Microsoft® Foundation Class Library 2.0: C++ Application Framework for Microsoft Windows™ and Microsoft Windows NT™

White Paper

June 1993

For more information contact:

Microsoft Corporation
Eric Lang, (206) 882-8080

Waggener Edstrom
Martin Middlewood, (503) 245-0905

Table of Contents

Introduction

The Microsoft® Foundation Class Library 2.0 (MFC 2.0[1]) is a robust C++ application framework designed for writing applications in C++ for the Microsoft Windows™ and Windows NT™ operating systems. Built upon the core set of functionality in the MFC 1.0 application framework, MFC 2.0 adds an architecture and set of prebuilt components that make it possible to write professional, full-featured applications for Windows in a fraction of the time it would take using C and the SDK or other application frameworks. MFC 2.0 offers a high level of abstraction that lets you focus on the details specific to your application, while allowing its classes to be customized and extended. Like its predecessor, MFC 2.0 also allows access to the native Windows application programming interfaces (APIs) for maximum flexibility and power.

MFC 2.0 makes programming in Windows and C++ a much more productive endeavor. Through its carefully designed architecture, MFC 2.0 provides substantial programming power in an intuitive and uncomplicated package.

MFC 2.0 offers more than 100 reusable C++ classes, including general-purpose classes that support the
non-graphical portion of your application, classes for the core Windows graphical user interface (GUI) features, architectural classes to help you organize and structure your application programming and high-level abstractions and canned functionality that provide major building blocks. Rigorous tuning and optimizing of the source code has resulted in very fast execution speeds and small executable sizes that are comparable to C.

The Microsoft Application Framework (AFX) group received a great deal of feedback on MFC 1.0 that was based upon many real-world projects and shipping applications. Microsoft received critical acclaim from both developers and the press, and were generally recognized as the "standard application framework for Windows."[2] AFX also received hundreds of suggestions and feature requests through CompuServe®, our MSAFX e-mail address and the dozens of conferences and shows where we sent AFX team members to hear from users who had developed mission-critical applications with MFC 1.0. MFC 1.0 was also provided as part of the Windows NT PDK program, which provided additional feedback from the beta developers of Windows NT . MFC 2.0 implemented many of these requested features, and this release of the application framework truly represents the collective wisdom of our user community. The ability to incorporate user feedback is important to the AFX group and will continue to be as we improve MFC for future releases. We also practice what we preach, and the AFX team at Microsoft has been using MFC 2.0 for our products as well. As a case in point, App Studio, AppWizard and ClassWizard were all built using MFC 2.0.

The Microsoft Foundation Classes are appropriately named because they represent the foundation of a class architecture that is constantly evolving to bring developers the best support for the

[1]In this document MFC 1.0 refers to features and metrics specific to the Microsoft Foundation Classes 1.0. MFC 2.0 refers to features and metrics specific to the Microsoft Foundation Classes 2.0. MFC refers to concepts, architecture and APIs common to all versions of the Microsoft Foundation Classes.

[2]A Preview of Microsoft C/C++ 7 and the Microsoft Foundation Classes for Windows, *Microsoft Systems Journal*, March/April 1992, Richard Hale Shaw.

- more -

Windows operating system. The class hierarchy, as well as the functionality it encapsulates, has been designed for scalability. Applications written for MFC 1.0 are compatible with MFC 2.0.

MFC applications written for 16 bits using Visual C++ for Windows 3.1 can be recompiled with very minor[1] changes with the Microsoft Visual C++ Development System, 32-bit Edition version 1.0 to run as a full 32-bit program under Windows NT or Windows 3.1 using Win32s™.

[1]The modifications required are due to converting any 16-bit specific code in your application. The application framework code does not rely on 16-bit or 32-bit implementation details.

- more -

This paper provides a technical description of the major features of the Microsoft Foundation Classes version 2.0. MFC 2.0 is integrated with the Microsoft Visual C++™ Development System, 32-bit Edition version 1.0 and, in particular, with App Studio, ClassWizard and AppWizard. Many features of MFC 2.0 are made simpler, less error-prone and more streamlined by using these tools and the Visual C++ integrated development system. The application framework, however, does not require any of the tools; all of the features are accessible to users choosing not to take advantage of the entire family of tools.

- more -

Distinguished Benefits

- **C++ Windows API:** MFC 1.0 introduced a standard application framework interface for programmers using C++ to develop applications for Windows. MFC 2.0 continues this standard. MFC 1.0 applications need only be recompiled to begin taking advantage of MFC 2.0 features.[1]

MFC follows a simple set of conventions that leverage the Windows API; those already familiar with the Windows API are able to look at MFC code and have a clear understanding of the concepts involved. To those that are new to Windows-based programming, this leveraging of the Windows API helps programmers take advantage of the many sources of information available for learning the concepts behind Windows. For example, sample code and concepts from Charles Petzold's Programming Windows can be easily translated to the MFC API. In addition to being able to reuse concepts, MFC lets you easily reuse low-level C code in MFC applications because of these conventions.

MFC provides a comprehensive object-oriented encapsulation of most Windows API functions. Its classes provide support for application start-up, window creation, multiple document interface windows, menus, dialog boxes, controls, list boxes, graphical primitives and so forth.

- **High-level Abstractions:** Many users commented that MFC 1.0 did not contain abstractions that make it easier to write Windows-based applications.[2] MFC 2.0 addresses this by providing high-level abstractions that let programmers concentrate on the real task of the application being written, rather than on mundane Windows-based tasks such as implementing a toolbar. MFC 2.0 provides classes that encapsulate thousands of lines of robust and optimized Windows C++ code. For example, the Print Preview feature, which requires no additional code on the programmer's part, consists of more than 2,000 lines of MFC 2.0 framework code. A programmer wishing to add a toolbar to an application needs to add less than 10 lines of code calling three APIs to exploit more than 1500 lines of MFC 2.0 framework code.

- **Canned Functionality:** MFC 2.0 includes a large amount of prebuilt, canned functionality. The primary benefit of an application framework is the use of professional code written by experienced developers. The C++ programming language, using inheritance, encapsulation and polymorphism, makes it much easier

[1]MFC 1.0 consisted of approximately 1,800 APIs (MFC 2.0 is nearly 3,000 APIs), and of those only about 20 changed in a manner that would result in a compile time error. A detailed techical note, TN 19, describes the required modifcations.

[2]Some considered MFC 1.0 to be less abstract than Borland's OWL. A comparison of the class hierarchies shows the two application frameworks have substantially similar structure. An analysis of the APIs demonstrates that both class libraries contained a large number of simple wrapper functions. MFC 1.0 contained significant abstraction in the areas of graphics and OLE. OWL, for example, contained no abstraction above the Windows API for graphical output, which means that all graphic calls were written as calls to C/SDK functions. For more details see Sizing Up Application Frameworks and Class Libraries, *Dr. Dobb's Journal*, October 1992.

to take advantage of reusable components. MFC 2.0 leverages these features of C++ to provide a body of expertly written, and easily customizable, Windows-based functionality. For example, the standard MFC 2.0 implementation of the File Open command automatically handles all the steps necessary to prompt the user, open a file, read the data, create a window, draw the contents and so on. Programmers only need to provide an implementation of application-specific file I/O and drawing code, and MFC 2.0 does the rest. Perhaps the most important part of MFC 2.0's prebuilt functionality is that it represents an evolved and standard implementation of the recommended techniques for solving common Windows-based programming problems.

- Small and Fast Executables: Research shows that the one of the most important concerns of Windows developers is the need for small and fast executables. Because MFC 1.0 was modeled so closely after the Windows API, the size of an MFC 1.0 C++ application was only slightly larger than its equivalent C/SDK application. MFC 2.0 is still the smallest and fastest application framework available for Windows. MFC 2.0 applications are only slightly larger than MFC 1.0 applications (if you recompile MFC 1.0 code and use MFC 2.0 libraries). If an application uses the high-level features and canned functionality of MFC 2.0, it is comparable in size to any other implementation of those same features.

MFC 2.0 Components

MFC 2.0 encompasses most of the functionality available through the Windows API. Since the nearly 60,000 lines of standard C++ source code for the application framework are included with the product, developers can use the framework in its original form or fully customize it for their own purposes. This source code serves as an example of robust and professional C++ code for Windows. In addition, programmers are able to use this code to learn new implementation techniques and look "under the hood" of MFC. Consistent naming conventions and coding style, along with standalone documentation, make the learning curve minimal. A tutorial is included that steps programmers through development of a substantial C++/MFC program for Windows that incorporates most application framework features. In addition, more than 20 complete sample applications are included that demonstrate the most common uses (and many advanced uses) of the framework.

