Hardwired Games Programming Style Guide

Contents

1. Introduction

This document specifies the company-wide programming style for Hardwired Games.

Why use a standard programming style? Several reasons:

1. **Type safety.** For example, if you ever change the type of a particular object, you need to change the name, which then causes the compiler to generate several compile-time errors that are easy to fix, as opposed to potentially more subtle bugs that might creep otherwise. (For instance, maybe there are defined casting operators that cast from the original type to the new type and thus may cause unwanted behavior.)
2. **It's easy to tell the type and scoping of variables.** Since we're using C++, it's especially useful to know which variables are global, which are members of classes, and which are local to the functions.
3. **Simpler Debugging.** Errors resulting from messy code or non-standard usage are hard to find. Software standards help prevent errors and make it easier to find errors.
4. **Readability!** It's vital for any programmer in the company to be able to step through your code and to debug it.

Having a standard makes the code much easier for everyone to understand, since they all know the conventions. The price? A small overhead of 1-2 characters in your names and perhaps a slight change in your own variable naming convention.

2. General Coding Style

These guidelines describe in general how software should be developed as well as how efficiency and portability issues should be resolved.

2.1 The Software Development Process

Start with a design phase in which you design the software on paper. Decide on the high level algorithms to be used and the logical structure. While structuring a project, always use an object-oriented methodology, even if the implementation does not require this. The logical structure should indicate how the functionality groups into separate source files. It should also provide enough detail to break down the project into separate, logical parts that directly correspond to functions in the implementation.

Implement the major functions. First, write pseudo-code for each major function. Doing so suggests possible subroutines. Next, write the subroutines. Translate the pseudo-code into a compilable form, compile it, and test it.

Perform the process of pseudo-coding, writing, and testing on a function-by-function basis, since this limits the number of mistakes that are introduced in each iteration. More importantly, since mistakes are localized to a small portion of the code, finding and correcting them is easier.

2.2 Efficiency

Efficiency can mean several things: maintenance efficiency (reliable, extensible, and easy to debug code), execution-time efficiency, or memory efficiency. All three kinds of efficiency are desirable in a program. Typically, making code more efficient involves trade-offs. For example, choosing a complicated, fast algorithm over a simple, slow one gains execution efficiency at the cost of maintenance efficiency.

In general, the priority for efficiency should be maintenance, then speed, then memory. Modify these priorities based on the goals of the software project and the target user platforms. For example, if the speed of a software game is critical, then speed becomes the top priority in the parts of the software that depend on speed. When speed is not critical, a general and straightforward algorithm should be chosen over a convoluted special case.

If an implementation is not fast enough or uses too much memory, then profile the code. Change the code based on the profile data, and not on a hunch that a certain change might optimize things. Also refrain from making peephole optimizations or declaring register variables: modern compilers are much better than humans in this respect. In particular, the readability of code should not be compromised by using constructs like `*(p+i)` instead of `p[i]`.

Often, achieving best efficiency involves choosing a different high-level algorithm. Changing algorithms will probably involve rewriting large parts of the code. To avoid this, it is essential to include efficiency in the design phase of the project.

2.3 Portability

Easily porting code is a goal in itself, and it helps instill programming standards that make the code easier to understand by multiple programmers.

Portability guidelines are:

- Do not assume file names or directory structures in the code; use environment variables instead.

- Since different architectures use different byte orders, try to avoid writing binary data.
- Explicitly declare the return type of a function.
- Avoid global variables! If global variables are unavoidable, declare them as static to confine them to a single file.
- Type conversions, including function parameter conversions, should be made explicit by the cast operator.
- All variables should be initialized explicitly.
- All constants should be named; i.e., code should not contain magic numbers.

2.4 Coding Goals

Important issues to think about when writing code:

- Size.
- Speed.
- Robustness.
- Safety.
- Testability.
- Maintainability.
- Simplicity.
- Reusability.
- Portability.

2.5 Our Use of C++

For Windows applications, we use a subset of C++ that includes using templates, referencing, inlining, single and some multiple inheritance, default arguments, operator overloading, `const`, and the `bool` basic type.

The only principal feature of the language we avoid is virtual base classes. This implies that the multiple inheritance we use is almost exclusively for inheriting from multiple pure virtual interface classes in an implementation class.

We do use in a limited way the STL (i.e., standard C++) container classes and algorithms, but only in code which is known to be non-speed critical. Do not use the STL container classes unless you understand their performance characteristics in time and space in the current Microsoft implementation.

C++ FOR NON-WINDOWS PLATFORMS

We adapt our C++ conventions for the peculiarities of each platform we code for. (For example, if we were developing for the Sega Dreamcast, we would not use any STL, which isn't supported by the Dreamcast development environment.) Note that any truly platform-independent code (such as a library intended for use on multiple platforms) must only use the subset of C++ that is supported on all our target platforms. For our current standards here, check with the lead programmer or director of engineering.

3. Naming Conventions

3.1 Files

> **NOTE: ALWAYS USE ALL-LOWER-CASE NAMES FOR FILES AND DIRECTORIES**
> Do not use any capital letters! Although mixed-case file names are useful, some operating systems we use do not necessarily preserve case (switching files between Windows 9x and Windows NT, for example, can lose case). Since the case of a file name can be significant for some operating systems or applications, using file names all in lower case helps avoids potential problems.

