Windows Programming/MFC Course Notes

SC447

What is Visual C++

- Visual C++ (VCPP) is not a special language as is Visual Basic.

  VCPP is the standard C++ language with a special *class library* (MFC).
  VCPP 6.0 is the latest version (32-bit only).

- VCPP includes a windows based *interactive development environment* (IDE) which allows Windows program development to be done entirely within the Windows environment. This is called the *Visual Studio*.

- Specific software aids have been provided to make writing and debugging programs much more easily than was possible in the past.

- VCPP also includes a number of development tools from the Windows *Software Development Kit* (SDK). These tools have been improved over the years. The Win32 SDK is still available and includes some additional features.

The Windows Programming Model

- Unlike MSDOS programming, Windows is an *event driven* programming model. Events are passed to your program be means of *messages*.
  - This type of model is necessary for a number of reasons including:
    - More than one application may be running at a time and the user can send input to **any** application (e.g., mouse clicks).
    - The *graphical user interface* (GUI) allows input to come in many different ways (mouse click, key hits, etc.).
- Windows programs do not communicate directly with the video display hardware. Instead, the *Graphical Device* Interface, GDI provides a powerful graphics capability.
- Windows programming decouples certain aspects of the *look and feel* of the program into a separate *resource file*.
- 32-bit Windows programs require the virtual addressing capabilities of Windows 95/98 or Windows NT.

The Windows Programming Model

- Windows makes use of *dynamic linking*.

  Dynamic linking allows programs to be smaller and for library routines to be shared among running programs.

  Library code is only loaded into memory if it is needed.

- Windows NT and Windows 95/98 are 32-bit programming environments. The *application programming interface* (API) for these systems is a superset of the now obsolete Windows 3.1 16-bit API with changes to allow 32-bit arguments.

The VCPP Components

- The Visual Studio.

  The overall interactive development environment that controls the build process.

- The Graphic Editor.

  Used to edit resources.

- The C/C++ Compiler

- The Linker

  Handles the special .EXE format for Windows programs.

- The Resource Compiler

  Compiles resources created with the Graphic Editor.

- The Integrated Debugger

  Moderately powerful debugger integrated within the IDE.

- The AppWizard

  A simple code generator to provide a skeleton application with various options.

The VCPP Components

- The Class Wizard

  Allows the easy management of classes in your application. Will insert and delete classes and member functions.

  You write the function bodies, the Class Wizard provides the skeleton in the right place.

- The Source Browser

  Very nifty tool for finding things. Lets you look at the class hierarchy, definitions and references, etc.

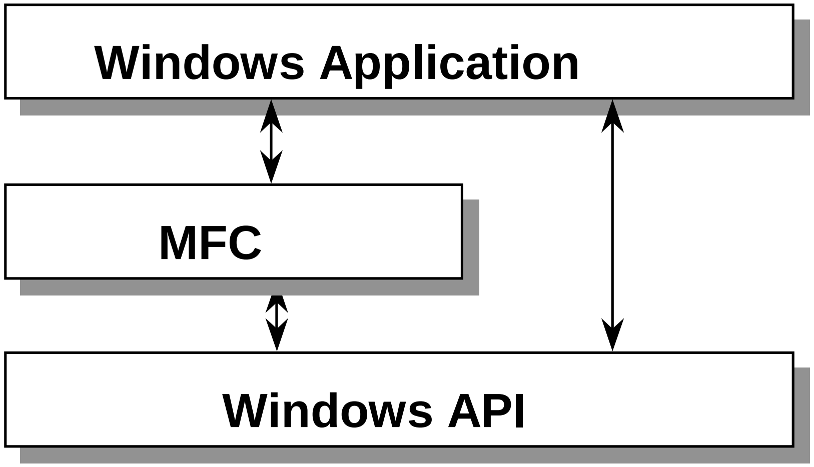
- Online Help

  Context sensitive help system. Throw out the hard copy manuals? I wouldn’t! You need to buy these as a separate product from Microsoft Press. A good investment!

The Microsoft Foundation Class Library

- MFC is the major component of an *application framework* for Windows development.
- An application framework is a paradigm for designing applications.

  It is more than just a collection of classes. It specifies the way in which the classes are to be used to build applications.
- An application framework makes portability and reusability easier.
- MFC encapsulates the Windows API.



MFC Features - partial list

- General purpose (non-Windows) classes.
  Collections, strings, time, file access, etc.
- Single rooted class hierarchy.
- Support for the Single and Multiple Document Interfaces (SDI and MDI).
- Support for Object Linking and Embedding (OLE), now part of COM, the component object model.
- Full file support.
- Printing capabilities.
  Print, Print review
- Scrolling and splitter windows.
- Database support - ODBC and DAO.
- OLE Controls (OCX), which replace the older VBX's.

MFC Features - contd.

- Docking toolbars.
- Status bars.
- Context sensitive help.
- Automatic access to dialog box data.
- DLL support.
- Networking through *Windows Sockets.*
- Lots of other stuff.

Minimal MFC Program

DEMO.H

```
//Simple MFC skeleton application - demo.h

class CDemoAp: public CWinApp
{
public:
  virtual BOOL InitInstance();
};

//Our frame window
class CDemoFrame: public CFrameWnd
{
public:
  CDemoFrame();
protected:
  afx_msg void OnLButtonDown(UINT nflags, CPoint point);
  afx_msg void OnPaint();
  DECLARE_MESSAGE_MAP()
};
```

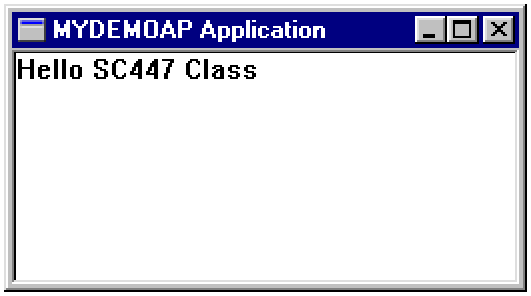
DEMO.CPP

```
#include <afxwin.h>
#include "demo.h"

//The single application class object
CDemoAp theAp;

BOOL CDemoAp::InitInstance()
{
 m_pMainWnd = new CDemoFrame();
 m_pMainWnd->ShowWindow(m_nCmdShow);

 m_pMainWnd->UpdateWindow();
 return TRUE;
}

BEGIN_MESSAGE_MAP(CDemoFrame, CFrameWnd)
 ON_WM_LBUTTONDOWN()
 ON_WM_PAINT()
END_MESSAGE_MAP()

CDemoFrame::CDemoFrame()
{
 Create(NULL, "MYDEMOAP Application");
}
```

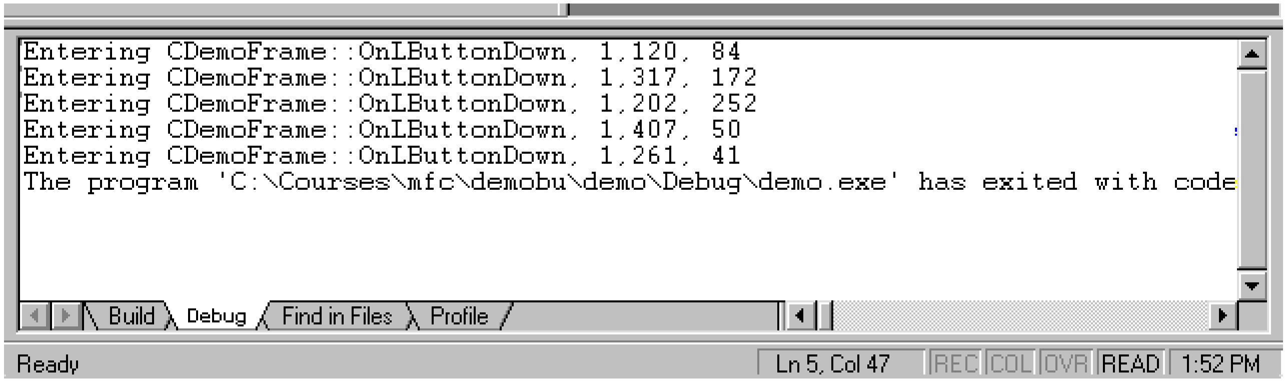
DEMO.CPP -contd.

```
void CDemoFrame::OnLButtonDown(UINT nFlags, CPoint point)
{
 //Use the TRACE macro
 TRACE("Entering CDemoFrame::OnLButtonDown, %lx,
        %d, %d\n", (long)nFlags, point.x, point.y);
}

void CDemoFrame::OnPaint()
{
 CPaintDC dc(this);

 //note-the constructor for CPaintDC calls the SDK
 //BeginPaint() and the destructor calls the SDK EndPaint()

 dc.TextOut(0, 0, "Hello SC447 Class");
}
```

Demo Program - output



Hello SC447 Class



```
Entering CDemoFrame::OnLButtonDown, 1,120, 84
Entering CDemoFrame::OnLButtonDown, 1,317, 172
Entering CDemoFrame::OnLButtonDown, 1,202, 252
Entering CDemoFrame::OnLButtonDown, 1,407, 50
Entering CDemoFrame::OnLButtonDown, 1,261, 41
The program 'C:\Courses\mfc\demobu\demo\Debug\demo.exe' has exited with code
```

MFC Application Structure

- *WinMain()*

  Windows calls a function *WinMain* instead of *main*. WinMain is inside the MFC classes and you don’t see it.

- The *CDemoAp* class

  This is the main application class. There is only one main application class and it is derived from *CWinAp*.

- *CDemoAp::InitInstance()*

  MFC calls this virtual function to allow the application to construct its main window. Why does this function have to be virtual?

- *CWinAp::Run()*

  This function in the base class is called by MFC and enters the message dispatch loop (familiar to SDK programmers).

- *CDemoAp::OnPaint()*

  This function is used to handle the WM_PAINT message from Windows and is called whenever the window needs repainting.

MFC Application Structure - contd.

- *CDemoAp::OnLButtonDown()*

  This function is called every time the user presses the left mouse button. It handles the WM_LBUTTONDOWN message from Windows.

  The demo program uses the TRACE macro to output a message to the debug window.

- Closing the demo application

  MFC provides the mechanism to end an application. Closing the application in any of the normal ways causes the run function to exit and hence WinMain.

Windows Message Mapping

- In SDK programming, Windows messages are sent to a *window procedure* or *WndProc* as it is usually known as. The programmer usually uses a large *switch* statement to handle the messages.
- MFC uses a different technique.

  Windows messages are directed to an appropriate class member function directly. You don't need to set up this large switch statement.
- Does MFC use virtual functions or special compiler features?

  NO! There is too much overhead with virtual functions and special compiler features reduce portability of the library. Borland tried this with OWL 1.0 and abandoned the idea with OWL 2.0.
- MFC (and also OWL) uses a *message mapping* technique based around the use of macros.

  The MFC tools set this all up for you and you don’t need to know the inner workings.

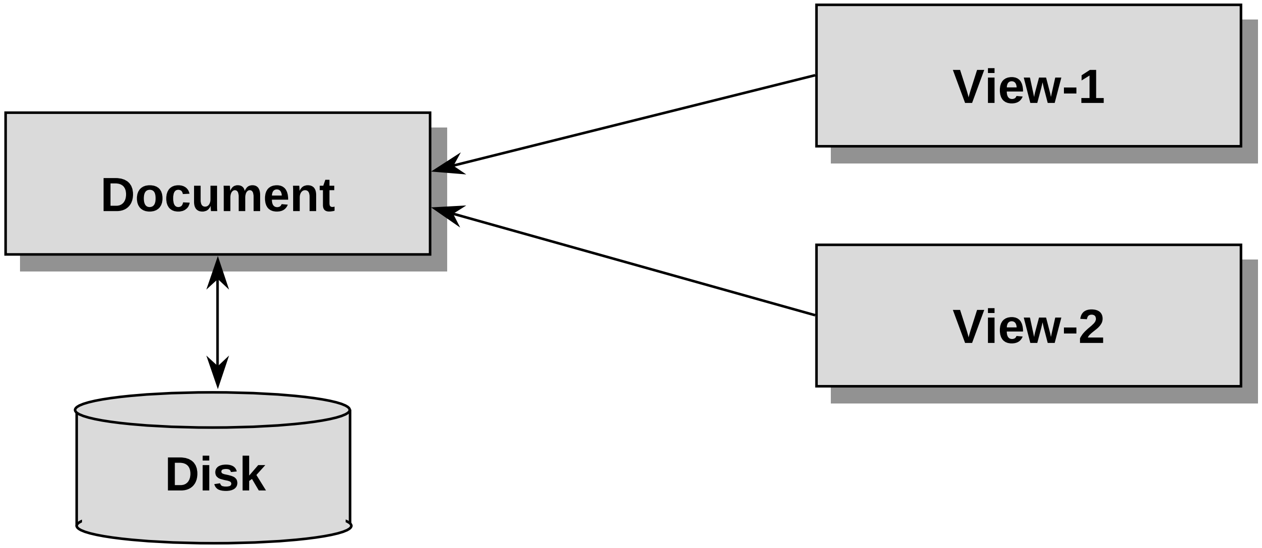
The Document View Architecture

- The *document view* architecture is central to virtually all current windows applications.
- The document is the data the application is operating on.

  This should not be taken literally as a document, although it may well be. A spreadsheet or database would be a suitable document. Consider a flight simulator. The document would be the data comprising the scenery.
- The view is the users visual representation of the data on the screen.

  A word processor document might be viewed at a particular point in the document and at a particular zoom. The flight simulator scenery might be a view out the side window of the aircraft.
- Applications may support more than one view on a document.
- Applications can also support more than one document at a time.

The Document View Architecture

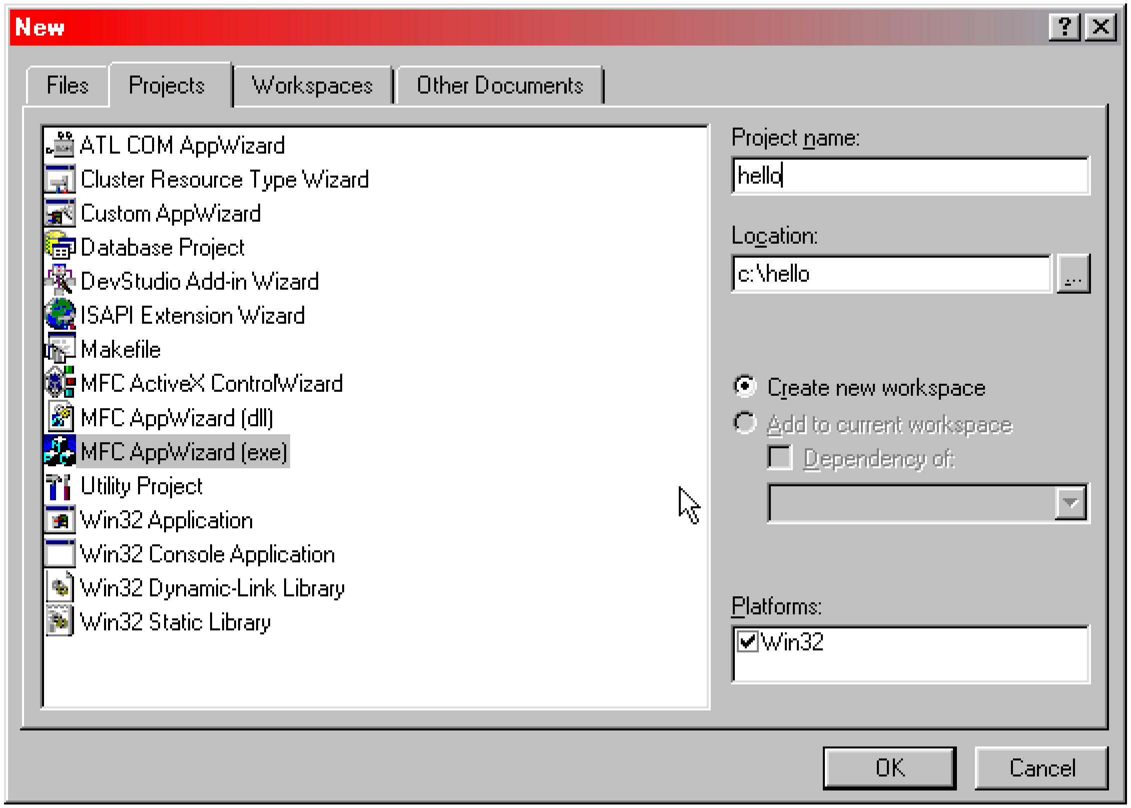


- Do not equate a document with a single disk file.
- However, documents are normally stored in one or more disk files
- In MFC, documents and views will be represented by their own classes.

SDI vs. MDI

- The *single document interface*, SDI, is an application structure in which only one document can be open and displayed at a time.
- The *multiple document interface*, MDI, allows zero or more different documents to be open and displayed simultaneously.
- MFC supports both application types and the AppWizard allows building a skeleton for either.
- We will cover SDI applications first, but the overall structure is similar and it is possible to convert an SDI application to MDI without a lot of work.
  - It’s not automatic though!

Using ApWizard to generate an MFC application



Project name: hello

Location: c:\hello

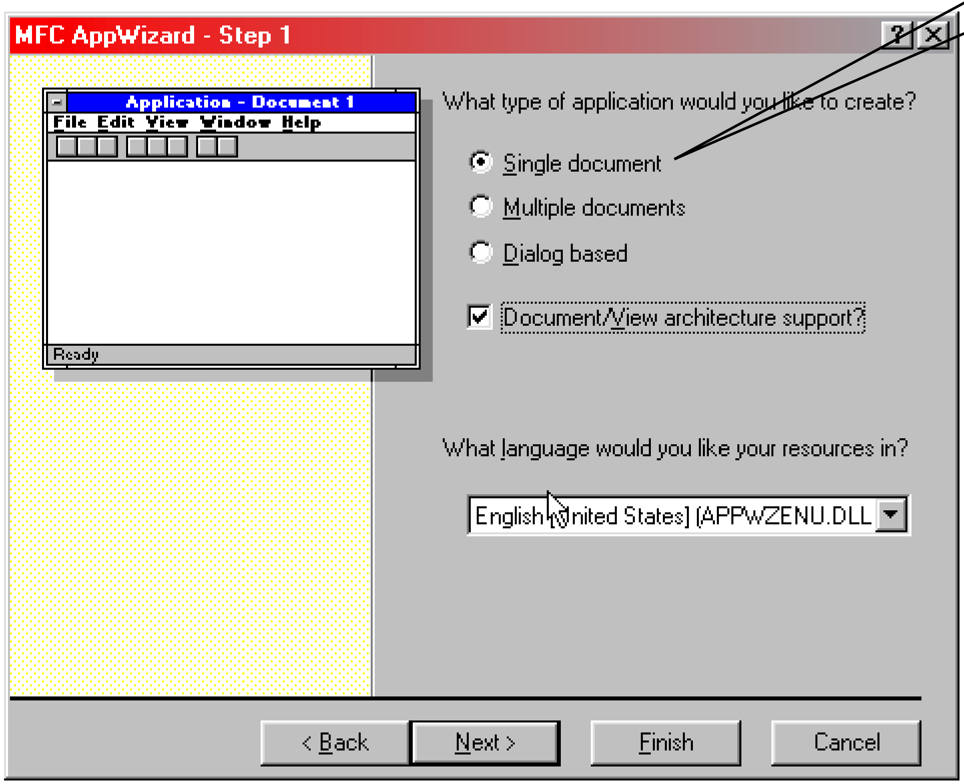
● Create new workspace

○ Add to current workspace

□ Dependency of:

Platforms: ☑ Win32

Set to single document



What type of application would you like to create?

● Single document

○ Multiple documents

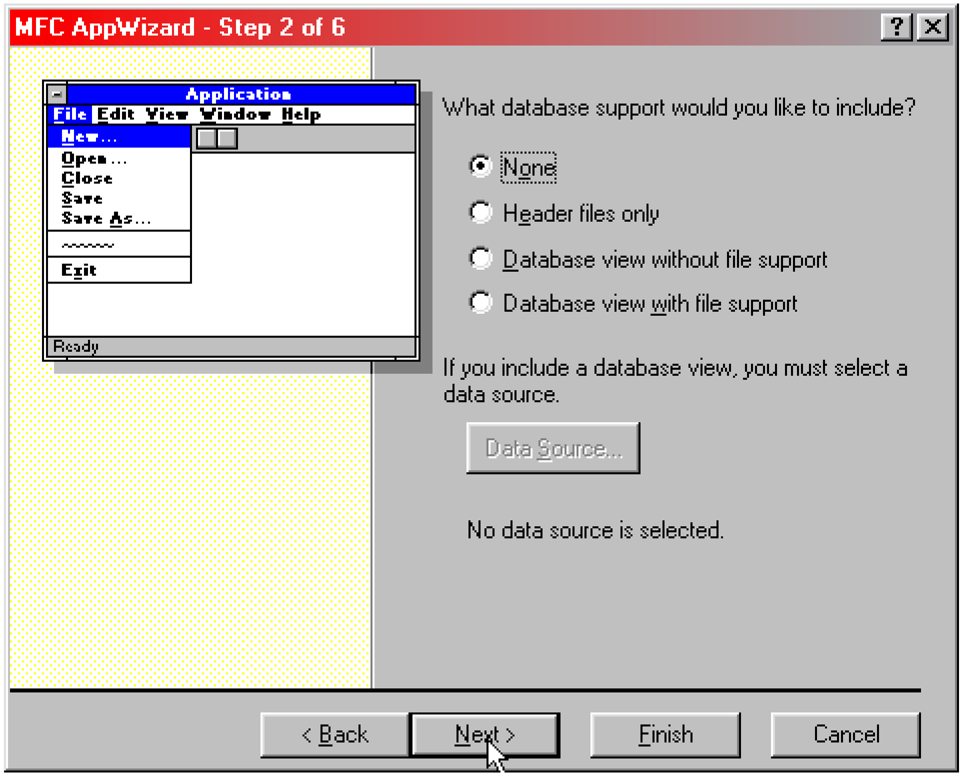
○ Dialog based

☑ Document/View architecture support?

What language would you like your resources in?

English (United States) (APPWZENU.DLL

No changes



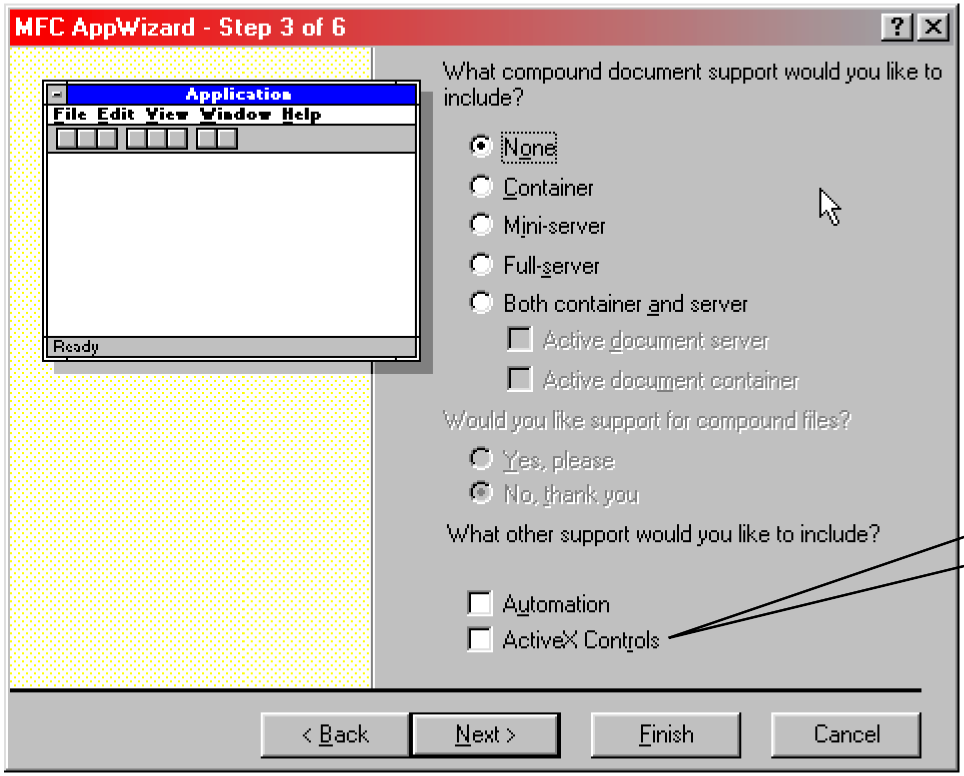
What database support would you like to include?

