Gamer Camp Pro Coding Standards Guidelines

Purpose

The purpose of these coding guidelines is to:

- Encourage use of coding idioms that reduce bugs.
- Allow us to work more effectively as a team throughout the entire code base.
- Promote consistency, clarity, and readability...
- ... and therefore maintainability.
- See also the defensive programming page - defensive programming forms part of our coding standards.
- (eventually) to auto generate code documentation (using doxygen, at some point)

Naming Conventions

All names for types, functions, instances etc. should be descriptive and use the CamelCaseCapitalisationConvention.

In addition to this we will define a hungarian notation (i.e. alphanumeric prefix that identifies the type) for all types and identifiers other than functions.

types

The names of types should prefix the CamelCase name with the following letters:

type of identifier	Prefix	Example Name
enum	E	enum ENetworkError { ... };
enum value	Exxx_	ENetErr_ConnectionLost, n.b. purpose is to disambiguate the enum it belongs to
struct	S	struct SNetworkPacketHeader { ... };
class	C	class CConnectionManager { ... };
templated type	T	template< typename TType > class TNetworkStateBase { ... };
typedef	as appropriate	typedef of template is usually considered a class e.g. typedef TList< CRenderable* > CRenderableList;
typedef for function pointer	PFN	typedef bool (*PFNFirstIsLarger)(int, int);
function	n/a	void StartWithACapitalAndUseCamelCase();

function used only as a 'callback'	CB	bool CBOnOSSaveGameUpdateNotify(SaveGame*);
'standalone' templated function i.e. not a member of a templated type	T	template< typename TType > void TThisIsATemplatedFunction
virtual function i.e. virtual destructors obviously exempt :)	V	virtual void VUpdate(float fTimeStep);
#define / preprocessor macro	ALL_CAPS	#define THIS_IS_A_CONSTANT_OF_PI (3.14159f)

instances of types

type	instance prefix
char	ch
int	i
float	f
struct	s
class	c
enum	e
bool	b
function pointer	pfn
C-style 'zero terminated string' pointer	psz
custom types as appropriate e.g. CVector3	v3

instance prefixed modifiers

These go in front of the regular instance hungarian prefixes above to indicate extra information about the instance's type.

modifier	prefix
class (not struct) member data	m_
class (not struct) static member data	sm_
pointer	p
reference	r
array	a
const	k
unsigned	u
volatile	v

scope prefixes

scope	prefix
global	g_
static (function or file scope)	s_
global scope constant	k_

The "Hungarian Heuristic"

The Hungarian Heuristic is that any hungarian prefixes you use should be concatenated as you would describe the type if you were talking in plain English...

array of pointers to floats	float* apfAppropriateIdentifier[SIZE];
pointer to an array of chars	char* pachWellChosenName = new char[SIZE];

array of unsigned ints	unsigned int auContiguousBlockOfUInts[SIZE];

code layout

- Tabs not spaces
- Whitespace around everything. Exception - brackets for functions / conditionals should touch what they relate to.
- Curlies on **separate** lines.
- Indent code between curlies (see below for examples / exceptions)
- Use C++ style comments. NOT /* this style */
- Follow all rules demonstrated below.

examples:

```
// declare variables one per line
int iAnInteger = 0;

// pointer or reference modifiers NEXT TO THE TYPE
int& riAnIntegerReference = iAnInteger;
int* piAnIntegerPointer   = &iAnInteger;

// * no space between fn name and opening bracket
// * spaces between braces and parameters
int SomeFunctionDeclaration( int iParam, int iAnotherParam );

// * no space between the keyword and the opening bracket
// * spaces between braces, identifiers, and operators
if( iDentifierOne > iDentifierTwo )
{
    // ALWAYS use curly braces for code following control statements
}

// * use brackets appropriately to disambiguate mathematical statements
// * DO NOT rely on operator precedence
float fTheAnswer = ( ( fAddThis + fToThis ) - fSubtractThis ) / fAndDivideByThis;

// case indents to switch level
switch( iDentifier )
{
case 1:
    // indent code after the case
    break;
default:
    // ALWAYS have a default
    break;
}

// * access specifiers indented at level of brackets
// * no implicit access specifiers
// * inheritance list as shown
class CSomeClass
: public CBase
, public COtherBase
{
private:
      int m_iThing;
      int m_iOtherThing;

public:
```

```
      // initiliser list as shown align brackets
      CSomeClass()
      : m_iThing ( 0 )
      , m_iOtherThing( 0 )
      {
      }

      // only define very simple inlines in declaration
      inline int GetData()
      {
           return m_iThing;
      }
};

// horizontally align vertically adjacent code
int FuncOne   ( int iParam );
int FuncTwo   ();
int FuncThree ( int iParam, int iParam2);
int FuncFour  ();

// maximum width of 128 chars for long single line expressions.
// If operators are involved in the split statement, they should line up to the left.
// The emphasis is on ease of visual digestion of semantics, and likelihood of breaking
// code by commenting / #if-ing parts of it out.

int i = SomeMassivelyLongFunctionName( SomeOtherReallyLongFunctionName( iSomeValue ),
                                       YetAnotherLongFunctionThatReturnsInt() );

// * break large logical tests across multiple lines
// * wrap individual tests in brackets to disambiguate logic
// * put the logic operators at the start of the line when splitting
if(   ( cSomeInstance.HasGotAVeryLongFunctionName() && cSomeInstance.IsTrue() )
   || ( cSomeInstance.IsNotEntirelyCertain() && cSomeInstance.LikesToMoveItMoveIt() ) )
{
      // do stuff
}
else
{
     // do other stuff
}

// example loops - MAX_LOOP is used as an array bound for illustrative purposes
for( int iLoop = 0; iLoop < MAX_LOOP; ++iLoop )
{
     // do stuff
}

// keep loop variable init close to the loop
int iLoop = 0;
while( iLoop < MAX_LOOP )
{
     // do stuff
     ++iLoop;
}

int iDoLoop = 0;
do
{
     // do stuff
     ++iDoLoop;
}
while( iDoLoop < MAX_LOOP );
```

