



Sponsored by

INTERFACE
TECHNOLOGIES, INC.

The original article for this PDF can be found on DevCentral.
<http://devcentral.iticentral.com>

Creating a Simple MFC Program

by Alan Oursland



[Comment on this article](#)



Introduction

MFC was created to make programming in Windows easier. As an object-oriented wrapper for Win32, it automates many routine programming tasks (mostly passing references around). Paradigms like the document/view architecture were added to automate even more tasks for the programmer, but in the process, some control was taken away.

This tutorial will show you how to create a very simple MFC program and how to customize it for your application. This program will be a base for other tutorials that I write for OpenGL and DirectX. I will be going through this tutorial in Visual C++ 6.0.

- [\[Download simple_MFC_basic.zip \]](#)
- [\[Download simple_MFC_extended.zip \]](#)

We will begin by creating a standard MFC Doc/View application using AppWizard. Select New from the File menu, click on the Projects tab, and select MFC AppWizard (exe). Enter SimpleMFC as the project name and press OK.

1. Select Single Document and press Next.
2. Select None for database support and press Next.
3. Select None for compound document support and remove support for ActiveX controls. Press Next.
4. Deselect all of the check boxes in Step 4. Press Next.
5. Generate comments and use a shared DLL. Press Next.
6. Press Finish.

We now have a standard Doc/View application with the following classes:

```
CAboutDlg
CMainFrame
CSimpleMFCApp
CSimpleMFCDoc
CSimpleMFCView
```

Go to the file view and remove the following files from the project and delete them from the directory:

```
SimpleMFCView.h
SimpleMFCView.cpp
SimpleMFCDoc.h
SimpleMFCDoc.cpp
```

You can also remove IDR_SIMPLETYPE from the icon resource tab and delete the associated file SimpleMFCDoc.ico (be sure to do it in that order).

We now have a project that doesn't compile.

Edit SimpleMFC.cpp:

- Remove the #includes for SimpleMFCView.h and SimpleMFCDoc.h.
- Edit the constructor to look like this:

```
CSimpleMFCApp::CSimpleMFCApp()
{
    m_pMainWnd = NULL;
    // Place all significant
    // initialization in InitInstance
}
```

- Edit InitInstance to look like this:

```
BOOL CSimpleMFCApp::InitInstance()
{
    // Standard initialization
    // If you are not using these features
    // and wish to reduce the size
    // of your final executable, you should
    // remove from the following
    // the specific initialization routines
    // you do not need.

#ifdef _AFXDLL
    Enable3dControls();
    // Call this when using MFC in a shared DLL
#else
    Enable3dControlsStatic();
    // Call this when linking to MFC statically
#endif

    // The one and only window has been initialized,
    // so show and update it.
    m_pMainWnd = new CMainFrame();
    m_pMainWnd->ShowWindow(SW_SHOW);
    m_pMainWnd->UpdateWindow();

    return TRUE;
}
```

- Remove everything below InitInstance. This includes CAboutDlg and the code to bring up the About dialog. (If you would like to leave the About dialog in, feel free to do so. You will be able to activate it later.)
- Remove the items in the message map at the top of the file. This section should look like this:

```
BEGIN_MESSAGE_MAP(CSimpleMFCApp, CWinApp)
//{{AFX_MSG_MAP(CSimpleMFCApp)
// NOTE - the ClassWizard will add
// and remove mapping macros here.
// DO NOT EDIT what you see in these
// blocks of generated code!
//}}AFX_MSG_MAP
// Standard file based document commands
END_MESSAGE_MAP()
```

Remove the definition for OnAppAbout from SimpleMFC.h.

We still have a program that doesn't compile. CMainFrame is expecting to be created dynamically within the Doc/View architecture, thus its constructor is protected. We will fix that.

Edit MainFrm.h:

- Remove "DECLARE_DYNCREATE(CMainFrame)"

- Change "protected: // create from serialization only" to "public:"

Edit MainFrm.cpp:

- Remove "IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)".
- Edit the constructor to look like this:

```
CMainFrame::CMainFrame()
{
    RECT r;
    r.left = 100;
    r.top = 100;
    r.right = 200;
    r.bottom = 200;

    Create(NULL, "Simple MFC App",
        WS_POPUP|WS_THICKFRAME,
        r, NULL, NULL, 0, NULL );
}
```

If you run the program, a window with a border will appear. It will not have a title or menu bar. This is about the simplest Windows application you can make. Press Alt-F4 and Windows will send a WM_CLOSE message to the application to close it.