3.1.1 File Names

File names should be descriptive of their contents.

C++ source file names use lower case letters and numbers. They do not use underscores or spaces to separate words or other items in the name. For example, something.cpp is the C++ source file for the some thing module. Easily-understood abbreviations can be used in file names, like "mgr" for "manager" in foomgr.cpp.

Header files are named according to their use:

- Public interface headers have the same file names as the source files; for example, foo.h and foo.cpp. These headers are include files that declare the public interface and are placed in the inc directory by the make system.
- Internal library (local) header files must have a preceding "_" (underscore) on their name; for example, _foo.h. These headers are local versions of the headers that are sometimes needed in the src directory.
- For projects without public headers (like executable targets), the naming convention is up to the person in charge of that project.

For more details on headers, see Section 6.11.

Raw resource files use lower case letters and numbers. They do not use spaces to separate words or other items in the file name (e.g., combatlaser). However, since less technical people may use these files, use underscores to separate words in the file name if this will improve readability (e.g., hacker_tools).

3.1.2 Extension Names

Extension	*Created By*	*Compile To*	*Description*
.cfg	By hand	—	**Configuration files**. They contain UI and other settings used in the play of a game.
.cpp, .h	Visual C++	.obj	**C++ source files; C++ header files**.
.dep	mkmf	—	**Dependencies files**. These are used in the make system.
.dlls		—	**DLL listing files**. Files generated and used by the make system to keep track of changes between successive builds.
.ico			**Icon files**.
.log		—	**Log files**. They contain compiler errors and warnings.
.old		—	**Old file**. These are older files renamed with an .old extension and are retained for reference reasons.
.pch		—	**Output files (pre-compiled headers)**.

.reg		—	**Registration files**.
.ver		—	**Version files**. These are used to track the versions of a game or application.

3.1.3 Directory Names

Directory names use lower case letters and numbers. Do not use underscores or spaces to separate words or other items in directory names.

3.2 Used-Defined Types

User-defined types use the format:

<type prefix><library prefix><type name>

Example:

`tMemGraze`

User-defined types are mixed case, starting with a type prefix, followed if needed by a 2- or 3-letter library prefix (capital first letter), followed by the type name (capital first letter; other letters capitalized as necessary for clarity). Make your type names descriptive; avoid unclear abbreviations. Other programmers need to understand your types when they read your code!

3.2.1 Type Prefixes

Type prefixes are:

`I`	Interface ("`I`" is always capitalized).
`t`	Other type ("t" is always lower case).

Simple data types (e.g., `Rect`, `Pt`) should not use the type prefix, since prefixing obscures more than it helps in this case.

3.2.2 Library Prefixes

A library prefix is necessary only if the type is defined by a library and is used outside of that library. If the type is defined within the library and never exposed to clients, do not include the library prefix.

3.2.3 Notes on Choosing Type Names

Avoid polluting the global namespace. (Windows.h has done a fine job of stealing every name you can think of.) Use `namespace` declarations.

For example, say you define `IOverlay` for use in a game. It turns out the Active Movie SDK defines an interface called `IOverlay`. To fix this problem without `IOverlay`, include the Active Movie SDK inside a `namespace` declaration:

```
namespace win32 {
   #include <mmstream.h>
};
```

Microsoft should probably live in its own namespace. Also, having a Win32 namespace makes it really easy to see what Windows calls are being used. We are contemplating doing this in some of our libraries.

3.2.4 Type Name Examples

Some examples of type names are:

`tMemFoo`	A client-visible class defined in the "Mem" library.
`tBar`	A structure local to a particular module.

3.3 Variables

Variables use the format:

`<scope prefix><type prefix><library prefix><variable name>`

Example:

`gpMemFoo`

Variable names are mixed case with no underscores, optionally starting with a lower case letter describing scope, followed by an all-lower-case type prefix, followed if needed by a 2- or 3- letter library prefix (capital first letter; lower case first letter if no type prefix is present), followed by the variable name (capital first letter; other letters capitalized as necessary for clarity). Make your variable names descriptive; avoid unclear abbreviations. Other programmers need to understand your variables when they read your code!

3.3.1 Scope Prefixes

The scope prefixes are:

`g`	Global variable.
`m`	Member variable.

Simple variables (e.g., i, j, k, x, y, z) should not use scope prefixes, since prefixing obscures more than it helps.

3.3.2 Type Prefixes

The type prefixes are:

`b`	Boolean.
`c`	Character.
`d`	Double.
`e`	Enumeration.
`f`	Float.
`k`	Constant.
`n`	Integer.
`p`	Pointer. (Use multiple `p`'s to show the depth of the pointer; e.g., `pp` for a pointer to a pointer, `ppp` for a pointer to a pointer to a pointer, etc.)
`u`	Unsigned integer.
`v`	Void.
`x`	Template.

If a variable does not have a type prefix, then by definition it has some other type that those listed above.

Simple variables (e.g., i, j, k, x, y, z) should not use type prefixes, since prefixing obscures more than it helps.