● None
○ Header files only
○ Database view without file support
○ Database view with file support

If you include a database view, you must select a data source.

No data source is selected.

No changes



MFC AppWizard - Step 3 of 6

What compound document support would you like to include?

- ● None
- ○ Container
- ○ Mini-server
- ○ Full-server
- ○ Both container and server
  - □ Active document server
  - □ Active document container

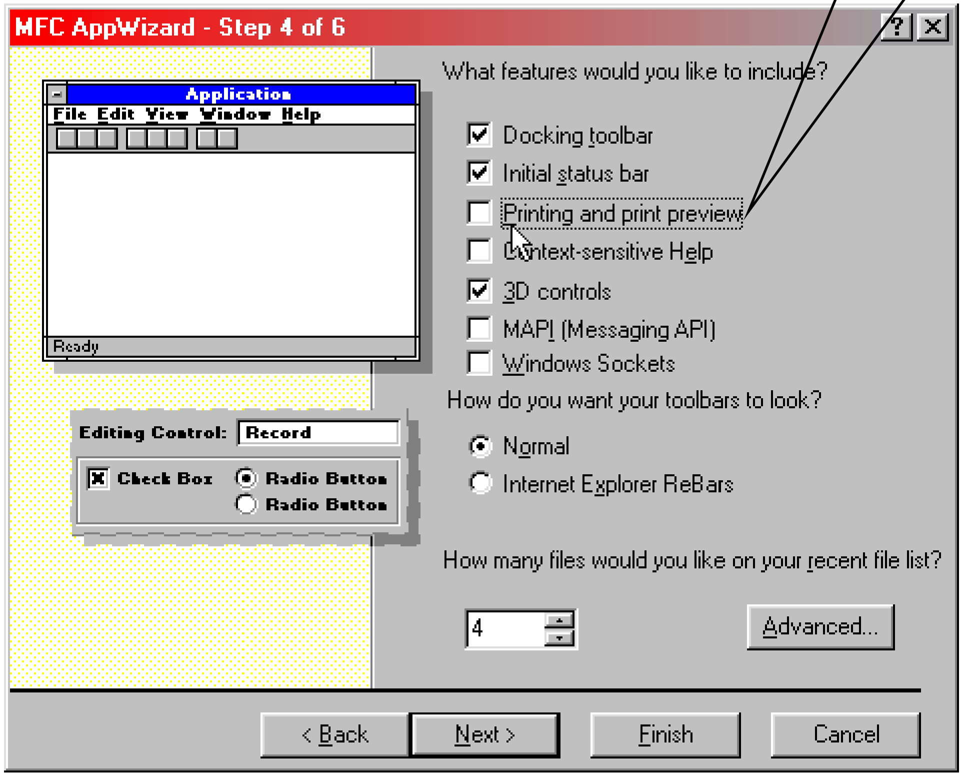
Would you like support for compound files?

- ○ Yes, please
- ● No, thank you

What other support would you like to include?

- □ Automation
- □ ActiveX Controls

Uncheck



MFC AppWizard - Step 4 of 6

What features would you like to include?

- ☑ Docking toolbar
- ☑ Initial status bar
- □ Printing and print preview
- □ Context-sensitive Help
- ☑ 3D controls
- □ MAPI (Messaging API)
- □ Windows Sockets

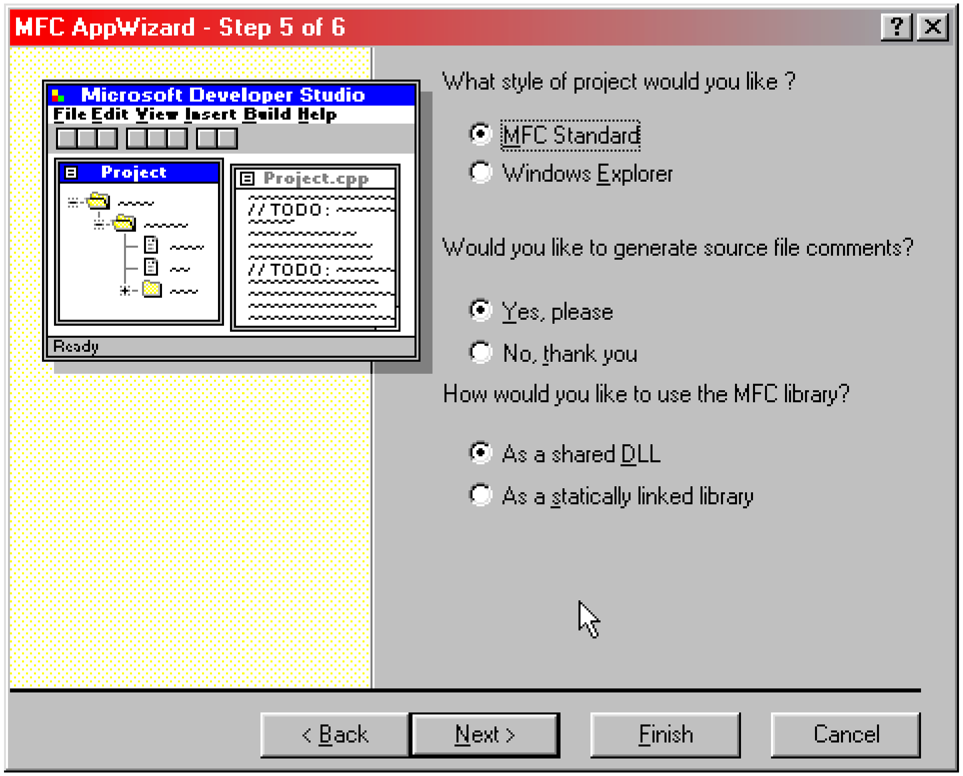
How do you want your toolbars to look?

- ● Normal
- ○ Internet Explorer ReBars

How many files would you like on your recent file list? 4

Uncheck

No changes



MFC AppWizard - Step 5 of 6
Microsoft Developer Studio
File Edit View Insert Build Help

What style of project would you like ?

● MFC Standard

○ Windows Explorer

Would you like to generate source file comments?

● Yes, please

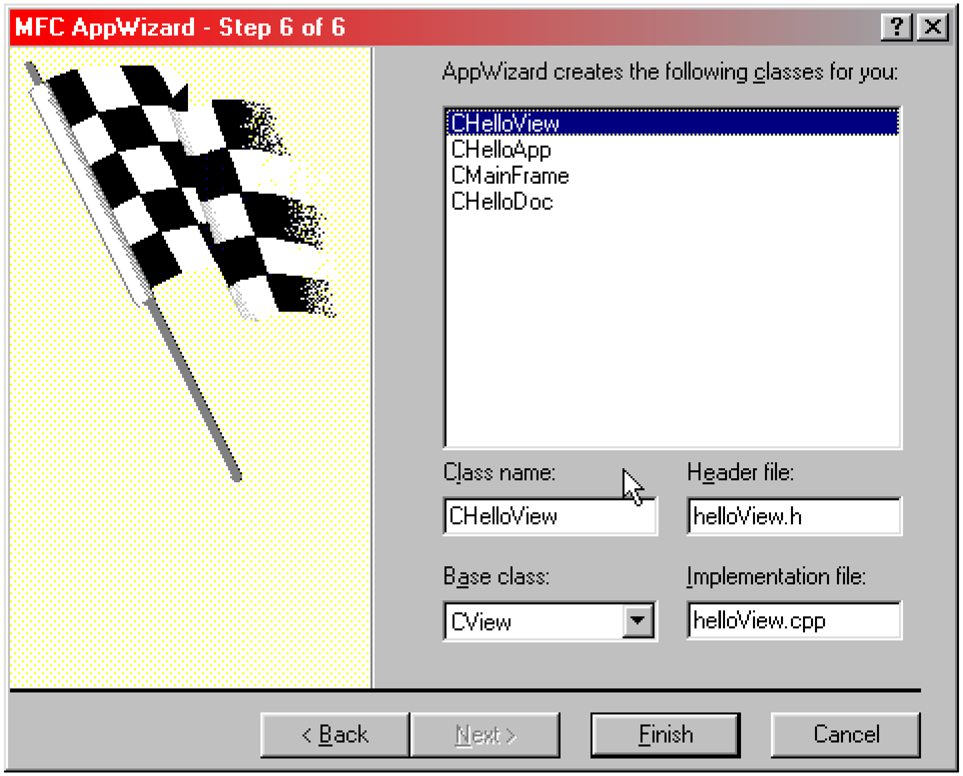
○ No, thank you

How would you like to use the MFC library?

● As a shared DLL

○ As a statically linked library

No changes



MFC AppWizard - Step 6 of 6

AppWizard creates the following classes for you:

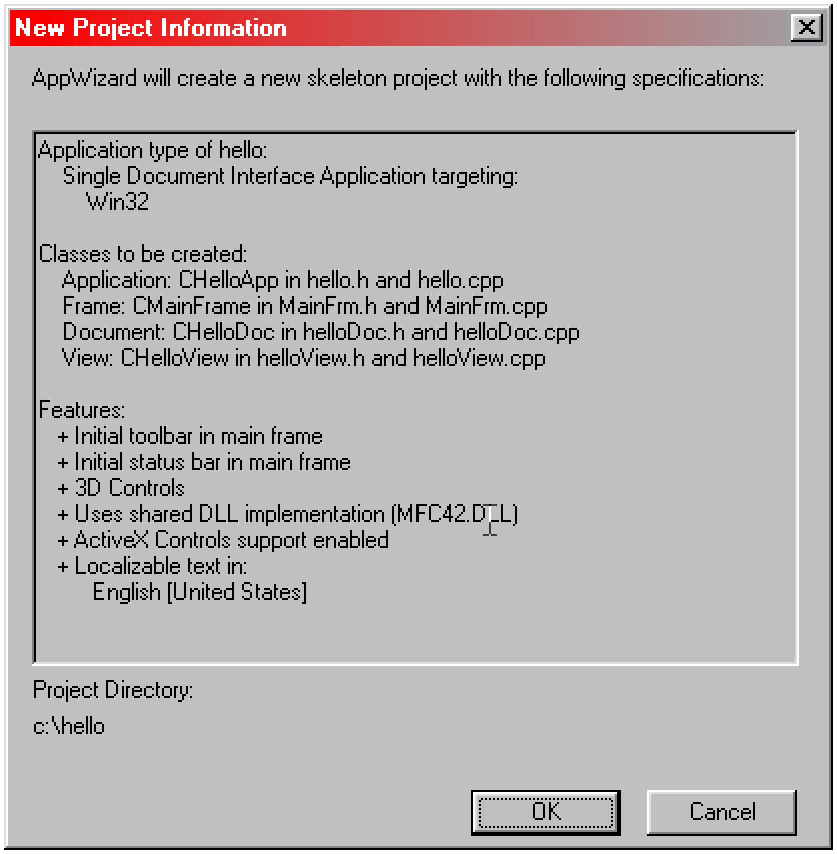
CHelloView
CHelloApp
CMainFrame
CHelloDoc

Class name: CHelloView

Header file: helloView.h

Base class: CView

Implementation file: helloView.cpp

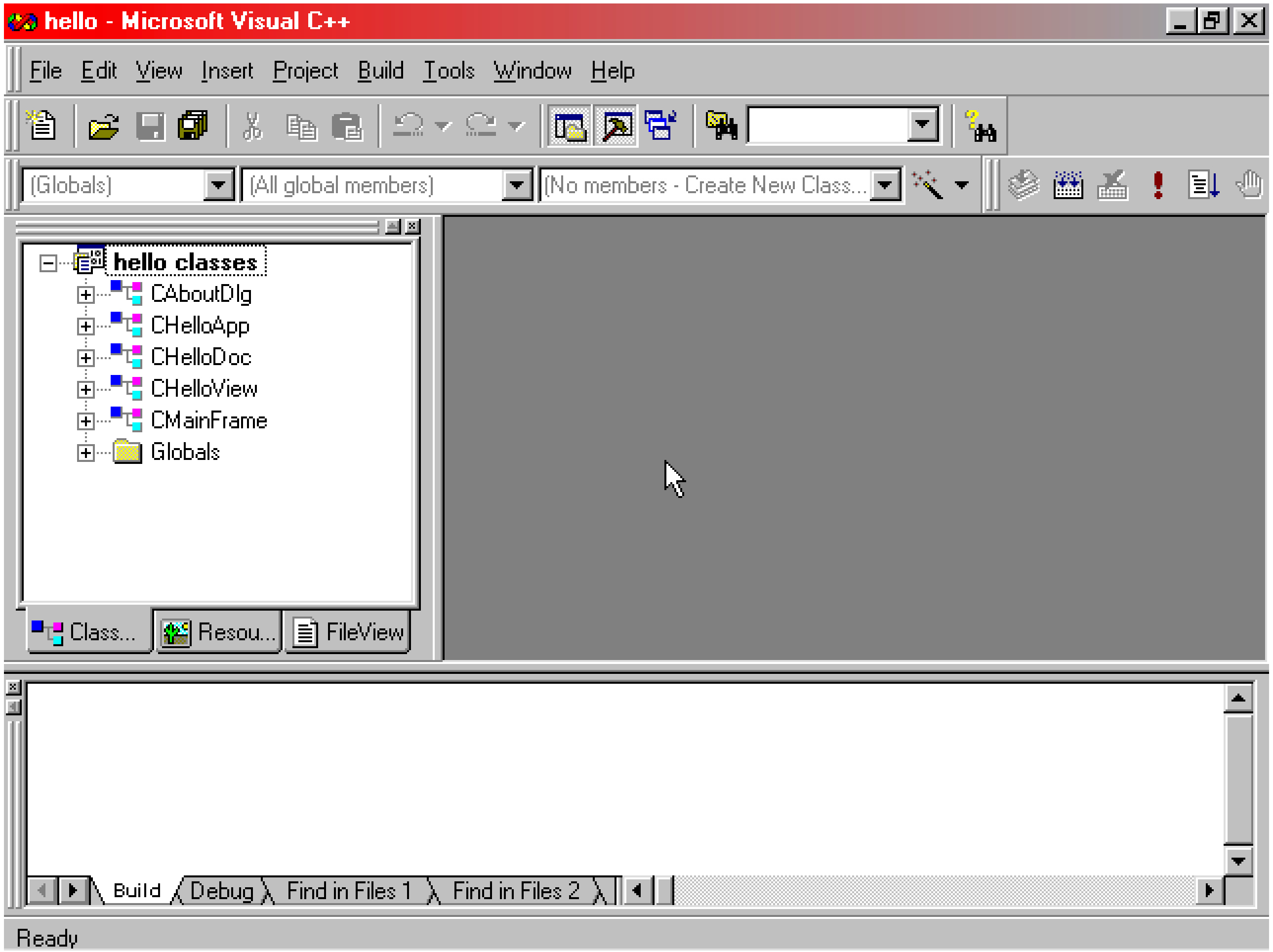


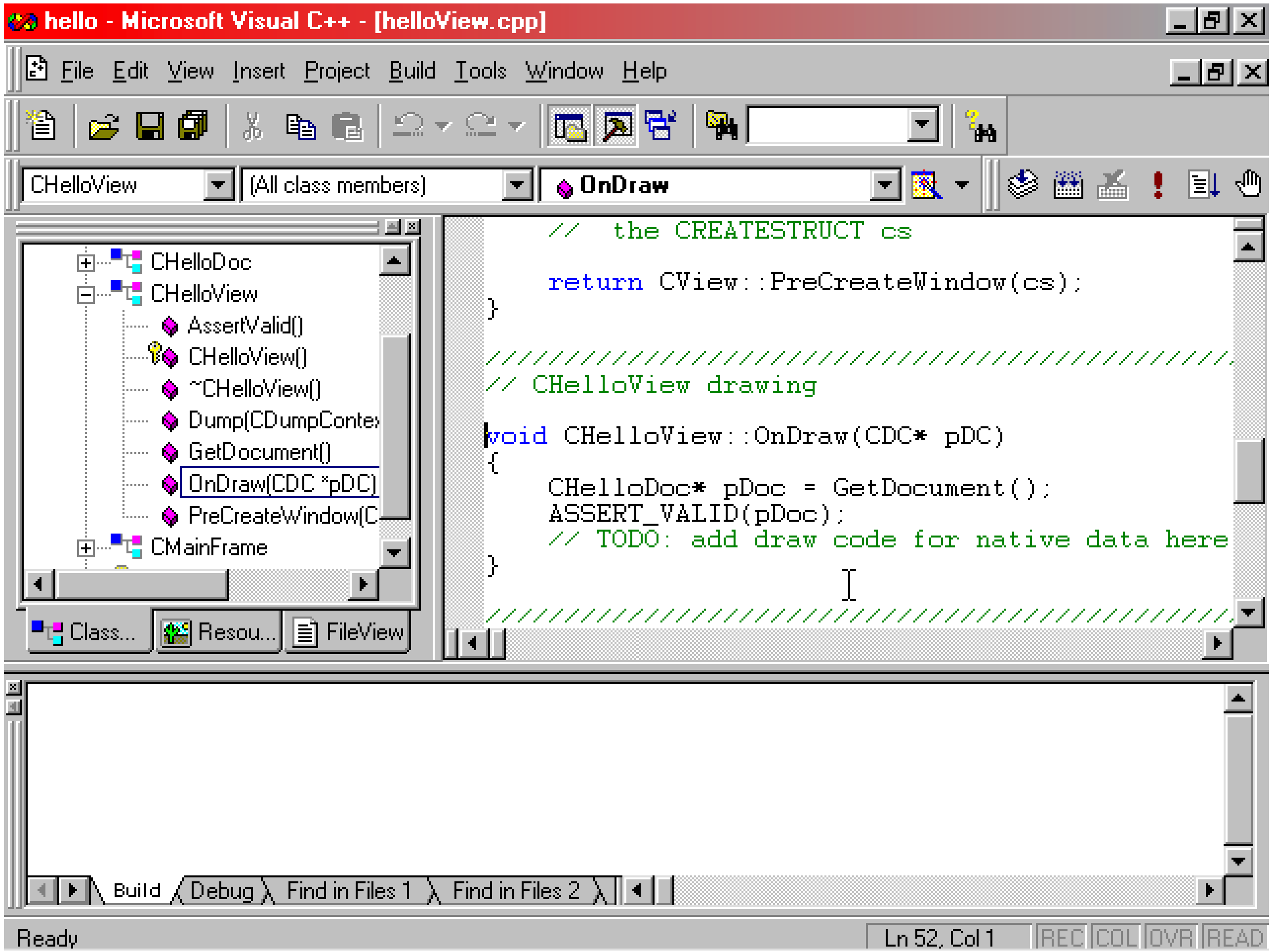
AppWizard will create a new skeleton project with the following specifications:

```
Application type of hello:
    Single Document Interface Application targeting:
        Win32

Classes to be created:
    Application: CHelloApp in hello.h and hello.cpp
    Frame: CMainFrame in MainFrm.h and MainFrm.cpp
    Document: CHelloDoc in helloDoc.h and helloDoc.cpp
    View: CHelloView in helloView.h and helloView.cpp

Features:
   + Initial toolbar in main frame
   + Initial status bar in main frame
   + 3D Controls
   + Uses shared DLL implementation (MFC42.DLL)
   + ActiveX Controls support enabled
   + Localizable text in:
         English [United States]
```

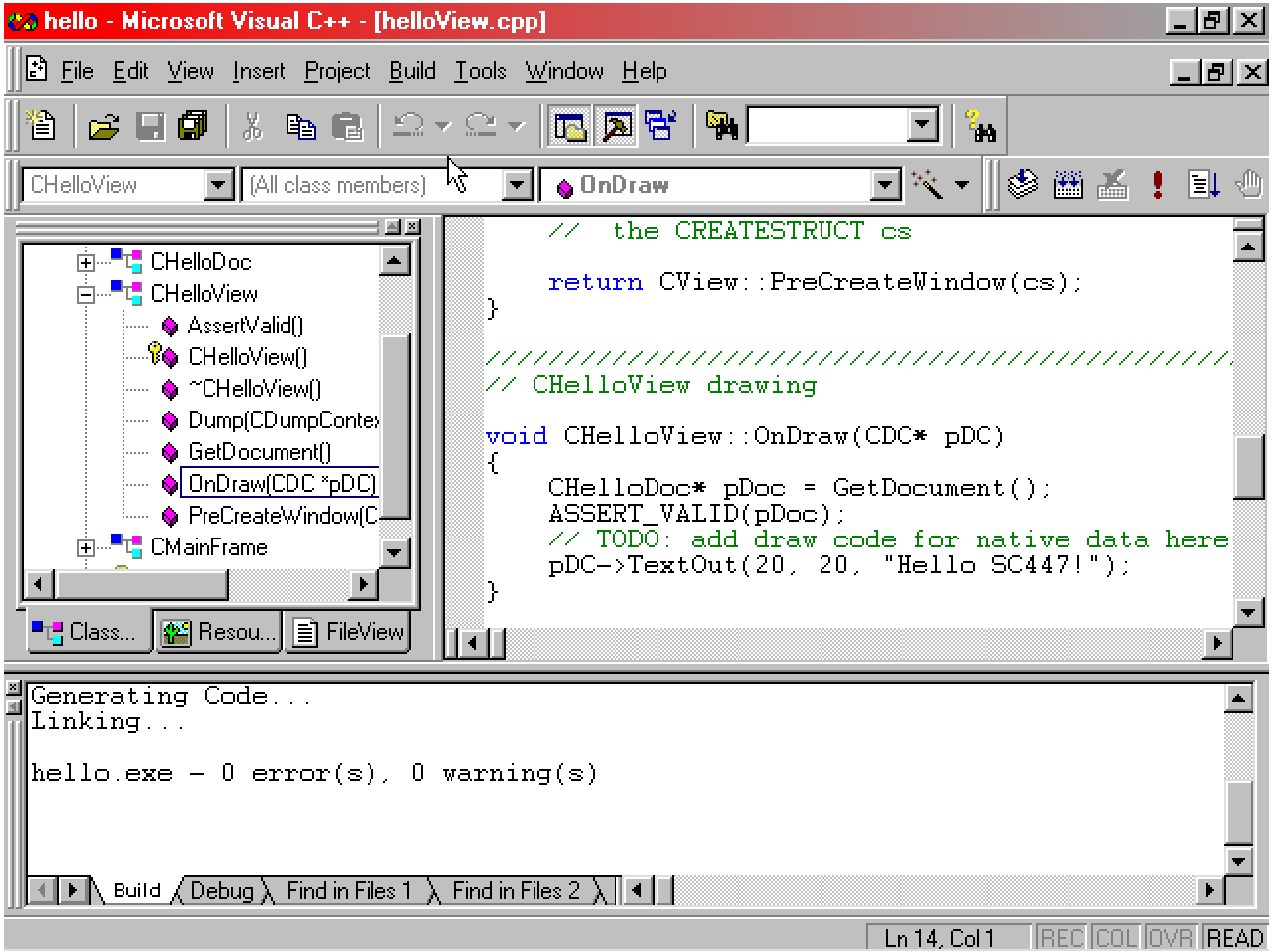
Project Directory:

c:\hello


hello - Microsoft Visual C++
File Edit View Insert Project Build Tools Window Help
(Globals)
(All global members)
(No members - Create New Class...
hello classes
CAboutDlg
CHelloApp
CHelloDoc
CHelloView
CMainFrame
Globals
Class...
Resou...
FileView
Build
Debug
Find in Files 1
Find in Files 2
Ready



