LAB MANUAL
FOR
VPT LAB

WCTM

HANDLING KEYSTROKES

1. Open Visual C++ and click the New item in the File menu, opening the New dialog box.

2. Now select the MFC AppWizard(exe) entry in the New dialog box.

3. Give the new program the name Keystrokes in the Project name box.

4. Click OK to starts the Visual C++ AppWizard.

5. Click the option marked Single Document in the AppWizard, click Finish.

6. Here, the AppWizard indicates four classes will be created: CKeystrokesApp, CMainFrame, CKeystrokesDoc, CKeystrokesView.

7. Declare the StringData variable on the document's header file,KeystrokesDoc.h, in the protected part.

```
class CKeystrokesDoc : public CDocument
{
protected: // create from serialization only
            CKeystrokesDoc();
            DECLARE_DYNCREATE(CKeystrokesDoc)
            CString StringData;
                    .
                    .
};
```

8. Initialize that string to an empty string -" "- in the document's constructor, which we find in KeystrokesDoc.cpp.

```
CKeystrokesDoc::CKeystrokesDoc()
{
```

```
        StringData=" ";
        // TODO: add one-time construction code here
}
```

9. Add a new event handler –OnChar( )- to our view class which is called every time the user types the character.

10. The character the user typed is now in the nChar parameter, which we store in the our data string object, StringData. The object in our document, so we need a pointer (pDoc) to our document object.

11. Add the character nChar to the string StringData.

```
void CKeystrokesView::OnChar(UINT nChar, UINT nRepCnt,
        UINT nFlags)
{
// TODO: Add your message handler code here and/or call default
        CKeystrokesDoc* pDoc=GetDocument();
        ASSERT_VALID(pDoc);
        pDoc->StringData+=nChar;
        Invalidate();

        CView::OnChar(nChar, nRepCnt, nFlags);
}
```

12. We'll handle the display of our data in the view's OnDraw( ). We need to draw the text string, which we do with TextOut( ).

```
void CKeystrokesView::OnDraw(CDC* pDC)
{
        CKeystrokesDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);

        pDC->TextOut(0,0,pDoc->StringData);
        // TODO: add draw code for native data here
}
```

WCTM

HANDLING THE MOUSE

1. Open Visual C++ and click the New item in the File menu, opening the New dialog box.

2. Now select the MFC AppWizard(exe) entry in the New dialog box.

3. Give the new program the name Carets in the Project name box.

4. Click OK to starts the Visual C++ AppWizard.

5. Click the option marked Single Document in the AppWizard, click Finish.

6. Here, the AppWizard indicates four classes will be created : CCaretsApp, CMainFrame, CCaretsDoc, CCaretsView .

7. Declare the StringData variable on the document's header file, CaretsDoc.h, in the protected part.

```
class CCaretsDoc : public CDocument
{
protected: // create from serialization only
        CCaretsDoc();
        DECLARE_DYNCREATE(CCaretsDoc)
        CString StringData;
            .
            .
};
```

8. Initialize that string to an empty string -" "- in the document's constructor, which we find in CaretsDoc.cpp.

```
CCaretsDoc::CCaretsDoc()
{
        StringData=" ";
        // TODO: add one-time construction code here
}
```

9. Add a new event handler –OnChar( )- to our view class which is called every time the user types the character.

10. The character the user typed is now in the nChar parameter, which we store in the our data string object, StringData. The object in our document, so we need a pointer(pDoc) to our document object.

11. Add the character nChar to the string StringData.

```
void CCaretsView::OnChar(UINT nChar, UINT nRepCnt,
    UINT nFlags)
{
// TODO: Add your message handler code here and/or call default
```

```
        CCaretsDoc* pDoc=GetDocument();
        ASSERT_VALID(pDoc);
        pDoc->StringData+=nChar;
        Invalidate();

        CView::OnChar(nChar, nRepCnt, nFlags);
}
```

12. Set a boolean variable named CaretCreated in view object to keep track that caret has been created or not & CaretPosition & x,y.

```
class CCaretsView : public CView
{
protected: // create from serialization only
        CCaretsView();
        DECLARE_DYNCREATE(CCaretsView)
        CPoint CaretPosition;
        int x,y;
        boolean CaretCreated;
// Attributes
        .
        .
};
```

13. Set CaretCreated to false in the View's Constructor.

```
CCaretsView::CCaretsView()
{
CaretCreated=false;
// TODO: add construction code here
}
```

14. We're ready to create our new caret. We'll make the caret the same height and width as our text.We call CreateSolidCaret() to actually create the caret.

15. We'll store caret's position in a new CPoint object named CaretPosition. The CPoint class has two data members x & y which holds the position of Caret.

16. Initially, set caret's position to (0,0) in OnDraw().

17. We set the caret's position with SetCaretPos(), show caret on the screen with ShowCaret(), and set the CaretCreated Boolean flag to true.

