A Technical Comparison of Borland ObjectWindows 2.0 and Microsoft MFC 2.5

Table of Contents

Executive Summary

ObjectWindows 2.0 offers a rich set of classes that make writing Windows applications much easier. The framework features a true object-oriented approach that is consistent, flexible, and extensible. ObjectWindows is completely ANSI compatible, and fully exploits the powerful features of the C++ language such as multiple inheritance, templates, and exception handling. These features dramatically increase reusability and help produce robust applications. On the other hand, the Microsoft Foundation Classes (MFC) provide only a thin layer of abstraction over the Windows API. The MFC framework is not built on a good object oriented design, and often makes use of C style constructs. As a result, the MFC framework is often difficult to use, error prone, and doesn't promote code reuse.

	OWL2.0	MFC 2.5
Object-oriented architecture	ObjectWindows 2.0 uses a high-level object oriented approach that offers more reusable objects and a more consistent, more robust framework.	MFC does not have a good object oriented design, and often requires the use of C style constructs.
ANSI compliance	ObjectWindows is completely ANSI compatible and fully exploits standard C++ facilities including templates and exceptions to increase reusability and robustness.	MFC has no support for ANSI standard templates or exceptions.
C++ exceptions	Exception support in ObjectWindows is ANSI compliant and applied thoroughly and consistently throughout to give a simple, robust exception mechanism to users.	MFC's exception support is clumsy, complex, non-standard and error-prone.
ANSI string class	Borland supports the ANSI standard string class.	Microsoft does not provide an ANSI string class, resulting in code which is non-portable and does not use ANSI exception handling.
C++ templates	Borland supports ANSI standard templates to allow easier code reuse without giving up type safety.	MFC doesn't use templates, resulting in code which is more error-prone and harder to re-use.
Special window types	ObjectWindows includes a number of special window types that facilitate the design of Windows applications. By having a richer set of classes, and more built-in functionality, ObjectWindows reduces the amount of code necessary to create modern user-interfaces.	MFC has no comparable support for layout windows, and its support for toolbars, status lines and palettes is significantly more difficult to use.
Layout windows	Constraint driven windows are important for configurability and flexibility because their size and shape are completely driven by a set of constraints that allow them to adapt as the controlling parameters change. ObjectWindows provides this through powerful Layout Windows.	MFC has no comparable capability.
Toolbars	ObjectWindows provides truly object-oriented toolbar classes, allowing more configurable toolbar-based application.	MFC does not use an object-based approach — it simply uses bitmaps. This makes it very hard to provide programmatic control over toolbars.
Status bars	ObjectWindows provides an object-oriented status bar class that results in a simpler yet much more customizable status bar in applications.	MFC uses a limited and rigid C-based approach to status bars.

Tool palettes	ObjectWindows uses objects of small size built up into a class hierarchy supporting tool palettes that allows features to be changed with minimal code changes.	MFC exposes a very complex and error-prone approach to tool palettes.
Dialog box controls	ObjectWindows treats dialog boxes and child controls just as any other object.	In MFC, use of these objects requires additional overhead through the use of "helper" functions.
VBX controls	ObjectWindows takes a very consistent approach to controls — VBX or otherwise.	MFC requires special VBX handling. In addition, MFC provides no drag-and-drop support for VBX controls.
3D Controls	ObjectWindows fully supports Borland's own 3D controls as well as those provided by Microsoft.	MFC does not support either Borland's 3D controls, or those provided by Microsoft.
Data transfer	ObjectWindows provides very simple straightforward mechanisms to transfer data from dialogs to the underlying object.	MFC's DDX data exchange mechanism is much more complex and therefore harder to use correctly.
Data validation	ObjectWindows uses a very object-oriented approach to data validation where a validator object is attached to a control; no extra code is needed because the validator handles it.	In MFC data validation is handled through a series of global functions and validation only happens during data exchange. This also makes MFC data validation dangerous and crash-prone.
GDI support	ObjectWindows provides a rich set of classes that support Windows graphics calls. The object oriented nature of ObjectWindows provides a great deal of flexibility and scalability.	MFC provides a very thin layer over GDI. This makes it difficult to use GDI in a true object oriented manner.
Printer support	ObjectWindows makes it easy to add printer support to an application regardless of the type of information being displayed. Printer support is very easy to use and is quite flexible	Printer support in MFC is difficult to use and is very restrictive.
Menus	ObjectWindows has very powerful capabilities when dealing with menus. The key concept is menu merging in which ObjectWindows takes care of the details for you	MFC does not contain support for sopisticated menu handling such as menu merging.
Bitmaps	ObjectWindows provides classes for device independent bitmaps, supports clipboard operations on bitmaps and supports reading and writing bitmaps to files.	MFC supports no advanced bitmap features such as reading and writing bitmaps to files
Metafiles	Windows metafiles — important efficient graphics storage objects — are encapsulated in ObjectWindows	MFC provides no support for metafiles.
Fonts	ObjectWindows demonstrates its clean, object-oriented architecture in its support for Windows fonts. Simple constructors with default arguments do all the work.	In MFC creating font objects is overly complex.

Containers	ObjectWindows really shows its object-oriented strength on containers. Important concepts like ownership, cleanup on deletion and iteration are key to ObjectWindows implementation that uses templates extensively. The BIDS classes provided by ObjectWindows include all the fundamental ones used in the object-oriented community. There are 11 basic types included.	MFC's containers are C-style and do not use templates and are thereby quite inflexible. Only 3 basic types are provided. Non-standard terminology inhibits understanding and communication. There is no type-safety and ownership is not enforced making memory leaks common occurrences.
OLE 2.0 encapsulation	Currently under development.	MFC 2.5 encapsulates OLE 2.0.
Database encapsulation	Currently under development.	MFC 2.5 encapsulates ODBC.

Introduction

Quick Summary: This guide provides a detailed technical comparison of Borland's ObjectWindows 2.0 application framework and Microsoft's MFC library.

When choosing a C++ development environment, the selection of an application framework or class library can play an important role in determining the overall productivity in developing new applications. After all, it's the reusable classes in the application framework that provide much of the leverage of code reusability that C++ offers.

Both Borland C++ 4.0 and Microsoft Visual C++ 1.5 include an application framework. Borland C++ 4.0 includes ObjectWindows 2.0, an application framework that focuses on providing high-level objects to reduce the overall code required to build sophisticated, robust applications. Microsoft provides the Microsoft Foundation Classes, known as MFC.

This document provides a detailed technical comparison of the ObjectWindows 2.0 application framework and Microsoft's MFC 2.5 library. This comparison will show how ObjectWindows more fully exploits the powers of C++ to provide greater code reusability, more high-level objects, easier development and a more robust set of classes.

The following areas are discussed in detail:

- ANSI compliance
- Message handling
- Document/View model
- Dialog box controls
- GDI classes
- Printer support
- Resources
- Containers
- Streamable objects
- Diagnostics and debugging
- OLE 2.0 encapsulation

Overview

Quick Summary: ObjectWindows 2.0 uses a high-level object-oriented approach that offers more reusable objects and a more consistent, more robust framework.

The first version of ObjectWindows application framework was introduced in 1991 and a great number of developers embraced the product as a better way to program C++ Windows applications. The ObjectWindows approach was unique since it focused on providing high-level objects that dramatically simplified Windows programming. ObjectWindows harnessed the power of C++ to eliminate many of the tedious details of Windows programming. As a result, Windows programming was opened up to thousands of developers who wanted a more productive way to build Windows applications. At present, there are over 300,000 ObjectWindows users, making it the most popular application framework for Windows development.

ObjectWindows 2.0 is Borland's next generation application framework, and is the result of a substantial expansion of the vision that began with ObjectWindows 1.0. The four major design goals that are reflected in the ObjectWindows architecture are:
- Make it easy to develop professional applications
- Make it easy to migrate between 16 and 32 bit Windows
- Take advantage of C++ power to increase programmer productivity
- Provide a strong foundation for component architectures

ObjectWindows 2.0 expands the coverage of the Windows API to provide high-level object-oriented encapsulations for GDI graphics and printing as well as complete support for document/view architecture. The "high-level" approach that ObjectWindows uses means that developers have a richer set of objects to draw from including support for sophisticated user interface elements such as speedbars, status lines, palettes and print preview.

ObjectWindows uses the full power of C++, including facilities such as multiple inheritance and polymorphism to allow users to derive new classes easily and with few restrictions. The result is a framework that is consistent in design and that hides many of the subtle complexities of programming for Windows, such as automatic GDI object creation and disposal.

MFC, on the other hand, uses a simple hierarchy, rooted at the class CObject. The hierarchy is only 4 classes deep, and makes little use of polymorphism — where a single interface is used over and over for similar things — , making MFC harder to learn and use. As a result, MFC is inordinately complex, in many cases providing little or no encapsulation of Windows details. And there are many issues in MFC which appear to be rather arbitrary, resulting in a somewhat inconsistent design depending on rigid data layouts.

Because ObjectWindows is more object oriented than MFC, it provides a stronger foundation for code reusability and is easier to learn. ObjectWindow's underlying use of exception handling makes it a far more robust framework suitable for every day and mission critical tasks. And, ObjectWindows provides a smooth migration to cross platform development via ObjectWindows for AppWare.

The following sections provide a detailed technical comparison of OWL and MFC.

ANSI Compliance

Quick Summary: ObjectWindows is completely ANSI compatible and fully exploits standard C++ facilities including templates and exceptions to increase reusability and robustness. MFC has no support for ANSI standard templates or exceptions.

ObjectWindows 2.0 leverages recent ANSI C++ additions. The main features are the use of template based response tables, standardized exception handling, the new standard class **string**, and templates, as described in the following sections. Use of templates and exception handling provides unequaled type safety and error handling capabilities.

ObjectWindows 1.0 made use of a C++ language extension -- called Dynamic Dispatch Virtual Tables (DDVTs) -- to bind Windows messages to C++ member functions. DDVTs represented an elegant solution to message binding, but were not portable to other ANSI-compliant C++ compilers. Borland has dropped the use of DDVT functions in favor of a new technique utilizing structures known as response tables that are fully ANSI compliant. MFC uses a similar technique, based on what are called message maps. See the section named *Message Routing* for a comparison of response tables with message maps.

Exception Handling

Quick Summary: Exception support in ObjectWindows is ANSI compliant and applied thoroughly and consistently throughout to give a simple, robust exception mechanism to users. MFC's exception support is clumsy, complex, non-standard and error-prone.

Exceptions allow programs to treat unusual or unexpected situations in a consistent and predictable manner locally at the sight of the unexpected event. Programmers spend most of their time on the normal case as they should. When a problem is detected at runtime, a function can throw an exception, which results in a non-local jump to another function that has established a handler for the type of exception thrown. Programs can throw exceptions which are bona fide C++ objects, which can then be caught by value or reference. Catching by reference allows polymorphic handling for exceptions that are part of a larger exception class hierarchy.

ANSI standard exception handling involves the introduction of three keywords into the C++ language: **try**, **throw** and **catch**, each with their own syntax. A function that executes code that may fail encloses the code in a try block, like this:

```
try {
  // do something
}
catch(xmsg& msg) {
  // use the string in msg to display an error message
}
```

Following the try block there are one or more catch blocks, each distinguishable by the exception type handled. The try block can contain any valid C++ expressions. Exceptions can be thrown by functions called within the try block, as well as in the try block itself. To throw an exception, the keyword **throw** is used like this:

```
// we ran out of disk space!
throw (xmsg("Disk Full!"));
```

The compiler locates the exception handler (if any) for the exception, and passes control to it, after unwinding the stack and destructing local objects that went out of scope in the process.

Uncaught exceptions are handled automatically. C++ defines a number of standard exceptions. These, and the standard OWL exceptions, are shown in the following figure:

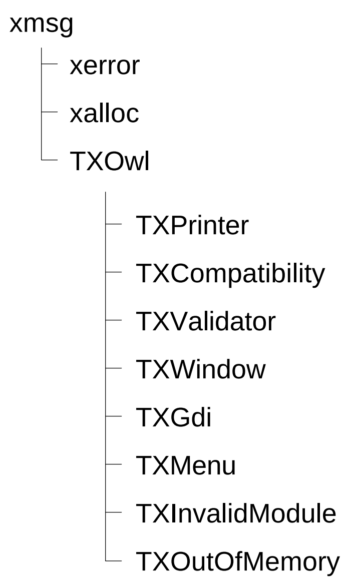


Figure 1 **-** The standard C++ and ObjectWindows 2.0 exceptions.

All the exceptions derived from TXOwl have default handlers in ObjectWindows, but applications can provide their own handlers for special cases. The default handlers typically display an error message and terminate the application. The handling of exceptions requires special intervention from the compiler, because non-local jumps can be performed, causing the stack to be unwound. During the process of stack cleanup, local objects whose stack is unwound must be destructed automatically, and the correct exception handler must be located and control given to it. Local objects must be destructed in the reverse order of construction, as is the case when local objects go out of scope normally.

MFC Exception Handling

Exception handling in MFC 2.5 is through the use of non-standard C style macros. MFC does not support ANSI standard exception handling, and as a result, it is limited, and awkward to use.

Although there are macros that use some of the same names as the ANSI keywords, their use is different. For example the CATCH macro is used like this:

```
CATCH (CFileException, theException) {
  if (theException->m_cause == CFileException::fileNotFound)
    ...
}
END_CATCH
```

The CATCH macro takes two parameters. The first specifies a type, the second is a pointer to an object of that type. This syntax has little in common with that of the standard **catch** keyword.

The MFC Exception Hierarchy

MFC defines a class hierarchy of exceptions as shown in the following figure:

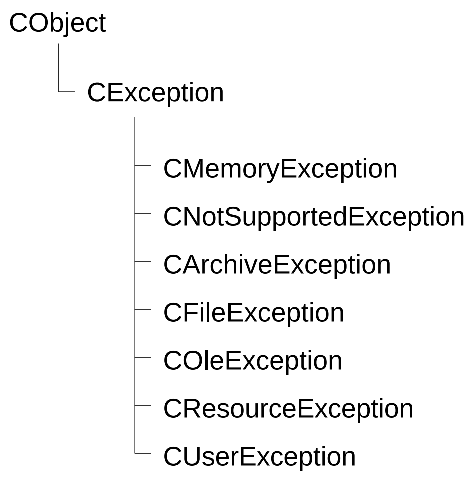


Figure 2 - The MFC exception class hierarchy.

The THROW macro is not used in user programs. To throw an exception, you must use one of the following MFC functions:

```
void AFXAPI AfxThrowMemoryException();
void AFXAPI AfxThrowNotSupportedException();
void AFXAPI AfxThrowArchiveException(int cause);
void AFXAPI AfxThrowFileException(int cause, LONG lOsError = -1);
void AFXAPI AfxThrowResourceException();
void AFXAPI AfxThrowUserException();
```

Using these macros makes it harder to deal with user defined exceptions. The THROW_LAST macro must be used to re-throw an exception from inside CATCH blocks.

Problems with MFC Exception Handling

Apart from the complexity imposed by MFC exception handling such as the confusion between macros and standard ANSI keywords and the combined use of macros and function calls, there are several other major problems with MFC exception handling.

1- You can't extend the MFC exception hierarchy by simply deriving your own class from class `CException`. To throw an application-specific exception, you must throw a `CUserException`, using the MFC function `AfxThrowUserException()`. There is no way to distinguish one user exception from the other.

2- Exceptions thrown in class constructors will cause memory leaks because the matching destructor is not automatically called. This completely precludes the throwing of exceptions inside constructors, a common usage with ANSI C++ exception handling.

3- Local objects are destroyed, but not destructed during the process of stack unwinding. In other words, after a function ends, any local objects disappear, because the place they are stored on the stack goes away. But the destructors for these local objects are not called. Thus, no clean up is executed. This can result in memory leaks, bad pointers in lists, file handles that are not closed, and many other critical and hard to detect problems.

4- The exception hierarchy is not consistent. For example, `CArchiveException` is not derived from `CFileException`, even though `CArchive` objects deal exclusively with `CFile` objects.

5- Interface specifications for exceptions are not supported. The standard function `unexpected()` is not supported. According to the proposed ANSI C++ draft, a function can declare the types of exceptions that it, or any functions called by it, can throw. If any other exception types are thrown, the function `unexpected()` is called.

6- Too complex. Microsoft has attempted to support limited exception handling through a plethora of macros and non-standard functions. The end result is confusing and non-portable.

Class string

Quick Summary: Borland supports the ANSI standard string class; Microsoft does not use this standard, resulting in code which is non-portable and does not use ANSI exception handling.

The X3J16 ANSI committee approved the new standard class **string**, designed to support the most common string operations, with automatic memory allocation and management. Class **string** has several overloaded operators, such as *operator+=*, *operator==* and *operator=*, to facilitate common string operations.

MFC doesn't support the ANSI string class , using in its place a class called *CString*, which has less functionality than **string**. One of the biggest differences between **string** and *CString* is that the former has the ability to throw standard C++ exceptions, while the latter doesn't.

Templates

Quick Summary: Borland supports ANSI standard templates to allow easier code reuse without giving up type safety. MFC doesn't use templates, resulting in code which is more error-prone and harder to re-use.

The draft C++ standard calls for the support of parameterized types, known as *templates*. Both functions and classes may be created using templates, allowing users to create specific function and class implementations for a given series of data types. Containers are good examples of where template classes are convenient. Using a template class, you can designate a generic container, such as a linked list, that handles objects of some generic type T. The advantage of using templates is that the compiler creates a complete typed class based on T, and guarantees type-safety, since it knows what types are actually being handled.

MFC doesn't support templates, as proposed by the ANSI committee. Generic classes, such as containers, deal with `void*` types, forcing you to use typecasting. The use of typecasts not only places the burden for type identification on the user, but also opens the door to bugs, caused by incorrect type conversions. In effect, by not using templates, you must give up the benefit of strong type checking.

