[draft] IPX Coding Convention

Contents

[Coding Style 3](#_Toc336355201)

[Spacing and layout 3](#_Toc336355202)

[Header file 6](#_Toc336355203)

[Coding Standards 7](#_Toc336355204)

[API Design Principle 7](#_Toc336355205)

[Include Files 8](#_Toc336355206)

[Header File Dependencies 8](#_Toc336355207)

[References vs. pointers 9](#_Toc336355208)

[class vs. struct 9](#_Toc336355209)

[Function overloading 9](#_Toc336355210)

[Operator overloading 10](#_Toc336355211)

[Templates 10](#_Toc336355212)

[Inheritance 10](#_Toc336355213)

[Casts (C-style, function-style, modern static\_cast<>…) 11](#_Toc336355214)

[memcpy, memset, ZeroBytes, etc.. on classes 11](#_Toc336355215)

[Constructors and Destructors 11](#_Toc336355216)

[malloc vs. new 12](#_Toc336355217)

[Macros 12](#_Toc336355218)

[No magic number in code 12](#_Toc336355219)

[Inline Functions 12](#_Toc336355220)

[Namespaces 12](#_Toc336355221)

[Nonmember, Static Member, and Global Functions 13](#_Toc336355222)

[Local Variables 13](#_Toc336355223)

[Static and Global Variables 13](#_Toc336355224)

[Naming convention 14](#_Toc336355225)

[Standard Type Prefix 14](#_Toc336355226)

[String Prefix 14](#_Toc336355227)

[Generic Prefix 15](#_Toc336355228)

[Scope Prefix 16](#_Toc336355229)

[OACR compliant buffer size name convention 16](#_Toc336355230)

[Comments 17](#_Toc336355231)

[Location 17](#_Toc336355232)

[Content 18](#_Toc336355233)

# Coding Style

## Spacing and layout

1. Don't indent the outer level {.

void Foo(void)  
{  
}

1. Indent first level code.

void Foo(void)

{

Bar();

}

1. Always use compound statement

To avoid the if-else problem pointed out in programming language, we should use compound statement for all if and else. So it must have "{}" for if or else.

if (expr)

{

return;

}

If there is any "else if" statement, we must attach a "else" as the end compound statement. No matter this else is an empty statement or not. And for the empty block, we should add /\*empty block\*/ inside to let other knows it is intended, not an accident or wrong merge.

if (expr1)

{

…

}

else

{

…

}

if (expr1)

{

...

}

else if (expr2)

{

...

}

else // an end “else” is required in “if-else if” statement

{ // even if the “else” statement is empty.

/\*empty block\*/

}

1. Avoid putting multiple statements on one line.

a=1;b=2; /\* don't do this \*/

1. Avoid using 1 line if statements or loops.

if (fFoo) return; // don't do this

1. The better way to define a pointer variable should be:

int\* pVal; // better, because int\* looks more like A TYPE

int \*pVal; // worse

1. Use the following indentation for switch statements.

switch (foo)  
{  
case 1:  
 Bar();  
 break;  
default:  
 Foo();

break;  
}

1. Use braces on if, for, and while, where the conditional takes more than one line. The operator should be at the end of each line.

if (cbFoo == 0 || cbFoo >= cbFooMax **||**

(hfoo = HAllocateCb(cbFoo)) == hNil)

{

return;

}

if (cbFoo == 0 ||

cbFoo >= cbFooMax ||

(hfoo = HallocateCb(cbFoo)) == hNil)

{

return;

}

1. Avoid using a label on the same line as code. In addition, place labels on the far left, and do not align them with the surrounding code.