18. In next step the caret is to move as the user types text.

19. Place the caret at the end of displayed text string.

20. To display the caret at the end of the text string, we first hide it using HideCaret().

WCTM

21. Set the CaretPosition & Show the Caret.

```
void CCaretsView::OnDraw(CDC* pDC)
{
        CCaretsDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        if(!CaretCreated)
        {
                TEXTMETRIC textmetric;
                pDC->GetTextMetrics(&textmetric);

                CreateSolidCaret(textmetric.tmAveCharWidth/8,textmetric.tm
                Height);//(14)
                CaretPosition.x=CaretPosition.y=0;
                        //(16)
                SetCaretPos(CaretPosition);//(17)
                ShowCaret();                    //(17)
                CaretCreated=true;          //(17)
        }
        pDC->TextOut(x,y,pDoc->StringData);
                                        //(18)
        CSize size=pDC->GetTextExtent(pDoc
        ->StringData);          //(19)
        HideCaret();            //(20)
        CaretPosition.x=x+size.cx;
        CaretPosition.y=y;
        SetCaretPos(CaretPosition);   //(21)
        ShowCaret();                          //(21)
        // TODO: add draw code for native data here
}
```

22. To handle left mouse button down we select LButtonDown, point parameter , an Object of the CPoint class, holds mouse's present location.

23. Store the variables in x & y.

```
    void CCaretsView::OnLButtonDown(UINT nFlags, CPoint point)
    {
        // TODO: Add your message handler code here and/or call default
                x=point.x;
                y=point.y;
       CCaretsDoc* pDoc=GetDocument();
                ASSERT_VALID(pDoc);
                pDoc->StringData.Empty();
                Invalidate();
                CView::OnLButtonDown(nFlags, point);
}
```


WCTM

CREATE MENU ADD SHORTCUT, ACCELERATOR KEYS, STATUS BAR AND TOOLS

1. Open Visual C++ and click the New item in the File menu, opening the New dialog box.

2. Now select the MFC AppWizard(exe) entry in the New dialog box.

3. Give the new program the name DME in the Project name box.

4. Click OK to starts the Visual C++ AppWizard.

5. Click the option marked Single Document in the AppWizard, click Finish.

6. Here, the AppWizard indicates four classes will be created : CDMEApp, CMainFrame, CDMEDoc, CDMEView .

7. Click Resources tab in the VC++ viewer window to open the Menu Editor, find the folder marked Menu & open it. Double click the entry in that folder, IDR_MAINFRAME, Opening the Menu Editor.

8. Add new menu item, Display to the Edit menu.

9. Double click this new menu item, opens the Menu Item Properties box. Place the caption &Display, & makes Display a shortcut menu. In Prompt Box write "Display the message". This text appear in the status bar when user lets mouse rest on this menu item.

10. To add Accelerator folder to Display menu item.Open Accelerator folder from the viewer window.Double-click the IDR_MAINFRAME entry, open Accelerator editor.

11. Double click on the last blank entry, opens the Accel Properties box. Select ID , ID_EDIT_DISPLAY & connect Ctrl+Alt+F9(VK_F9).

12. Open ClassWizard, ID_EDIT_DISPLAY is listed in the Object Ids box. Click ID_EDIT_DISPLAY, then click the Command entry in the Message Box. This makes class wizard suggest a name for the event handler of OnEditDisplay()- click OK to accept the name.

13. Declare the StringData variable on the document's header file, DMEDoc.h.

```
class CDMEDoc : public CDocument
{
public:
        CString StringData;
        .
};
```

14. Initialize that string to an empty string -" "- in the document's constructor.

WCTM

```
CDMEDoc::CDMEDoc()
{
        StringData=" ";
}
```

15. Double click on OnEditDisplay() in the Class Wizard Member functions box,

```
void CDMEView::OnEditDisplay()
{// TODO: Add your command handler code here
}
```

16. To Create new Dialog Box, select Resources item in Insert menu. Select dialog entry & click the New button.

17. Add controls, drag and drop, a button & a text box(or edit box), onto the dialog box.

18. Select the control(button) by clicking & type new caption(Click me) & control ID IDC_BUTTON1.In the same way, text box has IDC_EDIT1.

19. Create class, from Class Wizard, click Add Class. Type name Dlg for new class. Clicking OK opens Class Wizard.Select IDC_BUTTON1 from Object ID & Double click the BN_CLICKED entry in Message Box.

20. Start Class Wizard, & click the Member Variable tab. Now select text box control,IDC_EDIT1, click add variable button. Give name m_text in the Member variable Name box, Value in Category Box, CString in the Variable text box. Click OK.

21. The variable m_text is connected to the IDC_EDIT1 using special method that class wizard had added to our Dlg dialog box class.

```
void Dlg::DoDataExchange(CDataExchange* pDX)
{CDialog::DoDataExchange(pDX);
        //{{AFX_DATA_MAP(Dlg)
        DDX_Text(pDX, IDC_EDIT1, m_text);
        //}}AFX_DATA_MAP
}
```

22. Place text in the text box.

```
void Dlg::OnButton1()
{
        m_text="Welcome to VC++";
        UpdateData(false);
}
```

23. Add code to OK button .

```
void Dlg::OnOK()
{
        UpdateData(true);
        CDialog::OnOK();
}
```

24. To display the dialog box we have to connect it to “Display” item in the Edit

WCTM

menu. Create new object dlg of our dialog box's class Dlg. Use DoModal() method, this method returns an integer value, which we stored in results.

