C / C++ Coding Standards

Introduction

This document is meant as a guide.
The following are not unbreakable “rules”. They can be bypassed, but the guidelines are for good reason, and a better reason must be thought out before bypassing a guideline.

Different workplaces will use different standards that you will need to conform to, but the standards enclosed within this document are some of the more common standards that most places will adhere to.

Be a professional programmer, not a hacker!

Even if you’re in a rush, don’t try to save time by skipping these guidelines. Some might take a bit of time to adhere to, but always keep the long term goal in mind: you want to write clean, maintainable, error-free code, not a half-rate hack job that you will forget how it works a month later, or have co-workers unable to read or use your code.

Files

Naming

Files should be named accordingly, describing the module they are implementing.
In C++ it is suggested to name the .h / .cpp / .inl after the Class it implements. In C it is suggested the names reflect the functions contained therein.
Do not use spaces in names. Only make use of uppercase letters, lowercase letters, numbers and the underscore (_). Windows filenames are not case sensitive, but Unix is, so be sure to use the same case for filenames as you do #include statements. Cross-compiling to Linux may be desirable for code (such as for PS3/PSP/Wii), so using the same case sensitivity can save you headaches in the future.

Extensions

Always use the .cpp extension for source files, unless your program is strictly C code, in which case make use of .c instead.
Always use the .h extension for header files.
The .inl extension may be used if you wish to split inline functions into a separate file. These may then be included at the bottom of its corresponding .h file, before the closing include guard.

Headers

All files should have a commented header at the top of the file, in a similar format to below:

```
Example:
1. ////////////////////////////////////////////////////////
2. // File:           <filename>
3. // Author:         <your name>
4. // Date Created:   <date>
5. // Brief:          <description of the purpose of the file>
6. ////////////////////////////////////////////////////////
```

The header must contain at a minimum the author of the file and date created. A short descriptive brief can also be extremely useful. Optionally a “last edited by”, “last edited date”, and “last edit brief” can aid in tracking down recent edits, but you should try and keep the header short and simple.

Include Guards

Include guards **must always** be used.

The preferred format is:

```
Example:
1. #ifndef _FILENAMEASUPPERCASE_H_
2. #define _FILENAMEASUPPERCASE_H_
3.
4. // code
5.
6. #endif // _FILENAMEASUPPERCASE_H_
```

Avoid using **#pragma once** as it is not an industry standard, and not all compilers support it (GCC has deprecated it for example).

Include This.h at the top of This.cpp

The very first statement inside a .cpp, after the header comment ,should be #include “This.h” before all other statements, for a .cpp called “This.cpp”, as an example.

Headers Requiring other Headers

Header files should be stand alone, and not require other headers to be included before them. The header should contain everything it needs to work.

#include <> vs #include “”

Only use #include <> for Standard C++ headers as the <> searches system paths for files. Use #include “” for all other files (your own headers, 3rd party libs) as they pertain to relative locations.

Variables

The following guidelines relate to variable naming.

Naming

Variables should be named using Hungarian notation, and should be named descriptively, starting with a lowercase prefix.

```
Example:
1. int iMyAge;
2. float fDistanceToTarget;
3. unsigned int uiIndex;
```

Names should not contain underscores, and each word start with an uppercase.

Hungarian prefix’s for local variable types are as follows:

Type	Prefix	Example
int	i	iMyAge
float	f	fTimer
bool	b	bIsAlive
class	o	oMyCar

Type	Prefix	Example
unsigned int	ui	uiMonth
double	d	dDistance
char	c	cInitial
char* null-terminated string	sz	szName

Pointers, References, and Arrays of variables have the following additional prefix that goes before the local variable type prefix:

Type	Prefix	Example
Pointer (*)	P	pfFloatPointer
Reference (&)	r	rfFloatReference
Array ([])	a	aiArrayOfInts

Global Variables, Const Variables, and Static Variables

In addition to the previous prefix's, the following should be used, and can be combined:

Type	Prefix	Example
Global	g_	g_fGlobalTimer
Const	c_	c_uiMaxLife
Static	s_	s_iNextID
Static Const	sc_	sc_uiScreenWidth

Functions

The following guidelines relate to functions.

Naming

Functions should be named descriptively.
They should not contain underscores, and each word starting with an uppercase letter.
Functions that get a value should also start with the word 'Get'.
Functions that set a value should also start with the word 'Set'.

```
Example:
1. int GetScore();
2. void SetHealth(float a_fHealth);
3. void FireLaser();
```

Function Prototypes

Functions should always have a pre-declared prototype.
Global functions should have their prototype within a .h and the definition within a .cpp file.
Functions local to a .cpp should still have a prototype at the top of the .cpp after any includes, and the definition later in the .cpp.

Class declarations should only ever contain function prototypes, and the definition should always be located in a separate file. More on this within the class section.

Line Count

Functions should try to remain below 100 lines in size, so that the entirety of the function can be seen on a screen of common size.

