

Recommendations on Coding Standard.

Project: GAPI Benchmark Suite. Edition August 1, 2000.

Contents

<i>Contents</i>	1
INTRODUCTION	2
SYNTAX AND NAMING CONVENTIONS	2
<i>Common identifier notation</i>	2
<i>Object Names</i>	4
<i>Type Names</i>	4
<i>Method and Function Names</i>	4
<i>Class Attribute Names</i>	5
<i>Method Argument Names</i>	5
<i>Local Variables (on stack)</i>	6
<i>Global Variables</i>	6
<i>Pointer Names</i>	6
<i>Reference Variables</i>	6
<i>Global Constants and #define macros</i>	6
<i>Naming class files</i>	6
DOCUMENTATION AND COMMENTING	7
<i>Comments</i>	7
<i>Gotcha keywords</i>	7
LAYOUT AND FORMATING	9
<i>Standard File Header and Method Header</i>	9
<i>Class Layout</i>	9
<i>Formatting Methods with Multiple Arguments</i>	11
<i>Line length</i>	11
<i>Indentation, Tabs, Space Policy</i>	11
<i>Braces {} Policy</i>	12
<i>Parens () with Keywords and Function names</i>	12
<i>Some Statements Formatting</i>	13
<i>Alignment of Declaration Block</i>	14
MISCELLANEOUS	15
<i>No Magic Numbers</i>	15
<i>Error Return Check Policy</i>	15
<i>Boolean and CString Types</i>	15
<i>Use #if not #ifdef</i>	16
PORTABILITY	16
REFERENCES	17

Introduction

The open-source project is teamwork. As part of a team, your code needs to be understood by its members and vice versa. The main goal of the recommendation is to improve readability and thereby the understanding and the maintainability and general quality of the code. It is impossible to cover all the specific cases in a general guide and the programmer should be flexible.

Any violation to the guide is allowed if it enhances readability.

The rules can be violated if there are strong personal objections against them.

Syntax and Naming Conventions

Common identifier notation

General identifier format based on Microsoft version of Hungarian notation and looks like:

```
<scope>_<type><Name>
```

All names should be written in English. Identifier naming conventions make programs more understandable by making them easier to read. They also give information about the purpose of the identifier. Identifiers should be meaningful. That is, they should be easy to understand and provide good documentation about themselves.

Avoid abbreviations. Abbreviations make it hard for others to remember the spelling of your functions and variables. They also obscure the meaning of the code that uses them.

Single character variable names should be avoided because of the difficulty of maintaining code that uses them. However, single character names may be appropriate for variables that are essentially meaningless, such as dummy loop counters with short loop bodies or temporary pointer variables with short lifetimes.

Avoid variables that contain mixtures of the numbers 0 & 1 and the letters O and l, because they are hard to tell apart.

Avoid identifiers that differ only in case, like foo and FOO. Having a type name and a variable differing in only in case (such as `String string;`) is permitted, but discouraged.

Common abbreviations should not be uppercase when used as name. When confronted with a situation where you could use an all upper case abbreviation instead use an initial upper case letter followed by all lower case letters.

For example, `OpenDvdPlayer();` // NOT: `OpenDVDPlayer();`

All names can be based on abbreviations shown in Table 1 and Table 2.

Table 1: Required abbreviations.

Accum	Accumulation Buffer
Attrib	Attribute
CCW	Counter Clockwise
Clip	Clipping
Coeff	Coefficient
Coord	Coordinate
Cnt	Count
CW	Clockwise
Decr	Decrement
Dflt	Default
Dim	Dimension
Dst	Destination
Env	Environment
Eval	Evaluate
Exp	Exponential
Func	Function
Gen	Generate
Incr	Increment
Info	Information
List	Display List
LOD	Level of Detail
LSB	Least significant bit
Mag	Magnify
Max	Maximum
Min	Minify
Mult	Multiply
Num	Number
Op	Operation
Ortho	Orthographic
Pname	ParameterName
Pos	Position (function names only)
Quad	Quadrilateral
Rect	Rectangle
Ref	Reference
RGBA	Red Green Blue Alpha
Src	Source
Tex	Texture (function names only)

Table 2: Compound words.

Antialias
 Bitfield
 Bitmap
 Cutoff
 Doublebuffer
 Framebuffer
 Modelview
 Subpixel
 Viewport
 Writemask

Object Names

Names representing objects must be in mixed case starting with upper case, except the second word of compound words. Limit object names to two words. Use nouns for object names. (Just a recommendation). First character in a name is upper case. No underbars ('_').