3.3.3 Library Prefixes

A library prefix is necessary only if the variable is defined by a library and is visible to applications. This should almost never happen, as this kind of global is the evil enemy! Avoid implementations that export globals from libraries, since it can lead to all sorts of problems, especially race conditions as multiple clients all attempt to modify/read the same global.

3.3.4 Variable Name Examples

Some examples of variable names are:

`kPi`	A constant.
`mStatusVal`	A class member variable that holds status flags.
`gppMemFoo`	A global pointer-to-a-pointer defined in the "Mem" library and visible to clients.
`gppFoo`	A global pointer-to-a-pointer. (To contrast with `gppMemFoo` above, `gppFoo` could be defined in the "Mem" library but not be visible to clients.)

3.4 Functions

Functions use the format:

`<library prefix><function name>`

Example:

```
MemSplat()
```

Functions are mixed case, starting with a capital letter. There should be a library prefix (if needed), followed by the function name.

3.4.1 Library Prefixes

A library prefix is necessary only if the function is defined by a library and is used outside of that library. If the function is defined within the library and never exposed to clients, do not include the library prefix.

3.4.2 Notes on Choosing Function Names

Avoid polluting the global namespace. Use namespaces for global functions or the more useful prefixes.

For more on namespaces, see Section 3.2.3 above.

3.4.3 Function Name Examples

Some examples of function names are:

`MemPurge()`	A client-visible function defined in the "Mem" library.
`DoSomething()`	A function internal to an app or library.

3.5 Macros and Constants

For preprocessor macros, macros and constants that are used to generate (or compile out) code are all caps, with each word separated by an underscore. *Examples:*

```
#define FLOAT(val)   static_cast<Float>( val )
#define LISTENER_ON  1
#define LISTENER_OFF 0
#define TRUE         1
#define FALSE        0
```

4. Comments

Commenting your code is vitally important!

The most important place to comment your code is in its public interface, since that's what people are going to read to understand your code. Without good and complete documentation, you can be sure that people will use your code wrong (and therefore spend more time in debugging) or will continually pester you to explain your library to them. Both situations are bad.

It is also important to document your internal code, especially the algorithms within your code, so other people can come in and debug or modify your code. Invest the time; it's worth it.

Make sure to update all comments when you make changes to the code.

4.1 Block Comments for File Information

Every file must have an initial block of comments with correct information in it, including the copyright notice. Use the following arrangement:

```
//-----------------------------------------------------------------------------
// Copyright (c) 2000 Katerra Corp. All Rights Reserved.
//
// %File        : REQUIRED
// %Created     : REQUIRED
// %Author      : OPTIONAL
// %Maintainer  : OPTIONAL
// %Reference   : OPTIONAL
// %Description : REQUIRED
// %History     : OPTIONAL
//-----------------------------------------------------------------------------
```

Include a description of what the code does and of any important or unusual features about the code. Keep the description informative but concise. For example, "This function erases the user's hard drive" can be expressed as "Erases user's hard drive".

Example:

```
//-----------------------------------------------------------------------------
// Copyright (c) 2000 Katerra Corp. All Rights Reserved.
//
// %File        : matrix3d.cpp
// %Created     : 4-1-2000
// %Author      : Kim Kodar <kkodar@somewhere.com>
// %Description : Contains the non-inlined functions and methods of the
//                tMatrix3D class.
//-----------------------------------------------------------------------------
```

Comment blocks will be parsed by (yet-to-be-written) tools for certain magic keywords. These keywords start with % and end with white space or a colon. The information you prove should be such one might care about if some sort of HTMLizing tool were to process the file. Thus, use click-friendly data types such as URLs and e-mail addresses.

Use a backslash as the last non-white space character to continue a record. The record contents will continue after the comment intro, at the first non-white space character. Escape other material with backslash as needed.

Currently defined keywords:

- **%File** is the name of the file. Having the file name may seem redundant, but it is useful to have the name on a printout, when pasting files into e-mail, etc.
- **%Created** is the date of creation. Please use MM-DD-YYYY, even though everyone knows DD-MM-YYYY is much more sensible. To avoid possible confusion, do not use numbers for the month name. Abbreviating the month name is acceptable.
- **%Author** is the original creator(s). This is optional if unknown or the file is due to a large group effort. Include an e-mail address if possible.
- **%Maintainer** is the person you go bug when you want to make massive changes or when you need something explained.
- **%History** is an optional record for noting earth-shattering events in the lifetime of the source; i.e., complete rewrites, splitting into new files, major compatibility changes, etc. Include a date in the same format as "%Created" at the start of the record, and the name/e-mail of who made the change, if relevant. The reason why this record exists is to keep track of significant events when source code control comments are not available, or to preserve information across changes to the source control system.
- **%Reference** is an optional record for pointing to documentation, sources for algorithms, etc. Use URLs if available and possible.
- **%Description** describes the file.

Optional lines should be completely removed if not used.