Functionality

A single function should have a single task it performs, not multiple tasks.
If a code block is commonly repeated throughout a program it should be made to be a stand-alone function that can be re-used instead of retyping the same code block.

Inlines

Inline functions must not surpass 3 lines in length, or contain any loops or additional function calls.
Inline functions within a class should have their prototype within the class declaration, but the definition of the function must be located outside the class declaration, either below the class or within an included .inl file.
Example:

```
Example: Test.h
1. class Test
2. {
3. public:
4.    int GetScore();
5. };
6.
7. // either the following...
8. inline int Test::GetScore() const
9. {
10.   // iScore is just an example!
11.   int iScore = 0;
12.   return iScore;
13.}
14.// or...
15.#include "Test.inl"
```

```
Example: Test.inl
1. inline int Test::GetScore() const
2. {
3.    // iScore is just an example!
4.    int iScore = 0;
5.    return iScore;
6. }
```

Arguments

Function parameters names must always be prefixed with an 'a_', then follow standard Hungarian naming for the variable type. For example:

Type	Prefix	Example
int	a_i	a_iHealth
float&	a_rf	a_rfDistance
Object *	a_p	a_pPlayer

Complex objects such as classes and structs should **always** be passed in as a reference to a function, unless using a pointer instead, to avoid constructing new objects and miss-using copy constructors.

Pass all built-in variables by value, unless the function modifies them, in which case use a reference.

Default Arguments and Function Overloading

These should generally be avoided, as they can lead to logic errors.

Functions without Parameters

Do not use void as the argument list for functions of this sort, just use empty brackets.

Parameters

For functions with large argument lists you should split the arguments onto separate lines, as shown:

```
Example:
1. void AStarSeach( AStarNode* a_pStartNode,
2.                  AStarNode* a_pEndNode,
3.                  std::vector<AStarNode*> a_apPath)
4. {
5.    // calculate path
6. }
```

Function prototypes should always contain the argument's name along with the parameter's type. Never leave a function prototype just describing the variable type without a name!

Function Pointers

Function pointers should always be typedef'd to reduce chances of user error when retyping, and to improve readability.

```
Example:
1. typedef void (*EventProcessor)(void* a_pUserData);
2. EventProcessor fpProcessor = &ProcessOnCollision;
```

Layout

The following guidelines pertain to general code layout and design.

Code Blocks

Blocks of code should always be defined within a brace pair. This includes:

- If statements (including single line states)
- For / While / Do While loops

If statements should always have a brace pair, even when the statement is a single line!
Braces should always be located on their own line, with the exception of an empty code block that may have both braces in the brace pair '{}' on the same line.

Indentation

Code blocks should always be indented with tab stops, width 4, in addition to having brace pairs. This helps with readability.

```
Example:
1. // block 1
2. {
3.    // block 2
4.    while ( true )
5.    {
6.          // stuff
7.    }
4. }
```

Statements and Lines

All statements should be located on their own line.
This includes single line if statements. Do not have the comparison on the same line as the statement!
Same goes for all other statements. They must all have their own line.
Lines should not exceed 80 columns.

```
Bad Example:
1. if (m_iAmmo > 0) Shoot();
2.
3. m_iAmmo = 0; m_iScore++;
4.
5.
6.
7.
```

```
Correct Example:
1. if (m_iAmmo > 0)
2. {
3.    Shoot();
4. }
5.
6. m_iAmmo = 0;
7. m_iScore++;
```

For Loops

For loops that use an iterator (be it a class iterator, pointer, or integer) should use pre-increment (++i) instead of post-increment. This is a very minor optimisation that is very easy to get into the habit of doing.

```
Example:
1. for ( int i = 0 ; i < m_iPlayerCount ; ++i )
2. {
3. }
4. for ( ; oIterator < oEnd ; ++oIterator )
5. {
6. }
```

Switch Statements

Case statements should be on the same tab stop as the switch statement, with each case having its own code block and brace pair. Always use a break on each statement, unless you do desire it to fall through, in which case have a comment stating as such.

An exception is very simple switch statements, which may have the statement on the same line as the case statement, with the break statements aligned.

```
Example:
01. switch ( iState )
02. {
03. case 0:
04.   {
05.         break;
06.   }
07. case 1:
08.   {
09.         // fall through
10.   }
11. case 2:
12.   {
13.         break;
14.   }
15. default:
16.   {
17.         break;
18.   }
19. };
20.
```

```
Example:
01. switch ( iState )
02. {
03. case 0: DoActionA(); break;
04. case 1: // fall through
05. case 2: DoActionB(); break;
06. default: break;
07. };
```

Always finish with a default case!

While Loops and Do While Loops

While and Do While loops should both be written similar to If statements; the condition on a separate line to the statement, making use of a code block using a brace pair.

```
Example:
1. while ( bAlive )
2. {
3.    // statement
4. }
5.
```

```
Example:
1. do
2. {
3.    // statement
4. }
5. while ( bAlive );
```

GoTo Statements

Do not use goto, as it is considered bad coding structure, and its functionality can often be achieved through correct flow control and functions.

