Programming Guidelines

Author: Martin Ecker
Project Website: http://xengine.sourceforge.net
Last Modified: 18. September 2003 22.56

This paper describes the programming guidelines and other conventions used in the source code of XEngine. Since XEngine was supposed to become an open source project from the beginning and is with its currently more than 100,000 lines of code and comments an already rather large project, having programming guidelines right from the start is good practice. Additionally, this document contains a number of best practices used throughout the code. Code contributions to XEngine should adhere to the guidelines and practices outlined in this document as much as possible.

1.1 Blocks, Indentation, and Spaces

At the start of a new block all the following lines of the block must be indented by one tab. One tab is defined to be 3 spaces, but the tab is not replaced by spaces. Note that this is not the default tab width in most text editors and development environments, where a tab is usually 4 or 8 spaces. The choice of 3 spaces per tab is historical and can be considered to be a bit unfortunate from today's point of view, but XEngine's source code base is now too large to change it to e.g. 4 spaces.

The opening and closing brackets for a block are always in their own line, that is, the line does usually not contain any one-line comments. The following listing gives an example of blocks and correct indentation:

```
void SomeFunction()
{
   int x = 0;

   for (int i = 0; i < 100; i++)
   {
      printf("This is a test %d", i);
      printf(" and this finishes the line\n");
   }
```

```
    if (x > 5)
      printf("x is greater than 5\n");
}
```

If a block consists of only one line, as can be seen in the if-statement in the above piece of code, the block opening and closing brackets can be omitted. However, that single-line block must be in its own line and cannot follow immediately after the if-statement (or any other statement for that matter) in the same line.

After the keywords `for`, `while`, `if`, `catch` and `switch` there is always a space before the opening brackets. Between a function name and the corresponding opening bracket, however, there is never a space. When overloading operators, there is always a space between the `operator` keyword and the operator symbol being overloaded. This also applies to the function call, `operator ()`.Before and after an operator there is always a space. Exceptions to this rule are the following operators where there is no space:

!, ~, ++, --, - (unary negation operator)

Following are some examples of correct bracket placement:

```
void DoSomething();  // no space between function name and brackets
x = -var + (4 * var) / 2 * 5;  // spaces between operators
void operator () (const Object& o);
```

All source files must be terminated with a new line. This is actually a requirement of the ANSI C++ standard, even though most compilers ignore it. However gcc gives a warning if a file is not terminated with a new line, and therefore all source code files must be terminated with a new line to avoid such warnings.

1.2 Naming Conventions

All names should concisely express what they describe. In most cases, it is not permissible to have an input parameter to a method simply called "x". There are, however, exceptions to this rule. Very often used names, such as the well-known counter variable name "i", can be short and just one-letter names, as long as it is obvious from the context what the name means. A typical example of this in XEngine is:

```
Vector3 Multiply(const Matrix3x3& m, const Vector3& v);
```

In this function prototype it is obvious what the input parameter names "m" and "v" stand for.

All class, structure, function, and namespace names must start with a capital letter. All variable names must start with a non-capital letter. In names that consist of more than one word, each word after the first word must begin with a capital letter. This notation is sometimes referred to as the camel hump notation. The first word begins with either a capital or a non-capital letter depending on what kind of name it is. Variables begin with a non-capital letter, for example. Underscores should not be used to separate words, but can be used to separate groups of words if a name is rather long.

For variable names there are some additional name prefixes that indicate in which scope the variable is valid. A global variable is prefixed with "g_", a static member variable is prefixed with "sm_", a member variable is prefixed with "m_", and a static variable inside a function is prefixed with "s_". Note that this has nothing to do with the so-called Hungarian Notation that encodes the type of the variable into the name and is often used by Microsoft in Windows C programs. The XEngine source code does not use Hungarian Notation except for the special case of pointer variables. Pointer variables are usually prefixed with the letter "p", which stands for pointer. Sometimes, in particular when the pointer variable represents an array, this is omitted.