25. At this point, dialog box appears on the the screen. If user click OK, string from the text box will be stored in our document. And we place the text in the dialog's box m_text variable in the StringData object.

```
void CDMEView::OnEditDisplay()
{
    Dlg dlg;        //24
    int result=dlg.DoModal();  //24
    if(result==IDOK)            //25
    {
        CDMEDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        pDoc->StringData=dlg.m_text; //25
        Invalidate();
    }
}
```

26. In view's OnDraw() method, we draw the text from the dialog box:

```
void CDMEView::OnDraw(CDC* pDC)
{
    CDMEDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    pDC->TextOut(0,0,pDoc->StringData);
}
```

We can Open the item Display from Edit menu by following ways:

(i) Edit→Display.
(ii) Using shortcut key Alt+E then D.
(iii) Using Accelerator key Alt+Ctrl+F9.
(iv) Using first button of the toolbar.

When user highlights a menu item, a message appears at the bar at the bottom of our program's window to provide more information about the item.

WCTM

MDI (MULTIPLE DOCUMENT INTERFACE)

1. Open Visual C++ and click the New item in the File menu, opening the New dialog box.

2. Now select the MFC AppWizard(exe) entry in the New dialog box.

3. Give the new program the name Multiview in the Project name box.

4. Click OK to starts the Visual C++ AppWizard.

5. Click the option marked Multiple Document in the AppWizard, click Finish.

6. Here, the AppWizard indicates four classes will be created : CKeystrokesApp, CMainFrame, CKeystrokesDoc, CKeystrokesView .

7. Declare the StringData variable on the document's header file, MultiviewDoc.h, in the protected part.

```
class CMultiviewDoc : public CDocument
{
            CString StringData;
                  .
                  .
};
```

8. Initialize that string to an empty string -" "- in the document's constructor, which we find in MultiviewDoc.cpp.

```
CMultiviewDoc::CMultiviewDoc()
{
            StringData=" ";
}
```

9. We have to store data on the disk:

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

10. Add a new event handler –OnChar( )- to our view class which is called every time the user types the character.

WCTM

11. The character the user typed is now in the nChar parameter, which we store in the our data string object, StringData. The object in our document, so we need a pointer(pDoc) to our document object.

12. Add the character nChar to the string StringData.

13. We'll handle the display of our data in the view's OnDraw( ). We need to draw the text string, which we do with TextOut( ).

```
void CMultiviewView::OnDraw(CDC* pDC)
{
        CMultiviewDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        pDC->TextOut(0,0,pDoc->StringData);
}
```

14. Using UpdateAllViews() method, we're able to refresh all the views into this document now that user has typed a new character.

15. Set the document's modified flag using SetModifiedFlag(),means that if user tries to close the current document without saving it, the program will ask if they want to save it before the data in it is lost.

```
void CMultiviewView::OnChar(UINT nChar, UINT nRepCnt,
    UINT nFlags) //10 11
{
        CMultiviewDoc* pDoc=GetDocument();
        ASSERT_VALID(pDoc);
        pDoc->StringData+=nChar;   //12
        Invalidate();
        pDoc->UpdateAllViews(this,0L,NULL);//14
        pDoc->SetModifiedFlag();      //15
        CView::OnChar(nChar, nRepCnt, nFlags);
}
```

WCTM

CREATE A DIALOG BASED APPLICATION

1. Open Visual C++ and click the New item in the File menu, opening the New dialog box.

2. Now select the MFC AppWizard(exe) entry in the New dialog box.

3. Give the new program the name dialogbased in the Project name box.

4. Click OK to starts the Visual C++ AppWizard.

5. Click the option marked Dialog Based in the AppWizard, click Finish.

6. Here, the AppWizard indicates four classes will be created : CdialogbasedApp, CMainFrame, CdialogbasedDoc, CdialogbasedView .

7. Click the Resources tab, open the Dialog folder, and click the entry for our program main window, IDD_BUTTONS_DIALOG. This opens the dialog editor.

8 Add a new button with the caption "Click me" to our code.

9. Open Class Wizard to connect Click me button to our code.

10. Use Class Wizard to connect the button to an event handler, OnButton1().

11. Double-click IDC_BUTTON1 in the Object IDs box, and double click BN_CLICKED in the Message Box. This creates OnButton1() :

```
void  Cdialog::OnButton1()
{

}
```

12. Add a member variable, m_edit to EDIT box i.e IDC_EDIT1.

13. Add following code to OnButton1():

```
void  Cdialog::OnButton1()
{
      m_edit.SetWindowText(CString("Hello!!"));
}
```

WCTM

DATABASE CONNECTIVITY IN VC++

1. Open Visual C++ and click the New item in the File menu, opening the New dialog box.

2. Now select the MFC AppWizard(exe) entry in the New dialog box.

3. Give the new program the name dbvc in the Project name box.

4. Click Next button, There is a Question "What Database support would you like to include?", click the radio button labeled Database View with File Support.