"If it's complex enough to consider a goto, it's complex enough to encapsulate in a function and avoid the goto" – John Pirie (http://stackoverflow.com/questions/1024361/is-using-goto-a-legitimate-way-to-break-out-of-two-loops/1024395)

Classes

The following guidelines relate to C++ classes.

Always write class declarations for users of the class. Never try to make your implementation job easier at the expense of making the user's job of understanding the class harder.

Naming

Class names should be descriptive of the class' behaviour and use.
Names should not contain underscores, and each word should start with an uppercase letter.

```
Example:
1. class Player;
2. class Vector2D;
3. class CameraManager;
```

Layout

Class members should be grouped in the following way, with each group separated by a blank line:

- Friend declarations (private friends only)
- Typedefs and types
- Constructor / Destructor / Copy Constructor / Assignment Operator
- Functions
- Variables (all should be private)

Protection levels should be on the same tabstop as the class keyword, with each protection level tabbed across one tabstop further, and with public section first, followed by protected, then private.

```
Example:
1. class Test
2. {
3.    // friend declarations within private block
4. public:
5.    // typedefs such as containers, either public/private/protected
6.    // constructors / destructor / assignment operator
7.    // public members
8. protected:
9.    // protected members
10.private:
11.   // private members
12.};
```

Constructors and Destructors

If a class has either a:

- Destructor
- Copy Constructor
- Assignment Operator

Then it should have all 3 of them.

If a class is not meant to be able to be copied, then the Copy Constructor and Assignment Operator should both exist as prototypes only, made to be private.

Member Functions

A class declaration (within the .h) should only ever include member function propertypes, not the definition (body) of the function. The body of the function should always be located within a .cpp, unless inline, in which case it should be located either after the class declaration within the .h, or within a .inl file included in the .h after the class declaration.

Virtual Functions

If a class inherits from another class or can be derived from, then it should always have a virtual destructor.

If a class has any virtual functions then it should also have a virtual destructor as the above statement must stand true since the class is intended for inheritance.

Member Variables

Member variables should never be public. The only exception is for classes with obvious data members (such as a 2D position class or a pair), which can have public members rather than trivial accessor functions.

Member variables also contain an additional prefix in addition to the variable type prefix, and that is 'm_'.

```
Example:
1. class Test
2. {
3. public:
4.
5.    Test();
6.    ~Test();
7.
8. private:
9.
10.   float                    m_fX, m_fY;
11.   static unsigned int      sm_uiReferences;
12.};
```

Const Correctness

If a function does not modify the internal state of the object then it must be declared as const.

```
Example:
1. int     GetStrength() const;
2. void    GetXY(float& a_rfX, float& a_rfY) const;
```

Enumerations

Enumerations should be given descriptive names.
They should also be in uppercase.
The constants within the enumeration should contain the enumeration typename, followed by an underscore and the identifier's name.
Identifiers should also be commented beside the value if they require more description.
The last identifier should be an identifier to mark how many identifiers exist within the enumeration.

```
Example:
1. enum PLAYER_STATE
2. {
3.    PLAYER_STATE_IDLE,    /* comment here */
4.    PLAYER_STATE_WALK,
5.    PLAYER_STATE_DEATH,
6.
7.    PLAYER_STATE_COUNT,   /* total number of states */
8. };
```

Typedef Template Containers

All uses of template containers should be typedef'd to reduce user error and code length for readability.

```
Example:
1. typedef std::vector<Enemy*> EnemyList;
2. typedef std::vector<Player*> PlayerList;
3.
4. EnemyList::iterator oIter = m_aEnemyList.begin();
5. EnemyList::iterator oEnd = m_aEnemyList.end();
6.
7. // instead of
8. std::vector<Enemy*>::iterator oIter = m_aEnemyList.begin();
```

Magic Numbers

A magic number is an unnamed numerical value that isn't easily distinguishable as to its purpose. An example is assigning a variable to 0 is easily identified as setting it to "null" or invalidating it in most cases, whereas setting a variable to 7 is not clear as to its purpose.

Magic numbers should **NEVER** be used.
All constant numerical values should be represented with either a pre-processor define in straight-C, or a const variable in C++ so as to preserve variable type:

```
C Example:
1. #define MAX_LIVES 10

C++ Example:
1. const unsigned int MAX_LIVES = 10;
2. const float MAP_WIDTH = 256.0f;
```

Commenting

All classes should have a comment above their declaration describing what the class is used for.

All functions where their purpose cannot be easily understood should have a comment above them describing their behaviour.

```
Example:
1. // Class to represent an enemy soldier
2. class Soldier
3. {
4. public:
5.
6.    /*
7.          Soldier attacks player with machine gun fire
8.          and constantly turns to face player.
9.    */
7.    void SetAttacking();
8. };
```

Comments should be used often!
They are as much for other people as they are you.