Example verbose record:

```
//-----------------------------------------------------------------------------
// Copyright (c) 2000 Katerra Corp. All Rights Reserved.
//
// %File        : coollib.cpp
// %Created     : 4-1-2000
// %Maintainer  : Kim Kodar <kkodar@somewhere.com>
// %Reference   : http://woburn2/kanet/reference/doc/katerra/coollib.htm
// %Reference   : "Being Cool", March 2000 Game Developer Magazine, p22
// %Description : Adds really cool functionality. All sorts of fun        \
//                features are supported
// %History     : 5-2-2000 rewrote ZCoolCat class
// %History     : 5-10-10-2000 major rewrite to handle new container classes
//-----------------------------------------------------------------------------
```

Example minimal record:

```
//-----------------------------------------------------------------------------
// Copyright (c) 2000 Katerra Corp. All Rights Reserved.
//
// %File        : coollib.cpp
// %Created     : 4-1-2000
// %Description : Adds really cool functionality. All sorts of fun        \
//                features are supported
//-----------------------------------------------------------------------------
```

4.2 Public Interface Block Comments

Use // block comments to comment public interface code. Include a description of what the code does and of any important or unusual features about the code. Keep the description informative but concise. For example, "This function erases the user's hard drive" can be expressed as "Erases user's hard drive".

Example:

```
//---------------------------------------------------------------------------
// Constant kMatEpsilon:
// Description:
//    This is the epsilon value to use when determining whether a matrix is
//    orthogonal and for determining whether the scale factor is close enough
//    to unity to set the unity bool.
//---------------------------------------------------------------------------
```

4.3 Inline Comments

Use // to comment inline code. You can enhance the readability of your inline comments by aligning their starting points. For example:

```
union
{
   struct
   {
      Float t00, t01;
      Float t10, t11;
   }; // For direct element access
   Float mCompArr[2][2]; // For double index access
   Float mComps[4]; // For linear array access
};
```

would be more readable as:

```
union
{
   struct
   {
      Float t00, t01;
      Float t10, t11;
   };                        // For direct element access
   Float mCompArr[2][2];     // For double index access
   Float mComps[4];          // For linear array access
};
```

If a comment is on a line by itself, indent it as far as the indentation of the next line. *Example:*

```
   // Interface to the database parser
   void parseText( char*&      buffer,
                   size_t      size,
                   size_t&     linenum,
                   const char* filename);
```

Here are some guidelines for writing inline comments:

- Don't assume the code "explains itself," thereby making comments unnecessary. What may be self explanatory to you can be virtually unintelligible to someone else.
- If you need to make extensive comments (such as multiple occurrences of 10-line comments), you probably need a separate document to explain the code rather than just comments.
- Short comments should describe the **what** of the code (like "compute mean value" rather than the **how** (like "sum values and divide by n") unless the how is not obvious.

5. Formatting Style

These guidelines specify a consistent formatting style that helps make code more readable.

5.1 Lines of Code

5.1.1 Limit File Length

Avoid having large files. People have trouble grasping overly long files full of code. Also, long files of source code tend to slow things down. For example, compiling and linking two small files is usually faster than compiling and linking one large file.

5.1.2 Limit Line Length

Try to limit line length to 79 characters. When this is not possible (due to factors like heavily indented code with large amounts of nesting, or excessive line length due to our use of descriptive variable and function names), use the indentation styles discussed in Section 5.1.4 below.

5.1.3 Use Blank Lines

Have two blanks lines between the end of one function or class and the header comment for the next function or class.

Use a single blank line in the body of a function to separate the local variables from the code.

Use single blank lines to separate code segments inside the body of a function or class. When used appropriately, this can increase readability.

Example:

```
class tDBArray
{
   friend class tDatabase2;

public:
   size_t  NumAtoms() const { return mStore[mVal].NumUsed(); }
   tDBAtom GetAtom( size_t idx ) const;
   tDBAtom GetAtom( tDBAtomPath const& path, size_t i = 0 ) const;
   bool    AddAtom( size_t idx, tDBAtom atom );
   bool    AddAtom( tDBAtomPath const& path, tDBAtom atom );
   void    Clear();

   bool    Serialize(   tBrutlBuf& buf, bool shallow = true ) const;
   bool    Unserialize( tBrutlBuf& buf );
};
```

In resource files, leave one blank line between resource definitions. It is useful to have a small visual break between them to know when they end.

5.1.4 Use Consistent Indentation

Indent by 3 spaces (you can set this in MSDev), and don't use tabs. **Never use tabs!**

Example:

```
union
{
   struct
   {
      Float x, y, z, w;
   };
   Float mComps[4];
};
```

As mentioned in Section 5.1.2 above, it is a good idea to limit lines to a maximum of 79 characters. This is not always possible, especially in heavily indented code with large amounts of nesting, and because we try always to use descriptive variable and function names. There are two indentation techniques that are useful to handle situations like these. For instance, in a parameter list for a function or for calls to a function, the parameters may extend far to the right. In these cases, place each parameter on its own line, in one of two indention styles.