5. Next, click the button labeled Data Source to open the Database Options box.

6. We select the DAO button, and specify the dbs.mdb filesas our Data source.

7. When the Select Database Tables appears, click the Student table and the OK button.

8. Now we click the OK button and finish button in App Wizard, letting AppWizard create the New Program.

9. Open the main view, IDD_DBVC_FORM, in the dialog editor, and add the controls : two text boxes & a button with caption " Display the current record's fields ".

10. Using Class Wizard, we connect an event handler, OnButton1() to the button & four member variable m_text1, m_text2, m_text3, m_text4 to four text boxes & four static box to four text boxes.

11.Add codes to OnButton1():

```
void CDbvcView::OnButton1()
{
        m_text1=m_pSet->m_Name;
        UpdateData(false);

        m_text2=m_pSet->m_Roll_No;
        UpdateData(false);

        m_text3=m_pSet->m_Branch;
        UpdateData(false);

        m_text4=m_pSet->m_Semester;
        UpdateData(false);

}
```


WCTM

AN INTERNET APPLICATION IN VC++ (USING BROWSER CONTROL)

1. Create a dialog based application.

2. Add a web browser control on the dialog box. If it is not there in the control box then select Project->Add to project->components and controls, from the list of controls select the “Registered Active X controls”. Then select the “Microsoft Web Browser” and press insert button, this will add the web browser control in your control toolbox.

3. Add a variable to it as m_browser

4. Add a button to the dialog box and event handler for it.

```
void CBrowserDlg::OnButton1()
{
        m_browser.Navigate("http://www.microsoft.com",0,0,0,0);
}
```

WCTM

CREATING A WEB BROWSER WITHOUT USING BROWSER CONTROLS

1. Create a dialog based application, add a text box and a button on the dialog
2. Add event handler for the button
3. Include the header file "afxinet.h"
4. Create a new internet session by declaring an object of the MFC class CInternetSession. This supports HTTP, FTP and gopher sessions.

```
void CFtp1Dlg::OnButton1()
{
CInternetSession* pInternetSession;
pInternetSession = new CInternetSession();
if(!pInternetSession){
        AfxMessageBox("Could not establish Internet
session",NULL,MB_OK);
return;
}
pFTPConnection=pInternetSession->
GetFtpConnection(CString("ftp.microsoft.com"));
if(!pFTPConnection){
AfxMessageBox("Could not establish connection",MB_OK);
return;
}
else{
m_text="Downloading...";
UpdateData(false);
}
pFTPConnection->
GetFile(CString("disclaimer.txt"),CString("disclaimer.txt"));
pFTPConnection->Close();
pInternetSession->Close();
}
```

WCTM

CHECK BOXES

1. Open Visual C++ and click the New item in the File menu, opening the New dialog box.
2. Now select the MFC AppWizard(exe) entry in the New dialog box.
3. Give the new program the name Checks in the Project name box.
4. Click OK to starts the Visual C++ AppWizard.
5. Make it a dialog based program in AppWizard
6. Open the dialog box for the main window, ID=IDD_CHECKS_DIALOG, in the dialog editor
7. Add three check boxes on the dialog box and add three text boxes to label the check boxes from the toolbox
8. Now connect the check boxes to code. We connect the check boxes with the ClassWizard. Open the ClassWizard and find three check boxes IDC_CHECK1, IDC_CHECK2 and IDC_CHECK3. Add event handlers to these controls by clicking those id values one by one, and double clicking the BN_CLICKED message in the Message box.
9. This creates event handlers OnCheck1(),OnCheck2() and OnCheck3(). ClassWizard can be used to connect s member variable to the text in the text box, which we name as m_text.
10. when the user clicks checkbox1, we will simply display the string “Check1 clicked”

```
void CChecksDlg::OnCheck1()
{
        // TODO: Add your control notification handler code here
        m_text="Check1 clicked";
        UpdateData(false);
}


void CChecksDlg::OnCheck2()
{
        // TODO: Add your control notification handler code here
        m_text="Check2 clicked";
        UpdateData(false);
}

void CChecksDlg::OnCheck3()
{
        // TODO: Add your control notification handler code here
        m_text="Check3 clicked";
        UpdateData(false);
```

}

RADIO BUTTONS

1. Open Visual C++ and click the New item in the File menu, opening the New dialog box.

2. Now select the MFC AppWizard(exe) entry in the New dialog box.

3. Give the new program the name Radios in the Project name box.

4. Click OK to starts the Visual C++ AppWizard.

5. Make it a dialog based program in AppWizard

6. Open the dialog box for our main window, ID=IDD_RADIOS_DIALOG

7. Place a text box in the main window

8. Open ClassWizard and find the three radio buttons IDC_RADIO1, IDC_RADIO2, IDC_RADIO3, and connect the event handlers to each of these new controls, OnRadio1(), OnRadio2() and OnRadio3(). Finally connect a new member variable m_text to the text in the text box.

```
void CRadiosDlg::OnRadio1()
{
        // TODO: Add your control notification handler code here
        m_text="Radio1 clicked";
        UpdateData(false);
}

void CRadiosDlg::OnRadio2()
{
        // TODO: Add your control notification handler code here
        m_text="Radio2 clicked";
        UpdateData(false);
}

void CRadiosDlg::OnRadio3()
{
        // TODO: Add your control notification handler code here
        m_text="Radio3 clicked";
        UpdateData(false);
```