Summary

Borland's ObjectWindows 2.0 takes full advantage of ANSI standard C++ features including exceptions, templates and ANSI strings. These features make it easier to write code that is robust, reusable and portable. Because MFC does not support these facilities it is significantly harder to write reusable code and the resulting code is not only harder to use, it's non-portable.

Message Handling

Quick Summary: Message handling in ObjectWindows is easier to write, more flexible, and safer than MFC since it fully supports the use of multiple inheritance and is template based.

Windows programming poses special problems for C++ class hierarchies. Windows sends messages to a C callback function, where the message is decoded and processed in accordance with the message type. C++ class libraries for Windows must be able to direct Windows messages to C++ member functions, and provide a conversion of the generic WPARAM and LPARAM parameters into types that are message dependent -- a process known as message cracking.

Response Tables

Quick Summary: Both ObjectWindows and MFC use a message dispatch table mechanism to map Windows messages to the correct C++ member function. ObjectWindows uses a C++ standard approach that is clean and simple based on C++ templates. MFC's approach is non-standard and awkward and non type-safe. MFC is further limited to single inheritance.

ObjectWindows 2.0 handles Windows messages through entities known as *response tables*. These tables provide a connection between a Windows message and a C++ member function. ObjectWindows 1.0 accomplished this message mapping using virtual dispatch tables (DDVTs) and functions. Although the technique was elegant, it used a proprietary C++ language extension and was not portable. The new response tables are fully portable to ANSI-compliant C++ compilers, and provide additional flexibility. To use response tables with a window, you must declare the table in the window's header file, and define the table in the source file. Here is how a response table is declared in a sample window class:

```
class TMyWindow : public TFrameWindow {
  public:
    // ...
    void EvTimer(UINT);
    void CmAbout();
  DECLARE_RESPONSE_TABLE(TMyWindow);
};
```

The definition of the response table is put in the source file, and connects Windows messages to member functions of class `TMyWindow`. The definition would look like this:

```
DEFINE_RESPONSE_TABLE1(TMyWindow, TFrameWindow)
  EV_WM_TIMER,
  EV_COMMAND(CM_ABOUT, CmAbout),
END_RESPONSE_TABLE;
```

ObjectWindows 2.0 supports all the type of messages that can occur in Windows, i.e. Windows messages like WM_PAINT, user commands sent through WM_COMMAND messages, notification messages, etc. Standard Windows messages are mapped automatically to ObjectWindows member functions, so you don't have to specify their name on the response table. For example, the WM_TIMER message uses the EV_WM_TIMER macro to connect the message with the `EvTimer()` member function. For user commands, you must specify the member function name, using the EV_COMMAND macro.

MFC 2.5 uses an approach that is similar to ObjectWindows, but calls its message dispatching tables *message maps* instead of response tables. Different macros are involved, but the effect is the same. The code fragment below shows how the class `CMyWindow` would be implemented under MFC 2.5.

```
class CMyWindow : public CFrameWnd {
  public:
    // ...
    void OnTimer(UINT);
    void OnAbout();
    DECLARE_MESSAGE_MAP()
};

BEGIN_MESSAGE_MAP(CMyWindow, CFrameWnd)
        ON_WM_TIMER()
        ON_COMMAND(CM_ABOUT, OnAbout)
END_MESSAGE_MAP()
```

Message maps use an awkward syntax. Their declaration resembles a member function declaration, but has no semicolon at the end. Their definition resembles a struct declaration somewhat, but the individual map entries have no comma between them and the END_MESSAGE_MAP macro has no trailing semi-colon. Borland implemented response table so that their syntax is more in line with C++. The declaration DECLARE_RESPONSE_TABLE has a semicolon at the end. The definition DEFINE_RESPONSE_TABLE uses commas to separate the table entries, and the table ends with a semi-colon. There is another interesting point about message maps: notice how the DECLARE_MESSAGE_MAP macro doesn't take any parameters. MFC relies on an undocumented non-standard compiler extension to avoid having to know the name of the class being dealt with in a message map.

But apart from esthetics, the most limiting factor of message maps is that they don't support multiple inheritance. If you have a class that is derived from multiple base classes, you can only reference one of the base classes in the BEGIN_MESSAGE_MAP macro. The base classes you leave out will not be used during MFC's message dispatching, unless you add code of your own.

In ObjectWindows, response tables fully support multiple inheritance, thereby giving programmers greater flexibility in designing their applications.

Command Enabling

Quick Summary: ObjectWindows and MFC both have mechanisms to allow the response tables to control menu and toolbar items as focus changes.

The ObjectWindows response tables function in a way similar to the MFC message maps. Where ObjectWindows and MFC differ is in the way these tables are used by the application frameworks code. ObjectWindows has code to automatically enable or disable menu and toolbar items based on the contents of the response table of the active window. When the focus moves to a new window, ObjectWindows checks the response tables of the windows that are in the view chain. For each item on the menu and toolbars, ObjectWindows checks the response tables to see whether a handler is defined. If so, the item is painted normally, otherwise the item is grayed out and disabled. No code is necessary from the application.

MFC handles things in a similar fashion. For each item on the menu and toolbar, MFC checks the message maps in the view chain, looking for either a handler or an ON_UPDATE_COMMAND_UI entry that takes the item's ID as a parameter. If an ON_UPDATE_COMMAND_UI entry is found, MFC calls the member function bound to it, passing it a `CCmdUI*` parameter. The function can then use the argument to enable or disable the menu or toolbar item. If no ON_UPDATE_COMMAND_UI entry is found, MFC enables a menu item if there is handler for it, otherwise it grays the menu item out. A short example may be helpful. Assume your window has a menu item called **Edit | Select All**, with the ID `IDM_SELECTALL`. By creating the following entry in the window's message map:

```
BEGIN_MESSAGE_MAP (CMyWindow, CFrameWnd)
    ON_UPDATE_COMMAND_UI(IDM_SELECTALL, OnUpdateSelectAll)
    ON_COMMAND (IDM_SELECTALL, OnSelectAll)
END_MESSAGE_MAP ()
```

The member function `OnUpdateSelectAll()` would be called when `CMyWindow` became the active window. The `CCmdUI*` parameter will point at the menu or toolbar entity the MFC wants to get the status of. If `CMyWindow` wants to disable the **Edit | Select All** menu command, the code for `OnUpdateSelectAll()` might look like this:

```
afx_msg void CMyWindow::OnUpdateSelectAll(CCmdUI* pCmdUI)
{
  pCmdUI->Enable(FALSE);
}
```

ObjectWindows has a feature similar to let programs selectively enable or disable menu/toolbar entries. The feature is based on EV_COMMAND_ENABLE() entries in the window's response table. Before displaying a Window, ObjectWindows checks the response tables for the windows in the view chain for EV_COMMAND_ENABLE() entries that are mapped to menu and toolbar IDs. If it finds any, it calls the bound handler functions, passing to them a `TCommandEnabler&` parameter. Here is how the response table might be defined for class `TMyWindow`:

```
DEFINE_RESPONSE_TABLE1(TMyWindow, TFrameWindow)
  EV_COMMAND(IDM_SELECTALL, CmSelectAll),
  EV_COMMAND_ENABLE(IDM_SELECTALL, CmEnableSelectAll),
END_RESPONSE_TABLE;
```

The code for `CmEnableSelectAll()` might look like this:

```
void TMyWindow::CmEnableSelectAll(TCommandEnabler& CommandEnabler)
{
  CommandEnabler.Enable(... anything to select...)
}
```

The `TCommandEnabler&` parameter references the menu or toolbar item that ObjectWindows is inquiring about.

Summary

Both ObjectWindows and MFC fully support message handling . However, only ObjectWindows gives developers complete flexibility in allowing the use of multiple inheritance. This is important for sophisticated applications which may have multiple command sets enabled at different times.

The Document/View model

Quick Summary: ObjectWindows makes it easier to create applications based on the document/view model by using standard ANSI templates. This also eliminates common errors that can happen using MFC's approach.

A new feature added in ObjectWindows 2.0 is the Document / View model, which provides greater power and flexibility. In the document/view model, the management of a window's data is treated as a separate task from the visual presentation of the data. MFC 2.5 also supports the Document/View model, as discussed below.

The ObjectWindows Approach

Quick Summary: Separating the data from its presentation is an important abstraction concept. It allows multiple views on the same data. ObjectWindows does not require that the view object be a window and viewers can be DLLs.

In ObjectWindows 2.0, objects derived from `TDocument` are used to manipulate a window's data, and a class derived from `TView` displays the data on the screen. One immediate benefit of the separation of documents from views is that you can create multiple viewers for the same data. For example, you could have viewers that display document data in different formats. You might have two viewers for TIF (tagged image format) files. One could show the files in text format, the other could display the file images as bitmaps.

An important difference between `CView` and `TView` is that while `CView` is derived from `CWnd`, and is thus a window, `TView` is not derived from `TWindow`. `TView` is instead derived from `TEventHandler`, so it can be used to represent non-window items, such as OLE links.

Another difference is that ObjectWindows supports viewers that are linked both statically and dynamically. With DLL viewers, you can add new viewers to an application without adding any additional code. ObjectWindows simplifies the use of viewers and documents with drag and drop. For applications created with AppExpert, you can add a new viewer by simply dragging and dropping a viewer DLL into the application, with out even recompiling the code. The example program STEP12 — that ships with Borland C++ 4.0 — provides an example of how to do this.

Document objects are associated with View objects. The association is created through a template class, called by the document template, and you can create multiple instances of these templates. Each instance can specify details such as the default directory to search for the document files and what default file suffix to use with the documents. Here is how a document template is declared and defined:

```
DEFINE_DOC_TEMPLATE_CLASS(THexDocument, THexView, THexTemplate);
THexTemplate MyTemplate("Hex File Viewer", "*.*", "C:\\",
                        *.hex", dtAutoDelete);
```

The macro DEFINE_DOC_TEMPLATE_CLASS declares the document template class `THexTemplate`, and tells it to associate the `THexDocument` class with the `THexView` class. The next line instantiates an object of type `THexTemplate`, called `MyTemplate`. This document template object is passed a number of parameters. The first is a string description, which ObjectWindows displays in a floating popup menu when the **File | New** command is selected. The second is a filter used to display files in the **File | Open** dialog box managed automatically by ObjectWindows. The third parameter is the default directory of the document files, the fourth is the default extension to add to file names, and the last parameter is a flag telling ObjectWindows how to handle views and documents. The `dtAutoDelete` flag tells ObjectWindows to delete a document object when its last associated view is closed.

All the document template instances in your program are handled by a document manager object, attached to the application object. With ObjectWindows, you can create two kinds of applications:

1- Simple applications, in which each window handles its data and its presentation.
2- Document/View applications, which use documents, views and template classes.

The derived class is responsible for creating the manager in the `InitMainWindow()` member function, as shown in the following code fragment.

```
void TMyApp::InitMainWindow()
{
  DocManager = new TDocManager(dmMDI);
}
```

You tell the manager whether you want an MDI or SDI application, and from that moment on all the details are handled automatically for you, eliminating any possibility of errors.

The MFC Approach

Quick Summary: MFC is much more rigid and limited than ObjectWindows due to its lack of template use and because application objects must manage associations themselves.

Although MFC 2.5 supports the document/view model, its implementation is different in a number of areas. You must create document classes derived from `CDocument`, and view classes derived from `CView`, but that's where the similarity ends. Rather than use template classes to manage views and documents, MFC uses two classes: `CSingleDocTemplate` and `CMultiDocTemplate`. These classes do not use C++ templates, despite what their names suggest. There is no stand-alone document manager in MFC. The application object itself manages the document/view associations. You add associations using the member function `CApplication::AddDocTemplate()`, which must be called in `CApplication::InitInstance()`, like this:

```
BOOL CMyApp::InitInstance()
{
  // ...

  AddDocTemplate(new CMultiDocTemplate(IDR_MYFRAME,
                      RUNTIME_CLASS(CHexDocument),
                      RUNTIME_CLASS(CHexFrame),
                      RUNTIME_CLASS(CHexView)) );
// ...
}
```

The comparable ObjectWindows code is much simpler:

```
THexTemplate MyTemplate("Hex File Viewer", "*.*", "C:\\",
                        *.hex", dtAutoDelete);
```

The MFC code allows you to accidentally add both SDI and MDI document/view associations to the same application. In ObjectWindows, you make the SDI/MDI decision only once, when the document manager is created. After that, the system knows what kind of structure it is working with.

The first parameter passed to `CMultiDocTemplate` is the ID of the menu to associate with class `CHexView`. ObjectWindows has built-in support for menus, making it unnecessary for you to have to pass this information to the document manager. See the section entitled Menus for further information.

The remaining parameters passed to `CMultiDocTemplate` use the macro RUNTIME_CLASS, which essentially extracts fragments of classes for subsequent use. All of this unnecessary overhead could have been avoided through support for standard C++ templates.

Summary

Although both ObjectWindows and MFC support the document/view model, the ObjectWindows support is easier to use and less error-prone.

Special Window Types

Quick Summary: ObjectWindows includes a number of special window types that facilitate the design of Windows applications. MFC has no comparable support for layout windows, and its support for toolbars, status lines and palettes is significantly more difficult to use. By having a richer set of classes, and more built-in functionality, ObjectWindows reduces the amount of code necessary to create modern user-interfaces.

Windows applications today use a number of embellishments that have almost become standard features. Most of these features are supported through special windows that make programs easier to use and understand. Among the most common of these windows are toolbars, status bars and palettes.

ObjectWindows 2.0 provides full support for all of these windows, while MFC provides limited support. The following sections will discuss the various window types in more detail.

Layout Windows

Quick Summary: Constraint driven windows are important for configurability and flexibility because their size and shape are completely driven by a set of constraints that allow them to adapt as the controlling parameters change. ObjectWindows provides this through powerful Layout Windows while MFC has no comparable capability. Layout windows are particularly important for international applications.

Status bars, toolbars and palettes share common features: they all are positioned in a certain way with respect to their parents. The ability to position one window based on some attribute of the parent window is useful in many cases. Consider a window that displays a clock in its lower right corner. If the clock must occupy only a small proportion of the parent window's client area, then it must be able to compute not only its position, but also its size based on the parent's size. ObjectWindows 2.0 has a new class called `TLayoutWindow` that allows you to attach positioning and sizing constraints to a window . These constraints are handled internally as a set of linear equations, which ObjectWindows solves to determine how to display a window.

Suppose you have a window in which two child windows need to be positioned in a constrained way. One window needs to be positioned at the lower right of the parent window, and have a size that is a certain fraction of the parent's size. The other window might need to be positioned right next to the first child window. The following code shows how all this could be accomplished with ObjectWindows.

```
#include <owl\framewin.h>
#include <owl\applicat.h>
#include <owl\layoutwi.h>
#include <owl\color.h>


class TColorWindow : public TWindow {
public:
  TColorWindow(TWindow* parent, TColor color)
        : TWindow(parent, "") {
              SetBkgndColor(color);
              Attr.Style = WS_CHILD | WS_BORDER | WS_VISIBLE;
        }
};

class TMyWindow: public TLayoutWindow {
protected:
  TWindow* w1;
  TWindow* w2;
  void SetupWindow();
public:
  TMyWindow(TWindow* parent)
        : TLayoutWindow(parent, 0) {
        Attr.Style |= WS_BORDER;
        w1 = new TColorWindow(this, TColor::LtRed);
        w2 = new TColorWindow(this, TColor::LtCyan);
  }
};

void TMyWindow::SetupWindow()
{
  TLayoutWindow::SetupWindow();

  TLayoutMetrics metrics;

  // layout constraints for right window
  metrics.X.Set(lmLeft, lmPercentOf, lmParent, lmRight, 60);
  metrics.Y.Set(lmTop, lmPercentOf, lmParent, lmBottom, 60);
  metrics.Width.Set(lmRight, lmPercentOf, lmParent, lmRight, 95);
  metrics.Height.Set(lmBottom, lmPercentOf, lmParent, lmBottom, 95);
  SetChildLayoutMetrics(*w1, metrics);

  // layout constraints for left window
  metrics.X.Set(lmRight, lmSameAs, w1, lmLeft);
  metrics.Y.Set(lmTop, lmSameAs, w1, lmTop);
  metrics.Width.Absolute(100);
  metrics.Height.Absolute(20);

  SetChildLayoutMetrics(*w2, metrics);

  Layout();
}
class TLayoutApp : public TApplication {

public:
  void InitMainWindow() {
        MainWindow = new TFrameWindow(0, "Using Layout Windows",
                                     new TMyWindow(0) );
        }
};

int OwlMain(int, char**)
{
  return TLayoutApp().Run();
}
```

Listing 1 - A short example using `TLayoutWindow`.

The code in listing 1 produces a window that looks like this:

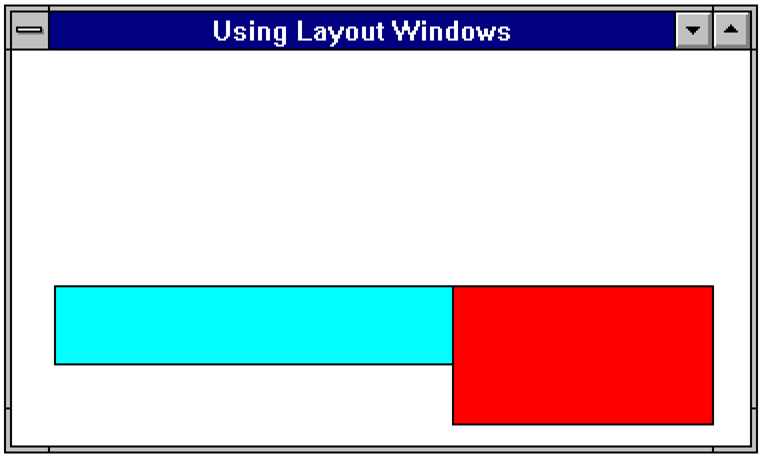


Figure 3 - The constrained child windows produced by the code in Listing 1.