Comments
Comments should be used for both **high level documentation of code**, and for **giving others (and yourself) insight into the workings of code** that may not be obvious at first glance (e.g. magic numbers, weird algorithms etc.)

High Level Documentation

Class / Module
Class / module code should have a (typically large!) comment in the relevant header that explains:
- The purpose of the class / module
- Any assumptions that it makes
- Any interactions with other code, or code it relies on
- If the code must be used in a certain way, either:
    - include sample code showing usage inside the comment, or
    - point the reader to somewhere it is already being used in the codebase
- Detail how the code works
- How the functions relate to one another and to the behaviour of the class / module
- Why you chose to do it that way as opposed to another way
- Known limitations
- Possible improvements

Functions
Functions should have a comment above their definition that explains:
- The purpose of the function
- What each of its parameters does (if it has any)
- What its return value means (if it returns a value)
- Any other functions in the same class / module that it interacts with
- Any assumptions it makes
- If it is virtual, which original base class function it overrides of

If a member function is virtual or static, this should also be mentioned in the comment above the definition for clarity.
Example comment on a virtual member function that:

```
// Updates the current state of the network when in the connected state.
// Param1: fTimeStep - time delta since last call
// Return: CNetworkStateBase::eNSR_OK if ok,
// on error appropriate value from CNetworkStateBase::EErrorCode.
// Virtual override of CNetworkStateBase::VUpdate
CNetworkStateBase::EErrorCode CNetStateConnected::VUpdate( float fTimeStep )
{
    return CNetworkStateBase::eNSR_OK;
}
```

Giving Extra Info in Code
In general, comments in the code should be informative rather than descriptive.

Don't just put comments that say "get a pointer, call update" etc. These will be obvious from the syntax and well chosen names used for the various functions you're calling.

Do explain any code that might not be 100% clear to someone at first glance by someone else - especially magic numbers or weird algorithms.

Always...
- use defensive programming techniques - pay special attention to use of assertions. If your code assumes something, use an assert to test that assumption.
- write code that has a traceable and 100% repeatable initialisation path (e.g. don't use lazy intialisation).
- write control statements (if, while, for, etc.) with curly brackets, even if they're only 1 line.
- use #if !defined(MACRO) rather than #ifndef MACRO (except for header include guards)
- use brackets around #define values e.g. #define SIXTY_NINE (69)
- use brackets around #define parameters in the expansions e.g. #define MAX(a, b) (((a)>(b))?(a):(b))
- use explanatory names for functions
- use accessor functions for getting and setting private or protected member variables...
- ... and use the accessors even from within the class that owns the data
- Place the * and & next to the type *not the instance* when declaring instances of types...
- ...and declare only one instance per line of code - e.g. int* piInteger;
- classes:
    - use inline on declaration and definition of functions
    - use virtual keyword and the override keyword in the declaration of overridden virtual functions for clarity

- use names for predicate functions (i.e. functions that return true/false) that approximate english descriptions when in-use. (for example):

```
class CType
{
public:
      bool IsEmpty      ();
      bool HasFinished  ();
      bool WasOverStated();
};

if( cTypeInstance.IsEmpty() )
{
}
```

Never...

- use #if 0 ... #endif : comment the code out (comment selection: ctrl+k, c. uncomment selection: ctrl+k, u).
- use logically negative flags - e.g. bMusicDisabled (WRONG!) vs. bMusicEnabled (RIGHT!).
- declare more than variable per line.
- use personal pronouns in variable names (e.g. iMyReturnValue).
- use numerical literals for magic numbers inline in code; these sorts of values should always be constants / or macros
- use non-standard integer or float types unless you need to. In general, **int** (i32) will be the fastest integral type and **float** (f32) will be the fastest floating point type.

Never, ever, ever...

- use dynamic_cast
- use lazy creation / initialisation (including and especially for singletons!)