ObjectWindows Library Coding Style

The ObjectWindows Team

1Purpose

This document provides a set of guidelines for anyone writing ObjectWindows Library (OWL) source code or examples. Since OWL's source code and examples are public, every effort must be made to maintain a consistent look.

This document is more specific to the current team's style. It is not meant to replace the *Borland Framework Team Guidelines*, but to clarify certain gray areas within that document for OWL.

2Naming Conventions

2.1Identifiers

In general, identifiers are mixed case. The only exception to this are macros, which are in UPPERCASE. Any identifier global variables, class data members, function names, and member function names begin with a capital letter. Parameters to functions and local variables begin with a lower case letter and are mixed case. OWL's name for its classes, enums, and structs begins with an initial capital T followed by its description with word capitalization. Example:

```
class _OWLCLASS TWindow : public TEventHandler {
  public:
    DataMember1;
    static DataMember2;
    void MemberFunc1(int arg1, char arg2);
};
```

Try not to use _ to separate words unless it is in a macro, which is all UPPERCASE.

Identifiers should be as descriptive as possible without being too long. If you cannot find one, perhaps the class or function is doing too much.

Boolean data variables should be named when the condition in true. For example, a boolean variable that maintains the opened state of a file would be named IsOpened instead of IsClosed. Conditionals based on the variable would be easier to read. In this case, if (IsOpened) rather than if (!IsClosed) to test if the file is open. Also, consider using enums like if (FileState == open).

2.1.1Resource Identifiers

For cross-platform compatibility, resources should be named with a number identifier rather than a named identifier. Windows supports both methods, but OS/2 only supports number identifiers.

The following table lists the prefix for the various resource types:

Accelerator	IDA_
Bitmap or Dib	IDB_
Cursor	IDC_
Dialog	IDD_
Font	IDF_
Icon	IDI_
Menu	IDM_
String	IDS_
VersionInfo	IDV_
Help	IDH_
User-defined	IDU_
Commands	CM_

2.1.2Miscellaneous Identifiers

Child windows	IDW_
Gadgets	IDG_

2.1.3Command Member Functions

Command member functions are named CmXXXX where XXXX is the command. For example:

```
void
TMyWindow::CmFileOpen()
{
}
```

The command enablers are named CeXXXX. For example:

```
void
TMyWindow::CeFileOpen(TCommandEnabler& enabler)
{
}
```

2.1.4Member Functions That Handle Child Notifications

Member functions that handle child notifications use this prefix convention:

Button	Bn
Combobox	Cbn
Edit	En
Listbox	Lbn

Listview	Lvn
Header	Hdn
Treeview	Tvn
Tooltip	Ttn
Common dialog	Cdn
Toolbar	Tbn
Animation control	Acn
Updown	Udn
Tab control	Tcn

2.2Types

OWL provides a wrapper for various data types to make it more portable and easier to read. In some cases, rather than using macros like MAKELONG, OWL provides safer inline functions. Use OWL's data types instead of Windows's.

For data that needs to be specific size, use int8, int16, and int32 instead of using short, int, and long which may not have the same size depending on target platform. Correspondingly use uint8, uint16, and uint32 for unsigned versions. Use int or long, when size is a suggestion.

OWL also provides MkUint16(), MkUint32(), LoUint16(), HiUint16(), LoUint8(), and HiUint8() inline functions to extract and put together various integer data types. Use these instead of the macros MAKELONG, MAKELPARAM, HIWORD, and LOWORD.

OWL provides a bool data type. Depending on the compiler, it may be the instrinsic C++ type or it will be emulated. In most cases, use bool instead of BOOL. Use true and false instead of TRUE and FALSE. Be sure to keep your usage consistent. Windows declare BOOL to be synonymous with int. The size of int changes when you switch from 16-bits to 32-bits (i.e. from 2 bytes to 4 bytes). If the compiler version of bool is used, the compiler will decide the most efficient size for bool.

OWL provides wrappers for the four types in a callback: LRESULT, UINT (message), WPARAM, and LPARAM. Use TResult, TMsgId, TParam1, and TParam2 instead.