The application framework divides logically into four components: general-purpose classes, Windows API classes, application architecture classes and high-level abstractions. General-purpose classes assist the programmer with the lower-level coding tasks such as file manipulation, string processing and building block data structures. Windows API classes provide developers with a complete object-oriented implementation of the GUI portions of the Windows API. The general-purpose and Windows classes were present in MFC 1.0, and have been enhanced for MFC 2.0. A complete class hierarchy diagram is included in this document.

- more -

MFC 2.0 introduces application architecture and high-level abstractions. The application architecture classes provide core implementations that are common among standard Windows-based applications, such as documents and views, printing and command processing. The high-level abstractions give programmers optimized and reusable building blocks to speed the development of sophisticated applications. The following sections explore each of these areas in more detail.

Architecture Classes

A key benefit of an application framework is that it not only provides a large body of prebuilt functionality, but offers an architecture in which to add your own functionality. When you need to implement a feature in your application, an elegant architecture provides you with a logical and obvious location to add your application-specific code. For example, when implementing the File Save command, the application framework should have an obvious technique (a member function, a hook, etc.) to add this functionality. It is not enough, however, to point programmers to the right place, since there is often no single right place, or the application framework could not foresee the exact situation. MFC 2.0 addresses these issues with a group of tightly integrated classes collectively known as the *application architecture* classes. These classes provide support for the important areas common to many parts of your application such as commands, documents, views, printing, Online help and dialog processing.

Commands

Menu items, keyboard accelerators and toolbar buttons are the most common sources of commands in an application. A command is an instruction to your program to perform a certain action. Unlike a procedure or function call, a command is a message that is routed to various command targets that may carry out the instruction. Command targets are objects derived from the CCmdTarget class, and include documents, views, windows and the application itself.

The command architecture ensures that any user-interface action, such as clicking a toolbar button or selecting a menu item, will route the command to the appropriate handler. Command routing can also be used to update the visual state of menu items or toolbar buttons. For example, the Edit Cut command might have both a menu item and a toolbar button that can be enabled or disabled. Using the command architecture, it is easy to maintain the visual enabled/disabled state of both the menu item and the toolbar button with a single line of code in a single location.

Another integral part of the MFC architecture is message maps, which were introduced in MFC 1.0 and have been extended in MFC 2.0. A message map provides a typesafe mechanism for directing any windows message, control notification or command to a C++ member function in the appropriate class. Each command target has a message map, and it contains entries that map each command ID defined in App Studio to a C++ member function. Since there can be a large number of commands (as well as Windows messages and notifications) handled by each command target, ClassWizard is usually invoked to manage the creation and maintenance of message maps and message handler functions. ClassWizard can be invoked from either App Studio or the Visual WorkBench.

- more -

Documents and Views

The document/view architecture of MFC 2.0 is the basis for managing the storage and display of application-specific data. The CDocument class provides support for managing your application's data, and an application will typically derive a new class from CDocument for each document type. A CDocument class will add member variables and member functions that allow you to manage application-specific data. The key feature of a document class is its ability to save a document object to a file for later use. The programmer's responsibility is to override the serialize member function, which saves and loads application-specific data to and from storage. By implementing this function, MFC 2.0 automatically supports high-level commands such as File New, File Save, File Save As and File Open. MFC 2.0 does all the work of displaying a dialog to gather information from the user and managing the disk file. Although most documents are typically associated with disk files, the CDocument architecture is flexible enough to allow manipulation of data stored in other ways, such as in a database file, or to allow data manipulation of data without any kind of stored representation.

Each document in a running application is attached to one or more views on that document. Views control the graphical display of your application's data on the screen. Programmers will typically derive a class from the MFC 2.0 CView class, which is itself derived from CWnd, and then implement the display code. A view represents the main area of a window on the screen, and is a simple child window that you can manipulate with CWnd member functions. This usually involves implementation of the OnDraw member function and writing the code that displays the currently visible data to the user. The OnDraw function replaces the low-level OnPaint handler of MFC 1.0 with a high-level abstraction. After implementing OnDraw, your program automatically supports printing and Print Preview. The CView-derived class is usually the best place to handle most of the commands and window messages that graphically manipulate the data. To support high throughput and fast updates, there are a number of APIs that enable optimization of the drawing process to support most professional applications. It is also easy to have many views on the same document, and each view can be a different CView-derived class. For example, a splitter window will have one view for each pane.

To coordinate documents and views, MFC 2.0 uses the helper class CDocTemplate. This class orchestrates the creation of documents, views and frame windows in response to user input. One document template object is created for each document type, and is the glue that connects the document and view types. The application object maintains the document templates. Two MFC 2.0 document template classes are supplied: one for Multiple Document Interface (MDI) and one for Single Document Interface (SDI). The significant differences between an MDI and SDI user-interface model are encapsulated in the document template and frame window classes.

Printing

By leveraging the document/view architecture, MFC 2.0 is able to provide an application with device-independent printing. This means that the same code written for OnDraw in the CView-derived class can be used to draw on the screen and printer. When the user asks to

- more -

print a document using the standard File Print command, MFC 2.0 calls the OnDraw member function with a special device context that is aware of the current printer and knows how to translate your screen display into appropriate printed output. MFC 2.0 also provides support for all the standard printing user-interface dialogs. Full-featured Print Preview is implemented using the printing architecture.

Dialog Data Exchange and Validation (DDX/DDV)

Through a new capability known as dialog data exchange (DDX), MFC 2.0 provides an easy way to initialize the controls in a dialog box and gather input from the user. An associated mechanism known as dialog data validation (DDV) provides validation of the dialog data. The heart of the DDX/DDV feature is the DoDataExchange member function. This function is called automatically by the application framework. Consider the following short example in Figure 1:

```
void CMyDialog::DoDataExchange(CDataExchange* pDX)
{
    DDX_Check(pDX, IDC_CHECKBOX, m_bUser);
    DDX_Text(pDX, IDC_EDIT, m_strName);
    DDV_MaxChars(pDX, IDC_EDIT, m_strName, 20);
}
```

Figure 1

This example in Figure 1 shows a data exchange function for a checkbox with a control ID of IDC_CHECKBOX. The state of the checkbox (checked or not) will be stored in a BOOL member variable called m_bUser. This variable is a member of the CMyDialog class, so the dialog is fully encapsulated and easily reused in other parts of the application. The application framework automatically calls DoDataExchange, and DDX_Check transfers the data between the control and the member variable. The example in Figure 1 also shows a data exchange entry and a validation entry for a string variable. The DDX_Text function is automatically called to transfer data between a CString member variable called m_strName and the edit control in the dialog box. The DDV_MaxChars function is called to validate that the data entered by the user into the edit control does not exceed 20 characters. If the entry is invalid, the application framework automatically displays a message box informing the user of the error and returns the input focus to the edit control.

Because there are so many possibilities for exchange and validation (e.g., the use of custom controls or application-specific validation schemes), Microsoft made the DDX/DDV architecture fully extensible. You can supply your own DDX and DDV functions and integrate them seamlessly with MFC 2.0. The DDX/DDV architecture supports data types and Windows controls and includes a number of prebuilt DDX/DDV routines. Standard support for data types includes 16-bit integers (signed and unsigned), 32-bit integers (signed and unsigned) and strings (raw and formatted). DDX/DDV supports all the standard Windows controls, including checkboxes, radio groups, list boxes and combo boxes. Built-in DDV support is provided for maximum string length and ranges of integers.

The DDX/DDV architecture is tightly integrated with ClassWizard, which enables you to define all necessary member variables and DDX/DDV routines without having to write any code. Of course, ClassWizard is only a tool, and it follows the logical steps you would normally take by directly editing the source code. ClassWizard, however, is generally faster than manual file editing and is also less error prone.

Help

Support for online and context-sensitive documentation is essential for most applications. MFC 2.0 provides an architecture that makes it easy to incorporate the two most common types of help support in applications for Windows. Help support includes a Help menu with the standard commands, and provides an architecture for the application framework to map from command or resource IDs to the various help contexts. Help contexts are easily created in Visual C++ since every time a user-interface element is created in App Studio, a help context for that element is automatically created. Help files (.HLP) are authored using standard authoring tools.

When a user presses the F1 key, MFC 2.0 automatically processes the keystroke as a help request for the current command target. For example, the CDialog class processes the help request by invoking WinHelp on the help topic for the currently displayed dialog. If no help context is defined for the current command target, then the application framework automatically launches the default help. The CFrameWnd, CMDIFrameWnd and CDialog classes all provide handler functions for help support. You can add support to any class that is a command target.

When a user presses SHIFT+F1, MFC 2.0 captures the mouse and changes the cursor into the standard context-sensitive help cursor (arrow+question mark). With this cursor displayed, clicking on a
user-interface object tells the application framework to invoke WinHelp with the correct help context based on the selected object.

