Coding Standard

Introduction

"Beware of the man who won't be bothered with details." -- William Feather

The purpose of a coding standard is to ensure that all code produced is clear and consistent, as well as quick and easy to work with, especially by people who didn't originally write it. This document describes our coding standard, and provides example files for reference. Much, but not all, of the material here is based on Clean Code, by Robert C. Martin. If you have not read that book yet, read it as soon as possible.

One of the key qualities that distinguishes an experienced programmer from an inexperienced one is an appreciation of the importance of applying a coding standard. Consistently applied, a coding standard is highly advantageous and gives rise to code that is:

- easier to read quickly, with 100% comprehension, by many people, not just the person who wrote it.
- more stable.
- more flexible.
- easier to modify without introducing bugs.
- much more long lived.

Applying the following coding standard is mandatory. You are expected to be familiar with everything here, and to apply it to all the code you write.

Meaningful Names

Communicate Intention

The name of a class, function or variable should reveal intent - why it exists, what it is for and how it is used. A good rule of thumb is, if a name needs a comment to explain its intent, then its a bad name. For example, this is a bad name:

```
    int d;
```

This is better:

```
int timeOfDay;
```

Avoid Disinformation

Avoid using terms in confusing ways, for example, don't call something an AccountList when it's not actually a variable of some kind of List type (and even then, don't use List). The word List has a very particular meaning in code. Instead use another term, like AccountGroup.

Avoid using single letter names like lower case l and upper case O, which in system fonts look indistinguishable from one and zero.

Avoid non-informative names like a1, a2 etc. If there is a difference between a1 and a2, the names should communicate that.

Interfaces and Implementations

Interfaces and base objects are the foundation of inheritance and polymorphic programming. Always consider interface names from the client code's point of view. Don't add "I" or "Base" to the name. If the code expects to consume a Graphics object, call the interface Graphics (not GraphicsI or GraphicsBase) and call the implementations GraphicsImpl, GraphicsGL or GraphicsD3D. The client code should have no idea it's dealing with anything other than a Graphics object, not a GraphicsI or worse, a GraphicsBase, which is extremely misleading.

Avoid Mental Mapping

A name that requires the reader to already have some particular knowledge above and beyond the normal level is a bad name. Clarity is more important. Single character variables are fine in short for loops for example (except l and O), but outside of that full names should be used.

Class Names

Class names should be nouns, like Texture, List. A class name should not be a verb.

Function Names

Function names should have a verb or verb phrase, like RenderPlayer, DeleteTexture. Accessors, mutators and predicates should be named for their value and prefixed with Get, Set and Is.

Avoid confusing boolean accesors / mutators that should really be using enums, for example:

```
void SetDirection( ut_boolean facingRight );
```

The above is extremely confusing in client code. What does the following mean?

```
SetDirection( false );
```

Does this mean the entity doesn't have a direction? Is now standing still? Anyone reading this would have to hunt down the method implementation to understand this code. The following would be clearer:

```
SetFacingRight( false );
```

The reader now knows exactly what the intent of the code is, and doesn't need to do anything other than read the line to be sure of what's going on. But, if the SetDirection signature is required, then the best solution would be an enum, as follows:

```
SetDirection( DIRECTION_LEFT );
```

This is 100% clear, and requires no additional work to understand what's going on.

Variable Names

Avoid abbreviations, unless the context makes it very clear, or they are obvious or well known. For pointers and references, keep the * and & characters attached to the type name rather than the variable name. This helps clarify the type as being either a pointer or a reference.

```
utGraphics* pGfx;

utGraphics& gfx;
```

Name Consistently

If the term Manager is being used in one place in the code, don't use the term Driver to mean the same thing in another. Be careful to keep names consistent with what's already there. If what's already there no longer makes sense, have it changed.

If there are three variants of utMessageBox that are named like this,

utMessageBox

utMessageBoxYN

utMessageBoxYNSymbol

then when you're adding another, name it consistently with the above. For example if you're adding a new variant that supports animated Yes / No buttons, the above would suggest a name like utMessageBoxAnimatedYN, not utAnimatedMessageBoxYN. These details are important.

