Introduction to MFC

Microsoft Foundation Classes

Where we're headed…

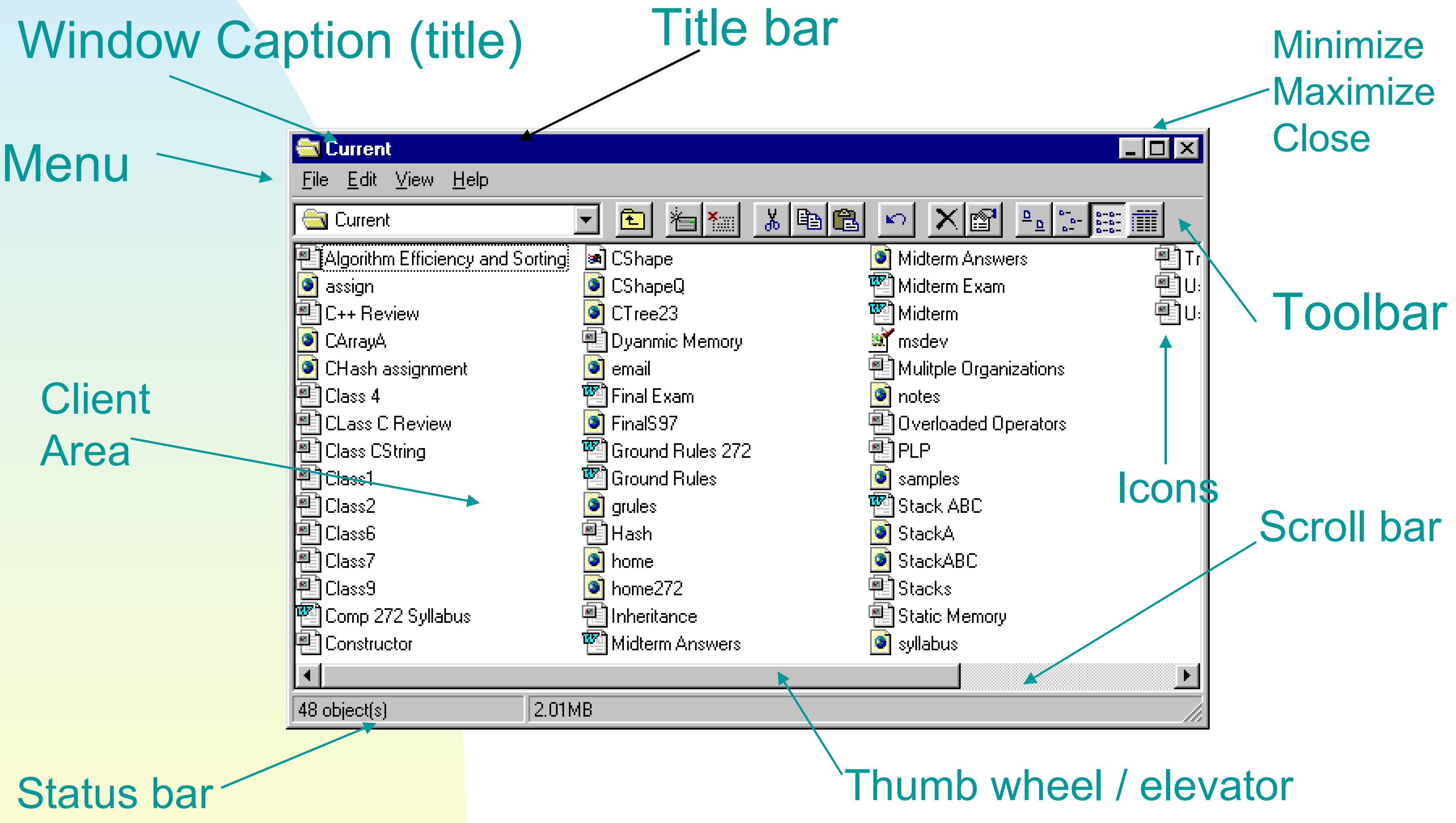
- What's that GUI stuff all about?
- Event-driven programming
- MFC History
- Win32 API
- MFC
- Message Handling
- Class Types & Hierarchy
- Different kinds of windows
- MVC (Doc/View Architecture)
- Dialog based vs. SDI/MDI
- Form elements (buttons, controls, etc.)
- GDI, DC, and Drawing
- **NOTE: The labs are mandatory** with these lectures; the slides present the theory and the labs provide tutorials on the implementation!

User Interfaces (UI)

- The UI is the connection between the user and the computer
  - Command line/console
    - Text based
  - Graphical User Interface (GUI)
    - Visually oriented interface (WYSIWIG)
    - User interacts with graphical objects
    - More intuitive
  - Other UIs: optical, speech-based, etc.

Main GUI Feature

- THE WINDOW!

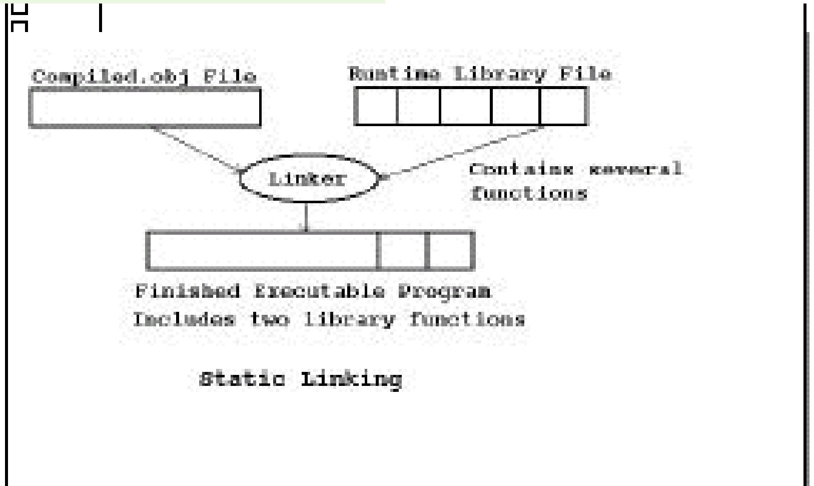


A Gooey History of GUI

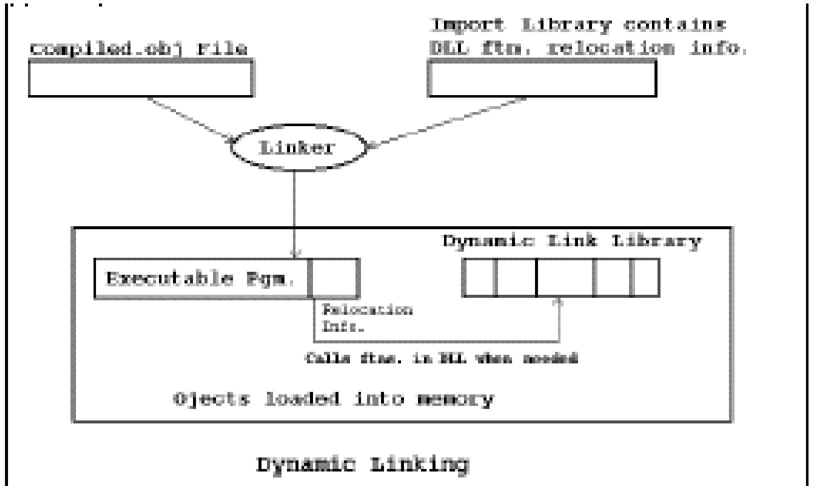
- 1968 → ARPA (Advanced Research Projects Agency) funded Stanford Research Center (Doug Englebart)
  - First windows, light pen, and mouse
- 1970-1972 → Xerox PARC (Palo Alto Research Center) produces Alto and Star
  - First GUI, WYSIWIG, Ethernet, Laser Printers, Postscript, etc.
- 1983 → Apple Lisa
- 1984 → Apple Macintosh
- 1985 → Windows 1.0 (Win16)
- 1987 → Windows 2.0
- 1990 → Windows 3.0
- 1993 → Windows NT followed by Win95, Win98, Win2k, WinXP, etc. → ALL are Win32

Other GUI OSs

- OS/2
- XWindows (OS independent)
- Commodore Amiga
- Atari GEM
- And many others like MenuetOS



Compiled.obj File
Runtime Library File
Linker
Contains several functions
Finished Executable Program
Includes two library functions
Static Linking



compiled.obj File
Import Library contains DLL ftns. relocation info.
Linker
Dynamic Link Library
Executable Pgm.
Relocation Info.
Calls ftns. in DLL when needed
Objects loaded into memory
Dynamic Linking

No ANSI Standard for GUI

- ANSI/ISO C++ does <u>not</u> provide capabilities for creating graphical user interfaces (GUI)
- MFC
  - A large collection of classes (and framework) that help Visual C++ programmers create powerful Windows-based applications
- Microsoft library documentation is available at: http://msdn.microsoft.com/library/

Gooey User Interaction

- Users interact with the GUI via messages
- When a GUI event occurs the OS sends a message to the program
- Programming the functions that respond to these messages is called **event-driven programming**
  - Messages can be generated by user actions, other applications, and the OS

Console vs. event-driven programming

- GUI programs have a fundamentally different structure than console-based programs
- Console-based program:

```
ask user for some input;
do some processing;
print some output;
ask user for some more input;
etc.
```

  - The application programmer controls when input and output can happen
- GUI program model: the user is in control!

Event-driven programming

- Structure GUI programs to respond to user *events*
- Events are: mouse clicks, mouse moves, keystrokes, etc.
  - in MFC parlance, usually called *messages*
- Main control structure is an event loop:

```
while (1) {
  wait for next event
  dispatch event to correct GUI component
 }
```

  - this code is always the same, so it's handled by MFC
- You just write the code to respond to the events.
  - functions to do this are called *message handlers* in MFC
- GUI model is: user should be able to give any input at any time → **Non-Sequential**!

Windows GUI Programming

- Program directly using the **API** (Application Programming Interface)
- An API is a *library* that provides the functions needed to create Windows and implement their functionality
- Use libraries that encapsulate the API with better interfaces → e.g., MFC, FCL, JFC, GTK, Tk (with TCL or Perl), Motif, OpenGL, QT, etc.
  - Cross-platform: JFC, wxWindows, or Wind/U

How can we use the API?

- Event-driven and graphics oriented
- How does it work?
- Suppose a user clicks the mouse in the client area:
    - Windoze decodes HW signals from the mouse
    - Figures out which window the user selected
    - Sends a message (an event) to the program that generated that window
    - Program reads the message data and does the corresponding function
    - Continue to process messages (events) from Windoze → The Message Loop

Overview of a Win32 API Program

- The loader looks for a `WinMain()` function as the point of entry, as opposed to the regular `main()`. `WinMain()` does the following (in C, not C++):
  - Variable declarations, intializations, etc.
  - Registers the window class (a C-style structure; **NOT** a C++ class (implementation of an ADT))
  - Creates a window based on that registered class
  - Shows the window & updates the window's client area
  - Enters the message loop
    - Looks for messages in the applications message queue (setup by the Windoze OS)
    - Blocks if there isn't one (basically does nothing and just waits for a message to appear)
    - When a message appears, it translates and dispatches the message to the window the message is intended for
      - Forwards to the correct callback message-processing function

Other stuff needed by an API program…

- The program file also contains a function called `WndProc()`, which processes the messages sent to that application
    - In other words, it listens for certain messages and does certain actions when it receives them (using a gigantic switch/case statement)
- Buttons, dialogs, etc. are all defined in resource script (`.rc`) files
    - Compiled by a separate “Resource Compiler”
- These resources are then linked into the code in your main `.cpp` program file (which houses your `WinMain()` and `WndProc()` functions)

What is MFC?

- A full C++ API for Microsoft Windows programming.
- Object-oriented *framework* over Win32.
- Provides a set of *classes* allowing for an easy application creation process.
    - It encapsulates most part of the Windows API, which is <u>not</u> class-based.
        - Win32 API is a C-based library of functions / types

History of MFC/Win32 API

- Turbo Pascal and IDE
- Turbo C and Quick C
- Microsoft C version 7: C/C++

  plus MFC 1.0
- Visual C++ 1.0
- Visual C++ 2,3
- Visual C++ 4, 5, 6
- .NET (Visual Studio 7)

What *isn't* MFC?

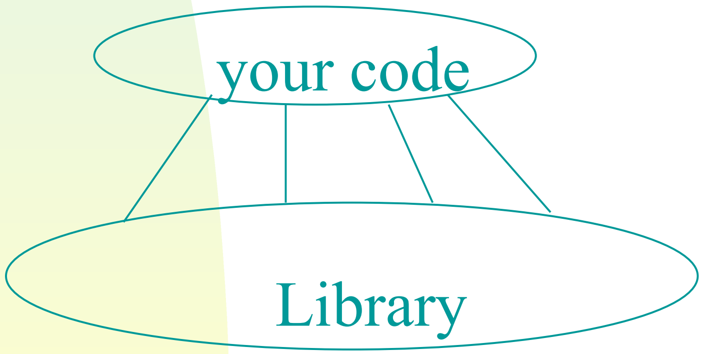
- MFC isn't a simple *class library*.
  - An ordinary library is an isolated class set designed to be incorporated in any program.
- Although it can be used as a library…
- MFC is really a *framework*.
  - A framework defines the structure of the program.

