C++ Coding Guidelines

General

Deviating from Guidelines

Deviations from these guidelines should include explanations documented in the relevant source members.

Separating Source Members

Classes definitions and implementations should be contained in separate files. Global functions definitions and implementations should be contained in separate files. Also, there should only be

one

class per source file.

All source members should assume the follow the naming convention:

- <descriptor>.h
- <descriptor>.cpp
- <descriptor>.idl
- <descriptor>.inl

where <descriptor> identifies the resource (class, function, interface, etc..) contained in the members.

Standard Comment Headers

All source members must have a standard header that includes the following information:

- Name of the source member
- Purpose
- Function/Object name defined
- Author

Example:

```
////////////////////////////////////////////////////////////////
// NAME:               foobar.h
//
// PURPOSE:            Definition of CFooBar class. This class is
//                     responsible for doing foobar stuff.
//
// FUNCTIONS/OBJECTS:  CFooBar
//
// AUTHOR:             DEV1
////////////////////////////////////////////////////////////////
```

Standard Methods

All classes must implement the standard methods (even if they are empty implementations).

These methods include:

- default constructor
- virtual destructor

Example:

```
class CFoo
{
public :
                    CFoo();
       virtual      ~CFoo();
};
```

Program Constants

All “magic numbers”, delimiters, and reusable string literals should be declared as const variables or enums

Example:

```
const int MAXSIZE = 1024;

const char PLUS = ‘+’;

enum estate = {WAITING, PROCESSING, DONE, ERROR=99};
```

Class Definition

Include Guards

All class definition and interface source members must have a preprocessor include guard to prevent compilation errors due to accidental multiple includes.

The include guard should come just before the Comment Header.

The scope should include all the entire contents of the. The #endif directive should be the last line in the module.

Example:

```
#ifndef _FOOBAR_H
#define _FOOBAR_H
   . . .
   some source code
   . . .
#endif  // _FOOBAR_H
```

Header Includes

Class header files should include

only

headers that are required for the class definition. Headers should not be included as a convenience or “shortcut” to
automatically include resources for other source members.

Headers should only be included for the following reasons:

- To provide class definition of super class(es)
- To provide class definitions for other classes referred to in data members and methods.

Note: pointers/references to objects do not necessarily require that the header for that object be included. These can usually be managed with forward declarations (see below).

Forward Class Declarations

Forward declarations should be used instead of class header includes for classes referred to by pointers or reference. For example, if a method takes a CFoo* parameter, a forward declaration for the CFoo class is sufficient to compile the header.

This helps to keep the interface for an object “lightweight” so that clients of the interface are not subject to unnecessary dependencies.
Using forward declarations is also good security practice in that it removes the need to expose sensitive interface source members to client source code.
Forward declarations also reduce build dependencies you would normally incur when header files are included. Even if there are no forward declarations, it is still good practice to include the comment place-holder so they can be added when needed.

Example:

```
// includes
//-----------------------------
#include “foobarbase.h”
#include “someclass.h”

// forward declarations
//-----------------------------
class COtherClass;

class CFooBar : public CFooBarBase
{
public :
      CSomeClass     m_SomeClass;
      COtherClass *  m_pOtherClass;
};
```

Type Defines

Any type-defines and macros exclusive to the class should be located in the class source members. Defines that are required for the interface should appear in the header where as defines that are only used in the implementation belong only in the implementation source file.

Note: defines that could have a more general scope should be located in a more common global source member. Only defines that are strongly coupled with the class should be included in the class source members.

Class Naming

All class names should begin with “C” and all words should begin with a capital letter.

The class name should be descriptive of the class intention(s).

Examples:

```
CAnimal

CLostDriver

COneWayStreet
```

Method Naming

All methods and functions should begin with a capital letter. The first letter of each word should also be capitalized.

Special characters such as dashes and underscores are prohibited.

The method name chosen should be descriptive of the method functionality.