3Comments

Comments provide insight to the code. Write them to explain *why* that piece of code exists and not what it is doing. The source code explains what. C++ style comments are used instead of C's /* */ sequence. The first words in comments are always capitalized. Class comments and block comments should be complete sentences, while statement-block and single-line comments do not need to be.

3.1Class Comments

Class comments are written in the header file of the class and look like this:

```
//
// class TLayoutWindow
// ~~~~~ ~~~~~~~~~~~~~
// TLayoutWindow provides a parent window that manages the
// location and sizes of its children.
//
class TLayoutWindow : virtual public TWindow {
  // ... Other stuff ...
  //
};
```

The comments describe what the class encapsulates and precedes the class declaration in the header file.

3.2Block comments

Block comments precede the definition of functions. They can also be used to describe global variables. They are used whenever more than one line of comments is required. These style of comments are used outside the scope of any functions or class. The always line up in the first column of each line.

OWL does not use C's block style comments /* */. Instead of

```
/*
  This is a block comment.
  Notice it has multiple lines.
*/
```

Use this style:

```
//
// This is a block comment.
// Notice it has multiple lines.
//
```

3.3Statement-Blocks Comments

Statement-blocks comments break up logical groups of statements and explain the purpose of each group. These comments are used inside of functions or class declarations and always indented by some number of spaces. They are preceded by a single line of whitespace if the statement above them is on the same level of indentation. They have this format:

```
void
SomeFunction()
{
  // Iterate through all choices
  //
  for (int i = 0; i < 0; i++) {
    // Do stuff
    //
  }
}
```

3.4Single-line Comments

Single line comments are usually very short and belong at the end of the statement. For example,

```
bool done;   // Gone through all the files?
```

4Layout

This section discusses how each of the following items should look, i.e. its spacing, its identation, and if applicable, its order of nested items.

4.1Copyright Notice

The following copyright notices must appear at the top of each file (the number of dashes, -, have been truncated to fit the page, there should be 74 of them):

```
//---------------------------------------------------------- (74 dashes)
// ObjectWindows
// Copyright (c) 1991, 1995 by Borland International, All Rights Reserved
//
// Description of what is contained within the file
//---------------------------------------------------------- (74 dashes)
```

The first year is the first published date of the file and the second year is the most recent date published, which should be updated every time there is a new release of the product.

A description of what is contained within the file follows the copyright notice. It uses the block comments style mentioned above.

4.2Files

4.2.1Header Files (.h)

Header files have a copyright notice at the top of the file. It is followed by a *sentry guard*. The sentry guard prevents the file from being included twice. The name of the sentry is typically named LIBRARY_FILENAME_H, so the header file looks something like

```
//
// Copyright notice
// ...
#if !defined(PRODUCT_FILENAME_H)
#define PRODUCT_FILENAME_H

//
// Declare stuff here
//

#endif  // PRODUCT_FILENAME_H
```

Within the guards are global variable declarations and global function prototypes. They are then followed by class declarations and inline implemenations for each class at the end. Inline functions should never be defined class inline. They should be defined as inline member functions. Do

```
class TClass1 {
  public:
    void Foo();
};

class TClass2 {
  public:
    void Foo();
};

inline void
TClass1::Foo()
{
  // Do stuff
  //
}

inline void
TClass2::Foo()
{
  // Do stuff
  //
}
```

instead of

```
class TClass1 {
  public:
    void Foo()
    {
       // Do stuff
       //
    }
};

class TClass2 {
  public:
    void Foo()
    {
       // Do stuff
       //
    }
};
```

4.2.2Resource header files (.rh)

Resource header files must have a copyright notice at the top of the file. It should contain only #defines for the resource compiler. It does not need sentry guards since the #defines are always the same.

4.2.3Resource Files (.rc)

Resource files must have a copyright notice at the top of the file. It should have #includes for any resource header it needs before defining any of the resource. Resources files should be self contained, for example, it should not need to refer to mybitmap.bmp. The resource file may contain other resource files.

4.2.4Source Files

Source files must have a copyright notice at the top of the file. It should include any files it needs to compile. Be sure to check if the .cpp file compiles without precompiled headers. Global variable definitions as well as class-static data members are defined near the top of the file. They are followed by function definitions. Each function definition must be preceded by block comments.