Add Meaningful Context

Be careful to use names that don't require extra context (in the form of comments) in order to understand them, for example, 'firstName', 'lastName', and 'street' all fit with the idea of an address, but the word 'state', on its own has several competing meanings in Computer Science. If you find the word state on its own in a method, you may need to dig to find out its meaning if its being used in the context of an address. Instead, wrap the variables into an explicit context, a class called 'Address'. This removes all ambiguity, and makes the code much more readable.

Functions

Do One Thing

Keep functions short and focused on one thing. When a function does more than one thing, you tend to need comments to clarify the different things that it does. A comment is a good 'code smell' for 'Refactor to Method'.

Don't Mix Abstraction Levels

The code within a function should be at the same abstraction level - don't mix high level and low level code within a function, refactor out low level code to achieve this, even if it's only a line or two. For example:

```
    void ReadDataFile( const ut_char8* filename )
{
        utInputStream* file = OpenFile( filename );
        if( NULL != file )
{
            file->Mark();

            if( file->Read() == 0xEF &&
                file->Read() == 0xBB &&
                file->Read() == 0xBF )
{
                file->Mark();
}
else
{
                file->Reset();
}
        }

        ReadHeader( file );

        ReadBody( file );
    }
```

Ignoring the other code smells present in the above function, it mixes abstractions levels. The first chunk of code is low level, opens a file and reads an optional utf-8 Byte Order Marker from the file, and then reads the header and body. To standardise the abstraction levels we need to refactor out the file open and BOM processing into a single function as follows (ignoring validation for this example):

```
    void ReadDataFile( const ut_char8* filename )
{
utInputStream* file = OpenFileAndSkipByteOrderMarker( filename );

        ReadHeader( file );

        ReadBody( file );
    }
```

This code now reads much more easily. The refactored method has a highly descriptive name (OpenFileAndSkipByteOrderMarker), and no longer needs a comment to explain what it does.

Wrapping and Scope Clarification

Split lines greater than 120 characters in length, and tab indent anything that wraps.

If the function signature exceeds 120 characters, put each parameter on its own line, tabbed in one level from the function name.

```
    void DoSomethingWithReallyLongParamList(
        const SomeLongClassName& param1,
        const SomeOtherLongClassName& param2,
        const SomeOtherOtherLongClassName& param3 )
{
    }
```

Place opening and closing braces on a new line. Do not write opening braces on the same line as an if / while clause etc.

```
    if( niceClearBracesThatAreVeryEasyForYourEyeToTrackQuickly )
    {
        // Having the braces visibly isolated like this allows your eye to very quickly
        // pick out the logical chunks of code much more quickly.
    }
```

Use Context

Keep the argument count as low as possible. In general, when you see an argument list getting long, it's highly suggestive that a wrapper context is required. For example:

```
    snailMail.PostTo(
        firstName,
        lastName,
        street,
        town,
        city,
        country );
```

The above might be better if the details of an address were wrapped in an Address container, and passed around that way:

```
    snailMail.PostTo( someAddress );
```

Comments are NOT context. Avoid comments at all costs. Always favor documentation that compiles.

Arguments

Place output arguments first, consider naming them outVarName to clarify the intent. In general 'out' arguments should be avoided, however, for performance reasons they tend to be quite useful.

Make non-fundamental types references. A reference amounts to a 32 bit quantity - much faster to pass to a function than copying an entire object (which will invoke constructors / destructors etc).

```
    // This does a lot of work it simply doesn't need to do... It creates two new Vector3 objects,
// and copies in the contents of each. Their respective destructors also need to fire on exit.
// None of that is actually required in order to add them together. The person who wrote this
// doesn't understand the performance characteristics of their code.
    Vector3 Add( Vector3 a, Vector3 b );

    // This on the other hand does just what it needs to... No new objects are created for the
// arguments. No constructors are called, and no destructors either. The addresses of the two
// arguments are passed in, and the function operates on the objects in place - this is
// another reason why they should be const.
    Vector3 Add( const Vector3& a, const Vector3& b );
```

Make non-output parameters const where possible. Const should be used to lock down anything that should not normally change. There should be a good reason for something not being const. For more information on Const Correctness please see the following document: Const Correctness.