Technique 1: Place the first argument on the same line as the function name itself. Place each remaining argument on its own line, indented to the same level as first one. Place the closing parenthesis and semi-colon at the end of the line with the last argument. *Example:*

```
tLongReturnType ThisIsALongFunctionName( tArgumentType1                arg1,
                                         tArgumentType2 const          &arg2,
                                         tArgumentType3 const * const *pArg3 );
```

Technique 2: Place the first argument indented by three spaces on the next line following the function name. Place each remaining argument on its own line, indented three spaces (i.e., to the same level as first argument). Place the closing parenthesis and semi-colon on a separate line outdented from the argument list by three spaces. *Example:*

```
tLongReturnType ThisIsALongFunctionName(
   tArgumentType1                arg1,
   tArgumentType2 const          &arg2,
   tArgumentType3 const * const *pArg3
);
```

In general, indent nested `#ifdef`s, e.g.:

```
#ifdef DEBUGGING

   { some debug code here }

#  ifdef TEST_DATA_STRUCTURES
       { foo with data-test }
#  else
       { plain foo }
#  endif

#endif
```

However, it is ***not*** necessary to indent the `#include` between the `#ifndef`/`#endif` pair. Although it would make the nesting clear; e.g.,

```
#ifndef _FOO_H
#  include "foo.h"
#endif
```

in practice the three lines never vary in form and it is therefore clear that the `#include` is conditional. So, use the more usual form:

```
#ifndef _FOO_H
#include "foo.h"
#endif
```

5.2 Other Formatting Elements

5.2.1 Have AT MOST One Statement Per Line

Have at most one statement on a line. It's easy to miss a statement that's on a line choked with multiple statements. Also, it is difficult to debug code with multiple statements on a line, since most debuggers can't place a breakpoint between statements on the same line. For example:

```
if ( foo ) DoSomething();
```

should instead be:

```
if ( foo )
   DoSomething();
```

Similarly, avoid inline code in headers of the form:

```
void Foobar() { expresion; ... }
```

as in debug this is not inlined, and if you want to step into the code you can't. Sometimes you want to examine the variables before anything in the expression list is executed and that isn't possible unless you go to assembly and examine the code and determine where to set a break point after the setup. Instead, use the style:

```
void Foobar()
{
   expression;
   expression;
}
```

5.2.2 Use Blanks to Enhance Readability

Typically, **there should be a blank on each side of a binary operator**; for example "+" should be " + ". Blanks around these operators usually enhance readability. However, drop the blanks when this would improve readability, such as to emphasize operator preference in complex formulas.

Use at least one blank after each comma. For example, an argument list like "`arg1,arg2,arg3`" should be "`arg1, arg2, arg3`".

For function calls, typically use at least one blank after the left parenthesis and before the right parenthesis. For example:

```
TDGetBoundingSphere(c,center,r,radius,center,radius);
```

should be:

```
TDGetBoundingSphere( c, center, r, radius, center, radius );
```

Place spaces around elements in `if`, `for`, `while`, `switch`, etc. For example, use

```
if ( foobar == 10 )
```

```
   ...;
```

rather than

```
if(foobar==10)
   ...;
```

or even

```
if (foobar == 10)
   ...;
```

Similarly, use

```
for( int i = 0; i < 10; ++i )
   ...;
```

rather than

```
for(int i=0;i<10;++i)
   ...;
```

and so on.

The space after the `if`, `for`, etc., and before the opening "(" is **optional**.

5.2.3 Match Up Statements

Match up assignments occupying consecutive lines so that the assignment operators are in the same column. *Example:*

```
   mPrev       = 0;
   mNext       = 1;
   mDirection  = 1;
   mStartTime  = zTime;
   mSwitchTime = pMorphData->mMorphModels[mPrev].mDuration;
```

Match up the arguments of consecutive function calls and other expressions occupying consecutive lines, so that the arguments start in the same column. *Example:*

```
   FOObar(  wiggle, woggle,  wump );
   FOOrabi( fold,   spindle, mutilate );
```

Match up the arguments of long function calls. Instead of have one long line, break the call into several lines with the arguments aligned. *Example:*

```
tLongReturnType ThisFunctionHasABunchOArgs( tArgumentType1               arg1,
                                            tArgumentType2 const         &arg2,
                                            tArgumentType3 const * const *pArg3 );
```

5.2.4 Position the Opening Brace

When creating code with `if`/`while` statements and the like, make sure the opening brace is on the line **after** the conditional, such as:

```
if ( fooey == splat )
{
   ... code code code ...
}
```

instead of:

```
if ( fooey == splat ) {
   ... code code code ...
}
```

since it makes the beginning of the scope much easier to find.

5.2.5 Use This `case`/`switch` Style

Indent each case label three spaces from the opening brace of the switch. Enclose each case in braces (unless there are only one or two statements). Indent the `break` statement along with the statements in the case code block.

Example:

```
switch ( foo )
{
   case 0:
   {
      statements;
      break;
   }
   case 1:
   {
      statements;
      break;
   }
   default:
   {
      statements;
      break;
   }
}
```

6. Other Issues

6.1 Portable Data Types

Use portable data types to represent numeric variables. The currently supported portable data types include:

`int`	Signed integer (fastest for computer).
`int8`	Signed 8 bit integer.
`int16`	Signed 16 bit integer.
`int32`	Signed 32 bit integer.
`uint`	Unsigned integer (fastest for computer).
`uint8`	Unsigned 8 bit integer.
`uint16`	Unsigned 16 bit integer.
`uint32`	Unsigned 32 bit integer.
`flags8`	Stores flags, 8 bits wide.
`flags16`	Stores flags, 16 bits wide.
`flags32`	Stores flags, 32 bits wide.