Example:

```
class BaseObject;
```

Type Names

Names representing types must be in mixed case starting with upper case, except the second word of compound words. Limit interface names to two words. Use nouns for type names. (Just a recommendation).

Prefix class names with 'C' for C++ classes and with 'J' for Java classes.

Prefix interface names with 'I'.

Prefix structure names with 'S'. We use structures as simple classes with all members public by default and without any methods except trivial initializer.

Prefix other type names with 'T'.

Example:

```
CBaseObject Item;  
IDrawable GetObject();  
SConfig ConfigChunk;  
typedef unsigned char TByte;
```

Method and Function Names

Usually every method and function performs an action, so the name should make clear what it does. Limit function names to two words, three for function that return state. Begin each word with a capital letter, except the second word of compound words. Use verb-noun (`DeleteList`) and adjective-noun (`EdgeFlag`) formats. Functions should be named after what they return and procedures after what they do. (Just a recommendation).

All functions with global scope should be prefixed by "::" on every access (`::CreateFile`), except those with corporate prefix (`glGetVersion`) and macros.

The name of the object is implicit, and should be avoided in a method name.

```
Line->GetLength(); // NOT: Line->GetLineLength();
```

Suffixes are sometimes useful (see Table 1):

Max – to mean maximum value something can have.

Cnt – the current count of a running count variable.

Num - suffix should be used for variables representing an entity number. `VertexNum`, `ColorNum`.

The first word of predicate function names should be `Is`.

A function that creates a new instance of an object from scratch begins with the word `Create`. A function to destroy such an object begins with the word `Destroy`. There should seldom be a need for an explicit `Destroy` function as reference counting with a `Release` function is typically used to control lifetime.

Functions to modify state start with word `Set`. Accessor functions start with the word `Get`. For example, `SetColor` and `GetColor`.

Accessor functions that must perform a search based on the input parameter start with the word `Find` rather than `Get`. (`Vertex->FindNearest()`; `Matrix->FindMinElement()`;)

The word `Compute` should be used in functions where something is computed.

```
ValueSet->ComputeAverage(); Matrix->ComputeInverse();
```

The word `Initialize` should be used where an object or a concept is established.

```
Printer->InitializeFontSet();
```

Complement names must be used for complement functions: `Get/Set`, `Add/Remove`, `Create/Destroy`, `Start/Stop`, `Insert/Delete`, `Increment/Decrement`, `Old/New`, `Begin/End`, `First/Last`, `Up/Down`, `Min/Max`, `Next/Previous`, `Open/Close`, `Show/Hide`. Names `AddRef` and `Release` are reserved for `IUnknown` interface.

Class Attribute Names

Attribute names should be prepended with the prefix `m_`. Prepending `m_` prevents any conflict with method names. Attributes must never be declared public.

Example:

```
class BaseObject
{
public:
    void Add(void);
private:
    UINT m_RefCnt;
}
```

Method Argument Names

The first character should be lower case. All word beginnings after the first letter should be upper case as with object names. You can always recognize which variables are passed in. You can use names similar to object names without conflicting with object names.

Input parameters precede output parameters.

Example:

```
CCursor CursorArrow = new CCursor(CURSAR_ARROW);
...
Bool InitializeVideo(CCursor& cursorArrow);
```

Local Variables (on stack)

Use all lower case letters. All local variables should be declared in the beginning of their scope and in the smallest scope possible. Related variables of the same type can be declared in a common statement.

Example:

```
int value, counter, i;
```

Global Variables

Global variables should be prepended with the prefix `g_`. Use of global variables should be minimized.

Example:

```
HWND g_MainWindow;
```

Pointer Names

Pointers should be prepended by a 'p' in most cases. Place the * close to the pointer type not the variable name. In C++ the pointer modifier only applies to the closest variable, not all of them. You want to have one declaration per line anyway so you can document each variable.

Example:

```
int* pValue;
```

Reference Variables

References should be prepended with 'r'. Place the & close to the type name not the variable name.

Example:

```
double& rXAxis;
```

Global Constants and #define macros

Global constants should be all upper case with '_' separators.

Example:

```
const int A_GLOBAL_CONSTANT = 5;  
MAX_ITERATIONS, COLOR_RED
```

Naming class files

Each class definition should be in its own file where each file is named directly after the class's name: `ClassName.h`

In general, each class should be implemented in one source file: `ClassName.cpp`

Documentation and Commenting

Comments

Use `//` for all comments, also multiline comments. Since multilevel C-commenting is not supported, using `//` comments ensure that it is always possible to out comment entire sections of a file using `/* */` for debugging purposes etc..