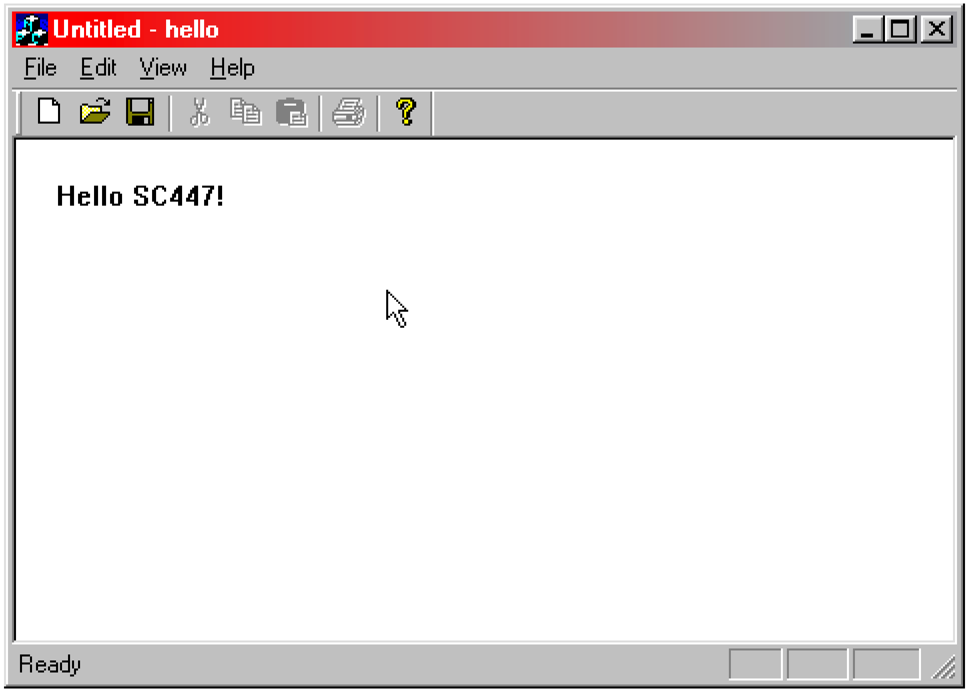
```
    //  the CREATESTRUCT cs

    return CView::PreCreateWindow(cs);
}

////////////////////////////////////////////////
// CHelloView drawing

void CHelloView::OnDraw(CDC* pDC)
{
    CHelloDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
}

////////////////////////////////////////////////
```



```
    //  the CREATESTRUCT cs

    return CView::PreCreateWindow(cs);
}

////////////////////////////////////////////////////////////////////////////
// CHelloView drawing

void CHelloView::OnDraw(CDC* pDC)
{
    CHelloDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
    pDC->TextOut(20, 20, "Hello SC447!");
}
```

```
Generating Code...
Linking...

hello.exe - 0 error(s), 0 warning(s)
```

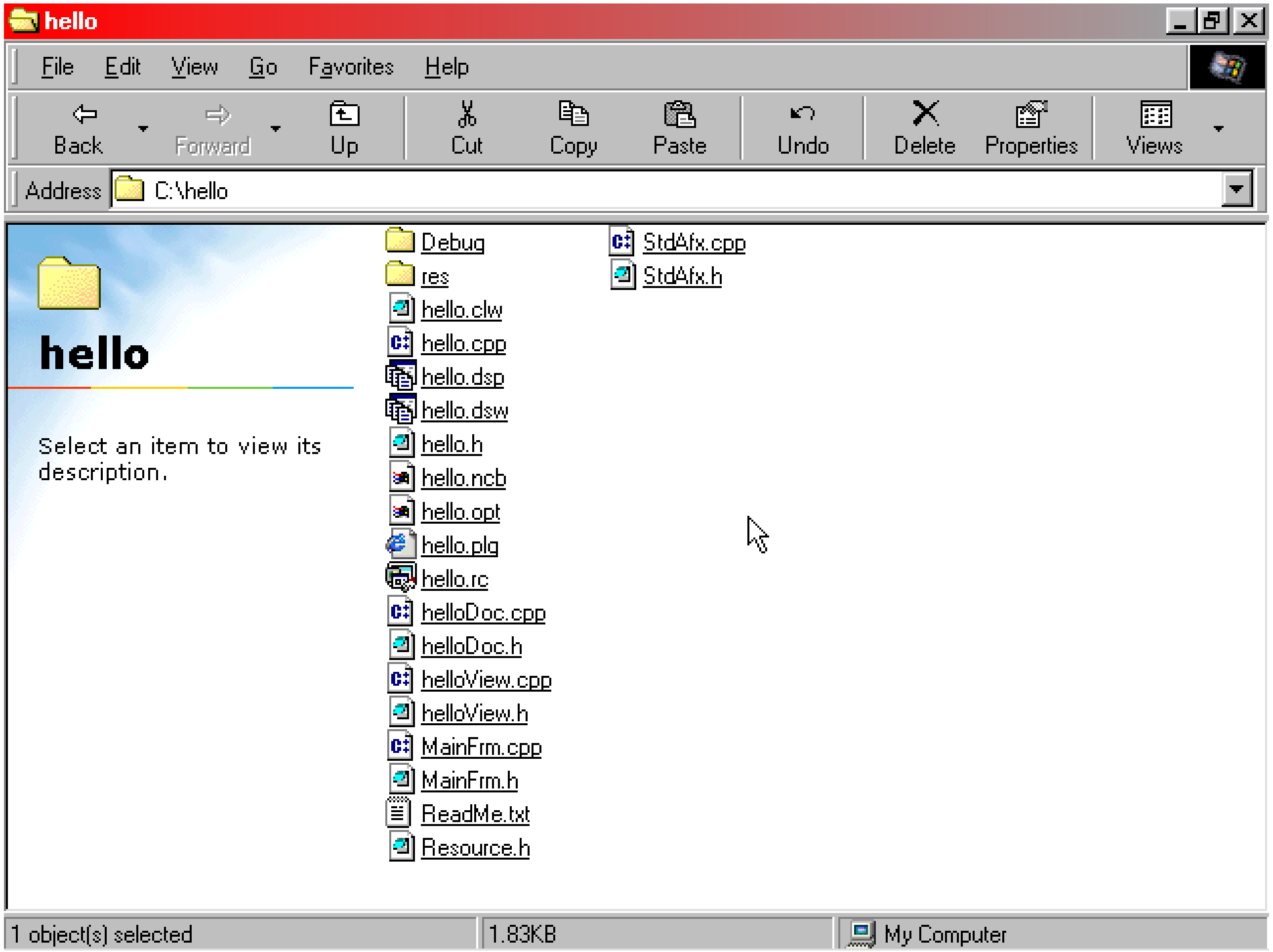
The program's execution



Hello SC447!

Ready

Files generated



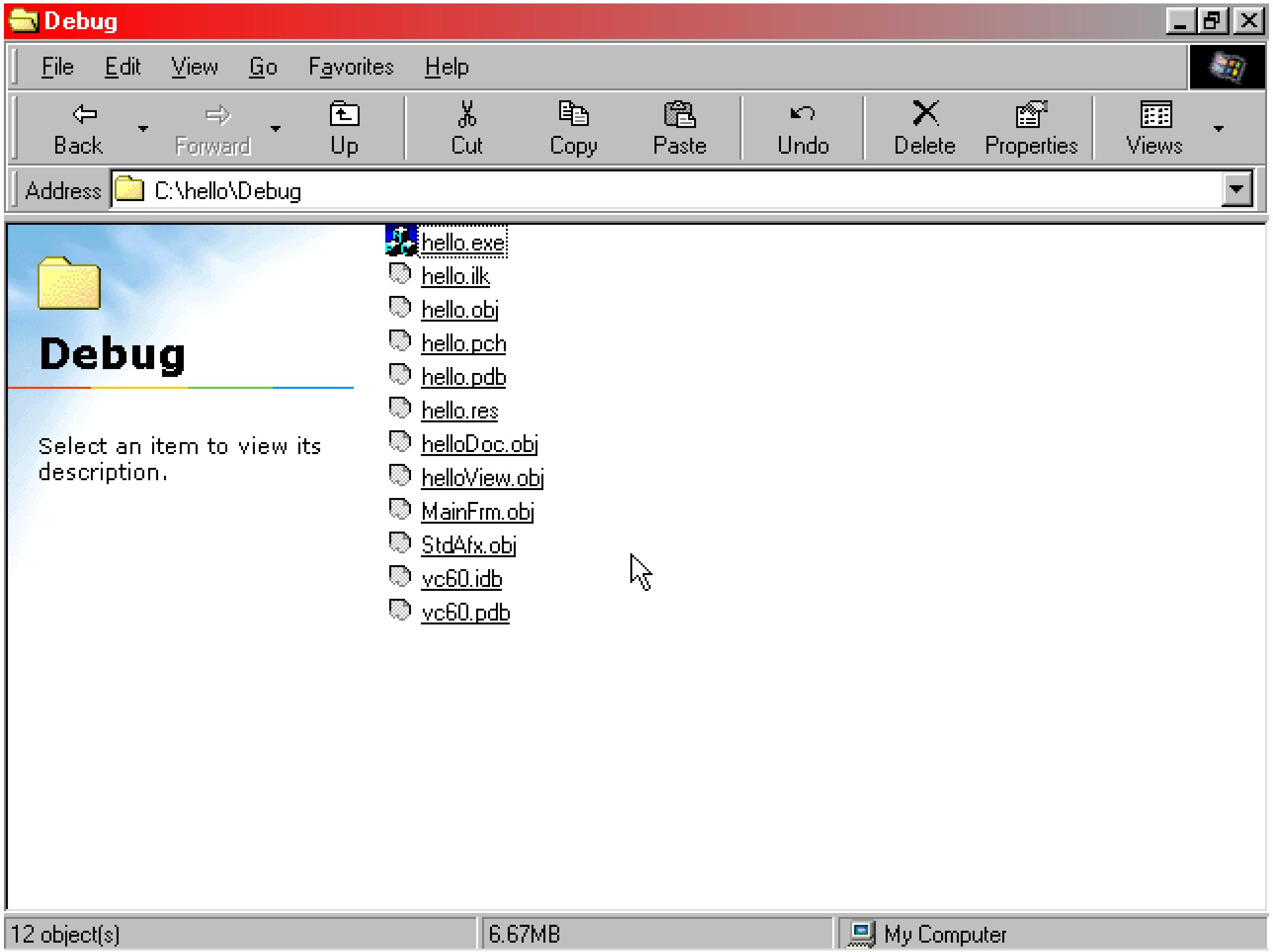
C:\hello

hello

Select an item to view its description.

Debug
res
hello.clw
hello.cpp
hello.dsp
hello.dsw
hello.h
hello.ncb
hello.opt
hello.plg
hello.rc
helloDoc.cpp
helloDoc.h
helloView.cpp
helloView.h
MainFrm.cpp
MainFrm.h
ReadMe.txt
Resource.h
StdAfx.cpp
StdAfx.h

1 object(s) selected | 1.83KB | My Computer



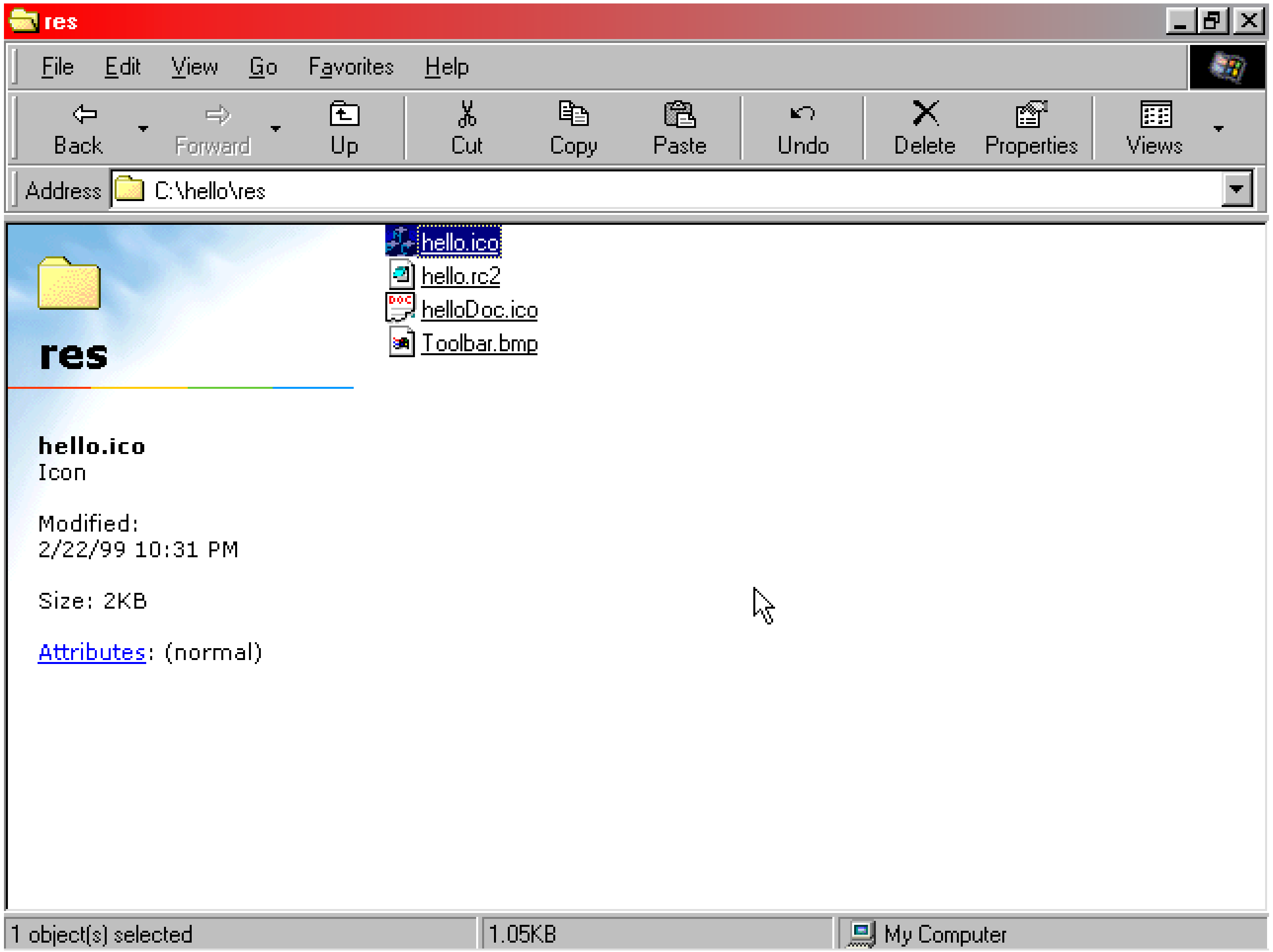
Address: C:\hello\Debug

Debug

Select an item to view its description.

hello.exe
hello.ilk
hello.obj
hello.pch
hello.pdb
hello.res
helloDoc.obj
helloView.obj
MainFrm.obj
StdAfx.obj
vc60.idb
vc60.pdb

12 object(s) | 6.67MB



C:\hello\res

hello.ico
hello.rc2
helloDoc.ico
Toolbar.bmp

res

hello.ico
Icon

Modified:
2/22/99 10:31 PM

Size: 2KB

Attributes: (normal)

1 object(s) selected | 1.05KB | My Computer

Files Created by AppWizard

PROJ.DSP	Project file (builds project)
PROJ.CPP	The main application class
PROJ.H	Header file for above
MAINFRM.CPP	The main frame window class
MAINFRM.H	Header file for above
PROJVIEW.CPP	The view class code
PROJVIEW.H	Header file for above
PROJDOC.CPP	The document class code
PROJDOC.H	Header file for above
PROJ.RC	Resource script file
RESOURCE.H	Constant definitions associated with the project resources
README.TXT	Description of the files generated by the AppWizard
PROJ.DSW	A workspace file (sets up the workspace
STDAFX.CPP/H	Used to generate the precompiled header

- **There are numerous other support files used with the Visual C++ environment.**

***CWnd* and *CView* classes**

- *CWnd* is the generic base class from which all windows are derived. It is possible to use on object of this type, but usually a derived class is a better choice because it will have the added functionality you need.
- *CView* is a simple view class. It provides the necessary interface to the rest of the application.
- The view is actually a borderless window that occupies the *client area* of the main frame window to be discussed later.

  The client area of a window is the area where application output is displayed. It does not include the caption bar, menu bar, scroll bars, or the window's border (if it has one).

Drawing in the View

- The *OnDraw* virtual member function of the CView class is used to output text and graphics to the view window.
- OnDraw is called by Windows (and hence through MFC) whenever the window or a portion of the window needs updating.
- The OnDraw function is passed a *device context* already set up for your use. It will be a device context to the client area of the view window.

```
/////////////////////////////////////////////////////////////////////////////
// CDemoView drawing

void CDemoView::OnDraw(CDC* pDC)
{
  pDC->TextOut(0, 0, "Hello, world!");  // prints in default font &
                                        //  size, top-left corner
  pDC->SelectStockObject(GRAY_BRUSH);   // selects a brush for the
                                        //  circle interior
  pDC->Ellipse(CRect(0, 20, 100, 120)); // draws a gray circle 100
                                        //  units in diameter
}
```

The Resource Editors

- The *Resource Editors* allow editing resources graphically. They are used to edit and create the resources used with your application.
- Resources include:

  Menus
  Icons
  Bitmaps
  Dialogs
  String tables
  Keyboard accelerators
  Toolbar Bitmaps
  Version Information

- The Resource Editors can be accessed directly from the *Developer Studio*. With a workspace open, the project’s resources will be displayed in the project workspace under the resources tab.

Precompiled Headers

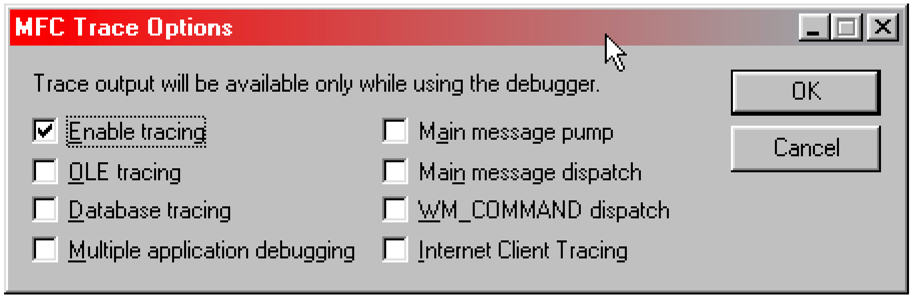
- Precompiled headers allow much quicker recompilation of modules in your project once an initial build has been performed.
- VCPP generates a file, *proj*.pch, which contains this information. It is usually large.
- The file *stdafx.cpp* is always added to your project. This file contains only #includes for the headers required. When it compiles it doesn't generate any code. It just forces the creation of the precompiled header file prior to compilation of the remaining project files.
- Precompiled headers is turned on by default.

Debug vs. Release Targets

	Release Build	Debug Build
Source code debugging	Disabled	Enabled for both compiler and linker
MFC diagnostic macros	Disabled (NDEBUG defined)	Enabled (_DEBUG defined)
Library linkage	MFC Release libraries	MFC Debug libraries
Compiler optimization	Speed optimization	No optimization (faster compile)

Diagnostic Macros

- The TRACE macro works like a *printf()* function call.
- Output goes to the debug window in the Visual Studio.
- You must enable these macros using the *TRACER* utility.
- You must also compile your program as a debug version and run it under the debugger for output to be generated.
- Release builds suppress code generation for TRACE macros (and others such as VERIFY).



MFC Trace Options

Trace output will be available only while using the debugger.

☑ Enable tracing
□ OLE tracing
□ Database tracing
□ Multiple application debugging
□ Main message pump
□ Main message dispatch
□ WM_COMMAND dispatch
□ Internet Client Tracing

Creating a New Project Workspace

- You can create a new workspace and project without the use of the AppWizard, or in the case where the project files are lost.
- Choose *New/Project* from the *File* menu, but do not use the AppWizard. Create the application type of your choice, e.g. WIN32.exe
- Insert the .CPP, .H, .RC, and other files for your project using *Insert/Files into Project*.
- You can set or change project options through the *Project/Settings* menu item.
- To save a project using minimal space, save:

  *.H

  *.CPP

  *.RC and res folder

  *.DSP

  *.DSW

The ClassWizard

- The *ClassWizard* is another automatic programming feature of VCPP that takes the drudgery out of coding.
- The ClassWizard will create and modify MFC classes and classes derived from MFC classes.
- The primary use of the ClassWizard is to create and manage member functions to handle Windows messages.
- As was mentioned earlier, Windows messages are passed to member functions directly without the use of virtual functions. This is accomplished by the use of a *message map*.
- ClassWizard will automatically manage the message map mechanism and create a skeleton member function that you can fill in.

  The message map contains macros of the form

  ```
  BEGIN_MESSAGE_MAP (......)
      ON_WM_mmmm()
  END_MESSAGE_MAP()
  ```

  The class header file also contains the line

  ```
  DECLARE_MESSAGE_MAP()
  ```

Where should application data go?

- Since we are working in C++, data should be placed within the appropriate class.
- Global variables should be avoided at all cost.

  Put them in the application class as a last resort.
- Data associated with the open document normally goes in the document class, not the view class. More than one view may be open at a time!
- By convention use *m_* as a prefix to all class data members.
- Constructors

  Don’t forget that data is best initialized in the constructor for the given class.

WM_PAINT Messages and *OnDraw()*

- Whenever some portion of the client area of a window needs to be updated, Windows sends you program a *WM_PAINT* message.
- Among the situations causing a paint are:

  1. Restoring a window from the taskbar (or icon)
  2. Uncovering a hidden portion of a window
  3. Changing a window's size
  4. By the program invalidating some part of the window

  Note - moving a window does not normally cause a paint.
- The *WM_PAINT* message is passed to *OnPaint()* in the *CWnd* class and it in turn calls *OnDraw()*. You normally do not provide a message handler for *OnPaint()*.

  OnDraw() normally serves a dual purpose since it is also called for printing. This eliminates the need to duplicate code.

Notes on the EX04A Example

- The rectangle changes color when the left mouse button is clicked inside the ellipse.
  - A handler for the *WM_LBUTTONDOWN* message is added using the Class Wizard (*OnLButtonDown()*).
- This is accomplished by changing the data member *m_nColor*.
- Note that *InvalidateRect()* is used rather than *Invalidate()*. This would result in a performance improvement in cases where there is a complex amount of data in the window.
  - It makes no noticeable difference in this example
- *SelectStockObject* and *Ellipse* are standard GDI functions used to select the color of a brush and to display an ellipse. These functions are passed through to the Windows API.
  - Note - many MFC functions are just passed through with the ***window handle*** automatically provided by MFC.