MFC 2.0 provides a tool to manage the help context information, which associates user-interface elements with help contexts. In addition, AppWizard provides much of the standard WinHelp format file with
pre-written information on all of the standard commands. All that is needed is an editor capable of editing rich text format (RTF) text, such as Microsoft Word, to add application-specific information. Therefore, the combination of MFC 2.0, AppWizard and App Studio works together to provide programmers with most of their application's help features automatically.

High-level Abstractions

MFC 1.0 was the cornerstone of a robust framework for building reusable classes, but it did not provide enough high-level abstractions to reduce programming time. MFC 2.0 addresses this with a set of classes that supports the most common user-interface idioms and provides capabilities for taking advantage of other prebuilt functionality. These classes, collectively

- more -

called the high-level abstractions, are designed to be used as supplied by MFC 2.0 and can result in a dramatic reduction in programming time. In a few lines of code, programmers can build a text processing window that integrates seamlessly with other MDI windows, or change the base class to turn a view into a scrolling view. In addition to this power, all of these high-level classes are designed to be easily modified using C++ inheritance.

Form View

One of the most common types of Windows-based applications is form processing. A form is like a dialog that the user can interact with to fill in edit controls, select options from listboxes and radio groups, as well as work with other dialog controls. For example, an order/entry database application would probably use forms to allow customer service representatives to enter order information.

The problem is that the Windows dialog manager does not support much of what true form processing applications require, such as scrolling, multiple forms for the same data, synchronous update and printing. MFC 2.0 enhances this model with the CFormView class. A CFormView provides a view (a class derived from CView) based on a dialog resource you edit with App Studio. You can use this view to create form views containing arbitrary Windows controls. The user can scroll the form view and tab among controls. The benefit of CFormView over standard dialogs is that CFormView objects integrate with the entire application framework architecture, so you get automatic support for command handling and document management. A form view can also be an MDI child window.

Edit View

A number of MFC 1.0 users requested that MFC 2.0 provide an abstraction that makes it easy to create a simple text editor. MFC 2.0 responded to that request by introducing the CEditView class. CEditView is like the low-level CEdit class since it provides the functionality of the standard Windows edit control. In addition, however, CEditView supports high-level functionality such as printing, Find and Replace, Cut, Copy, Paste and Undo, as well as the standard File commands (Open, Save, Save As). Of course, since CEditView is derived from CView, all of the architectural benefits described above apply. The sample program discussed later in this document shows the power of the CEditView; by simply creating a document template that uses CEditView, without even the need to derive your own view class, an application can have an MDI text editor.

Scrolling View

Most applications can only show a portion of their data files on the screen at a single time. The CScrollView class, which is another high-level view class derived from CView, supports views that scroll and views that are automatically scaled to the size of the frame window that displays them. By deriving from CScrollView, you can add the ability to scroll or scale to your view class. CScrollView manages window sizes and mapping mode for graphics, and scrolls automatically in response to user-interface actions, such as clicking on the scroll bar. It is also possible to specify that you require all of your data to fit within the frame window, in which case the CScrollView will stretch or shrink the logical view to fit within the main drawing area of the window.

- more -

Splitter Window

In a splitter window, the window can be split into two or more separately scrollable panes. A splitter control in the window frame next to the scroll bars allows the user to adjust the relative sizes of the panes. Each pane is a different view on the same document. This type of user interface is useful, for example, when a user wishes to view both the beginning and end of a very long document on a single screen.

MFC 2.0 provides the high-level class CSplitterWnd to support this user-interface model. The CSplitterWnd class also supports the two most common types of splitters: dynamic and static. With dynamic splitters the user can add or remove arbitrary split panes, while static splitters have a predefined number of panes. Each of the splitter pane's views can be the same class, or each can be a different derived CView class. In all cases, the application framework automatically manages all aspects of the user interface and standard commands.

Print Preview

In combination with the printing and document/view architectures, MFC 2.0 supports print preview functionality. Print Preview shows a reduced image of either one or two pages of a document as they would appear when printed on the currently selected printer. The implementation provides the standard user interface for navigating between pages, toggling between one- and two-page viewing, as well as zooming the display in and out to different levels of magnification. The ability to support Print Preview is a true measure of the amount of prebuilt functionality and high-level of abstraction in MFC 2.0. The Print Preview feature represents several thousand lines of code in the application framework, but programmers only need to handle the display output code in the OnDraw member function of class CView and make sure that the File Print Preview menu command is available — the framework does the rest.

- more -

Toolbar

One of the most commonly requested user-interface elements is the toolbar. A toolbar is a row of buttons represented by bitmaps and optional separators. These bitmap buttons can behave like push buttons, checkbox buttons or radio group buttons. The MFC 2.0 class CToolBar supports the recommended standard toolbar look and feel. All the toolbar buttons are normally taken from a single bitmap image, which is edited using App Studio, and contains one image for each button. One of the key advantages of the MFC 2.0 CToolBar implementation is that by using commands, the various buttons in the toolbar can be enabled and disabled in conjunction with any menu items for those same commands. This is important because toolbar buttons almost always duplicate menu items, as is recommended in the Windows Application Design Guide. If you require additional standard Windows controls on the toolbar, such as drop-down listboxes or edit controls, the CToolBar class can easily support them. In addition, CToolBar provides programmatic APIs for dynamically changing the buttons on the toolbar for highly customized user interfaces.

Status Bar

The CStatusBar class implements a row of text output panes, or indicators. The output panes are commonly used as message lines and status indicators. Examples include the menu help-message lines that briefly describe the selected menu command and the indicators for the keyboard states of Num Lock, Scroll Lock and Caps Lock. The CStatusBar class supports any number of panes and automatically lays them out based on the width of the contents. Each pane can have a customized style, including 3-D borders, pop-out text, disabled and stretchy. The MFC 2.0 command architecture supports automatic menu prompt strings, and when using App Studio to edit menus for MFC 2.0 applications, you can also define the prompt string for the menu item. The standard status bar code that is output by AppWizard in new applications supports the Windows Application Design Guide recommendations.

Dialog Bar and Control Bar

The CDialogBar class is like a modeless dialog in that it easily supports any combination of Windows controls and is created from a dialog template edited by App Studio. Dialog bars support tabbing among controls and can be aligned to the top, bottom, left or right edge of the enclosing frame window. The most common example of a dialog bar is the Print Preview user interface.

CToolBar, CStatusBar and CDialogBar all derive from the common base class CControlBar. The CControlBar abstraction enables the MFC 2.0 implementation to reuse code among these classes. CControlBar provides the functionality for automatic layout within the parent frame window of the derived classes. CControlBar demonstrates the power of a base class that provides a partial implementation that is completed in a series of closely related derived classes.

OLE 1.0 Support

MFC 2.0 provides nine classes that support Object Linking and Embedding (OLE) 1.0.

- more -

These classes build upon the support in MFC 1.0, making it even easier to build applications that support OLE.

MFC support for OLE includes classes for implementing a client, implementing a server and several helper classes, as well as most of the standard dialogs and menu items. The main benefit of the OLE support is its integration with the document/view and command architectures, which smoothes the transition to an OLE application. Checking the OLE Client support option in AppWizard provides an excellent start with OLE because it automatically enables an application to act as an OLE client. In the future, MFC will provide support for OLE 2.0, so using the OLE 1.0 support today will make the transition to the new OLE 2.0 paradigm much easier.

Windows API Classes

MFC 2.0 provides classes that simplify programming for Windows while at the same time permitting application developers to leverage both existing Windows code and programming experience. For the inexperienced programmer of Windows-based applications, the Microsoft Foundation Classes simplify programming for Windows by providing canned functionality for many standard Windows-based programming idioms. These classes have evolved from the MFC 1.0 implementation, but backward compatibility has been maintained. Developers with MFC 1.0 applications need only recompile their application to begin using MFC 2.0. A brief technical note provided with Visual C++ describes the few minor changes that must be made to the application’s source code.

Standard Application Support

MFC encapsulates the standard application structure in an easily customizable application object. In addition to standard initialization, message processing and termination, the CWinApp class supports idle time processing of user-defined operations. The scope of the CWinApp class has been expanded in
MFC 2.0 to include support for profile settings, context-sensitive help, File Manager drag and drop, shell registration for launching the application from File Manager and other user-interface features. The CWinApp class frees the programmer from the details of the WinMain, LibMain and WEP routines, and provides a standard abstraction across Windows platforms.

Frame Windows

Along with an application object, most programs use a standard frame window. MFC 2.0 provides support for both the single-document interface and the multiple document-interface (MDI). The CMDIFrameWnd and CMDIChildWnd structure of MFC 1.0 has been maintained. In MFC 2.0, however, many of the common MDI commands and user-interface functionality, such as changing the menu bar that is based on the active window, are now provided as canned functionality by the framework. In addition, error-prone areas of Windows-based programming, such as keyboard accelerators and implementation of default behavior, are handled in a seamless manner by the application framework. Frame windows are managed by the document template class in applications that take advantage of the document/view architecture. A view is contained within a frame window (usually a CFrameWnd or a CMDIChildWnd).