WCTM

SERIALIZAING (WRITING OR READING DATA ON TO THE DISK)

1. In the Doc.cpp – under the serializable() method-

```
void CAbcDoc::Serailizable(CArchieve& ar)
{
        if(ar IsStoring()
        {
        ar<<StringData;
        }
        else
        {
        ar>>StringData;
        }
}
```

2. To inform the document that the data has changed use
   pDoc->SetModifiedFlag() in OnChar()

Serializing our own class:-

1. Include a C/C++ header file
2. Define the class

```
Class CData
{
private:
        CString data;
public:
        CData()
        {
                data = CString text)
                {data+=text)};
                void DrawText(CDC* pDC)
                        {pDC->TextOut(0,0data)};
                void ClearText(){data=" "};
```

3. in the Doc.h file

```
#include "CData.h"

public:
        CData DataObject;
```

4. OnChar-

```
        pDoc->DataObject.AddText(CString(nChar));
OnDraw-
        pDoc->DataObject.DrawText(pDC);
```

5. Making class serializable
   a) class must be derived from CObject

WCTM

```
CObject
        Class CData:public CObject{
Private:
        CString data;
        DECLARE_SERIAL(CData);
Public:
        Void Serialize(CArchieve & archieve);
```

b)add another file – Cdata.cpp

```
#include "stdafx.h"
#include "serializeDoc.h"

Void CData::Serialize(CArchieve& archieve)
{
CObject::Serialize(archieve);
If(archieve IsStoring())
{
archieve<<data;
}
else
{
archieve>>data;
}
}
IMPLEMENT_SERIAL(CData, CObject,0)
```

c) Doc.cpp serialize method

```
        DataObject.Serialize(ar);
```

WCTM

OUTPUT

MENU DISPLAYING A DIALOG BOX

1.Make a new project name 'menu' from Applicationwizard(exe).select single document option and finish the program.

2. Adding Dialog Box:

3.Go to Resources in the Insert menu bar button and add Dialog Box.

4. Now add control buttons from the tool bar.

5. Two Control Buttons 'edit text' and 'button' buttons are added on the Dialog Box.

6. Right click on the Dialog box and go to classwizard add new class name say 'dialog'.

7. Add the variables to the buttons by clicking the member Variables option. For 'Button' click on BN_CLICKED. On 'Edit button' add a variable name m_text.

8. Adding Menu:

9. Go to Resource Class , add a menu item in the default Menu bar.Name the menu as 'Show Dialog' in the Caption. Remove the popup option. Also name the ID of menu as ID_Showdialog.

10. Now go to classwizard and double click the ID_Showdialog .Go to Edit Code option and add the following steps in the Onbutton method:

11. Add the class of the dialog box named 'dialog'.

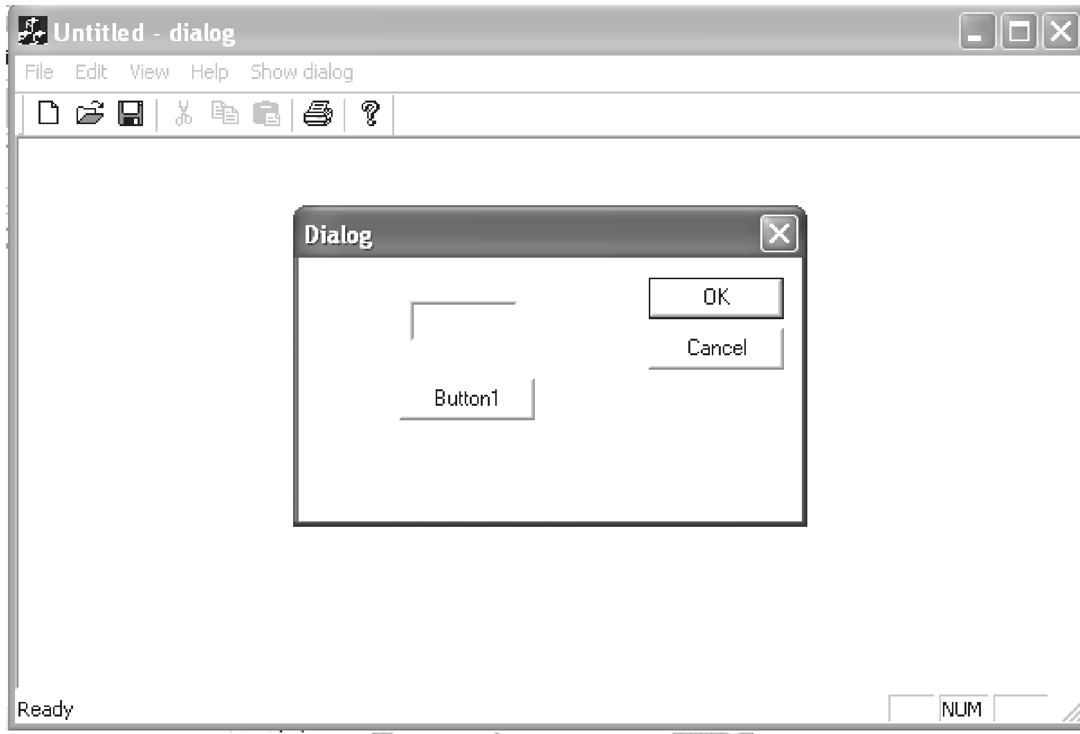
12. Make an object 'obj' of this class in Onbutton method.
    Dialog obj;