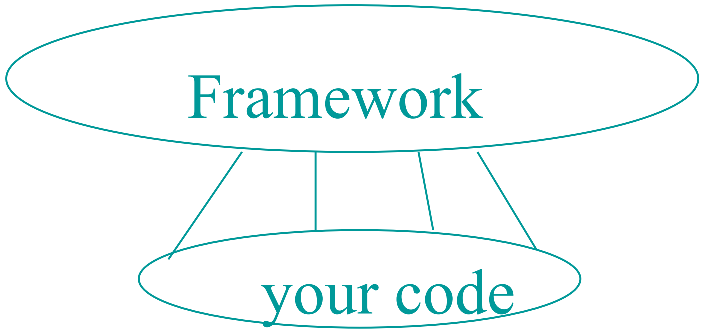
GUI Libraries

- GUI programs involve a lot of code.
- But for many different applications much of the code is the same.
  - A "class library" is a set of standard classes (including properties and methods) for use in program development
- The common code is part of the library. E.g.:
  - getting and dispatching events
  - telling you when user has resized windows, redisplayed windows, etc.
  - code for GUI components (e.g. look and feel of buttons, menus, dialog boxes, etc.)
- But the library is upside-down: library code calls *your* code.
  - this is called an ***application framework***
- You can also call library code

Application Frameworks

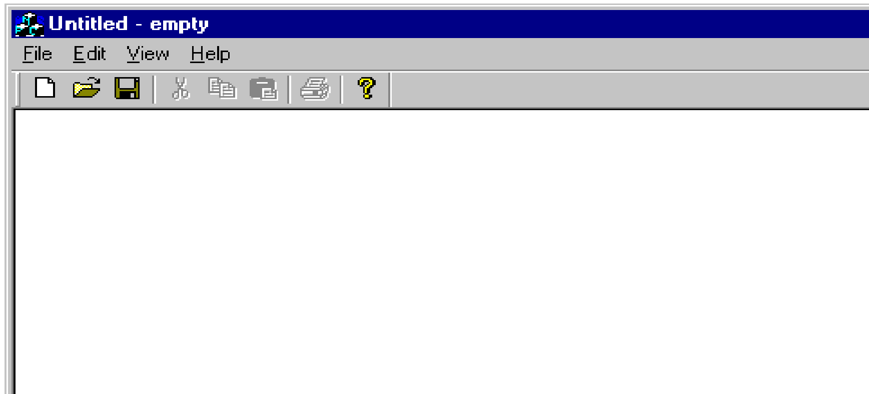
- Sometimes called *software architectures*
- Reusable software for a particular domain of applications:
  - general purpose set of classes with pure virtual functions as hooks for more specific versions
  - plus, protocol for using the virtual functions:
    - overall control structure given by framework code
  - application writer supplies exact functionality of the methods
- Contrast with a simple class library:





MFC vs. other libraries

- All GUI libraries are top-down like this.
- Using an OO language means we can employ class reuse, templates, and polymorphism.
- MFC provides more in the framework than some other smaller GUI libraries.
  - e.g. “empty” application, get a bunch of menus, and a toolbar → MFC provides **skeleton** code for your application
  - richer set of components: color-chooser dialog, file browser, and much more.
  - widely adopted (used by everyone)

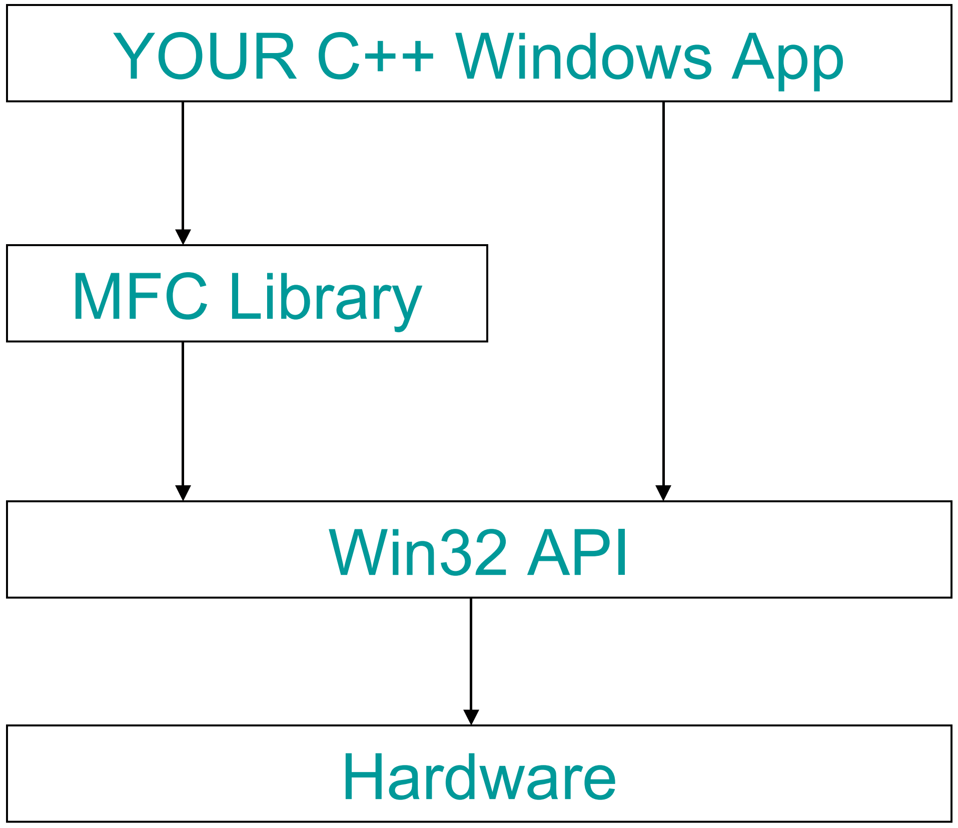


Application Framework Pros & Cons

- Advantages to application framework:
  - less code for you to write:
    - application gets built faster & including less low-level tedious code
  - more reuse of code between applications:
    - we can focus on what's different about our application
  - uniform look and feel to applications produced
    - less frustration for users/customers
- Disadvantages to application framework
  - larger learning curve
  - may produce a slower application
  - may be harder to do exactly what you want
    - e.g., you want a different look-and feel, or you want a new component

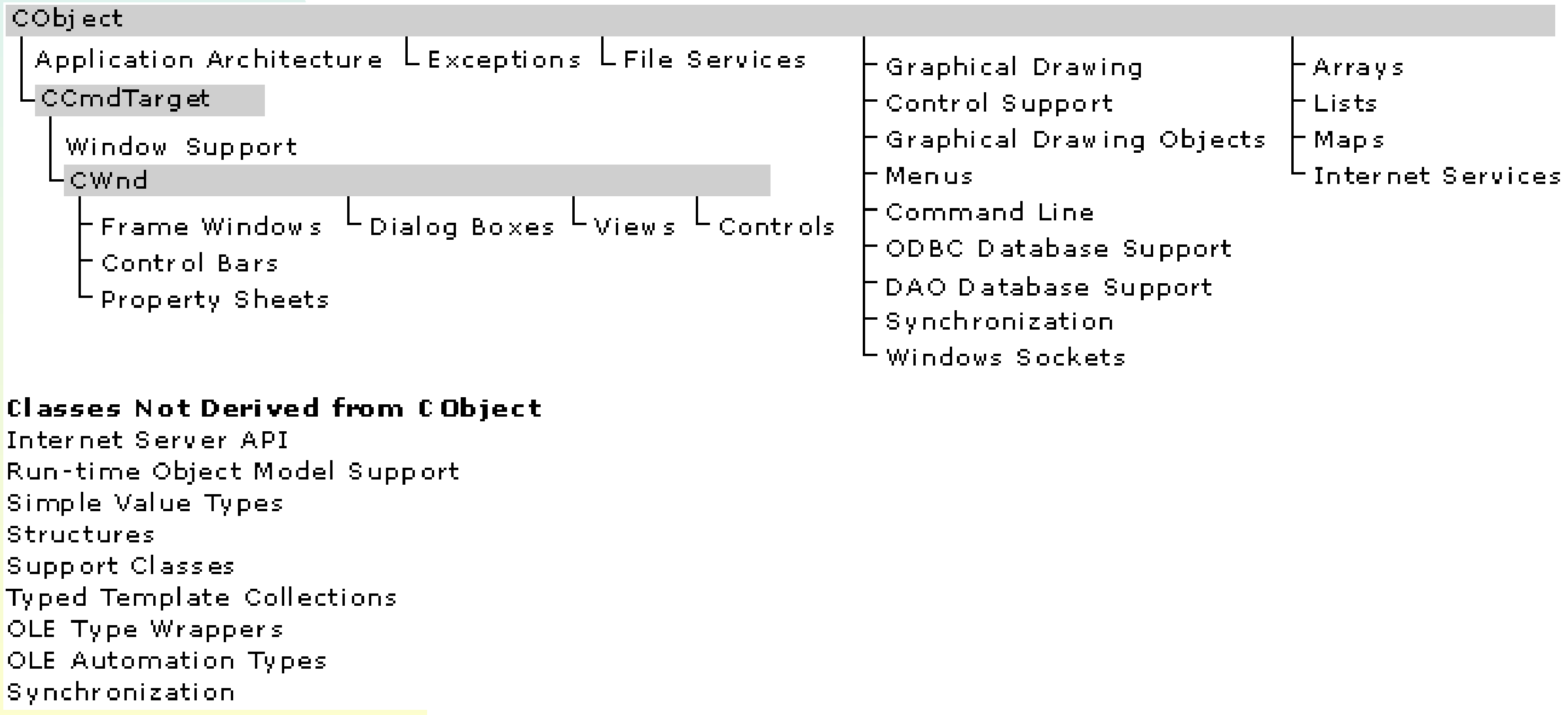
Programming Windoze Applications

- Use either Win32 API or MFC!



How to use MFC

- Derive from base classes to add functionality
- Override base class members
  - Add new members
- Some classes can be used directly



The “Main” Objects

- CObject is the base class from which all other MFC classes are derived
- CWnd is the base class for all the window types and controls
- CDialog, CFrameWnd, and CWinApp are the primary classes used in applications
  - CDialog and CFrameWnd encapsulate the functionality for creating windows
  - CWinApp encapsulates the functionality for creating and executing the Windows applications themselves
  - You need objects derived from **both** kinds of classes in order to create a functional MFC application

CWinApp

- CWinApp class is the base class from which you always derive a windows application / system object.
- Each application that uses the MFC classes can only contain one object derived from CWinApp.
- CWinApp is declared at the global level. Creating an instance of the application class (CApp) causes:
  - WinMain() to execute (it's now part of MFC [WINMAIN.CPP]) which does the following:
    - Calls AfxWinInit(), which calls AfxRegisterClass() to register window class
    - Calls CApp::InitInstance() [virtual function overridden by the programmer], which creates, shows, and updates the window
    - Calls CWinApp::Run(), which calls CWinThread::PumpMessage(), which contains the GetMessage() loop
      - After this returns (i.e., when the WM_QUIT message is received), AfxWinTerm() is called, which cleans up and exits

The one and only CWinApp

- Derived from CObject → CCmdTarget → CWinThread
  - CObject: serialization and runtime information
  - CCmdTarget: allows message handling
  - CWinThread: allow multithreading, the CWinApp object is the primary thread
- Derive one CWinApp-based class and then declare one instance of that class per application
- Encompasses `WinMain()`, message pump, etc. within the CWinApp-derived object

CWnd and CFrameWnd

- Holds a HWND (handle to a window) and all of the functions associated with it
  - A handle is a strange type of pointer (structure)
- Hides the `WndProc()` function and handles all the message routing
- CWnd is the base class for everything from CButton to CFrameWnd
  - CFrameWnd is the class used as the main window in most applications

CWnd (cont.)

- Two-stage initialization of CWnd objects:
  1. constructor inits C++ object (m_hWnd is NULL).
  2. `Create` func inits Windows object inside the C++ object
     - (can't really use it until second step happens.)
- In Doc-View model
  - CView ISA CWnd
  - CMainFrame ISA CWnd
- All CWnd objects can receive events (messages)

CObject

- **Serialisation**; the ability to load and save the object to /from structured permanent storage
- **Runtime Class Information;** the class name its position in the hierarchy can be extracted at run time.
- **Diagnostic Output**; ability if in debug mode to use trace info in debug window
- **Compatibility**; all objects must be a member of the MFC collection itself.
- There are several non-CObject-inherited classes.
  - This is because CObject defines five virtual functions. These functions are annoying when binary compatibility with existing C data types is needed

Minimal MFC Program

- Simplest MFC program just needs two classes, one each derived from CWinApp and CWnd
  - An application class derived from CWinApp
  - This class will define the application and provide the message loop
  - A window class usually derived from CFrameWnd which defines the applications main window
- Use #include <afxwin.h> to bring in these classes
  - Also includes some standard libraries
- Need a resource file (.rc) if you want it to be dialog based or include dialogs

Simplified `WinMain`

```
int AFXAPI AfxWinMain( ) {
  AfxWinInit( );
  AfxGetApp( )->InitApplication( );
  AfxGetApp( )->InitInstance( );
  AfxGetApp( )->Run( );
}
```