Let's take a look at what is going on here. CSimpleMFCApp is your application derived from CWinApp. CWinApp contains a pointer to the main application window. We have a global instance of CSimpleMFCApp. When it is created it instantiates an instance of our CMainFrame and assigns it as the CWinApp main window. It then displays the window. This is all that the application has to do. We will not be modifying it further.

CMainFrame creates the actual internal system window. Remember that MFC is just a wrapper for Win32. CMainframe, which is derived from CFrameWnd, is not actually an MS-Windows window. It just manages the handle to a window that MS-Windows creates. (The fact that the product is named Windows and the objects themselves are called "windows" may make this confusing.)

In the Doc/View architecture, the window handle is created "dynamically" (which basically means the base class creates it in a predefined way). Since we are not using the Doc/View architecture, and we want more control over the window, we create the window handle ourselves by calling CFrameWnd::Create in the CFrameWnd constructor. CFrameWnd::Create eventually calls the Win32 create, but it manages the window handle for us. Create takes several parameters which are the key to customizing this window.

```
BOOL Create( LPCTSTR lpszClassName,
            LPCTSTR lpszWindowName,
            DWORD dwStyle = WS_OVERLAPPEDWINDOW,
            const RECT& rect = rectDefault,
            CWnd* pParentWnd = NULL,
            LPCTSTR lpszMenuName = NULL,
            DWORD dwExStyle = 0,
            CCreateContext* pContext = NULL );
```

- lpszClassName - This is a string that describes the "window class" that you want to use. The window class tells Windows if this window will be a frame, or a button, or a scrollbar, or whatever. You can also define your own classes. This value can not be changed once the window has been created. We pass in NULL to use the default window class for a CFrameWnd.
- lpszWindowName - This is the name of the window. If the window has a title bar, it will display this string.
- dwStyle - This is probably the most important parameter in defining how your window will look. But don't worry, you can always change the styles once the window is created. We'll play with some different styles later on. The styles we use create a basic top level (popup) window with a resizable border.
- rect - This defines the size and position of the window on the screen.
- pParentWnd - This is the parent of the window you are creating. Since we are creating a top level window, we have no parent.
- lpszMenuName - This tells the application which resource to use for the menu. If you have a menu resource defined you can pass in "#128", where 128 is the resource id of your menu. (128 should be the resource id of the menu in this particular example.)
- dwExStyle - These are extended styles and are very useful in further controlling the appearance of your window. You can set these at any time after the window has been created.
- pContext - This is used in the Doc/View architecture. Just pass in NULL for this parameter.

That's about it. Once the window is created, it uses a function inside MFC for the message handling, so you don't need to do anything there. In the next section, we'll play around with this window, so you can customize it to suit your needs.

[Previous Page](#)

[Return to beginning of article](#)

[Next Page](#)

© 2001 [Interface Technologies, Inc.](#) All Rights Reserved
Questions or Comments? devcentral@itcentral.com
[PRIVACY POLICY](#)



The original article for this PDF can be found on DevCentral.
<http://devcentral.iticentral.com>

Creating a Simple MFC Program

by Alan Oursland

Creating a New Project

We want to create a new project based on SimpleMFC. We can either repeat the process described above (not my favorite - I won't remember all of those steps), or we can copy the existing one. Copy the entire directory for SimpleMFC into its parent directory. You should get a new directory with the name "Copy of SimpleMFC". Change the name of this directory to "SimpleExtMFC". Delete any object files and executables you may have copied (or just delete the "Debug" and "Release" directories if they exist).

Change the names of all of the files containing SimpleMFC to SimpleExtMFC. The easiest way to do this is in a DOS prompt. Set the prompt to the new directory and type "rename SimpleMFC*. * SimpleExtMFC*. *". Do the same in the "res" directory.

Do not open the project file yet. In Developer Studio, do an "Edit-Find In Files" for SimpleMFC. Be sure to set the search location to the new directory. Search all file types (*. *). Do not match whole word only. You will find matches in the following files:

- MainFrm.cpp
- ReadMe.txt
- SimpleExtMFC.clw
- SimpleExtMFC.cpp
- SimpleExtMFC.dsp
- SimpleExtMFC.dsw
- SimpleExtMFC.h
- SimpleExtMFC.plg
- SimpleExtMFC.rc
- StdAfx.cpp
- SimpleExtMFC.rc2

Replace all instances of SimpleMFC with SimpleExtMFC in these files. (You can actually delete SimpleExtMFC.clw. This is the classwizard file, and will be rebuilt if you delete it.) Open each file and use the Replace menu to replace all instances. (Hint: if you press F3, the first occurrence of SimpleMFC will become highlighted, which makes using the Find and Replace dialog much easier.) Save all of the files and open your brand new project.