The above Vector3 Add function can be made faster by using an out argument as follows:

```
    void Add( utVec3& outResult, const Vector3& a, const Vector3& b );
```

This avoids the constructor / destructor overhead of the return variable and can be quite important in tight loops - for example iterating over particles in a complex particle system.

Avoid booleans in functions other than simple boolean setters, use enums instead.

Avoid Side Effects

Don't allow the function to have side effects - if the function does something the user should really know about, encode that into the name of the function.

DRY: Don't Repeat Yourself

Don't repeat yourself - refactor out duplicated code. Use the 'Rule of Three' - tolerate a duplication once - the third time something gets copied, refactor.

Comments

If you need a comment, consider refactoring your code so that the comment isn't required. Comments are a last resort of communication because they tend not to get updated with the code they relate to when that code changes, especially when multiple people are editing the code.

```
    // Always returns 1        <- SPOT THE OUT-OF-DATE COMMENT!
    int GetNumThings() const
{
      return 0;
    }
```

A comment usually indicates a 'Refactor to Method' opportunity.

When you add a comment to code you didn't write from scratch, or are updating code, it is a good idea to attribute your comment by prepending your initials with the @ character, for example "@kd". This allows people to query you after the fact if something doesn't make sense. If you don't do this, it makes it difficult for someone to address issues later because they may not understand why you did what you did and if they can't identify you, they can't ask for clarification.

Use recognized tokens to indicate "TODO" and "FIXME" - these usually have some kind of IDE support. Note that these are CAPITALIZED so they are hard to miss. Todo comments are usually required when code has been taken to a certain degree of completion, but the final hurdle can't be reached because of time / complexity constraints. It is necessary to capture 'where' the code currently is and what should be done to complete it, otherwise you have to read a lot of code, and figure out where it is - a time consuming exercise. Fixme comments are essentially urgent Todos. At any given time there should be very few Fixme comments. Any that remain should be actively removed by implementing the required code to address the underlying issue.

Formatting

Code needs to be formatted carefully for one reason, and one reason alone - so that it can be quickly read by someone. It's extremely important that one formatting style is used throughout the code base so that you are not blocked every 30 seconds, wrestling with yet another formatting layout (or the complete absence of one) forcing you to spend significant time deciphering what's going on in the code. Rather than list specific, individual rules, which tend to be poorly followed, it is better to understand the basic ideas. Essentially, the idea of formatting code is to present it in a way that simplifies how it is read, and focuses the reader on the information that is important, while removing all other distractions. Good formatting conventions also make it easier to write and change code without spending large amounts of time formatting the code. The most important concepts revolve around using ordering and white space, both horizontally and vertically, to focus your eye on the important information

Vertical Formatting

Information should be ordered from high level to low level as you move down a file. This allows the reader to delve as deeply as necessary and no further in order to find the relevant information. In other words, if a high level function makes low level calls, those lower level functions should appear after the high level function. Files should be short enough that the user can move around the file reasonably quickly. This is especially important for people who didn't write the code and may not be as familiar with it as the author. Having to spend large amounts of time hunting down details is a huge waste of effort.

Our layout for class header files is straight forward. We place public information at the top, private member functions in the middle, and private data members at the very bottom. The reason for this layout is that the vast majority of the time someone looks at a header to understand how to consume the class at a client level. The information required is the publicly consumable parts of the class. This is shown up front. The lower level details are pushed down.

Use vertical spacing to group related chunks of code. This can be within a function, with groups of functions, with function signatures in the header, with data members etc. A good metric for how well you're vertically spacing information is if you can pick out the logical groups when you blur your eyes while looking at the code. You should be able to see the logical units clearly simply by the coarse shape they make on the screen. Vertical spacing separates concepts and vertical density implies closeness.