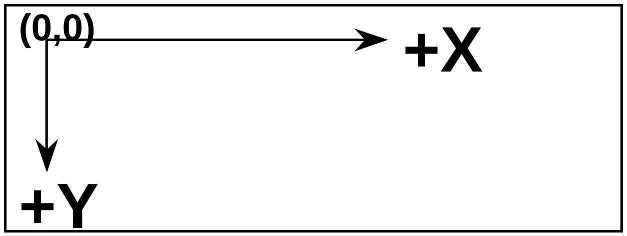
Regions and the Invalid Rectangle

- A number of functions are available to determine if a point is within a region and to manipulate geometric areas.
- To speed up Windows painting you should always strive to invalidate only the *smallest* rectangle necessary for update.
  - Windows will erase only the invalid rectangle and clip all output to the dimensions of the invalid rectangle.
- Among the classes you should study are:
  - *CRect*
  - *CPoint*
  - *CSize*
  - *CRgn*
- Common set operations such as *union* are provided.
- Note - *overloaded operators* are provided to allow class objects to be passed to and from Windows API functions.

Mapping Modes

- Device Coordinates

  Addressed in pixels with (0, 0) being the upper left hand corner of the screen, window, or client area. Note that positive Y is down.



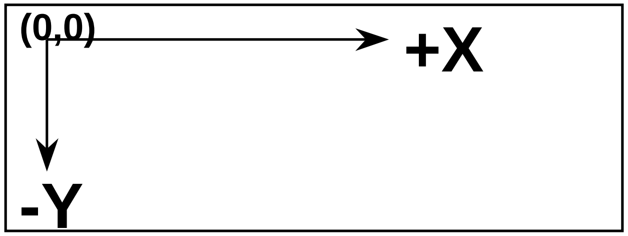
- The default *mapping mode* is *MM_TEXT*. In this mode *logical coordinates* are the same as *device coordinates*.
- It is not always desirable to work in *MM_TEXT* mapping mode (even for text as we will see).

Mapping Modes - contd.

- Eight mapping modes are available.

Mapping Mode	Logical Unit
MM_TEXT	Pixels
MM_LOENGLISH	0.01 inch
MM_HIGHENGLISH	0.001 inch
MM_LOMETRIC	0.1 mm
MM_HIMETRIC	0.01 mm
MM_TWIPS	1/1440 inch
MM_ISOTROPIC	Variable (fixed aspect)
MM_ANISOTROPIC	Variable

All but MM_TEXT have the following coordinate system:



- Note - the origin is still the upper left hand corner, but the sign of the Y axis is reversed.

The *Viewport*

- The *viewport* is the *physical* display. It is always addressed in **device coordinates**.
- The *window* is the virtual output device and we use **logical coordinates**.
- Normally the viewport and window origins correspond.
- You can use the *SetViewportOrg()* function to place the logical origin anywhere in the physical device window.

```
pDC->SetViewportOrg(100, 100);
```

- Now if you output something at logical coordinate (0, 0) it will be at (100, 100) on the screen.
- *SetWindowOrg()* works similarly.It sets the viewport origin anywhere in the logical window. Use logical coordinates.
- You shouldn’t use both calls at the same time.

MM_ISOTROPIC **and** ***MM_ANISOTROPIC*** **Modes**

- These mapping modes allow complete flexibility in setting the scaling to be used between logical and device coordinates.
- MM_ISOTROPIC mode retains the *aspect ratio* of one to one regardless of the scaling. A circle will always be a circle.
- MM_ANISOTROPIC allows *fitting* objects to a window by a combination of scaling **and** changing the aspect ratio.
- All this works by the use of *extents*. Extents are used to form a ratio for scaling.

MM_ISOTROPIC **and** ***MM_ANISOTROPIC*** **Modes - contd.**

- Example

  Suppose we want a logical coordinate system 10000 x 10000 to fit into the client area of a window regardless of the dimensions or aspect ratio. The following code does the trick.

  ```
  CRect client;
  GetClientRect(client);
  pDC->SetMapMode(MM_ANISOTROPIC);
  pDC->SetWindowExt(10000, 10000);
  pDC->SetViewportExt(client.right, -client.bottom);
  ```

- The negative value of the second argument to *SetViewportExt()* is required to maintain +Y in the up direction.

MM_ISOTROPIC and *MM_ANISOTROPIC* Modes

- Note the minus sign in the viewport extent. This is required because of the opposite sign of the Y-axis in these mapping modes.
- We could output a rectangle that would always fit in the window with the following:

```
pDC->Rectangle(1000, -1000, 9000, -9000);
```

- Note the negative coordinates for the Y-axis.
- We could set the origin to the bottom left cornet with:

```
pDC->SetViewportOrg(0, client.bottom);
```

and then use positive Y-axis values.

Coordinate Conversion

- It is not possible to work in logical coordinates at all times.
- Some functions require device coordinates.

  All CWnd member functions work in device coordinates.
- Conversion between the two coordinate systems is easy.
- Two CDC overloaded member functions are provided for this purpose, *LPtoDP()* and *DPtoLP()*. They each take a variety of argument types including a CPoint or CRect object.
- The argument is used for both input and output.

  ```
  CPoint mypoint(100, 200);
  CClientDC dc(this);
  dc.SetMapMode(MM_LOMETRIC);
  dc.LPtoDP(&mypoint);
  ```
- CDC member functions work in logical coordinates.

The EX04B Example

- This program is EX04A with a conversion to the ***MM_HIMETRIC*** mapping mode.
- The function *OnPrepareDC()* is overridden to allow changing the mapping mode.
- *OnLButtonDown()* must call *OnPrepareDC()* and then convert the logical coordinates to device corrdinates.
- Note - the mouse position is always reported in device coordinates.

Scrolling a View

- Scrolling is not directly supported by the *CView* class.
- The *CScrollView* class is a quick and dirty way to implement a scrolling view.

  Scrolling large amounts of data with *CScrollView* is not efficient and additional steps are needed to improve performance.
- *CScrollView* implements scrolling by changing the *viewport* relative to the actual window.
- Windows messages *WM_HSCROLL* and *WM_VSCROLL* are processed by *CScrollView.* These messages cause the viewport position to be changed.

Notes on the EX04C Example

- The *OnInitialUpdate()* function is called right after the view window is created. This is the place to set up sizes etc.
- The *SetScrollSizes* function is used to initializes the scrolling range for the *CScrollView* class. It must be called.
- Windows handles the use of the mouse for scrolling, but NOT the keyboard. Processing WM_KEYDOWN messages can be used to allow scrolling with the keyboard.

  Note - this is identical code to what the Windows SDK programmer would do. (See Petzold's book)

The Windows *GetSystemMetrics()* function can be used to determine the size of the *screen* in pixels.

Important Windows Messages

- WM_CREATE

  The first message sent to a view. Do any setup in this function that should be done before the view is displayed.

- WM_CLOSE

  Sent to a window when the user has closed it. Do any cleanup here. Also, this is a good place to make sure the user is prompted to save data etc. The base class *OnClose()* function needs to be called.

- WM_QUERYENDSESSION

  Sent to all running applications when the user exits Windows.

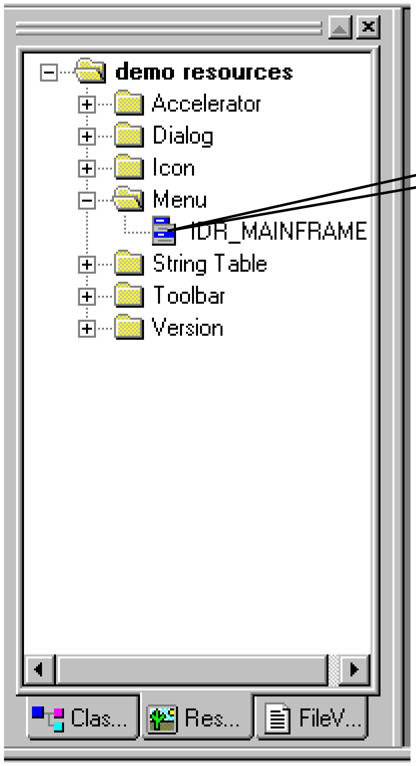
- WM_DESTROY

  Sent after WM_CLOSE. The window is no longer visible, but it is still active and child windows are still active.

- WM_NCDESTROY

  The last message sent to a window. Any child windows have already been destroyed.

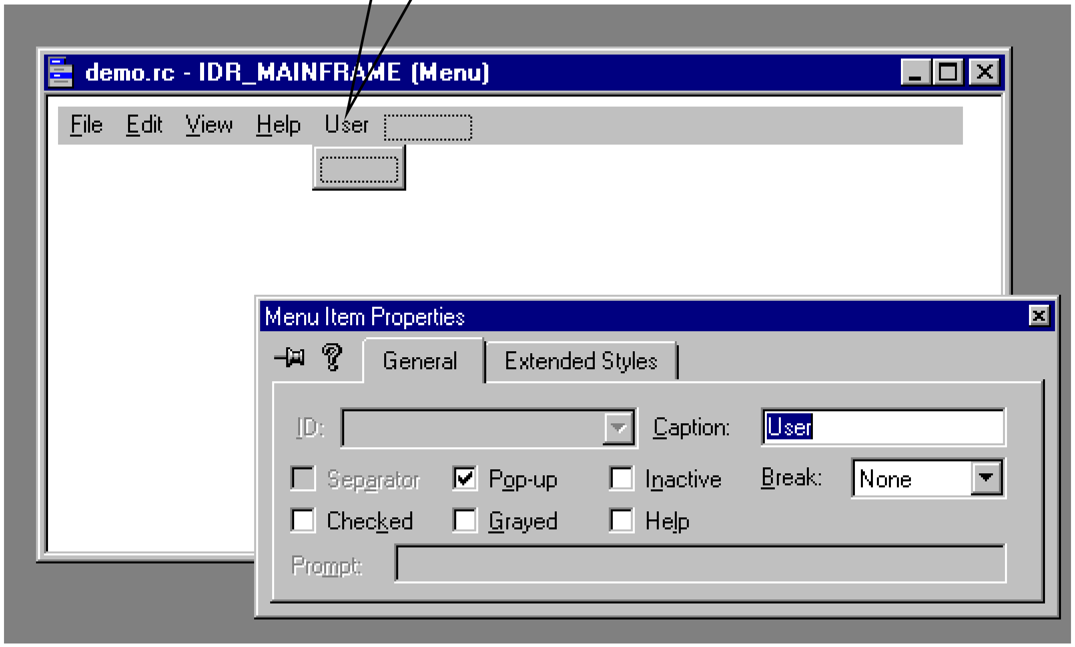
Adding a Menu Item and Handler



Double click

The Menu Editor Window

Type Name



ID:
Caption: User

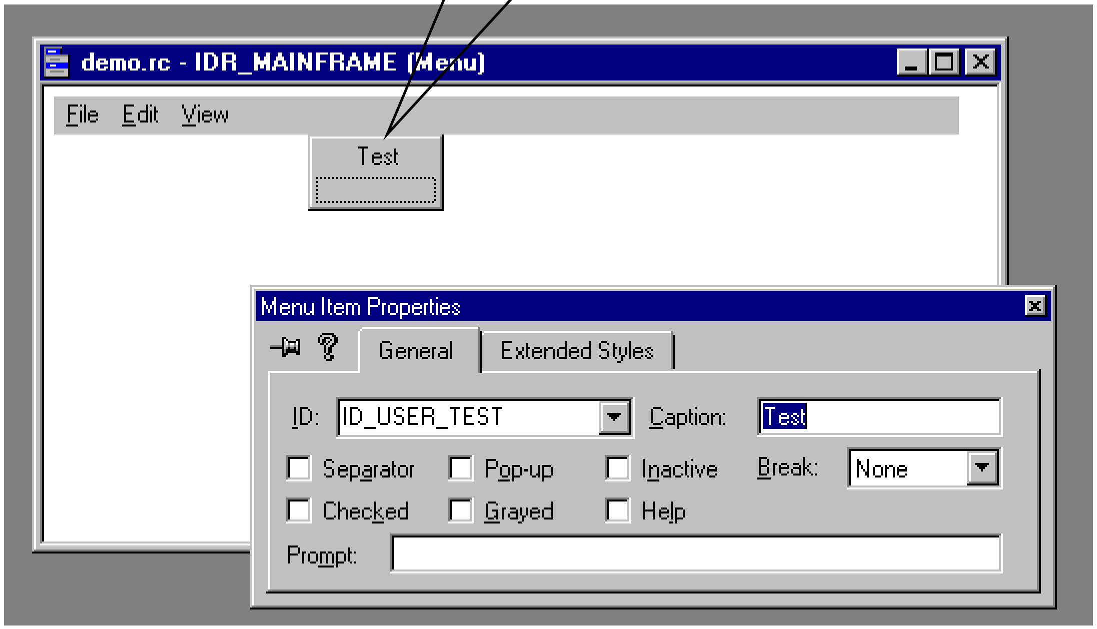
☐ Separator ☑ Pop-up ☐ Inactive Break: None

☐ Checked ☐ Grayed ☐ Help

Prompt:

Adding a Popup Item

Add Popup Item



ID: ID_USER_TEST

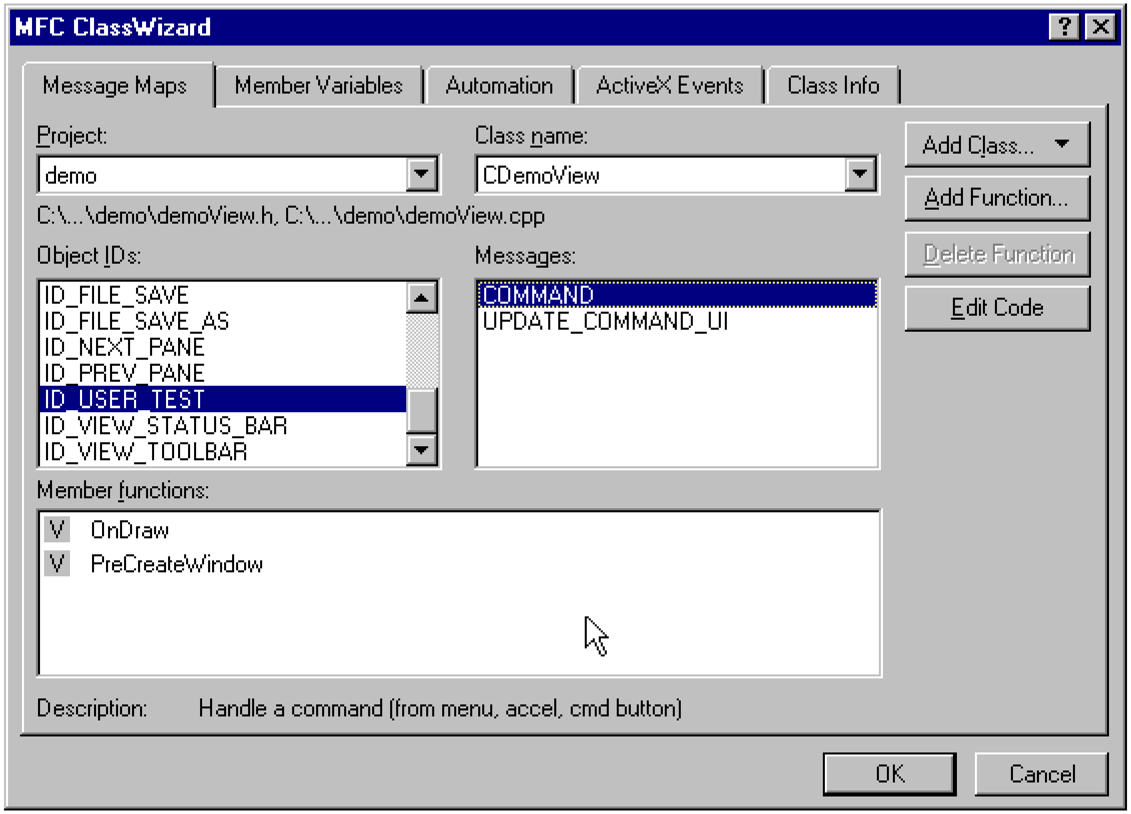
Caption: Test

□ Separator □ Pop-up □ Inactive Break: None

□ Checked □ Grayed □ Help

Prompt:

Adding the Message Handler



MFC ClassWizard

Message Maps | Member Variables | Automation | ActiveX Events | Class Info

Project: demo

Class name: CDemoView

C:\...\demo\demoView.h, C:\...\demo\demoView.cpp

Object IDs:

```
ID_FILE_SAVE
ID_FILE_SAVE_AS
ID_NEXT_PANE
ID_PREV_PANE
ID_USER_TEST
ID_VIEW_STATUS_BAR
ID_VIEW_TOOLBAR
```

Messages:

```
COMMAND
UPDATE_COMMAND_UI
```

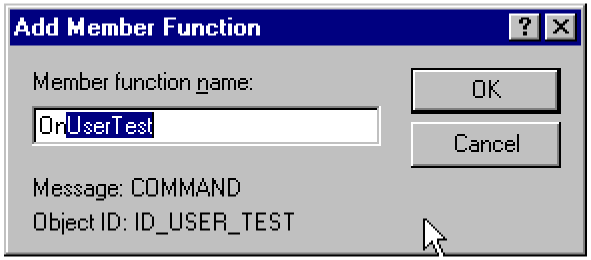
Member functions:

```
V  OnDraw
V  PreCreateWindow
```

Description: Handle a command (from menu, accel, cmd button)

Steps Involved

- Select class wizard from view menu
- Select message map tab
- Select the view class from drop down list
- Highlight the ID of the menu item from the list
- Highlight COMMAND from the Message window
- Click on Add Function to complete the operation
- Click on OK



Member function name:
OnUserTest

Message: COMMAND
Object ID: ID_USER_TEST

Resulting Code Generated

```
////////////////////////////////////////////////////////
// CDemoView message handlers

void CDemoView::OnUserTest()
{
       // TODO: Add your command handler code here

}
```

The Graphics Device Interface - GDI

- The Windows GDI has the capability to display:
  - Points
  - Lines
  - Rectangles
  - Polygons
  - Ellipses
  - Bitmaps
  - Text
- Graphics programming is really a topic unto itself and doesn’t present any significant MFC related issues.
- We will concentrate on text since there are a number of essential details that you need to know for almost any application.

The Device Context

- In order to use any GDI functions you must have a *device context* to the device (window, screen, printer, etc.)
- The underlying Windows device context is encapsulated in a number of different MFC classes all derived from the class *CDC*.
- If you want to find a GDI function, look in the list of member functions for CDC.
- *CClientDC*

  A device context to the client area of your window.
- *CWindowDC*

  A device context to the *entire* window, including the borders, caption bar etc.
- *CPaintDC*

  A special device context similar to a CClientDC, but only used with WM_PAINT message processing.
- *CMetaFileDC*

  Used to create A windows *metafile*.

Working with *CDC* Objects

- Windows only has a limited number of device contexts. They must be shared among all the running applications.
- Construct and destroy a device context all in the course of processing a single Windows message.
- This is most easily done if the DC is created on the stack. The constructor and destructor will be called automatically in that case.
- Each CDC class object has an associated Windows device context object.
- The constructor of the CDC object must be provided a pointer to the window for which the device context is desired. Normally:

```
CClientDC mydc(this);
```

- An alternative way to obtain a device context is to use the *GetDC()* function call.

```
CDC* pdc = GetDC();
//use the DC
ReleaseDC(pdc);  //Don't destroy the DC, release it
```

The State of a Device Context

- A fresh device context is set up with default state information.
- DC state information includes:
  - GDI drawing objects
    - Pens, Brushes, fonts, etc.
  - The mapping mode
  - Misc. other information
- Each time you want to use a non-default GDI object you must *select* the GDI object you require into the device context.
- Non-default GDI objects should be selected out of the DC before they are destroyed. This prevents attempts to access non-existent objects. Also, <u>a GDI object can't be destroyed (at the Windows level) if it is selected into a device context.</u>

GDI Objects

- GDI objects include instances of the following classes:

  CBitmap, CBrush, CFont

  CPalette, CPen, CRgn

- *CGdiObject* is the base class.
- When you select a new GDI object into a DC you should save the old object and restore it later.

```
CPen newPen(PS_DASHDOTDOT, 2, (COLORREF) 0);
CPen* pOldPen = pDC->SelectObject(&newPen);

//Use the device context

pDC->SelectObject(pOldPen);
```

- Note - The new pen is destroyed automatically when the function returns by the C++ destructor.
- When a device context is destroyed, its GDI objects are deselected. <u>Don’t rely on this as an excuse not to deselect the objects.</u>

Stock GDI Objects

- Windows provides a number of *stock* GDI objects.
- The *SelectStockObject()* function is used to access them.

```
pDC->SelectStockObject(GRAY_PEN);
```

- The SDK function is *GetStockObject()* which has to be selected into the DC. This function is still available through MFC, but has been implemented as *CreateStockObject()* because a C++ class object needs to be created to encapsulate the Windows object.
- Stock GDI objects can be selected back into a DC to avoid the necessity of saving the old GDI object's pointer.

Some Device Context Caveats

- Normally the lifetime of a device context is the time required to process a single Windows message.
- Longer life DCs can be created by establishing a *private* DC in the Window class structure. This the NOT a C++ class, but rather the Windows *class*. (See a book like Petzold or the SDK documentation if you want to get involved with this.)
- There is a problem with pointers obtained with the *SelectObject()* function call. MFC dynamically creates a C++ object which is apt to be destroyed shortly after the function return from the message handler. Therefore you can't save it and use it later.
- The workaround for this is the use of the Windows *handle* which can be obtained with the *GetSafeHandle()* member function. (See the book.)
- Keep this in mind, but don't worry if everything you do with a DC is within a single message handler. Then nothing will go wrong.

Colors

- Windows provides hardware independent color mapping.
- A Windows color is a 24-bit color value with 8-bit red, green, and blue primary values.
- These colors are mapped to the particular video display mode in use.
- Colors not capable of being displayed as a pure color are displayed by the use of *dithering*.
- The common VGA modes are:
  1. 16 color mode
  2. 256 color mode
  3. 24-bit color - 16 million colors
- Windows uses a 32-bit representation of the 24-bit color. The data type is *COLORREF*.
- The *RGB* macro can be used to create a COLORREF.

```
CBrush redbrush(RGB(255, 0, 0));
```

- The text background color could be set to blue using:

```
pDC->SetBkColor(RGB(0, 0, 255));
```

16 Color VGA Mode

Red	Green	Blue	Color
0	0	0	Black
0	0	255	Bright blue
0	255	0	Bright green
0	255	255	Cyan
255	0	0	Bright red
255	0	255	Magenta
255	255	0	Bright yellow
255	255	255	White
0	0	128	Dark blue
0	128	0	Dark green
0	128	128	Blue-green
128	0	0	Brown
128	0	128	Dark purple
128	128	0	Olive
128	128	128	Dark gray
192	192	192	Light gray

256 Color VGA Mode

- Even though 256 colors are possible, unless a palette is used, only 20 pure colors are available.
- Four additional colors are added.