All comments should be written in English. Write code clearly enough not to need comments. Any section of code that are tricky enough to require comment should be rewritten so that they are clear enough for their function to be understood without comments.

Each part of the project has a specific comment layout.

Prefer block comments over trailing comments. Use block comments regularly. Only use trailing comments for special items.

Very short comments may appear on the same line as the code they describe, but should be tabbed over far enough to separate them from the statements. Trailing comments are useful for documenting declarations

Sometimes large blocks of code need to be commented out for testing. The easiest way to do this is with an `#if 0` block:

```
void ExampleFunction()
{
    great looking code

#if 0
    lots of code
    ...
#endif

    more code
}
```

Use header comments for all files. (See below)

Use header comments for all functions. The size of the comment should mirror the size and complexity of the code. Class interface functions and procedures should be commented in the header. Use JavaDoc commenting style (`///` and `/** ... */`).

It is recommended a program like cdoc (<http://www.joelinoff.com/ccdoc>) be used to document C++ classes, method, variables, functions, and macros. The documentation can be extracted and put in places in a common area for all programmers to access.

Gotcha keywords

Explicitly comment variables changed out of the normal control flow or other code likely to break during maintenance. Embedded keywords are used to point out issues and potential problems. Consider a robot will parse your comments looking for keywords, stripping them out, and making a report so people can make a special effort where needed. Make the gotcha keyword the first symbol in the comment. Comments may consist of multiple lines, but the first line should be a self-containing, meaningful summary. The writer's name (nickname) and the date of the remark should be part of the comment. Embedding date information allows

other programmer to make this decision. Embedding who information lets us know whom to ask.

:TODO: description – means there's more to do here, don't forget.

:BUG: [bug ID] description – means there's a known bug here, explain it and optionally give a bug ID.

:KLUDGE: - when you've done something ugly say so and explain how you would do it differently next time if you had more time.

:TRICKY: - tells somebody that the following code is very tricky so don't go changing it without thinking.

:WARNING: - beware of something.

:COMPILER: - sometimes you need to work around a compiler problem. Document it. The problem may go away eventually.

Example:

```
// :TODO: km 20000810: possible performance problem
// We should really use a hash table here but for now we'll
// use a linear search.

// :KLUDGE: km 20000810: possible unsafe type cast
// We need a cast here to recover the derived type. It should
// probably use a virtual method or template.
```

Layout and Formatting

Standard File Header and Method Header

```

////////////////////////////////////
// File: "BaseObject.cpp"
//
// Header file: "BaseObject.h"
//
// (C) Copyright note and disclaimer (optional)
//
// @Author: John Smith (js) jsmith@rightmark3d.org
//
// Specification: "http://righmark3d.org/documents/BaseObject.htm"
//
// Purpose:
//   Implements the basic CBaseObject class, which defines
//   functionality common to all names.
//
// History:
//   2000/05/10 created. John Smith (js)
//   2000/05/15 added some functions. John's dad (jd)
//   2000/05/20 fixed some bugs. John's mom (jm)
//
////////////////////////////////////

/**
 * Purpose:
 *   Exports data from a mesh file.
 *
 * @param: (i)meshData - the memory bufferfile to read
 * @param: (o)fileName - the file to export
 *
 * @return: false if an error occurred, true otherwise
 */

```

Class Layout

A common class layout is critical from a code comprehension point of view and for automatically generating documentation. This layout based on template used by MSVC++ class Wizard.

```

// SYSTEM, PROJECT, LOCAL INCLUDES
//

// FORWARD REFERENCES
//

class XX
{
public:
// LIFECYCLE

/**
 * Default constructor.

```

```

*/
    XX(void);

/**
 * Copy constructor.
 *
 * @param from The value to copy to this object.
 */
    XX(const XX& from);

/**
 * Virtual Destructor.
 */
    ~XX(void);

// OPERATORS

/**
 * Assignment operator.
 *
 * @param from The value to assign to this object.
 *
 * @return A reference to this object.
 */
    XX& operator=(XX& from);

// OPERATIONS, ACCESS, INQUIRY

protected:
private:
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

```

You shall always initialize variables. Every time.

More problems than you can believe are eventually traced back to a pointer or variable left uninitialized. C++ tends to encourage this by spreading initialization to each constructor.

Objects with multiple constructors and/or multiple attributes should define `Initialize()` method to initialize all attributes. If the number of different member variables is small then this idiom may not be a big win and C++'s constructor initialization syntax can/should be used.

Do not do any real work in an object's constructor. Inside a constructor initialize variables only and/or do only actions that can't fail. Again, create and use `Initialize()` method for an object that completes construction. `Initialize()` should be called after object instantiation.