Horizontal Formatting

If a line exceeds 120 characters, split it, and tab indent the subsequent lines on tab level.

Use horizontal white space to help identify the key components on a line, for example:

```
    ut_float varName=zfactor+t*10.f+offset[a];
```

The above requires quite a bit of time to read - too much time considering the amount of information it actually contains. The first thing we can do is help the reader by showing that there is an assignment - a logical structure that has a left and right side. We do this by introducing white space around the = operator like so:

```
    ut_float varName = zfactor+t*10.f+handle[offset];
```

Already there's an improvement - from the corner of my eye, before I've even looked directly at the line, my brain knows there is an assignment here because I'm so used to scanning for it. We can do even better. There's an equation on the right hand side with three operators. The parameters at tightly packed and difficult to pick out. There are also precedence rules at play. These can be emphasised like so:

```
    ut_float varName = zfactor + t*10.f + handle[offset];
```

Notice that we can quickly pick out the three main components of the calculation. Also not that the t*10.f part has been left tightly packed to emphasise the precedence. This helps show in a flat way that the calculation is actually tree-like in nature - the multiplication occurs first, because of precedence, and then the result operates at the same level as the other two components, zfactor and handle[offset]. Alternatively, brackets can be used:

```
    ut_float varName = zfactor + (t * 10.f) + handle[offset];
```

Note above, that the use of white space was not applicable in a blind, one-for-all kind of way. It was used selectively to emphasis components. This takes some practice to do well. You need to be able to consider the code from another person's perspective. You're not writing the code just for you. You're writing the code to make your, and your colleagues', lives easier.

Simplified Layout

We format the code in a way that minimizes the amount of effort required to write new code, and change existing code. For example, when dealing with line wrapping for function signatures greater than 120 characters we place each parameter on its own line, tabbed in one level. This is easy to write. We avoid formatting like this:

```
// THIS IS BAD FORMATTING - INHERENTLY INCONSISTENT

    void DoSomethingWithReallyLongParamList(
                                            const SomeLongClassName& param1,
                                            const SomeOtherLongClassName& param2,
                                            const SomeOtherOtherLongClassName& param3 )
{
    }
```

The reason the above is bad practice is because if you want to change the function signature you have to re-tab all of the parameters. The less you have to do to keep the code clean, the better. Instead of the above, we make it much simpler:

```
    void DoSomethingWithReallyLongParamList(
        const SomeLongClassName& param1,
        const SomeOtherLongClassName& param2,
        const SomeOtherOtherLongClassName& param3 )
{
    }
```

The above is actually much clearer anyway, but it has the obvious advantage of the signature being free to change in width independently of the parameter placement.

Another example of laborious, unwieldy formatting appears quite frequently in class members:

```
    Class SomeClass
{
        ut_int                  _member1;
        utSomeOtherClass        _member2;
        utSomeOtherOtherClass   _member3;

        ..
        ..
    };
```

The above is a bad idea for the same reason as the initial example. If you add a new member variable that doesn't fit, you need to re-layout all the existing members to keep things consistent - this doesn't scale. We simplify things by declaring variables independently, and grouping vertically where it makes sense.

```
    Class SomeClass
{
        ut_int _member1;

utSomeOtherClass _member2;
utSomeOtherOtherClass _member3;


        ..
        ..
    };
```

Some IDEs will auto format - this is great, but unfortunately many don't so we need to take charge of formatting in a lot of cases, or the code tends to become a mess of competing formats as different people edit on different platforms.

Consistency

Above all, keep your code consistent with what's already there. If you add code that isn't consistent, because what's there doesn't adhere to this standard, take a moment and update the code to adhere to the standard. Don't let the code rot.