LFooWrong: if (fBar) // don't do this  
 ;  
LFooRight:  
 if (fBar) // do this  
 ;

1. Classes and namespaces should use non-indented block characters.

namespace  
{  
 class FOO  
 {  
 }  
}

1. Always declare the return type of a function; always put the return type on the same line.

bool FFoo(bool fPrintPreview);

void TakeAHikeBucko();

1. The declaration of each parameter should be on a separate line if the number of parameters makes the line exceed 80 characters. Generally, all local variables should be placed on a separate line unless the same logical type. Use your judgment.

bool FFoo(MMO mmo,

int a,

int b)

1. Use tabs for indentation. Tabs should equal four spaces. To avoid visual problems in other tools, we need to set "insert spaces" instead of "keep tabs" in VC’s IDE option.
2. Break long expressions between logical operators. Currently, most of the code favors the logical operators being on the right, but either way is okay. Lines should not exceed 80 characters unless splitting a line would make it very difficult to read.

if (a && b ||

c && d ||

e && f)

if (a && b

|| c && d

|| e && f)

1. Use parentheses on a judgment basis on complicated expressions and possibly to avoid an error, but do not use more parentheses than necessary to understand the expression.

((dk | fdk) == x) // has a completely different meaning than

(dk | fdk == x)

if (a && b || c && d) // does not need any extra parens

1. Use empty parentheses for functions with no parameters.
2. Spaces should be avoided in the following places:
3. inside parentheses with one argument, e.g., "(int iBlah)";
4. between the function name and the left parenthesis, e.g., "FFoo(";
5. between the left parenthesis and the first argument, e.g., "(a, b)"
6. between a variable and a comma (see above example);
7. between a variable and the right parenthesis (see above example);
8. between a variable and a semicolon;
9. next to arrows, e.g., a->b;
10. next to variable/dot combinations, e.g., a.b.
11. in empty function parameter parentheses, e.g., JustDoIt()
12. Spaces should be used in the following places:

* surrounding equals signs, e.g., a\_=\_b;
* between argument/variable combinations in parentheses, e.g., (a,\_b,\_c)
* around keywords, e.g., if\_( . . . ).

## Header file

Every file begins with a header comments and layout similar to the following:

#pragma once // Always use #pragma once in a header  
/\*--------------------------------------------------------------------Microsoft IPX

Copyright (C) 2011-2012 Microsoft Corporation. All rights reserved.

%%File: PROPS.H  
%%Unit: Core  
%%Contact: hasun

%%History: 2012 hasun created

All character and paragraph properties.  
  
 table properties in table.h  
 fields in field.h  
 borders in border.h  
 patterns in pattern.h  
  
--------------------------------------------------------------------\*/  
// include public headers to get needed defintions

// all header files should be self-sufficient  
#include "file.h"

// forward declare types rather than include unnecessary headers

struct CHP;

struct CESPRM;

// public constant and struct declarations; e.g.   
#define FTCDEFAULT 0  
  
// extern statements for public global variables; e.g.  
extern CESPRM g\_rgdnsprm[];  
  
// procedure declarations for public procedures; e.g.  
void ApplySprm(char \*hpprl, int sprm, int val, CHP \*prgbProps);

# Coding Standards

## API Design Principledee3

* 1. Out-parameters:

Which is best for out-parameters, pointers or references?

void getHsv(int \*h, int \*s, int \*v) const

void getHsv(int &h, int &s, int &v) const

Most C++ books recommend references whenever possible, according to the general perception that references are “safer and nicer” than pointers. In contrast, we tend to prefer pointers because they make the user code more readable. Compare:

color.getHsv(&h, &s, &v);

color.getHsv(h, s, v);

Only the first line makes it clear that there’s a high probability that h, s, and v will be modified by the function call.

* 1. The Boolean Parameter Trap

Boolean parameters often lead to unreadable code. In particular, it's almost invariably a mistake to add a bool parameter to an existing function. For example the function repaint(), which takes an optional bool parameter specifying whether the background should be erased (the default) or not. This leads to code such as

widget->repaint(false);

which beginners might read as meaning, "Don't repaint!"