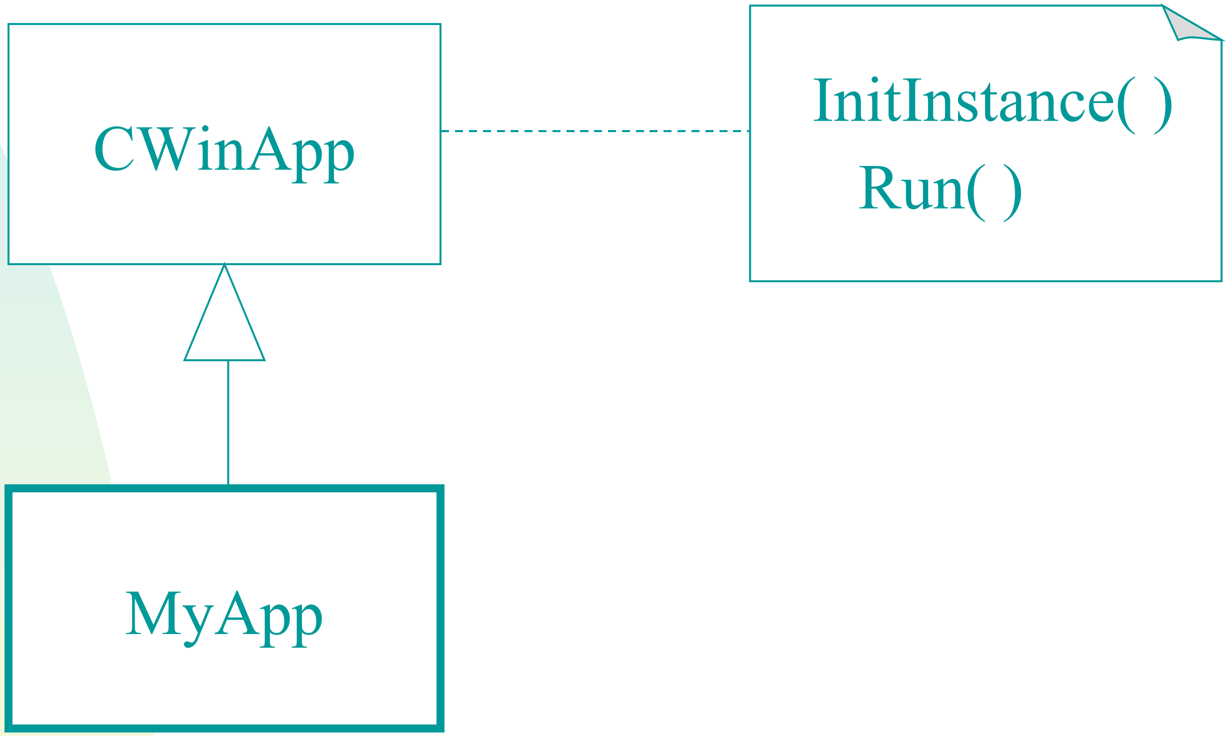
Some Operations in `CWinApp`

```
virtual BOOL InitApplication( );
virtual BOOL InitInstance( );
virtual int Run( );
Virtual int ExitInstance( );
```

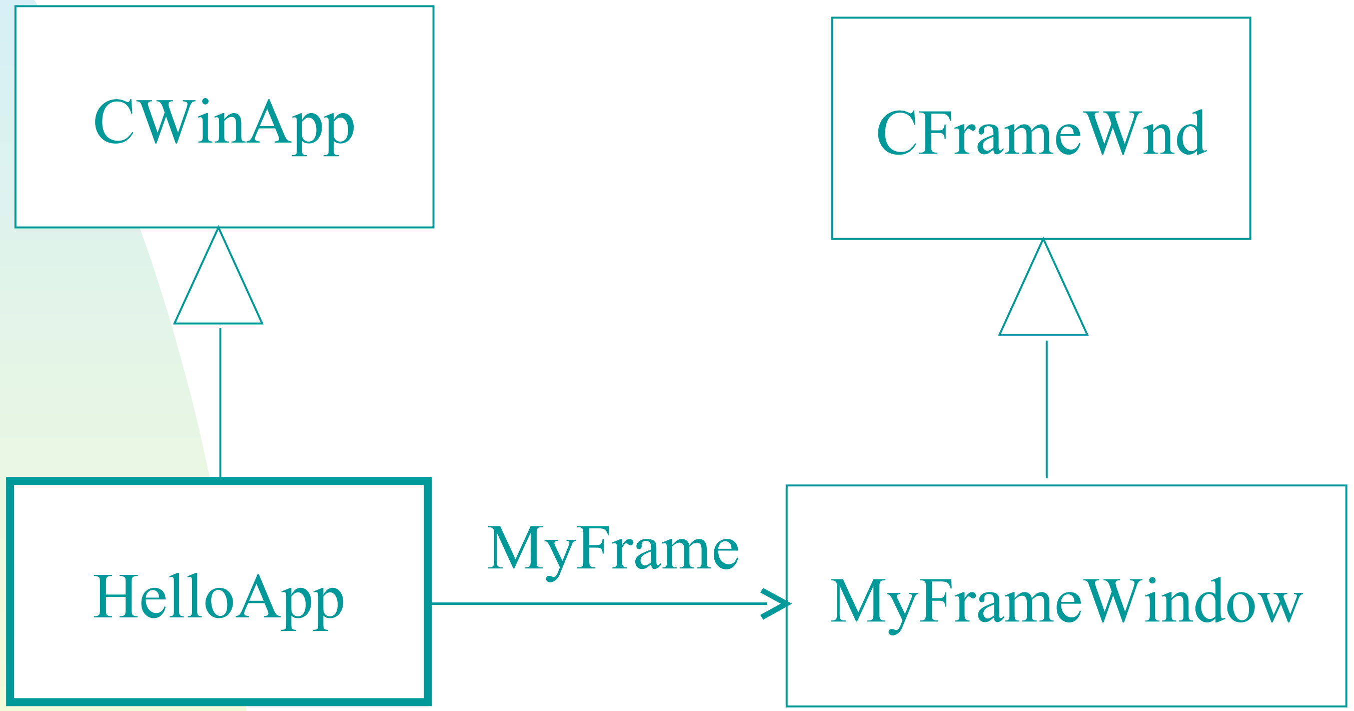
Simplified CWinApp::Run( )

```
int CWinApp::Run( ) {
  for( ; ; ) {
     //check to see if we can do
  // idle work
     //translate and dispatch  //
  // messages
  }
}
```

Template Method Design Pattern



First Window Program



Message map

- The *message map* for a class maps
  - **messages** (e.g., `WM_LBUTTONDOWN`, `WM_PAINT`)

  to

  - **message handlers** (e.g., `CMyView::OnLButtonDown`, `CMyView::OnPaint`)
- Virtual function-*like* mechanism
  - Can “inherit” or “override” message handlers
  - But does not use C++ virtual function binding
  - Space-efficient implementation
- We use macros which generate the code for this mechanism.

Class MyFrameWindow

```
#include <afxwin.h>
class MyFrameWindow : public CFrameWnd {
public:
  afx_msg void OnPaint( ) {
     CPaintDC paintDC( this );
     paintDC.TextOut( 0, 0, “Hello world!” );
  }
  DECLARE_MESSAGE_MAP( )
};
```

Message Map and Class HelloApp

```
BEGIN_MESSAGE_MAP(MyFrameWindow, CFrameWnd)
  ON_WM_PAINT( )
END_MESSAGE_MAP( )

class HelloApp : public CWinApp {
public:
  HelloApp( ) : CWinApp( “Hello World!”) { }
  BOOL InitInstance( );
} theApp;
```

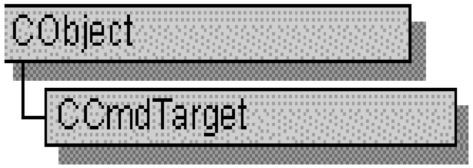
Method `InitInstance`

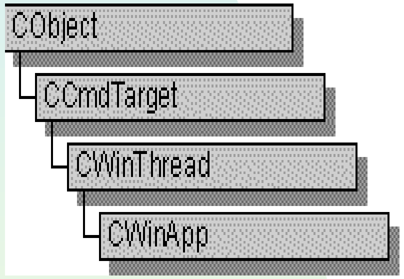
```
BOOL HelloApp::InitInstance( ) {
   CFrameWnd * MyFrame = new MyFrameWindow;
   m_pMainWnd = MyFrame;
   MyFrame->Create(NULL,  (LPCTSTR)”Hello”);
   MyFrame->ShowWindow( SW_SHOW );
   return TRUE;
}
```

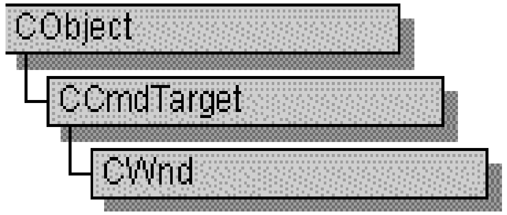
CPaintDC

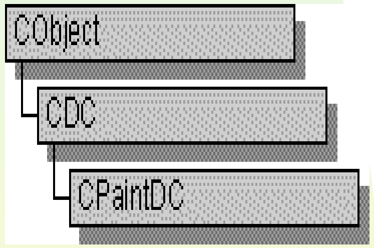
- The CPaintDC class is a device-context class derived from CDC
- The CDC class defines a class of device-context objects
  - All drawing is accomplished through the member functions of a CDC object
- Although this encapsulation aids readability, it offers very little improvement on the basic GDI in the native API's

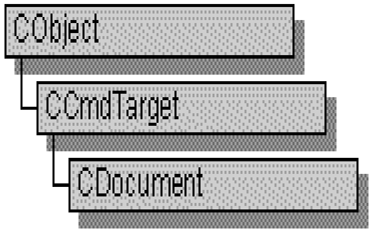
Some Typical Structures

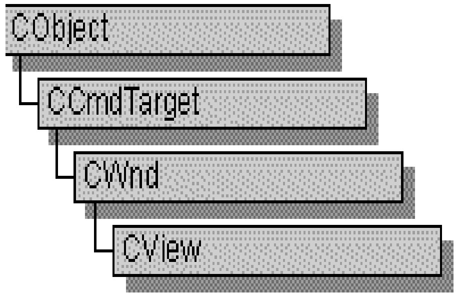












Library itself can be broadly categorised

- General purpose: strings, files, exceptions, date/time, rectangles, etc.
- Visual Objects: windows, device context, GDI functions, dialogs, etc.
- Application architecture: applications, documents (the data), views (on the data), etc.
- Collections: lists, arrays, maps, etc.
- Other specific classes: OLE, ODBC, etc.

MFC Global Functions

- Not members of any MFC Class
- Begin with Afx prefix
- Some important Global Afx func:
  - AfxMessageBox() – message box
  - AfxAbort() – end app now
  - AfxBeginThread() – create and run a new thread
  - AfxGetApp() – returns ptr to application object
  - AfxGetMainWnd() – returns ptr to apps main window
  - AfxGetInstanceHandler() – returns handle to apps current instance
  - AfxRegisterWndClass() – register a custom WNDCLASS for an MFC app

Message Maps

- Each class that can receive messages or commands has its own "message map"
- The class uses message maps to connect **messages** and commands **to** their **handler** functions
- Message maps are setup using MFC macros
  - They're essentially lookup tables and they replace the gigantic switch/case statement used in the API

Message Handling

- In MFC, the message pump and `WndProc()` are hidden by a set of macros
  - MFC messaging acts virtually without the overhead
- To add message routing put the following line in the class *declaration*,
  `DECLARE_MESSAGE_MAP()`
- For every message you want to handle you have to add an entry to the message map itself (appears outside the class declaration)

Adding to a message map

To have your class do something in response to a message:

1. Add **`DECLARE_MESSAGE_MAP`** statement to the class *declaration*
2. In the *implementation* file, place macros identifying the messages the class will handle between calls to **`BEGIN_MESSAGE_MAP`** and **`END_MESSAGE_MAP`**
   - need to tell it the class name and the superclass name
     - Example:

```
BEGIN_MESSAGE_MAP(CHello2View, CView)
   ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()
```

3. Add member functions to *handle* the messages (uses fixed naming scheme). Here's an example function prototype:

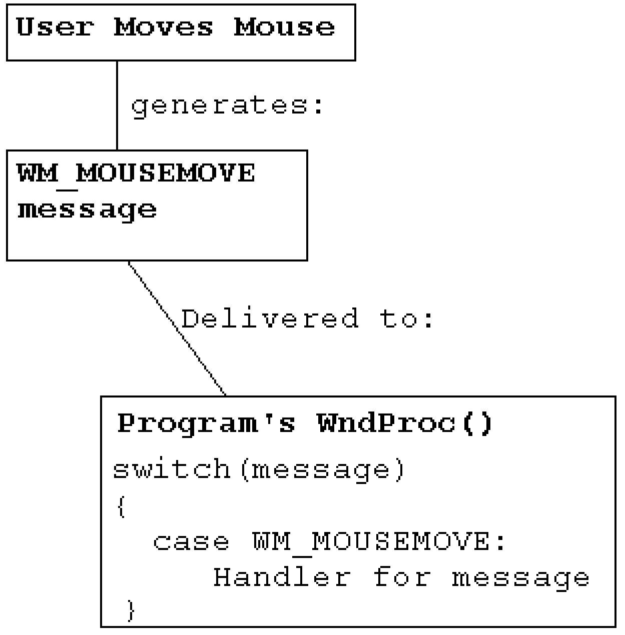
```
afx_msg void OnLButtonDown(UINT nFlags,
                              CPoint point);
```

Message maps (cont.)

- Class wizard can add this code for us.
  - Invoke class wizard from View menu (go to **Message Maps** tab)
    - choose what class will respond to the event
    - choose the message to respond to
    - can go right from there to editing code . . .
      - We have to write the body of the handler
        - (What do you want to happen on a left mouse click?)