Example

Here is an example c++ / header from UtopiaGL. These files implement utZipFilePath, the zip part of the file system. The class implementation is highly focused, and most member functions are very short and do only one thing, with all steps at a particular abstraction level. There is a nested class to further encapsulate the data.

utZipFilePath.h

```
#pragma once

#include "Utopia.h"
#include "utFilePath.h"
#include "utHashTable.h"
#include "utHashObject.h"
#include "utMemoryExpanderPool.h"

class utInputStream;
class utZipFileInputStream;

class utZipFilePath : public utFilePath
{
class FileDesc : public utHashObject
{
public:
FileDesc();
~FileDesc();

ut_boolean InitFile( void* const uf, utMemoryExpanderPool& fileNamePool );
utZipFileInputStream* OpenFile(
const ut_char8* const pszHostZipFile,
const ut_uint32 embeddedHostZipFileOffset = 0,
const ut_uint32 embeddedHostZipFileLength = 0 );

const ut_char8* GetName() const;
ut_int GetNameLength() const;

private:

ut_char8* _pszName;
ut_int32 _nameLength;
ut_uint32 _offset;
    };

public:
utZipFilePath(
const ut_char8* const pszPath,
const ut_uint32 embeddedHostZipFileOffset = 0,
const ut_uint32 embeddedHostZipFileLength = 0 );
~utZipFilePath();

ut_boolean Init();
utInputStream* OpenFile( const ut_char8* const pszFile );
ut_boolean FileExists( const ut_char8* const pszFile );

ut_boolean IsInitialised();

private:

ut_boolean Load( const ut_char8* const pszFile );
ut_boolean AllocFileDescriptions( void* const uf );
ut_boolean LoadFileDescriptions( void* const uf );
ut_boolean InitFiles( void* const uf, FileDesc* const pFiles, const ut_int numFiles );
ut_boolean BuildHashTable();
ut_int GetHashTableSize( const ut_int numFiles );
void Unload();
```

```
private:
utHashTable _fileTable;
utMemoryExpanderPool _fileNamePool;

ut_int32 _numFiles;
FileDesc* _pFiles;

ut_uint32 _embeddedHostZipFileOffset;
ut_uint32 _embeddedHostZipFileLength;
};
```