Red	Green	Blue	Color
192	220	192	Money green
166	202	240	Sky blue
255	251	240	Cream
160	160	164	Medium gray

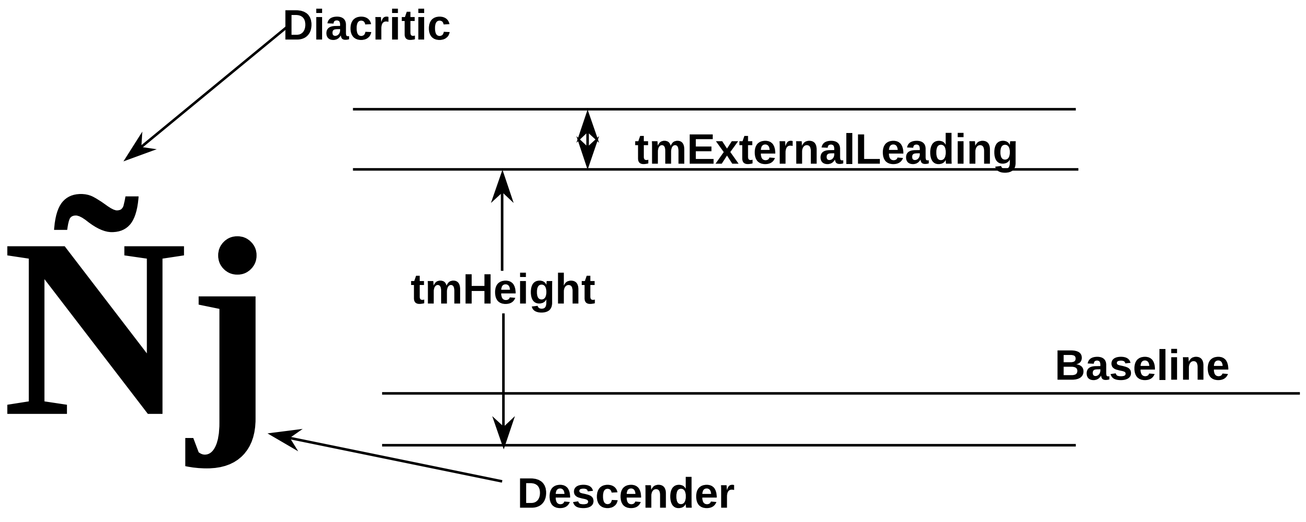
Fonts

- To fully examine the issue of fonts and WYSIWYG would require a significant amount of time.
- For most Windows applications (other than word processors) the use of different fonts is not too difficult.
- Fonts are GDI objects and are selected into your device context.
- The introduction of TrueType fonts in Windows 3.1 was a major improvement to the ease of Windows programming.
- A *font* is a collection of characters from a *character set* in a particular *typeface*.
- Character heights are measured in *points*. A point is approximately 1/72 inch.
- The *MM_TWIPS* mapping mode is based on a unit which is 1/20 of a point or 1/1440 inch.

  Example: a 12 point character would be 240 logical units in MM_TWIPS mapping mode.

Computing Character Height

- **Once a font is selected into the DC we can get the character height in logical units using the *GetTextMetrics()* function.**
- **A special structure, *TEXTMETRICS*, is filled in by this call.**
- **The height of a character is *tmHeight* plus *tmExternalLeading*.**



```
TEXTMETRIC tm;
pDC->GetTextMetrics(&tm);
int cychar = tm.tmHeight + tm.tmExternalLeading;
```

Computing Character Height - contd.

- The function *OnPrepareDC()* is called automatically prior to the *OnDraw()* function. OnPrepareDC can be used to set up the mapping mode prior to drawing in the window (e.g. MM_TWIPS).
- *OnPrepareDC()* **must** be called explicitly for other message handling functions in which you wish to draw using a mapping mode other than MM_TEXT.
- Under Windows 3.1 there was an issue of the metrics used to measure the physical screen vs. the logical pixels per inch. This meant that text using MM_TWIPS was slightly smaller than desired. This "bug" has been fixed in WIndows 95 and you can forget about this issue.

Printing vs. Display of Fonts

- *What you see is what you get* is not really possible when dealing with a 300 DPI printer and a 96 DPI display.
- The font engine needs to map each and every character to a box based on the number of pixels (dots) for the device.
- This results in a slightly different size for the characters in a line on the screen and on the printer.
- The issue is more pronounced when working with fonts other than True Type where the font used on the display may be different from that used on the printer. (EX05B)
- *Print Preview* techniques (discussed later) are used to give the user a display that matches the printer as far as the exact spacing of characters is concerned. However, character quality is sacrificed.

Notes on the EX05C Example

- Three mouse messages are used:

  WM_LBUTTONDOWN

  Initiates the dragging process.

  WM_MOUSEMOVE

  Tracks the mouse position and causes the circle to follow.

  WM_LBUTTONUP

  Terminates the tracking operation.

- Capturing the mouse

  When tracking the mouse using WM_MOUSEMOVE messages it is important to prevent the mouse messages from going to other applications when the mouse is moved *outside* the application’s window. Normally Windows sends mouse messages to the application over which the mouse is positioned.

  SetCapture() and *ReleaseCapture*() are used for this purpose.

- The *SetCursor()* and *LoadCursor()* Windows API functions are used to change the cursor to a *cross*.

Notes on the EX05C Example - contd.

- *OnPrepareDC()* is used to configure the mapping mode, origin, and scale factor.
- *SetScrollSizes()* is used to establish the values to be used within the *OnPrepareDC()* function.
- Mouse coordinates are in *device* coordinates and need to be converted with *DPtoLP()*.
- The *SetBrushOrg()* function is required to ensure that the pattern used to fill the ellipse object lines up during scrolling.
- *CScrollView* has an interesting mode called *SetScaleToFitSize* mode. In this mode the entire scrollable area is always displayed. As you might guess, MM_ANISOTROPIC mapping mode comes into play here. A call to *SetScaleToFitSize()* is used to set it up.

Dialogs

Windows supports two standard dialog types and and a third obsolete type

- Modal

  The most common type of dialog. The user must enter data in order to continue with the application. However, the user can switch to another application and come back to the application displaying the dialog.

- Modeless

  A dialog that can be activated and stays active while the user does other work within the application. The user must go back to the dialog and close it to remove it from the screen.

- System Modal (Windows 3.1, now obsolete)

  Is like a modal dialog except the user was not allowed to switch to another application (Windows 3.1). Should no longer be used.

  Note - under 32-bit windows the dialog is stuck on top of all other applications, but the user can still switch to these other applications.

The *CDialog* class

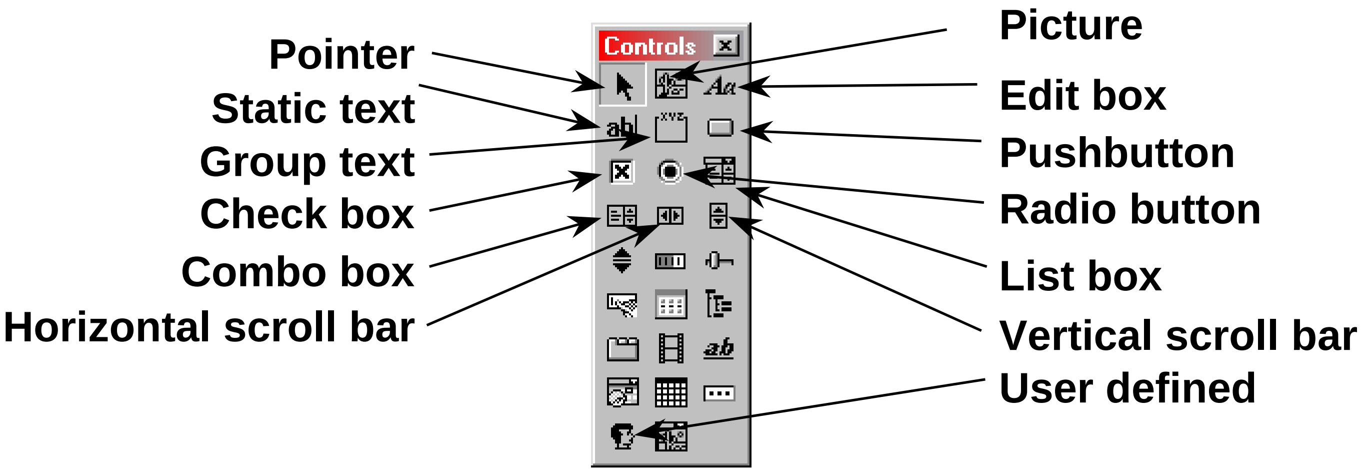
- The *CDialog* class encapsulates a Windows dialog object.
- You use the Developer Studio to create a dialog resource. When the dialog is activated the resource information is used to create the on screen dialog.
- A dialog box is a window that contains *child window controls*. MFC makes the job of interacting with the child window controls much easier than with the native API.
- What is going on behind the scenes is that the child window controls are sending messages to the dialog window. SDK programmers had to provide message handlers for these controls.
- MFC provides a mechanism for *Dialog Data eXchange* (DDX). Data between a class data member and the dialog control is automated. The Class Wizard makes the programming job almost trivial in most cases.

The Steps Involved

- Use Developer Studio to create the resource. Identifiers are automatically set up in the header file resource.h.
- Class Wizard is then used to create a class derived from *CDialog*. This class is associated with the resources just created.
- Data members and their associations with the dialog controls are added to the derived class.
- Message handlers for any buttons in the dialog are programmed.
- Some controls (scroll bars, list boxes) require some special handling. This code needs to be added.
- The dialog box is activated in response to a user action such as clicking on a menu item. A message handler must be added to instantiate and activate the dialog box.

Developer Studio Control Palette

- The Developer Studio’s resource editor is an easy to use and intuitive application.
- The layout is “*what you see is what you get*”.



Pointer

Static text

Group text

Check box

Combo box

Horizontal scroll bar

Picture

Edit box

Pushbutton

Radio button

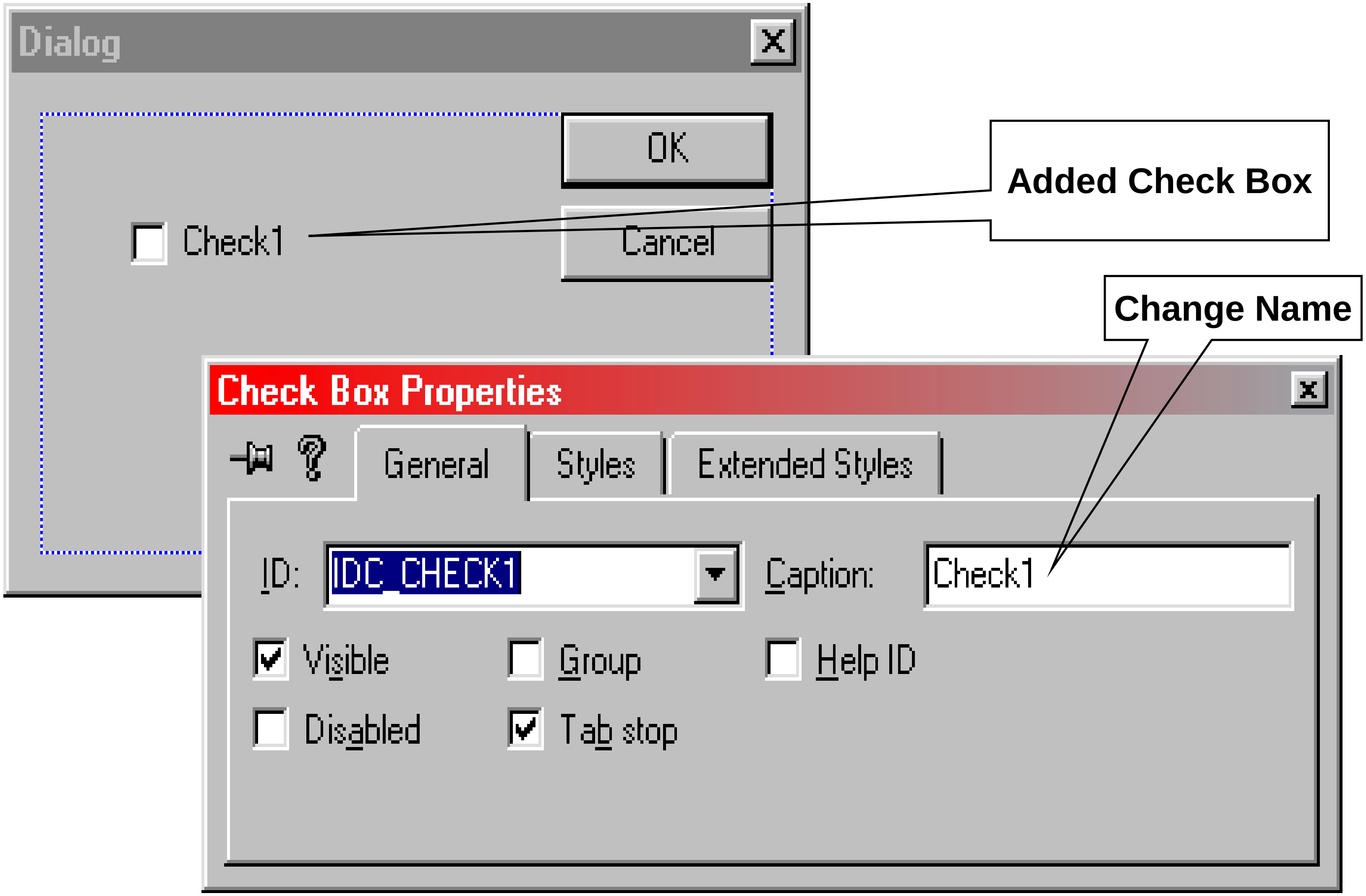
List box

Vertical scroll bar

User defined

The remainder are more advanced controls that were added to Windows 95/98

Control Properties



□ Check1

ID: IDC_CHECK1

Caption: Check1

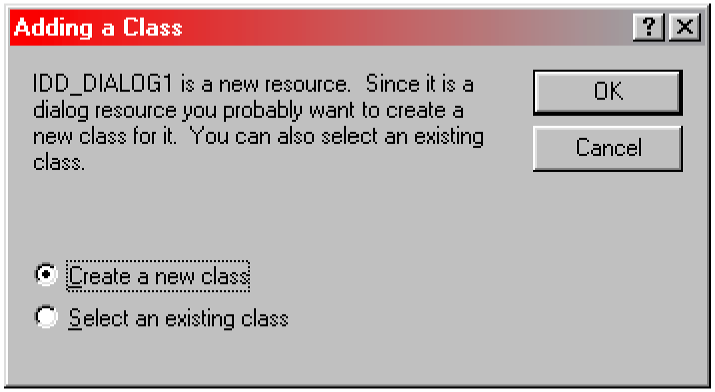
☑ Visible
□ Group
□ Help ID
□ Disabled
☑ Tab stop

Added Check Box

Change Name

Creating the Dialog Class with the Class Wizard

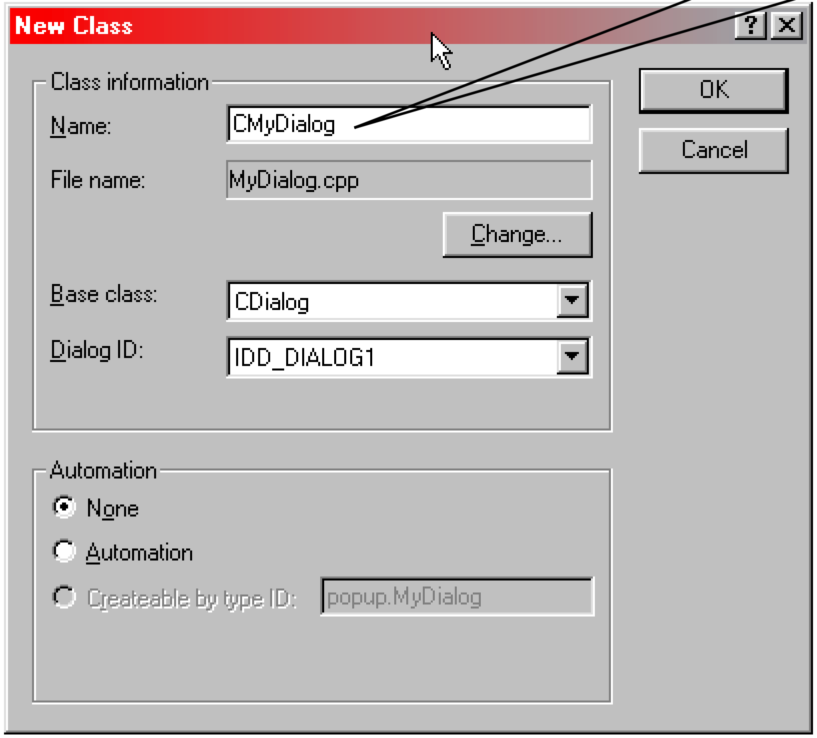
- Change the name of the dialog itself using the properties window or use the default, IDD_DIALOG1 etc.



IDD_DIALOG1 is a new resource. Since it is a dialog resource you probably want to create a new class for it. You can also select an existing class.

● Create a new class
○ Select an existing class

Choosing the Dialog's Class Name



Choose a Name

New Class

Class information

Name: CMyDialog

File name: MyDialog.cpp

Base class: CDialog

Dialog ID: IDD_DIALOG1

Automation

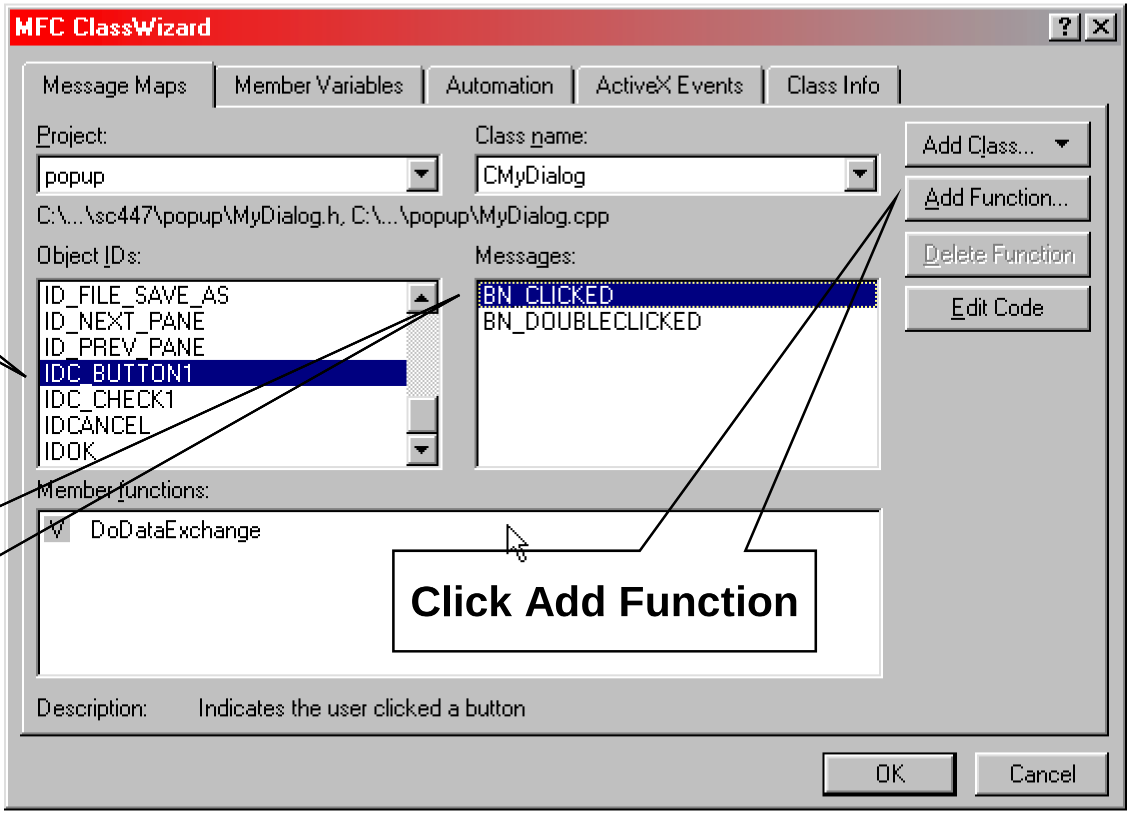
● None

○ Automation

○ Createable by type ID: popup.MyDialog

Adding a Handler Function

Choose Correct ID



MFC ClassWizard

Project: popup

Class name: CMyDialog

C:\...\sc447\popup\MyDialog.h, C:\...\popup\MyDialog.cpp

Object IDs:

- ID_FILE_SAVE_AS
- ID_NEXT_PANE
- ID_PREV_PANE
- IDC_BUTTON1
- IDC_CHECK1
- IDCANCEL
- IDOK

Messages:

- BN_CLICKED
- BN_DOUBLECLICKED

Member functions:

V DoDataExchange

Description: Indicates the user clicked a button

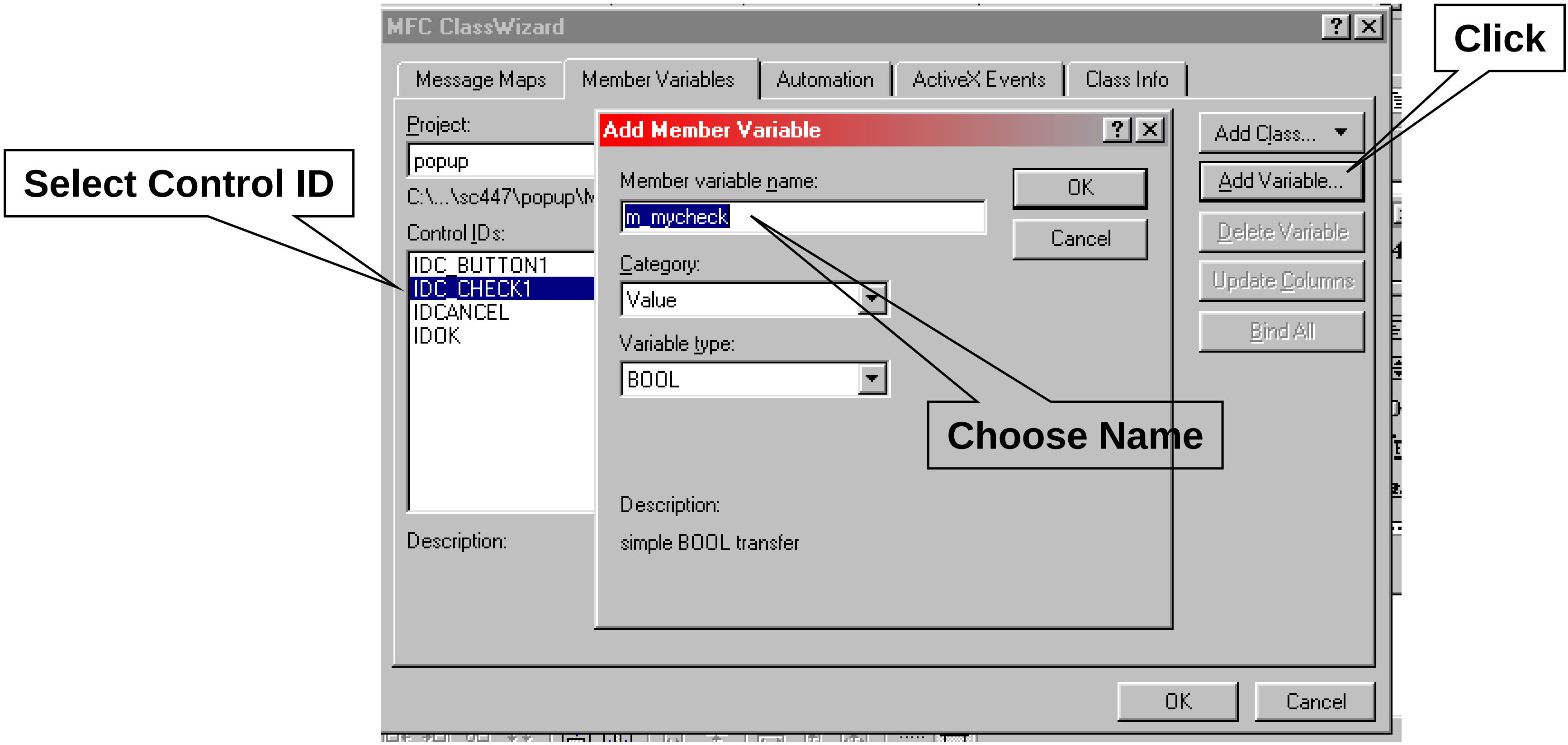
Click Add Function

Select BN_CLICKED

Visual Studio 6 Shortcut

- After you lay out the dialog, double clicking on any control will ask if you want to create a class using the Class Wizard.
- Subsequent double clicks on a control will ask if you want to add a handler.
- Be careful, don't add handlers for controls such as check boxes.
- After a handler has been created, double clicks take you directly to the code.