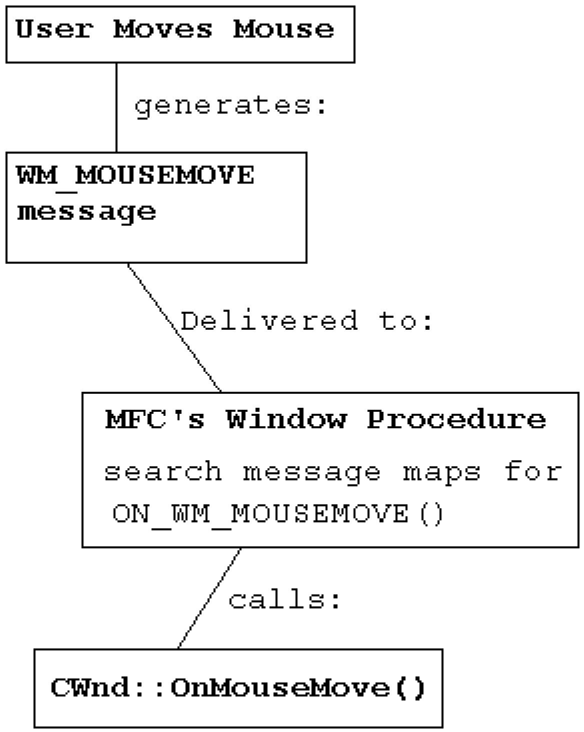
A Typical Message Map

```
BEGIN_MESSAGE_MAP(CMyWnd, CFrameWnd)
      ON_COMMAND( IDM_EXIT, OnExit )
      ON_COMMAND( IDM_SHOW_TOTAL, OnShowTotal )
      ON_COMMAND( IDM_CLEAR_TOTAL, OnClearTtotal )
END_MESSAGE_MAP()
```



```
switch(message)
{
  case WM_MOUSEMOVE:
     Handler for message
}
```

Win32 API Message Handling



MFC Message Handling

Document /Views

- Up to this point, looking at the classes that are the basis of an application, MFC can still be considered simply as wrappers for C++ around the basic 'C' API calls.
- CWinApp offers the control of the application.
- Start-up $\longrightarrow$ Execution $\longrightarrow$ Termination

Model-View-Controller Architecture

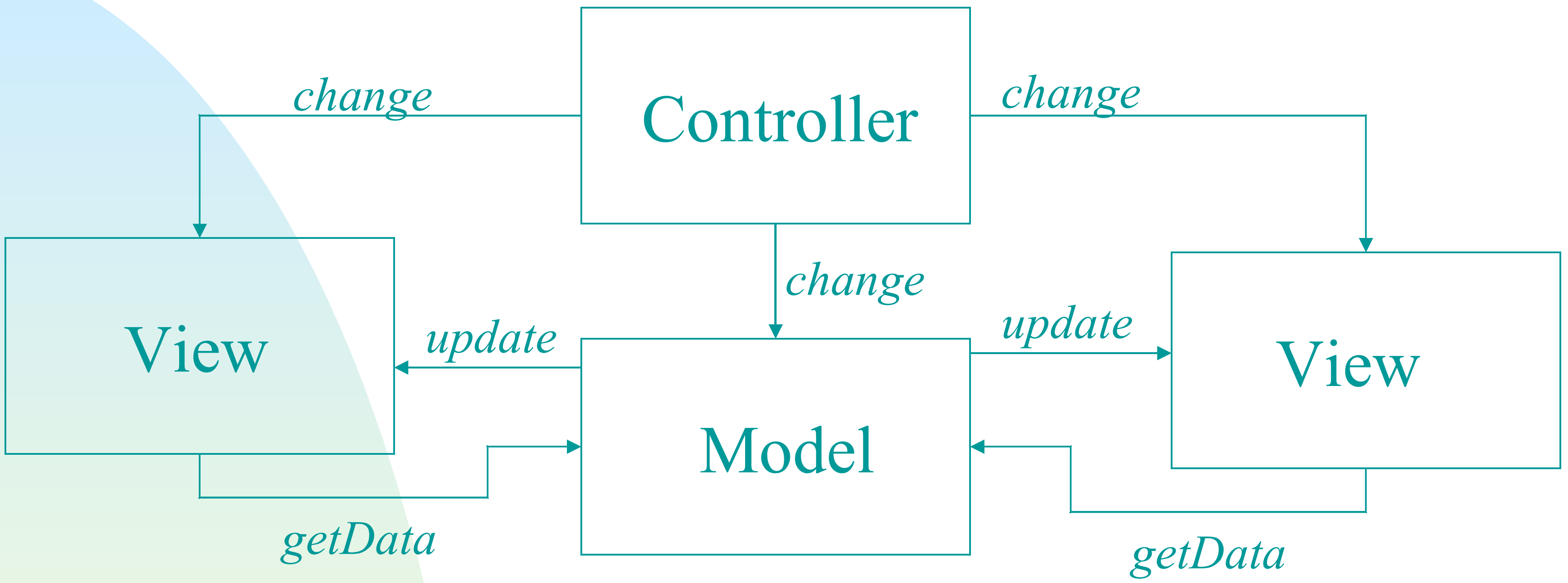
- Model-View-Controller (MVC)
  - example of an OO design **pattern**
  - started with Smalltalk
  - a way to organize GUI programs
- Main idea: separate the GUI code from the rest of the application.
- Why?
  - more readable code
  - more maintainable code
  - more details later

MVC (cont.)

- *Model* classes maintain the data of the application
- *View* classes display the data to the user
- *Controller* classes allow user to
  - manipulate data in the model
  - or to change how a view is displayed
- Modified version: controllers and views combined

  (MFC does this)

MVC structure

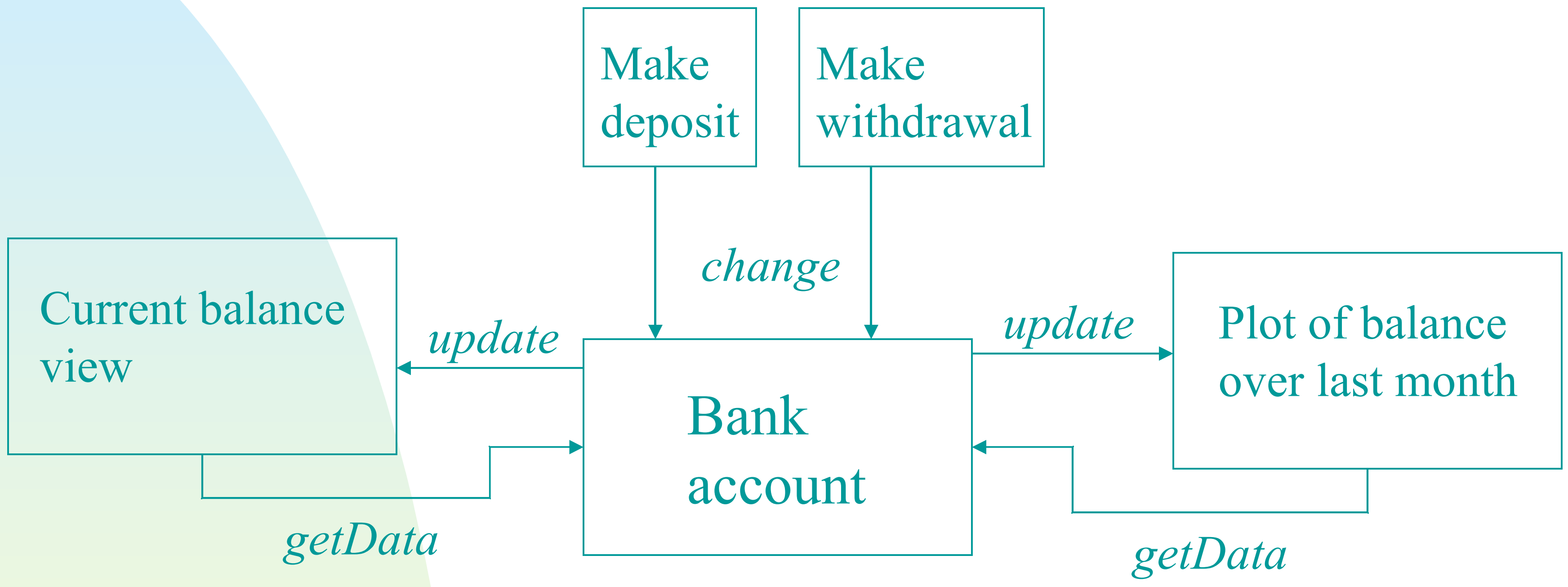


model
maintains
data

view
displays current
state to user

controller
user manipulates
data in a model
or how view displayed

MVC example: Bank account



Document-view architecture

- In MFC version of Model-View-Controller:
  - Models are called Document objects
  - Views and Controllers are called View objects
- Example: in Microsoft Word
  - Views:
    - multiple windows open displaying same document
    - different types of views (normal, page layout, outline views)
  - Document:
    - same data regardless of the view above
    - contains text/formatting of Word document

Benefits of Document/View

- Recall organization:
  - GUI stuff is in View classes
  - non-GUI stuff is in Document (and related) classes
- Benefits: modifiability and readability
  - Can add new Views fairly easily
    - would be difficult if data were closely coupled with its view
    - Examples:
      - spreadsheet: have a grid of cells view; add a bar graph view
      - target a different platform (with different GUI primitives)
  - Can develop each part independently
    - clear interface between the two parts

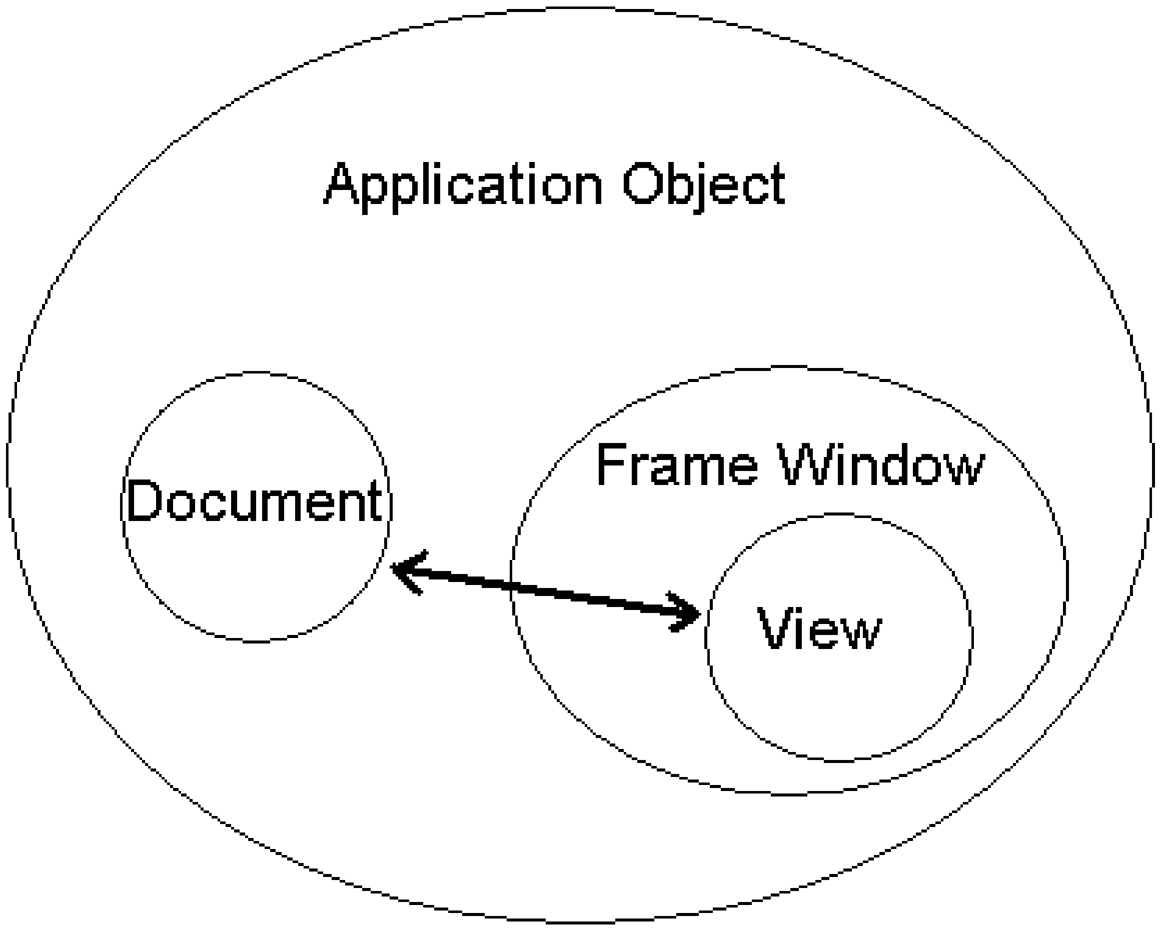
What is Doc/View?

- The central concept to MFC
- Provides an easy method for saving or archiving objects
  - Takes the pain out of printing - uses the same code that’s used to draw to the screen
- A possible downfall for apps that want MFC, but not Doc/View

Document /Views

- The concept of documents and views builds this into a framework.
- The CDocument class provides the basic functionality for user-defined document classes.
- Users interact with a document through the CView object(s) associated with it.
- The CFrameWnd class provides the functionality of a Windows single document interface (SDI) overlapped or pop-up frame window, along with members for managing the window.
- It is within these CFrameWnd that we will normally derive the CView onto our CDocument data.

Application Framework



The Document

- Derived from CDocument
- Controls application data
  - provides a generic interface that encapsulates data
- Loads and stores data through serialization (saving objects to disk)

The View

- Attached to the document
- Acts as an intermediary between the document and the user
- Derived from CView
    - Document/View Relationship is One-to-Many.
        - Data can be represented multiple ways

The Main Frame Window

- The main, outermost window of the application
- Contains the views in its client area
- Displays and controls the title bar, menu bar, system menu, borders, maximize and minimize buttons, status bar, and tool bars

The Application Object

- We have already discussed CWinApp
  - Calls WinMain(), contains main thread, initializes and cleans up application, and dispatches commands to other objects
- In Doc/View it manages the list of documents, and sends messages to the frame window and views

Document/View Concept

- The document object is responsible for storing your program’s data
- The view object is responsible for displaying program data and usually for handling user input
- Your application document class is inherited from CDocument, the view is inherited from CView

Document Object

- Stores application data such as:
  - Text in a word processing application
  - Numeric values in a tax application
  - Shape characteristics in a drawing program
- Handles commands such as
  - Load, save, save-as, new

View object

- Displays document data and typically allows users to enter and modify data
  - Can have more than one view (as in Excel)
  - May only show part of the data
    - Data may be too large to display
    - May only display a certain type of data
  - Principle graphic user interface
- Handles most commands especially
  - Paint (draw), Print, Inputs (WM_CHAR) etc.