The child window on the right of figure 3 is constrained with the code:

```
  metrics.X.Set(lmLeft, lmPercentOf, lmParent, lmRight, 60);
  metrics.Y.Set(lmTop, lmPercentOf, lmParent, lmBottom, 60);
  metrics.Width.Set(lmRight, lmPercentOf, lmParent, lmRight, 95);
  metrics.Height.Set(lmBottom, lmPercentOf, lmParent, lmBottom, 95);
```

so its width and height are 35% of the parent window's width and height. The second child window is positioned to the immediate left of the first window, and is constrained to have a width of 100 pixels and a height of 20 pixels. The window is constrained using the code:

```
  metrics.X.Set(lmRight, lmSameAs, w1, lmLeft);
  metrics.Y.Set(lmTop, lmSameAs, w1, lmTop);
  metrics.Width.Absolute(100);
  metrics.Height.Absolute(20);
```

where the X and Y constraints use the first child window as the reference window. Because only the first child window's size is related to the parent window's, resizing the parent window causes the first child to be resized, but not the second one.

Layout windows allow you to specify different combinations of position and size constraints. For example you can make the height of a window a function of the width, or vice versa, or the height/width or the window a function of a parameter of the parent window. Class `TLayoutWindow` is a base class used for the status bars, toolbars and tool boxes used in ObjectWindows.

MFC has no equivalent class to `TLayoutWindow`. As a result, developers must code from scratch the logic to reposition child windows, toolbars or palettes manually.

Layout windows are particularly important for international applications, because they allow dynamic resizing of dialogs and windows during translation of user interfaces.

Toolbars

Quick Summary: ObjectWindows provides truly object-oriented toolbar classes. MFC does not use an object-based approach — it just uses bitmaps. This makes it very hard to provide programmatic control over toolbars and means ObjectWindows allows a much more configurable toolbar-based application.

ObjectWindows 2.0 has a built-in class to support toolbars. The class is called `TControlBar`, is derived from `TLayoutWindow`, and displays itself right under the menu window's menu bar. MFC has a class called `CToolBar` that also displays a toolbar under the menu bar, but the similarity ends there. Toolbars are designed to hold buttons. The buttons in ObjectWindow's `TControlBar` are derived from the ObjectWindows class `TGadget`, a class that supports most of the functionality needed by toolbar buttons. ObjectWindows gives you full programmatic control over the placement, sizing, and functions of toolbar buttons. To create a toolbar, you only need a few lines of code. Consider the sample toolbar shown in figure 4.

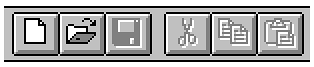


Figure 4 - A sample toolbar created with ObjectWindows.

The code required to create this toolbar is the following:

```
  TControlBar* cb = new TControlBar(parent, direction);
  cb->Insert(*new TButtonGadget(CM_FILENEW, CM_FILENEW));
  cb->Insert(*new TButtonGadget(CM_FILEOPEN, CM_FILEOPEN));
  cb->Insert(*new TButtonGadget(CM_FILESAVE, CM_FILESAVE));
  cb->Insert(*new TSeparatorGadget(6));
  cb->Insert(*new TButtonGadget(CM_EDITCUT, CM_EDITCUT));
  cb->Insert(*new TButtonGadget(CM_EDITCOPY, CM_EDITCOPY));
  cb->Insert(*new TButtonGadget(CM_EDITPASTE, CM_EDITPASTE));
```

You can control the spacing and positioning of toolbar buttons at runtime. Toolbars and toolbar buttons are handled the same way dialog boxes and child controls are, from the programmer's perspective. ObjectWindows automatically disables buttons for which there is no handler in the response tables along the view chain.

In contrast, MFC doesn't use objects at all in its toolbars -- all you specify is a series of bitmaps. MFC handles the individual buttons by itself, making it very difficult for you to customize a standard behavior. You have no programmatic control over the positioning and spacing of the items on the toolbar. To make changes, you must develop different sets of toolbar bitmaps, and switch between them. Consider the toolbar shown in figure 5.

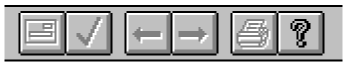


Figure 5 - A toolbar created with MFC.

To create such a toolbar, you would first declare a data `CToolBar` data member in the main window class, like this:

```
class CMainFrame : public CFrameWnd
{
  // ...
protected:
  CToolBar   m_wndToolBar;
// ...
};
```

Then would create a `CToolBar` window inside the `OnCreate()` member function of the main window, like this:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
  // ...
  m_wndToolBar.Create(this);
  m_wndToolBar.LoadBitmap(ID_TOOLBAR);
  // ...
}
```

To connect the bitmaps of the toolbar buttons with command IDs, MFC requires you to create an array of ID values. The order of the array must correspond exactly to the order of the bitmaps in the resource file. Here is how the ID array might look for the toolbar in figure 5:

```
static UINT BASED_CODE buttons[] =
{
       ID_EDIT_NEW_CHECK,
       ID_EDIT_COMMIT_CHECK,
       ID_SEPARATOR,
       ID_PREV_CHECK,
       ID_NEXT_CHECK,
       ID_SEPARATOR,
       ID_FILE_PRINT,
       ID_APP_ABOUT,
};
```

It is interesting to see how ObjectWindows and MFC compare when it comes to making changes to toolbars. For example, by default MFC creates a toolbar to handle button represented by bitmaps 16 pixels wide and 15 pixels high. If you want larger buttons, you can't just paint bigger buttons with AppStudio and expect MFC to resize the toolbar. Instead you need to call the function `CToolBar::SetSizes()`, passing to it the size of the toolbar buttons and the size of the bitmap images in each button. You would use code like this:

```
SIZE buttonSize = {24, 24};
SIZE imageSize = {18, 18};
m_wndToolBar.SetSizes(buttonSize, imageSize);
```

You have to be careful that the button size is 6 pixels larger than the image size in both width and height. Otherwise, MFC will not display the buttons correctly.

Using ObjectWindows, all you need to do is draw bigger bitmaps for the toolbar images. ObjectWindows is smart enough to figure out that it needs to draw bigger buttons and a bigger toolbar without any need to write code.

Status Bars

Quick Summary: ObjectWindows provides a very object-oriented status bar class that results in a simpler yet much more customizable status bar in applications. MFC uses a limited and rigid C-based approach.

Both ObjectWindows and MFC have classes to support status bars. The main difference between the classes is their design. ObjectWindows uses an object-oriented approach to simplify toolbar handling without limiting its ability to be customized. MFC uses a traditional C approach which requires calling functions with a limited set of pre-defined arguments. ObjectWindows uses the class `TStatusBar`, MFC the class `CStatusBar`. Both support automatic keyboard tracking for the toggle keys such as the CAPS lock key, the NUM lock key, the insert key, etc.

Consider how indicators are inserted into a status bar. Indicators are fields that show the status of a toggle key, such as the CAPS lock key.

ObjectWindows uses an object-oriented approach. Class `TStatusBar` has options that allow you to enable the indicator fields you want, and also to control where the status bar is displayed in its parent window. The following code shows how a status bar is typically created:

```
void TMyApp::InitMainWindow()
{
  Frame = new TDecoratedMDIFrame(...);

  TStatusBar* sb = new TStatusBar(0, TGadget::Recessed,
        TStatusBar::CapsLock | TStatusBar::NumLock | TStatusBar::Overtype);
  Frame->Insert(*sb, TDecoratedFrame::Bottom);
}
```

In MFC, you pass an array of values to the `Create()` member function of `CStatusBar`, so you would create a status bar like this:

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpcs)
{
  static UINT BASED_CODE indicators[] = {
  ID_SEPARATOR,
  ID_INDICATOR_OVR,
  ID_INDICATOR_CAPS,
  ID_INDICATOR_NUM
};

 if (CFrameWnd::OnCreate(lpcs) == -1)
    return -1;

  CStatusBar  myStatusBar;
  if (!myStatusBar.Create(this) ) return;
  if (!myStatusBar.SetIndicators(indicators, sizeof(indicators)/sizeof(UINT)))
    return;
}
```

MFC status bars are relatively inflexible, because they only support features for which there is an option bit. ObjectWindows builds status bars using objects (derived from `TGadget`), so you can add any functionality you want by deriving a new class from `TGadget` and inserting it into the status bar. For example, say you wanted to put an edit control on the status bar, as shown in figure 6.

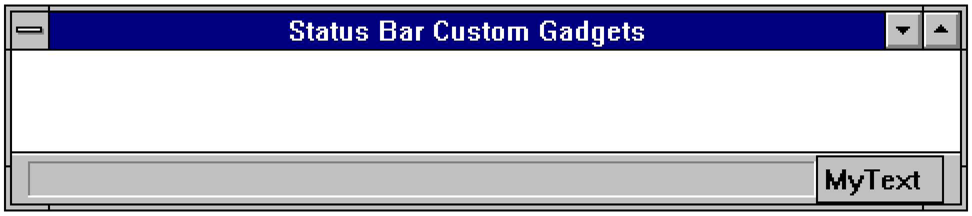


Figure 6 - A edit on an ObjectWindows status bar.

ObjectWindows has a class called `TControlGadget`, derived from `TGadget`, that allows you to associate a gadget control to any window you want. Using `TControlGadget`, an edit control could be inserted into a status bar using the code:

```
  TStatusBar* sb = new TStatusBar(0, TGadget::Recessed);

  const int ID_SEARCHTEXT = 100;
  char buffer [80];
  TEdit* searchText = new TEdit(sb, ID_SEARCHTEXT, buffer, 0, 0, 60, 20, 80);
  sb->Insert(*new TControlGadget(*searchText) );
```

There are a number of `TStatusBar` member functions you can use to determine the exact positioning of controls. You can insert controls on the left or right of another control, or you can insert a control at a specific absolute position. `TStatusBar` also has functions to make it shrink to the height of the largest control, and status bar controls can be tailored with specific window styles (e.g. borders, colors, etc.).

Tool Palettes

Quick Summary: ObjectWindows uses objects of small size built up into a class hierarchy supporting tool palettes that allows features to be changed with minimal code changes. MFC, on the other hand, exposes a very complex and error-prone approach to tool palettes.

Tool palettes are small moveable windows that contain bitmaps of "tools". Such palettes are used in paint programs, desktop publishing programs, etc. ObjectWindows and MFC both support Tool Palettes, but under different names.

ObjectWindows tool palettes are built using a number of different classes. The individual buttons are derived from class `TButtonGadget`. The buttons are coordinated by the parent window class `TToolBox`. The frame of the tool palette is built using class `TFloatingFrame`. Because of the breakdown into small classes, you can change any tool palette feature with minimal code. Here is how a typical ObjectWindows tool palette is created:

```
void TMyApp::InitMainWindow()
{
  TWindow& client = *new TWindow(...);
  TDecoratedFrame* frame =  new TDecoratedFrame(0, Name, client);
  SetMainWindow(frame);

  TToolBox* tb = new TToolBox(0);
  tb->Insert(*new TButtonGadget(CM_TOOL+0, CM_TOOL+0,
                  TButtonGadget::Exclusive,   TRUE, TButtonGadget::Down));
  tb->Insert(*new TButtonGadget(CM_TOOL+1, CM_TOOL+1,
                  TButtonGadget::Exclusive, TRUE));
  tb->Insert(*new TButtonGadget(CM_TOOL+2, CM_TOOL+2,
            TButtonGadget::Exclusive, TRUE));
  // insert additional buttons
  // ...

  new TFloatingFrame(&client, "", tb, TRUE,
             TFloatingFrame::DefaultCaptionHeight, TRUE);
}
```

Since the buttons on the tool palette are full-blown C++ objects, they can support any functionality you want. MFC uses a different approach, similar to its status bar implementation. MFC has a class called `CToolBar`, from which you derive a class to support a tool palette. You need the derived class, because `CToolBar` doesn't have a frame around its window. Tool palettes typically have a frame and caption, allowing the user to move the window around on the screen. Class `CToolBar`'s `Create()` member function takes an array of bitmap IDs. When the user clicks the tool palette, the owner window receives a command with the bitmap ID. Here is how you create a tool palette in MFC:

```
class CPaletteBar : public CToolBar {...};

class CMyWnd : public CFrameWnd
{
protected:
  CPaletteBar m_wndToolPalette;
  // ...
};

int CMyWnd::OnCreate(LPCREATESTRUCT lpcs)
{

static UINT BASED_CODE palette[] =
{
  ID_TOOL1,
  ID_TOOL2,
  // ...
};

 if (CFrameWnd::OnCreate(lpcs) == -1)
    return -1;

  if (!m_wndToolPalette.Create(this, nLeft, nTop) ||
    !m_wndToolPalette.LoadBitmap(IDB_PALETTE) ||
    !m_wndToolPalette.SetButtons(palette,
         sizeof(palette)/sizeof(UINT), 3) )
    return -1;
  return 0;
}
```

IDB_PALETTE is the resource ID of a bitmap which actually contains an array of bitmaps. The array `palette` must contain IDs that appear in the same order as the bitmaps in the resource IDB_PALETTE. Rather than implementing individual classes, then combining them to make a tool palette, with MFC you have to draw a series of bitmaps, create a parallel array of bitmap IDs, derive a new class, write all the code to draw borders, captions, ... pass arrays around, call Create member functions... That's an awful lot of work to accomplish what should be a standard feature. The possibility of errors and bugs is high. Using ObjectWindows, the code is trivial.

Summary

ObjectWindows was designed from the outset to allow developers to create sophisticated user interfaces with automatic resizing of child windows, toolbars, palettes and status lines. The objects that support these user-interface elements were created to allow easy use, without limiting the types of items that could be displayed. MFC, on the other hand, has no support for automatic resizing of child windows and it's other objects are limited in how they can be customized.

Dialog Box Controls

Quick Summary: ObjectWindows treats dialog boxes and child controls just as any other object. In MFC, use of these objects requires additional overhead through the use of "helper" functions.

C++ programmers expect to be able to handle a dialog box's child controls in an object-oriented manner. ObjectWindows supports this style of programming directly. All you have to do is create an interface object for each item you wish to manipulate on a dialog box. Assume you use Resource Workshop to create the dialog box with a static text field. To create a C++ interface object for the field, the constructor of the dialog box would have the statement:

```
textField = new TStatic(this, IDC_TEXT, 10);
```

where the data member `textField` is declared `TStatic*`. Once the interface object is created, it can be used as a C++ replacement for the static text windows element. To set its text you would use the code:

```
textField->SetText("Hello");
```

ObjectWindows allows you to handle dialog box controls exactly the same way as other C++ objects.

With MFC it is a different story. In MFC you don't create C++ objects for items that are part of a dialog box resource. Instead you let the Window dialog box manager construct the dialog box, then, when you need to access the control, you must have an inline helper function to retrieve a `CWnd` pointer, which you must then typecast into an appropriate object. To set a text field in an MFC dialog box, you would first need to have the helper member function shown below:

```
class CMyDialog : public CDialog
{
public:
  // ...
  CStatic&  Text()  {return *(CStatic*)GetDlgItem(IDC_TEXT); }
// ...
};
```

You would think that you could call `GetDlgItem()` at dialog box creation time, inside `OnInitDialog(...)`, and store the returned pointer in a data member. This won't work, because the pointer returned by `GetDlgItem()` is subject to change at runtime. Using typecasting to return an object of the correct type is unsafe and totally violates the idea of object-oriented programming. If you use the wrong control ID value, you will probably get a pointer to the wrong type of control, with unpredictable consequences.

To set the text of the `CStatic` object, you must use the helper function to get a reference to an MFC interface object, then use the object like this:

```
Text().SetWindowText("Hello");
```

VBX Controls

Quick Summary: ObjectWindows takes a very consistent approach to controls — VBX or otherwise. MFC requires special VBX handling. In addition, MFC provides no drag-and-drop support for VBX controls.

VBX controls are Windows custom controls, developed primarily for Visual Basic. VBX controls are available from a wide number of vendors, and differ from traditional controls in the number of properties and attributes that are under user control. Both ObjectWindows and MFC provide support for VBX 1.0 controls, but differ in the manner and extent of their support.

Under ObjectWindows, VBX controls are handled the same as standard ObjectWindows controls: you add them to a dialog box using Resource Workshop and then add C++ code to create an interface object for them. VBX controls send special notification messages to their parent. ObjectWindows handles these through the class `TVbxEventHandler`. Dialog boxes that incorporate VBX controls need to be multiply derived from the base class `TVbxEventHandler`, and no additional logic is required. Assume you want to include a spreadsheet VBX control in the resource file for the dialog class `TMyDialog`. The class would be declared like this:

```
class TMyDialog: public TDialog, public TVbxEventHandler {

protected:

  TVBXSpreadsheet* spreadsheet;

public:

  //...
};
```

where the class `TVBXSpreadsheet` is assumed to be the name of the VBX control class. Handling the control is not any different from handling ObjectWindows controls. To create an interface object for a VBX control, you simply create an object of the correct type. For `TVBXSpreadsheet`, you would do something like this in the constructor for `TMyDialog`:

```
spreadsheet = new TVBXSpreadsheet(this, ID_SPREADSHEET);
```

Then you could manipulate the control through its member functions. VBX controls are designed for use with Visual Basic, and therefore lack the member functions that other controls often have, such as `SetText()`, `GetSelIndex()`, etc. With VBX controls, there are standard interface functions to get and set the control's various properties. Each control has its own set of properties, so each control is different. Assuming the `TVBXSpreadsheet` control has a property of type `NumberOfColumns`, you might set this property with the ObjectWindows code:

```
spreadsheet->SetProp("NumberOfColumns", 5);
```

To read a control's property, you similarly use the `GetProp()` function, like this:

```
int columns;
spreadsheet->GetProp("NumberOfColumns", columns);
```

VBX notification messages are handled by ObjectWindows. You associate a particular notification message with an ObjectWindows handler using response tables. The response table for `TMyDialog` might look like this:

```
DEFINE_RESPONSE_TABLE2(TMyDialog, TDialog, TVbxEventHandler)
  EV_VBXEVENTNAME(ID_SPREADSHEET,"LostFocus",EvLostFocus),
  EV_VBXEVENTNAME(ID_SPREADSHEET,"GainedFocus",EvGainedFocus),
END_RESPONSE_TABLE;
```

The response table has EV_VBXEVENTNAME entries to bind VBX notification messages to ObjectWindows handlers. The member function `TMyDialog::EvLostFocus()` would be called when the VBX control with ID ID_SPREADSHEET sent the notification message "LostFocus". The function `EvLostFocus()` would be called by ObjectWindows with a parameter pointing to a struct containing information about the VBX event and the control that send it.