4.3Whitespace, Indentation, and Braces

Whitespace is very important for readability. Cramming too much code onto a page is usually not a good idea. In general, looking at OWL's source code or examples, it should feel open with comments sprinkled throughout. Indentation plays another key role to quickly pick groups of statements, without actually reading the code. Braces are language punctuation that helps indentation make things clearer.

4.3.1Tabs and Indentation

OWL does not use tab characters, it always use spaces. There are no control-z characters in the file to denote an end of file. Control-z is an anachronism from CP/M that has been obsolete for 13 years.

Each level of indentation is 2 spaces.

4.3.2Line Breakage

A statement in C++ ends with a semicolon (;). Each statement should be on its own line. The line is usually never longer than 80 characters. If the line is too long and needs to be separated, try to separate it as you would word-wrap a paragraph. But unlike a paragraph, the continuation should be indented one level or line up vertically to the logical group in the previous line. For example,

```
void
Foo(TArgType1 arg1, TArgType2 arg2,
    TArgType3 arg3)
{
}
```

4.3.3Keywords and Braces

The following sections discuss many of the keywords in C++ and how they are formatted in OWL.

4.3.3.1Function Definition

```
ReturnType TypeModifier
FuncName(ArgType1 argName1, ArgType2 argName2, ...)
{
}
```

For constructors with initialization list, use this format:

```
ReturnType FunctionModifier
ClassName::FuncName(TArgType1 argName1, TArgType2 argName2, ...)
:
  Initialization(List)
{
}
```

If the line is too long, break the line at the comma and indent the next line.

4.3.3.2Operators

All binary and trinary operators must have a space on each side of the operator.

```
r = sqrt(x * x + y * y);
smaller = (a < b) ? a : b;
```

If a group of assignment statements follow each other and they should be logically grouped, then the assignment operator should line up vertically:

```
rect.left   = 0;
rect.top    = 0;
rect.right  = 100;
rect.bottom = 100;
```

In addition the group should be preceded by a block comment.

4.3.3.3Control Flow

The following section describes the various control-flow statements in C++ and how they are separated in OWL. Most control flow statements are preceded and followed by a blank line. A blank line separates the various cases in a switch statement. If the control flow statement is fairly long or contains multiple closing braces ({}), it is advantageous to include a single line comment that says what the braces closes when the opening brace is more than screenful or pageful away. If it is too long, consider breaking it up into logical parts.

do/while

```
do {
  statement1;
  statement2;
} while (condition);
```

while

```
while (condition) {
  statement1;
  statement2;
}
```

if/else/else if

```
if (condition) {
  statement1;
  statement2;
}
else if (condition2) {
  statement3;
  statement4;
}
else {
  statement5;
  statement6;
}
```

for

```
int j;
for (j = 0; j < 3; j++) {
  step1;
  step2;
}
```

or

```
for (int j = 0; j < 3; j++) {
  step1;
  step2;
}
```

The biggest difference between the two example for loops is the scope of the j. In the first example, j lives outside the scope of the for loop, whereas in the second example, j lives within the scope of the for. If the statement, j = 0, follows the closing brace of the for loops, it will always work in the first example, but not in the second.

switch/case/break/delete

```
switch (expression) {
  case choice1: {
    choice1statement1;
    choice2statement2;
    break;
  } // Choice1

  // more cases
  //

  default: {
    defaultstatement1;
    defaultstatement2;
    break;
  } // default
}; // switch
```

The braces around the case statements are not always needed. But if one of the statement inside the case is a definition of a variable, then you will need to start a new block with the braces.