Communication

- View requires access to the document
  - GetDocument() method of your view clas returns a pointer to the document object
  - Can be used in any view method
- Does the document need access to the view?
  - No, not really. It only needs to tell the view that the data has been updated ***UpdateAllViews*** method does this

Review

- Store your data in your document class
- Put your input and drawing (paint) code in your view class
- Get a pointer to your data by using the GetDocument method

SDI and MDI

- MFC has two flavors of applications
  - SDI = Single document interface
  - MDI = Multiple document interface
- Examples
  - Word uses MDI
    - can have multiple documents open simultaneously
    - may see multiple smaller windows in the larger window
  - Notepad uses SDI
    - only can have one document open at a time
    - view fills up frame of window
- We’ll focus on SDI

SDI classes

- Every SDI application has the following four classes:
  - CApp
  - CDoc
  - CView
  - CMainFrame
- Our application will have classes derived from these classes
  - AppWizard will create them automatically when we ask for an SDI MFC application
- The relationship between these classes is defined by the framework.

Four classes of SDI Application

- Instances:
  - Always one App
  - Always one MainFrame
  - Always one Document
  - May have multiple views on Same Document
- Key part of learning MFC:
  - familiarize yourself with these four classes
  - learn what each one does
  - learn when and where to customize each of them

Examples of Customization

- Views
  - OnDraw handles most output (you write; MFC calls)
  - respond to input (write message handlers; MFC calls them)
- Document
  - stores data
  - most of the (non-GUI) meat of the application will be in this object or objects accessible from here
- CMainFrame
  - OnCreate is used to set up control bars
  - (rarely need to customize in practice)
- CWinApp
  - can use to store application-wide data
  - (rarely need to customize in practice)

SDI or MDI?

- SDI - Single Document Interface
  - Notepad
- MDI - Multiple Document Interface
  - Word, Excel, etc.

AppWizard and ClassWizard

- AppWizard is like the Wizards in Word
  - Gives a starting point for an application
- Derives the four classes for you
  - SDI, MDI   (even dialog, or non-Doc/View)
- ClassWizard helps you expand your AppWizard generate application
  - ex. Automatically generates message maps

Other AppWizard classes

- App class
  - A class that instantiates your application
- Key responsibilities
  - Can the application start(multiple copies OK?)
  - Load application settings (ini, registry…)
  - Process command line
  - Create the document class
  - Create and open the mainframe window
  - Process about command

Mainframe class

- Container for the view window
- Main tasks
  - Create toolbar and status bar
  - Create the view window

CAboutDlg

- Small class to display the about dialog
- Rarely modified (only for Easter eggs or special version display information)

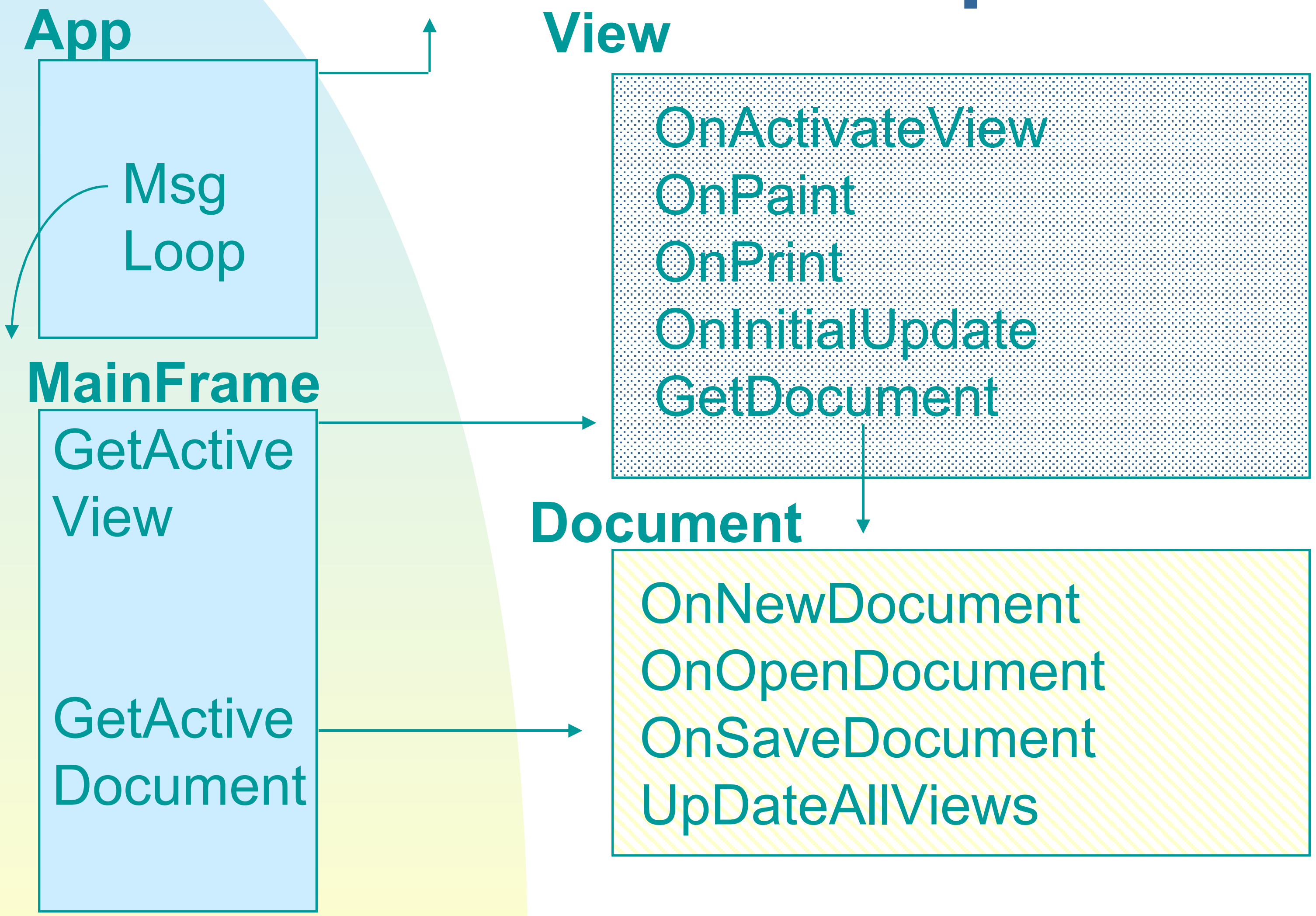
Other classes

- Document Template class
  - Container for all the documents of the same type. Normally one per application.
  - CSingleDocTemplate for SDI
  - CMultiDocTemplate for MDI
- Other classes
  - For dialogs, Active X objects etc.

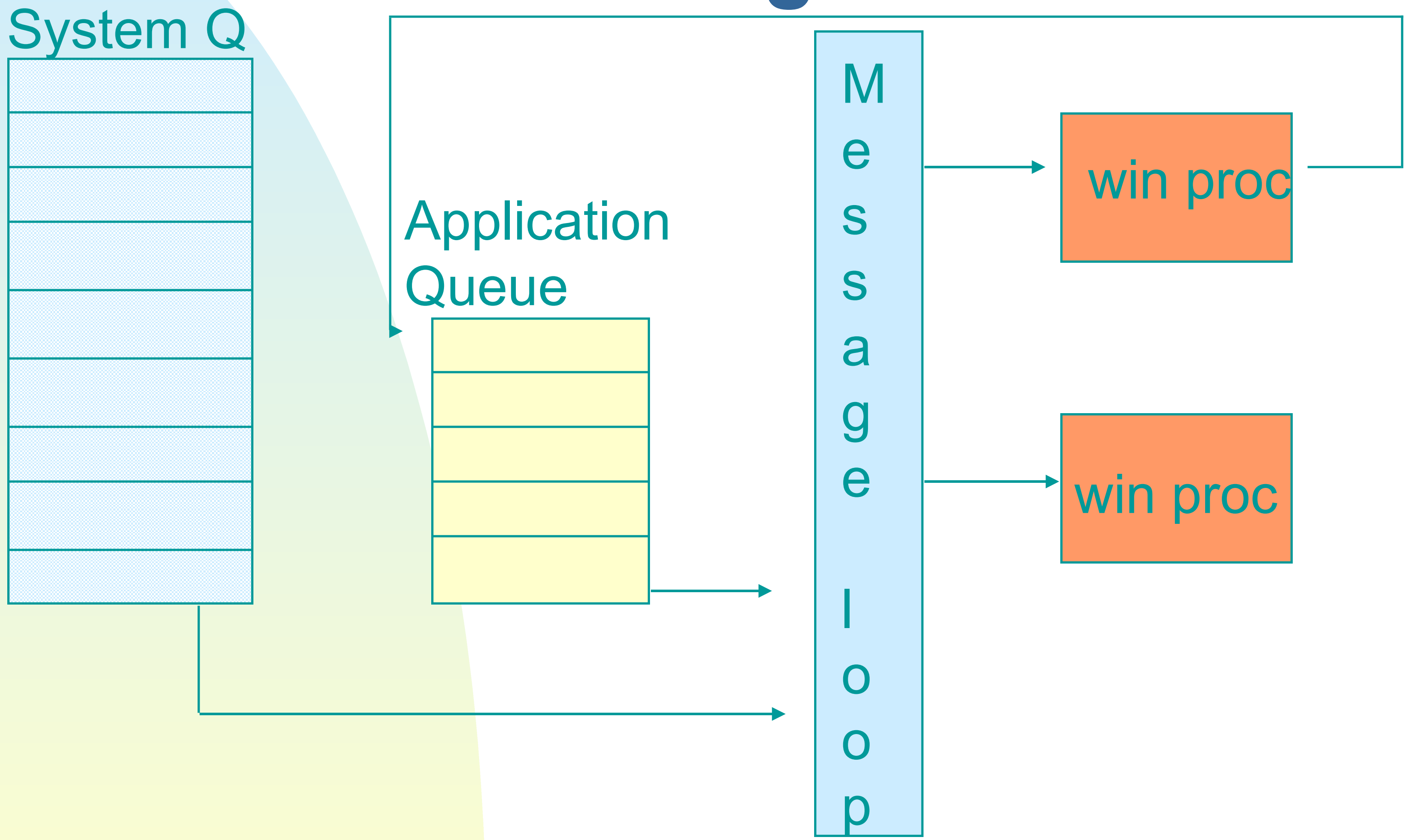
Message Queues

- Windows processes messages
  - Hardware, operating system, software and application messages are all processed in the same way
  - Sent to an application message queue, one for each application (Win95 and above)

MFC Components



Message Queues



Messages and MFC

- The MFC supervisor pulls messages from the queue and routes them to the different components of your application
  - Components register for messages they are interested in
  - Unregistered messages are discarded
  - Each message may be processed multiple times

Messages and MFC

- Review
  - Any MFC object may register an interest in a message through a **message map**.
  - Messages percolate through all components
  - Messages are the key communication mechanism within windows and MFC

MFC message map (.h)

```
// Generated message map functions
protected:

        //{{AFX_MSG(CTextView)
        afx_msg void OnFontSmall();
        afx_msg void OnFontMedium();
        afx_msg void OnFontLarge();
        afx_msg void OnUpdateFontSmall(CCmdUI* pCmdUI);
        afx_msg void OnUpdateFontMedium(CCmdUI* pCmdUI);
        afx_msg void OnUpdateFontLarge(CCmdUI* pCmdUI);
        afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
        afx_msg void OnMouseMove(UINT nFlags, CPoint point);
        afx_msg void OnText();
        afx_msg BOOL OnSetCursor(CWnd* pWnd, UINT nHitTest, UINT message);
        afx_msg void OnUpdateText(CCmdUI* pCmdUI);
        afx_msg void OnRectangle();
        afx_msg void OnUpdateRectangle(CCmdUI* pCmdUI);
        afx_msg void OnChar(UINT nChar, UINT nRepCnt, UINT nFlags);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

MFC message map (.cpp)

```
BEGIN_MESSAGE_MAP(CTextView, CView)
          //{{AFX_MSG_MAP(CTextView)
          ON_COMMAND(IDM_FONTSMALL, OnFontSmall)
          ON_COMMAND(IDM_FONTMEDIUM, OnFontMedium)
          ON_COMMAND(IDM_FONTLARGE, OnFontLarge)
          ON_UPDATE_COMMAND_UI(IDM_FONTSMALL, OnUpdateFontSmall)
          ON_UPDATE_COMMAND_UI(IDM_FONTMEDIUM, OnUpdateFontMedium)
          ON_UPDATE_COMMAND_UI(IDM_FONTLARGE, OnUpdateFontLarge)
          ON_WM_LBUTTONDOWN()
          ON_WM_MOUSEMOVE()
          ON_COMMAND(IDM_TEXT, OnText)
          ON_WM_SETCURSOR()
          ON_UPDATE_COMMAND_UI(IDM_TEXT, OnUpdateText)
          ON_COMMAND(IDM_RECTANGLE, OnRectangle)
          ON_UPDATE_COMMAND_UI(IDM_RECTANGLE, OnUpdateRectangle)
          ON_WM_CHAR()
          //}}AFX_MSG_MAP
          // Standard printing commands
          ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
          ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
          ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
          // Color commands
          ON_UPDATE_COMMAND_UI_RANGE( IDM_RED, IDM_GRAY, OnUpdateColor)
          ON_COMMAND_RANGE( IDM_RED, IDM_GRAY, OnColor)