Example:

```

class Device
{
public:
    Device()    { /* initialize and other stuff */ }
    HRESULT Initialize() { return E_FAIL; }
};

```

```
dev = new Device();
if (FAILED(dev->Initialize())) exit(1);
```

Formatting Methods with Multiple Arguments

We should try and make methods have as few parameters as possible. If you find yourself passing the same variables to every method then that variable should probably be part of the class. When a method does have a lot of parameters format it like this:

```
int AnyMethod( int arg1    // description
              , int arg2    // description
              , int arg3    // ...
              , int arg4
              );
```

When we add/remove the last parameter, we should not think about comma.

Line length

Avoid lines longer than 80 characters, since they are not handled well by many terminals and tools. Code parts intended to use in printed documentation or papers should have no more than 70 characters. When an expression will not fit on a single line, break it according to these general principles:

- Break before a comma. Comma should start a new line.
- Break before an operator.
- Prefer high-level break to low-level break.

Example:

```
if (( LongLogicalTest1 || LongLogicalTest2)
    && LongLogicalTest3 )
{
    ...;
}

a = ( long_identifier_term1 - long_identifier_term2)
    * long_identifier_term3;

m_hWndMain = ::CreateWindowEx(WS_EX_APPWINDOW|WS_EX_TOPMOST
                              , m_szWindowClass
                              , strCaption
                              , WS_VISIBLE|WS_POPUPWINDOW|WS_CAPTION
                              , CW_USEDEFAULT, CW_USEDEFAULT
                              , CW_USEDEFAULT, CW_USEDEFAULT
                              , NULL
                              , NULL
                              , m_hInstance
                              , NULL);
```

Indentation, Tabs, Space Policy

Indent using 4 spaces for each level. Do not use tabs, use spaces. Indent as much as needed, but no more. There are no arbitrary rules as to the maximum indenting level. If the indenting level is more than 4 or 5 levels you may think about factoring out code.

There should be only one statement per line unless the statements are very closely related.

The function type should be put in the left column immediately above the function name.

Example:

```
int _stdcall
MyClass::SomeFunction()
{
    if (something bad)
    {
        if (another thing bad)
        {
            while (more input)
            {
            }
        }
    }
}
```

Conventional operators should be surrounded by a space character. C++ reserved words should be followed by a white space. Commas should be followed by a white space. Colons should be surrounded by white space. Semicolons in for statements should be followed by a space character.

Braces {} Policy

Braces shall be aligned using Ulman style where the braces are at the same scope as the statement that proceeds them and the code within the braces is indented one level. Braces are always on a line by themselves.

Example:

```
    if (very_long_condition && second_very_long_condition)
    {
        ...
    }
    else if (...)
    {
        ...
    }
```

To move from block to block you just need to use cursor down and your brace matching key. No need to move to the end of the line to match a brace then jerk back and forth.

Parens () with Keywords and Function names

Do not put parens next to keywords. Put a space between. Do put parens next to function names. Do not use parens in return statements when it's not necessary.

Example:

```
    if (condition)
    {
    }

    while (condition)
    {
    }

    strcpy(s, s1);

    return 1;
```

Some Statements Formatting

if then else formatting.

One common approach is:

```

if (condition)                // Comment
{
}
else if (condition)          // Comment
{
}
else                          // Comment
{
}

```

If you have `else if` statements then it is usually a good idea to always have an `else` block for finding unhandled cases. Maybe put a log message in the `else` even if there is no corrective action taken.

Always put the constant on the left hand side of an equality/inequality comparison. For example:

```

if (0 == errorNum)
{
    cout << "no errors.";
}

```

By reversing the order of the equality test and putting the constant 0 on the left, it's impossible to make the error. If programmer mistypes the equality test by using the assignment operator, then code will automatically refuse to compile.

Do not use one-line `if` clause. Use compounded `if` statements. By enclosing the `if` clause with curly braces, we have made it abundantly clear what we mean. This is defensive coding for situation when programmer needs to add an extra statement to the `if` clause. This guideline has no exception for one-liner:

```

if (true == BaseObject->Action()) { BaseObject->Draw(); }

```

switch formatting.

Falling through a case statement into the next case statement shall be permitted as long as a comment is included. The `default` case should always be present and trigger an error if it should not be reached, yet is reached. If you need to create variables put all the code in a block.