Adding a DDX Member Variable



Select Control ID

Click

Choose Name

Project: popup

C:\...\sc447\popup\

Control IDs:

- IDC_BUTTON1
- IDC_CHECK1
- IDCANCEL
- IDOK

Description:

Member variable name: m_mycheck

Category: Value

Variable type: BOOL

Description:

simple BOOL transfer

Activating the Dialog Box

- You provide a message handler in your derived view class to activate the dialog. This is usually tied to a menu item, but a mouse click can also be used.
- Normally the dialog class is instantiated on the stack.

  ```
  CMyDialog dlg;
  ```

  CMyDialog is the derived class.
- The next step is to initialize the data members of the dialog class. Where the data comes from is up to you. Normally the information is state from other parts of the application. DDX will take care of actually placing the data in the child window controls.
- Finally, the dialog is activated with a call to *DoModal()*.

  ```
  int ret = dlg.DoModal();
  ```
- The return code will be *IDOK* or *IDCANCEL* depending on the users action. Upon return the DDX mechanism will have copied back the updated data to the class data members.

The Tab and Group Attributes

- Two important control attributes require special attention.
- The *tab* attribute must be set in order to allow tabbing to the control. However, the control can still be selected with the mouse if the tab attribute is not set.
- All active controls (not static text) should have the tab attribute set.
- The *tab order* (on layout menu) is used to order the control descriptions in the resource script. This ordering determines the actual physical order in which the controls are selected when the user tabs.
- The *group* attribute is in a sense a <u>super tab</u>. It is placed on the first control in a group and controls the action of the arrow keys. Individual controls can have the group attribute set since they are essentially a group of one element. The concept of the group is critical to the action of radio buttons. The group attribute is set only on the *first* button in the group <u>and</u> on the *first* control *following* the radio button group.

Static Text

- *Static Text* controls can’t be selected.
- The text in the static control can be changed with the *SetDlgItemText()* function call.

```
p->SetDlgItemText(IDC_STATIC1, “New text”);
```

- If you make the static text control large enough, you can use this technique to output a paragraph of text at run time. Use the '\n' character to end a line It is handled properly.
- An ampersand character (&) can be used in a static text control to allow the user to go to the control following the text in the tab order. The & is placed before a unique character to be used with the ALT key. Pressing ALT and this character activates the control.

Advanced Dialog Features

- The examples in the text show you how to how to handle certain advanced dialog features.
  - Setting up list box data in the *OnInitDialog()* function.
  - Preventing the dialog from being exited when *enter* is hit.
  - Using scroll bar controls.
  - Changing the dialog and control colors.
  - Painting in the dialog.
- Even more features are available. Study the individual child window control classes.
- Note - the Class Wizard does not handle all the features and you will have to add the code manually.

OnInitDialog()

- The *OnInitDialog()* function is called just prior to the dialog being displayed.
- To provide an override for this virtual function you can use the Class Wizard. Set up a handler for the WM_INITDIALOG message
- The Class Wizard is smart enough to recognize that *OnInitDialog()* is a virtual function and generates a proper skeleton. Nothing is actually added to the message map.
- Use this function to set up dialog controls that require initializations that dialog data exchange doesn't provide.
  - E.g., a list box, scroll bar, etc.

Windows Common Controls

- These controls are new and are located in *COMCTL32.DLL*.
- Unfortunately the *ClassWizard* offers no member variable support and you must use the *OnInitDialog()* and *OnOK()* member functions to handle data exchange to and from the control.
- Among these new controls are:

  The progress Indicator Control

  A moving bar graph.

  The trackbar Control

  A slider that functions much like a scroll bar.

  The Spinner Control

  A tiny scroller with associated "buddy" edit control.

  The List Control

  A list control that allows images as well as text.

  The Tree Control

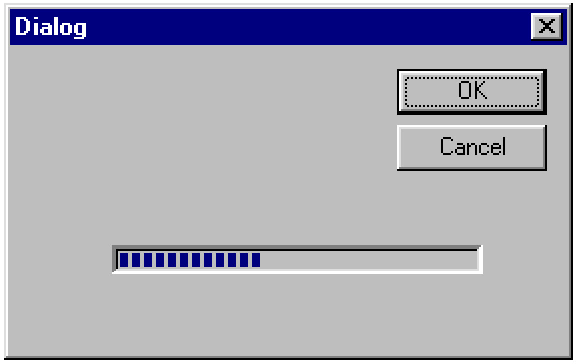
  The familiar tree structure found in the workspace window.

- The programming examples in the book are straightforward and can be easily used as models for your own code.

Progress Control Example

- The default range on the progress control is 0 to 100. The default step size is 10.
- Here is an example using the right mouse button to advance the progress control:

```
void CMyDialog::OnRButtonDown(UINT nFlags, CPoint point)
{
   CProgressCtrl *p = (CProgressCtrl *)
                                    GetDlgItem(IDC_PROGRESS1);
   p->StepIt();
}
```

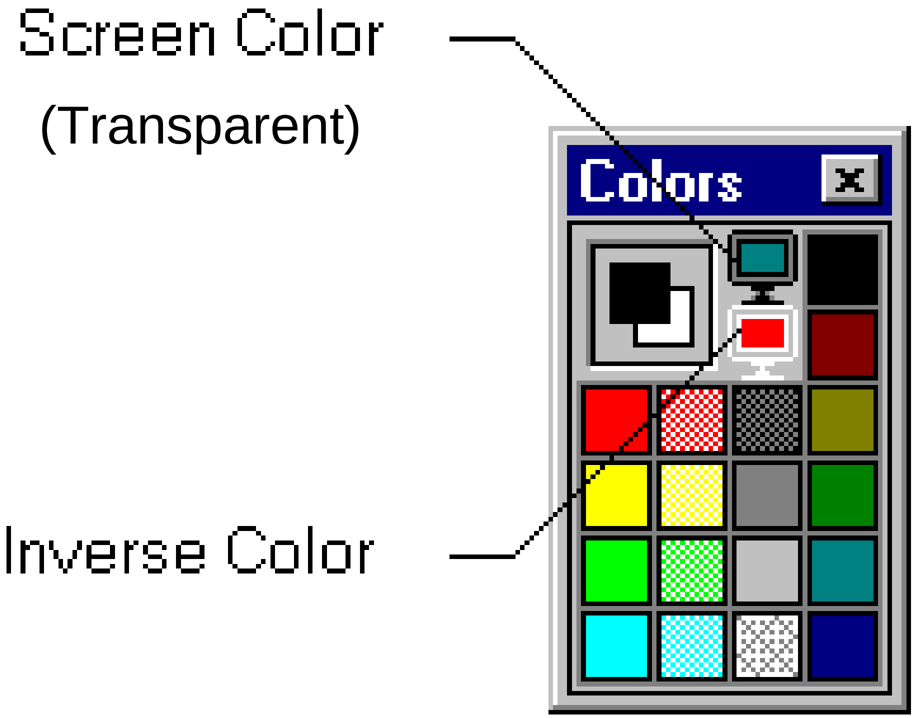


Notes about Icons

- Windows 3.1 supports icons that are 32 x 32 pixels.
- Windows 95 requires two different icon sizes:
  1. 32 x 32 for normal size icons.
  2. 16 x 16 for the taskbar and Explorer file listings displaying small icons.
- The icon editor allows selection of both sizes.
- If you do not provide a small icon then Windows reduces the size of the 32 x 32 icon with perhaps less than satisfactory results.

Transparent and Inverted Pixels

- In addition to solid (opaque) colors you can draw transparent and inverted pixels.
- Note - Windows95 seems to handle inverted pixels in a somewhat inconsistent manner compared to Windows 3.1.



Screen Color

(Transparent)

Inverse Color

Modeless Dialogs

- Both *modal* and *modeless* dialogs use the *CDialog* class.
- The *modeless* dialog allows the user to leave the dialog and work in other application windows. The dialog remains on the screen.
- *DoModal()* is not used to activate modeless dialogs.
- An object of the *CDialog* class is created on the stack or heap and its *Create()* function is called. This acts like *DoModal()*, but the function returns immediately.
- Care must be taken to make sure that the *CDialog* object stays in scope while the dialog is on the screen. One way to ensure this is to declare a pointer to a *CDialog* derived object as a data member of your *View* class. The view's constructor instantiates the object from the heap and its destructor deletes it.

Opening and Closing the Dialog

- Opening the dialog is normally straightforward. *Create()* is called in response to a menu item or even a mouse event (as in the example). Normally the view handles this.
- Closing the modeless dialog is somewhat more difficult. Clicking on *OK* invokes a message handler in the *dialog*. Unlike with *DoModal()*, the default *OnOK()* and *OnCancel()* handlers <u>must not be used</u>.
- Two approaches:
  1. Call a member function in the *View* class.
  2. Send the *View* class a message.
- The second choice is more versatile and does not *marry* the dialog to the view class.

  Note - the implementation in the book does in effect marry the dialog to the view since the C++ object is created and destroyed by the view, This could also be done in the main application class. This should be considered for MDI applications in which views are created and destroyed during the life of the program. However, there are other alternatives we will explore when the Document View Architecture and MDI is covered.

Sending Windows Messages

- This is a well known *trick* of the SDK programmer.
- Two methods:
  1. Send a message.
  2. Post a message.
- Sending a message causes the message to be delivered immediately.
- Posting a message adds the message to the Windows message queue and there may be a small delay if other messages are ahead in the queue.
- You can define your own Windows messages. Message *ID's* that do not conflict with other Windows messages can be used. *WM_USER* is the first *ID* available for user messages. Skip the first five as the MFC framework uses some.
- Windows does not interpret these messages. You can send whatever information you want in the *lparam* and *wparam* of the message.

Sending Windows Messages - contd.

- Example:

```
#define WM_GOODBYE WM_USER+5

m_pView->PostMessage(WM_GOODBYE, IDOK);
```

  and

```
m_pView->PostMessage(WM_GOODBYE, IDCANCEL);
```

- The dialogs *DestroyWindow()* function is called by these handlers to close the dialog window.

Other Considerations - Modeless Dialogs

- When the user tries to activate the dialog, you should check if it is already on the screen. This can be done by seeing of the Windows handle is not NULL. Use the *GetSafeHwnd()* call for this purpose.
- Data must still be moved from the dialog controls to the data members. Use the *UpdateData()* function as needed to move the data back and forth. This is no longer automatic when the dialog is closed by clicking on *OK*. Call *UpdateData()* before sending the WM_GOODBYE message.
- Even though the Windows dialog object is no longer in existence, the C++ dialog object will, and the data members can subsequently be read.

The Windows 95 Common Dialogs

- Windows 95 provides a number of very functional *built-in* dialogs. These are the *Common Dialog Boxes*. They are encapsulated by MFC for easy use.

Class	Purpose
CFileDialog	Open and close files
CFontDialog	Select a font
CColorDialog	Select or create a color
CPrintDialog	Set up print options
CFindReplaceDialog	Text search and replace
CPageSetupDialog	Page measurement parameters

The Windows 95 Common Dialogs - contd.

- In most cases you will not have to derive your own classes from these dialogs. You can just instantiate one of them and invoke it directly using the *DoModal()* function member.

```
CFileDialog fd(TRUE, “txt”, “*.txt”);
if (fd.DoModal() == IDOK)
{
    CFile f;
    f.Open(fd.GetPathName(), CFile::modeRead);
    //use file
    f.Close();
}
```

Deriving from the COMMDLG Classes

- There is no reason that you can’t derive from the standard MFC COMMDLG classes. Any tricks that the SDK programmers use can be applied here.
- The text has an example of adding a *Delete* button to the *CFileDialog* class.
- The *OnInitDialog()* function does most of the work.
- A *CButton* object is created as a child window to the dialog window just as are the built-in controls.
- Communication with this control must be handled explicitly since there is no pre-existing code to handle this.
- Of course the actual delete operation needs to be programmed when the dialog returns. The outcome of the dialog’s *Delete* button operation can be left behind as a member variable in the derived dialog class (e.g., *m_bDeleteFlag*).

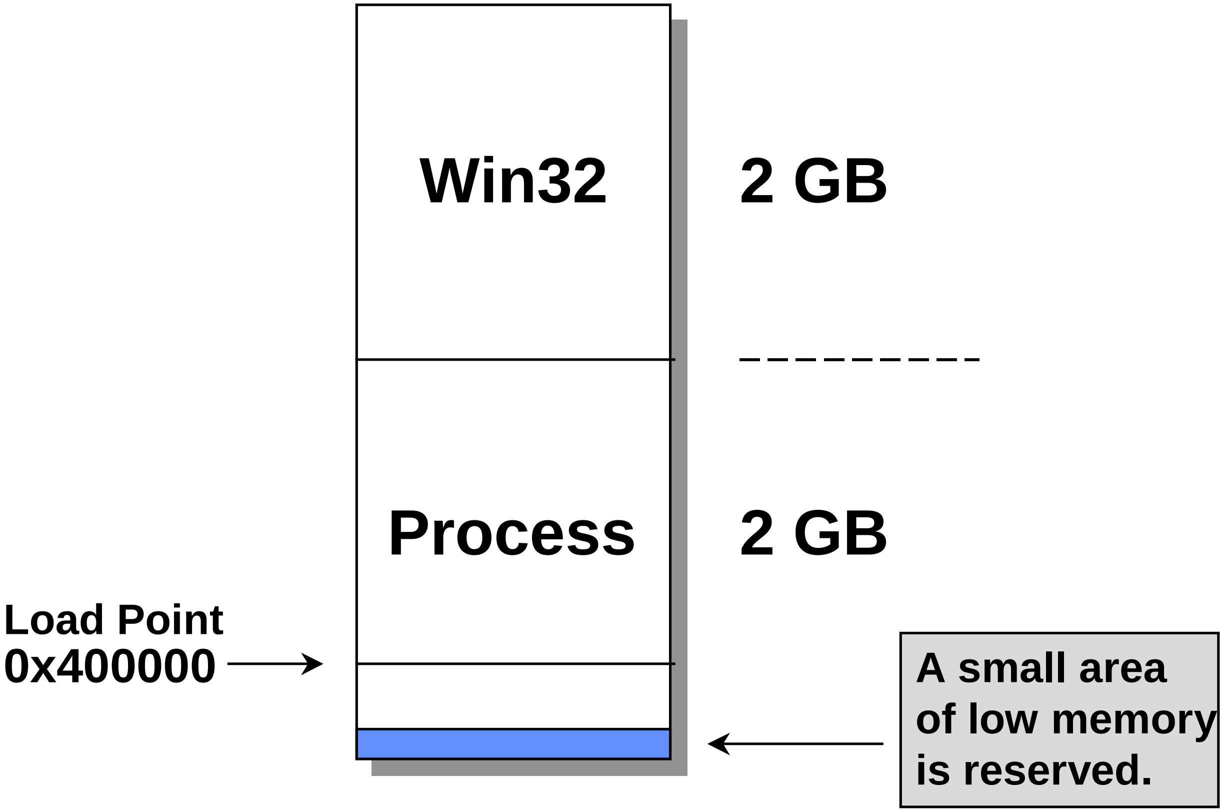
Win32 Memory Management

- With 32-bit addressing, all the problems of 16-bit memory management go away.
- Memory is organized as a linear addressing space of 4 GB. The segmentation hardware is all still there, but it is usually of no importance to the MFC programmer.
- Win32 manages the 4 GB addressing space using *paging*.
- Each memory page is 4 K and a two level page table organization is used.

  It is beyond this course to discuss paged memory management. See a suitable operating systems text such as *Tanenbaum*.
- Each *process* (application) gets its own 4 GB virtual addressing space. One process can't access the addressing space of any other process.

Win32 Memory Organization

- The 4 GB process address space is divided into two halves. The upper 2 GB is where Windows itself lives. This space is shared. The other 2 GB is private to the process.



A small area of low memory is reserved.

The C Runtime Heap

- Unlike the 16-bit world, the C++ runtime heap can grow to very large sizes.
- The underlying functions are *malloc()* and *free()*.
- They use the *VirtualAlloc()* function to allocate blocks of virtual memory directly.
- For the C++ programmer using MFC you would normally manage your memory by using the *new* and *delete* operators which in turn call *malloc()* and *free()*.
- To get started just pretty much ignore memory issues and use the *new* and *delete* operators. When you encounter a special need, study the material in the chapter and the online documentation from the Win32 SDK.

Bitmaps

- Two types of *bitmaps* are used in Windows programming:
  - GDI Bitmaps
  - Device Independent Bitmaps (DIBs)
- A bitmap contains a representation of an image consisting of some number of pixels in a specific number of colors (e.g., monochrome or 16 color).
- GDI Bitmaps are bitmaps in the specific device format for the video board and mode in use.
  - The details of these formats is rarely a concern of the programmer.
- DIBs are bitmaps that are not tied to a specific device. They are normally files with the extension *.BMP*. They can be displayed in any video mode and even printed.

Displaying a Bitmap

- A bitmap is a GDI object, just like pens and brushes.
- A bitmap must be loaded into a CBitmap class object.

  There is a Windows object encapsulated by the C++ object as is usually the case.
- The bitmap object is then selected into a *device context* just like fonts, brushes etc. <u>However, a bitmap can not be selected directly into a display device context.</u>
- A *memory device context* is created to hold the bitmap. The memory device context is made *compatible* with the display device context and then the bitmap is selected.
- The final step is to use a *bit block transfer* or *BitBlt* operation to move the pixels from the memory device context to a screen device context (usually the client DC).

Basic Steps to Display a Bitmap

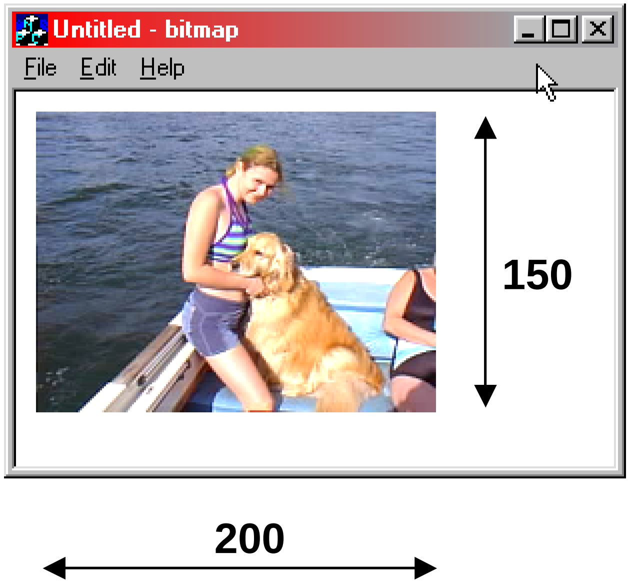
- The file *demo.bmp* was created and has the resource id IDB_DEMO. Here is how we would display it using the *OnDraw()* function:

```
void CBitmapView::OnDraw(CDC* pDC)
{
    CBitmap bmap;
    CDC mdc;
    mdc.CreateCompatibleDC(pDC);
    bmap.LoadBitmap(IDB_DEMO);
    mdc.SelectObject(&bmap);
    pDC->BitBlt(10,10, 200, 150, &mdc, 0, 0, SRCCOPY);
}
```

- The bitmap is 200 x 150 pixels and is displayed with the upper left hand corner at (10, 10).
- *SRCCOPY* is a raster operation (ROP) that merely copies the source pixels to the destination without change, and overwriting the pixels on the screen.

The Demo Bitmap

- The display would look like this:

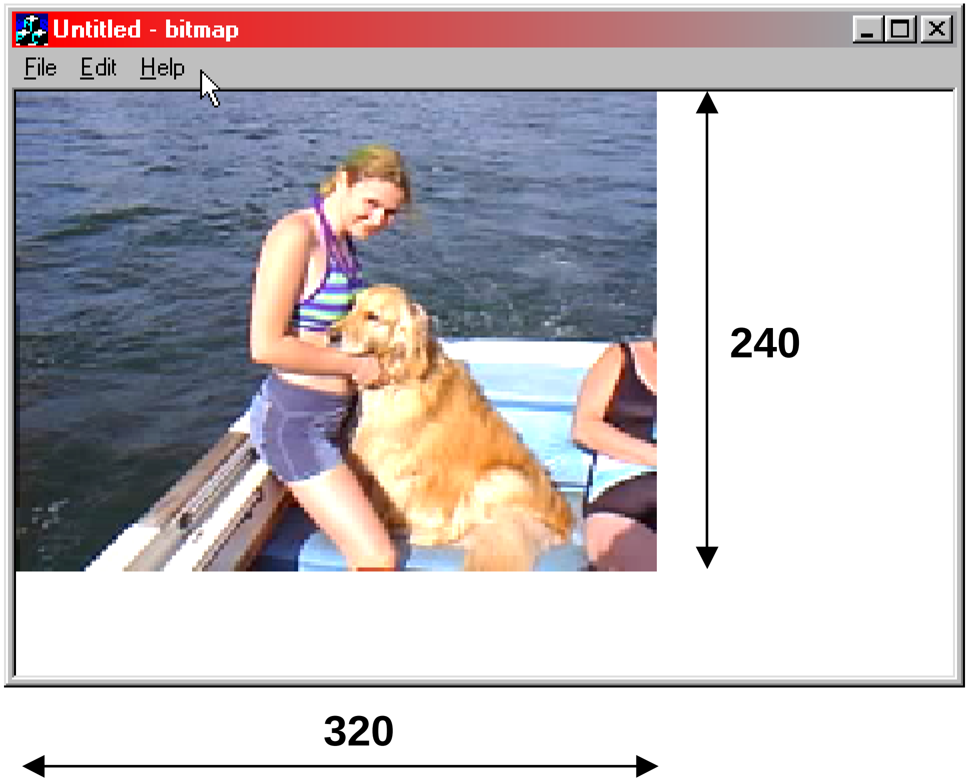


Mapping Modes and Bitmap Display

- Mapping modes are applied to the device context in the usual manner.
- It is important to remember to convert the bitmap’s size in pixels into the appropriate dimensions for mapping modes other than MM_TEXT.
- The following program demonstrates the use of of the MM_LOENGLISH mapping mode and the use of the *StretchBlt()* function call to enlarge the image to 320 x 240.

```
void CBitmapView::OnDraw(CDC* pDC)
{
    CBitmap bmap;
    CDC mdc;
    mdc.CreateCompatibleDC(pDC);
    bmap.LoadBitmap(IDB_DEMO);
    mdc.SelectObject(&bmap);
    pDC->SetMapMode(MM_LOENGLISH);
    CRect rect(0, 0, 320, 240);
    pDC->DPtoLP(&rect);
    pDC->StretchBlt(0, 0, rect.Width(),
        rect.Height(), &mdc, 0, 0, 200, 150, SRCCOPY);
}
```

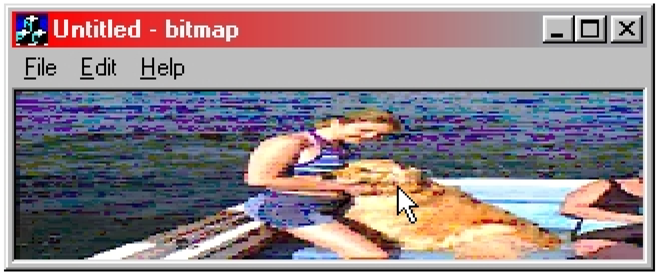
- The stretched bitmap at coordinates 0,0.