utZipFilePath.cpp

```
#include "utZipFilePath.h"
#include "utZipFileInputStream.h"
#include "utSystem.h"
#include "utString.h"
#include "utMemory.h"
#include "utString.h"
#include "utFileUtils.h"
#include "zlib/unzip.h"

#define ZIPFILEPATH_MAX_HASH_SIZE 1024

utZipFilePath::FileDesc::FileDesc()
{
_pszName = NULL;
_nameLength = 0;
_offset = 0;
}

utZipFilePath::FileDesc::~FileDesc()
{
}

ut_boolean utZipFilePath::FileDesc::InitFile( void* const uf, utMemoryExpanderPool& fileNamePool )
{
ut_char8 szTmpFilename[ UT_MAX_PATH ];
unz_file_info fi;
UT_Ensure( UNZ_OK == unzGetCurrentFileInfo( uf, &fi, szTmpFilename, UT_MAX_PATH, NULL, 0, NULL, 0 ) );
UT_Ensure( UNZ_OK == unzGetCurrentFileInfoPosition( uf, (unsigned long*) &_offset ) );
_nameLength = fi.size_filename;
_pszName = (ut_char8*) fileNamePool.Alloc( _nameLength + 1 );
UT_Ensure( _pszName );
utFileUtils::MakeCanonicalPath( _pszName, szTmpFilename, _nameLength );
UT_RetBool();
}

utZipFileInputStream* utZipFilePath::FileDesc::OpenFile(
const ut_char8* const pszHostZipFile,
const ut_uint32 embeddedHostZipFileOffset,
const ut_uint32 embeddedHostZipFileLength )
{
utZipFileInputStream* pS = UT_new_temp utZipFileInputStream();
if( !pS || !pS->Open( pszHostZipFile, _offset, embeddedHostZipFileOffset, embeddedHostZipFileLength ) )
{
UT_SafeDelete( pS );
}
return pS;
}

const ut_char8* utZipFilePath::FileDesc::GetName() const
{
return _pszName;
}
```

```
ut_int utZipFilePath::FileDesc::GetNameLength() const
{
return _nameLength;
}


utZipFilePath::utZipFilePath(
const ut_char8* const pszPath,
const ut_uint32 embeddedHostZipFileOffset,
const ut_uint32 embeddedHostZipFileLength ) : utFilePath( pszPath )
{
_embeddedHostZipFileOffset = embeddedHostZipFileOffset;
_embeddedHostZipFileLength = embeddedHostZipFileLength;
_numFiles = 0;
_pFiles = NULL;
}

utZipFilePath::~utZipFilePath()
{
Unload();
}

ut_boolean utZipFilePath::Init()
{
return Load( _pszPath );
}

ut_boolean utZipFilePath::Load( const ut_char8* const pszFile )
{
unzFile uf = unzOpen( _szPath, _embeddedHostZipFileOffset, _embeddedHostZipFileLength );
UT_EnsureMsg( uf, (":: Failed to open zip %s", pszFile) );
UT_EnsureMsg( LoadFileDescriptions( uf ), (":: Failed to load file descriptions from zip %s", pszFile) );
unzClose( uf );
return true;

UT_ExitWithError:
unzClose( uf );
Unload();
utLog::Err( "utZipFilePath::Load - Failed!" );
return false;
}

ut_boolean utZipFilePath::LoadFileDescriptions( void* const uf )
{
UT_Ensure( AllocFileDescriptions( uf ) );
UT_Ensure( InitFiles( uf, _pFiles, _numFiles ) );
UT_Ensure( BuildHashTable() );
utLog::Info( "utZipFilePath::LoadFileDescriptions - found %d files", _numFiles );
UT_RetBool();

}

ut_boolean utZipFilePath::AllocFileDescriptions( void* const uf )
{
unz_global_info gi;
UT_EnsureMsg( UNZ_OK == unzGetGlobalInfo( uf, &gi ), (":: Failed to get global info") );
_numFiles = gi.number_entry;
UT_EnsureMsg( _numFiles > 0, (":: num files is 0") );
_pFiles = UT_new_const FileDesc[_numFiles];
UT_Ensure( _pFiles );
UT_RetBool();
}

ut_boolean utZipFilePath::InitFiles( void* const uf, FileDesc* const pFiles, const ut_int numFiles )
{
unzGoToFirstFile( uf );
for( ut_int i=0; i<numFiles; i++ )
{
UT_Ensure( _pFiles[i].InitFile( uf, _fileNamePool ) );
unzGoToNextFile( uf );
}
UT_RetBool();
}
```

```
ut_boolean utZipFilePath::BuildHashTable()
{
UT_Ensure( _fileTable.Init( GetHashTableSize(_numFiles) ) );
for( ut_int i=0; i<_numFiles; i++ )
_fileTable.Put( _pFiles[i].GetName(), _pFiles[i].GetNameLength(), &_pFiles[i] );
UT_RetBool();
}

ut_int utZipFilePath::GetHashTableSize( const ut_int numFiles )
{
ut_int hashSize;
for( hashSize=1; hashSize<ZIPFILEPATH_MAX_HASH_SIZE; hashSize*=2 )
{
if( hashSize >= numFiles )
break;
}
return hashSize;
}

void utZipFilePath::Unload()
{
UT_SafeDeleteArray( _pFiles );
_numFiles = 0;
_fileTable.Free( utHashTable::DONT_FREE_OBJECTS );
_fileNamePool.Free();
}

utInputStream* utZipFilePath::OpenFile( const ut_char8* const pszFile )
{
FileDesc* pF = (FileDesc*) _fileTable.Get( pszFile );
return pF ? pF->OpenFile( _szPath, _embeddedHostZipFileOffset, _embeddedHostZipFileLength ) : NULL;
}

ut_boolean utZipFilePath::FileExists( const ut_char8* const pszFile )
{
return NULL != _fileTable.Get( pszFile );
}

ut_boolean utZipFilePath::IsInitialised()
{
return _pFiles ? true : false;
}
```