Example:

```

switch (...)
{
case 1:
    ...
    // :WARNING: FALL THROUGH

case 2:
{
    int v;
    ...
}
}

```

```
break;

default:
}
```

Use of goto, continue, break and ? :

Goto statements should be used sparingly, as in any well-structured code. Perhaps, the only good place for goto statement is the one-shot error cleanup.

Example:

```
for (...)
{
    while (...)
    {
        ...
        if (disaster)
            goto error;
    }
}
...
error:
    clean up the mess
```

When a goto is necessary, the accompanying label should be alone on a line and to the left of the code that follows. The goto should be commented (possibly in the block header) as to its utility and purpose.

The use of break and continue in loops should be minimized. Mixing continue with break in the same loop is a sure way to disaster. Explicit continue should be used in empty loops:

```
while (*p++ = *q++)    // NOT: while (*p++ = *q++);
    continue;
```

This makes it obvious that the loop is left empty intentionally.

The trouble is people usually try and stuff too much code in between the ? and :. Here are a couple of clarity rules to follow:

- Put the condition in parens so as to set it off from other code.
- If possible, the actions for the test should be simple functions.
- Put the action for the then and else statement on a separate line unless it can be clearly put on one line.

Example:

```
(condition)
    ? long statement
    : another long statement;
```

Alignment of Declaration Block

Block of declarations should be aligned. Similarly, blocks of initialization of variables should be tabulated. The '&' and '*' tokens should be adjacent to the type, not the name.

Example:

```

DWORD    m_Dword
DWORD*   m_pDword
char*    m_pChar
char     m_Char

m_Dword  = 0;
m_pDword = NULL;
m_pChar  = 0;
m_Char   = NULL;

```

Miscellaneous

No Magic Numbers

The use of magic numbers in the code should be avoided. Numbers other than 0 and 1 should be considered declared as named constants instead. A magic number is a bare-naked number used in source code. It's magic because no one has a clue what it means including the author inside 3 months. For example:

```

if      (22 == foo) { StartThermoNuclearWar(); }
else if (19 == foo) { RefundLotsoMoney(); }
else if (16 == foo) { InfiniteLoop(); }
else           { CryCauseImLost(); }

```

In the above example what do 22 and 19 mean? If there was a number change or the numbers were just plain wrong how would you know?

Instead of magic numbers, use a real name that means something. You can use `#define` or constants or enums as names. Which one is a design choice? For example:

```

#define  PRESIDENT_WENT_CRAZY  (22)
const int WE_GOOFED = 19;
enum
{
    THEY_DIDNT_PAY = 16
};

if      (PRESIDENT_WENT_CRAZY == foo) { StartThermoNuclearWar(); }
else if (WE_GOOFED           == foo) { RefundLotsoMoney(); }
else if (THEY_DIDNT_PAY      == foo) { InfiniteLoop(); }
else           { HappyDaysIknowWhyImHere(); }

```

Error Return Check Policy

Check every system call for an error return, unless you know you wish to ignore errors. For example, `printf()` returns an error code but rarely would you check for its return code. In which case you can cast the return to `(void)` if you really care. Include the system error text for every system error message. Use `HRESULT` from Win32 `WINERROR.H`. To check return state use `FAILED()` and `SUCCEEDED()` macros.

Boolean and CString Types

Use native boolean type `bool` defined in C++ standard. Always compare result with `false`.

Use CString class instead of bare C strings. CString object names should be prepended with the prefix `str`. Prepending `str` prevents any conflict with C string names prepended with `sz`.

Example:

```
bool    IsFullScreen = false;
CString strCompany;
char    szVendor;    // avoid C strings
```

Use #if not #ifdef

If you really need to test whether a symbol is defined or not, test it with the `defined()` construct, which allows you to add more things later to the conditional without editing text that's already in the program:

```
#if !defined(USER_NAME)
    #define USER_NAME "john smith"
#endif
```

Portability

Nothing to say at the moment..

References

- [1] C++ Coding Standard, Todd Hoff
<http://www.cs.umd.edu/users/cml/cstyle/CppCodingStandard.html>
- [2] Code Complete, Steve McConnell - Microsoft Press
- [3] Wildfire C++ Programming Style, Keith Gabryelski, Wildfire Communications Inc.
<http://www.wildfire.com/~ag/Engineering/Development/C++Style/>
- [4] C++ Programming Style Guidelines. Geotechnical Software Services.
<http://geosoft.no/style.html>
- [5] C++ Coding Standards. The Corelinux Consortium
<http://corelinux.sourceforge.net/doc/cppstnd.html>
- [6] Syntax Rules for OpenGL Extensions.
<http://oss.sgi.com/projects/ogl-sample/registry/doc/syntaxrules.txt>