13. Now add the coding:
    Obj.DoModal( );

14. Execute and Run the program.

WCTM

OUTPUT



STUDY WINDOW'S API.
THEIR RELATIONSHIP WITH MFC CLASSES

API is an acronym for Application Programming Interface. It is simply a set of functions that are part of Windows OS. Programs can be created by calling the functions present in the API. The Programmer doesn't have to bother about the internal working of the functions. By just knowing the function prototype and return value he can invoke the API Functions.

A good understanding of Windows API would help you to become a good Windows programmer. Windows itself uses the API to perform its amazing GUI magic.The Windows APIs are of two basic varieties :

→ API for 16-bit Windows(Win16 API)
→ API for 32-bit Windows(Win32 API)

Each of these have sub-APIs within it. If you are to program Windows 3.1 then you have to use Win16 API, whereas for programming Windows95 and WindowsNT you have to use Win32 API.

Win16 is a 16-bit API that was created for 16-bit processors, and relies on 16-bit values. Win32 is a 32-bit API created generation of 32-bit CPUs and it relies on 32-bit values.

Win16 API	Win32 API	Description
USER.EXE	USER32.DLL	The USER components is responsible for window management, including messages, menus, cursors, communications, timer etc.
GDI.EXE	GDI32.DLL	The GDI management is the Graphics Device Interface; it takes care of the user interface And graphics drawing, including Windows metafiles , bitmaps, device contexts, and fonts.
KRNL386.EXE	KERNEL32.DLL	The KERNEL component handles the low level functions of memory, task, and resource Management that are the heart of Windows.

The Win16 and Win32 APIs ere similar in most respects, but the Win16 API can be considered as a subset of Win32 API.Win32 API contains almost everything that the Win16 API has, and much more.

At its core each relies on three main components to provide most of the functionality of Windows. These core components along with their purpose are shown in the table given above.

Although the Win16 versions of these componets have .EXE extensions, they are actually all DLLs and cannot execute on their own.

The Win32 API has many advantages, some obvious and others that are not so

WCTM

obvious. The following lists major advantages that applications have when developed with the Win32 API and a 32-bit compiler (such as Microsoft's Visual C++) :

• **True multithreaded applications**.
Win32 applications support true preemptive multitasking when running on Windows 95 and Windows NT.

• **32-bit linear memory.**
Applications no longer have limits imposed by segmented memory. All memory pointers are based on the applications virtual address and are represented as a 32-bit integer.

• **No memory model.**
The memory models (small, medium, large, etc.) have no meaning in the 32-bit environment. This means there is no need for near and far pointers, as all pointers can be thought of as far.

• **Faster .**
A well-designed Win32 application is generally faster. Win32 applications execute more efficiently than 16-bit applications.

• **Common API for all platforms**. The Win32 API is supported on Windows 95, Windows NT on all supported hardware, Windows CE, and the Apple Macintosh.

MFC (Microsoft Foundation's Class)

The C++ class library that Microsoft provides with its C++ compiler to assist programmers in creating Windows-based applications. MFC hides the fundamental Windows API in class hierarchies, so that, programmers can write Windows-based applications without needing to know the details of the native Windows API. The MFC classes are coded in C++. It provides the necessary code for managing windows, menus and dialog-boxes. It also helps in carrying out basic tasks like performing basic input-output, storing the collection of data objects etc. It provides the basic framework for the application on which the programmer can build the rest of the customized code. MFC Library is a collection of classes in hierarchical form, where classes are derived from some base class. For most of the classes provided by MFC, base class is CObject
from which most of the classes are inherited. The rest few classes are independent classes. The classes in MFC Library can be categorized as:

- Root Class: CObject
- MFC Application Architecture Classes
- Window, Dialog, and Control Classes
- Drawing and Printing Classes
- Simple Data Type Classes
- Array, List, and Map Classes
- File and Database Classes
- Internet and Networking Classes
- OLE Classes
- Debugging and Exception Classes

WCTM

In addition to these classes, the Microsoft Foundation Class Library contains a number of global functions, global variables, and macros. The Microsoft Foundation Class Library (MFC) supplies full source code. The full source
code is supplied in the form of Header files (.H) that are in the MFC\Include directory and implementation files (.Cpp) that are in the MFC\Src directory
MFC Library was especially designed to provide an object oriented interface to Windows, simultaneously providing the compatibility with the windows programs written in C language. The compatibility with C language was required as the Windows was developed long before MFC came into existence and hence all the applications for Windows were initially written in C, a large number of which is still in use.
MFC Library provides the following features:

- Significant reduction in the effort to write an application for Windows.
- Execution speed comparable to that of the C-language API.
- Minimum code size overhead.
- Ability to call any Windows C function directly.
- Easier conversion of existing C applications to C++.
- Ability to leverage from the existing base of C-language Windows programming experience.
- Easier use of the Windows API with C++ than with C.
- Easier-to-use yet powerful abstractions of complicated features such as ActiveX, database support, printing, toolbars, and status bars.
- True Windows API for C++ that effectively uses C++ language features.