Example:

```
class CSomeClass
{
public:
     CSomeOther * FindObjectByName( CString& strName );
};
```

Data Variable Naming

All data members, arguments, and local variables should assume the following format:

[m_]<type><identifier>

where [m_] indicates class data, <type> is a data type prefix and <identifier> describes what the data represents. The identifier should always begin with a capital letter.

The conventions for data type descriptors include:

```
short int i

long int  l

float     f

double    d

bool      b

CString   str

char      c

char*     sz

pointer   p

enum      e
```

Examples:

```
unsigned short       m_iMaxRetries;
long                 m_lNumberOfRows;
float                m_fPercentComplete;
double               m_dCountResult;
bool                 m_bCacheItemsYN;
```

```
CString             m_strName;
char                m_cSwitchType
char*               m_szLogBuffer;
CFoo*               m_pFooObject;
eObjectType         m_eFirstObjectType
```

Although not required, it is sometimes preferable to solicit the <type> field for class instance variables too.

Examples:
```
CCriticalSection  m_csMutableData;
CColumn           m_colPrimaryKey;
```

Note: The int data type should be avoided due to its ambiguous size across platforms.

Use bool instead of BOOL or boolean as it is a universally accepted portable type.

Member Descriptive Comments

All methods and data members are required to be prefaced with single line comments describing the purpose of the member.

Example:
```
class CVXFoo
{
...
// registers the foo factory interface
void RegisterFactory(CFooFactoryInf* pFact);
...
// factory interface used to allocate new foos
CFooFactoryInf * m_pFooFactory;
};
```

Security

All data members and methods should be defined under appropriate security constraints.

No data members should ever be made public. Rather, they should be private or protected with appropriate access methods.

General guidelines for when to choose the proper constraint are:

Methods

`public`	only when expected to be invoked from other classes
`protected`	never invoked from other classes, but could be invoked from subclasses
`private`	only can be invoked by local class

Data

`public`	disallowed; data members should never be public
`protected`	only when expecting subclasses to access member
`private`	only when accessed exclusively by local class

Virtual Methods

All methods that overload virtual behavior should also specify the virtual keyword and include a comment indicating that it is overloaded behavior (even though it is not required to be virtual since the base class holds the vtable).

Whenever you have a virtual method in a class make the destructor of the class virtual.

Example:

```
// override superclass behavior for data processor
virtual void ProcessData();
```

-
-

Example:

```
// abstract behavior for data processor
virtual void ProcessData() = 0;
```

Class Member Organization

All class definitions should adhere to a standard organization. Class methods should be defined first followed by data member declarations. Generally, methods and data members should be organized by security in the order of public, protected, and then private. However, it is acceptable to organize groups of related operations under a single security constraint.

Example:

```
class CFoo : public CFooBase
{
// methods
//-----------------------------
public :
protected :
private :

// attributes
//-----------------------------
public :
protected :
private :
};
```

Inline Methods

Inline methods should only be used under unique circumstances. These are typically used for operations that are called frequently and would otherwise present performance issues.

Inline methods in general should be avoided because they will introduce build dependencies when the behavior of the inline method is changed.

Using inline methods as a convenient shortcut to implementation but is not an acceptable coding practice.

If inline methods are used, they should be implemented in a separate source member file with the naming convention: <descriptor>.inl.

Method Signatures

Parameters

All method parameters should include an argument name following the type. This helps to identify the purpose of the parameters.

It is helpful to identify the directional usage of parameters with “in” and “out” name qualifiers.

If possible, “in” parameters should be passed as a *const* reference to prevent accidental data changes. Use of const can be tricky and should be used carefully.

Parameter naming must follow the conventions outlined above for all variable data.

All class instances should be passed as references (&) where possible. One exception to this is “Set” and “Register” methods that may require pointers (*).

Examples:

```
BOOL    CopyTables( const CPList& inTables, CPList& outTables);

BOOL    CopyMoreTables( /*in*/  const CPList& curTables,
                        /*out*/ CPList& newTables);
```

Return Types

If a method is returning access to data that is managed by the class, then the data access should *ideally* be returned as *const* to prevent accidental data changes. Sometimes, this may not be possible due to the strict cascading rules with *const*.