[Previous Page](#)

[Return to beginning of article](#)

[Next Page](#)

© 2001 [Interface Technologies, Inc.](#) All Rights Reserved
Questions or Comments? devcentral@iticentral.com
[PRIVACY POLICY](#)



Sponsored by



INTERFACE
TECHNOLOGIES, INC.

The original article for this PDF can be found on DevCentral.
<http://devcentral.iticentral.com>

Creating a Simple MFC Program

by Alan Oursland

Adding a Popup Menu

One of the most useful controls in modern applications is the popup menu on right-clicking the mouse. We are going to add a popup menu to our application.

First, we need to create a menu resource. Select Insert-Resource from the menu, select Menu, and press New. Right click on the IDR_MENU1 label and select Properties (see how useful that was). Change the name to IDR_POPUPMENU.

You should also have an empty menu available for editing. Normal menu bars lay out their options horizontally, left to right. Popup menus lay out their components vertically, top to bottom. We need to create a menu pane for our popup menu. In the first menu cell type the label "Popup Top" and press Enter. Select "Popup Top" and our menu pane will appear below it. Type "&Close" into the pane. Enter "ID_POPUP_CLOSE" as the ID. Save your resource definitions. We have just created the resource definition for our popup menu. Now we have to make our application open it.

Open ClassWizard (Ctrl-W or View-ClassWizard). Rebuild the database if asked. If a dialog pops up that asks you to associate a class with the new menu resource, you can associate the CMainFrame class with it. Verify that CMainFrame is selected as the class. We want to display the popup menu on a right mouse button up event, so locate WM_RBUTTONDOWN in the Messages list. Double click on it to add a message handler. Double click on the new method to edit it. Edit OnRButtonDown to look like this:

```
void CMainFrame::OnRButtonDown(UINT nFlags, CPoint point)
{
    CMenu popupMenu;
    popupMenu.LoadMenu(IDR_POPUPMENU);
    CMenu* subMenu = popupMenu.GetSubMenu(0);
    ClientToScreen(&point);
    subMenu->TrackPopupMenu(0, point.x, point.y,
        AfxGetMainWnd(), NULL);

    CFrameWnd::OnRButtonDown(nFlags, point);
}
```

LoadMenu loads the menu resource. GetSubMenu gets the particular popup menu we are using to display the Close menu item. ClientToScreen translates the cursor's screen position to the window position. Finally, TrackPopupMenu actually opens the menu. Simple, right? Go ahead and run the program. You'll notice that our option is greyed out in the popup menu. We still need to hook it up to do something. Close the program and go back into ClassWizard.

Verify that CMainFrame is still selected as your class. Under Object IDs, select ID_POPUP_CLOSE. Double click on COMMAND under Messages. This will create the member function OnPopupClose (don't change the name). Edit OnPopupClose to look like this:

```
void CMainFrame::OnPopupClose()
{
    PostMessage(WM_CLOSE);
}
```

Now, when the Close option is selected, we will send a WM_CLOSE message to the application ourselves. Compile and run the program. The popup menu should be fully functional.

[Previous Page](#)

[Return to beginning of article](#)

[Next Page](#)

© 2001 [Interface Technologies, Inc.](#) All Rights Reserved

Questions or Comments? devcentral@iticentral.com

[PRIVACY POLICY](#)



The original article for this PDF can be found on DevCentral.
<http://devcentral.iticentral.com>

Creating a Simple MFC Program

by Alan Oursland

Moving the Window

Windows generally lets you move a window by dragging it around by the title bar. We don't have a title bar, so we need another way to move the window. We could just add a "Move" item to our popup menu and post a WM_ENTERSIZEMOVE event in the same way we posted WM_CLOSE, but I think I'd like the user to be able to drag the window around the screen.

The first part is easy. Add another menu item called "Move" to our popup menu under the "Close" menu item. Create a function to handle it called OnPopupMove. Edit OnPopupMove to look like this:

```
void CMainFrame::OnPopupMove()
{
    PostMessage(WM_SYSCOMMAND, SC_MOVE);
}
```

You can use WM_SYSCOMMAND to trigger events from the system menu. In fact, why don't we change OnPopupClose to use the WM_SYSCOMMAND as well. Change "PostMessage(WM_CLOSE)" to "PostMessage(WM_SYSCOMMAND, SC_CLOSE)". There isn't any change in functionality, but it helps keep things consistent.