In the declaration of pointer variables, the * symbol always follows right after the type without any white spaces in between because the * symbol is logically considered to belong to the type. The same goes for reference variables where the & symbol always follows immediately after the reference type. For example:

```
int* pSomewhere;
int& someVar.
```

The following example that shows a header file for a class illustrates all the naming conventions introduced above. It also shows how to correctly use indentation when writing a class. Note how the initialization list of the constructor is indented by a tab before the colon followed by yet another tab.

```
#ifndef XCounter_INCLUDED
#define XCounter_INCLUDED

namespace XEngine
{

class Counter
{
public:
   Counter(int initialValue)
      :   m_counter(initialValue)
   {}

   int GetCounterValue() const { return m_counter; }
   int IncreaseCounter(int value)
   {
      static int s_timesCalled = 0;
      m_counter += value;
      ++s_timesCalled;
   }

   int DecreaseCounter(int value);

   //... some more functions follow

private:
  int m_counter;
  static int sm_counterMaxValue;
};
```

```
}    // namespace XEngine

#include "XCounter.inl"

#endif    // XCounter_INCLUDED
```

Usually every header file only defines one single class. There are exceptions to this, however, when there is a nested class or when a small simple data class is needed.

Every header file must have a so-called header guard to prevent multiple inclusions of the header file during the compilation process and thus also multiple definitions of classes and function prototypes. The header guard pre-processor variables used always have the form FileName_INCLUDED, where FileName is replaced with the file name of the header file without the file extension. Note how the #endif that closes the header guard in the above listing has a comment stating the guard pre-processor variable name and also how the namespace block closing bracket has a comment stating the namespace to which the closing bracket belongs.

For short inlined functions, like GetCounterValue above, it is permissible to write the entire function in one single line. However, it is advisable to always put inlined functions are in a separate file with the file extension ".inl". These files get included into the header file at the very end after a potential namespace is closed and before the header guard is closed, as shown in the above listing.

The naming convention for pre-processor defined variables used by XEngine are slightly different from what has been discussed so far. All pre-processor variables begin with "XEngine_", and the following name of the variable begins with a capital letter. Unlike many other programming guidelines, the name of a pre-processor defined variable does not contain only capital letters. Multi-word pre-processor variable names are separated by an underscore, and again every word begins with a capital letter. This is an example of a pre-processor variable in XEngine:

```
#define XEngine_Precompiled_Headers
```

1.3 Namespaces

All classes, data types, variables, etc. of XEngine must be contained in a namespace called "XEngine". Using subnamespaces is allowed, and in fact all classes of XEngine-Math are contained in a subnamsepace called "Math", all XEngineCore classes in a namespace called "Core" with the renderer library classes contained in yet another sub-namespace called "Renderers". All scene graph-related classes are in the subnamespace "Scene".

A namespace scope is opened in a header file after the inclusion of other header files. The namespace scope is closed before a potential inline file is included and before the final #endif of the header guard with a comment after the closing bracket with the name of the now closed namespace. In inline files the namespace scope opening brackets are typically the first source code line in the file, unless additional header files must be included which then come first. In a source file, a ".cpp" file, the namespace scope open-

ing brackets also follow immediately after the inclusion of header files. Following is an exemplary header and source file that follow these guidelines:

```
// header file
#ifndef XSomeClass_INCLUDED
#define XSomeClass_INCLUDED

#include <map>
#include <vector>

namespace XEngine
{
namespace Core
{

...class declarations go here...

}   // namespace Core
}   // namespace XEngine

#include "XSomeClass.inl"

#endif   // XSomeClass_INCLUDED

// source file
#include <boost/shared_ptr.hpp>
#include "XSomeClass.h"

namespace XEngine
{
namespace Core
{

...definitions go here...

}   // namespace Core
}   // namespace XEngine
```

1.4 File Naming Conventions

Each source code file in the XEngine source code carries the name of the class it contains and its name is prefixed with the letter "X", which is mostly for historical reasons. Class names, and therefore also file names, are unique. A class declaration must always be in a header file with the file extension ".h", and the corresponding class definition in an implementation file with the file extension ".cpp". If there are inlined functions, these will be put in a file with the extension ".inl". A class "File" would therefore be declared in a file called "XFile.h" and implemented in a file called "XFile.cpp" with possible inlined functions put in a file called "XFile.inl".

1.5 Classes and Structures

Class declarations have the access keywords, `public`, `protected`, and `private`, vertically aligned with the `class` keyword and the opening bracket for the class declaration. The order of member declarations is typically as follows, with an empty line separating them.

- Public member functions, starting with public constructors and the destructor, if it is public.
- Protected member functions
- Private member functions
- Protected member variables
- Private member variables

The astute reader will notice that the above list is missing public member variables. In XEngine, public member variables should not be used. Instead, Get- and Set- member functions should be provided to access protected or private member variables, even if these are just simple one-liners.

