
Coding Conventions for C++

Copyright © 2007 Majoron.com

Revision 1.0

Revision History
2007.11.12

The first approved revision

Abstract

This book declare coding conventions and standards for C++ language.

Table of Contents

Variables	1
Naming conventions	1
Naming convention for constants	2
Classes	3
General layout	3
Naming conventions	4
Class fields	4
Class methods	4
Interface definition	4
Namespaces	5
File organization	5
User types (typedef's)	6
Scope definition	6
Try - catch block	6
User exceptions	7
Comments	7
License text	7

Variables

Naming conventions

A variable name should be represent as:

typePrefix *NameOfVariable*

where typePrefix is:

Table 1. Allowed type prefixes

<i>n</i>	Integer (short, int, long e.t.c) variables
<i>f</i>	Floating (float, double) variables
<i>b</i>	Boolean variables
<i>s</i>	STL string (std::string) variables
<i>ws</i>	Wide (Unicode) STL string (std::wstring) variables
<i>sz</i>	Null terminated C-string (char*). <i>Please note! If a variable typed with char* contains a byte buffer instead of null terminated string, type prefix should be "pa"</i>
<i>wsz</i>	Wide (Unicode) null terminated C-string (wchar_t*) variables
<i>c</i>	Character (char) variables
<i>wc</i>	Wide character (wchar_t) variables
<i>a</i>	Arrays ([], std::vector, std::list, std::set, e.t.c)
<i>m</i>	Maps/Dictionaries (std::map, hash_map, and any others collection structured by pair of key/values)
<i>it</i>	Iterators (STL or custom)
<i>o</i>	An object (class instance) in the stack or by C++ reference
<i>p</i>	A pointer prefix. Should be used with other prefix types. E.g.: po is pointer for object, pa - pointer to array
<i>pf</i>	A function pointer
<i>mx</i>	A mutex instance
<i>ev</i>	An event/condition instance (as threads synchronization primitive)
<i>h</i>	An object handle in 3rd-party API which uses this primitive. E.g. Win32 API.

and NameOfVariable is should be nouns, in mixed case with the first letter of each internal word capitalized.

Example 1. Variable declaration example:

```
std::string sName // is string variable
```

Naming convention for constants

The names of variables declared (class or global) constants should be all uppercase with words separated by underscores ("_").

Example 2. Constants declaration example

```
const int MAX_VALUE;  
#define VERSION "0.5"
```

Classes

General layout

There are some general recommendation about layout of typedefs, methods, members and etc inside class. First section is public typedefs, next section is protected and private typedefs. This section also can be located at member's section. Next section is public static consts, next section is protected and private static consts. This section also can be located at member's section. Next section is constructors and destructors. Next section is public, protected, private methods (order is essential). Next section is public, protected and private members (order is essential).

Example 3. A General layout

```
class Test {
public:
    typedef x y;
protected:
    typedef x1 y1;
private:
    typedef x2 y2;

public:
    static const int nX = 10;
protected:
    static const int nX1 = 10;
private:
    static const int nX2 = 10;

public:
    Test();
    ~Test();

public:
    void method();
    void method1();
protected:
    void method2();
    void method3();
protected:
    void method4();
    void method5();

public:
    int x1;
protected:
    static const int nX2 = 10;
    typedef z1 z2;
    int z2;
private:
    static const int nX2 = 10;
    typedef z3 z4;
    int z4;
}
```

Naming conventions

A class name should be nouns, in mixed case with the first letter of each internal word capitalized.

Class fields

Class fields must have an additional prefix *m_*

Example 4. A class field definition

```
class Test {
    std::string m_sName;
public:
}
```

Class methods

A class method name should be start in the lower case and all others letter of each internal word must be capitalized. A class property must be defined as getter and (optionally) setter methods and a private field. The setter method must have *set* prefix, the getter - *get* prefix or *is* for boolean type:

Example 5. Class setters/getters

```
...
void setValue(const std::string&);
const std::string& getValue();

// for boolean getter must have another prefix is
boolean isBooleanValue();
...
```

Interface definition

C++ language doesn't have a special construction for interface declaration, and these conventions declare the following rules:

An interface name must have an additional prefix *I* for class name

Must have defined the virtual destructor

Interface cannot have implemented members or declared fields

Example 6. Example interface definition

```
class IMyInterface {
public:
    virtual ~IMyInterface() {}; // Mandatory declaration
    virtual void mySomeMethod() {}=0; // Custom abstract method

    typedef boost::shared_ptr<IMyInterface> SharedPtr; // Typedef's
}
```

Namespaces

The namespace is always written in all-lowercase ASCII letters.

File organization

For each C++ class must be created separate header and source files. Header must be defined with *.hpp extension for C++, and source with *.cpp. An implementation code (e.g. for template) in the header can be moved to separate inline file with *.inl extension. Each header must have the following format:

```
<LICENSE_DEFINITION>
#ifndef <FILE_NAME>
#define <FILE_NAME>
...
#endif // <FILE_NAME>
<LINE_BREAK>
```

Each source file must have the following format:

```
<LICENSE_DEFINITION>
#include "<FILE_NAME>"
```

The system header includes (or includes for 3rd-party libraries) must be first and specified by <...> before user includes specified by "...".

Example 7. Definition system and user includes

```
...
#include <string>
#include <boost/shared_ptr.hpp>
#include "MyClass.hpp"
...
```

User types (typedef's)

An user type name should be nouns, in mixed case with the first letter of each internal word capitalized. When using *STL-containers* must be defined an user type for specialisation for simple ability to change a type of container in the future. *STL iterators* must be defined as `UserType::iterator` where `UserType` is *STL-container* typedef. When using "*shared ptr concept*" (e.g. for `boost::shared_ptr`) typedef for class must be specified inside the current class definition.

Example 8. Typedef usage

```
class MySuperClass {
    typedef std::list<MyItem> MyItems;
    MyItems::iterator itMyItemPos;
public:
    typedef boost::shared_ptr<MySuperClass> SharedPtr;
}
```

Scope definition

We are using the Java-style for definition scopes:

```
statement {
}
```

Try - catch block

We are using two types of the try-catch block:

```
try{
} catch(...){
}
```

```
try{
}
catch(...){
}
```

User exceptions

User exception class must be inherited from `std::exception` (or from `std::logic_error`).

Comments

Comments must be entered in the English language and must be defined with doxygen/javadoc style.

For each new interface doxy-comments are required.

For each class that haven't a parent or an implemented interface doxy-comments are required

```
/**
 * <Class/interface description>
 * ...
 */
class MyClass {
    ...
}
```

Allowed to use `@todo`, `@note` and any other needed doxy-command (if necessary).

All doxy-command must be begin from '@'.

License text

Each header and source file must have the appropriate license. Below are currently available license texts.

Here is license text for Majoron projects.

```
/*
Copyright 2007 Majoron.com (developers@majoron.com)
Original sources are available at www.majoron.com
```

```
Licensed under the Apache License, Version 2.0 (the "License");
```

you may not use this file except in compliance with the License.
You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

*/