Notice that when we select Move, initially we can move the window with the arrow keys, but not with the mouse. The same thing would happen if we tried to post an SC_MOVE event on mouse down. Windows doesn't provide us access to the code to move things, so we will have to write it ourselves. We want to start moving on a left mouse button down, move on a mouse move, and stop moving on a left mouse button up. First, add functions to handle ON_WM_LBUTTONDOWN, ON_WM_MOUSEMOVE, and ON_WM_LBUTTONUP.

To move the window, we first need to know when the mouse was pressed inside of it. Add the following member variable to CMainFrame:

```
BOOL m_bMouseDown;
```

Initialize it to FALSE in the constructor, set it to TRUE in OnLButtonDown, and set it to FALSE in OnLButtonUp. Add an if clause on it in OnMouseMove. This isn't good enough to tell when the mouse is pressed. It is possible for the button to be pressed while the cursor is inside of the window and released while the cursor is outside of the window. In that case we would never receive the mouse release and would continue moving the window. We need to capture the mouse so that we always get mouse events - even if the mouse is not in the window.

Add "SetCapture()" to OnLButtonDown and "ReleaseCapture()" to OnLButtonUp. During OnMouseMove, we want to move the window. We will need to know where in the window the mouse was pressed, and move the window by the distance between it and the current position. Add The following member variable to CMainFrame:

```
CPoint m_mouseDownPoint;
```

Set the variable to the point parameter in OnLButtonDown. In OnMouseMove, check that the mouse is down. Then find the size difference between the "mouse down point" and the new point. Get the window rectangle. Subtract the rectangle from the size and move the window to the new position.

That's it. We now have a window that we can drag around with the mouse or move via the menu using the arrow keys. The final code for OnLButtonDown, OnMouseMove, and OnLButtonUp follows:

```
void CMainFrame::OnLButtonDown(UINT nFlags, CPoint point)
{
    m_bMouseDown = TRUE;
    SetCapture();
    m_mouseDownPoint = point;

    CFrameWnd::OnLButtonDown(nFlags, point);
}

void CMainFrame::OnMouseMove(UINT nFlags, CPoint point)
{
    if( m_bMouseDown )
    {
        CSize s = m_mouseDownPoint - point;
        CRect r;
        GetWindowRect(&r);
        r = s - &r;
        MoveWindow(r);
    }

    CFrameWnd::OnMouseMove(nFlags, point);
}

void CMainFrame::OnLButtonUp(UINT nFlags, CPoint point)
{
    m_bMouseDown = FALSE;
    ReleaseCapture();

    CFrameWnd::OnLButtonUp(nFlags, point);
}
```

[Previous Page](#)

[Return to beginning of article](#)

[Next Page](#)

© 2001 [Interface Technologies, Inc.](#) All Rights Reserved
Questions or Comments? devcentral@itcentral.com
[PRIVACY POLICY](#)



Sponsored by



INTERFACE
TECHNOLOGIES, INC.

The original article for this PDF can be found on DevCentral.
<http://devcentral.iticentral.com>

Creating a Simple MFC Program

by Alan Oursland

More Menu Fun

We've added a few of the items from the system menu. Let's go ahead and add the rest: Restore, Size, Minimize, and Maximize. First, add menu options for the four items. Second, in the ClassWizard, add member functions to handle each option. Third, post the appropriate WM_SYSCOMMAND event for each function (here's a hint: SC_RESTORE, SC_SIZE, SC_MINIMIZE, SC_MAXIMIZE). Run the program. Try out the menu items.

Notice that all of the options are always enabled. When you are maximized, you can still select maximize even though it doesn't do anything. We want to disable some of the menu items. Use the UPDATE_COMMAND_UI message map to do this. Create an UPDATE_COMMAND_UI handler for Move, Maximize, Minimize, Restore and Size. These functions pass in a CCmdUI object. Call Enable() to enable or disable the menu option for that function (you can also use CCmdUI to set checks and text).

We want to disable Move and Size when the application is not maximized:

```
pCmdUI->Enable(!IsIconic()&&!IsZoomed());
```

We want to disable Maximize when the application is maximized:

```
pCmdUI->Enable(!IsZoomed());
```

We want to disable Minimize when the application is minimized:

```
pCmdUI->Enable(!IsIconic());
```

We want to disable Restore when the application is not maximized or minimized:

```
pCmdUI->Enable(IsIconic()||IsZoomed());
```

Now when we use the menu, the options should be enabled appropriately.