4.3.3.4Exception Handling

```
try {
  statement1;
  statement2;
}
catch (condition1) {
  condition1statement1;
  condition1statement2;
} // condition1
catch (condition2) {
  condition2statement1;
  condition2statement2;
} // condition2
```

4.3.3.5Enums and Unions

```
enum Type {
  Constant1 = 0,
  Constant2,
};
```

4.3.4Preprocessor macros

The indentation for preprocessor macros is a little different from code because of # being the first character in the line requirement. So instead of putting spaces at the front of the line to indent, put the spaces after the #, but make sure the indentation levels still match the code:

```
#if defined(BI_PLAT_WIN16)
# pragma xyz
# if 1
#   pragma morestuff
# endif
#endif

void
foo()
{
  if (condition)
    printf(...);
}
```

4.3.4.1Length

If the length of the preprocessor macro exceeds 80 characters, use the continuation character \ to break the line. Use the same concept of breaking macros as code.

```
#define WONDERFULMACRO(x) some gobbledee gook that will space \
  that continues to the next line.
```

4.3.4.2Conditionals

Use the #if !defined() operator to check for negative conditions and not #ifndef. For example,

```
#if !defined(BI_PLAT_WIN32)
# error Must be compiled for 32-bits
#endif
```

4.4Class and Struct Declaration

```
//
// class ClassName
// ~~~~~ ~~~~~~~~~
class ClassModifier ClassName : public BaseClass {
  public:
    // Constructors and destructors
    //

  public_data:
    // Public data that may be hidden depending on OWL_STRICT_DATA
    //

  protected:

  protected_data:

  private:

  DECLARE_RESPONSE_TABLE(ClassName);

  friend class FriendClass;
  friend function();
};
```

Any friends of a class are declared at the bottom. It is indented one level from the class keyword as are public, public_data, protected, protected_data, private, and any class specific macros such as DECLARE_RESPONSE_TABLE or DECLARE_AUTOCLASS (OCF). Each section is separated by a blank line. Each item within a section is indented one level from the section heading they are in.

4.5Pointers and References

Pointers and references belong with the types and not with the variable. Do this

```
int* j;
double& xAlias = x;
```

and not

```
int *j;
double &xAlias = x;
```

4.6Multiple Declarations

If you have multiple declarations involving pointers and references, it is better to have each as their own separate statement, rather than lumping them together with a comma (,). Do this:

```
int* a;
int* b;
int* c;
```

and not

```
int* a,
   * b,
   * c;
```

and definitely do not do this:

```
int* a, * b, * c;
```

5General Coding

5.1Makefiles

All examples in OWL must use makefile.gen. At the top of the makefile, it should clearly state which operating systems the example works with:

```
SYSTEMS = WIN16 WIN32 CON32
```

as well as which memory models the example supports:

```
MODELS = S M L C H D X F
```

The example must also have a corresponding .ide file. Additionally, there is a readme.txt in that same directory that describes the functionality of the example. The format of the readme.txt is

```
Copyright Borland International
[any text]
Title: [Title of readme]
[any text]
Keywords: [List of keywords separated by ;]
  [if last character in preceding line is ;, continue list of keywords]
[Text of readme.txt]
[EOF]
```

5.2Supporting Various Platforms

OWL compiles equally well for both 16-bits and 32-bits. All examples should do the same unless the example demonstrates features specifically to that operating platform. In which case, the source files should test for those conditions using this method:

```
#include <owl/pch.h>
#include <owl/defs.h>    // guarantee loading of the needed macros
#if !defined(condition)
# error This example does not support the target platform.
#endif
```

Here is a list of platforms OWL knows about and the define you can check.

Windows (all)	BI_PLAT_MSW
16-bit Windows	BI_PLAT_WIN16
32-bit Windows	BI_PLAT_WIN32
IBM OS/2	BI_PLAT_OS2
DOS	BI_PLAT_DOS

Within each platform, you can check for the various memory models (some do not apply):

Tiny	BI_MODEL_TINY
Small	BI_MODEL_SMALL
Compact	BI_MODEL_COMPACT
Medium	BI_MODEL_MEDIUM
Large	BI_MODEL_LARGE
Huge	BI_MODEL_HUGE

Alternatively, you can check for various attributes of the memory model:

Pointer is segmented	BI_PTR_16_16
Pointer is flat	BI_PTR_0_32
Pointers are near	BI_DATA_NEAR
Pointers are far	BI_DATA_FAR
Code is near	BI_CODE_NEAR
Code is far	BI_CODE_FAR

You can also check if the module being built is for a DLL target or an EXE target:

EXE	BI_APP_EXECUTABLE
DLL	BI_APP_DLL

To know whether the operating platform supports multithreaded programming, check for the BI_MULTI_THREAD define.