An obvious solution is to replace the bool parameters with enum types.

widget->repaint(MODE::EraseBackgroud);

widget->repaint(MODE::KeepBackgroud);

* 1. Function Parameter Ordering

When defining a function, parameter order is: inputs, then outputs. This is not a hard-and-fast rule. Parameters that are both input and output (often classes/structs) muddy the waters, and, as always, consistency with related functions may require you to bend the rule.

## Include Files

When using include files never use relative paths to include those files.

//Correct:

#include <immex.h>

#include <rulerctl.h>

#include <fooey.h>

//Wrong:

#include “..\..\ime\inc\immex.h”

## Header File Dependencies

Don't use an #include when a forward declaration would suffice.

When you include a header file you introduce a dependency that will cause your code to be recompiled whenever the header file changes. If your header file includes other header files, any change to those files will cause any code that includes your header to be recompiled. Therefore, we prefer to minimize includes, particularly includes of header files in other header files.

You can significantly reduce the number of header files you need to include in your own header files by using forward declarations. For example, if your header file uses the File class in ways that do not require access to the declaration of the File class, your header file can just forward declare class File; instead of having to #include "file/base/file.h".

How can we use a class Foo in a header file without access to its definition?

* We can declare data members of type Foo\* or Foo&.
* We can declare (but not define) functions with arguments, and/or return values, of type Foo. (One exception is if an argument Foo or const Foo& has a non-explicit, one-argument constructor, in which case we need the full definition to support automatic type conversion.)
* We can declare static data members of type Foo. This is because static data members are defined outside the class definition.

On the other hand, you must include the header file for Foo if your class subclasses Foo or has a data member of type Foo.

Sometimes it makes sense to have pointer (or better, scoped\_ptr) members instead of object members. However, this complicates code readability and imposes a performance penalty, so avoid doing this transformation if the only purpose is to minimize includes in header files.

Of course, .cpp files typically do require the definitions of the classes they use, and usually have to include several header files.

## References vs. pointers

1. Use const references where it helps. Avoid non-const references;
2. Use a pointer, to indicate that a value may be changed.

## class vs. struct

Important semantic difference

* Struct (POD – plain old data)
  + No associated code
  + Pure data container (e.g. predefined tables, serialization)
* Class (object)
  + To encapsulate data & code

The obvious distinction between classes and structs is the encapsulation that goes along with classes. If you write a struct, expect that countless arbitrary pieces of code that you did not originally intend to will be manipulating the data in the struct in ways that you did not originally intend. Intent is very important in Object-oriented programming. Establishing the initial intent of your design using the public, private and protected attributes clues future developers into how to evolve your code without creating bugs. See Martin Fowler’s book, *Refactoring*.

## Function overloading

Never overload functions. An overloaded function looks like this:

void DoSomething(FOO \*pfooBar);  
void DoSomething(BAZ \*pbazBlat);

In this case, it is better to follow the Hungarian specification for function declarations to disambiguate parameters.

void DoSomethingFoo(FOO \*pfooBar);  
void DoSomethingBaz(BAZ \*pbazBlat);

Never use default parameters for functions:

void DoSomethingFoo(FOO \*pfooBar, bool fFlag = false);

## Operator overloading

Do not use operator overloading.

Exceptions:

* It is often useful to overload operator= in order to disable it from functioning whatever; see item 45 of *Effective C++*. If you need to write a more complicated assignment operation, consider using an ordinary method.
* Ordinal or scalar value types, such as Matrix, Vector, or LargeInt, etc… These types are reasonable candidates for overloading standard arithmetic operators. Of course these come with performance caveats.

## Templates

Template declarations should avoid using space between the angle brackets and the generic type variable, unless absolutely necessary due to nested templates.

Templated classes or functions should be named as other classes or functions.

template <T>  
class BAR : T  
{  
 …  
}  
  
template <T>  
class BAZ : T  
{  
 …  
}

typedef BAR<CP> BARCP; // preferred spacing  
typedef BAZ< BAR<CP> > BAZBARCP; // necessary when you can’t use…  
typedef BAZ<BARCP> BAZBARCP;

## Inheritance

When deriving a class from any other classes, you must use the following convention:

class DERIVED : [public|private|protected] BASE  
{  
…etc…  
};

The key to the above example is the DERIVED : public, with a space on either side of the colon. This is crucial for quick and easy searching (incremental or otherwise) of header files for class declarations.

