C++ Coding Standard

Gwendal Daniel
gwendal.daniel@gmail.com

July 28, 2013

Contents

Abstract

This article presents the guidelines I try to follow on my C++ projects. They are based on my personnal experience in team and individual projects.

The recommendations listed below are a merge between the Geosoft's guidelines [2], the Toff Hoff's coding standard [1], common practices found around projects and my personnal experience.

You are free to copy, modify and reuse this standard as your own.

1 Names

1.1 Classes

- Names representing classes must be in mixed case starting with a upper case : `Line, Item`.
- Acronyms and abbreviations must follow the same rule : `NbcViewer, HtmlHelper`.
- Only acronyms and abbreviation known by everyone should be used, otherwise use the entire word.
- If the class has a parent one don't include the parent name in the class name : use `Book`, not `BookItem` if the parent is `Item`.
- Don't use underscores as word delimiter in class names.
- Class names composed of more than three words may be considered as a design error. In general class names should be composed of up to three words.
- Suffixes are helpful when providing additional information : `ItemManager, EventHandler`.

1.2 Types

- Type names must follow class name rules. Do not add a `_Type` suffix.

1.3 Class Attributes

- Class attribute names must be in mixed case starting with a lower case and should have underscore suffix : `Item* parent_`.
- Class attributes representing a collection must be plural, other attributes must be singular.
- Class attributes representing a collection must not contain the collection type in their names : `childs_` instead of `childList_`.

1.4 Methods

- Method names must be in mixed case starting with a lower case : `findItem()`.
- Acronyms and abbreviations must follow classe rules.
- The name of a method must specify the action performed by the method. Do not hesitate to create relative long names if it is necessary : `checkForErrors()` instead of `errorCheck()`.
- Try to start method names with verbs : it improves code readability.
- As for classes, suffixes are helpful when they provide additional information : `getCount()`.
- Prefixes :
  - `is*` when the method returns a boolean.
  - `get*` for class attribute accessors.
  - `set*` for class attribute modifiers.
- Parameter names must be same as attribute names (without the final underscore) when it is appropriate : `setParent(Item* parent)` if the class attribute is `parent_`.
- Do not repeat the object name in the method name : use `line.getLength()` instead of `line.getLineLength()`.

1.5 Variables (on the stack)

- Variable names must be in lower case and use underscore as word delimiter. This emphasis the scope of the variable by giving it a different style : `local_variable` instead of `localVariable`.

1.6 Pointers

- Do not use prefix in pointer names, the variable name give information about the role of the variable, not about the type : use `Item* first_child` instead of `Item* p_first_child`.
- Append the `*` to the type, not the variable name : `Item* first_child`, not `Item *first_child`.
- Do not declare multiple pointers with same type on a single line, it require a `*` before each variable name and is not very intuitive.

1.7 References

- Apply the same rules as pointer names.

1.8 Booleans

- Do not use negative form in boolean names : use `isError` instead of `isNoError`.
- Use `is` prefix when it is appropriate.

1.9 Global constants

- Global constant names must be all uppercase using underscore as word delimiter.
- In general case implementing the constant as a method is a better choice.

1.10 Enumerations

- Enumeration names must follow class and type rules (mixed case starting with an upper case).
- Enumeration values must follow global constat rules (all uppercase using underscore as word delimiter) and must be prefixed with the enumeration name. The prefix avoids name conflicts.

```
enum Color {
        COLOR_RED,
        COLOR_BLUE
};
```

1.11 Defines and Macros

- Define and macro names must follow global constant rules.

1.12 Exceptions

- Exception class names must follow class rules and must be prepended by `Exception` : `CoreException`.

1.13 Namespaces

- Namespace names must be lower case and should be a single word.

1.14 Files

- Header files must have '.h' extension.
- Implementation files must have '.cpp' extension.
- Header and implementation files must be named as the class they contain : `BookManager.h`. It helps package exploration.

2 Formatting

2.1 Classes

- Sort scopes in header as follow : `public`, `protected` and `private` : a client developper only needs to focus on the top of the file.
- Implement destructors and constructors (copy, default) when necessary. When they are not implemented they should be commented to tell it has been considered, otherwise it could be a forgetting.
- Initialize **all** the class attributes in the constructor.

2.2 Methods

- Method signatures must be on a single line, without the opening brace
- Opening and closing braces must be on a new line.

```
void Book::setAuthor(Author* author)
{
        // method body
}
```

- Methods should not be longer than a page (about 60 lines).

2.3 Conditional instructions

- If a constant is used in a conditional comparison it must be on the left part of the expression. Using this formatting rule the compiler will detect missing '=' in equality tests.
- Do not use complicated conditions, prefer temporary boolean.

2.4 Blocks

- If the block is opened by a conditional instruction the opening brace must be on the same line as the condition. Closing brace must be on a new line.

```
if("bob" == author_) {
       // block body
}
```

2.5 Lines

- Long lines must be splitted consistently and explicitly
  - After ',' or ';'.
  - After an operator.
  - Align new line with the previous one.

2.6 Floats

- Float variables and constants should always be written with decimal point and at least une decimal : it show the developer understands that he is using a float.

3 Documentation

Use a four level documentation : package, class, method and inline. Each one has a different goal / scope :

- Package documentation is a global view of the package fonctionnalities. It doesn't enter into details (but can mention classe names). It is basically a README file at the top of the package and any other specification / high level documentation.
- Class documentation describe at a high level class fonctionnalities. It will be read by client developpers. The documentation must be in the '.h' file before the class name (because client developpers doesn't need to go into the '.cpp' files).
- Method documentation describe the action performed by each method. It will be read by project developpers. The documentation must be in each '.cpp' file (before each method signature) and should at least describe the action performed, the parameters, return and possible exceptions. You can also include pre and post conditions when it gives additional informations.

- Inline documentation is in method's body, it must explain algorithm decisions and tricky part of the code. It will be read by project developers.

4 Process

- Write tests and documentation at the same time as you code.
- Run unit tests of the modified parts before each commit (avoid error propagation).
- Set a bugtracker at the beginning of your project.
- Use a code checker to ensure the producted code is conform to this standard (if it is not verified it won't be done).

5 Miscellaneous

- Add a label to enumerations representing the error case / uninitialized case. If possible put it at the first label position : `COLOR_UNDEFINED`.
- If a type is based on a base c++ type use a `typedef` to create it : `typedef int volume`.
- Exceptions must derive from `std::exception`. You should have one exception per package, if you need more than one create derivated classes from your base exception class to avoid catch breaks in client code.
- Do not use `using namespace ***` in class headers (it forces the client code to use the entire namespace and may create name conflicts).
- You can use `using namespace ***` in class implementation files (being careful of name conflicts).
- Use `nullptr` instead of `NULL` or `0`.
- Type casts must allways be explicit : it shows the developer understand the consequences of the cast : `float_value = static_cast<float>(int_value)`.

References

[1] Todd Hoff. C++ coding standard. http://www.possibility.com/Cpp/CppCodingStandard.html, March 2008. Accessed: 2013-07-20.

[2] Geotechnical Software Services. C++ programming style guidelines. http://geosoft.no/development/cppstyle.html, January 2011. Accessed: 2013-07-25.