> **NOTE:**
> Pentium processors slow down using `int16` and `uint16`. P6-generation processors slow down using `int8`, `uint8`, `int16`, and `uint16`. (See Michael Abrash's *Zen of Code Optimization* (Coriolis Group Books, 1994) about what the penalties are.)

It's a good idea to make code as portable as possible. Not very long ago, for example, people were developing for the Pentium, and now everyone's developing for the P6 family and Pentium III. With any luck, the code we generate now will be useful for a long while, and it'd be nice to be able to recompile our code without (much) change on other machines. It also tends to make data a little more clear and make overflow-type problems more obvious.

6.2 Argument Parameter Passing

When defining functions, use the following format for parameter passing:

```
<return type> FunctionName( <action params>,
                            <in params>,
                            <in/out params>,
                            <out params> );
```

Action parameters are the parameters on which the operation is performed (although in practice in C++, the action parameter tends to not be used—as the `this` pointer really is the action parameter).

In parameters are parameters that provide data needed for the function and are not changed by the function.

In/Out parameters are parameters that supply an important input value but are changed by the function.

Out parameters are simply used to return the results produced by the operation. (Error codes or status codes may be passed out of a function either via the out parameters or, more commonly, through the function's return value.)

6.3 Forward Declare to Minimize Mutual Dependencies

There is almost always a "Forward Declarations" section at the top of header files, where you can forward declare types used only in pointer or reference declarations in class definitions. Forward declare these types as much as possible in the headers, instead of using `includes` to other files to provide forward declarations in your headers. Doing this minimizes mutual dependencies in your .cpp files. (When you use needless `includes` in your headers, all .cpp files which include the same header automatically get a dependency on all the extraneous files the header itself includes.)

The actual .cpp files using the class and the particular types must of course include the specific header files. However, other .cpp files that also include the same header do not get needless dependencies. This results in faster compilations, since you don't have to compile the world (or a major piece of it) every time you compile. Why rebuild files that use pointers to types and/or functions defined in the header files but never actually dereference those pointers or invoke those functions?

When you compile, if you get an error in a .cpp file about using a non-defined type, explicitly include the appropriate header file in your .cpp file.

Inline Definitions Warning: This strategy for forward declaration breaks down if you make many inline definitions in a header. When you do this, the implementations often need the actual type definitions, and thus their actual header files are needed in the header in which you are putting these inlines. For this reason, only make inlines for speed-critical items. Everything else should go into the associated .cpp file so that the dependencies go there, even if it is just a one or two line implementation.

6.4 COM Considerations

6.4.1 `struct`, `class`, and `enum` Declarations

Structures, classes, and enumerated types declared for use with COM interfaces are defined in the following manner:

```
typedef struct/enum/class <name>
{
    ...
} <name>;
```

This binds the identifier *<name>* to both the `struct` and global namespaces, so the type can be referenced either by `struct` *<name>* or *<name>* . This makes the use of forward declarations possible (i.e., `typedef struct` *<name>* `;`). The reason to use this method with COM interfaces is because this code will properly compile under C and C++.

> **NOTE:**
> In code that is only going to be compiled in C++, you can define structures and classes the normal way:
>
> ```
> struct/enum/class <name>
> {
>     ...
> };
> ```

6.4.2 Pointers, Generated Objects, and Reference Counts

Don't return pointers to generated objects which have reference counts!

In standard COM, you get new interfaces to things through `QueryInterface`. You must pass in a pointer to be filled as one of the parameters; it doesn't return the pointer. Since a return of `NULL` would generally be sufficient to indicate an error, why does this work this way?

Suppose you gave all of the classes in the TD library a `MakeInstance` method, which directly returns a new instance of the object requested, with reference count of 1. That is, if you have `pOldObj` then:

```
tObjType *pNewObj = pOldObj->MakeInstance();
```

returns a new version of `pOldObj` with reference count set appropriately. However, using this syntax makes it very easy to pass `pOldObj->MakeInstance()` directly into a method that requires a `tObjType *`, ***thereby losing the pointer to the object***. If the function to which this object is passed does reference counting appropriately, then the object from `MakeInstance` will have its reference count incremented again. But, one handle to this object is lost, since the return value of `MakeInstance` was used as a temporary.

So the appropriate signature for this `MakeInstance` should be:

```
int tObjType::MakeInstance( tObjType **ppNewObj );   // Return error code
```

This forces you to make a `tObjType **` that can then be released correctly when that is appropriate, and `*ppNewObj` can be passed to the appropriate reference count function without problem.

6.5 Use the Prefix ++ Operator, not Postfix

It's best to use the prefix form of the ++ (and --) operator. There's a slight performance impact if you use the postfix version, as any container type which has ++ defined correctly needs to make a temporary copy when using postfix. You should just be in the habit of **always** using the prefix version `++foo`, regardless of `foo` type. Of course, there is the occasional reason to use postfix ++ (or --), but you can almost always restructure slightly and use the prefix version instead. So, for instance, all `for` (and `while` and `do`) loops should use the form:

```
for ( p = End(); --p >= Begin(); )
{...}
```

or

```
for ( i = 0; i < end; ++i )
{...}
```

and so on. Prefix is just better!