When a method is declared virtual in a base class, all methods in subclasses that override that method must also have the `virtual` keyword in their declaration, even though this is not required by the C++ language. That way it is not necessary to look up the entire class hierarchy to find out if a method uses dynamic binding.

1.6 Commenting

Commenting of code is essential for later understanding and maintenance and is thus mandatory in XEngine's source code. However, it is not good to over-comment code. Only passages that cannot be immediately understood by looking at the code should be commented. Complicated calculations and operations should always be commented by adding a multi-line comment block with detailed descriptions.

XEngine uses the documentation system doxygen to automatically generate an API reference documentation from the source code. Refer to the doxygen manual for more information on how to use it. Every public class, function, and variable has to be documented by using doxygen comment blocks. The following list summarizes the suggested guidelines for using doxygen comments:

- All namespaces must have a brief and detailed description.
- All classes that belong to the public interface must have a brief and detailed description that is usually put right in front of the class declaration in the header file. The brief descriptions must be specified by using the \brief tag (because of a bug in doxygen).

- All member functions that are part of the public interface of a class must have a brief description in the header file. This can either be put right in front of the method declaration or right after it on the same line with a //!< comment. A detailed description, if applicable, must be put just before the method definition in the implementation file.
- The documentation for public member variables or static class variables can only contain a brief description and need not have a detailed description. Brief and detailed descriptions are either just before the declaration or on the same line right after the declaration of the variable. Note that public member variables should not be used in general.

Additionally, every source file must have a header like the following right at the beginning:

```
//////////////////////////////////////////////////////////////
// Author:      Name list
// Modified by: Name list
// Copyright:   Name list
// CVS-ID:      $Id$
// License:     XEngine license
//////////////////////////////////////////////////////////////
```

1.7 Use of Debug Helpers

To facilitate debugging the use of assert macros is highly advisable. Use of the wxWindows debug macros (`wxFAIL`, `wxASSERT`, etc.) is encouraged, for example, to check invariants or the validity of input parameters. This does not impede runtime efficiency of release builds, since the macros are inactive when a release version is built, but greatly helps reduce the time necessary to find programmer mistakes and bugs in debug builds.

1.8 Best Practices

This section lists a number of best practices in random order.

- Use declarative names as much as possible, even if this forces long names.
- Use abbreviations in names judiciously. Not everyone might be familiar with a particular abbreviation, even though the meaning is clear to you.
- Write explicit code that does not require implicit knowledge.
- Write concise but readable code.
- Write const-correct code.
- Do not use C-style casts. Use static_cast, reinterpret_cast, and const_cast instead. C-style casts are a relict of old times and there is no reason to use them in C++ programs (except for programmers being lazy and not being able to let go of old habits ;-).
- Pass values of fundamental types per value and instances of structures and classes by constant reference to functions. Return const references to member variables from member functions.

- Be familiar with and use STL containers. They are reliable, well-tested, typically fast (use the non-standard hash_map and hash_set, if applicable), and are well-designed.
- Be familiar with and use STL algorithms for working with containers. Learn how to use boost::bind, boost::function, or boost::lambda to write complicated transformations on containers as one-liners.
- Never put a using declaration or using directive, such as `using namespace std`, in a header or inline file at global or namespace scope. Only use them in a source file, if necessary.
- Put using declarations or directives - should you need them - only inside the classes or functions you want to use them for.
- Explicitly qualify STL and Boost library identifiers. That is, use `std::vector` instead of a using declaration and then using the unqualified identifier `vector`.
- Prefer `++it` over `it++` when using STL iterators, unless, of course, you really need the unincremented copy returned by `it++`.
- Generally prefer `++i` for all types, not only iterators.
- Be wary of std::auto_ptr. In most cases you will want to use boost::shared_ptr or boost::scoped_ptr instead.
- Pass STL iterators by value.
- Instead of passing const references to STL containers, prefer to pass a templated (first, last) pair of iterators. This allows the client of a function to pass any STL container to the function as it sees fit.
- Use the `explicit` keyword for constructors that potentially only take one parameter to prevent automatic conversions (note that this includes constructors with more parameters where all but the first parameters use default arguments).
- Instead of hard-coding event callback function prototypes, use boost::function to be able to use functors, free or static functions, and member functions as callbacks.
- Read Scott Meyer's Effective C++ book series.
- Read Herb Sutter's Exceptional C++ book series (most of it also available online on his Guru of the Week webpage, http://www.gotw.ca/gotw/index.htm).
- Read Andrei Alexandrescu's Modern C++ Design book.