Controls

Controls are windows that are drawn in the client area of frame windows, or as controls

- more -

in a dialog box. MFC 2.0 provides classes for all of the standard controls: static text, buttons, edit control, list boxes, combo boxes, scroll bars, handwriting controls and user-defined child windows. MFC 2.0 makes it easy to derive your own child windows (including deriving from the standard Windows controls) and to customize their behavior using C++ inheritance and message maps. MFC 2.0 has also enhanced the CBitmapButton class introduced in MFC 1.0.

Graphics/GDI

The MFC 1.0 device context class, CDC, provided a simple Windows API wrapper. MFC 2.0 extends the DCC implementation to allow polymorphic implementations of device context output functions. This enables a virtual display context that allows MFC 2.0 applications to use the same drawing code to send output to the screen, a printer, a metafile or a Print Preview view. MFC 2.0 provides a complete set of classes for drawing graphical objects and managing device contexts. These graphical object classes include all of the standard Windows objects, including pens, brushes, bitmaps, fonts, regions and palettes. Several device context classes are also supplied to make the handling of common idioms (such as window repainting) simple and less error prone. The graphical objects are designed to automatically free system resources when they are no longer needed, which simplifies common object-ownership problems and enables an application to run safely in a resource-constrained environment.

Dialogs

MFC 2.0 makes it even easier to use dialogs within an application. The application framework manages many of the intricate Windows operating system-oriented details of dialogs automatically, including the handling of dialog-specific messages. Dialogs are handled with the CDialog class, which supports both modal and modeless dialogs. Programmers simply derive from a dialog class and customize it by overriding member functions and message handlers. This customization model is exactly like every other CWnd-derived class, which provides good programming consistency.

General-purpose Classes

The general-purpose classes provide programmers with a wide range of functionality designed to take advantage of the powerful features of C++. These classes are available for programmers to develop the non-graphical portion of the application. In many respects, the general-purpose classes, together with the Windows API classes, are the building blocks for the entire application framework and provide fundamental functionality to those classes as well as programmer-defined classes.

Run-time Object-type Information

Most MFC classes are derived, either directly or indirectly, from the class CObject, which provides the most basic object-oriented features of the framework. CObject supports dynamic type checking, which allows the type of an object to be queried at run time. This feature provides programmers with a type-safe means to cast down a pointer from a base class to a derived class. Without dynamic type checking, this cast can be a source of errors and can break the type safety of C++. Most programmers find this feature useful, but since it incurs a very small runtime overhead (approximately 24 bytes per class) its use is optional.

- more -

Object Persistence

Persistence is the ability of any object to save its state to a persistent storage medium such as a disk. For example, if a collection is made persistent, then all members of that collection are made persistent. The CArchive class is used to support object persistence and allows type-safe retrieval of object data. To use persistence, a class implementor must override the Serialize member function, call the base class's Serialize function, and then implement the data storage routines for member data that is specific to a derived class. Entire networks of objects, with references to other objects, including both multiple and circular references, can be saved with a single line of code. As with dynamic type checking, the use of persistence is optional.

Data Structures

The efficiency of standard data structures is an area in which MFC 2.0 excels. The provided collection classes, a standard component of any C++ class library, are well-tested, well-coded and highly reusable. The MFC collection classes include double-linked list classes, map (dictionary) classes and dynamic (growable) array classes. All of these have been implemented using the proposed ANSI template syntax for type-safe usage. For example, the list class is supplied with variants supporting UINT, BYTE, WORD, DWORD, void*, CObject* and CString elements. The map and array classes have similar sets of variants. In all, MFC 2.0 supplies 17 collection classes. For users who wish to take advantage of the template syntax to generate a type-safe variant of a supplied implementation (or write their own template), a template expansion tool written using MFC 2.0 is provided as a sample application.

Strings

The CString class supports a very fast string implementation that is compatible with standard C "char*" pointers. This class allows strings to be manipulated with syntax similar to the Basic language that includes concatenation operators and functions such as Mid, Left and Right. CString also provides its own memory management, freeing the programmer from allocating and freeing string memory.

Files

MFC 2.0 offers three file classes: CFile and its two derived classes, CStdioFile and CMemFile. CFile supports low-level binary file I/O (read, write, seek). CStdioFile provides buffered file I/O similar to the standard I/O run-time libraries. CMemFile supports file semantics in RAM-resident files for managing clipboard data as well as other forms of interapplication communication. The polymorphism provided by the three file classes (CFile, CStdioFile and CMemFile) allows the same code to be used for sending data to a variety of destinations using the CFile interface. MFC 2.0 has added support for reading and writing huge data blocks greater than 64K.

Time and Date

In addition to the standard time and date functions, a class is provided to conveniently support time and date arithmetic using overloaded operators. Binary time values are automatically formatted into human readable form.

Exception Handling

MFC 2.0 supports a robust exception-handling mechanism that is upwardly compatible with the proposed ANSI try/throw/catch standard. The CException class (including several derived classes) provides exception-specific support for memory, archiving, filing, resource and other exceptions. Rather than using return codes that are often overlooked and result in inefficient code, the exception syntax is a clean and efficient mechanism for handling fatal conditions.

Debugging and Diagnostic Support

An area overlooked by many class libraries is the inclusion of sophisticated diagnostic and debugging facilities. Incorporated directly into the fabric of MFC 2.0 is a backbone of diagnostic code that is supported in the debug version of the framework. Applications written with MFC 2.0 and compiled for debugging can be up to twice as large as their non-debug counterparts — an indication of the extensive diagnostic support within the application framework.

Programmers can add debug code anywhere in an application that will print out all currently allocated heap objects. This capability is invaluable for the detection of serious memory leaks that are often impossible to track by other means. A memory leak is a slow depletion of system resources that can go undetected for several days until all resources are consumed. For all heap-allocated objects, a record is kept of the size, source file and line number of the allocation. After a debug version of an MFC 2.0 application terminates, the application framework automatically displays all heap objects the programmer failed to free.

Other debug support includes functions that are able to validate any pointer and determine if it

- more -

refers to a genuine C++ CObject-derived object. Other facilities provided by the framework are run-time assertions and class invariants, which were popularized by the Eiffel programming language. Every class in MFC 2.0 implements a member function that checks the current state of the object and causes a debug assert if the object is not in a proper state. Library member functions validate parameters to functions in the debug version of the framework.

There are more than 2,300 ASSERT statements within the implementation of MFC 2.0, each of which checks the condition of the internal state of a class or parameters passed into an API. If a programmer erroneously causes the application framework to enter an unpredictable state, the application will immediately break into the debugger (if it is running) or an alerting message box will be displayed. ASSERT statements catch errors much earlier and can save hours of development time. All major Microsoft applications use assertion statements extensively. In the release (non-debug) version of an MFC application, ASSERT statements are not executed and generate no code. They are designed for testing purposes only and thus incur no cost to the application's end users. The ASSERT mechanism is provided for users of MFC as well, and programmers are encouraged to take advantage of it within their own code.

MFC 2.0 also provides TRACE statements, which are formatted information messages. As with ASSERT statements, TRACE statements are executed only in the debug version of an application. The TRACE statements in the application framework display possible misuse of a feature, low memory conditions, rarely executed boundary conditions, and full message and command tracing. Since the output can be verbose (there are nearly 300 TRACE statements in MFC 2.0), it is easy to select which categories of messages are reported using the TRACER.EXE application. For example, if you are only interested in information about OLE, you can filter out all the other TRACE output. The TRACE facility can also be used by programmers within their own code.

Professional Windows-based Development

MFC 2.0 takes into account the needs of engineers developing large-scale professional applications for Windows.

Much effort has gone into optimizing the code size of executables and keeping them as small as possible. A combination of the advanced features of the Microsoft C++ compiler/linker (such as function-level linking) and the module granularity of MFC 2.0 reduce the size of MFC applications.

The Microsoft Foundation Classes use C++ idioms in the commonly accepted manner and do not overlook advanced issues such as copy construction, assignment operators and correct object destruction. These issues are a common source of errors in user code, and can be difficult to track down. MFC 2.0 code serves as an example of both professional C++ code and professional Windows code.

Sample Program: MultiPad

In the following pages, a sample application using MFC 2.0 is demonstrated. The purpose is to

- more -

show the power and simplicity of the framework's architecture and implementation. The application is a simple MDI text editor, similar to the MultiPad sample application shipped with the Windows SDK. MFC 1.0 also supplied a similar sample application.