## Casts (C-style, function-style, modern static\_cast<>…)

Do not use function-style casts; they are hard to find with grep and they work like C-style casts.

If you must cast one non-ordinal variable to another type, use modern C++ style casts.

HRESULT WWSERVEROBJ::InnerQI(REFIID riidReq, void \*\*lplpUnk)  
{  
…  
 else if (IsEqualIID(riidReq, IID\_IOleDocumentView))  
 {  
 \*lplpUnk = static\_cast<IOleDocumentView \*>(this);  
 hr = NOERROR;  
 }  
…  
}

## memcpy, memset, ZeroBytes, etc.. on classes

For the same reason that one should never use sizeof on classes with vtables, one should never use memcpy, memset, etc. on the like. Further, we can go so far as to say that one should never use these functions on classes in general. If you are calling memset on a class, then your class should probably be a plain old data-structure (struct), not a class. Also, don’t worry about performance when writing code of this type. The Microsoft C++ compiler is perfectly capable of optimizing copy operations. It is generally a bad idea to try to think for the compiler when it comes to performance of non-algorithmic operations

## Constructors and Destructors

Minimal initialization is perfectly reasonable to take place in constructors, but anything that might fail or consume valuable system resources such as memory and time should be delayed to a function such as FInit. Similarly, destructors should not perform operations that can fail.

Never call virtual functions from within a constructor or a destructor.

## malloc vs. new

New is preferred over malloc. For classes, using new is mandatory. For ordinal types, it is optional but recommended. If you intend to resize an array or structure, then you must use alloc, realloc, free.

## Macros

In general for macro, our proposes are:

* Function: we should use template inline instead of macro
* Variable: we should use enum hack and const instead of macro

For naming macro, we should use MACRO\_ADDRESS\_SOMETHING such as your idea.

## No magic number in code

Hardcode constant other than 0 and 1 is NOT allowed in code statements. This rule are not applicable to those constants which API used as return value, such as 0xFFFFFFFF.

Define hardcoded number as macro, constant variable or in an enumeration type.

for (i = 0; i < 7; i++) // NOT allowed! hardcode magic number -- 7!

const int MAX\_UI\_WND\_NUM = 7; // or do “#define MAX\_UI\_WND\_NUM 7”

for (i = 0; i < MAX\_UI\_WND\_NUM; i++) // OK. MAX\_UI\_WND\_NUM is defined elsewhere

{

…

}

## Inline Functions

A decent rule of thumb is to not inline a function if it is more than 10 lines long. Beware of destructors, which are often longer than they appear because of implicit member- and base-destructor calls!

Another useful rule of thumb: it's typically not cost effective to inline functions with loops or switch statements (unless, in the common case, the loop or switch statement is never executed).

## Namespaces

Unnamed namespaces are allowed and even encouraged in .cpp files, to avoid runtime naming conflicts.

You may not use a *using-directive* to make all names from a namespace available.

// Forbidden -- This pollutes the namespace.

using namespace foo;

You may use a *using-declaration* anywhere in a .cc file, and in functions, methods or classes in .h files.

// OK in .cc files.

// Must be in a function, method or class in .h files.

using ::foo::bar;

## Nonmember, Static Member, and Global Functions

Prefer nonmember functions within a namespace or static member functions to global functions; use completely global functions rarely. Rather than creating classes only to group static member functions which do not share static data, use namespace instead.

## Local Variables

Place a function's variables in the narrowest scope possible, and initialize variables in the declaration.

## Static and Global Variables

Static or global variables of **class type** are forbidden: they cause hard-to-find bugs due to indeterminate order of construction and destruction.