Advantages of MFC

Microsoft Foundation Class Library greatly reduces the development time, makes the code more portable, provides support without reducing the programming freedom and flexibility and provides easy access to user interface elements and technologies like ActiveX and Internet programming which are otherwise very hard to program . By using MFC, developers can add many capabilities to their applications in an easy, object-oriented manner. MFC simplifies database programming by providing Data. Access Objects (DAO) and Open Database Connectivity (ODBC) and network programming through Windows Sockets.
MFC offers many advantages like:

- Application Framework
- Largest base of reusable C++ source code
- Integration with Visual C++
- Flexible and fast database access
- Support for Internet and ActiveX technology
- Support for messaging API
- Support for multithreading

MFC is not only library of classes. MFC is also an application framework.MFC helps to define the structure of an application and handles many routine chores on the application's behalf.

Starting with CWinApp, the class that represents that represents the application itself, MFC encapsulates virtually every aspect of a program's operation. The framework supplies the WinMain() function, and WinMain() in turn calls the application object's member functions to make the program go. One of the CWinApp member functions called by WinMain()-Run() – encapsulates the message loop that literally runs the program.

STUDY ESSENTIAL CLASSES IN DOCUMENT VIEW ARCHITECTURE THEIR RELATIONSHIP WITH EACH OTHER

Parts of Application

The application source code generated for you by AppWizard actually consists of 4 major classes. All these classes can be seen by expanding the class listing in the Class View tab. For each of these classes corresponding header file (.h) and an implementation file (.cpp) is generated.

The MainFrame Window Class

This class is named CMainFrame and is derived from the MFC class CFrameWnd. This class manages the main window of the application that contains the window frame, Menu bar, Toolbar, Status bar, System menu and Minimise, Maximise and Close boxes and also contains the view window of the application.

The Application Class

This class named C ProjectNameApp is derived from the MFC class CWinApp. It manages the application as a whole by managing the tasks, like initialising the application instance, managing the message loop and performing final cleanup of the program.

The Document Class

This class named C ProjectNameDoc is derived from the MFC class CDocument. It is responsible for storing the program data as a single string and reading and writing this data to disk files.

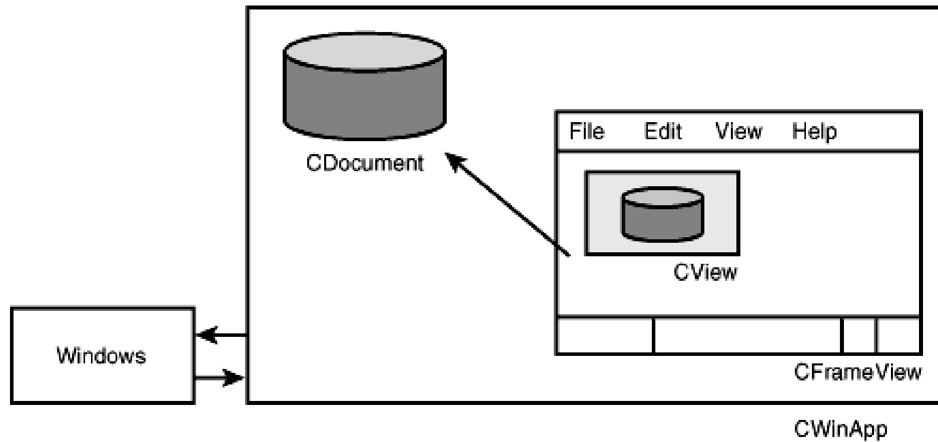
The View Class

The class named C projectNameView is derived from CView class from the MFC classes.It manages the view window, which is used to display program data on the screen and also used in processing the input from the user.

MFC and AppWizard use the Document/View architecture to organize programs written for Windows. Document/View separates the program into four main classes:

- A document class derived from CDocument
- A view class derived from CView
- A frame class derived from CFrameWnd
- An application class derived from CWinApp

Each of these classes has a specific role to play in an MFC Document/View application. The document class is responsible for the program's data. The view class handles interaction between the document and the user. The frame class contains the view and other user interface elements, such as the menu and toolbars. The application class is responsible for actually starting the program and handling some general-purpose interaction with Windows. Figure shows the four main parts of a Document/View



The Document/View architecture.

Although the name "Document/View" might seem to limit you to only word-processing applications, the architecture can be used in a wide variety of program types. There is no limitation as to the data managed by CDocument; it can be a word processing file, a spreadsheet, or a server at the other end of a network connection providing information to your program. Likewise, there are many types of views. A view can be a simple window, as used in the simple SDI applications presented so far, or it can be derived from CFormView, with all the capabilities of a dialog box. You will learn about form views in Section 23, "Advanced Views."