We have one more thing to do. Notice that you can still move the window when the application is maximized. We want to disable this. Modify OnMouseMove to look like the following:

```
void CMainFrame::OnMouseMove(UINT nFlags, CPoint point)
{
    if( ( m_bMouseDown )&&( !IsZoomed() ) )
    {
        CSize s = m_mouseDownPoint - point;
        CRect r;
        GetWindowRect(&r);
        r = s - &r;
        MoveWindow(r);
    }
}
```

```
CFrameWnd::OnMouseMove(nFlags, point);  
}
```

This will prevent us from moving the window manually while it is maximized. Following is the code added for this section:

```
void CMainFrame::OnPopuptopRestore()  
{  
    PostMessage(WM_SYSCOMMAND, SC_RESTORE);  
}  
  
void CMainFrame::OnPopuptopSize()  
{  
    PostMessage(WM_SYSCOMMAND, SC_SIZE);  
}  
  
void CMainFrame::OnPopuptopMinimize()  
{  
    PostMessage(WM_SYSCOMMAND, SC_MINIMIZE);  
}  
  
void CMainFrame::OnPopuptopMaximize()  
{  
    PostMessage(WM_SYSCOMMAND, SC_MAXIMIZE);  
}  
  
void CMainFrame::OnUpdatePopupMove(CCmdUI* pCmdUI)  
{  
    pCmdUI->Enable(!IsIconic() && !IsZoomed());  
}  
  
void CMainFrame::OnUpdatePopuptopMaximize(CCmdUI* pCmdUI)  
{  
    pCmdUI->Enable(!IsZoomed());  
}  
  
void CMainFrame::OnUpdatePopuptopMinimize(CCmdUI* pCmdUI)  
{  
    pCmdUI->Enable(!IsIconic());  
}  
  
void CMainFrame::OnUpdatePopuptopRestore(CCmdUI* pCmdUI)  
{  
    pCmdUI->Enable(IsIconic() || IsZoomed());  
}  
  
void CMainFrame::OnUpdatePopuptopSize(CCmdUI* pCmdUI)  
{  
    pCmdUI->Enable(!IsIconic() && !IsZoomed());  
}
```

[Previous Page](#)

[Return to beginning of article](#)

[Next Page](#)

Creating a Simple MFC Program

by Alan Oursland

Changing Your Style

The last part of this introduction to a simple MFC application deals with changing the style of your window.

When the window is maximized, I'd like to get rid of the borders (after all, if I'm dealing with graphics I want all the screen space I can get). We will remove the `WS_THICKFRAME` style from the window on `WM_MAXIMIZE` and add it back on `WM_RESTORE`.

Add `"ModifyStyle(WS_THICKFRAME, 0, 0)"` to `OnPopuptopMaximize` and `"ModifyStyle(0, WS_THICKFRAME, 0)"` to `OnPopuptopRestore`. The thick borders will be removed when you maximize the window.

There is still some border left. This is the 3D look that Windows provides. I want to remove this as well. The 3D look is controlled by the Extended Style `WS_EX_CLIENTEDGE`. Add `"ModifyStyleEx (WS_EX_CLIENTEDGE, 0, 0)"` to `OnPopuptopMaximize` and `"ModifyStyleEx (0, WS_EX_CLIENTEDGE, 0)"` to `OnPopuptopRestore`.

Just for fun, let's add a real System Menu to our window. This will let us right-click on the icon in the task bar. Add `"ModifyStyle(0, WS_SYSMENU, 0)"` to the constructor after `Create` (we could just add it into `Create`, but what fun would that be?).

Using `ModifyStyle` and `ModifyStyleEx` you can change the appearance of you window in all sorts of ways - add/remove menus, scrollbars, make the window stay on top, give it a title bar, among other things. Play around with it. Note that you will have to repaint your window after the style changes. The maximize and restore do it for us here.

Conclusion

I hope that at this point you find yourself a little more familiar with how windows work in Windows. I've provided the source code with the article. Until next time, have fun.

Developed Under:

Visual C++ 6.0 Service Pack 5

[Previous Page](#)

[Return to beginning of article](#)

[Next Page](#)

© 2001 [Interface Technologies, Inc.](#) All Rights Reserved

Questions or Comments? devcentral@iticentral.com

[PRIVACY POLICY](#)