END_MESSAGE_MAP()
```

MFC Message routing - SDI

- View
- Document
- Document Template
- Mainframe Window
- Application

MFC Message routing - MDI

- Active view
- Document associated with the active view
- Document Template for the active document
- Frame window for the active view
- Mainframe Window
- Application

Message Categories

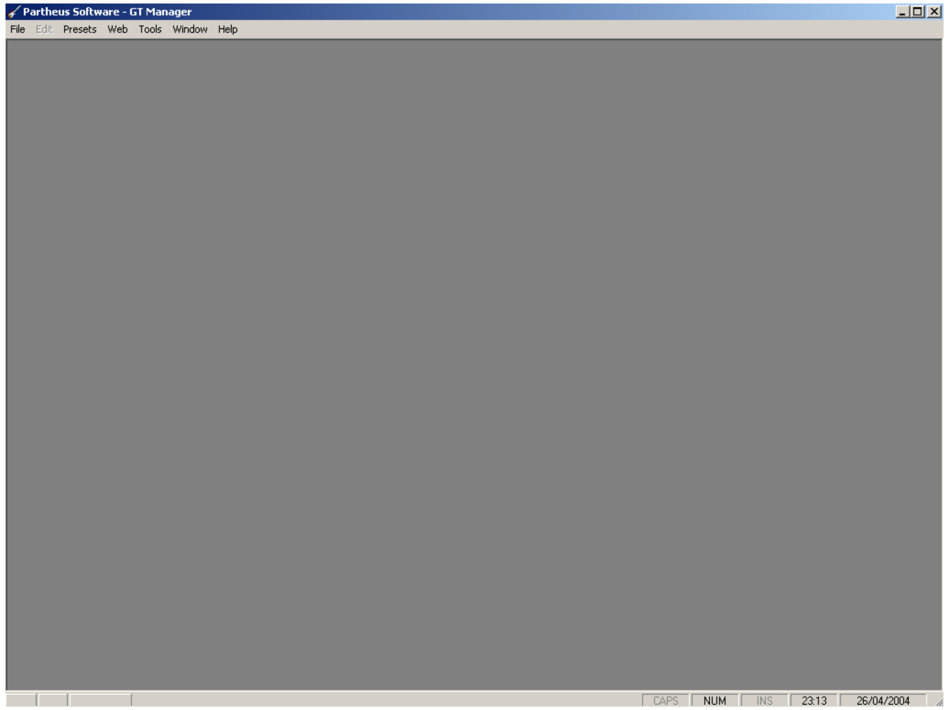
- Windows Messages
  - Standard window messages. Paint, mouse, keyboard etc. All WM_XXX messages except for WM_COMMAND
- Control Notification Messages
  - WM_COMMAND messages sent to a control
- Command messages
  - WM_COMMAND messages that are sent by UI elements such as menu, toolbar etc.

Message delivery

- Only the CWnd class can handle Windows messages or control notification messages
  - Or any class derived from CWnd (like CView)
  - Review derivation hierarchy

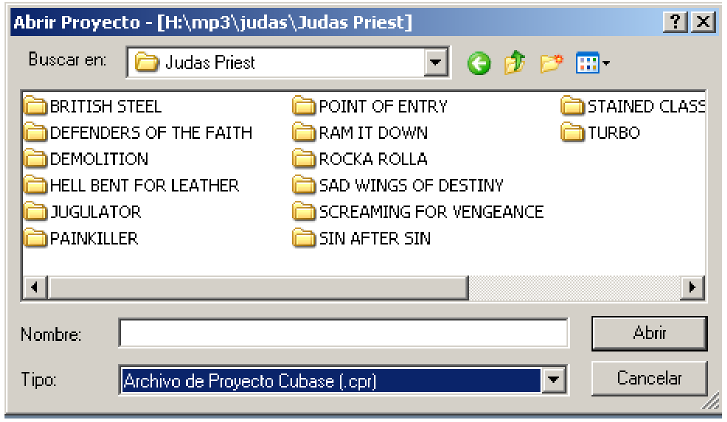
Container Windows

- Provide the structure to the user interface
- Used for managing the contained windows
- *Frame*: application main window
  - CFrameWnd
  - CMDIFrameWnd

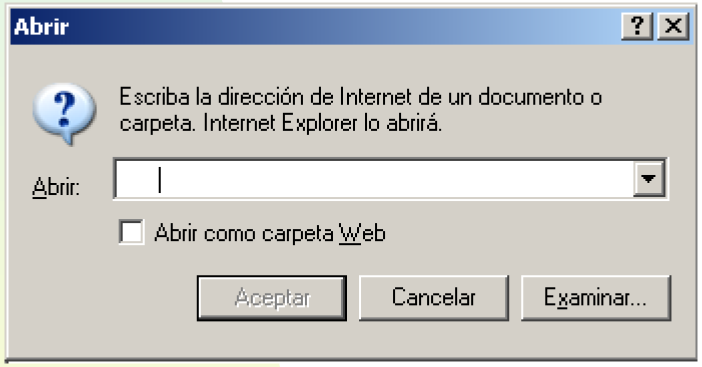

Partheus Software - GT Manager
File Edit Presets Web Tools Window Help
CAPS NUM INS 23:13 26/04/2004

Container Windows (contd.)

- *Dialog*: only contains dialog box controls
  - *CDialog*
  - Nineteen classes more, useful to the most common needs


Abrir Proyecto - [H:\mp3\judas\Judas Priest]
Buscar en: Judas Priest
Nombre:
Tipo: Archivo de Proyecto Cubase (.cpr)
Abrir
Cancelar

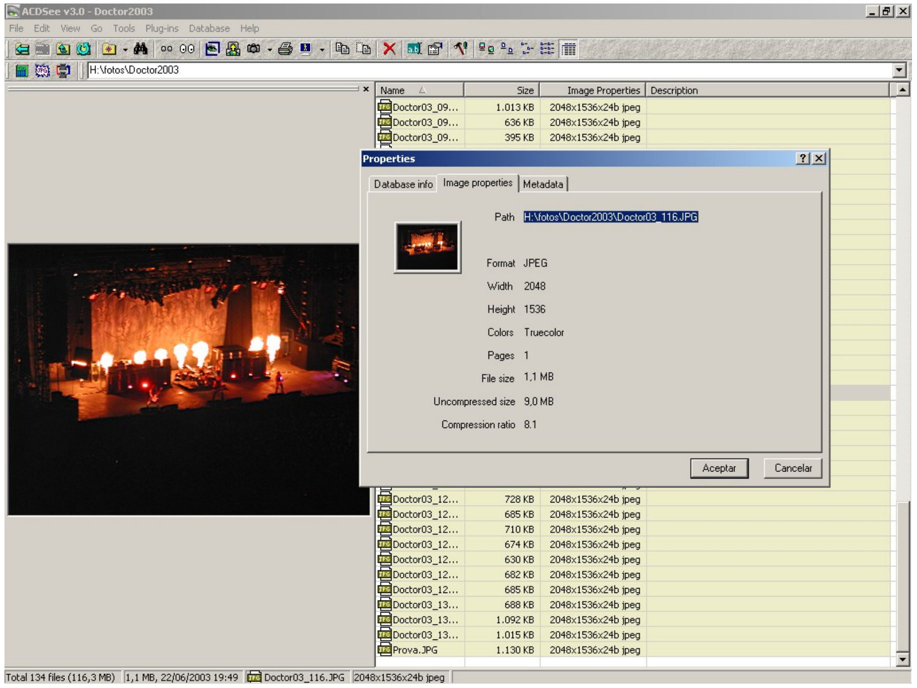
BRITISH STEEL
DEFENDERS OF THE FAITH
DEMOLITION
HELL BENT FOR LEATHER
JUGULATOR
PAINKILLER
POINT OF ENTRY
RAM IT DOWN
ROCKA ROLLA
SAD WINGS OF DESTINY
SCREAMING FOR VENGEANCE
SIN AFTER SIN
STAINED CLASS
TURBO



Escriba la dirección de Internet de un documento o carpeta. Internet Explorer lo abrirá.

Data Windows

- Windows contained in Frame or Dialog windows, managing some kind of user data
  - Control bars
  - View windows
  - Dialog box controls



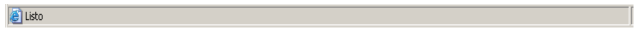
Name	Size	Image Properties	Description
Doctor03_09...	1.013 KB	2048x1536x24b jpeg	
Doctor03_09...	636 KB	2048x1536x24b jpeg	
Doctor03_09...	395 KB	2048x1536x24b jpeg	
Doctor03_12...	728 KB	2048x1536x24b jpeg	
Doctor03_12...	685 KB	2048x1536x24b jpeg	
Doctor03_12...	710 KB	2048x1536x24b jpeg	
Doctor03_12...	674 KB	2048x1536x24b jpeg	
Doctor03_12...	630 KB	2048x1536x24b jpeg	
Doctor03_12...	682 KB	2048x1536x24b jpeg	
Doctor03_12...	685 KB	2048x1536x24b jpeg	
Doctor03_13...	688 KB	2048x1536x24b jpeg	
Doctor03_13...	1.092 KB	2048x1536x24b jpeg	
Doctor03_13...	1.015 KB	2048x1536x24b jpeg	
Prova.JPG	1.130 KB	2048x1536x24b jpeg	

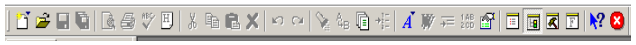
Path	H:\fotos\Doctor2003\Doctor03_116.JPG
Format	JPEG
Width	2048
Height	1536
Colors	Truecolor
Pages	1
File size	1,1 MB
Uncompressed size	9,0 MB
Compression ratio	8.1

Total 134 files (116,3 MB) | 1,1 MB, 22/06/2003 19:49 | Doctor03_116.JPG | 2048x1536x24b jpeg

Control Bars

- Inherit from *CControlBar*; Frame window ornaments
  - CStatusBar
  - CToolBar
  - CDialogBar





View Windows

- Inherit from *CView*
- Provide graphic representation and edition of a data group
  - *CScrollView*: when the object is bigger than the window
  - *CRecordView*: connect the form values to the fields of a data base

Dialog Box Controls

- Seven classes:
  - *CStatic*: system static controls: Text, rectangles, icons and other non-editable objects.
  - *CButton*: system buttons
  - *CBitmapButton*: connect a bit map to a button
  - *CListBox*: lists
  - *CComboBox*: combo boxes
  - *CScrollBar*: scroll bars
  - *CEdit*: text edition controls

Messages and Windows Control

- Message: window entry unit
  - Create, resize, close window
  - Keyboard and mouse interaction
  - Other object events
- Message map: data structure used to capture messages. Matrix connecting message values and class functions.
- Intensive use to manage command inputs via menus, keyboard accelerators and toolbars

MFC and Database Access

- Target: ODBC made easier
- Three main classes:
  - *CDatabase*
  - *CRecordset*
  - *CRecordView*
- Exception
  - *CDBException* (inherits from *CException*).

MFC and Database Access

- *CDatabase*: data source connection
  - Might be used with one or more *CRecordSet* objects or by itself (e.g. when we want to execute an SQL command without receiving any result)
- *CRecordset*: set of records in a data source
  - *Dynaset*: data synchronized with the updates commited by the other data source users.
  - *Snapshot*: static image of the data in a determined moment

MFC and Network Access

- *Winsock*: low level Windows API for TCP/IP programming.
- MFC Winsock classes: *CAsyncSocket*, *CSocket*
- Not recommended in 32 bit programming: it’s a dirtily-patched Win16 code, based in messages

MFC and Internet Access

- *WinInet*
  - A higher level API than Winsock
  - Used to build *client* programs
  - Useless to build *server* programs
  - Used in Internet Explorer
  - Only available in Win32
- MFC provides an quite good WinInet envelope

MFC and Internet Access

- MFC adds exception processing to the underlying API
  - *CInternetException.*
- MFC classes for Internet access:
  - *CInternetSession*
  - *CHttpConnection*
  - *CFtpConnection*
  - *CGopherConnection*
  - *CInternetFile*
  - *CHttpFile*
  - *CFtpFileFind*
  - *CGopherFileFind*

CDC and CPaintDC

- MFC device context classes.
- Holds a device context handle, and wraps the SDK DC functions.
- CPaintDC is used when responding to WM_PAINT messages.
- CPaintDC encapsulates calls to BeginPaint and EndPaint in the constructor and destructor respectively.

Simple types and GDI classes

- MFC does a very simple encapsulation of structures like RECT and POINT with classes like CRect and CPoint.
- One of the most useful classes in MFC is CString.
  - Similar to a character array, with a handful of useful methods
- All of the GDI structures have their own classes
  - creation and destruction is handled by C++
  - can be used freely with the old structures

GDI - A Little Background

GDI - Graphics Device Interface

- Provides a single programming interface regardless of the graphics device being used.
- Program to a display the same as a printer or other graphics device.
- Manufacturer provides the driver Windows uses to interface with a graphics device instead of programmers having to write code for every device they wish to support.

The Device Context

- The DC contains information that tells Windows how to do the job you request of it. In order for Windows to draw an object it needs some information.
  - How thick should the line be?
  - What color should the object be?
  - What font should be used for the text and what is its size?
- These questions are all answered by you configuring the DC before requesting an object to be drawn.

More on drawing

- GDI = Graphics device interface
- allows for device independent drawing
  - could draw to different kinds of displays, or a printer
- Device Context (or DC)

  (class is called CDC)
  - object contains all information about how to draw:

    pen, brush, font, background color, etc.
  - member functions for get/set of all of the above attributes
  - all drawing functions are member functions of CDC

GDI Objects

- GDI objects are for storing the attributes of the DC
  - base class is CGdiObject
  - subclasses include:
    - CPen -- lines and borders
      have width, style, and color
    - CBrush -- filled drawing
      can be solid, bitmapped, or hatched
- Changing DC attributes:
  - can “select” GDI objects into and out of device contexts
  - if you change an attribute for drawing, you must restore it back to the old value when you are done.

Example: save and restore GDI object

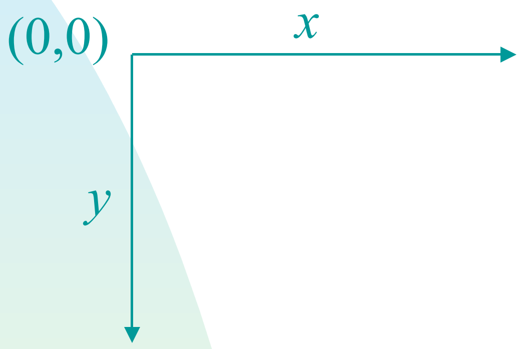
```
void CHelloView::OnDraw(CDC* pDC)
{
    CHelloDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    CPoint loc = pDoc->getLoc();