Both ObjectWindows and Resource Workshop support VBX drag and drop controls. Inspecting a control's properties, you will see entries like `DragIcon` and `DragMode` (if the control supports drag and drop). In MFC, drag and drop properties are not supported at all for VBX controls. Using AppStudio, the properties don't even show up in the properties dialog box.

Under MFC, VBX controls are supported in a different manner. For a dialog box to use VBX controls, its class must declare data members to point at each control. This is similar to ObjectWindows. The problem is that regular controls and non-VBX controls are not allowed to have pointers to them in the class declaration. So there are different initial rules for VBX and non-VBX controls. VBX notification messages are bound to member functions using message maps, as in ObjectWindows, but with an extra level of complexity. A class `CMyDialog` using VBX controls that sent "LostFocus" and "GainedFocus" notification messages to the parent would be declared like this:

```
class CMyDialog : public CDialog
{
 public:
 // ...
  //{{AFX_DATA(CMyDialog)
  CVBControl* m_spreadsheet;
  //}}AFX_DATA
// ...

protected:

  // ...
  // Generated message map functions
  //{{AFX_MSG(CMyDialog)
       afx_msg void OnLostFocus(UINT, int, CWnd*, LPVOID);
       afx_msg void OnGainedFocus(UINT, int, CWnd*, LPVOID);
//}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
```

The data member `m_spreadsheet` would be used to access the VBX control. The message map would need to look like this:

```
BEGIN_MESSAGE_MAP(CMyDialog, CDialog)
  //{{AFX_MSG_MAP(CMyDialog)
  ON_VBXEVENT(VBN_LOSTFOCUS, ID_SPREADSHEET, OnLostFocus)
  ON_VBXEVENT(VBN_GAINEDFOCUS, ID_SPREADSHEET, OnGainedFocus)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

But as you can see, there is no mention of the original VBX properties "LostFocus" and "GainedFocus", which are the actual messages sent by the control. Instead, you must go through a process of registering VBX events with a VBX registration map, and obtain integer values like VBN_LOSTFOCUS and VBN_GAINEDFOCUS to use in the message map. For class `CMyDialog`, the VBX registration map would look like this:

```
//{{AFX_VBX_REGISTER_MAP()
       UINT NEAR VBN_LOSTFOCUS = AfxRegisterVBEvent("LostFocus");
       UINT NEAR VBN_GAINEDFOCUS  = AfxRegisterVBEvent("GainedFocus");
//}}AFX_VBX_REGISTER_MAP
```

VBX notification message handlers are sent 4 parameters by MFC. The function `OnLostFocus()` would be declared like this:

```
void CMyDialog::OnLostFocus(UINT uCode, int nIndex, CWnd* pWnd, LPVOID lpParams);
```

`uCode` is the notification code, and is almost never used. `nIndex` is also fairly useless, and is the index of the event in the VBX control's event table. `pWnd` points to the control that sent the notification, and `lpParams` points to a block of parameters that describe the notification in detail.

VBXGen

Borland C++ 4.0 provides a tool called VBXGen to make VBX controls more accessible to C++ programmers. VBXGen reads a VBX binary and creates a C++ class for the VBX control, including data members and member functions for its properties. This makes accessing and subclassing a VBX control far easier than using the normal VBX interface.

Enhanced Controls

Quick Summary: ObjectWindows fully supports Borland's own 3D controls as well as those provided by Microsoft. MFC has support for neither.

The best-looking applications today use 3D controls. The standard Windows controls are 2D, so Borland developed its own library of custom 3D controls, contained in the file BWCC.DLL. The following figure shows an example of a dialog box using the BWCC enhanced controls.

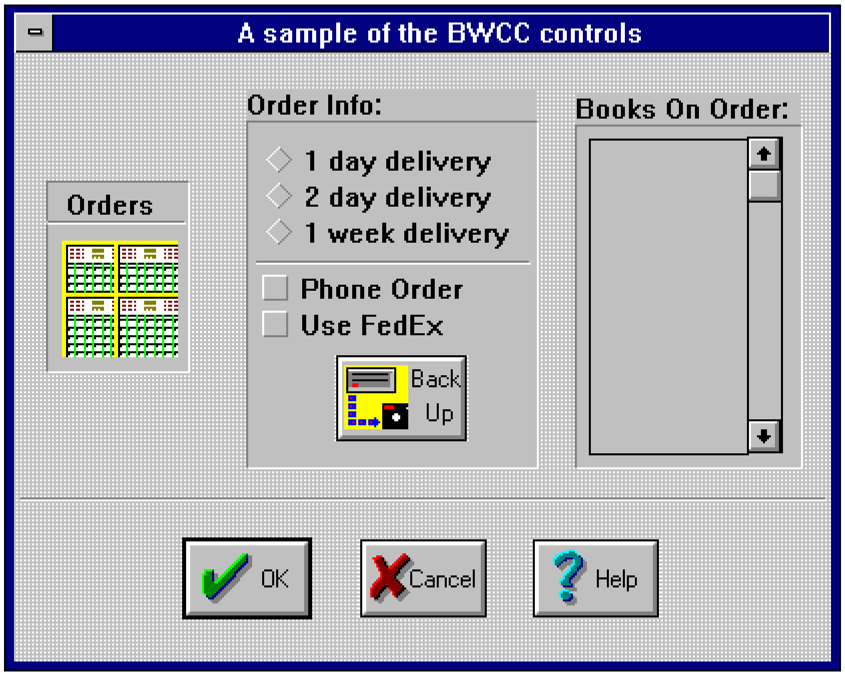


Figure 7 - A dialog box using some of the BWCC controls.

The `Back Up` bitmapped button was created by assigning a bitmap to a BWCC button. Resource Workshop supports BWCC controls, handling them no differently from other standard Windows controls. Resource Workshop is also fully extensible, allowing third party custom controls and VBX controls to be added and used like the built-in controls.

ObjectWindows supports BWCC controls by default, and BWCC controls can be added to MFC applications by explicitly loading the BWCC.DLL library. Although BWCC controls are usable in both application frameworks, there are differences in the amount of support each environment provides. For example, MFC's AppStudio does not display BWCC controls (or other third party Windows controls), making it difficult -- at best -- to add them to dialog boxes.

Although Borland has been using 3D controls for over 2 years, Microsoft only recently recognized the need for standard 3D controls, issuing a technical note entitled *Adding 3-D Effects To Controls*, by K. Marsh and W. Cherry on the Developers' Network CD. The note describes a DLL called CTL3D.DLL developed by Microsoft that gives a 3D look to the standard Windows controls, like listboxes, group boxes and radio buttons.

Transferring Data

Quick Summary: ObjectWindows provides very simple straightforward mechanisms to transfer data from dialogs to the underlying object. MFC's DDX data exchange mechanism is much more complex and therefore harder to use correctly.

Dialog box child controls are used to get input and show results. C++ child controls are useful if they simplify the way data is transferred to and from the Windows elements attached to them. ObjectWindows supports two methods for transferring information to child controls. The first uses so-called *transfer buffers*, the second uses C++ data members. ObjectWindows transfer buffers are structs that contain one field for each child control that is enabled to transfer its data. The layout of the struct must match exactly the order of creation of the dialog box's controls. For example, the following transfer buffer:

```
struct TTransferBuffer {
  BOOL  MrTitle;
  char  NameEdit [10];
  BOOL  CheckBox1;
} MyTransferBuffer;
```

could be associated with the dialog box whose constructor looked like this:

```
TMyDialog::TMyDialog(TWindow* parent)   : TDialog(parent, ID_MYDIALOG)
{
  new TRadioButton(this, ID_RADIOBUTTON, 0);
  new TEdit(this, ID_EDIT, 10);
  new TCheckBox(this, ID_CHECKBOX);

  SetTransferBuffer(&MyTransferBuffer);
}
```

Each control type is associated with a specific type in the transfer buffer: radio buttons use BOOL values, edit controls use `char` arrays, checkboxes use BOOL values, etc. Once a transfer buffer has been created, all ObjectWindows needs is a way to access it, accomplished with the statement:

```
SetTransferBuffer(&MyTransferBuffer);
```

in the constructor of `TMYDialog`. ObjectWindows automatically transfers data from the dialog box controls to the transfer buffer when the user closes the dialog with the OK button. You can also call the function `TransferData(tdGetData)` at any time to cause an immediate transfer of data into the buffer on demand.

When a dialog box is opened, ObjectWindows automatically initializes the box's controls with the data stored in the transfer buffer. You can also copy data from the transfer buffer to the dialog box at any time by calling the function `TransferData(tdSetData)`. The use of `TransferData()` is not limited to child controls. You can use the function to get/set the data of any child window, nested arbitrarily deep in a window hierarchy.

You can also transfer data in or out of dialog box controls using standard C++ data members. This is a new method that was previously unavailable under ObjectWindows 1.0. Basically, a dialog box (or a window using child controls) can be equipped with a data member for each control that is enabled to transfer data. Checkboxes and radio buttons use a BOOL data member, edit controls use a char array, listboxes use a `TListBoxData` member, etc. To initialize the data in a dialog box, you only need to setup the dialog's data members before displaying the window, using code like:

```
TMyDialog dlg;       // create a local dialog box object

// initialize the dialog's child control data
dlg.radioButton = TRUE;
dlg.checkBox = FALSE;
strcpy(dlg.edit, "Name");
```

```
// display the dialog
if (dlg.Execute() == IDOK) {

  // use the data entered into the dialog box...
  if (dlg.radioButton)
    ...
  if (dlg.checkBox)
    ...
}
```

The function `TDialog::Execute()` no longer deletes the dialog box before returning (as it did in ObjectWindows 1.0), so you can access the dialog's data members after `TDialog::Execute()` returns. Using local objects lets the compiler take care of deleting the dialog box object.

MFC handles the transfer of data to and from child controls through a mechanism called *Dialog Data Exchange*, or DDX for short. With DDX, each control in the dialog box is associated with data member of the dialog box class. Consider a dialog box containing a radio button. The dialog class would be declared something like this:

```
class CMyDialog : public CDialog
{
public:
  // ...
  //{{AFX_DATA(CMyDialog)
  int m_RadioButton;
  CString m_Edit;
  int m_CheckBox;
  //}}AFX_DATA

protected:

  virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support

  // ...
};
```

The declarations bracketed by AFX_DATA are the dialog box's data map. Each entry is attached to a control at runtime, using special DDX function calls called in the member function `DoDataExchange()`, like this:

```
void CMyDialog::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);

  //{{AFX_DATA_MAP(CMyDialog)
  DDX_Radio(pDX, ID_RADIOBUTTON, m_RadioButton);
  DDX_Text(pDX, ID_EDIT, m_Edit);
  DDX_Check(pDX, ID_CHECKBOX, m_CheckBox);
  //}}AFX_DATA_MAP
}
```

Each type of control has a series of different DDX functions, based on the type of the data member associated with the control. MFC calls `DoDataExchange()` automatically to initialize the child controls before displaying a dialog box, and to get the new values of the controls when then dialog is closed with the OK button. You can cause information to be transferred to or from the exchange data members by calling the function `UpdateData(BOOL)`. The call `UpdateData(TRUE)` causes data to be transferred from the child controls to the data members. The call `UpdateData(FALSE)` causes a transfer in the opposite direction.

Data Validation

Quick Summary: ObjectWindows uses a very object-oriented approach to data validation where a validator object is attached to a control; no extra code is needed because the validator handles it. In

MFC data validation is handled through a series of global functions and validation only happens during data exchange. This also makes MFC data validation dangerous and crash-prone.

When a user enters data into a dialog box and clicks the OK button, you normally need code to check that the data is valid. Numbers may have to be range-checked, strings looked up in a table, etc. MFC and ObjectWindows both support data validation, in varying degrees, but the two frameworks use completely different approaches. ObjectWindows 2.0 adopts a fully object-oriented method, using a class hierarchy of validators. MFC uses the usual C approach, using a series of global functions for various validation types.

With ObjectWindows, the idea is that controls that need to be validated can have a validator object attached to them. When the dialog is closed with the OK button, ObjectWindows calls the `CanClose()` member function for each control. The `CanClose()` function of `TEdit` objects checks to see if a validator is attached to the control, and invokes it if so. The validators are supported by the class hierarchy shown in figure 8:

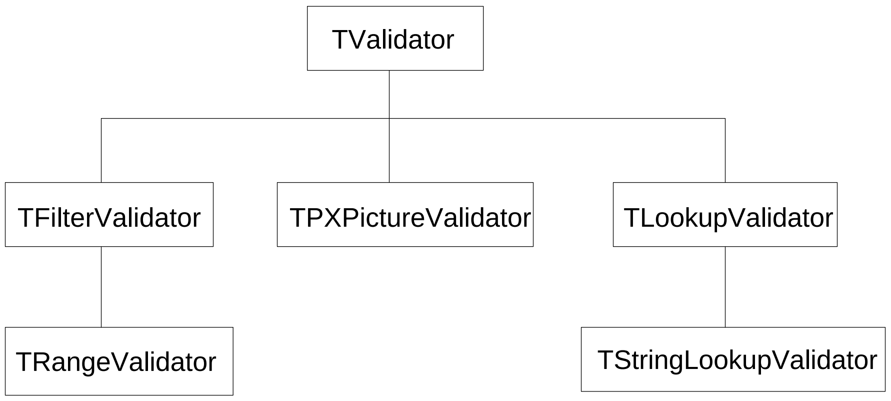
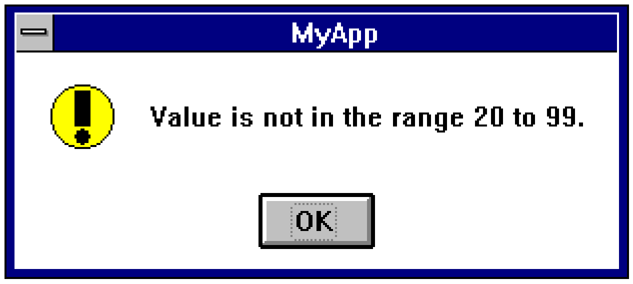


Figure 8 - The class hierarchy of ObjectWindows validator objects.

To use one of the validators, you first create a `TEdit` control to be validated, then create a validator object, then attach the validator to the `TEdit` control. The following code creates an edit field that accepts numbers in the range 20..99:

```
edit = new TEdit(this, 103, 10);
edit->SetValidator(new TRangeValidator(20, 99) );
```

The validator not only checks the edit control when the dialog box is closed with the OK button, but also has the option of checking every character types into the associated edit field. Validators of type `TRangeValidator` are associated with edit fields expecting numbers. `TRangeValidator` allows only digits to be entered. All other characters are ignored. When the control losses the focus, or the dialog box is closed with the OK button, the validator does a range check on the number entered. If the value is out of range, ObjectWindows displays a dialog box that looks like this:



Value is not in the range 20 to 99.

Figure 9 - The error message displayed by an ObjectWindows `TRangeValidator` object.

MFC handles the entire subject of data validation differently. MFC couples data validation very tightly to a dialog box's data exchange mechanism. To validate data, there are no objects involved, just global function calls. You have to add explicit code to your application to support validation. For example, if the dialog box `CMyDialog` contained an edit field expecting a number in the range 20..99, you would first need to setup the dialog box for data exchange, using the member function `DoDataExchange()`. Data validation is not supported unless data exchange is enabled. The dialog box would need to declare a data member to accept the control's data, like this:

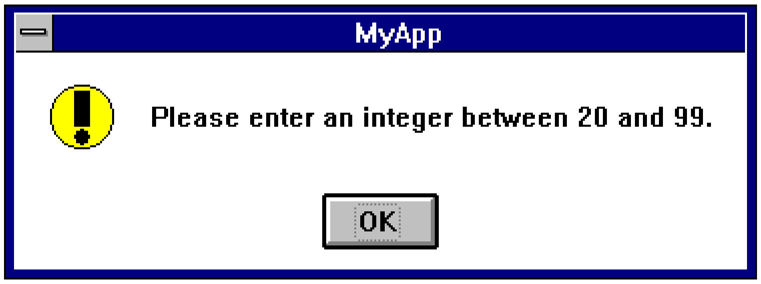
```
class CMyDialog : public CDialog
{
  // ...
public:
  //{{AFX_DATA(CMyDialog)
  int m_nEmployeeAge;
  //}}AFX_DATA
protected:
  virtual void DoDataExchange(CDataExchange* pDX);
};
```

To validate the data, you need to add code to the dialog's `DoDataExchange()` member function, doing something like this:

```
void CMyDialog::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);

