
- Effective STL [6]

Use these books as references. Each book contains a list of the items/patterns they contain. Familiarize yourself with these lists and look at them every time you think about the design of something. Most likely they describe the problem you are considering.

4 Preparing the Code for Revision Control

We use *Subversion* for revision control (see [7]). *Subversion* can be set up to substitute keywords in the source files with information to identify the version, author, date, etc. of the file. We use some of these keywords in our code. The advantage with this is that files can be identified even if they are exported from the development environment.

Minimally all implementation modules should begin with a comment line

```
// $Id$
```

that is expanded by a properly set up *Subversion* environment to

```
// $Id: c++_coding_guidelines.tex 46 2008-08-19 13:33:30Z jari $
```

whereas for header files the comment line should be placed after the #ifndef/#define construct as

```
#ifndef symbol
#define symbol
// $Id: c++_coding_guidelines.tex 46 2008-08-19 13:33:30Z jari $
<the complete body of the header file>
#endif
```

References

[1] Robert C. Martin, *Designing Object Oriented C++ Applications Using the Booch Method*, Prentice Hall, 1995

[2] Jari Häkkinen, The C/C++ documentation guidelines[2]

[3] E. Gamma et. al., *Design Patterns*, Addison Wesley Professional, 1995

[4] Scott Meyers, *Effective C++*, 3rd ed., Addison-Wesley Professional, 2005

[5] Scott Meyers, *More Effective C++*, Addison Wesley, 1996

[6] Scott Meyers, *Effective STL*, Addison-Wesley Professional, 2001

[7] Jari Häkkinen, Subversion guidelines[3]

A Indentation in Selected Text Editors

Note, **every** indentation level must be one `tab` each.

A.1 Emacs

For `emacs` I want 2 space indentation, and this is accomplished in the below excerpt from my `.emacs` file. Every indentation level will generate a `tab` character in the file, but displayed as 2 space characters. If you want to change indentation to something else than 2, you must change both `2` characters below.

```
(defun my-c-mode-hook ()
       (setq c-basic-offset 2)
       (setq-default tab-width 2)
       (setq-default indent-tabs-mode t))
(add-hook 'c-mode-hook 'my-c-mode-hook)
(add-hook 'c++-mode-hook 'my-c-mode-hook)
```

[2] http://www.thep.lu.se/~jari/documents/
[3] http://www.thep.lu.se/~jari/documents/

A.2 Kate

In `kate` you set the indentation setting through the graphical user interface. Do the following

1. Choose menu `Settings` → `Configure Kate`.
2. In the tree, choose `Editor` → `Editing`.
3. Set your preferred value in `Tab and indent width`.

A.3 Vi

To get proper display of `tab` in vi (and clones?) you need to add one line in `.vimrc`:

```
set tabstop=2
set sw=2
set ai
```

where you change the 2s to fit your preference. `set ai` turns auto indentation on.

B Operator Precedence Order Summary

Name	Associativity	Operators
Scope resolution	l → r	`::`
Primary	l → r	`() [] . -> dynamic_cast typeid`
Unary	r → l	`++ -- + - ! ~ & * (type_name) sizeof new delete`
Member selection	l → r	`.* ->*`
Multiplicative	l → r	`* / %`
Additive	l → r	`+ -`
Bitwise shift	l → r	`<< >>`
Relational	l → r	`< > <= >=`
Equality	l → r	`== !=`
Bitwise AND	l → r	`&`
Bitwise XOR	l → r	`^`
Bitwise OR	l → r	`|`
Logical AND	l → r	`&&`
Logical OR	l → r	`||`
Conditional	r → l	`? :`
Assignment	r → l	`= += -= *= /= <<= >>= %= &= ^= |=`
Throw exception	l → r	`throw`
Comma	l → r	`,`