This example is not designed to cover every aspect of MFC 2.0 nor is it designed to make you an expert in all the features of MFC 2.0. The intent is to illustrate the powers of the MFC 2.0 architecture, and to show how useful the high-level abstractions can be for quickly implementing large amounts of functionality. This example originated as an AppWizard-created application, but has been merged into a single source file in order to serve better as an example.

```
// MultiPad : Simple MDI text editor written using MFC 2.0

/////////////////////////////////////////////////////////////////////////////
// Interface

#include <afxwin.h>   // MFC core and standard components
#include <afxext.h>   // MFC high-level abstractions
#include "resource.h" // Resource symbols

class CMultiPadApp : public CWinApp
{
	virtual BOOL InitInstance();

	afx_msg void OnAppAbout();
	DECLARE_MESSAGE_MAP()
};

class CMainFrame : public CMDIFrameWnd
{
	CStatusBar  m_StatusBar;
	CToolBar    m_ToolBar;

	afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
	DECLARE_MESSAGE_MAP()
	DECLARE_DYNCREATE(CMainFrame)
};

class CPadDoc : public CDocument
{
	virtual void Serialize(CArchive& ar);

	DECLARE_DYNCREATE(CPadDoc)
};

/////////////////////////////////////////////////////////////////////////////
// Implementation

CMultiPadApp NEAR theApp;           // define a single application object

BEGIN_MESSAGE_MAP(CMultiPadApp, CWinApp)
	ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
	ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)     // file commands...
	ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
	ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

BOOL CMultiPadApp::InitInstance()
{
	SetDialogBkColor();
	LoadStdProfileSettings();

	AddDocTemplate(new CMultiDocTemplate(IDR_TEXTTYPE,
		RUN-TIME_CLASS(CPadDoc), RUN-TIME_CLASS(CMDIChildWnd),
		RUN-TIME_CLASS(CEditView)));
```

- more -

```
42.         m_pMainWnd = new CMainFrame;
43.         ((CFrameWnd*)m_pMainWnd)->LoadFrame(IDR_MAINFRAME);

44.         m_pMainWnd->DragAcceptFiles();
45.         EnableShellOpen();
46.         RegisterShellFileTypes();

47.         m_pMainWnd->ShowWindow(m_nCmdShow);
48.         if (m_lpCmdLine[0] == 0)
49.             OnFileNew();
50.         else
51.             OpenDocumentFile(m_lpCmdLine);
52.         return TRUE;
53.     }

54.     void CMultiPadApp::OnAppAbout()
55.     {
56.         CDialog about(IDD_ABOUTBOX)
57.         about.DoModal();
58.     }

59.     IMPLEMENT_DYNCREATE(CMainFrame, CMDIFrameWnd)
60.     BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
61.         ON_WM_CREATE()
62.     END_MESSAGE_MAP()

63.     static UINT buttons[] =
64.     {
65.         ID_FILE_NEW, ID_FILE_OPEN, ID_FILE_SAVE, ID_SEPARATOR,
66.         ID_EDIT_CUT, ID_EDIT_COPY, ID_EDIT_PASTE, ID_SEPARATOR,
67.         ID_FILE_PRINT, ID_APP_ABOUT
68.     };

69.     static UINT indicators[] =
70.     {
71.         ID_SEPARATOR, ID_INDICATOR_CAPS, ID_INDICATOR_NUM, ID_INDICATOR_SCRL
72.     };

73.     int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
74.     {
75.         CMDIFrameWnd::OnCreate(lpCreateStruct);
76.         return ((m_ToolBar.Create(this) &&
77.             m_ToolBar.LoadBitmap(IDR_MAINFRAME) &&
78.             m_ToolBar.SetButtons(buttons, sizeof(buttons)/sizeof(UINT)) &&
79.             m_StatusBar.Create(this) &&
80.             m_StatusBar.SetIndicators(indicators, sizeof(indicators)/sizeof(UINT)))
81.                 ? 0 : -1);
82.     }

83.     IMPLEMENT_DYNCREATE(CPadDoc, CDocument)

84.     void CPadDoc::Serialize(CArchive& ar)
85.     {
86.         ((CEditView*)m_viewList.GetHead())->SerializeRaw(ar);
87.     }
```

Figure 2

The following is a description of the major program elements of the MultiPad application.

Lines 1-6:

These include the standard MFC 2.0 header files. The file RESOURCE.H, which isn't shown here, contains the symbol definitions for the commands and user-interface elements; it is

- more -

usually edited only by App Studio.

Lines 7-12:

Every MFC application must declare a class derived from CWinApp. The CWinApp class encapsulates much of the mundane work normally associated with getting an application started and running. There are overridable member functions for initialization, message-loop processing, idle-loop processing, as well as support for File Manager drag and drop, Shell registration, and most recently used file list management. In this example, we override InitInstance to perform some one-time initialization. Since a CWinApp-derived class is a command target, there is also a message map for this class. The OnAppAbout command handler is used for the About box. Lines 10 and 11 are usually maintained by ClassWizard, so this is code you do not normally write manually.

Lines 13-20:

CMainFrame is a class derived from the MFC 2.0 class CMDIFrameWnd, which provides support for MDI window management. The CMainFrame class adds two member variables for the toolbar and status bar, each of which is one of MFC 2.0's high-level abstractions. Lines 17-19 are usually maintained by ClassWizard and indicate that the class has a message map that will handle the OnCreate message (WM_CREATE). The DECLARE_DYNAMIC macro allows the application framework to dynamically create the frame window.

Lines 21-25:

MultiPad manages documents that are merely standard text files. The CPadDoc class is needed to implement the Serialize overridable member function. Serialize is called automatically by the application framework in response to File Open, File Save and File SaveAs commands.

- more -

Line 28:

By declaring a CMultiPadApp class object, the constructor for that object will be invoked at program start-up. Every MFC application must define a single application object that replaces the normal WinMain functionality. When the program starts, MFC 2.0 automatically calls the initialization functions, and if they are successful, the framework's message loop will be run. When the user executes the File Exit command, the application terminates.

Lines 29-34:

These lines implement the message map for the CMultiPadApp. Each one is a command handler and each handles one of MFC 2.0's standard commands. If you are familiar with MFC 1.0, you will notice that the message-map structure is unchanged. Further down in the code is a definition for the OnAppAbout member function, which is called when the user executes the Help About command. The three other commands are all implemented using the canned implementation, which is why there are references to the application framework class CWinApp. These message map entries implement the File New, File Open and File Print Setup commands. Although these entries could automatically be supplied for all applications, MFC 2.0 requires that you pay only for functionality that you use. For example, if an application does not support printing, then MFC 2.0 does not force the application to link in all of its printing code. MFC 2.0 follows this swap-tuning practice frequently. Rather than having large executables, MFC 2.0 allows programmers to add a single line of code that enables the canned library implementation.

Lines 35-36:

The InitInstance function is actually the bulk of the code that needs to be written to implement the application. This function is called automatically by MFC 2.0 when the application starts.

Line 37:

MFC 1.0 supported new look gray-colored dialog boxes by default. However, a number of users commented that they would prefer a single function that lets them choose the background color of dialogs. MFC 2.0 therefore added the SetDialogBkColor function for that purpose. The default arguments, which are not shown, set the dialog background and text colors to the MFC 1.0 values.

Line 38:

The LoadStdProfileSettings function loads the user's preferences from the MULTIPAD.INI file stored in the user's Windows directory, that is the preferred mechanism for saving profile settings. If an application has other settings which should be restored, then InitInstance is the best place to restore them, along with the standard
MFC 2.0 settings. The standard profile settings include the most recently used file list in the File menu, and some print preview information. This API is an example of how MFC 2.0 provides prebuilt functionality that a program does not need to pay for if it does not use the feature. If an application does not need the profile settings, then omitting this line will not require the application to maintain an .INI file, and leaves the MFC 2.0 code that implements the feature out of the final executable.

Lines 39-41:

In order to use the document/view architecture, it is necessary to create a document template for each document type, which is done by calling the AddDocTemplate API. The document template orchestrates the creation of the document, view and frame window. Since this is an

- more -

MDI application, the CMultiDocTemplate class is used. Also, since the document template needs to be used after InitInstance, it is allocated on the heap using the C++ **new** operator. MFC 2.0 automatically frees the memory associated with the document template. The constructor for CMultiDocTemplate requires four parameters. The first parameter is the ID of a string resource that contains several strings, including the default window title, the description of the document type and the strings needed by the standard file dialog. Each of the other three arguments is a run-time class, which gives the application framework enough information to create objects of the given type. The second argument is the
run-time class of the document type, which in this program is CPadDoc. The third argument is the run-time class of the frame window to be used for this application. Since this is an MDI application, the CMDIChildWnd class is used directly. The frame window can also be an SDI frame or it can be any class derived from CFrameWnd. The fourth argument is the name of the view class. MultiPad requires a view that can draw and edit text, manage the clipboard, and implement find/replace and printing. MFC 2.0 provides this canned functionality in the high-level abstraction class CEditView. The programmer only needs to pass the run-time class for CEditView to the document template and the application will use the built-in MFC 2.0 class with the document class. For more specialized applications one can supply any CView-derived class to the document template.