Stretching to Fit

- *StretchBlt()* can be used to fit an image to the client area. This will change the aspect ratio of the image if necessary.

```
void CBitmapView::OnDraw(CDC* pDC)
{
    CBitmap bmap;
    CDC mdc;
    mdc.CreateCompatibleDC(pDC);
    bmap.LoadBitmap(IDB_DEMO);
    mdc.SelectObject(&bmap);
    CRect rect;
    GetClientRect(&rect);
    pDC->StretchBlt(0, 0, rect.Width(),rect.Height(),
     &mdc,0,0, 200,150, SRCCOPY);
}
```



Determining the Size of a Bitmap at Runtime

- The size of a bitmap can be determined at runtime but the use of the *GetObject()* function call.
- For a bitmap we fill a *BITMAP* structure:

```
BITMAP bm;
pBitmap->GetObject(sizeof(bm), &bm);
width = bm.bmWidth;
height = bm.bmHeight;
```

- *pBitmap* is a pointer to a *CBitmap* object which has the bitmap already loaded from the resource file.

Using Bitmaps to Speedup Screen Display

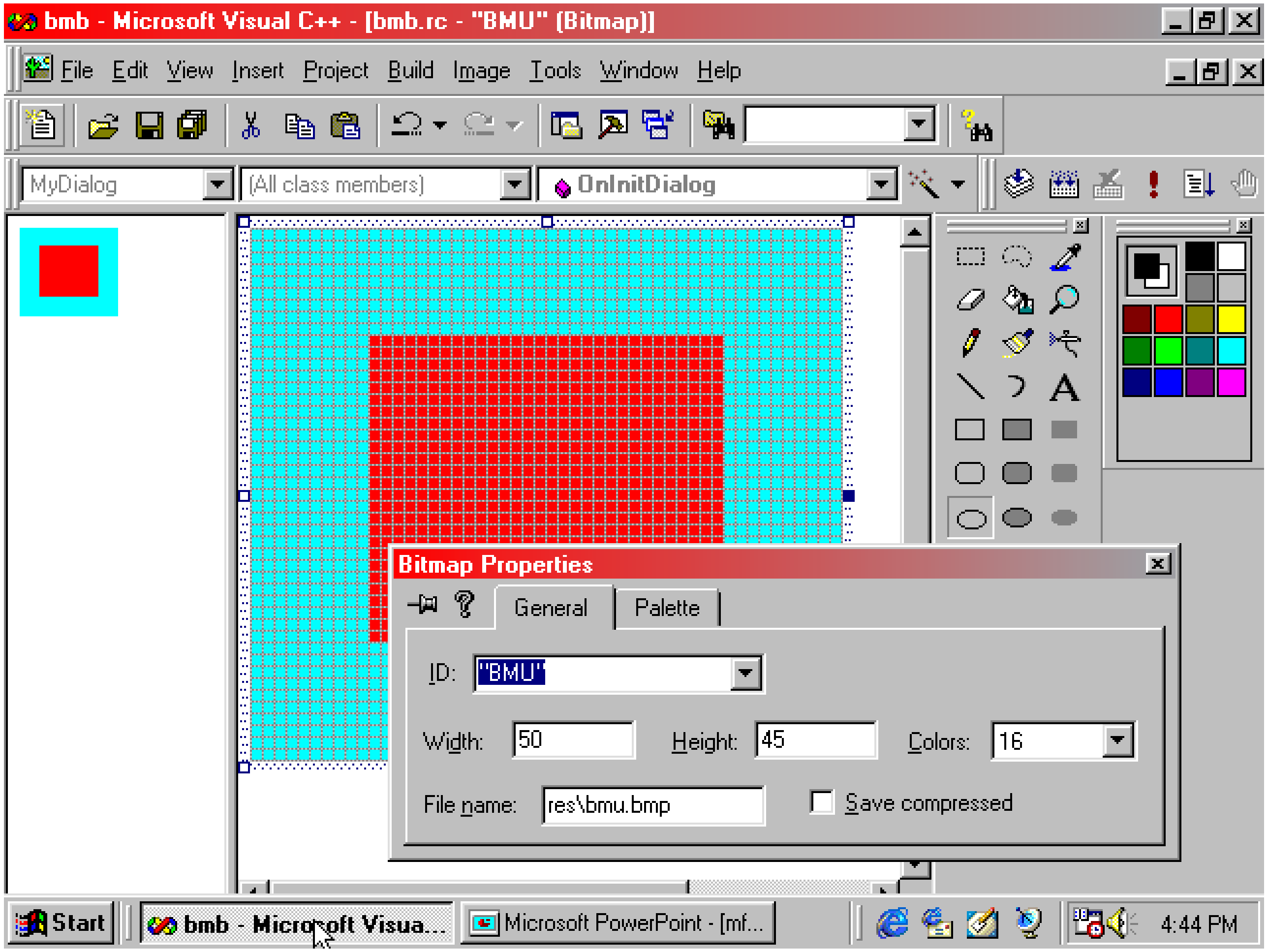
- The example in which we moved a filled ellipse with the mouse showed noticeable flicker.
- This was due to the fact that Windows was erasing the background and repainting the ellipse on each WM_MOUSEMOVE event.
- Considerable speedup can be achieved for many applications by drawing to a memory device context and then *blitting* the result to the actual screen.
- The size of the area that is blitted is set to match the *invalid* rectangle. Windows is prevented from erasing the physical screen by changing the call to *InvalidateRect()* so that the second parameter is *FALSE* rather than *TRUE*.
- See EX11B.

  Note - OnPaint() is used since this technique is only relevant for the display device.

Bitmap Buttons

- You can add buttons to your dialogs that have a customized look by the use of a bitmap.
- These buttons are actually of a type known as *ownerdraw*.
- The *CBitmapButton* class does just about all the work for you.
- These are the steps:
  1. Create the bitmap images for the *up* and *down* button states. The resource names must end in "U" and "D". Note - resource names must be strings and not integers.
  2. The down state button is usually the up state button with inverted colors, but it can be anything.
  3. Create the buttons in the dialog. Check the *Owner Draw* option. The button label is the same as the bitmap's name WITHOUT the U or D.
  4. Implement functions to handle the button presses.
  5. Use the dialogs *OnInitDialog()* function to call the *CBitmapButton::AutoLoad()* function for each button This sets up the button's image. That's it!

Creating the Buttons



bmb - Microsoft Visual C++ - [bmb.rc - "BMU" (Bitmap)]
File Edit View Insert Project Build Image Tools Window Help
MyDialog
(All class members)
OnInitDialog
Start
bmb - Microsoft Visua...
Microsoft PowerPoint - [mf...
4:44 PM

Bitmap Properties

General | Palette

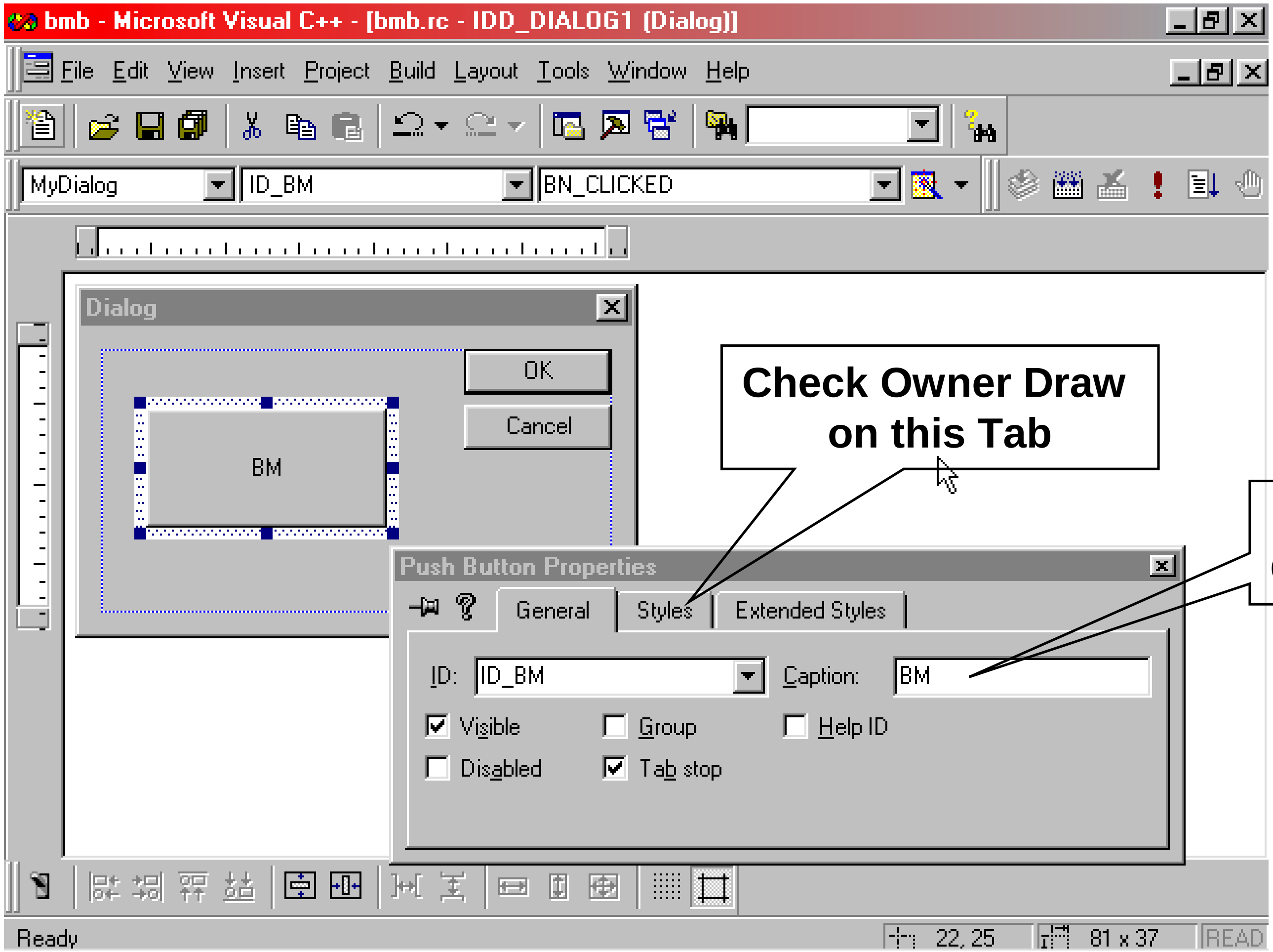
ID: "BMU"

Width: 50 Height: 45 Colors: 16

File name: res\bmu.bmp

□ Save compressed

Creating the Dialog



bmb - Microsoft Visual C++ - [bmb.rc - IDD_DIALOG1 (Dialog)]

MyDialog | ID_BM | BN_CLICKED

Push Button Properties

ID: ID_BM
Caption: BM

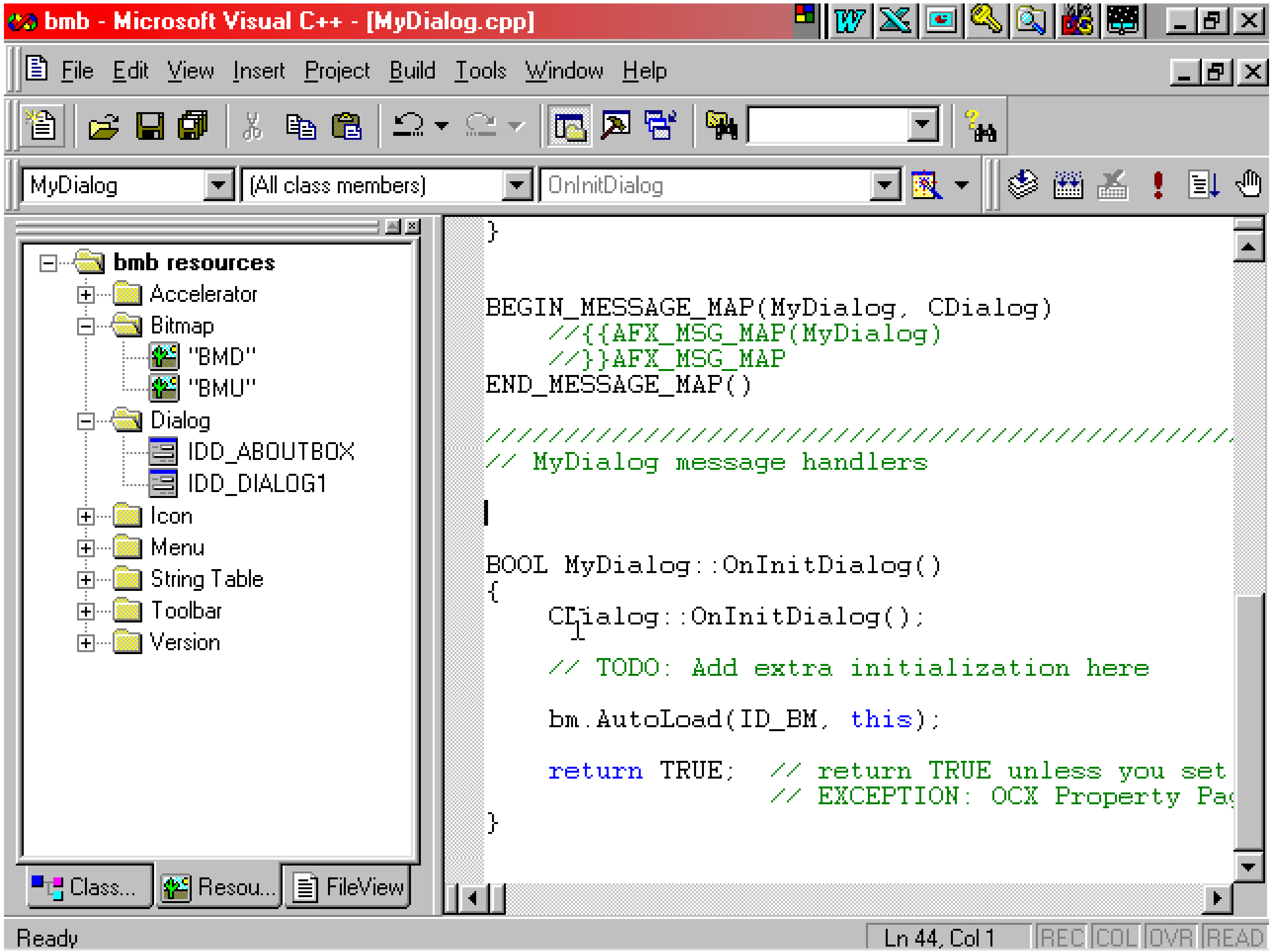
☑ Visible
□ Group
□ Help ID
□ Disabled
☑ Tab stop

Ready
22, 25
81 x 37
READ

Check Owner Draw on this Tab

Note Caption

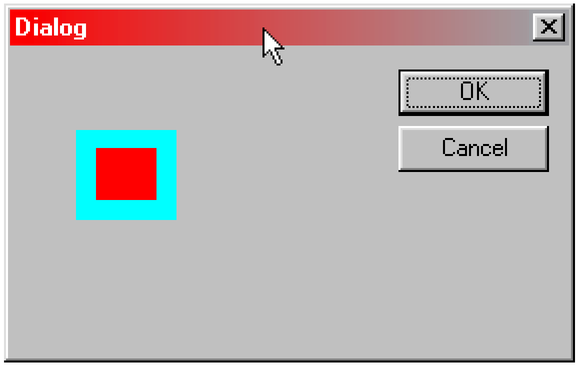
OnInitDialog Code



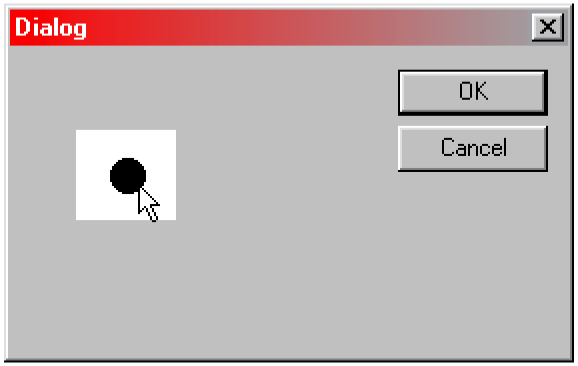
```
}

BEGIN_MESSAGE_MAP(MyDialog, CDialog)
    //{{AFX_MSG_MAP(MyDialog)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

///////////////////////////////////////////////////////////////
// MyDialog message handlers


BOOL MyDialog::OnInitDialog()
{
    CDialog::OnInitDialog();

    // TODO: Add extra initialization here

    bm.AutoLoad(ID_BM, this);

    return TRUE;  // return TRUE unless you set
                  // EXCEPTION: OCX Property Pag
}
```

The Dialog

Up State



Down State



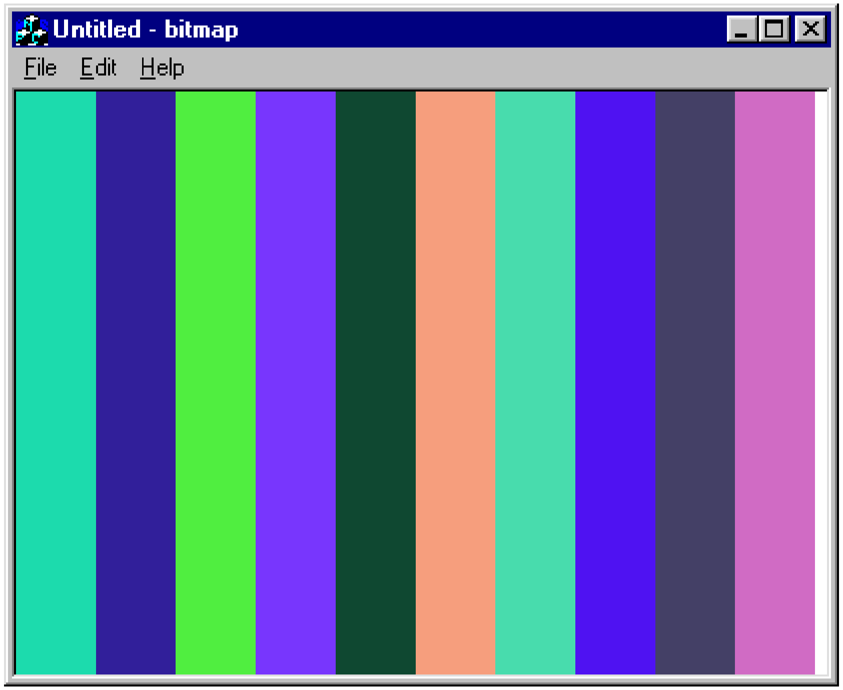
Palettes

- The use of palettes is pretty much restricted to 256 color mode.
- Both 16-bit and 24-bit color modes will map RGB values directly.
- Palette programming can be fairly tricky and we won't go into the details.
- The book has an implementation of a *CDib* class that can be used to view .BMP files and for other purposes. It handles the loading of a palette.
- The following program is a simple demonstration of drawing using a palette. The program draws 10 vertical color bars in random colors. Every time the window is repainted the colors change because a new palette is created and *realized* into the system palette.

Rainbow OnDraw() Code

```
void CxxxView::OnDraw(CDC* pDC)
{
    CPalette mypal, *poldp;
    CRect rect;
    int i;
    const int WIDTH=40;    //width of bars in pixels

    LPLOGPALETTE pLogPal;
    int Entries=10;

    pLogPal = (LPLOGPALETTE) new char[2 * sizeof(WORD) +
        Entries * sizeof(PALETTEENTRY)];
    pLogPal->palVersion = 0x300;
    pLogPal->palNumEntries = Entries;
    for(i = 0; i < Entries; i++) {
        pLogPal->palPalEntry[i].peRed = rand()%256;
        pLogPal->palPalEntry[i].peGreen = rand()%256;
        pLogPal->palPalEntry[i].peBlue = rand()%256;
        pLogPal->palPalEntry[i].peFlags = 0;
    }
    mypal.CreatePalette(pLogPal);
    delete pLogPal;
    poldp=pDC->SelectPalette(&mypal, 0);
    pDC->RealizePalette();

    for (i=0; i<10; ++i)
    {
        rect=CRect(i*WIDTH, 0, i*WIDTH+WIDTH, 300);
        pDC->FillSolidRect(rect, PALETTEINDEX(i));
    }
    pDC->SelectPalette(poldp, 0);
}
```

Rainbow Output



The Windows Timer

- Windows allows a maximum of 32 timers to be active. Each timer is able to measure time in milliseconds.

  Note - the PC clock tick is actually used to increment the timers. This clock runs at approximately 18 Hz.

- The *CWnd* member functions *SetTimer()* and *KillTimer()* are used to manage the timers.
- Once a timer is set to the appropriate number of milliseconds, *WM_TIMER* messages will be generated after each time interval until the timer is killed.
- Windows merges multiple pending WM_TIMER messages, so these messages can’t be used for accurate time keeping!
- Compute bound applications can give up control by using the Windows *PeekMessage()* function. (See example in the text.)

Idle Processing

- When the application framework does not have any messages to dispatch it performs On-Idle processing.
- The *CWinApp* function *OnIdle()* can be used to do some housekeeping when there is nothing else to do.
- The default *OnIdle()* function does nothing.
- Note - *OnIdle()* does not get called when a modal dialog or a menu is active.

Main Frame Window and the Document Class

- The main frame window (*mainfrm.h* and *mainfrm.cpp*) is the window that contains:

  The title bar.