5.3Headers

Always use the forward slash to include files in subdirectories:

```
#include <owl/window.h>
```

5.3.1Precompiled Headers

Precompiled headers save a lot of time when used properly. But double-check to make sure each .cpp file can be compiled without precompiled headers. To use precompiled headers, add this to top of each .cpp file:

```
#include <owl/pch.h>
```

5.3.2System Headers

OWL typically will include headers in the right order. For example, it figures out whether you want OLE or not. Because it gets the header files, you should not need to explicitly include any of the Windows header files. But if you really want to, make sure you are including OWL's headers first. Otherwise, there may be some conflicts.

The order you should include files is OWL, OCF, BIDS, Windows, and finally the RTL.

5.4Warnings

The example should compile without any warnings when all warnings are turned on from the compiler.

5.5Diagnostics

Diagnostic macros should be used liberally. They are only enabled when a diagnostic build is requested and can be controlled by the user by editing the OWL.INI file. TRACE, CHECK, and WARN also have option forms which allow a group name and a level to be specified. This provides very fine level control over which messages are generated.

5.5.1PRECONDITION (BoolExp), PRECONDITIONX (BoolExp, Message)

This is used to prove that the function has been called correctly. Use it to validate any argument values which should not be encountered in normal usage. This also documents the assumptions that are made. So if you assume that an integer arg will never be 0, check it with a precondition. An exception will be thrown if BoolExp is false.

5.5.2CHECK (BoolExp), CHECKX (BoolExp, Message)

This is used to prove that the function works correctly (as opposed to being called correctly). Use it like precondition, but not for arg validation. An exception will be thrown if BoolExp is false.

A good place to use CHECK in in places where you find a comment like "// Should never get here"

5.5.3WARN (BoolExp, Message), WARNX(Group, BoolExp, Level, Message)

This is used for things that are possible problems. If you know it's a problem, then use CHECK because it will throw an exception where CHECK will only display the message.

5.5.4TRACE (Message), TRACEX(Group, Level, Message)

Trace macros are normally used as a debugging aid and may be placed anywhere. They should normally be removed before ship unless the add to the user's understanding of the code. For example, a TRACE statement may be left in constructors if users have had difficulty understanding the order of construction.

5.6Catch Memory Leaks

All examples should be run through some form of memory checker such as CodeGuard, Bounds Checker or the Debugging version of Windows. These tools help track memory and resource leaks and help debug OWL in general.

5.7C++ Usage

5.7.1Extern "C"

If the header can be use from C, be sure to put the proper sentries around the prototypes:

```
#if defined (__cplusplus)
extern "C" {
#endif

//
// Some function prototypes here
//

#if defined (__cplusplus)
}
#endif
```

5.7.2Scoping

Use the scope resolution operator (::) to refer to global objects or functions.

5.7.3Inline functions

Whenever possible, use inline functions instead of macros.

5.7.4Constants

Avoid using hardcoded constants. Use the const keyword to create a descriptive alias for the constant. Instead of using NULL, use 0 instead.

5.7.5Goto

Gotos should be avoided.

5.7.6Casts

Do not use explicit casts anywhere. They are error prone. Instead, use the C++ new-style casts :

- dynamic_cast<>()
- static_cast<>()
- const_cast<>()
- reinterpret_cast<>()

or the Classlib macros:

- TYPESAFE_DOWNCAST()
- STATIC_CAST()
- CONST_CAST()
  - REINTERPRET_CAST()

It is better to use the macros because on other compilers where the C++ new-style casts are not defined, the macros emulate the right behaviour.

5.7.7New and Delete

All memory allocation should use new and delete instead of malloc and free.