Lines 42-43:

The next step in initializing the application is to create an application window. (The allocation of the application window was declared previously in lines 13-20). The LoadFrame API creates the application window and integrates it with the application framework architecture. For example, the API assigns a help context to the window, and loads the appropriate icon, accelerator table and menu from the resources in the executable file. The cast is required because the m_pMainWnd member variable in the CWinApp class is a CWnd pointer and CMDIFrameWnd is derived from that class. The application framework is flexible in allowing any window such as a dialog to be the application window.

Lines 44-46:

In order to support the File Manager drag and drop and the File Manager Open and Print commands, these three lines are required. Adding these three lines enables these features for applications that require them, and eliminates the need to add a larger amount of code and a separate registration file (.REG file). In fact, if a document extension is specified when creating an MFC 2.0 application with AppWizard, these lines are automatically placed in the application's InitInstance, as they are in this example.

Lines 47-53:

At this point, the application only needs to process the command line. If a file name is present on the command line, the OnFileOpen command handler is called directly; otherwise the OnFileNew handler is called to display a blank "Untitled" window. These functions are implemented in the application framework. In line 47, the ShowWindow API is called to display the application window. As with MFC 1.0, any MFC 2.0 API that is implemented as a direct call to the Windows API, such as ShowWindow, is named the same as the corresponding Windows API. Returning TRUE from InitInstance indicates that the initialization was successful.

- more -

Lines 54-58:

The OnAppAbout command handler demonstrates how easy it is to bring up a modal dialog in MFC 2.0. The constructor for CDialog requires the ID of the dialog resource, which is created and edited with App Studio. The DoModal call creates the dialog and processes messages until the end user clicks OK.

Lines 59-62:

These lines create the message map for the application window. Although MFC 1.0 message maps were edited by hand, MFC 2.0 includes ClassWizard, which maintains message maps automatically. Of course, you are still free to use a standard text editor to manage message maps, as the syntax is unchanged.

Lines 63-72:

The toolbar and status bar both require command IDs for the commands that these user-interface abstractions handle. The commands are defined in a simple array of integers. For a toolbar, the array implements one-to-one mapping based on the positions of the button tiles in the toolbar bitmap, which is edited within App Studio. For status bars, the array implements a one-to-one mapping based on the indicator fields of the status bar. Separator entries in either array indicate buttons or indicators that will be skipped.

Lines 73-82:

The OnCreate message handler is called in response to the LoadFrame call. OnCreate is the first message a window receives, and is usually the best place to implement one-time initialization of a window. Notice that instead of having to parse wParam and lParam, as is required in C code, the message was mapped directly to a C++ member function. The OnCreate interface provides type-safety and recompile-only portability to Windows NT and Win32s™, where a number of messages have changed their wParam and lParam encoding. For the application window, the toolbar and status bar need to be created. First, the base class OnCreate function is called, which is an MFC convention, and then several CToolBar member functions are called. Next, the toolbar is created, the bitmap resource is loaded from the executable, and the buttons are hooked to the commands using the array defined above. This example shows the buttons defined at program initialization, but it is just as easy to alter the buttons programmatically for a fully end-user customizable user interface. The status bar is handled in two steps: First it is created, and then the indicators are set using the commands in the array defined above. Should any of these calls fail (for example, if there are not enough system resources to load the bitmap) then the OnCreate function will fail and the application will not run.

Line 83:

The IMPLEMENT_DYNCREATE macro initializes the data structure required by the run-time class mechanism. There will be such a line for each class that maintains run-time type information.

Lines 84-87:

In order to read and write the text data of a CPadDoc document, it is necessary to override the Serialize member function. The CEditView class provides an interface that reads and writes the data in the view, so calling the view's Serialize member function is all that is required. As previously mentioned, a document can have any number of views attached to it, which is why there is a reference to the list of views, m_viewList. This line simply gets the first view, which

- more -

is known to be a CEditView, and calls the CEditView API that reads and writes the text of the file in a *raw* (unmodified) format.
A screen image of the running application is shown in Figure 3. Although it shows a number of useful features, it is difficult to capture them all on one screen. MFC 2.0 MultiPad implements the following menu commands: File (New, Open, Close, Save, SaveAs, Print, Print Setup, Exit and four most recently opened files); Edit (Undo, Cut, Copy, Paste, Delete, Find, Find Next, Replace, Select All, View) (toolbar and status bar); Window (Cascade, Tile, Arrange Icons and an MDI child window list); and Help (About). There are toolbar buttons for File New, File Open, File Save, Edit Cut, Edit Copy, Edit Paste, File Print and Help About. The status bar has indicators for the prompt string of the currently selected menu item (or Ready if no menu is selected), Num Lock, Caps Lock and Scroll Lock. MultiPad supports opening files from the File Manager by double clicking on any .TXT file. From the File Manager, any .TXT file can be dragged onto MultiPad and it will automatically be opened. The application also follows standard MDI window management and has the appropriate title bars and minimized/maximized behavior.

Sample screen for MultiPad: Note the MDI windows, toolbar, status bar (with prompt string), standard Search/Replace dialog and standard menu structure.

Figure 3

Looking at this example program, which contains only 87 lines, there are a number of key points to notice beyond the features.

First, the application is standard C++ and makes use of the standard Windows APIs where appropriate. The code contains no language extensions and can compile with any standard C++ compiler for Windows. This example application only required one call to a Windows API that is wrapped by an MFC 2.0 class (this is the same function as in MFC 1.0). In general, MFC 2.0 will have fewer calls to these wrapped functions because of the higher level of abstraction. On the other hand, when a programmer requires direct access to the Windows API, MFC 2.0 is designed to facilitate that as well.

Several times in this example we took advantage of the canned functionality within the MFC 2.0 application framework. The functionality is a robust, small and fast implementation of standard programming paradigms for Windows. These features exist for two reasons: First, they save programmers time and effort; second, they show programmers the standard way of implementing a Windows user-interface idiom. For example, the OnFileOpen command handler prompts the user with a standard File Open dialog, selects files of the appropriate type, validates the existence of the file, opens the file, reads the data into the document object, creates a frame window, sets the title of the window using the Windows standard, and resets the menu bar as appropriate for the particular document. If programmers need to customize any of this

- more -

functionality, they only need to override a function. Since MFC 2.0 source code is included and serves as an example of a correct implementation, programmers have all the information needed to implement any user-interface paradigm required by an application.

MFC 1.0 was often categorized as being a thin veneer or a literal wrapping of the Windows API. This was true, because the goals of MFC 1.0 were such that providing something close to the Windows API would help get C and SDK programmers into the world of C++. As it turns out, most MFC applications took fewer lines of code to implement an application than other application frameworks.[1] For MFC 2.0, the AFX group set out to help more programmers write programs for Windows in C++ by providing high-level abstractions that encapsulate a large amount of functionality with a few lines of code.

An important design goal of MFC 1.0 was that applications written with MFC 1.0 be as small and fast as those written with C and the SDK. MFC 1.0 received much praise for the small size of the executables compared to all other application frameworks. In terms of execution speed, MFC 1.0 proved to be faster than other techniques for handling Windows messages.[2] With MFC 2.0, we have maintained the same commitment to size and speed. If an MFC 1.0 application is recompiled and linked with MFC 2.0 libraries, only a minor increase in size is expected — on the order of 20-30K. This increase is due to the fact that some functions, even if they are not directly called, cannot be removed by the Microsoft smart linker. However, as MFC 2.0 features are used, an application will grow in size depending on the particular features that are used.

[1]Sizing Up Application Frameworks and Class Libraries, *Dr. Dobb's Journal*, October 1992. A comparison of various application frameworks for Windows, including Borland OWL and Inmark zApp, which showed the MFC 1.0 application had the smallest executable size and required the fewest lines of code.

[2]C++ Q&A, *Microsoft Systems Journal*, September 1992, Bob Chiverton. A comparison of the Borland DDVT implementation and the Microsoft Foundation Classes message map facility.

- more -

Developer Support

MFC 2.0 provides developer support in a number of key areas. Based on the feedback from MFC 1.0, Microsoft made a number of improvements in the support provided. Online documentation and technical notes are supplied in WinHelp format: the tutorial has been expanded to implement more features; the amount of technical overview material in the documentation has increased; and the dependencies on the Microsoft C++ compiler have decreased.

- **Complete API Reference:** A complete printed reference for all public member functions and member variables is available. This information is also available via the online help system (in WinHelp format). In addition, references are provided to Windows APIs where appropriate.

MFC 2.0 includes much more overview material and has been written assuming less Windows API knowledge on the part of the programmer.