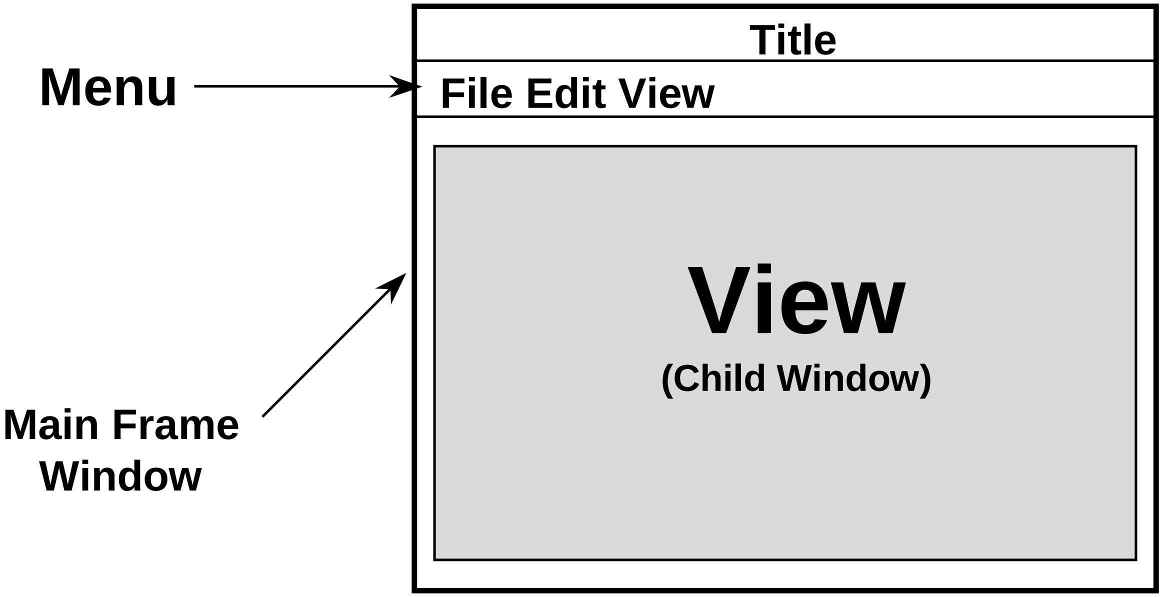
  The menu bar.

  The view (for SDI).

  A toolbar and/or status bar window if those options are selected.

- MFC handles routing messages from the frame window to the view (or document).
- The document is displayed by its view (or views) and is not itself a class derived from *CWnd*.

Main Frame Window and the Document Class - contd.



Menus and Accelerators

- Adding menus is as simple as using the Graphic Editor.
- Once the menu is created (or edited from the default), you add message handlers for the menu items. This is not any different than the methods we have been using to add handlers.
- *Class Wizard* will automatically generate the skeleton code for each menu item.

Keyboard Accelerators

- In addition to the ability to add an ampersand in the item name to define the letter for a *hot key* for the item, you can define any key to activate the menu item at any time.
- The Graphic Editor lets you set this up at the same time you edit the menu (in a different window).
- The handling of accelerators is automatic and within the MFC code.

Command Routing

- MFC routes command messages according to a specific sequence until it finds a handler:

SDI Applications

View
Document
SDI main frame
Application

MDI Application

View
Document
MDI child frame
MDI main frame
Application

- The usual places to place handlers are in the view or document classes.
- Note - your derived classes are called rather than the corresponding base classes for the classes listed above.

Update Command User Interface Messages

- You have probably seen situations where menu items are either checked or grayed out.
- To accomplish this in SDK programming is a bit of work. MFC simplifies things by providing a special message associated with each *popup* menu item.
- When a user activates the popup menu, MFC generates a special message, an *ON_UPDATE_COMMAND_UI* message.
- This message can be used to handle checking or graying the menu item.
- MFC has a built in mechanism to disable (gray) menu items for which there are no active message handlers.
- The following code is an example of enabling or disabling a menu item, *Edit-ClearAll*:

```
void CMyDoc::OnUpdateEditClearAll(CCmdUI* p)
{
      p->Enable(m_ClearAllFlag);   //flag is true or false
}
```

Advanced Menu Features

- The *CMenu* class can be used for a variety of menu operations.
- A menu can be *dynamically* created and modified.
- A windows menu can be changed using the *CWnd::SetMenu()* function call.
- Graphics can be used in menus. The Petzold book has an example of this.

Toolbars

- The application framework provides all the necessary code to set up the main toolbar.
- The toolbar is an object of class *CToolBar*.
- The toolbar sizes and repositions itself if the main frame window changes.
- The toolbar *buttons* come from a <u>single</u> bitmap in the resource file. The Graphic Editor in VC++ version 4 has a new toolbar editing feature, rather than using the bitmap editor.
- When clicked, toolbar buttons function just like menu items and send command and command UI messages.
- Each button’s bitmap is a *tile* which is 15 x 16 pixels.
- Toolbar buttons can act as simple pushbuttons or can be toggled up and down (similar to checking a menu item).

Toolbars - contd.

- Toolbar buttons can have the following states:

State	Meaning
0	Normal, unpressed
TBBS_PRESSED	Pressed with mouse
TBBS_CHECKED	In the down (checked) state
TBBS_DISABLED	Unavailable for use
TBBS_INDETERMINATE	Enabled (not up or down)
TBBS_CHECKED | TBBS_DISABLED	In the checked state, but unavailable for use

- Normally a toolbar button has a corresponding menu item.
- A toolbar has an associated resource. It is of the form:

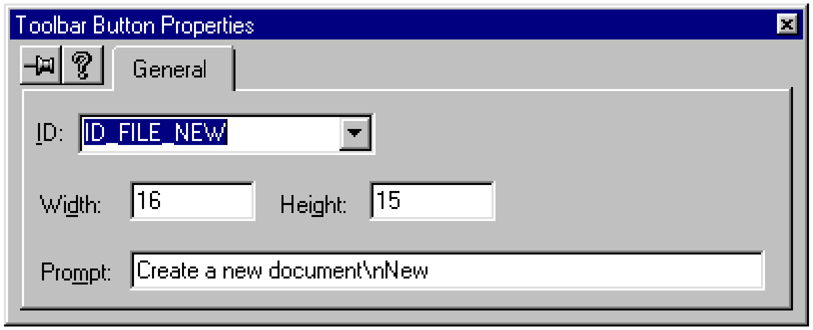
```
IDR_MAINFRAME TOOBAR DISCARDABLE 16, 15
BEGIN
   BUTTON         ID_FILE_NEW
   BUTTON         ID_FILE_OPEN
   BUTTON         ID_FILE_SAVE
   SEPARATOR
   BUTTON         etc.
END
```

Toolbar Update UI Messages

- Basically this mechanism works the same way as for menu items.
- The *CCmdUI::Enable()* function is used to enable/disable the toolbar item.
- *CCmdUI::SetCheck()* can also be used.
- Unlike menus, command UI messages for toolbars are sent during the application's idle processing loop.
- Note - keyboard accelerators continue to work even when a toolbar button is disabled. You must handle this problem yourself.

Tool Tips

- These are the little flyout windows that pop up when you position your mouse over a toolbar button.
- They are implemented automatically and all you need to do is provide the text for the tip.
- This is done by appending "\n*tiptext*" to the prompt string for the menu item/toolbar buton. Just double click to bring up the properties window and add the text to the prompt as shown below.



ID: ID_FILE_NEW

Width: 16 Height: 15

Prompt: Create a new document\nNew

Locating the Main Frame Window

- For SDI applications the main frame window is the parent window of the view. You can use the *CWnd::GetParentFrame()* function to get a pointer to this class.
- For MDI applications there is an extra window in-between the view and the main frame window.
- A safer and more general method is to get a pointer to the main application class using a global function, *AfxGetApp()*, and then to look at the *m_pMainWnd* data member.
- This data member holds a pointer to the main frame window. You can then get pointers to the toolbar, status bar, and child windows such as the view.

The Status Bar

- The status bar is used strictly for output.
- The standard status bar displays:

  A menu item prompt
  Caps lock state
  Num lock state
  Scroll lock state

- The *indicators* array and following calls in *mainframe.cpp* sets up the status bar:

```
static UINT BASED_CODE indicators[] =
  {
        ID_SEPARATOR, // status line indicator
        ID_INDICATOR_CAPS,
        ID_INDICATOR_NUM,
        ID_INDICATOR_SCRL,
  };

   if (!m_wndStatusBar.Create(this) ||
                !m_wndStatusBar.SetIndicators(indicators,
                  sizeof(indicators)/sizeof(UINT))) etc.
```

The Status Bar - contd.

- You can output to the *message line pane* by using the *SetPaneText()* function. The trick is in getting a pointer to the status bar’s window.

```
CStatusBar* pStatus = (CStatusBar*)
     AfxGetApp()->m_pMainWnd->GetDescendantWindow
           (AFX_IDW_STATUS_BAR);
 pStatus->SetPaneText(0, "Some message text");
```

- Note that a cast is needed of the return value from *GetDescendantWindow()* to (*CStatusBar *)*.
- The framework will overwrite your text with "Ready" if you do anything to cause the status bar to be changed, e.g., activate a menu.
- To take full control of the status bar you must change the status bar's ID to one of your own.

The Status Indicators

- The status indicators are linked to command UI messages that test the state of the appropriate keys.
- *GetKeyState()* function calls are made for each of the indicators.
- It is possible to provide a custom status bar with more than one message pane and numerous indicators of your choice.
- EX13B demonstrates how to hook into the status bar mechanism. Using this as a guide. It is not difficult.
- Note - you must change the ID in the default handling of the *OnViewStatusBar()* and *OnUpdateViewStatusBar()* functions in the main frame class. This allows the user to turn your custom status bar on and off as is done with the default.

The *CFrameWnd* Class

- Most MFC applications will have a main frame window.
- Your main frame window derived class can be used to handle various initializations that are not ideally placed in the application class.
- For example, initializations for the size and position of the main frame window itself.
- *CFrameWnd::ActivateFrame()* can be used to set up particular window details before the window is actually displayed on the screen.
- *CWnd::SetWindowPlacement()* can be used to set the size and position.
- Note - be sure to call the base class *ActivateFrame()* if you override this function.
- The *nCmdShow* parameter can be changed to display a window iconized, maximized, etc.

PreCreateWindow() Function

- *PreCreateWindow()* allows you to take control of the parameters passed to the Win32/16 *CreateWindow()* function.
- This allows the window style to be changed, e.g., add scroll bars.
- Look up *CREATESTRUCT* in the Win32/16 SDK documentation for details of this structure. You make modifications to this structure in your *PreCreateWindow()* code.

  Take a look at the *CreateWindow()* function and *WM_CREATE* message while you are at it for more details.
- Starting with version 4 an override is automatically provided. WIth 1.52 you need to provide this override yourself.

Using *.INI* Files

- Originally, virtually all Windows applications placed their initialization values in the *win.ini* file in the *windows* directory.
- This is no longer considered good practice. Each application should have its own initialization file (usually named after the application’s *.exe* file name.
- Note - you can override this by setting *m_pszProfleName* in the application class (not mentioned in the book). This is initiallly set from the AFX_IDS_APP_TITLE string in the application's resources. Edit this string if you want to change the applications name as well. It defaults to the workspace name.
- Under Windows 95 you can optionally have this information placed in the Windows Registry. You must make a call to *CWinApp::SetRegistryKey()* with the name of the top level of the application's hierarchy. This call should be made in your *InitInstance()* function.

Using *.INI* Files - contd.

- Four *CWinApp* member functions can be used to read and write this information:

  GetProfileInt()

  WriteProfileInt()

  GetProfileString()

  WriteProfileString()

- Multiple *section* names can be used and as many data items in a section as you want. Only integer and string data types are allowed. However, you can always use strings to hold almost anything.

Using *.INI* Files - contd.

- Example:

```
AfxGetApp()->WriteProfileInt("Init", "Retries", 100);
```

- This would result in a *.ini* file containing:

```
[Init]
Retries=100
```

- We can obtain the value of a particular item with:

```
 int retries = AfxGetApp()->GetProfileInt
            ("Init", "Retries", 0);
```

- The third argument is a default value to be returned if the file does not exist or the item does not exist.

The *CString* Class

- This extremely valuable class is one you should get to know.
- Objects of type *CString* can be used as regular C style strings because of operator overloading that allow type conversions. (a bit of care is needed though)
- *CString* objects have lots of other nice features such as the ability to concatenate them.
- *CString* objects can dynamically grow without needing to explicitly handle reallocation.
- Note - the DDX mechanism uses *CString* objects rather than C style strings.
- There are a few cases where you need to worry about explicit type casting for strings.
  - The *sprintf* and *wsprintf* functions are slightly different.
  - *wsprintf* is preferred because it is a built in Windows function.
- You can *lock down* the *CString* object's buffer and use the string directly as a C style string.

The *CDocument()* Class

- The example so far have not used the document class to hold the application’s data.
- For applications with more than one view, or MDI applications, it is essential to decouple the document data from the view.
- Navigating between the view and the document is quite simple. All views on a given document will have a pointer back to the document obtainable by the *GetDocument()* member function. The returned pointer can be used to directly access the document’s data.
- Caution - *GetDocument* is NOT a virtual function since it returns a type cast pointer. Be sure to call this function in your derived view class.

View Updates

- Going from the document to the view is not as simple because there may be more than one view.
- If data is changed in the document, the *UpdateAllViews()* function should be called.
- This results in *CView::OnUpdate()* being called for each view. Override this function to perform whatever needs to be done.
- One more function deserves mention. The *CView::OnInitialUpdate*() function is called when the application starts or the user clicks on *New*.
- If you override this function be sure to call the base class's *OnInitialUpdate*().

The *CFormView* Class

- This very nice class can be used as the base class for a view that is built from a dialog resource object.
- Although this class is similar to a dialog, it is not a *CDialog* derived class and doesn't support the same functionality as a true dialog. However, DDX and DDV data exchange functions can be used.
- Some tricks can be played using this class by casting a *CFormView* pointer to point to a *CDialog* class. (Note - in my opinion this is NOT good C++, but seems to be very much encouraged using MFC.) The only reason this works is that all these calls are doing is calling the corresponding API function. The alternative would be to make these Win32/16 API calls directly.

Diagnostic Dumping

- One of the nice features of MFC is the built in debugging checks performed when you compile under the DEBUG build mode.
- We have already seen the TRACE and VERIFY macros in action.
- An alternative to the use of TRACE is to use the *afxDump* object. This is an output stream similar to *cout* under DOS programming. For example:

```
#ifdef _DEBUG
  afxDump << "Count= " << count << "\n";
#endif
```

- As an extension to this basic mechanism, all classes derived from *CObject* have a *Dump()* member function that you can override to provide any information you want.

Reading & Writing Documents

- AppWizard generated code makes things much easier than you might imagine.
- *Serialization* is the process of reading or writing an object to a disk file.
- Classes can be made *persistent* by serializing them to disk. The exact format in the disk file is up to the programmer.
- We have already seen how easy it is to use the *CFile* class. MFC provides a *CArchive* class which is a bit more functional than *CFile* (e.g., overloaded << and >>).
- *CArchive* acts as a buffer class between the document object and the *CFile* object (the disk file).
- When you open or save a document, MFC will call the *Serialize* member function and pass it a *CArchiveI* object. You either read or write to this *CArchive* object just as you would to a file.

Reading & Writing Documents - contd.

- Example:

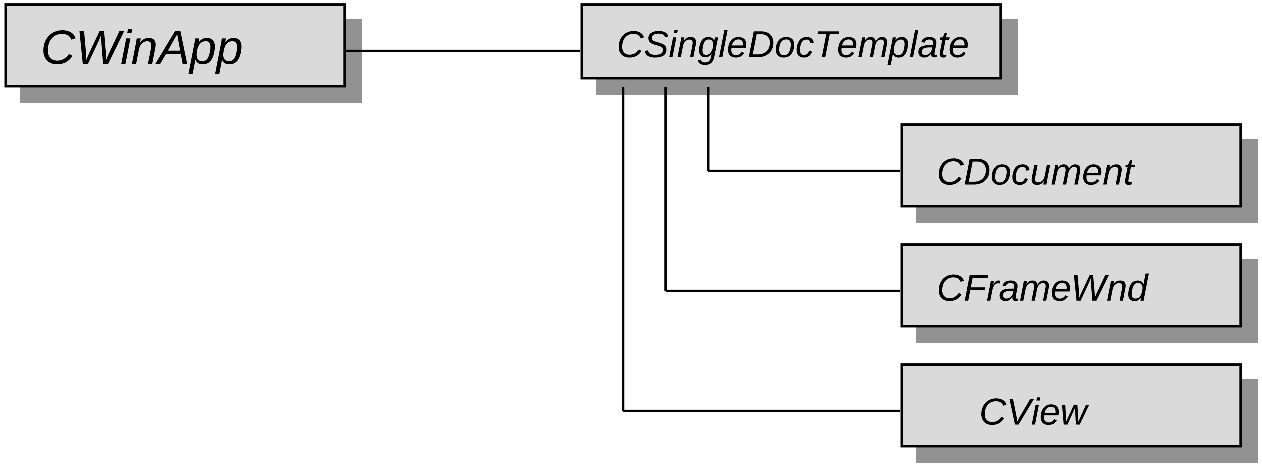
```
void CMydoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
        ar << ........;
    else
        ar >> ........;
}
```

- You would use stream I/O to input or output your class data members.
- This can be simple or complex depending on the application, but the method remains the same.
- For portability issues you should take into consideration the size of integers, etc. The use of user types such as WORD, BYTE, LONG, etc. is a good idea.
- Any object derived from *CObject* can override the virtual *Serialize* function. Serialization can be generalized for objects other than the simple data types by invoking the objects *Serialize()* function either directly or through the use of overloaded insertion or extraction operators.

The Document Template Class

- Both SDI and MDI applications make use of special classes known as document template classes. (*CSingleDocTemplate* and *CDocTemplate*)
- The document template ties together the frame window, document, and view classes.
- The setup of the document template mechanism is done in the application class derived from *CWinApp.*
- You don’t normally need to get involved with this mechanism unless you want to do something fancy, such as having more than one view on a document.

Standard Dcoument Template Organization



SDI Functionality

- Upon starting up the application or after the user has clicked on New in the File menu, the applications *OnFileNew* function is called. The default code sets up a chain of events.
- Of significance is the call made to the document classes *OnNewDocument()* function. Here is where you should initialize your document. The constructor can’t be used because several new documents may be created during the lifetime of the document class object.
- Opening existing documents ultimately results in a call to the documents *OnFileOpen()* function with a *CArchive* object already created as a result of the file open dialog. *OnFileOpen()* then calls *Serialize()*.You then need to serialize the file into your document.
- The *DeleteContents()* function is also called by both *OnFileNew* and *OnFileOpen*. Here is where you clean up the document since the destructor is not called until the application is closed.

SDI Functionality - contd.

- File save and save as menu items result in a call to the document's *OnSaveDocument()*.
- *OnSaveDocument()* calls *Serialize()*. Once again you can put all the I/O in *Serialize().*
- MFC maintains a document data member, *m_bModified,* that indicates if the document needs saving. It is set with the function *SetModifiedFlag()* and tested with *IsModified()*.
- You can tie this flag to the File menu Save item with a simple command UI handler.

  ```
  pCmdUI->Enable(IsModified());
  ```

- When you look over the AppWizard generated code for the application class and the document class you will see that handling input/output for the document is not difficult. The framework has done most of the work.

The Multiple Document Interface

- MDI applications are the most popular.
- MDI applications allow the user to have more than one document open at the same time.
- An MDI application is automated by the MFC framework. The App Wizard will set up an MDI application just like an SDI application. Most of the methods used for SDI applications are applicable for MDI applications.
- MFC adds all the functionality to *tile* and *cascade* MDI views.
- There is one big difference in the way in which the windows are set up for MDI vs. SDI. Your MDI view is not an immediate child window of the main frame window. Rather, it is a child of a special MDI child frame window.

The Multiple Document Interface - contd.

- The document template mechanism handles creating these windows automatically. Just remember that the view is in a child window of the main frame window.
- A built in feature is the ability to have more than one view on a document. Be prepared for this. This concept should drive home the importance of the document class.
- Unlike SDI applications, the document class objects are created and destroyed in response to opening and closing of the document. Constructors and destructors are invoked. You don’t have to worry about about clearing the document contents before loading. In this respect MDI is simpler than SDI.
- It is possible to have multiple document templates so that documents of different types can be opened by the same application. (Too advanced for us to cover.)

Printing and Print Preview

- Once again the MFC framework has come through with an enormous amount of work already done for you.
- Operations like printer setup and selection are handled by common dialogs activated from the built in code base.
- Print Preview is another feature built in. The appearance of a page on the screen in your view is not going to be exactly the same as the printer. The print preview mechanism lets you view the output as close as possible to the actual printed output.
- The view’s *OnPrint()* function calls the view’s *OnDraw()* function. Remember, that’s why you used *OnDraw()* and not *OnPaint()*. You can make double duty of the *OnDraw()* function.
- You can use *OnPrepareDC()* for setting up mapping modes if required. The *IsPrinting()* function will tell you if *OnPrepareDC()* is being called for printing.

Printing and Print Preview - contd.

- Two special functions in the view are called to set up information about the printing operation. Skeletons are provided by the App Wizard.
- *OnPreparePrinting()* is called prior to opening the print dialog box. The page range defaults are set up here.
- *OnBeginPrinting()* is called after the print dialog box is closed and before printing. Here is where you can set up special GDI objects such as fonts.
- *OnPrint()* (and then *OnDraw()*) is called once for each page of the document. Outputting is done by the usual GDI calls.
- *OnEndPrinting()* is called after printing the last page in order to delete any GDI objects etc.
- The examples in the text can be used as models for your own printing. Of course, sophisticated word processing applications will require a lot of work in setting up the page layout via GDI calls. However, this is not really any different than the screen.

Splitter Windows and Multiple Views

- A splitter window is a special window that holds two or more views in different panes.
- Each pane can have horizontal and vertical scroll bars.
- Multiple views on the same document do not have to all be of the same type. For example, a file viewer might have one view that displays the file in text and another view in hexadecimal notation. A debugger might have one view in source code and another in machine language. You get the idea.
- Splitter windows and multiple views can be combined in a variety of ways for both SDI and MDI applications.
- A quick look at the examples in the text as well as the *Scribble* application in the MFC tutorial will give you some models.

Context- Sensitive Help

- Windows provides the *WINHELP* help engine to display help files.
- Help files are written in *Rich Text Format (RTF)*.
- You can use a WYSIWYG text editor such as Word that supports RTF so that you don’t have to work directly in ASCII.
- After the RTF file is created with the proper hypertext links you run it through the help compiler under the control of a help project file. The resulting file can be viewed with WINHELP directly.
- App Wizard will set up the context sensitive help mechanism for you. Of course you need to author the actual help file.
- WINHELP can be activated by the *WinHelp()* function in the application class. It can be started up on any topic desired.
- To implement context sensitive help you provide a keyboard accelerator for the F1 key and use it to call *WinHelp()*.

Where to go from here

- What we have covered should allow you to write fairly sophisticated applications.
- Rolling out simple Windows programs is a snap if the user interface fits the standard App Wizard generated framework.
- The MFC library is growing with each new version. Topics such as the use of ODBC, DAO and OLE are not simple. The text gives only a brief introduction to such MFC use. Don’t even think about serious work in those areas unless you are reasonably comfortable with the basic MFC framework.
- The best way to continue is to review all the examples in the text and the Visual C++ documentation and then just start writing some applications. Learn the MFC class hierarchy so you will know where to look for the existence of an appropriate function.
- Don’t be afraid to plunge into something. Set up the basic application and then add features one by one. Since App Wizard sets up your skeleton it’s just a question of filling it in.

Where to go from here - contd.

- One point that the book does not stress enough is the need to go to the Win32 or Win16 API documentation. Always remember that MFC encapsulates the underlying functionality. You really won't know what you can do without better familiarity with the API.
- Consider buying a copy of Petzold as a reference. Serious developers will probably want to buy the complete hard copy VC++ documentation as well as the complete Win32 API documentation set. Although all this is online, most people find the hard copy easier to use while learning.
- Join the Microsoft Developer Network and subscribe to the CD ROMs. You get the latest documentation, bug lists, ap notes and can keep up with the new releases of the OS's etc.