SDI and MDI Applications

There are two basic types of Document/View programs:

- SDI, or Single Document Interface
- MDI, or Multiple Document Interface

An SDI program supports a single type of document and almost always supports only a single view. Only one document can be open at a time. An SDI application focuses on a particular task and usually is fairly straightforward.

Several different types of documents can be used in an MDI program, with each document having one or more views. Several documents can be open at a time, and the open document often uses a customized toolbar and menus that fit the needs of that particular document.

Why Use Document/View?

The first reason to use Document/View is that it provides a large amount of application code for free. You should always try to write as little new source code as possible, and that means using MFC classes and letting AppWizard and ClassWizard do a lot of the work for you. A

large amount of the code that is written for you in the form of MFC classes and AppWizard code uses the Document/View architecture.

The Document/View architecture defines several main categories for classes used in a Windows program. Document/View provides a flexible framework that you can use to create almost any type of Windows program. One of the big advantages of the Document/View architecture is that it divides the work in a Windows program into well-defined categories. Most classes fall into one of the four main class categories:

- Controls and other user-interface elements related to a specific view
- Data and data-handling classes, which belong to a document
- Work that involves handling the toolbar, status bar, and menus, usually belonging to the frame class
- Interaction between the application and Windows occurring in the class derived from CWinApp

Dividing work done by your program helps you manage the design of your program more effectively. Extending programs that use the Document/View architecture is fairly simple because the four main Document/View classes communicate with each other through well-defined interfaces. For example, to change an SDI program to an MDI program, you must write little new code. Changing the user interface for a Document/View program impacts only the view class or classes; no changes are needed for the document, frame, or application classes.

Documents

Documents are the basic elements that are created and manipulated by the application. Windows provides a graphic interface that gives a user a natural way to use the application. To implement this interface, the developer has to provide ways to see and interact with the information that the application creates and uses. A document is simply a place to collect common data elements that form the processing unit for the application.

Documents and the Application Relationship

The application class uses documents as a way to organize and present information to the user. Each application derived from the MFC defines at least one type of document that is a part of the application. The type of document and the number of documents that the application uses are defined in the code of the application class. As we have already discussed, MFC supports two types of applications – MDI and SDI. MDI - In applications that support Multiple Document Interface, a number of text files can be opened for editing at once; each in a different window. Each of the open files has a corresponding document. Also, in MDI, the same document can have multiple views,
where a window is split. Each pane of the window can show a different portion of the data whereas this data is coming from a common source, the document.
SDI - In applications with Single Document Interface only one document is open at a time. A SDI application does not have a Window menu and the File menu does not have a Close option because only one document can be open at a time, opening a document automatically closes the current document. These are two common characteristics of a SDI application.

WCTM

Why Documents
The fundamental responsibility of the document is to store all the elements that constitute an application unit. An application may support more than one type of data like numbers, text, drawings etc. The document also controls all the views associated with it. As the user opens and manipulates windows on the screen, views are created and the document is associated with each view. The document is responsible for controlling and updating the elements within a given view. The view may either request the document to draw its components or directly request the components to draw themselves. It must provide a device context where the drawing occurs. All elements of a drawing must be able to display themselves correctly upon request.

Views
The view is the user's window to the document. The user interacts with a document usingthe view. Each active document will have one or more active views available on the display. Generally, each view is displayed in a single window.

Why Views
The view gives the document a place to display information. It is the intermediary between the document, which contains the information and the user. The view organises and displays the document information onto the screen or printer and takes in the user input as information or operation on the document. The view is the active area of the document. It has two functions –

→ Acts as the display area for the document data and
→ Acts as the input area where the user interacts with the document, normally providing additional commands or data for the document to process.

All the information passes through the view before reaching the document. Therefore, if an application handles mouse messages in functions like OnLButtonDown(), it first receives the message, translates the information into an appropriate form and then calls the required document function to process the mouse command. Although view is responsible for displaying the document and its information, there are two possibilities –

→ Let the view directly access the document's data elements
→ Create a document function that handles access to the appropriate data Member.

The choice of writing directly in the view has two advantages –

→It is the responsibility of the view to handle document display, so having the code there is more appropriate
→ Placing the code in the view keeps all the code regarding the device context in the view where it seems natural.

A view is always tied to a single document. One document can have several views. Opening another window on the document can create two views of the same document. Alternative ways can be presented to look at the same information by implementing different types of views for the document. For example, as we discussed in the beginning of the chapter, in MS Excel same data can be viewed either in the form of a table or in the form of different graphs.

Gaining Access to Document Data from the View
The view accesses its document's data either with the GetDocument() function, which returns a pointer to the document or by making the view class a C++ friend of the document class. The view then uses its access to the data to obtain the data when it is ready to draw or

otherwise manipulate it. For example, from the view's OnDraw() member function, the view uses GetDocument()to obtain a document pointer. Then it uses that pointer to access a CString data member in the document. The view passes the string to the TextOut() function.

WCTM