Note: Scott Meyers in *Effective C++* (1997; Addison-Wesley) describes the difference of prefix and postfix ++ operators and how to implement them.

6.6 `const` is Cool; `mutable`'s no Fool

6.6.1 `const` Correctness

`const` is really very easy to use. Keeping your code `const` correct is a good way to help the compiler help you. Also, just thinking about whether a parameter or method should be `const` helps you to really be aware what the data is going to be used for.

Here's when to use `const`:

- If a method doesn't change an internal member or any visible aspect of the class, declare the method itself `const`.
- If a reference or pointer type passed to a method doesn't change in the body of the method or function, declare it `const`.
- If a class has per-instance data which doesn't change once the class is initialized, declare it `const`.
- Likewise, even locals used in a procedure can be `const`!

6.6.2 `const` Syntax

Because of the syntactic equivalence between "`tFoo const`" and "`const tFoo`", to reduce confusion always use "`tFoo const`", as then the pattern will match the pattern you must use when there are pointer types involved.

`const` is pretty straight forward in its meaning, or so you might think. **Some trickiness occurs when you use `const` with pointer types**. For instance, there is a difference in meaning between:

```
tFoo const * const *ppFoo;
```

and

```
tFoo * const * const ppFoo;
```

There is an especially confusing issue among the usage of

```
tFoo const * pFoo;
```

and

```
const tFoo * pFoo;
```

and

```
tFoo * const pFoo;
```

There are two things to understand when it comes to all this, one of which is simply syntax:

> **Rule 1:** "`const tFoo`" and "`tFoo const`" are equivalent. `const` modifies the item **to its left**, unless `const` appears as the first item in a declaration, whereupon it modifies the item **to its right**.
>
> **Rule 2:** In pointer declarations, if the "`tFoo const`" syntactic version is used, then parsing the meaning is straight forward. The `const` modifies the part of the declaration **to the left** of the `const` keyword.

For example, "`tFoo const * const *ppFoo`" is a declaration indicating that both `tFoo` and `(tFoo *)` are `const`. Hence both `**ppFoo` and `*ppFoo` may not be altered.

For another example, "`tFoo * const * const ppFoo`" is a declaration indicating that both `(tFoo *)` and `(tFoo **)` are `const`. Hence both `*ppFoo` and `ppFoo` itself may not be altered.

Writing "`const tFoo * const * const ppFoo`" is the same as writing "`tFoo const * const * const ppFoo`", which, as you might guess by this point, means `tFoo`, `(tFoo *)`, and `(tFoo **)` are `const`, hence `ppFoo`, `*ppFoo`, and `**ppFoo` may not be altered. That is, nothing you can possibly refer to through `ppFoo` can change; you can only read from it. (Per our style, "`tFoo const * const * const ppFoo`" is preferred to "`const tFoo * const * const ppFoo`" when writing code.)

Remember that `&` indicates the specified variable is really a reference, so `&` must be syntactically bound to that variable name. Hence something like "`tFoo & const foo`" makes no sense. Also, when mixing with pointers, the `&` must be bound to the variable, as in "`tFoo *&pFoo`" but not "`tFoo &*pFoo`", as again the latter has no meaning.

6.6.3 `mutable` Members of `const` Methods

The `mutable` keyword is something you occasional should use in class definitions. A `mutable` member is one that can be modified even in the code of a `const` method. Suppose you have a `struct tFoo` with a mutable member `x`. Then, even if `foo` is declared `tFoo const &foo`, you can modify `foo.x`; i.e., it is an l-value. This allows you to have "hidden" members used for caching or something that needs updating even in the context of a method that really should be `const`, like some accessors.

6.7 Casting Operators are Way Cool

The C++ standard defines operators for type casting. They are:

- `reinterpret_cast`
- `dynamic_cast`
- `static_cast`
- `const_cast`

Example:

```
char *buf = reinterpret_cast<char*>( malloc( 1024 ));
```

`reinterpret_cast` is the type casting you get when you code the old way. It basically says to the compiler, "I know what type you think you have but pretend it's a pointer to something else for a minute. And just trust me on this one, OK? I know more about the pointer that you do." These types of casts are needed occasionally, and sometimes they are extremely useful. But, why use the new way? Using `reinterpret_cast` makes it easy to spot the parts of the code that use this operator. You get a ***big visual clue*** that something dangerous might be happening. Also, you can `grep` for them!

`dynamic_cast` is a very cool idea, but we do not use it since it requires `rtti` (run time type checking) to be enabled. (If we were using `dynamic_cast`, it would return `NULL` if the type to which you were casting is not derived from the pointer you have. This can be useful for downcasting safely.)

`static_cast` is very very cool as it will verify ***at compile time*** that the cast you are making is legal! If you use an old cast operator instead of `static_cast` and make a casting mistake, your code would compile just fine and then it would crash hideously somewhere, wasting everyone's time and effort.

`const_cast` is also very useful, because it's easy to spot it in your code and it's an explicit reminder about why you need to cast. Its use should really be necessary only for interfacing with non-`const` correct code (see Section 6.6 above). It again is a nice way to indicate that something unmentionable is actually occurring and points to those places where `const` correctness is not happening.