5.7.8Class declaration

A class should have at the following member functions: default constructor, copy constructor, destructor (possibly virtual), assignment operator, and the comparison operator. For example, if you have a class called AClass, these are the declarations of those member functions:

```
class AClass {
  public:
             AClass();
             AClass(const AClass&);
    virtual ~AClass();
    AClass& operator = (const AClass&);
    int operator == (const AClass&) const;

  protected:

  private:
};
```

Some of the member functions may not apply in certain situations, but you should declare them and not define them to prevent the compiler from generating a default ones. If you do accidentally use them, you will get an undefined symbol from the linker. You may also change the visibility of those member functions to prevent others from using them accidentally. For example, moving the default constructor to the private section to prevent anybody from using it in either of these manners:

```
AClass instance;
```

or

```
AClass* instancePtr = new AClass;
```

Default constructor

```
AClass::AClass()
{
}
```

Copy constructor

```
AClass::AClass(const AClass& otherInstance)
{
  *this = otherInstance;
}
```

Destructor

```
AClass::~AClass()
{
}
```

Assignment operator

```
AClass&
AClass::operator = (const AClass& otherInstance)
{
  // copy as necessary
  //
  return *this;
}
```

Comparison operator

```
int
AClass::operator == (const AClass& otherInstance)
{
  if (compareIsEqual)
    return true;
  return false;
}
```

5.8Locality of Reference for Local Variables

In C, variables can only be defined at the beginning of the block. In C++, a variable definition is considered as another statement, and therefore can be anywhere within the block. So keep the local variable definitions close to where they are used. Do the following

5.9No Redundant Parentheses

In conditional expressions, remove any redundant parentheses. Use the C++ precedence rules guide the behavior, rather than using redundant parentheses. Use

```
if (c == '\n' || c == '\t')
```

rather than,

```
if ((c == '\n') || (c == '\t'))
```

5.10Function Preferences

For portability, use the standard C library functions over the operating system's API. For example, use strcpy() rather than lstrcpy() unless there is a need to use the operating system's. When using the operating system API, it should explain why it was chosen.

5.11Optimizing Memory Usage

5.11.1Global Variables

The number of global variables should be kept to a minimum. If a global variable is needed, try to make it a global pointer to an object, rather than a global instance. Then at run-time, dynamically allocate the object and free it.

5.11.2Stack

Local variables consume stack space. Keep local variables to a minimum as well. If you need to have a local array or a local object that is pretty large, then use the same pointer technique for global variables.

Another technique to use for local variables is using smart pointers. With smart pointers, you get all the benefits of pointers, but you can forget about explicitly deleting the object. For example, if you have this type of code:

```
void
func()
{
  TSomeObject object;

  // Do stuff with object
  //
}
```

Change it to using pointers and references:

```
void
func()
{
  TSomeObject* objectPtr = new TSomeObject;
  TSomeObject& object = *objectPtr;   // reference for compatability

  // Do stuff with object
  //

  delete objectPtr;
}
```

And use this form for smart pointers:

```
#include <classlib/pointer.h>

void
func()
{
  TPointer<TSomeObject> objectPtr = new TSomeObject;
  TSomeObject& object = *objectPtr;

  // Do stuff with object
  //

  // No need to call delete
  //
}
```

A big benefit from using smart pointers is whenever the smart pointer goes out of scope, regardless if the function returned or an exception is throw, it automatically deletes the object. This is not true for the case of using only pointers because an explicit delete is required.

If you have an array, use TAPointer to get a smart pointer for the array.

6General OWL Usage

6.1Frame Windows

In general, do not derive from frame windows such as TFrameWindow, TMDIFrame, or TDecoratedFrame. All user-specific windows should be derived from TWindow and inserted as a client to one of the frames. If there are messages needed to be caught at the frame level, then it could be a problem with the frame not fowarding the message to the client (i.e. a bug in OWL).

If there are messages that specifically need to be caught at the application level, do not put them in the response table for the main window, put them in the derived TApplication class.

6.2OwlMain

Use OwlMain. Do not create your own WinMain for your applications.

Use OwlMain for the entry point of your DLLs as well. It will be called with the right arguments on startup of the DLL.

6.3InitMainWindow

In TApplication::InitMainWindow, do not use

```
MainWindow = new TFrameWindow(...);
```

Instead use

```
SetMainWindow(new TFrameWindow(...));
```

6.4Resources

Resources can be named with a number or a string identifier. Use the number identifier, rather than the string.