Example:

```
const  CFoo*  GetFirstObject();
```

Methods that could encounter errors should always specify return codes (if they do not throw exceptions).

Accessors

Access to any data member must be done via an accessor method.

In general, and if possible, all access to data member instances should be done through *const* references.

Accessors that involve complex behavior and could potentially fail should specify a boolean (or some coded type) return type and an “out” parameter type.

Examples:

```
long          GetCount();
const  CFoo*  GetFirstObject();
BOOL          GetAllObjects( /*out*/ CPList& FirstObj );
```

Class Implementation

Method Headers

All class methods/functions should have a comment header that includes:

- Method/function name

- Description
- Author
- Date created

Also, all methods/functions should include a trailing comment just after the closing brace of the function scope that identifies the method/function name.

Example:

```
//=====================================================================
// NAME        : GetName
//
// DESCRIPTION : Accessor for the name of this object.
// AUTHOR      : DEV1
// DATE        : 10/10/99
//=====================================================================
void CFoo::GetName( CString& strName )
{
    strName = m_strName;

} // GetName
```

Super Classes

All overloaded methods should take into account that the method on the super class may need to be invoked. Typically, the super class method is invoked before the subclass continues with execution of its implementation of the method.

If a super class method is to be invoked by an overloaded method, the rule is that the subclass will invoke the method on its immediate base class (even if the immediate base class has not implemented the behavior).

Example :

```
void CFoo::Initialize()
{
    // always forward to base class
    CFooBase::Initialize();

    // now init our local data
    m_strName        =   T("");
    m_lFooObjectID   =   NEW_ID;
}
```

Functional Comments

All functional code must be clearly commented to indicate exactly what the code is doing. Any numerical calculations should have descriptive comments stating the purpose of the calculations

In general, there should be sufficient documentation to understand the method logic without having to read the executable code.

Functional Coding Practices

Variables and Parameters

All local and global variables should always be initialized when declared.

All variables and parameters should have a meaningful name and must begin with a type prefix (*see Data Member Naming)*.

Examples:

```
BOOL     bForceYN    =  FALSE;
CString  strName     =  T("");
ObjectId lObjectID   =  NEW_ID;
```

Arithmetic

-
-
-

Memory Management

Data Members

If at all possible, try to avoid heap memory allocation for data members. This only complicates the management of the class and introduces potential access violations and memory leaks.

Instead, consider using an embedded object rather than a pointer.

Example:

```
CFoo * m_pFooObject;  //memory management required
CFoo   m_FooObject;  //no memory issues
```

Stack Allocation

Whenever temporary memory is required for the scope of some function, use stack allocated memory rather that heap allocated. This will ensure reliable cleanup upon exiting the scope of the function (even when exceptions are thrown). Besides, it's less code to manage.

Example:

```
//heap allocated aproach is risky
CFoo * pFoo = new CFoo();
pObj->SomeMethod( pFoo );
delete pFoo;

//stack allocation is safer
CFoo   Foo;
pObj->SomeMethod( &Foo );
```

Ambiguous Ownership

If an object owns a pointer to memory that may have been allocated by that object or some other object, there is a certain ambiguity introduced for cleaning up the memory.

The safest way around this is to have the object own two pointer data members. One would be used for the object to manage the memory that *it* had allocated. The other would be the pointer used to *refer* to the data.

This way, the object knows when it has allocated memory and can be sure it

is not deleting a bad pointer or some other object's memory.

Example:

```
m_pMyBuffer = new Data();
m_pBuffer   = m_pMyBuffer;
```

Unit Testers

All classes should either support or be exercised by some unit test. The unit test should be defined as a static behavior of the class and return a *long int* as a result code.

The convention for the unit test method signature is:

```
static long UnitTest();
```

The use of arguments is optional for parameterized testing.

```
static long UnitTest( CFoo * pHost );
```

This page was last updated October 18, 2002 by Charles Dale.