    CFont newFont;
    newFont.CreatePointFont(24*10,
               "Harlow Solid Italic", pDC);

    CFont *pOldFont =
                 pDC->SelectObject(&newFont);
    pDC->TextOut(loc.x, loc.y, "Hello world!");
    pDC->SelectObject(pOldFont);
}
```

Drawing Basics

- Logical coordinate system



- Line drawing Example:

```
pDC->MoveTo(5, 10);
pDC->LineTo(50, 100);
```

- To get total coordinates of View:

```
CWnd::GetClientRect(CRect *);
```

  - View classes are derived from CWnd

Drawing Utility classes

- CPoint, CRect, CSize subclasses of Win32 structs POINT, RECT, SIZE
  - data is public; have some arithmetic operators
- Also, RGB macro, for specifying colors:
  - for use where type COLORREF is required
  - Examples:

    RGB(0,0,0) RGB(255,0,0) RGB (255, 255, 255)

    black, red, white
  - Example use:

```
CPen pen (PS_SOLID, 2, RGB(0,255,0));
```

Using Stock GDI objects

- Some built-in GDI objects
- Example:

```
 CBrush *pOldBrush;
pOldBrush = static_cast<CBrush *>(
                    pDC-
>SelectStockObject(GRAY_BRUSH));
. . .
pDC->SelectObject(pOldBrush);
```

- Need down-cast because SelectStockObject returns CGdiObject *

Acquiring a DC
(not while in an OnPaint method)

- To acquire a DC pointer in an MFC application outside its OnPaint method, use CWnd::GetDC. Any DC pointer acquired in this fashion must be released with a call to CWnd::ReleaseDC.

```
CDC* pDC = GetDC();
  // Do some drawing
ReleaseDC(pDC);
```

Acquiring a DC
(While in an OnPaint method)

- To respond to a WM_PAINT message in an OnPaint method, MFC provides functions:
  CWnd::BeginPaint and CWnd::EndPaint

```
PAINTSTRUCT ps;
CDC* pDC = BeginPaint(&ps);
  // Do some drawing
EndPaint(&ps);
```

Acquiring the DC - Yet Even Easier

- So you don’t have to remember procedures for acquiring and releasing the DC, MFC encapsulates them in 4 classes.
  - CPaintDC - For drawing in a window’s client area in an OnPaint method.
  - CClientDC - For drawing in a window’s client area outside of an OnPaint method.
  - CWindowDC - For drawing anywhere in the Window, including the nonclient area.
  - CMetaFileDC - For drawing to a GDI metafile

CPaintDC

- Using CPaintDC makes the example from before even easier and safer.

Before:

```
Void CMainWindow::OnPaint()
{
   PAINTSTRUCT ps;
   CDC* pDC = BeginPaint(&ps);
    // Do some drawing
   EndPaint(&ps);
}
```

After:

```
Void CMainWindow::OnPaint()
{
   CPaintDC dc (this);
   //Do some drawing
}
```

The Device Context – Attributes

- The Attributes of the Device Context supplies Windows with the information it needs to draw a line or text or …
  - The LineTo function uses current pen to determine line color, width and style.
  - Rectangle uses current pen to draw its border and current brush to fill its area.

The Device Context - SelectObject

- The function used more than any other is the SelectObject function which changes current Pen, Brush or Font of the DC. The DC is initialized with default values but you can customize the behavior of functions like LineTo by replacing the current CPen and CBrush with your own.

```
//Assume pPen and pBrush are pointers to
CPen and CBrush objects.

dc.SelectObject (pPen);

dc.SelectObject (pBrush);

dc.Ellipse(0, 0, 100, 100);
```

SelectObject Method

- **CPen* SelectObject(CPen*** *pPen* **);**
- **CBrush* SelectObject(CBrush*** *pBrush* **);**
- **virtual CFont* SelectObject(CFont*** *pFont* **);**
- **CBitmap* SelectObject(CBitmap*** *pBitmap* **);**
- **int SelectObject(CRgn*** *pRgn* **);**

The Device Context - Drawing Mode

- The drawing mode determines how Windows will display pixels that represent an object being drawn. By default, Windows will just copy pixels to the display surface. The other options combine pixel colors, based on Boolean expressions. For example, you can draw a line just by NOTing the pixels required to draw the line, which inverts the current pixel color. The drawing mode is R2_NOT.

```
dc.SetROP2(R2_NOT);

dc.MoveTo(0,0);

dcLineTo(100,100);
```

- SetROP2 means “Set Raster Operation to”

The Device Context - Mapping Mode

- The mapping mode is the attribute of the device context that indicates how logical coordinates are translated into device coordinates.

- *Logical Coordinates* are the coordinates you pass to CDC output functions. Logical coordinates are in some unit of measurement determined by the mapping mode.

- *Device Coordinates* are the corresponding pixel positions within a window. Device Coordinates always speak in pixels.

Device Context- Mapping Mode

- The default mapping mode is MM_TEXT with units in pixels. This doesn't have to be the case.
- MM_LOENGLISH is a mapping mode who's units are in inches. One unit = 1/100 of an inch. One way to ensure something you draw is exactly 1 inch for instance.
- Non-MM_TEXT mapping modes allow for consistent sizes and distances regardless of a device's physical resolution.

```
dc.SetMapMode(MM_LOMETRIC)

dc.Ellipse(0, 0, 500, -300)
```

The Device Context - Mapping Mode

- Orientation of the X and Y axes differ in some mapping modes. For the default, MM_TEXT, mapping mode, x increases to the right and y increases down with origin at upper left.
- All others (except for the user defined modes) have x increasing to the right and y decreasing down, with origin at upper left.
- MM_ANISOTROPIC(scale independent) , MM_ISOTROPIC(scale evenly) have user defined units and x,y axes orientation.

The Device Context - Mapping Mode

- The origin is separate from the mapping mode. By default, the origin is the top left corner but like the mapping mode can be customized.

```
Crect rect;
GetClientRect(&rect);
dc.SetViewportOrg(rect.Width()/2, rect.Height()/2);
```

- This example moves the origin to the center of the client area.

Drawing with the GDI - Lines and Curves

- The GDI supplies a long list of output functions to draw all sorts of graphics.
- The simplest objects are lines and curves and a few of the supporting functions follow.
  - MoveTo - sets current position
  - LineTo - draws a line from current pos to new pos and updates current pos
  - Polyline - Connects a set of pts with line segments.
  - PolylineTo -PolyLine but updates current pos with last pt.
  - Arc - Draws an arc
  - ArcTo - Arc but updates current pos to equal the end of arc

Drawing with the GDI
- Ellipses, Polygons and Other Shapes

More advanced shapes are also supported by GDI functions.

- Chord - Draws a closed figure bounded by the intersection of an ellipse and a line.
- Ellipse - Draws a circle or ellipse.
- Pie - Draws a pie-shaped wedge
- Polygon - Connects a set of points for form a polygon
- Rectangle - Draws a rectangle with square corners
- RoundRect - Draws a rectangle with rounded corners

Pens and the CPen Class

- The Device Context has an Attribute referred to as a Pen. Windows uses the Pen to draw lines and curves and also to border figures drawn with Rectangle, Ellipse and others.
- The default pen creates a black, solid line that is 1 pixel wide.
- Users can customize a pen by creating a CPen object and specifying it's color, width and line style then selecting it into the Device Context with the SelectObject member function.

```
Cpen pen;
pen.CreatePen(PS_DASH, 1, RGB(255, 0, 0));
dc.SelectObject(&pen);
```

Brushes and the CBrush Class

- The current Brush is an attribute of the Device Context. The current brush is how Windows determines how to fill objects drawn with functions like Rectangle, Ellipse and others. Brush indicates both color and style (solid or Hatch)

```
//Solid Brush
CBrush brush (RGB(255,0,0));
//Hatch Brush
CBrush brush (HS_DIAGCROSS, RGB(255,0,0));
```

Drawing Text

- As with drawing objects, the GDI offers supporting functions for drawing text.
- DrawText - Draws text in a formatting rectangle
- TextOut - Outputs a line of text at the current or specified position.
- TabbedTextOut - Outputs a line of text that includes tabs
- ExtTextOut - Outputs a line of text and optionally fills a rectangle, varies intercharacter spacing

Drawing Text - Supporting Functions

- Drawing text and getting things to line up space properly can be a little cumbersome. The following functions are available to supply needed information:
    - GetTextExtent - Computes width of a string in the current font.
    - GetTabbedTextExtent - Width including tabs
    - GetTextMetrics - Font metrics(character height, average char width …)
    - SetTextAlign - Alignment parameters for TextOut and others
    - SetTextJustification - specifies the added width needed to justify a string
    - SetTextColor - Sets the DC text output color
    - SetBkColor - Sets the DC background color for text

Fonts and the CFont Class

- MFC represents a Font with the CFont class. Like Pens and Brushes, you can change the default Font by creating an instance of the CFont class, configuring it the way you wish, and selecting it into the DC with SelectObject.

```
//12 pt Font (pt parameter passed is 10 * desired_pt_size)
CFont font;
fond.CreatePointFont(120, _T("Times New Roman"));
```

Types of Fonts

- Raster Fonts - fonts that are stored as Bitmaps and look best when they’re displayed in their native sizes.
- TrueType Fonts - fonts that are stored as mathematical formulas which allows them to scale well.

Stock Objects

- Windows predefines a handful of pens, brushes, fonts and other GDI objects that can be used without being explicitly created and are not deleted.

```
dc.SelectStockObject(LTGRAY_BRUSH);
dc.Ellipse(0,0,100,100);
```

Example without Using Stock Objects

Drawing a light gray circle with no border:

```
//Example Without Stock Objects
CPen pen (PS_NULL, 0, (RGB(0,0,0)));
dc.SelectObject(&pen);
CBrush brush (RGB(192,192,192));
dc.SelectObject(&brush);
dc.Ellipse(0, 0, 100, 100);
```

Using Stock Objects

Drawing a light gray circle with no border:

```
//Example With Stock Objects
dc.SelectStockObject(NULL_PEN);
dc.SelectStockObject(LTGRAY_BRUSH);
dc.Ellipse(0 ,0, 100, 100);
```

GDI Object Management

- GDI objects are resources and consume memory. This makes it important to ensure an object is deleted when you are finished. The best way is to always create instances of CPen’s, CBrush’s and CFont’s on the stack, as local objects, so their destructors are called as they go out of scope.

- Sometimes, new’ing an object is required. It’s important to make sure that all GDI objects created with the new operator are deleted with the delete operator when they are no longer needed.

- Stock Objects should NEVER be deleted.

GDI Object Management - Deselecting GDI Objects

- Ensuring GDI objects are deleted is important. But, it is also extremely important to avoid deleting an object while it's selected into a device context.
- An easy and good habit to get into is to store a pointer to the default object that was installed when you retrieved the device context and then re-select it back into the context before deleting the object you created.

GDI Object Management -Deselecting GDI Object Example

```
//Option 1
CPen pen (PS_SOLID, 1, RGB(255, 0, 0));
CPen* pOldPen = dc.SelectObject(&pen);
   :
dc.SelectObject(pOldPen);

