# Neverending ODC
# Coding Conventions 1.3.0

Holger Dammertz

January 29, 2003

**Abstract**

This are the coding conventions for Neverending ODC. They have to be applied to all C++ code written for the client and server of the project. If you find inconsistencies, errors or have some further additions, please contact me.

## Contents



## 1 General

- Every time a rule is broken, this must be clearly documented.
- Use Unix line endings for text files (”LF” as line ending) and not DOS line endings (”CR-LF” line endings). So be carefull when developing under Windows!
- There should be only one statement per line unless the statements are very closely related. (i.e. the initialization of a 3D vector could be in one line)
- Methods should limit themselves to a single page of code. Sometimes it is inevitable to create large methods but usually there are some tasks in such a method that could be put in an extra method. (tends to result in better code sharing and easier debugging)
- Don’t #define macros or constants, use inline methods and const.
- Usually avoid embedded assignments (things like while (i++ != 0)).
- Each class should have it’s own .h and .cpp file (and these are the only allowed extensions). Templates and fully abstract classes don’t need a corresponding .cpp file, of course.

- Always implement an abstract method and trigger an error if it is not an abstract destructor.
- You shall always initialize variables. Always. Every time. Classes with multiple constructors and/or multiple attributes should define a private initMembers() method to initialize all attributes.
- The constructor of a ”big” class[1] should never rely on the success of another function call → only initialize member variables and set pointers to NULL. Instead define a public init() method that prepares the object for its use (and manages errors). An exception to this rule is, that you are allowed to create STL containers in a constructor (or before).
- For each member variable there should be a getXXX() and a setXXX(val) method. If it is a bool, there should also be a isXXX() method. Generally try to apply the information hiding paradigm and make as few parameters as possible public.
- Never use the old C-style casts, instead use the explicit cast syntax that C++ provides:

| | |
|---|---|
| **static_cast** | For ”well-behaved” and ”reasonably well-behaved” casts, including things you might now do without a cast (such as an automatic type conversion). |
| **const_cast** | To cast away const and/or volatile. (You generally should not do this) |
| **reinterpret_cast** | To cast to a completely different meaning. The key is that you’ll need to cast back to the original type to use it safely. The type you cast to is typically used only for bit twiddling or some other mysterious purpose. This is the most dangerous of all the casts. |
| **dynamic_cast** | For type-safe downcasting. |

Table 1: C++ casts [5]

- Each class has to be subclassed from the overall base class **OdcObject**. This will become very handy while debugging the whole project.
- All data stored in files or send through the network has to be big endian.

# 2 Formatting Conventions

- Use one tab per indentation level (and use an editor that does not replace tabs). Use only tabs at the beginning of a line and not to indent comments.
- No spaces after an open or before a closing bracket.
- One space after a comma, an “if” or an “switch”:

```
if ((1 == count) || (5 == calcMax(7, 19)))
{
 ...
}
```

[1] This is a class where normally only one instance will be created

- Lines should not exceed 78 characters with a tab width of 2. (not only because it's easier to get the code on screen, but because of printing the source code (and diff's) gets much easier).
- Only one variable declaration per line (except they are no pointers and they have really very much in common)
- The '&' and '$*$' tokens should be adjacent to the type, not the name:
  int* vertexIndices;
- Each "if" statement has to be bracketed with no exception.
- ```
  if (condition)          // Comment
  {
  }
  else if (condition)     // Comment
  {
  }
  else                    // Comment
  {
  }
  ```
- 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.

  ```
  switch (...)
  {
      case 1:
      ...
      // FALL THROUGH
      case 2:
      {
          int v;
          ...
      }
      break;

      default:
        // Error?
   }
  ```
- The class-member declaration in the .h files has to use the following template:

  ```
  /**
   * Documentation (see documentation section)
   */
  class ClassName
  {
    public:
      /** all public variable declarations (static variables at the
       *  beginning) */
      ...

    protected:
      /** all protected variable declarations */
  ```
  ```

```
    ...

  private:
    /** all private variable declarations */
    ...

  //------------------------ [ Methods ] -------------------------

  public:
    ...

  protected:
    ...

  private:
    ...
}
```

- The implementation of the methods in the .cpp files has to be in the same order as the declaration in the .h file but with larger seperators:

```
/*************************************************************
 *               [Public/Protected/Private] Methods
 *************************************************************/
```

- The method implementations in the .cpp file should be seperated by at least two newlines.
- For the sake of clarity there should be separators in large code files. These separators can optional include a short description:

```
//-------------- [short description or section name] ------------
```

There will be an astyle[2] configuration file, that formats the code appropriately. This should be used before committing large chunks of code.

In the doc folder of the client project directory you will find two template files (for .cpp and .h) that can be copy-pasted in your new source file and contain all described seperators and the order of all parts.

# 3 Naming Conventions

## 3.1 General

- Abbreviations should be written with only the first letter uppercase: GuiManager (not GUIManager). Only use abbreviations when they are commonly used and known by everyone.

## 3.2 Class Naming

- The first character in the name is upper case
- Upper case letters are used as word separators (no underscores)

[2] astyle (Artistic Style) is a automatic code beautifier: http://astyle.sourceforge.net/

## 3.3 Method Naming

- The first character in the name is lower case
- Upper case letters are used as word separators (no underscores)

## 3.4 Member Variable Naming

- Generally prefer long variable names.
- Each variable starts with a prefix (describing access type) and then the same rules as for method naming apply starting with a capital letter:

```
Access Prefix:
 m_  before the name of a member variable
 ms_ before the name of a static variable
 g_  for a global variable (but there should not be much of them)
 gs_ for a global static variable
 f_  for a funciton parameter

Use a second prefix for a pointer or reference:
 p for a pointer
 r for a reference
```

The access prefix enables you to directly see whether a member variable or a local variable is used in the sourcecode:

```
...
void ObjectRenderer::Render(Object3D* f_pObject)
{
    m_pMesh = f_pObject->getGMesh();

    f_pObject->m_Rotation->generateMatrix4x4(m_pRotMatrix);
    m_pTextureManager->bindTexture(f_pObject->getTextureID());
    ...
```

Only local variables have no prefix and start with a lower case letter.

## 3.5 Singleton classes

- Classes which only have one instance during runtime and which are accessed by multiple other classes should be made singleton.
- Provide a method called *instance()* in your class: This method returns the pointer to your class instance, so that other objects can access it.
- You should also provide a method *initInstance()* in your class. This method is used to initialize the single instance of your class.
- Furthermore, you should provide a method *destroyInstance()* in your class. This method is used to destroy the single instance of your class.
- The destructor and constructor of the singleton class should be made protected or private, so instantiation and destruction is only possible through the singleton methods.

# 4 Documentation

- All files must follow the format of the standardized header which can be found in: doc/(h/cpp)_file_template.txt (there are also the separators described in section 2).
- All comments are to be written in English.
- Classes are documented in their header files.
- Document every method (in-depth and comprehensive) and every important member variable (short but also comprehensive) using the javadoc format (we use Doxygen[3] to generate the documentation and since version 0.4 it fully supports JavaDoc)

```
/** this is a really important variable */
int  m_MaxBlah;
/** this is also a really important variable, that has some more
 * important documentation so it needs more than one line */
float m_MinScale;

/**
 * The first sentece should summerize the use of the method.
 * More documentation follows.
 *
 * @param description of the parameters
 * @return description of the return value
 */
void doSthImportant();
```

- Every class should have a good documentation (how and why it is used):

```
/**
 * The first sentece should summerize the use of the class.
 * More documentation follows.
 *
 * @author Your Name
 * [@version x.x] (Only when the class is important to many other
 *         people)
 */
class Mesh3D...
```

- You can also include html tags in your documentation. For example you can put code in comments between a $< code >$ ... $< /code >$ and doxygen will display it with another font.
- single line comments (in the code) should be written in this form:

```
/* comment */
some code...

multiple line comments this way:
/* comment
   more comments
   and even more */
some code...
```

[3] http://www.stack.nl/ dimitri/doxygen/

- when you use this style of documentation you cannot comment out multiple lines of code with just a /*....*/ but instead you can do the following:

```
void SomeClass::Example()
{
 great looking code

 #if 0
 .
 lots of code
 .
 #endif

 more code
}
```

# 5 In-Development comments (Gotchas)

After each gotcha keyword, the current date could appear:
!!TODO: 2002/08/27 description of task

## 5.1 Gotcha keywords

**!!TODO: topic** Means there is more to do here that you should not forget.

**!!BUG: description** There is a known bug but either you don't how to fix it or you have sth. more important to do.

**!!UGLY:** When you've done something ugly say so and explain how you would do it differently next time if you had more time.

**!!OPT:** You know, that sth. could be done significantly faster but have no time to do it yet; so explain here what could be optimized, and why it is better/faster.

**!!FIXME:** The primordial gotcha (similar to !!TODO: but the task should be done quite fast)

**!!TRICKY:** The following code was done a bit tricky and you should explain why you have not done it simpler.

The keywords are marked special so you can find them eaysily by searching for !! in your source files. It is also possible to create a tool that automatically generates a report.

# References

[1] C/C++ Kompendium - Dirk Louis - 1998 Markt und Technik, Buch und Software-Verl. ISBN 3-8272-5386-1

[2] C++ Coding Standard - Tedd Hoff - 2000 http://www.cs.umd.edu/users/cml/cstyle/CppCodingStandard.html

[3] Programming in C++, Rules and Recommendations - FN/Mats Henricson and Erik Nyquist - 1992 http://www.cs.umd.edu/users/cml/cstyle/Ellemtel-rules.html

[4] OGRE Coding Standards - http://ogre.sourceforge.net

[5] Thinking in C++, 2nd ed. Volume 1 - Bruce Eckel - 2000
http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html