- **Tutorial:** To familiarize users with MFC 2.0, a tutorial is provided. Users are guided through a step-by-step procedure for developing a non-trivial Windows-based application that includes windows, dialogs, graphics, menus, commands, scrolling, files, printing and persistent data. The source code to the application is also provided.

- **Cookbook:** Topics covered in the Class Libraries User's Guide require more detail than the tutorial and API reference provide. A broad range of topics is covered in depth and includes programming examples for exceptions, collections, diagnostics and application design. Users can refer to these chapters for answers to more complex questions.

- **Technical Notes:** Many questions and problems faced by programmers are not easily documented in traditional forms such as the Class Libraries User's Guide, tutorial and API reference. Realizing this, MFC 2.0 includes technical notes, which are concise notes written by the AFX development and quality-assurance engineers. These notes describe specific problems and solutions encountered by users of the system, and include source-code examples and detailed technical information for intermediate to advanced users. The notes also provide details on the implementation of the application framework. The technical notes are provided in WinHelp format in MFC 2.0.

- **Sample Source Code:** Many feel that the best way to learn how to program for Windows
  and use an application framework is by using sample programs written by the developers of the product. MFC 2.0 includes 24 sample programs consisting of over

- more -

22,000 lines of C++ code. These programs demonstrate nearly all aspects of the framework in a series of non-trivial applications including OLE clients and servers, a full-featured MDI text editor, a charting application, use of DLLs and so forth. To help navigate the samples, a complete description of the sample programs organized by feature area (in addition to an alphabetic reference) is provided in WinHelp format.

- **Source Code:** The complete source code for the MFC 2.0 library is supplied. This code follows consistent naming and formatting conventions that make it easy to use and understand. If necessary, developers can build a custom version of the framework — for example, if a memory model was not provided by the default installation (MFC supports all memory models), or if different compiler and linker flags were required.
- **Compiler Support:** MFC 2.0 has been written using techniques that facilitate the use of third-party compilers. In general, the work required to use MFC 2.0 with a third-party compiler involves changing a few #define statements in a compiler-specific version file, adding a compiler-specific .CPP file, and creating a compiler-specific makefile.

- more -

Scalable Architecture for the Future

The Microsoft Foundation Classes represent an entire family of class libraries. The design of MFC 2.0 incorporates an architecture that is highly scalable; as new versions of the Windows operating system are released, the Microsoft Foundation Classes will grow in a natural manner to encompass new capabilities. As a case in point, MFC 2.0 is designed to facilitate application portability to the new 32-bit Windows API, including Windows NT. Programs written using the Microsoft Foundation Classes need only be recompiled in order to work with 32-bit Windows APIs.

Design Philosophy

With the introduction and acceptance of MFC, many developers have asked Microsoft to provide a succinct set of design goals that the AFX group followed in designing MFC. The central design tenet of MFC is to facilitate and simplify programming for the Windows environment, while providing a path for growth into future Windows environments. Achieving this design goal included adherence to a set of design principles which were the result of a design and prototype phase.[1]

- Application frameworks for Windows should fully exploit the power of the C++ language without overwhelming the programmer. C++ is a complex language with hundreds of new features. A class library should therefore utilize a sensible subset of the C++ language while at the same time permitting the use of the entire C++ language.

- Application frameworks for Windows should present a model that requires minimal relearning on the programmer’s part. Developers who have experience with the Windows Software Development Kit (SDK) should be able to quickly comprehend the programming model, class hierarchy and naming conventions. By mixing traditional C-language SDK code with C++ objects for Windows, programmers should be able to produce readable source code that can be easily maintained. In addition, programmers should be able to leverage third-party materials such as books, sample code and courses, and should not be forced to learn an entirely new paradigm or API. Developers who are not familiar with C/SDK programming should be able to learn Windows-based programming faster using an application framework, because the application framework shields the programmer from the low-level details and labor associated with SDK programming.

- Recognizing that a major end-user benefit of Windows is the standard user-interface paradigm shared by most applications, an application framework should support the standard Windows interface with minimal coding. At the same time, the application framework should be flexible enough to be used for specialized user-interface

[1]A Tale of AFX at Microsoft, Microsoft Developer Network News, Fall 1992.

- more -

elements. The ability to customize the user interface by overriding a member function is important; but equally important is the ability to call native Windows APIs for maximum flexibility and power.

- An application framework should represent a balance between power and efficiency. Application frameworks that attempt to provide too high a level of abstraction through “heavy” classes with many virtual member functions usually produce large, slow applications. But abstraction does not necessarily imply big and slow. The most elegant solutions are usually those that are the smallest and fastest.

- With its broad market support, the Windows environment will be around for many years. Its capabilities will grow substantially as new versions, such as Windows NT and Win32s, are released. An application framework must therefore represent more than a set of classes; it must be a scalable architecture that grows as the Windows environment grows.

As this paper demonstrated, the MFC 2.0 application framework working in concert with AppWizard, ClassWizard, and App Studio implements these principles. Following these principles over the course of the development of MFC 1.0 and MFC 2.0 has resulted in an application framework that is the C++ interface to Windows needed by C++ programmers; that provides an architecture and high-level functionality expected from an application framework; and that at the same time satisfies the size and speed requirements of professional developers for Windows.

Microsoft Foundation Classes 2.0: Class Hierarchy

µ §

- more -

Appendix A: AppWizard

AppWizard makes it easy to get started using MFC 2.0 because it automates the first steps in using the application framework. For nearly all applications, the fastest and easiest way to get started using MFC 2.0 is to run AppWizard and work with the application template that it creates. AppWizard creates a fully compilable, ready-to-go, full-featured, standard application for Windows.

One of the toughest problems faced by a developer trying to use an application framework is trying to figure out where to start. The C/SDK programmer starts a new application by copying a template, such as GENERIC, that has the standard message loop, WinMain, and other associated grunge. With MFC 2.0 these steps are not needed, since the architecture supports them already without any coding. If you want your application to use some of the high-level architectural features, such as printing documents and views, then you need an infrastructure to build upon. AppWizard provides that infrastructure by supplying the project files you need to immediately reap the benefits of MFC 2.0 for writing Windows-based applications.

AppWizard is run from the Visual WorkBench's Project menu. To get started, you just supply a directory for your application. Since AppWizard is tuned for substantial applications, it enforces the rule that each application must have its own source directory. You can select a number of options to customize the initial contents of your application.

Before we explore what AppWizard does for you, it is important to understand what AppWizard does not do for you. AppWizard is not a CASE tool or code generator, but rather it is an efficient way to leverage the power of the code within the application framework. AppWizard is only run a single time for each application. If at a later time you require some of the functionality that AppWizard can provide, you must add the code manually; or often you can use ClassWizard. This permits MFC 2.0 applications to be free form and not impose any artificial structure on programmers, as is common with CASE tools and code generators. AppWizard does not require you to maintain any special data files, but rather it works the way you do. The output of AppWizard is a set of source files that use the application framework, and as such provides more structure than code. The code that does the work for you is in the application framework.

As an indication of the powerful combination of AppWizard and MFC 2.0, the following is a list of features in a fully enabled AppWizard-created application template.

- MDI application (SDI is optional)
- Toolbar with buttons mapping to the menu commands File New, File Open, File Save, File Print, Edit Cut, Edit Copy, Edit Paste, Help About and context-sensitive SHIFT-F1
- Status bar with menu command prompt strings for all commands and indicators for Num Lock, Scroll Lock and Caps Lock

- more -

- File menu commands for New, Open, Close, Save, Save As, Print, Print Preview, Print Setup, Exit and recent file list
- Standard dialog support for obtaining file name information from the user (for Open, Save As)
- Standard dialog support for Print and Print Set-up
- Print preview support including Next/Prev Page, Two Page view and Zoom In/Out
- Edit menu hooks for Undo, Cut, Copy and Paste
- OLE 1.0 command and user-interface support for Edit menu commands Paste Link, Insert New Object, Links and Object
- View menu to enable or disable the toolbar and/or status bar
- Window menu commands New Window (including correctly naming each window by appending a numeric index), Cascade, Tile, Arrange Icons and an active window list
- Online Help support including a help file with content covering all of the standard commands

This functionality requires no coding on your part — just invoke AppWizard from the Visual WorkBench and compile the program. This application contains an enormous amount of functionality, but since it leverages MFC 2.0's reusable classes this AppWizard application contains fewer than 700 lines of C++ source code and its executable occupies only 117K.