//Option 2
CPen pen (PS_SOLID, 1, RGB(255,0,0));
dc.SelectObject(&pen);
   :
dc.SelectStockObject(BLACK_PEN);
```

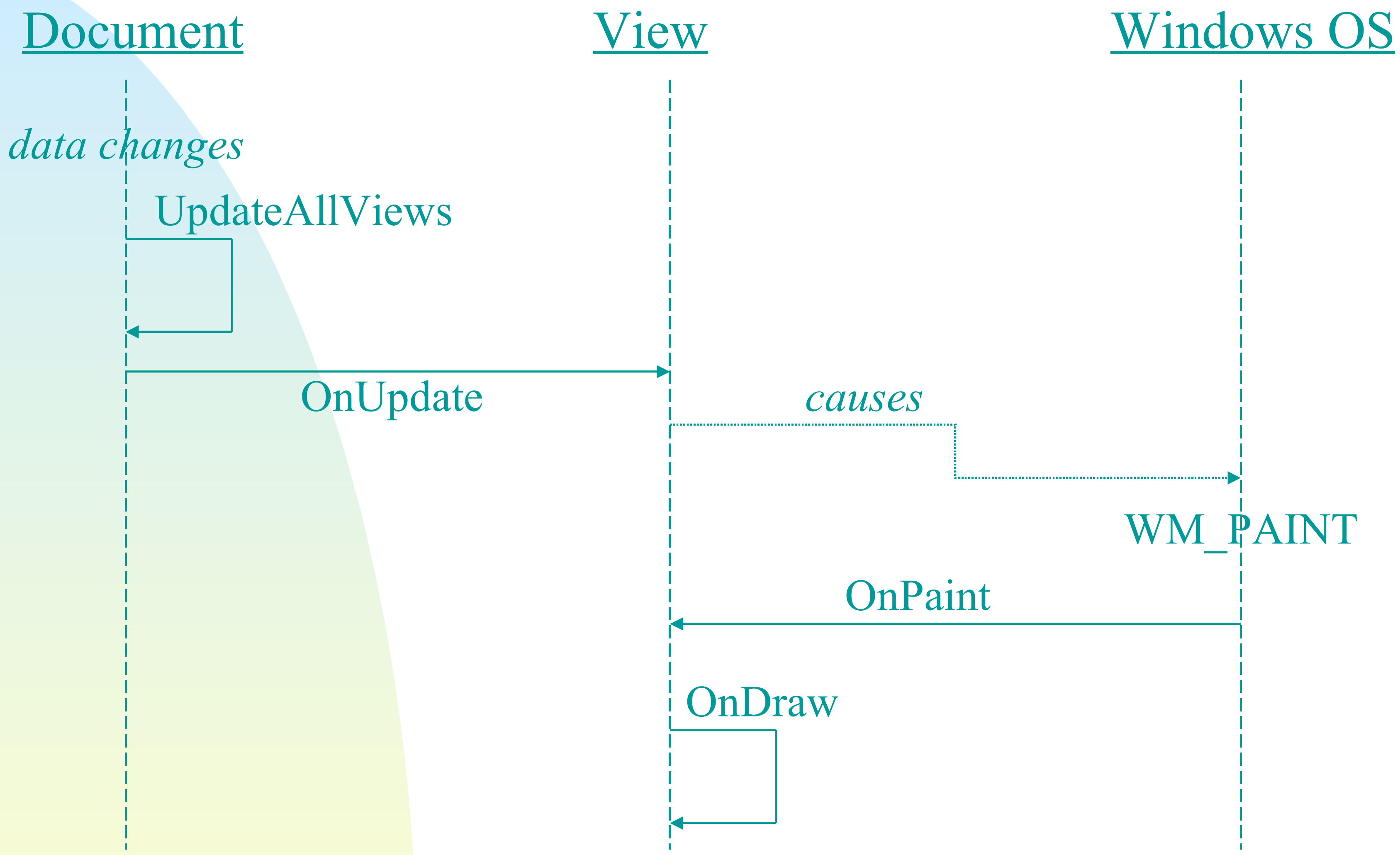
WM_PAINT / OnPaint

- WM_PAINT is the message that gets sent by Windows if a Window needs to be repainted.
  - because exposed, resized, etc.   OR
  - because someone called `Invalidate` (i.e., data has changed; need to redraw to reflect changes)
- OnPaint is message handler for WM_PAINT
  - textbook has examples of writing this handler
- except, CView has a different mechanism.
  - Instead of OnPaint, we'll write OnDraw
  - more details to follow….

Updating a View

- Recall Doc-View model:
  - Doc has program data
  - View is for I/O
- How to handle updating a view . . .

Updating a view: sequence diagram



Updating a view: application code

- When data changes, call `UpdateAllViews` (from Document class). (I.e., in Doc member funcs)
- Write `OnDraw` for View class
  - CDC variable is passed in as a parameter
    - DC = device context (first C is for Class); needed for drawing
      (more about CDC later)
    - means you don't have to create and destroy it
    - means this function can work for both drawing and printing
  - (Note: OnDraw is not part of the message map; it's a real virtual function)
- All the other stuff is taken care of by MFC

Using CDocument

- Application data goes here:
  - you'll add data members and associated member functions. Functions that modify an object should call UpdateAllViews(NULL)
- In SDI: one document object for the lifetime of the application.
  - Don't initialize in constructor or
  - cleanup in destructor
  - When we open new file documents, we reuse the same document object
  - Constructor and destructor only get called once per run of the application

CDocument: Possible Overrides

(the following applies to SDI apps)

- **`OnOpenDocument`**, **`OnNewDocument`**
  - initialize
  - make sure overriding version calls base class version (see below)
- **`DeleteContents`**
  - cleanup (not in destructor)
  - gets called by base class versions of OnOpenDocument and OnNewDocument
  - also gets called on application startup and finish
- Other funcs for processing commands and files: we'll discuss in future lectures.

- When a message is received the application framework looks up the message identifier in the Windows message map and calls the appropriate message handler.
- The message map is declared using
  - DECLARE_MESSAGE_MAP()
  - The Actual message map in the source begins with BEGIN_MESSAGE_MAP(ownerclass,baseclass)

    and ends with END_MESSAGE_MAP()
  - Between these lines the programmer ties message identifiers to message handlers using message macros.
- Predefined MFC message identifies are located in header file <afxwin.h> with range 0 – 1023
- Programmer defined range: 1025 to 65535

- Message handlers are functions MFC calls to respond to messages passed to the program by Windows. Message handlers are mapped to message identifiers by the message map.
- We use the macro _N_COMMAND to associate a message identifier with a programmer defined message handler.

MFC Resources

- Visual C++ provides a resource definition language to specify each GUI control's location, size, message identifier.
- The resource files are identified with the .rc file extension.
- Be sure to edit the .rc file as TEXT FILE.
- Clicking on it opens the graphical resource editor.

Hungarian Notation

- Controversial since placing in a name a prefix that indicates the data type violates data abstraction and information hiding.
- BUT, it makes a complex MFC C++ program easier to read and maintain if the notation is used consistently.
- Pg 32

Win32 Applications

- When creating the project select
  - Win32Application
  - NOT Win32 Console Application
- BE SURE TO SELECT
  - Use MFC in a Shared DLL on the Project Settings Screen.

Fig 2.8

- CWelcomeWindow is derived from MFC class CFrameWnd.
- By inheriting CFrameWnd, our class starts out with the basic windowing functionality such as the ability to move, resize and close.
- #include <afxwin.h> application framework header

- Create creates the main window .
  - NULL indicates a standard CFrameWnd window.
  - “Welcome” name in the title bar
  - WS_OVERLAPPED – create a resizable window with system menu (Restore, Move, Size, Minimize, Maximize and Close)
    - Ie a full featured window
  - CRect – SCREEN COORDINATES
    - X axis – 0 to +x horizontal in pixels
    - Y axis – 0 to +y vertical in pixels
    - 100,100 top left coordinate of the window
    - 300,300 bottom right coordinate of the window

- CStatic object – an object that displays text but does not send messages.
  - m_pGreeting = new CStatic; //static control
- Create windows control
  - m_pGreeting->Create (text, window styles and static object styles, WINDOW COORDINATES, the context that owns the child window)
  - WS_CHILD – a window that is always contained inside another window.
  - WS_VISIBLE – visible to user
  - WS_BORDER – window to have a border
  - SS_CENTER – text displayed in the CStatic window should be centered.
  - Coordinates 40,50,160,100
    - Upper left 40,50 in parent window
    - With size 160,100

- Remember to call delete in the destructor on any dynamically allocated objects.
- Application start-up, execution and termination are controlled by CWinApp.
- } welcomeApp;
  - Creates an instance of CWelcomeApp called welcomeApp.
  - When welcomeApp is instantiated the default constructor calls the base-class constructor (CWinAPP) which stores the address of this object for WinMain to use. WinMain calls function InitInstance which creates CWelComeWindow.

Section 2.8 Menus

- Figure 2.11 creates a window containing 4 menus. When an item is selected the price is added to a running total. When show total is selected the total price is displayed.
- Four files –
  - CMenusWin.h – class definition
  - Menus.cpp -   class implementation
  - Menus.rc – resource file that defines the menus
  - Menus_ids.h – defines message identifiers.

CMenusWin.h

- CMenusWin extends CFrameWnd
- Defines methods to be message handlers.
- Instantiate object of type output string stream.
- Declares the message map.

- Associates message identifiers.
- Menus.rc
  - Defines Food MENU
    - Associates menuitem with a message identifier.
- Menus.cpp
  - Message map ties message identifiers to the message handlers.
  - NOTE: Standard boilerplate
  - Note: initialization of ostrstream in CMenusWin constructor.
  - Create 5th argument – NULL ie not a child window.
  - 6th argument “Food” is the label on the MENU definition in the resource file.

Dialog Boxes

- Class CDialog is used to create windows called dialog boxes used to get information input by the user.
- Figure 2-12 uses a dialog box containing 2 buttons and 2 edit text controls to prompt the user for a series of numbers.
- When the user enters a number in the white edit box and clicks Add, the number is added to a running total and displayed in the Total exit box. The Total box is gray because it does not accept input.

- CAdditionDialog.h
  - Is derived from class CDialog.
  - The dialog resource name “Addition” is passed to the CDialog base-class constructor to initialize the dialog box.
- GetDlgItem is used to retrieve the addresses of the two edit boxes in the dialog box. Note: the addresses returned can change from one message to another because Windows reallocates memory as it creates and deletes windows.
- The ID codes IDC_TOTAL and IDC_NUIMBER are defined in the header addition_ids.h

- ES_NUMBER – edit style only permits numeric input into the edit box.\
- DoModal is called to display the dialog as a modal window ie no other windows in the program can be interacted with until this dialog is closed – alternative is to use Create.
- Style resource definition statement:
  - DS_MODALFRAME – other windows in the application cannot be accessed until the frame is closed.
  - WS_POPUP – standalone window
  - WS_CAPTION – title bar
  - WS_SYSMENU – indicates close button (x)

- IDC_STATIC – static control does not generate messages or need to be accessed by the program so it does not require a unique control identifier.
- IDC_TOTAL edit style is read-only ie ES_READONLY.

Mouse Messages

- Programmer writes message handlers to respond to mouse messages.
- Override baseclass message handlers OnLButtonDown and OnRButtonDown.
- Use UINT value to determine which mouse button was pressed and use CPoint object to get the coordinates of the mouse click.

- Before we can draw on a window we must get a device-context object that encapsulates the functionality for drawing on a window.
- To draw inside the window’s client area we need a CClientDC device_context object.
- CClientDC dc(this); gets the contexst for CMouseWin’s client area by passing the this pointer to the CClientDC constructor. Using the object dc, we can draw in the window’s client area.

Processing Keyboard Input

- Fig 3.4
  - When the user types a character on the keyboard, Windows passes the WM_CHAR message to our program. The message dispatcher looks up the WM_CHAR message identifier and calls our overridden message handler OnChar.
  - InvalidateRect is called to repaint the client area (sends a WM_PAINT message to the message dispatcher for our class)/ Argument NULL indicates that the entire client area should be repainted
  - Message handler OnPaint handles the paint message ie the message passed by Windows when the client area must be redrawn. Generated when a window is minimized, maximized, exposed (becomes visible)
  - NOTE: Function OnPaint must use a CPaintDC device context to draw in the window.
    - CPaintDC dc (this).

- TextOut is passed the top-left coordinates where drawing is to begin, the text to display, and the number of characters of text to display.

Figure 3.5

- Demonstrates how to determine the size of the screen, control the color and placement of text on the screen and determine the width and height in pixels of a string of text.
  - OnPaint used the CPaintDC
  - GetClientRect()
  - GetTextExtent to determine a strings width and height.
  - RGB(255,0,0) red Red Green Blue
  - Note what happens when you shrink the window to be smaller than the text – clipped.

Figure 4.1

- Allow the user to enter text in a multiline edit text control. When the user clicks count it counts and displays the count in the same edit box.
  - ES_MULTILINE – edit text is multiline.
  - ES_WANTRETURN – Respond to ENTER key with a newline
  - WS_VSCROLL – display a vertical scrollbar

Figure 4.2

- Check Boxes – On/Off State for each.
- Clicking toggles.
- .RC Features
  - GROUPBOX
  - AUTOCHECKBOX
  - GetButtonStatus

Figure 4.3

- Radio Buttons
  - Only 1 radio button in a group can be true.
- .RC
  - GROUPBOX
  - NOTE: WS_GROUP indicates first radio button in a group. Distinguished groups NOT GROUPBOX
  - AUTORADIOBUTTON
  - GetCheckedRadioButton

Figure 4.4

- ListBox displays a list of items from which the user can select 1 or MORE.
- Mostly a matter of string manipulation and indexes.
  - Fish has index 0
  - Salad index 1
  - Chicken index 2

Ch 5 - Graphics

- At lowest level MFC has SetPixel and GetPixel.
- But higher level functions for drawing lines and shapes are provided.
- An MFC device context provides the functions for drawing and contains the data members that keep track of the BITMAP(array of pixels), PEN (object for drawing), BRUSH (object for filling regions), COLOR PALETTE (available colors)

- When a function needs to draw in a window, it creates a CClientDC object to write in the client area of the dinwos.
- The function OnPaint creates a CPainDC object to access the regions of the window that needs to be updated.
- Colors Fig 5.1
- Drawing functions clip an image to fit into the bitmap.

- First Step
  - Clear the bitmap by calling PatBlt – pattern block transfer
  - Often more efficient, after creating an image, to copy the image rather than redraw.
    - CDC member function BitBlt (bit block transfer)
- Drawing uses 2 object to specify drawing properties
  - Pen and Brush (CPen/CBrush)
- CreateStockObject(HOLLOW_BRUSH) used to prevent filling of the enclosed region.

- Note: NO predefined RGB colors so use the macro RGB to make colors as needed.