Objects with static storage duration, including global variables, static variables, static class member variables, and function static variables, must be Plain Old Data (POD): only ints, chars, floats, or pointers, or arrays/structs of POD.

The order in which class constructors and initializers for static variables are called is only partially specified in C++ and can even change from build to build, which can cause bugs that are difficult to find.

As a result we only allow static variables to contain POD data. This rule completely disallows vector (use C arrays instead), or string (use const char []).

If you need a static or global variable of a class type, consider initializing a pointer (which will never be freed), from either your main() function or from pthread\_once(). Note that this must be a raw pointer, not a "smart" pointer, since the smart pointer's destructor will have the order-of-destructor issue that we are trying to avoid.

# Naming convention

## Standard Type Prefix

Type prefixes indicate the type of the variable. They are generally one to three characters long but contain up to five characters in extreme cases. Here is a list of standard type prefixes:

|  |  |  |
| --- | --- | --- |
| BYTE | by | 8 bit unsigned |
| SHORT,short | sh | 16 bit signed integer |
| WORD | w | 16 bits (non-integer use, such as flags) |
| DWORD | dw | 32 bits (non-integer use, such as flags) |
| INT, int | n | 32 bits integer |
| UINT | u | unsigned integer |
| LONG,long | l | 32 bit signed integer |
| float | fl | standard floating point type |
| double | dbl | standard floating point type |
| CHAR,char | ch | one-byte character |
| WCHAR | wch | 16 bit UNICODE character |
| TCHAR | tch | Flexible UNICODE / ANSI DBCS character |
| HANDLE | h | Handle |
| VOID | v | A void |
| HRESULT | hr | COM functional call result |
| BSTR | bstr | Known as basic string or binary string |
| VARIANT | var | Variant structure |
| VARIANTARG | varg | Variant argument stucture |
| IUnknown \* | punk | COM IUnknown Interface |

## String Prefix

|  |  |
| --- | --- |
| sz | ANSI null-terminated string (actual storage) |
| psz | Pointer to ANSI null-terminated string |
| str | ANSI string (actual storage) |
| pstr | Pointer to ANSI string |
|  |  |
| tsz | null-terminated TCHAR string (actual storage) |
| ptsz | Pointer to null-terminated TCHAR string |
| tstr | TCHAR string (actual storage) |
| ptstr | Pointer to TCHAR string |
|  |  |
| wsz | Wide null-terminated string (actual storage) |
| pwsz | Pointer to wide null-terminated string |
| wstr | Wide string (actual storage) |
| pwstr | Pointer to wide string |

## Generic Prefix

|  |  |  |
| --- | --- | --- |
| Boolean | f | flags and options |
| Count | c | A count |
| Index | i | Zero-based index |
| Pointer | p | Any type of pointers, except for function pointers |
| Function pointer | fn | Function pointer |
| Array | rg | an array, or a pointer to it. |
|  |  |  |
| Enum | e | enumeration element |
| Class | c | class object |
| Struct | s | struct instance |
| Constant | c | constant value |
|  |  |  |
| Structure Type | S | Prefix for a structure name. |
| Enumeration Type | E | Prefix for a enumeration name. |
| Class Type | C | Prefix for a class name. |
| Template Type | T | Prefix for a template name. |
| Pointer Type | P | Prefix for a pointer to any type name. |

For those counter variables used in for() loop could simply use i, j, k, etc.

Example:

class CDic;

CDic cDic, \*pcDic;

typedef CBitmap \*PCBitmap;

SReading sReading;

enum EFood;

EFood eFood;

int rgnFood[FOOD\_NUM];

const char\* pcstr;

for (int i = 0; i < cElement; i++)

{

…

}

## Scope Prefix

The scope prefix indicates the scope for both constants and variables.

|  |  |
| --- | --- |
| m\_ | Class data member. |
| g\_ | Public global variable. |
| s\_ | File/Function static variable. |