In order to use the architectural features in MFC 2.0, you usually need to derive one or more C++ class(es) that will contain your application-specific code. AppWizard automatically derives classes for the most common MFC 2.0 components. First, AppWizard will derive an application class from CWinApp, as every MFC application requires this. Every application also has a main window, which can be either MDI or SDI, and AppWizard will provide you with a derived CFrameWnd or CMDIFrameWnd class for your application based upon your preference. AppWizard will derive a CDocument and CView class for you and install a document template so that you are automatically taking advantage of the document/view architecture. For your CDocument-derived class, AppWizard lets you specify the file type (suffix) and descriptive name for the document types, and provides intelligent defaults for all class and file names. Most MFC 1.0 users indicated they are implementing toolbar and status bar user-interface elements in their applications, so AppWizard will create these automatically if you require them. Similarly, if your application will support printing and print preview, AppWizard will add the necessary menu commands and command handlers to your application. Other options available for creating an MFC 2.0 application include context-sensitive help and OLE client support.

In addition to the required and optional classes that AppWizard provides for your application, AppWizard also creates a number of supporting files. Every Windows-based application requires a resource script, so AppWizard provides one. The supplied script includes all of the

- more -

application's menus, strings, keyboard accelerators, default icons, a default toolbar bitmap, version resource and even a dialog for the About box. The binary files such as bitmaps and icons are placed in a subdirectory, so they do not clutter your project's main directory. If your application requires OLE client or context-sensitive help support, then AppWizard will create several additional files, such as an OLE registration file or a subdirectory containing online documentation in WinHelp format.

Finally, AppWizard will create a project file (.MAK file) for use with the Visual WorkBench. This file will contain all the required compiler settings for building a debug or retail version of your application template. This project will be opened by the Visual WorkBench after AppWizard is finished, so all you need to do is build the project and you will have a running application.

After running AppWizard you will have a project directory that contains a number of files. For example, AppWizard creates a separate header file (.H) and separate source file (.CPP) for each of the classes in the application, which includes classes derived from CWinApp, CDocument, CView, and CFrameWnd (or CMDIFrameWnd). AppWizard is only used to get you started with the application framework, and is only used a single time for a given project. Since AppWizard creates only standard source files and places no restriction on how they are organized, you are free to structure your project in any manner that you see fit. Using AppWizard, you are free to work the way you are used to working. To make it easy to document the project structure, AppWizard also creates a standard text file that describes all the files in the project. It is a good practice to keep this file up to date as you add new files to your application source.

Once you have completed your application template with AppWizard, you are ready to begin adding application-specific code and user-interface elements. AppWizard will, optionally, comment your derived classes with indications of where to add specific code. The following example shows how AppWizard indicates where to add code for loading and saving a document. The class CMyDoc is the CDocument-derived class created by AppWizard.

```
void CMyDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }
}
```

Figure 6

You can use the Visual WorkBench to browse your application's source code and add your own functionality. At this point in the application development cycle you are

- more -

done with AppWizard. Another tool is provided that enables you to connect user-interface elements to your code as easily as Microsoft's Visual Basic programming system and to add classes and maintain message maps. ClassWizard, which is discussed in the next section, works with standard C++ source code to provide all of this support.

- more -

Appendix B: ClassWizard

ClassWizard ties together MFC 2.0, App Studio and the Visual Workbench, and makes it easy for you to move seamlessly between user-interface elements and application code. ClassWizard is modeled after the point-and-click interface in Visual Basic. In Visual Basic you can double-click on a user-interface element such as a menu item, and immediately edit the code for that command. ClassWizard gives you that same functionality using interfaces that are built using standard Windows resources and App Studio connected to standard C++ code and MFC 2.0. Thus you gain the benefits of moving directly between C++ code and user-interface elements as in Visual Basic, while at the same time you have the full power of industry-standard C++, MFC 2.0, as well as the Windows API. ClassWizard assists you in handling all standard Windows messages and commands by maintaining the class' message map, adding new MFC 2.0-derived classes, and maintaining all of the dialogs' data exchange code.

After you create an application template with AppWizard, you have several derived classes to customize. Normally you will be adding code for application-specific responses to standard Windows messages, and you will be adding code to handle commands for the new user-interface elements you will create with App Studio. In MFC 1.0 each CWnd-derived class had a message map that you were required to maintain manually. The MFC message-map implementation requires some macro syntax. We chose to use macros rather than implement a non-standard compiler extension so that MFC would be compiler-independent. Unfortunately MFC 1.0 programmers found it somewhat cumbersome to maintain the message-to-member function mapping. Microsoft introduced ClassWizard to automatically maintain all message maps for you, while at the same time continuing to use only standard C++. ClassWizard performs source-code maintenance exactly as you would manually, but is less error-prone and much faster. Just as with AppWizard, ClassWizard was designed to work the way you do. You can continue to organize your source code in the manner you require, without fear of losing any data, because ClassWizard works on small, well-defined portions of your source.

As an example of how ClassWizard works for you, let's consider adding a message handler for the WM_MOUSEMOVE message to a CView-derived class. Assuming a message map has already been set up, which ClassWizard does automatically, you will need to add three pieces of code to two different files: a function prototype for OnMouseMove in the .H file, a message map entry in the .CPP file, and a member function definition in the .CPP file. ClassWizard does these steps for you. In the Figure 7 all of the ClassWizard added lines are highlighted.

```
class CMyView : public CView

{
// ...

// Generated message map functions
protected:
    //{{AFX_MSG(CMyView)
    afx_msg void OnFilePrint();
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
    afx_msg void OnMouseMove(UINT nFlags, CPoint point);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

BEGIN_MESSAGE_MAP(CMyView, CView)

    //{{AFX_MSG_MAP(CMyView)
            ON_COMMAND(ID_FILE_PRINT, OnFilePrint)
    ON_WM_LBUTTONDOWN()
    ON_WM_LBUTTONUP()
    ON_WM_MOUSEMOVE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

void CMyView::OnMouseMove(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    CView::OnMouseMove(nFlags, point);
}
```

So as you can see from this example, ClassWizard saves you programming and editing time, greatly reduces the number of potential errors in your code, and removes the bookkeeping responsibility associated with messages and commands. As with AppWizard, ClassWizard uses comments to indicate where you should insert code and any special notes for a particular message. The OnMouseMove also contains a call to the base-class message handler, which will perform any default processing of the message if required. Since ClassWizard is smart about which message you are handling, it will automatically insert message-specific code in the handler stub, such as calling the base-class handler when needed. Using this context-based intelligence, ClassWizard filters the list of messages that a given user-interface object or class can handle, based on the type of the object. The comments containing *AFX_MSG* delimit the portions of your code that ClassWizard will modify. ClassWizard will **never** modify any code outside of these comments. In fact, if you wish to prevent ClassWizard from modifying a particular message handler or message map entry, you are free to move it outside of the comment blocks. By using these standard C++ comments as indicators to ClassWizard, your code can be maintained as standard C++ code without any secondary files or additional database, and without parsing C++ code. ClassWizard's main function is to assist the programmer with managing user-

- more -

interface to C++ code mapping, which utilizes the MFC message-map structure and the MFC 2.0 command architecture.

In addition, ClassWizard also makes it easy to add new classes that are derived from the MFC 2.0 class CCmdTarget, or any of its derived classes. With ClassWizard you can create a new class derived from any of the basic Windows classes such as CWnd, CFrameWnd CMDIChildWnd and CDialog. ClassWizard also lets you create new classes derived from the CView and CDocument architectural classes, which makes it easy to add a new view type to your existing application. ClassWizard can derive new classes based on the high-level abstractions CFormView, CScrollView and splitter windows. Each class you create with ClassWizard is a standard C++ class that uses the MFC 2.0 base class. ClassWizard allows you to specify the header file and implementation file for a class, and supports maintaining multiple classes in a single file.

If you have an MFC 1.0 application that you wish to use with ClassWizard, all you need to do is add the AFX_MSG comments described above and use the ClassWizard Import Class feature, which interprets the class declaration. Migrating your classes to ClassWizard is discussed in the documentation.

ClassWizard, in conjunction with App Studio, supports several additional features for managing CDialog-derived classes. When you are editing a dialog resource in App Studio, you can invoke ClassWizard and automatically create a C++ class for that dialog if one does not exist. Thus without typing any code you can connect a dialog resource to a C++ class derived from an MFC 2.0 base class. Also, for dialogs (as well as CFormView classes, since they are based on dialog resources) ClassWizard can be used to implement DDX/DDV functionality for initializing, validating and obtaining information from end-user dialogs. ClassWizard's Edit Variables command lets you attach class member variables to each control in the dialog. Since ClassWizard understands all of the Windows controls, the variables you add are correctly typed. For example, if your dialog contains a checkbox control, ClassWizard knows that it should be represented by a Boolean variable in the CDialog-derived class. The application framework uses these variables to initialize the state of the controls in your dialog. While the dialog is visible to the user, MFC 2.0 will validate any information entered by the user based on criteria specified using ClassWizard. If the user clicks the OK button, MFC 2.0 will automatically transfer the contents of the controls to these variables, which can then be used by your code as needed. ClassWizard handles all of this DDX/DDV functionality for dialogs without any additional coding.

#########

- more -