  //{{AFX_DATA_MAP(CMyDialog)
  DDX_Text(pDX, ID_EMPLOYEEAGE, m_nEmployeeAge);
  DDV_MinMaxInt(pDX, m_nEmployeeAge, 20, 99);
  //}}AFX_DATA_MAP
}
```

The `DDX_Text()` call transfers data from the edit control to the data member `m_nEmployeeAge`, after converting the data to integer format. The next `DDV_MinMaxInt()` call validates the data. MFC poses the arbitrary restriction that a DDV call immediately follow a DDX call for a given control. DDV calls placed elsewhere in your code are not guaranteed to work. The DDV functions are only called during the data exchange process. Because of this, the user is not restricted from entering alphabetic characters into a number field. Only after entering data and attempting to close the dialog box with the OK button do you get an error message. No error occurs by simply moving the focus to another control. The DDV function displays an error message if you attempt to close a dialog box containing out-of-range numbers. The function `DDV_MinMaxInt()` displays the following message in the example code above:



Please enter an integer between 20 and 99.

Figure 10 - The error message displayed by the MFC function `DDX_MinMaxInt()`.

The data validation process buried inside MFC is capable of throwing MFC exception macros. If you create a dialog box that handles its child controls through pointers or references to `CWnd` objects, the exception handling mechanism may cause a non-local jump, resulting in the destructors not being called for the child controls. Your code would most likely terminate or crash at this point.

Custom Validators

Quick Summary: Due to ObjectWindow's object-orientation, it is easy to add custom validation. Not so in MFC where its decidedly non-trivial to do so.

It is easy to customize data validation for ObjectWindows controls: all you do is derive a class from one of the standard validators, and add or change the necessary features. For example, say you had a dialog box in which the state of a checkbox determined which set of strings were valid for an edit control. You could handle this situation easily by deriving a class from `TLookupValidator`, overriding the function `Lookup`, like this:

```
class TMyLookupValidator: public TLookupValidator {

public:

  BOOL Lookup(const far char* str) {
    TMyDialog* dlg = TYPESAFE_DOWNCAST(TMyDialog*,Parent);
    if (!dlg) return 0;
    if (dlg->CheckBox->GetCheck() == BF_CHECKED) {
      // the checkbox is checked
      return !stricmp(str, "String 1");
    }
    else {
      // the checkbox isn't checked
      return stricmp(str, "String 2");
    }
  }
};
```

You would then use a `SetValidator()` call to connect an object of type `TMyLookupValidator` to your an edit control.

With MFC, customizing data validation is not trivial, and can involve anything from the addition of C code to the `DoDataExchange()` function of a dialog box to the creation of several C functions. Data validation is further complicated by the fact that it is intimately tied in with dialog data exchange. To use the example above, an MFC implementation would require a new DDV function, but not a new DDX one, since the data being transferred (of type `CString`) is already supported by the built-in function `DDX_Text()`. The dialog box code might look like this:

```
void CMyDialog::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);

  //{{AFX_DATA_MAP(CMyDialog)
  DDX_Check(pDX, ID_CHECKBOX, m_bCheckBox);
  DDX_Text(pDX, IDC_EDIT1, m_strEdit);
  DDV_MyText(pDX, m_bCheckBox, m_strEdit);
  //}}AFX_DATA_MAP
}
```

The function `DDV_MyText(CDataExchange*, BOOL, CString&)` is the custom validator, and must be called immediately after the DDX function for the edit control being processed. `DDV_MyText(...)` would be implemented using code like this:

```
void AFXAPI DDV_MyText(CDataExchange* pDX, BOOL bCheckBox, CString& strEdit)
{
  if (bCheckBox) {
    if (strEdit == "String 1")
      return TRUE;
    else {
      AfxMessageBox("String 1 expected");
      pDX->Fail();
      return FALSE;
    }
  }
  else {
    if (strEdit == "String 2")
      return TRUE;
    else {
      AfxMessageBox("String 2 expected");
      pDX->Fail();
      return FALSE;
    }
}
```

Note that `DDV_MyText(...)` is a global function, so it belongs to no classes. The tendency to use global helper functions throughout MFC goes against the grain of good object-oriented programming practices and is something you expect in a C library, not a C++ class library. The matter isn't just cosmetic. Not utilizing class objects or class data members means you don't get to reuse or inherit functionality. For example, if you need to validate a custom data type, you need to write a DDX function for it, containing statements that would be unnecessary in a C++ class hierarchy. A typical custom DDX function could look like this:

```
void AFXAPI DDX_Check(CDataExchange* pDX,
                      int nEditID,
                      BOOL bCheckBox,
                      CString& strEdit)
{
  HWND hWndCtrl = pDX->PrepareEditCtrl(nEditID);
  if (pDX->m_bSaveAndValidate) {
    if (!GetEditData(hWndCtrl, bCheckBox, strEdit) ) {
      AfxMessageBox(IDS_INVALID_VALUE);
      pDX->Fail();
    }
  else
    SetEditData(hWndCtrl, strEdit);
}
```

The calls to `CDataExchange::PrepareEditCtrl()` and `CDataExchange::Fail()` wouldn't be necessary if the data exchange and validation mechanism were built around class objects. There are also lots of subtle details that you may need to be aware of about class `CDataExchange` to write a correct set of custom DDX and DDV functions.

MDI

Quick Summary: ObjectWindows makes it easy to create MDI and SDI applications. MFC uses a somewhat similar approach.

Multiple Document Interface (MDI) applications are very common these days, and ObjectWindows makes it just as easy to create an MDI app as an SDI one. The main window of the application determines whether an app is MDI or SDI. The main window is created in the `TApplication::InitMainWindow()` member function. For an MDI app, the code would look like this:

```
void TMyMDIApp::InitMainWindow()
{
  MainWindow = new TMDIFrame("App Name", "MDIMenuID");
}
```

The constructor used to create the `TMDIFrame` window is declared like this:

```
TMDIFrame(const char far* title, TResId menuResId,
                      TMDIClient& clientWnd = *new TMDIClient,
                      TModule*  module = 0);
```

The constructor takes a reference to a `TMDIClient` object, which is created automatically by default. `TMDIClient` manages the MDI child windows, and often is sufficient for ordinary programs. If you wish to use a custom `TMDIClient` object, you only need to derive a class from `TMDIClient`, and use it in the constructor call for `TMDIFrame`, like this:

```
  MainWindow = new TMDIFrame("App Name", "MDIMenuID", *new TMyMDIClient);
```

ObjectWindows MDI programs usually derive a class from `TMDIClient` to support menu and tool bar commands, but you can also handle these commands at the application object level, or by deriving a class from `TMDIFrame`. SDI applications can be built either by deriving a class from `TFrameWindow`, or by creating a customized `TWindow`-derived object to handle the main window's client area. The following code shows how an SDI app might be created using the first approach:

```
class TMySDIWindow : public TFrameWindow {
public:
  TMySDIWindow(TWindow* parent, const char* title)
       : TFrameWindow(parent, title), TWindow(parent, title)
        {     AssignMenu(MYMENU_ID);    }

// ...
};

class TMySDIApp : public TApplication {
  public:
    TMySDIApp() : TApplication("MyApp") {}
    void InitMainWindow()
        { MainWindow = new TMySDIWindow(0, "SDI Window"); }
};
```

Both classes `TFrameWindow` and `TMDIFrameWindow` have built-in support for certain standard menu commands, such as **File | Open**, **File | Exit**, etc.

Under MFC, an MDI application is created using a process similar to ObjectWindows: you derive a class from `CMDIFrameWnd`, create an dynamically allocated instance of the class, and assign its address to the data member `TWinApp::m_pMainWnd`, like this:

```
class CMyMDIFrameWnd : public CMDIFrameWnd
{
  // ...
};

BOOL CMyMDIApp::InitInstance()
{
        m_pMainWnd = new CMyMDIFrameWnd;
        pMainFrame->ShowWindow(m_nCmdShow);
        pMainFrame->UpdateWindow();
        return TRUE;
}
```

To create an SDI application with MFC, the process is almost the same, except the main window is derived from `CFrameWnd`, instead of `CMDIFrameWnd`, like this:

```
class CMySDIFrameWnd : public CFrameWnd
{
  // ...
};
```

The code in `TApp::InitInstance()` would remain as previously shown for `CMyMDIApp`.

GDI Classes

Quick Summary: ObjectWindows provides a rich set of classes that support Windows graphics calls. The object oriented nature of ObjectWindows provides greater flexibility and scalability than MFC does.

ObjectWindows 2.0 has a large variety of classes that completely encapsulate the Windows GDI objects, such as device context, pens, brushes and fonts. Figure 11 shows the class hierarchy of these classes:

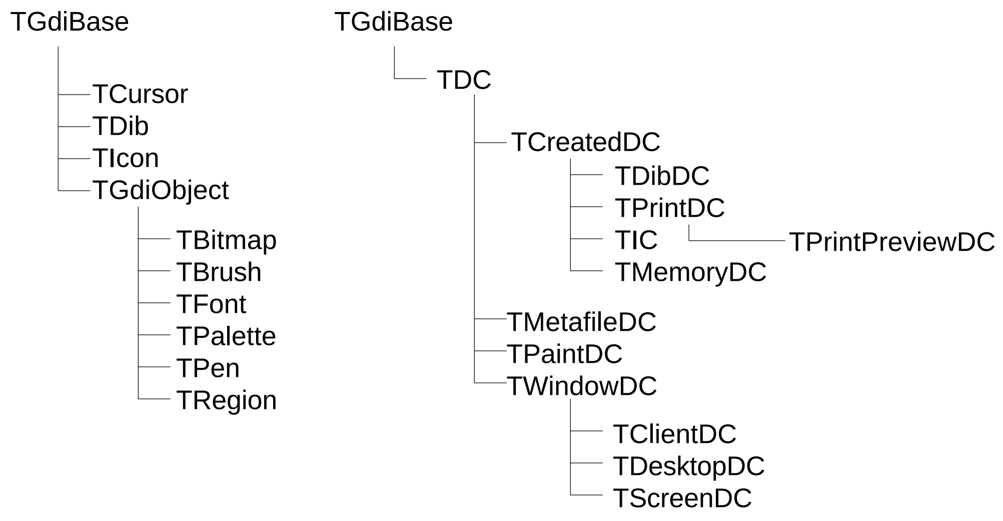


Figure 11 **-** The hierarchy of ObjectWindow's GDI classes.

A common problem with traditional Windows programs is memory leakage, related to GDI operations. For example, if you create a pen, use it, then forget to delete it (or delete it while it is selected into a device context), you'll have a GDI memory leak. A leaked GDI object is also called an *orphan*. ObjectWindows helps you avoid orphans, using the base class `TGDIObject`, from which the ObjectWindows GDI classes are derived. When a GDI object, such as a `TPen`, goes out of scope, ObjectWindows can automatically delete the Windows pen attached to it as soon as it is able (when it is finally deselected from the DC). By default ObjectWindows attempts to delete GDI objects attached to ObjectWindows objects that go out of scope, but the option can be disabled if necessary. The option is known as *orphan control*.

Creating ObjectWindows GDI objects is very simple. To create a `TPen` object, you would use code like:

```
TPen pen(TColor::Magenta);
```

To create a `TBrush`, you would use code like:

```
TBrush brush(TColor(100, 100, 100) );
```

ObjectWindows makes heavy use of default parameters, to simplify constructor calls. Obviously you can supply actual arguments for those parameters whose defaults are not what you want. Using ObjectWindows GDI objects is much simpler than using straight Windows GDI calls. To use a `TPen` you would use code like this:

```
void TMyWindow::DrawLine(TDC& dc)
{
  // create a pen
  TPen pen( TColor(100, 200, 300) );

  // select it into the paint context
  dc.SelectObject(pen);

  // draw a line
  dc.MoveTo(30, 30);
  dc.LineTo(10, 100);

  // unselect the pen
  dc.RestorePen();
}
```

When the `TPen` object goes out of scope, it will automatically delete the associated GDI pen object. In order for ObjectWindows to be able to delete the GDI pen, you must select the pen out of the device context, using the function `TDC::RestorePen()`. If RestorePen() was not called, the pen handle would remain usable in the DC until it was deselected. At that point the orphan control would delete it. The real power of ObjectWindows GDI objects becomes apparent when you use multiple objects together.

MFC also has classes to encapsulate GDI operations, but offers much less power and variety. The following table shows the ObjectWindows GDI classes, with the corresponding MFC ones.

ObjectWindows Class	Equivalent MFC Class
TGdiBase	
TGdiObject	CGdiObject
TIcon	
TCursor	
TDib	
TRegion	CRegion
TBitmap	CBitmap
TFont	CFont
TPalette	CPalette
TBrush	CBrush
TPen	CPen
TDC	CDC
TWindowDC	CWindowDC
TPaintDC	CPaintDC
TCreatedDC	
TMetafileDC	CMetaFileDC
TDesktopDC	
TScreenDC	
TClientDC	CClientDC
TDibDC	
TPrintDC	
TIC	
TMemoryDC	
TPrintPreviewDC	

Table 1 - The ObjectWindows GDI classes, with the corresponding MFC classes.

Although MFC has many corresponding classes to ObjectWindows, the classes are normally not equivalent. MFC supports an API that corresponds almost directly to the lower level Windows API. This may make it easier for C programmers to use MFC, but certainly doesn't help in reducing the complexity of Windows programming. The following code shows how to use a CPen object to draw a line.

```
void CMyWindow::DrawLine(CDC* pDC)
{
  CPen pen;
  if (!pen.CreatePen(PS_SOLID, 2, RGB(0,0,0) ) )
    return;
  CPen* pOldPen = pDC->SelectObject(&pen);
  pDC->MoveTo(30, 50);
  pDC->LineTo(40, 50);
  pDC->SelectObject(pOldPen);
}
```

Every time you create a GDI object like a pen or a brush with MFC, you always have to check the return value, because resource creation is subject to failure, and MFC only performs a simple translation of function calls like `pen.CreatePen(...)` into direct Windows API calls. ObjectWindows is considerably more sophisticated, and uses exception handling to deal with resource allocation errors, making it unnecessary for you to constantly check return values. This is a subtle but very important point because programmers are much more productive if they are allowed to program for what is supposed to happen instead of always thinking of every possible eventuality for every line of code.

MFC makes almost no use of default arguments in the GDI objects, forcing you to pass a great deal of parameters around. When you select objects into a device context, you have to store a pointer to the previous object, so that you can later restore the original object.

Printer Support

Quick Summary: ObjectWindows makes it easy to add printer support to an application regardless of the type of information being displayed. Printer support is far easier to use and far more flexible than with MFC and is not as restricted.

ObjectWindows 2.0 makes it simple to add printer support to your application. Two classes do the basic work. Class `TPrinter` handles the printer, calling the necessary DLLs. Class `TPrintOut` is a class you derive your own class from to print your application. For each page to be printed, ObjectWindows calls the virtual function `TPrintout::PrintPage(int, TRect&, unsigned)`.

To implement printer support, a class needs to create both a TPrintout-derived and a TPrinter object like this:

```
class TWindowPrintout : public TPrintout {
protected:
    TWindow* Window;
public:
 // ...
 TWindowPrintout(const char* title, TWindow* window)
    : TPrintout(title) {Window = window;}
  void PrintPage(int page, TRect& rect, unsigned flags);
};

class TMyWindow: public TFrameWindow {
  TPrinter* Printer;
public:
  TMyWindow(TWindow* parent, const char* title)
    : TFrameWindow(parent, title),
      TWindow(parent, title)
  { Printer = new TPrinter;
    // ...
  }
void CmFilePrint()
  { TWindowPrintout printout("Printout", this);
    printout.SetBanding(TRUE);
    Printer->Print(this, printout, TRUE);
  }
  // ...
};
```

Of course the printer object will need to be deleted in the class destructor. The printing function can be as simple as this:

```
void TMyWindow::PrintPage(int, TRect& rect, unsigned)
{
  Window->Paint(*DC, FALSE, rect);
}
```

Normally you will want to scale the print DC so the printout has the same aspect ratio as the screen, an operation involving calls to the printer DC member functions inside class `TPrintDC`.

MFC also supports printing, but using few classes and lots of function calls. The class `CView` is where printing is handled. The class has a number of member functions, such as `OnPreparePrinting()`, `OnBeginPrinting()`, `OnPrint()` and `OnEndPrinting()`, which you must override in your class derived from `CView`. Incidentally, printing is supported only when using the document/model mode. Simple applications that just have windows, without document objects, will not support printing.

You call the function `CView::DoPreparePrinting()` to make MFC display a Print dialog box and create the Print device context. Because you can't override this function (it isn't declared **virtual**!), in your derive `CView` classes, it is difficult to change the default behavior. Many applications use a custom Print dialog box, or need considerable flexibility in setting up the Print DC. MFC doesn't even have a class to handle Print DC's, so there are a lot of details that you need to take care of in your own code.

Resources

Quick Summary: Menus, bitmaps, metafiles and fonts are richly supported in ObjectWindows in a hierarchical, flexible, object-oriented approach which gives finer control over behavior while at the same time requiring less coding. This makes ObjectWindows programmers working with resources more productive than those using MFC.

Both ObjectWindows and MFC have classes or member functions to support the basic Windows resources: menus, bitmaps, fonts, accelerators and strings. The two frameworks, as usual, differ in their degree of support. The following sections describe the resource support briefly.

Menus

Quick Summary: ObjectWindows has very powerful capabilities when dealing with menus. The key concept is menu merging in which ObjectWindows takes care of the details for you while MFC forces the programmer to sweat those details.

ObjectWindows has two basic classes to encapsulate Windows menus: `TMenu` and `TSystemMenu`, with simple member function calls to manipulate all the elements of menus and allowing you to create bitmapped menus. One of the most interesting innovations in ObjectWindows menus is the ability to design menus as combinations of other menus, allowing the system to combine menus together at runtime -- just as in OLE 2.0. In traditional Windows programs, you create menus using Resource Workshop. You then load the menu resource and attach it to your main window. With ObjectWindows 2.0, there is much more flexibility.

Consider designing the menus for an MDI editor. You can design a barebones menu to be shown when the application starts, when no editors are open. The menu might have only **File** and **Help** submenus. When a child editor window is opened, you will want to extend the main menu to include perhaps an **Edit**, **Search** and **Window** submenu. First you create separate menus in Resource Workshop, then you use objects of class `TMenuDescr` in your code to manage these menu fragments.

You attach a menu to the main window using the code:

```
MyApp::InitMainWindow()
{
    // create a main window
  MainWindow =  new TMDIFrame("Main Window", 0);

  // attach a menu to the main window
  MainWindow->SetMenuDescr(TMenuDescr(IDM_MAINMENU,1,0,0,0,0,1) );
}
```

`TMenuDescr` is a class whose constructor accepts a series of integer values right after the menu resource ID. These integers indicate the number of submenus the resource contains, and at which location in the main menu to insert each of these submenus. The two 1's appearing in the argument list indicate that there is one **File** submenu and one **Help** submenu. The submenus indicated in the argument list are assumed to refer to menu groups in the following order: {**File** menu, **Edit** menu, **Container** menu, **Object** menu, **Window** menu, **Help** menu}.

The child editor windows may have a small menu attached to them, using code like:

```
void TMyApp::CmFileNew()
{
  TMDIChild* child = new TMDIChild(*Client, "", new TEditFile(0, 0, 0));
  child->SetMenuDescr(TMenuDescr(IDM_EDITFILE_CHILD, 0, 2, 0, 0, 0, 0));
}
```

The child window's menu is assumed to contain two submenus, the first to be inserted in the **Edit** submenu, the second submenu right after it on the right side. When a child editor window is opened, ObjectWindows will merge any menu attached to it with the already existing main menu. The ability to design menus piecewise not only reduces the effort to design complex menuing systems, but also vastly decreases the amount of code necessary to support them in your application.

MFC has no equivalent classes or functions to handle the merging of menus.

Bitmaps

Quick Summary: ObjectWindows provides classes for device independent bitmaps, supports clipboard operations on bitmaps and supports reading and writing bitmaps to files. MFC supports none of this.

ObjectWindows 2.0 support bitmaps with the two classes `TBitmap` and `TDib`. The former supports device-dependent bitmaps, the latter device-independent bitmaps. The two classes have many similar member functions, such as `Width()`, `Height()`, `ToClipboard(Clipboard&)`, but `TDib` provides quite a bit more functionality than `TBitmap`. To create a `TDib` or `TBitmap`, all you need is one line of code:

```
TDib* MyDib = new TDib(*GetModule(), "MYDIB");
TBitmap* MyBitmap = new TBitmap(*GetModule(), "MyBITMAP");
```

To display a `TDib` or `TBitmap` in response to a WM_PAINT message, you would write something like this:

```
void TMyWindow::Paint(TDC& dc, BOOL, TRect&)
{
  TMemoryDC memDC(dc);
  memDC.SelectObject(*MyBitmap);
  dc.BitBlt(0, 0, MyBitmap->Width(), MyBitmap->Height(), memDC, 0, 0, SRCCOPY);
}
```

Class `TDib` knows how to read and write itself to a file. Both `TDib` and `TBitmap` also support clipboard operations, requiring only a few lines of code. For example, to paste a `TDib` from the clipboard, all you need to get a clipboard object, check that it contains an object of the right type, then create a new object based on the clipboard, like this:

```
void TMyWindow::CmPaste()
{
  TClipboard clipboard;
  if (!clipboard)  return;

  TDib*  newDib = 0;

  if (clipboard.IsClipboardFormatAvailable(CF_DIB) )
    newDib = new TDib(TDib(clipboard));
}
```

To paste bitmaps to the clipboard, the code is even simpler, reducing to this:

```
void TMyWindow::CmCopy()
{
  TClipboard clipboard;
  if (clipboard.EmptyClipboard() ) {
    TDib(MyDib).ToClipboard(clipboard);
  }
}
```

MFC has the class `CBitmap` to support bitmaps. There is no class for device-independent bitmaps. Class `CBitmap` provides only minimal encapsulation of Windows bitmaps. Clipboard operations are not supported, nor does the class know how to read or write bitmaps to a file.

Metafiles

Quick Summary: Windows metafiles — important efficient graphics storage objects — are encapsulated in ObjectWindows while MFC provides no support for these metafiles.

ObjectWindows 2.0 provides encapsulation of Windows metafiles, using the two classes `TMetaFileDC` and `TMetaFilePict`. Metafiles are used to store pictures as collections of GDI calls rather than as arrays of pixels, and therefore use very little storage. Metafiles are frequently used for cutting and pasting pictures to/from the clipboard. Metafiles can also be stored on disk, or even used to create `TBitmap` images. You can create a `TMetaFilePict` with code like this:

```
 void Image()
 {
  TMetaFileDC dc;
  TPen pen(TColor::Black);
  dc.SelectObject(pen);
  dc.MoveTo(0, 100);
  dc.LineTo(100, 100);
  TMetaFilePict picture(dc.Close() );
 }
```

Once you have a `TMetaFilePict` object, you can use it to create other objects, such as a `TBitmap`, with code like this:

```
TPalette* palette = new TPalette((HPALETTE)GetStockObject(DEFAULT_PALETTE));
TBitmap* bitmap = new TBitmap(*picture, *palette,
                              GetClientRect().Size() );

// use the bitmap...
```

MFC offers no support or encapsulation for metafiles.

Fonts

Quick Summary: ObjectWindows demonstrates its clean, object-oriented architecture in its support for Windows fonts. Simple constructors with default arguments do all the work. In MFC creating font objects is overly complex.

ObjectWindows uses the class `TFont` to encapsulate Windows fonts. You can create a `TFont` object either by passing all the font attributes, by passing a LOGFONT structure, or by passing another TFont object. The constructor has defaults for almost all its arguments, so you don't normally need to pass too many arguments. For example, you can create a complete font with the code:

```
TFont* font =  new TFont("Courier New", 10);
```

`TFont` objects are used in conjunction with TDC objects, which handle device context details. A font is used to display text using code like this:

```
void TMyWindow::Paint(TDC& dc, BOOL erase, TRect& rect)
{
  TFont font("Courier New", 10);
  dc.SelectObject(font);
  dc.TextOut(0, 0, "Text");
}
```

MFC uses class `CFont` to encapsulate fonts. You create `CFont` objects in two steps. First you declare a variable of type `CFont`. The constructor takes no parameters. Then you call the function `CFont::CreateFont()` or `CFont::CreateFontIndirect()` to setup the font characteristics. `CFont::CreateFont()` is declared like this:

```
BOOL CreateFont(int nHeight, int nWidth, int nEscapement,
                int nOrientation, int nWeight,
                BYTE bItalic, BYTE bUnderline,
                BYTE cStrikeOut, BYTE nCharSet,
                BYTE nOutPrecision, BYTE nClipPrecision,
                BYTE nQuality, BYTE nPitchAndFamily,
                LPCSTR lpszFacename);
```

As you can see, there are no defaults for the arguments, forcing you to supply an endless list of items -- most of which are not normally of interest -- and lookup all the details of things like the clipping precision or the strikeout mode. Possibly because of this, fonts are usually created using `CFont::CreateFontIndirect()`, which takes a pointer to a LOGFONT struct. `CFont` doesn't support much object-orientation. You create a `CFont` objects with code that looks almost the same as old C code did:

```
LOGFONT logfont;
memset(&logfont, 0, sizeof(logfont));
logfont.lfHeight = 40;
logfont.lfWeight = FW_BOLD;
strcpy(logfont.lfFaceName, "Arial");
CFont font;
font.CreateFontIndirect(&logfont) );
```

Actually, creating fonts this way is harder than just using straight C code by itself. CFont objects work with CDC classes, which handle device context details. You use `CFonts` like this:

```
void CMyWindow::OnPaint()
{
  // create a new font
  CFont font;
  font.CreateFontIndirect(...);

  // use the new font
  CPaintDC dc(this);
  CFont* pOldFont = dc.SelectObject(&font);
  dc.TextOut(10, 10, "Text");
  dc.SelectObject(pOldFont);
}
```

It is hard to see any benefit from the use of `CFont` over standard Windows API calls. Even the process of selecting and unselecting a font into the device context is required with `CFont`.

Containers

Quick Summary: ObjectWindows really shows its object-oriented strength on containers. Important concepts like ownership, cleanup on deletion and iteration are key to ObjectWindows implementation that uses templates extensively. The C-style collections provided by MFC are not really worth using.

One of the main components of medium and large OOP projects is the container. Novice C++ programmers usually resort to C methods to handle collections of things, not realizing that they are missing one of the greatest advantages of the language.

ObjectWindows Containers

Quick Summary: The BIDS classes provided by ObjectWindows include all the fundamental ones used in the object-oriented community. There are 11 basic types included.

ObjectWindows 1.0 used two different kinds of container classes: one that handled items derived from class `Object`, the other based on template classes that could handle any type of data. The former containers are still supported in ObjectWindows 2.0, but are considered obsolete, and are being phased out. The latter are called the BIDS (Borland International Data Structures) containers, and are designed to be the workhorses of larger ObjectWindows 2.0 applications. BIDS containers are split into two basic categories: FDS (Fundamental Data Structures) and ADT (Abstract Data Types). FDS containers represent basic memory organizations of objects, such as vectors, lists and hashtables. ADT containers are higher level ones, built using an FDS containers. Stacks, Arrays, Dictionaries, Bags and Sets are examples of ADT containers. Table 2 shows the FDS and ADT container types.

FDS Containers	ADT Containers
Binary Search Tree	Array
Hash Table	Dequeue
Singly Linked List	Dictionary
Doubly Linked List	Queue
Vector	Set
	Stack

Table 2 - The FDS and ADT containers in the BIDS container class hierarchy.

The table contains all the typical containers used throughout the object-oriented programming community. All containers are template classes, so you can create containers for any type of data you want. You can also create your own class hierarchy of objects, and insert any of them into the containers, using standard C++ template class notation. Each type of container can use direct objects, or pointers to objects. Many container types support sortable objects.

MFC Containers

Quick Summary: MFC's contains are C-style and do not use templates and are thereby quite inflexible. Only 3 basic types are provided. Worse, non-standard terminology inhibits understanding and communication. There is no type-safety and ownership is not enforced making memory leaks common occurrences.

In contrast, MFC doesn't support template classes, and hence resorts to a C style in dealing with collections. MFC has only three basic types of containers: Arrays, Lists and Maps. Microsoft calls maps what most of the OOP world calls dictionaries. Dictionaries are containers that manage associations. For example a symbol table is a dictionary, in which strings are associated with numeric values. Figure 12 shows the class hierarchy of MFC containers.

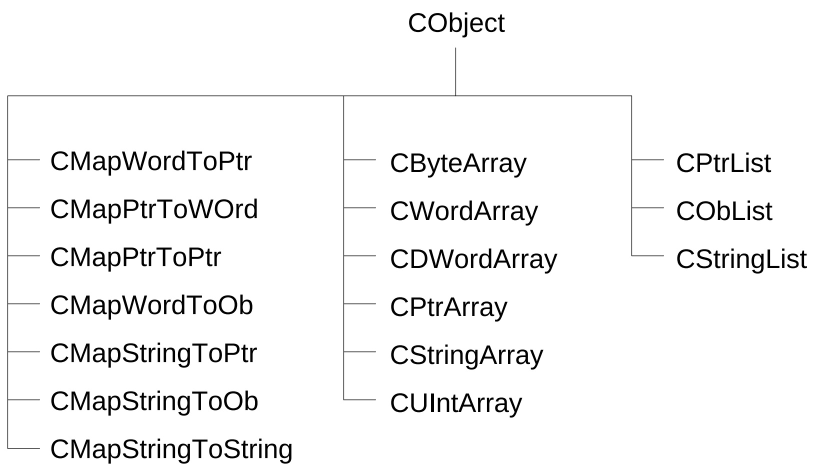


Figure 12 - The MFC container class hierarchy.

The figure is unequivocal: MFC only supports 3 basic types of containers. MFC doesn't support important container types such as Sets, Stacks, Queues, or Hash Tables. In fact, MFC doesn't even support the notion of containers for sortable objects, so you would have a hard time building a standard container to sort objects. MFC uses a different container class for each type, rather than use templates to do the same thing in a more object oriented style. What if you need to put objects of a type `MyType` into an MFC container? The only choice is to use one of the containers that take `void*` types, and perform explicit typecasting. This violates the principles of OOP, not only forcing the programmer to keep track of types, but also increasing the likelihood of bugs, since there is no inherent type safety when typecasting.

Another important issue with containers is ownership. A container is said to own the items it contains if it can delete those items when the container goes out of scope. Items in containers are often allocated from the heap, so it is important to delete them when they are no longer in use. If you put the same object in two different containers, you must make sure that only one of the containers owns the object, otherwise both will attempt to delete the same object. Ownership ensures that containers can clean up after themselves, preventing memory leaks. Ownership is handled in ObjectWindows by a class called `TShouldDelete`, which is a base class for all the containers.

MFC has no concept of ownership in its container class library. MFC containers don't delete their contents when going out of scope. Unless you explicitly iterate over a container and delete its elements, you may have memory leaks. Even worse, deleting an association in a Map doesn't delete the value object associated to the key -- you have to do that yourself. All these things add up in a real application, making it easy for an application to spring leaks that are hard to find.

Iteration

Quick Summary: ObjectWindows provides convenient invariant iterators for its containers. MFC still uses the clumsy, non-object-oriented first, next style of iteration.

One of the most basic operations used on containers is iteration. To iterate over a container is to visit each element in the container. Iteration usually entails executing a function on one or more of the items in a container.

Iteration is only part of the operation you perform on container objects. Often you want to know if a certain object is in a container, or locate a particular object, or find the first or last item that satisfies some condition. ObjectWindows 2.0 containers have member functions like `FirstThat()`, `LastThat`, `ForEach()and` `Find()` that do support these kinds of actions. MFC has no such functions.

Iteration with ObjectWindows
ObjectWindows 2.0 has an entire class hierarchy of container iterators, making it easy to iterate over a container. The notation is invariant for all the container types, so you can iterate over a Stack the same way you iterate over an Array. The following code shows a brief example of container iteration:

```
// create two short-hand types
typedef TSVectorImp<string> vector;
typedef TSVectorIterator<string> iterator;
vector names;

// add some names to the container
numbers.Add( string("Albert") );
numbers.Add( string("Victoria") );
// ...

// iterate over the container, and print out the contents
iterator iter(vector);
while (iter)
  os << iter++;
```

You create an iterator by passing it a reference to the container you wan to iterate over. The while loop checks the value of iterator, and when the value is 0, iteration stops. The `Current()` member function returns a reference to the next item in the container.

Iteration with MFC

MFC handles iteration the way C programmers used to iterate over a DOS directory. There is no concept of iterator class, and each container type has a function to get it first item, and the next item. The following code shows how iteration would work with a sample list container:

```
CNameList myList;  // assume CNameList contains CString pointers

// get the first item in the list
POSITION pos = myList.GetHeadPosition();
```

```
// iterate over all the list elements
while (pos) {
  CString* pName = (CString*) myList.GetNext(pos);
  // do something with the CString ...
}
```

Although the code is short, it has two problems: there are two functions to call during the course of iteration (`GetHeadPosition()` and `GetNext()`), and typecasting is necessary. But there is a much bigger problem, which isn't apparent in the code above: each container type has a different iteration method, involving different function calls. For example, to iterate over a Map, you would use the code:

```
CMapStringToString couples; // the map associates CStrings to CStrings

POSITION pos = couples.GetStartPosition();

while (pos) {
  CString* husband;
  CString* wife;
  couples.GetNextAssoc(pos, husband, wife);
  // use the two CString values...
}
```

Containers of type Array are iterated with yet another procedure, using C-style loops with array indices, rather than an iterator object. ObjectWindows 2.0 does allow you to iterate over arrays using the array index, but the preferred manner is through iterator objects. Using different procedures, as is required by MFC, means that changing a container type, such as from a list to an array, requires a great deal of recoding. The ObjectWindows containers, by contrast, are easy to interchange.

Streamable Objects

Quick Summary: ObjectWindows provides an object-oriented versus a C-style approach to persistent objects. The ObjectWindows approach is simple and uses the familiar C++ streams constructs. MFC's persistent objects are further limited to disk files only and use a non-standard serialization approach.

Persistence is an attribute of objects. Persistent objects have the ability to save and restore themselves from a stream (typically a file). Persistence can be applied to single objects, to groups of objects, or even to an entire application. ObjectWindows and MFC offer support for persistence, but while ObjectWindows takes an OOP approach, MFC takes a C approach. The following sections discuss persistence in more detail.

The ObjectWindows Approach

Quick Summary: ObjectWindows persistent objects act just like C++ streams making them familiar and standard. They also support in-memory streams not just disk files.

Streaming is based on a global stream manager and a class hierarchy of persistent file streams. To stream an object, you create an output persistent stream object, then insert the object into it, using standard C++ stream notation, like this:

```
TMyData myData;  // an arbitrary object that supports persistence

// create the output stream
ofpstream os("DATA.BIN");

// stream the object out
os << myData;
```

With this simple code, an object of type `myData` is streamed out. Behind the scenes, the stream manager is at work. The manager handles the non-trivial details of ensuring that pointers to a objects are streamed out correctly, so they can be restored when streaming objects back in. To stream an object in, the following code could be used:

```
// create a temporary "empty" object
TMyData myData(streamableInit);

// open the input stream
ifpstream is("DATA.BIN");

// stream the object in
is >> myData;
```

Objects that are streamable need to supply a bit of support to the stream manager, because the manager is responsible for coordinating the streaming of objects at a high level, and does not know how to stream each object at the bit level. To stream an object in and out, the stream manager makes calls to the object's nested streamer's `Read()` and `Write()` functions. These functions invoke the corresponding functions in their base class, then add code to read and write those data members that were not inherited. To completely support persistence, a class must only satisfy the following conditions:

1 - Be derived from class `TStreamableBase`.
2 - Its declaration must include the macro DECLARE_STREAMBLE.
3 - Its code must include the macro IMPLEMENT_STREAMBLE.
4 - It must have Streamer::`Read()` and `Write()` member functions.

ObjectWindows doesn't limit you to streaming with file streams. You can also create in-memory streams, of type `ipstream` or `opstream`, attached to `strstreambuf` buffers, and use them just like you would a file. You can stream an object out to an in-memory `opstream` object, and create multiple copies of the object by streaming it back in repeatedly from an `ipstream` stream object. The following code shows how to make a copy of a window through an in-memory stream:

```
  TWindow* window = new TWindow(NULL, "");
  strstreambuf buffer;

  // stream the object out
  opstream out(&buffer);
  out << window;

  // stream the object back in
  ipstream in(&buffer);
  TWindow* newWindow;
  in >> newWindow;
```

You aren't limited to `TWindow`-derived objects with streaming. Any class objects that satisfy some basic requirements can be streamed.

Streaming is a conceptually simple subject, and ObjectWindows makes it also simple to implement, allowing you to use familiar stream notation to stream objects in or out. ObjectWindows supports streaming to generalized streams, allowing you to stream objects to disk files, to modems, to pipes and in-memory streams.

MFC has only limited support for streams dealing with disk files.

The MFC Approach

Quick Summary: MFC's serialization is complex, doesn't user standard stream constructs and is limited to disk files.

MFC calls the process of streaming objects in and out serialization. MFC uses a technique that is much more complicated than that in ObjectWindows. To begin, persistence is not built into a separate class that you can derive a class from through multiple inheritance. All the support for persistence is built into class `CObject`, from which most MFC objects are derived. Persistent classes must include the macro DECLARE_SERIAL in their declarations and override the member function `CObject::Serialize()`. You also have to declare a default constructor. Internally, MFC handles the streaming in of objects through a relatively obscure class of type `CRunTimeClass`, which is supported through (what else?) a series of macros, and aids in identifying types at runtime.

MFC persistence doesn't work with actual streams. Instead, `CArchive` objects are used, which support file reading and writing. Curiously, `CArchive` is not derived from the class `CFile`, even though its purpose is to deal with files. When an object is streamed in or out, MFC calls the object's `Serialize()` member function, passing it a `CArchive` reference. You then call the member function `CArchive::IsStoring()` to determine whether to read or write the object to the archive, using code like this:

```
void CMyType::Serialize(CArchive& ar)
{
  if (ar.IsStoring() ) {
    // write the object to the archive
  }

  else {
    // read the object
  }
}
```

To initiate a serialization operation, you need to create a `CFile` object, open the file, create a `CArchive` object, then use the insertion and extraction operators. To stream out an object, you would use code like this:

```
// create the archive
CFile myFile;
myFile.Open("DATA.BIN", CFile::modeWrite);
CArchive ar(&myFile, CArchive::store);

// stream an object out
CMyType myType;       // assume CMyType is serializable
ar << myType;
```

The way files and archive objects are treated mirrors the way a C programmer would handle DOS files. Its seems redundant that you have to tell both the `CFile` and the `CArchive` object that you want to perform an output operation. To stream an object back in, you would use code like this:

```
// create the archive
CFile myFile;
myFile.Open("DATA.BIN", CFile::modeRead);
CArchive ar(&myFile, CArchive::load);

// stream an object in
CMyType myType;       // assume CMyType is serializable
ar >> myType;
```

Class `CFile` supports the bulk of file management. Microsoft completely ignored the existence of standard C++ streams when it designed MFC serialization. CFile is a big and monolithic class, making no use of inheritance, and providing a seemingly redundant wrapper around DOS files.

Clipboard Encapsulation

Quick Summary: ObjectWindows supplies clipboard support, whereas MFC does not.

ObjectWindows manages the clipboard through a class, MFC doesn't. ObjectWindows also has a clipboard viewer class, called `TClipboardViewer`, to let you browse the contents of the Windows clipboard.

Using ObjectWindow's class `TClipboard` is easy, because ObjectWindows covers most of the obscure details. For example, to copy a bitmap to the clipboard, you would use code like this:

```
void TMyindow::CmCopy()
{
  // create the clipboard object
  TClipboard clipboard;

  // create a TBitmap object
  TBitmap myBitmap(...);

  // move the bitmap to the clipboard
  if (clipboard.EmptyClipboard() )
    TBitmap(myBitmap).ToClipboard(clipboard);

  // we're done, destructor automatically closes
}
```

You can copy `TPalette` and `TDib` objects to the clipboard using the same notation. Pasting data from the clipboard is just as easy. The following code shows how you would paste a `TBitmap` into your application:

```
void TMyWindow::CmPaste()
{
  TClipboard clipboard;
  if (!clipboard) return;

  TBitmap* myBitmap;

  if (clipboard.IsClipboardFormatAvailable(CF_BITMAP))
    myBitmap = new TBitmap(TBitmap(clipboard) );
}
```

Pasting `TPalette`, `TDib, and TMetaFilePict` objects is just as easy.

Diagnostics and Debugging

Quick Summary: ObjectWindows provides a rich and extendible set of diagnostics, where MFC's diagnostics are limited. ObjectWindows lets you create multiple levels of diagnostic messages greatly aiding in the debugging process. These diagnostics can be interactively modified at run time, providing further flexibility during testing.

ObjectWindows 2.0 has a number of built-in features to aid in debugging and diagnosing problems in your code. There are essentially two levels at which ObjectWindows provides this support: through a series of macros, and through special diagnostic classes.

Macros have been used since ObjectWindows 1.0 to produce error messages. The macros PRECONDITION and CHECK support argument checking similar to the ASSERT macro. Error reporting is controlled through the symbol __DEBUG. By assigning values to __DEBUG, such as

```
#define __DEBUG 1
```

you can determine the amount of error checking you want. PRECONDITION should be used for conditions that must be true in order for the function to work correctly. Typically this is used to validate parameters. CHECK is used to be sure that internal computations make sense. When developing a library, both should be enabled. When the library is shipped to users the diagnostic version should have PRECONDITIONs enabled, since these will detect misuse of the library.

ObjectWindows 2.0 also has diagnostic message system for errors. Built-in diagnostic messages are divided into 6 categories, though programmers can add any number of additional categories:

1 - application-related message
2 - Window-related messages
3 - Window-message tracing
4 - GDI messages
5 - GDI orphan control messages
6 - Document View messages

GDI orphans are GDI objects that are created, but never destroyed. ObjectWindows has the ability to automatically destroy left-over GDI objects, reporting such occurrences in category 5 messages.

Control over the 6 diagnostic message categories is through the file ObjectWindows.INI, in which there are profile strings that deal with diagnostic levels. Each category can be enabled independently of the others and is assigned a diagnostic level, using an integer value between 0 and 255. The error messages are sent out through the Windows programs OX.SYS or DBWIN.EXE. Diagnostic messages are output with the macro TRACEX, like this:

```
// setup a diagnostic group for special conditions
DIAG_DEFINE_GROUP_INIT(ObjectWindows_INI, MyGroup, 1, 0);

void TMyWindow::SetupWindow()
{
  TRACEX(MyGroup, 2, "Entering TMyWindow::SetupWindow()");

  TMyWindow::SetupWindow();

  // do special processing
  TRACEX(MyGroup, 2, "Beginning special processing");
}
```

The macros TRACEX and WARNX write to output streams, and allow you to use stream inserters and manipulators to output messages, without using the old `printf` formatted string notation. With TRACEX you can have expressions like:

```
int value = 1;
float number = 3.14;
TRACEX(MyGroup, 2, "The value is " << value << "and the number is " << number);
```

The macro DIAG_DEFINE_GROUP sets up a new diagnostic group associated with file ObjectWindows.INI, and enable the group diagnostics. The group is associated with the file ObjectWindows.INI, and diagnostic error messages are output to OX.SYS or DBWIN.EXE with the TRACEX macro. The macro references the `MyGroup` diagnostic group, issuing an error message at diagnostic level 2.

ObjectWindows diagnostic groups are very flexible, not only because they let you organize different types of errors, but also because each type can be tailored specifically. But there's more: the diagnostic messages are a function of the diagnostic level of each category, and the level can be changed at runtime -- without recompiling any code. All you have to do is edit the ObjectWindows.INI file, adjusting the levels to your requirements, and then restart your application. ObjectWindows 2.0 ships with a small utility called DIAGXPRT.EXE that allows you to set the ObjectWindows diagnostic levels and display diagnostic output messages, obviating for the need of OX.SYS and DBWIN.EXE.

MFC has only minimal diagnostics support, with no groups or levels. Diagnostics messages are produced through the two macros ASSERT and TRACE macros. ASSERT messages are always enabled, TRACE message aren't. The TRACE macro also has 3 cousins: TRACE1, TRACE2 and TRACE3, each taking a formatting string (like printf does), and a certain number of additional parameters. No streams are used by the macros, forcing you to revert to C-style `printf` statements.

To enable MFC TRACE messages, there is whole procedure you must follow. First you define the _DEBUG identifier. Then you recompile all your code. Then you run the utility function TRACER.EXE, which prompts you for the categories of messages you wish to have diagnostic message for. Then you run your program. To turn off TRACE messages, you must undefine the _DEBUG macro, recompile your code, and run TRACER.EXE again.

MFC also supports a runtime tracing function, called `afxDump()`. To use this function, you put it a conditionally compiled section of code, like this:

```
#ifdef _DEBUG
afxDump("Dump this");
#endif
```

Of course you need to recompile your code to switch `afxDump()` messages on or off.

OLE 2.0 Encapsulation

Quick Summary: To help implement OLE 2.0 applications, MFC 2.5 encapsulates various interfaces of OLE 2.0 into existing or new MFC 2.5 classes. Developers are only required to fill in a handful of overridable methods in those classes through inheritance to obtain OLE 2.0 features. OWL 2.0 does not yet include an encapsulation of OLE 2.0.

The MFC 2.5 OLE 2.0 classes include support for:

- in-place activation and editing
- open editing
- drag-and-drop
- OLE automation support
- clipboard copy, paste and paste link of OLE objects
- container object verb menu initialization
- automatic registration of server applications
- managing lists of embedded objects
- managing lists of actively linked objects
- user interface classes for handling standard OLE operations
- classes for signaling exceptional conditions during OLE operations

MFC 2.5's implementation of OLE 2.0 encapsulation has some limitations. The classes violate the OLE reference counting model which allows destruction prior to removal of all references which can lead to application instability and crashes. There is no support for localization. There is no support for building legacy applications into OLE-enabled applications.

OLE 2.0 classes are currently under development. A key advantage to the OWL approach is the ease with which existing applications can be OLE 2.0 enabled.

OLE 2.0 Visual Editing Classes

Quick Summary: MFC 2.5 has added OLE 2.0 visual editing classes that help build visual editing servers, containers or both.

MFC 2.5's OLE 2.0 visual editing classes support in-place editing, fully opened editing, clipboard copy, paste and paste link of OLE objects, and drag and drop visual editing. The visual editing classes help build visual editing servers, containers, or both.

OLE 2.0 Automation Classes

Quick Summary: MFC 2.5 has added OLE 2.0 automation classes that help expose member functions and variables of C++ classes to other applications via OLE automation.

Base functionality of these classes is to expose member functions and member variables of C++ classes to other applications via OLE automation. New methods and properties can be added to application-specific classes that support OLE automation as well.

MFC 2.5's automation support only allows for single inheritance which is very restrictive. The automation support uses highly platform and compiler-dependent code which violates the OLE 2.0 guidelines for portability. Only classes derived from CCommandTarget can be automated. Most MFC classes derive from CCommandTarget which has been expanded to contain data members and virtual functions supporting automation. This automation baggage is now carried around even for classes not using OLE interfaces.

Database Encapsulation

Quick Summary: MFC 2.5 has added a thin layer of classes to support database application development.

New classes have been added to MFC to support the creation of database applications that allow the entering, displaying and updating of ODBC data sources.

Limitations in this database support include the lack of any table viewer control. Due to the use of the aforementioned limited DDX, it is not possible to have validation or computations during data transfers.

ObjectWindows 2.0 does not yet encapsulate database classes.

Accessing Data Sources

Quick Summary: MFC 2.5 now provides classes for accessing ODBC data sources.

MFC 2.5 provides classes for accessing ODBC data sources. These classes provide for automatically exchanging data between a C++ recordset object and columns of a table or query result. Dynasets as well as recordsets are supported. Database transactions such as commit and rollback are supported. Common database access functions such as adding, changing and deleting individual records are supported as well.

Database Forms

Quick Summary: MFC 2.5 has a CRecordView class that supports form design which is then used with data exchange to transfer data to and from some database record.

A new MFC 2.5 class called CRecordView has been added that supports database form design. Text fields and other controls can be added to turn a dialog template into a database form. The existing MFC 2.5 data exchange mechanisms — DDX and DFX — are then used to exchange data between this form and some record in the underlying database.

Conversion

OWL and MFC have both evolved since their initial versions. Borland's commitment is to make this transition as easy as possible. A conversion utility is provided called OWLCVT to convert source code for OWL 1 to OWL 2 that makes most changes automatically. This allowed the ObjectWindow's designers to create a major enhancement that is not just an incremental improvement over the previous version.

MFC-ObjectWindows conversion guide

This section shows how to convert MFC code to equivalent ObjectWindows code. Keep in mind that there are many ObjectWindows features that have no equivalent in MFC.

General Windows

Topic	MFC Code	ObjectWindows Code

Declaring response tables	`class CMyWnd: public CWnd`
	`{`
	`  // ...`
	`protected:`
	`  afx_msg int`
	`    OnCreate(LPCREATESTRUCT);`
	`  afx_msg void OnCmd1();`
	`  afx_msg void OnCmd2();`
	` DECLARE_MESSAGE_MAP()`
	`};`
		`class TMyWnd : public TWindow {`
		`public:`
		`  // ...`
		`  void EvKeyDown(UINT, UINT,`
		`               UINT);`
		`  void CmCommand1();`
		`  void CmCommand2();`
		`DECLARE_RESPONSE_TABLE(`
		`      TMyWnd);`
		`};`

Defining response tables	`BEGIN_MESSAGE_MAP(CMyWnd, CWnd)`
	`  ON_WM_CREATE()`
	`  ON_COMMAND(IDM_1, OnCmd1)`
	`  ON_COMMAND(IDM_2, OnCmd2)`
	`END_MESSAGE_MAP()`
		`DEFINE_RESPONSE_TABLE1(`
		`   TMyWnd, TWindow)`
		` EV_WM_KEYDOWN,`
		` EV_COMMAND(CM_1, CmCmd1),`
		`  EV_COMMAND(CM_2, CmCmd2),`
		`END_RESPONSE_TABLE;`

Creating a Window	`CMyWnd* myWnd = new CMyWnd;`
	`myWnd->Create(...);`
		`TMyWindow* w = new`
		`TMyWindow(...);`
		`w->Create();`
		`//children are autocreated by`
		`parent`
		`//MainWindow is autocreated by`
		`app`

Creating an MDI frame window	`class CMyWnd :`
	` public CMDIFrameWnd {...};`
	
	`class CMyApp :`
	`    public CWinApp {`
	`public:`
	`// ...`
	` virtual BOOL InitInstance() {`
	`   CMyWnd* w = new CMyWnd;`
	`   if (!w)->LoadFrame(IDRES) )`
	`     return FALSE;`
	`   w->ShowWindow(m_nCmdShow);`
	`   w->UpdateWindow();`
	`   m_pMainWnd = w;`
	`   return TRUE;`
	`  }`
	`};`
		`class TMyWnd :`
		`  public TMDIFrame {...};`
		
		`class TMDIFileApp :`
		`    public TApplication {`
		`public:`
		` void InitMainWindow() {`
		`  Frame = new TMDIFrame(..);`
		`  Frame->Attr.AccelTable =`
		`         IDRES;`
		`  Frame->SetMenuDescr(...);`
		`  MainWindow = Frame;`
		` }`
		`};`

Creating an SDI frame window	class CMyWnd :
	public CFrameWnd {...};
	
	class CMyApp : public CWinApp {
	public:
	// ...
	virtual BOOL InitInstance(){
	CMyWnd* w = new CMyWnd;
	if (!w->LoadFrame(IDRES))
	return FALSE;
	w->ShowWindow(m_nCmdShow);
	w->UpdateWindow();
	m_pMainWnd = w;
	return TRUE;
	}
	};
		class TMyWnd :
		public TFrameWindow {...};
		
		class TSDIFileApp : public
		TApplication {
		public:
		void InitMainWindow() {
		Frame = new
		TFrameWindow(...);
		Frame->Attr.AccelTable =
		IDRES;
		Frame->SetMenuDescr(...);
		MainWindow = Frame;
		}
		};

Creating documents	class CMyDoc :
	public CDocument {..};
		class TMyDoc : public
		TDocument {...};

Creating Views	class CMyView : public CView {..};	class TMyView :
		public TView
		{...};

Creating Doc/View templates	class CMyWnd : public
	CMDIChildWnd {..};
	
	class CMyApp : public CWinApp {
	public:
	// ...
	virtual BOOL InitInstance() {
	AddDocTemplate(new
	CMultiDocTemplate(IDRES,
	RUNTIME_CLASS(CMyDoc),
	RUNTIME_CLASS(CMyWnd),
	RUNTIME_CLASS(CMyView)));
	
	CMyWnd* w = new CMyWnd;
	if (!w->LoadFrame(IDRES))
	return FALSE;
	w->ShowWindow(m_nCmdShow);
	w->UpdateWindow();
	m_pMainWnd = w;
	}
	};
		DEFINE_DOC_TEMPLATE_CLASS(
		TMyDocument, TMyView,
		MyTemplate);
		MyTemplate btpl("My files",
		"*.txt", 0,
		"TXT", dtAutoDelete);
		
		class TMyApp : public
		TApplication {
		public:
		// ...
		void InitMainWindow() {
		SetDocManager(new
		TDocManager(dmSDI | dmMenu));
		}
		};

Adding a toolbar	class CMyWnd :
	public CFrameWnd {
	// ...
	protected:
	CToolBar m_Bar;
	};
	
	int CMyWnd::OnCreate(
	LPCREATESTRUCT lpcs)
	{
	if (CFrameWnd::OnCreate(
	lpcs) == -1)
	return -1;
	if (!m_Bar.Create(this)
	|| !m_Bar.LoadBitmap(IDRES))
	return -1;
	return 0;
	}
		TControlBar* cb =
		new TToolBox(0);
		cb->Insert(*new
		TButtonGadget(CM_TOOL1,
		CM_TOOL1,
		TButtonGadget::Exclusive,
		TRUE,
		TButtonGadget::Down));
		cb->Insert(*new
		TButtonGadget(CM_TOOL2,
		CM_TOOL2,
		TButtonGadget::Exclusive,
		TRUE));
		
		frame->Insert(cb, Top);
		
		}
		};

Adding a status bar	class CMyWnd: public CFrameWnd
	{
	// ...
	protected:
	  CStatusBar m_Bar;
	};
	
	int CMyWnd::OnCreate(
	 LPCREATESTRUCT lpcs)
	{
	  if (CFrameWnd::OnCreate(
	    lpcs) == -1)
	    return -1;
	  if (!m_Bar.Create(this) ||
	      !m_Bar.SetIndicators(
	       indicators,
	        sizeof(indicators) /
	          sizeof(UINT)))
	    return -1;
	  return 0;
	}
		class TMyApp : public
		TApplication {
		public:
		 void InitMainWindow() {
		
		  TStatusBar* sb =
		    new TStatusBar(0,
		     TGadget::Recessed,
		     TStatusBar::CapsLock |
		     TStatusBar::NumLock |
		     TStatusBar::Overtype);
		  Frame->Insert(*sb,
		   TDecoratedFrame::Bottom);
		 }
		};

Iterating over child windows	void CMyWnd::Iterate()
	{
	 for (CWnd* w = GetTopWindow();
	      w != NULL;
	      w = w->GetNextWindow()) {
	   // use child window 'w'
	 }
	}
		static void f(TWindow* w,
		              void*)
		{...do something with 'w'}
		
		void TMyWindow::g()
		{ ForEach(f); }

Locating a child window	CWnd* CMyWnd::FindChild()
	{
	 for (CWnd* w = GetTopWindow();
	      w != NULL;
	      w = w->GetNextWindow()) {
	  // see if child window found
	  if (w is the right window)
	    return w;
	 }
	 return 0;
	}
	
	...useChild()
	{
	  CWindowfirst = FindChild;
	...
	}
		static BOOL f(TWindow* win,
		              void*)
		{
		  return(win satisfies some
		      condition);
		}
		
		void TMyWindow::useChild()
		{
		  TWindow* first =
		      FirstThat(f);
		
		}

Finding the active MDI child window	class CMyWnd: public CMDIFrameWnd
	{
	// ...
	public:
	
	  void f() {
	    CMDIChildWnd* w =
	            MDIGetActive();
	    if (!w) return;
	    // use w ...
	  }
	};
		class TMDIFileApp : public
		TApplication {
		public:
		// ...
		  MDIClient* Client;
		
		protected:
		  void f() {
		  TMDIChild* w =
		Client->GetActiveMDIChild();
		  if (!w) return;
		  // use w..
		}
		};

Dialog boxes and Child Controls

Topic	MFC Code	ObjectWindows Code

Creating a modal dialog box	`CDialog dlg(IDD_ABOUTBOX);`
	`dlg.DoModal();`
		`TDialog(this,  ID).Execute();`

Creating a modeless dialog box	`void CMyWindow::Tools() {`
	`  CDialog dlg(IDD_TOOLS);`
	`  dlg.Create(this);`
	`}`
		`void TMyWindow::Tools() {`
		`  TDialog(this,ID).Create();`
		`}`

Initializing the controls in a dialog box	`class CMyDlg : public CDialog {`
	`public:`
	`// ...`
	`  //{{AFX_DATA(CMyDlg)`
	`    int m_Value1;`
	`    int m_Value2;`
	`  //}}AFX_DATA`
	`protected:`
	`  DECLARE_MESSAGE_MAP()`
	`};`
	
	`void CMyDlg::DoDataExchange(`
	`  CDataExchange* pDX)`
	`{`
	` CDialog::DoDataExchange(pDX);`
	` DDX_Text(pDX, IDC_EDIT1,`
	`          m_Value1);`
	` DDV_MinMaxInt(pDX, m_Value1,`
	`               -10, 20);`
	` DDX_Text(pDX, IDC_EDIT2,`
	`          m_Value2);`
	` DDV_MinMaxInt(pDX, m_Value2,`
	`               0, 100);`
	`}`
		`struct {`
		`  // transfer buffer`
		`  // ...`
		`} Buffer`
		
		`class TMyDlg : public TDialog {`
		`public:`
		`  // ...`
		`  TMyDlg(...) {`
		`  // .. create controls`
		`  SetTransferBuffer(&Buffer);`
		` }`
		`};`

Reading the controls in a dialog box	`Same as above.`	`Same as above`
Setting a dialog's child control	`SetDlgItemInt(nID, value);`	`TEdit(...).SetText("this");`

Reading a dialog's child control	`CEdit& edittedData =`
	`  *(CEdit*) GetDlgItem(nID);`
		`TEdit* e = new TEdit(...);`
		`char name [80];`
		`e->GetText(name, 80);`

Validating Data	`DDV functions. See code under`
	`"Initializing the control in a`
	`dialog box".`
		`TEdit* e = new TEdit(...);`
		`e->SetValidator(new`
		` TPXPictureValidator("&&&");`

Bitmapped buttons	`class CMyDlg : public CDialog {`
	`public:`
	`  enum {IDD = IDD_BITMAPDLG};`
	`  CMyDlg();`
	`// ...`
	`protected:`
	`  CBitmapButton button1;`
	`};`
	
	`CMyDlg::CMyDlg()`
	`      : CDialog(CMyDlg::IDD)`
	`{`
	`  if (!button1.LoadBitmaps(`
	`     "Up", "Down", "Focus") ) {`
	`   TRACE("Problem!");`
	`   AfxThrowResourceException();`
	`  }`
	`}`
		`no code necessary`

GDI Operations

Topic	MFC Code	ObjectWindows Code
Creating a pen	`CPen pen; pen.CreatePen(PS_SOLID, 1, RGB(0,0,0));`	`TPen pen(TColor(0, 0, 0) );` or `TPen pen(TColor::Black);`
Drawing a line	`void CMyWnd::Line(CDC& dc) { CPen pen; pen.CreatePen(PS_SOLID, 1, RGB(0,0,0) ); CPen* pOldPen = dc.SelectObject(&pen); dc.MoveTo(10, 10); dc.LineTo(20, 30); dc.SelectObject(pOldPen); }`	`void TMyWnd::Line(TDC& dc) { TPen pen(TColor(0, 0, 0) ); dc.SelectObject(pen); dc.MoveTo(0, 100); dc.LineTo(100, 20); }`
Painting with a brush	`void CMyWnd::Box(CDC& dc) { CBrush brush(RGB(0, 0, 0) ); CBrush* pOldBrush = pDC->SelectObject(&brush); dc.Rectangle(30, 30, 100, 100); dc.SelectObject(pOldBrush); }`	`void TMyWnd::Box(TDC& dc) { dc.SelectObject( TBrush(Color:Black)); dc.Rectangle(0,20,30,400); }`
Creating fonts	`void CMyWnd::Font(CDC& dc) { LOGFONT lf; memset(&lf, 0, sizeof(lf)); lf.lfHeight = 20; lf.lfWeight = FW_BOLD; strcpy(lf.lfFaceName, "Arial"); CFont font; font.CreateFontIndirect(&lf)); }`	`void TMyWnd::Font(TDC& dc) { TFont font("Arial", 20, FW_BOLD); }`
Drawing text	`void CMyWnd::DrawText(CDC& dc) { CRect rect(20, 30, 100, 200); dc.DrawText("Text", -1, rect, DT_CENTER); }`	`void TMyWnd::Text(TDC& dc) { dc.DrawText("Text", -1, TRect(0, 0, 10, 200), DT_CENTER); }`
Creating bitmaps	`CBitmap bm; bm.LoadBitmap("MYBITMAP");`	`TBitmap* bm = new TBitmap( *GetModule(), "ID");`
Displaying bitmaps	`void CMyWnd::DrawBM(CDC& dc) { CBitmap bm; bm.Create("MYBITMAP"); CBitmap* pbmOld; CDC dcMem; dcMem.CreateCompatibleDC(&dc); pbmOld = dcMem.SelectObject(&bm); dc.BitBlt(100, 100, 50, 50, &dcMem, 0, 0, SRCCOPY); dcMem.SelectObject(pbmOld); dcMem.DeleteDC(); }`	`void TMyWnd::Draw(TDC& dc) { TBitmap* bm = new TBitmap(*GetModule(), "ID"); TMemoryDC memoryDC(dc); memoryDC.SelectObject( *Bitmap); TRect rect(0, 0, 40, 40); dc.BitBlt(rect, memoryDC, TPoint(0,0), SRCCOPY); }`

Containers

Topic	MFC Code	ObjectWindows Code

Creating an array	`CByteArray myArray;`	`TIArrayAsVector<int>`
		`myArray(5,0,5);;`

Copying an array	`CByteArray myArray;    // array to be copied`
	`CByteArray copyArray;  // array copied into`
	
	`for (int i=0; i < myArray.GetSize(); i++)`
	`copyArray [i] = myArray [i];`
		`TVectorImp<int> myArray;`
		`TVectorImp<int> copyArray;`
		`for (int i = 0;`
		`i < myArray.Count();`
		`i++)`
		`copyArray [i] =`
		`myArray [i];`

Adding elements to an array	`CByteArray myA;`
	`BYTE value = 2;`
	`myA.Add(value);`
		`TArrayAsVector<int> myA(5,0,5);`
		`int value = 5;`
		`myA.Add(value);`

Removing elements from an array	`CByteArray myArray;`
	`myArray.RemoveAt(10);`
		`TArrayAsVector<int>`
		`myArray(5,0,5);`
		`myArray.Detach(3);`

Searching an array for an item	`CByteArray myArray;`
	`int FindItem(BYTE value)`
	`{`
	`for (int i=0; i < myArray.GetSize(); i++) {`
	`if (myArray [i] == value)`
	`return i;`
	`}`
	`return -1;`
	`}`
		`TArrayAsVector<int>`
		`myArray(5,0,5);`
		`int value = 5;`
		`int index =`
		`myArray.Find(value);`

Deleting the items in an array	`CStringArray myArray;`
	
	`void DeleteArray()`
	`{`
	`for (int i=0; i < myArray.GetSize(); i++)`
	`delete myArray [i];`
	`myArray.RemoveAll();`
	`}`
		`TArrayAsVector<int>`
		`myArray(5,0,5);`
		`myArray.Flush();`

Creating a list	`CStringList myList;`	`TListImp<string> myList();`

Copying a list	`CStringList myList;  // list`
	`// to copy`
	`CStringList copyList; // list`
	`// copied to`
	
	`void CopyList()`
	`{`
	`POSITION pos = myList.GetHeadPosition();`
	`while (pos)`
	`copyList.AddTail(`
	`myList.GetNext(pos) );`
	`}`
		`TListImp<string> myList;`
		`TListImp<string> copyList;`
		
		`static void DoCopy(`
		`string& s, void*)`
		`{copyList.Add(s);}`
		
		`void f()`
		`{ myList.ForEach(DoCopy, 0); }`

Adding items to a list	`CStringList myList;`
	`myList.AddTail("Hello");`
	`myList.AddHead("Good-bye");`
		`TListImp<string> myList;`
		`myList.Add("Test");`

Removing items from a list	`CStringList myList;`
	
	`void RemoveItem(CString& target)`
	`{`
	`  POSITION pos =`
	`myList.GetHeadPosition();`
	`  while (pos) {`
	`        CString& str =`
	`myList.GetNext(pos);`
	`        if (str == target)`
	
	`       myList.RemoveAt(pos);`
	`              delete str;`
	`  }`
	`}`
		`TListImp<string> myList;`
		`myList.Detach("Test");`

Searching a list for an item	`CStringList myList;`
	
	`BOOL HasString(CString& target)`
	`{`
	`  POSITION pos =`
	`myList.GetHeadPosition();`
	`  while (pos) {`
	`        CString& str =`
	`myList.GetNext(pos);`
	`        if (str == target)`
	`              return TRUE;`
	`  }`
	`  return FALSE;`
	`}`
		`TListImp<string> myList;`
		`if (myList.Find("Test") ) {`
		`  // the item was found...`
		`}`

Deleting all the items in a list	`CStringList myList;`
	
	`void DeleteList()`
	`{`
	`  POSITION pos =`
	`myList.GetHeadPosition();`
	`  while (pos)`
	`        delete`
	`myList.GetNext(pos);`
	`  myList.RemoveAll();`
	`}`
		`TListImp<string> myList;`
		`myList.Flush();`

Creating a dictionary	`CMapStringToOb myMap;`	`// create a hashable class`
		`class HashString : public`
		`string {`
		`public:`
		` HashString() : string() {}`
		` HashString(const char* s) :`
		`     string(s) {}`
		` unsigned HashValue() const`
		`{ return hash(); }`
		`};`
		
		`void f()`
		`{   typedef`
		` TDDAssociation<HashString,`
		`HashString> symbol;`
		`        TDictionaryAsHashTable`
		`<symbol> Dictionary;`

Copying a dictionary	`CMapStringToOb myMap;  // map to`
	`copy`
	`CMapStringToOb myCopy; // map`
	`copied to`
	
	`POSITION pos =`
	`myMap.GetStartPosition();`
	`while (pos) {`
	`  CString string;`
	`  CObject* pObject;`
	`  myMap.GetNextAssoc(pos, string,`
	`pObject);`
	`  copyMap.SetAt(string, pObject);`
	`}`
		`typedef TDDAssociation`
		`<HashString, HashString>`
		`   symbol;`
		`typedef`
		`  TDictionaryAsHashTable`
		`  <symbol> dictionary;`
		`dictionary myTable;`
		`dictionary copyTable;`
		
		`static void DoCopy(`
		`  symbol& s, void*)`
		`{ copyTable.Add(s); }`
		
		`void f()`
		`{ myTable.ForEach(`
		`   DoCopy, 0);`
		`}`

Adding items to a dictionary	`CMapStringToOb myMap;`
	`CString string;`
	`CObject* pObject;`
	`myMap.SetAt(string, pObject);`
		`Table.Add(Symbol("K", "U"));`

Removing items from a dictionary	`CMapStringToOb myMap;`
	
	`void RemoveItem(CString& str)`
	`{`
	`  CObject* pObject;`
	`  if (!myMap.Lookup(str, &pObject)`
	`)`
	`         return;`
	`  myMap.RemoveKey(str);`
	`  delete str;`
	`  delete *pObject;`
	`}`
		`Table.Detach(Symbol("K", "U"));`

Searching a dictionary for an item	`CMapStringToOb myMap;`
	
	`BOOL HasItem(CString& str)`
	`{`
	`  CObject* pObj;`
	`  return myMap.Lookup(str, &pObj)`
	`;`
	`}`
		`symbol* r =`
		`Table.Find(Symbol("K", "U"));`
		`if (r) {`
		`   // found...`
		`}`

Deleting all the items in a dictionary	`CMapStringToOb myMap;`
	
	`POSITION pos =`
	`myMap.GetStartPosition();`
	`while (pos) {`
	`  CString string;`
	`  CObject* pObject;`
	`  myMap.GetNextAssoc(pos, string,`
	`pObject);`
	`  delete pObject;`
	`}`
	`myMap.RemoveAll();`
		`dictionary myTable;`
		`myTable.Flush();`

Persistence

Topic	MFC Code	ObjectWindows Code

Creating an input stream	`CFile myFile;`
	`myFile.Open("T.TST",`
	`CFile::modeRead);`
	`CArchive myArchive(&myFile,`
	`CArchive::load);`
		`ifpstream is("T.TST");`

Streaming an object in	`int i;`
	`myArchive >> i;`
		`int i;`
		`is >> i;`

Creating an output stream	`CFile myFile;`
	`myFile.Open("T.TST",`
	`CFile::modeWrite);`
	`CArchive myArchive(&myFile,`
	`CArchive::store);`
		`ofpstream os("T.TST");`

Streaming an object out	`int i;`
	`myArchive << i;`
		`int i;`
		`os << i;`

Conclusion

Both ObjectWindows and MFC are extensive application frameworks that make Windows programming easier. ObjectWindows is very object-oriented system, utilizing advanced C++ features such as multiple inheritance, class templates and exceptions handling. ObjectWindows is programmaticaly a very safe class library to program with. Exception handling eliminates the need to constantly check for successful resource allocation and GDI operations, allowing you to concentrate on what your application does than on recovering from Windows API failures. MFC is very little object-oriented, utilizing C language constructs pervasively. There are numerous pitfalls in MFC programming that can be particularly difficult to debug. For example, MFC exception handling does not properly destroy objects in the course of stack unwinding during exception handling. This is a fiendish trap, that MFC programmers are destined to be caught in. There are many other problems with MFC, such as lack of GDI orphan control and standard run-time type identification that collectively make MFC programming much more difficult and much less effective than ObjectWindows programming.

The bottom line is productivity. ObjectWindows provides a much higher degree of abstraction from Windows details, allowing you to build complex system quickly and with little coding. Containers are one area in which ObjectWindows is spectacularly better the MFC, but the list of ObjectWindows strengths is long. ObjectWindows is better than MFC in handling persistence, exceptions, GDI, printing, tool palettes, dialog box child controls and debugging diagnostics -- to name a few. ObjectWindows is a mature C++ product, ready to take on even the toughest assignments.