## OACR compliant buffer size name convention

reference: <http://officenet/smackdown/shared%20documents/oacr%20phase%20i%20security%20plugin.htm>

|  |  |  |
| --- | --- | --- |
| **Buffer size unit** | **Buffer name** | **Buffer size name** |
| size in BYTEs | pbyXXX, p*Type*XXX | cb, cbXXX |
| size in chars | pchXXX, pstrXXX | cch, cchXXX |
| size in TCHARs | ptchXXX, ptstrXXX | ctch, ctchXXX |
| size in WCHARs | pwchXXX, pwstrXXX | cwch, cwchXXX |
| size in elements | p*Type*XXX, rg*Type*XXX | c*Type*, c*Type*XXX |

# Comments

## Location

1. Always indent the comment along with the code. This may or may not include column 1.
2. For multi-line comments, always use one of these three styles of comments:

Style 1 Style 2 Style 3

<blank line> <blank line> <blank line>

// This is a test /\* This is a test /\* This is a test

// of the Emergency of the Emergency of the Emergency

// Broadcast System. Broadcast System. Broadcast System. \*/

// System. \*/

For styles #2 and #3 above, multi-line comments should start each subsequent line (i.e., all lines after the first) with "<space>\*". In addition, for multi-line comments, it is easer to use style #2 or #3 for maintainability reasons.

If a different style exists in an existing module use that style instead, except for the first rule under "Comments - Location" listed above.

1. If a parameter that represents a constant such as "fTrue" is used in a function call, comment it with a copy of the parameter name.

if (FFoo(hwnd, fTrue /\*fShow\*/))

{

. . .

}

1. Use the following format for REVIEW comments (Richard leaving a comment for Yoshi):

/\* REVIEW yoshitoy(richardb): will mathpack affect this? \*/

1. A completed feature should not contain review comments.
2. Use the following keywords for comments indicating future considerations, inefficiency designations. Use PERF as a modifier to FUTURE, REVIEW and IE if you have information that would be useful or critical in dealing with performance issues later on.

/\* FUTURE bradch (herbk): we can omit with scaleable views \*/

/\* IE tomsax (chrism): check the border cases here \*/

/\* FUTURE PERF dannyp (davidlu): can we adjust this during idle? \*/

/\* REVIEW PERF patcrens (patcrens): can get index instead of table lookup \*/

1. Space between declarations and code.
2. Two blank lines between functions is good. Use three blank lines between logical sections in a file. This is especially nice if you have banner text to separate things. Here is an example of banner text, again shortened for this document. 78 characters wide is a good size:

// ----- NUMERICS ---------------------------------------------------

## Content

1. All functions should have a function header (a description with keywords in a comment). The only exceptions should be functions declared inside a class declaration, which should be less than five lines long.  
     
   When creating a function header, list the following:

|  |  |
| --- | --- |
| *(if known)* | contact |
| *(if not obvious)* | basic purpose |
| *(if any)* | a list of known side effects |
| *(if not obvious)* | return variable |
| *(optional)* | cross reference to a checked-in project document |
| *(required)* | Parameters -- Designate the In/Out-ness on each parameter. Some parameters are both In and Out parameters. |

The format for function headers is different than for other comments and looks like the following: (other than the ----) lines end at 78 characters. The example is shortened for this document.

/\* F F O O B A R \*/  
/\*-----------------------------------------------------------------  
 %%Function: FFooBar  
 %%Contact: dougt  
 Allows cellular telephone support into EL in conjunction with  
 voice annotations. (requires CELTEL.DLL and springy antenna)  
  
 Returns: fTrue if call completes, else fFalse  
 Side effects: Calls FormatLine when complete  
------------------------------------------------------------------\*/

1. Whenever an [#ifdef:#ifndef],[#else]/#endif is used, use the same keyword on all parts.

#ifdef DEBUG #ifndef DEBUG

. . . . . .

#else // DEBUG #else // DEBUG

. . . . . .

#endif // DEBUG #endif // !DEBUG

In the second example, the #ifndef and #else refer to code following and the #endif refers to code preceding that line.