6.8 Templatized Function Usage

The ANSI C++ standard allows for templatized functions, in which the templatized type does ***not*** occur in the parameter list. For instance:

```
template <class T>
T * CreateT();
```

or

```
template <class T>
int SomeFunc( float f );     // No T whatsoever; it simply occurs in the body
```

However, Microsoft's compiler cannot handle such constructs. Worse, it accepts these constructs without issuing errors and then proceeds to do the wrong thing with them! So beware.

Our workaround is to make a templatized class which has an `operator()`. For instance, this:

```
template <class T>
T * CreateT();
```

can be structured as:

```
template <class T>
struct CreateT
{
   T * operator()();
};
```

The only difference is that you have to type "`CreateT<type>()()`" instead of `CreateT<type>()`, which would work if the Microsoft compiler handled templatized functions correctly.

Similarly, this:

```
template <class T>
int SomeFunc( float f );     // No T whatsoever; it simply occurs in the body
```

can be structured as:

```
template <class T>
struct SomeFunc
{
   int operator()( float f );
};
```

and would be called like this:

```
SomeFunc<type>()( f );
```

6.9 Define Copy Constructors and Assignment Operators!

Always define a copy constructor and an assignment operator for any non-trivial class. If copying should not be allowed (i.e., it is too expensive to do a deep copy), then define the functions anyway but assert that they are never called. Or make them private.

Why? The C++ compiler will generate these functions if you do not define them, but the compiler-generated versions do not always work correctly. (A client of your class could inadvertently call the assignment operator or copy constructor in a variety of ways.)

Thus, in general it's best to write versions of these functions that will work correctly.

6.10 Use class Keyword in friend Declarations!

Don't forget the `class` keyword in `friend` declarations.

Incorrect:

```
friend tIter
```

Correct:

```
friend class tIer
```

While DevStudio maybe more forgiving, some compilers like CodeWarrior won't compile syntactically incorrect code.

6.11 Headers

6.11.1 Header Macros

Public header file names need only be unique in the set of that libraries headers, but the `#define` wrapping that header must have a unique name in the global namespace of all libraries. Therefore, the `#define` used to wrap a public header must have the form:

```
_LIBNAME_FILENAME_H
```

unless `LIBNAME` and `FILENAME` are the same, in which case:

```
_LIBNAME_H
```

is the appropriate macro with which to wrap the file. So for instance ztl\ztlbuf.h has `#define _ZTL_ZTLBUF_H` at the top of it, while mem\mem.h should have just `#define _MEM_H`.

Internal library (local) header files must have a preceding "_" (underscore) on their name; for example, _foobar.h. These headers are local versions of the headers that are sometimes needed in the src directory.

The `#define` used to wrap an internal library (local) header must have the form:

```
_FILENAME_H
```

where `FILENAME` is the root of the name. So for instance _foobar.h has `#define __FOOBAR_H` (note the double initial underscore; the first for the macro and the second for the file name—all local headers start with _, see Section 3.1.1).

6.11.2 Including Headers

Use <> to include public headers and use "" to include local headers. Note that when files within a library include public headers from that same library, they need to be included using <>.

6.12 Compiler Issues

6.12.1 C Run-Time Functions

Please be very careful when using CRT functions that start with a leading underscore (like _findfirst, _filelength, etc). These are not "official" CRT functions and hence not necessarily supported by some compilers like CodeWarrior (_findfirst is, _filelength and _exit are not for instance). Also, do not use CRT functions starting with _, since they are much less likely to be portable. It should be easy enough to write our own versions of such things.

6.12.2 Use string.h, Not memory.h

memory.h (for `memcmp`, `mempcy`, etc.) apparently is not universal. Use string.h instead (it also defines these things).

6.12.3 Avoid Using goto

Do not use `goto`s in your code, even though they are (very) occasionally useful. They can cause problems. For example, a code generation problem with CodeWarrior was tracked down to the use of a `goto`. It was causing the floating point stack not to be cleaned up appropriately.

6.12.4 Include porttype.h in Translation Units

Make sure porttype.h is included in every one of your translation units, preferably at the top of the code. This is because CodeWarrior requires a pragma to be defined, and this seems the best place to put it. The pragma allows for certain C++ extensions, in particular for anonymous unions, which aren't officially part of ISO C++.

6.12.5 Do Not Use In-Statement Declaration

Be aware that C++ conforming compilers like CodeWarrior do **not** accept old-style `for` loop iteration declaration parameters used outside of the scope of the loop. That is:

```
for( int i; i < 10; ++i )
{
   ...
}
```

The Microsoft compiler will accept the use of `i` after the end of the `}`, which according to standard C++ is the end of scope of `i`. So if you want to use `i` later, declare it before the first `for` loop it is used in.

(Note that the Microsoft compiler forces you not to redeclare the same iterator if you used in-statement declaration. For example, if you want to do two `for` loops that use the same iterator and you declare the iterator within the `for`, then DevStudio forces you to not redeclare the iterator:

```
{
   for ( int i = 0; ++i < 10; )
   {
      ...
   }
   for ( int i = 0; ++i < 10; )    // MSDEV will not compile this;
                                   // you *must* take out the int
   {
      ...
   }
}
```

So the moral is to just not use in-statement declaration...)