子类化，通俗来讲就是用自己的窗口处理函数来处理特定消息，并将自己其他消息还给标准（默认）窗口处理函数。在SDK中，通过SetWindowLong来指定一个自定义窗口处理函数：SetWindowLong(hwnd, GWL\_WNDPROC, (LONG)UserWndProc);。可是到了MFC中，大部分基础的东西都被封装起来了，那么，这是该怎么实现子类化呢？  
       先来看一个例子：  
       要求：定义一个Edit控件，让它能够对输入进行特定的处理输入进行处理-----只能输入英文字母，对其他输入作出提示。  
       分析：1）处理输入当然是响应WM\_CHAR消息了，然后对输入字符进行判断，并做相应处理。那么，我们有怎么才能让Edit自己处理输入呢？  
                  2）我们知道Windows为我们设计Edit控件时，已经将常用操作通过成员函数的形式封装在CEdit类中了，直接由CEdit生成的对象自己并不能改变原有方法或是定制自己的方法（除了虚函数，但有时我们想实现的并不是虚函数啊！），那么现在想达到这些情况应该怎么办呢？这就用到本篇文章的主题-----MFC子类化。  
                  3）我们可以从CEdit类派生一个新类CSuperEdit，然后通过子类化方法是Edit窗口来使用我们指定的消息处理函数。  
        实现：先CSuperEdit，并为其添加WM\_CHAR消息响应函数，这样CSuperEdit对象就拥有了自己WM\_CHAR响应函数（这正是子类化的效果所在----面向对象----自己的方法封装在自己的类中），然后在其父窗口类（这里我们用一个基于对话框的MFC程序）中声明一个CSuperEdit类对象m\_edit，当然m\_edit需要和一个实际存在的窗口关联起来，因此，在CXXXDialog::OnInitDialogj中添加：m\_edit.SubclassDlgItem(IDC\_EDIT1,this);这样  
就将m\_edit这个c++对象和IDC\_EDIT1窗口关联起来了，然后我们只需要在CSuperEdit::OnChar()中添加相应的操作就OK了。

**原理探讨**

    追溯的目标：在整个程序中的哪个位置改变了m\_edit关联窗口的消息处理函数。

        首先，来探讨一下m\_edit和窗口关联实现：m\_edit.SetclassDlgItem(IDC\_EDIT1,this); 我们进入该函数中看看：

BOOL CWnd::SubclassDlgItem(UINT nID, CWnd\* pParent)

{

     ASSERT(pParent != NULL);

     ASSERT(::IsWindow(pParent->m\_hWnd));

     // check for normal dialog control first

     HWND hWndControl = ::GetDlgItem(pParent->m\_hWnd, nID);

     if (hWndControl != NULL)

**return SubclassWindow(hWndControl);**

// 省略无关代码

       … …

     return FALSE;   // control not found

}

查看MSDN：

CWnd：：SubclassDlgItem

This method dynamically subclasses a control created from a dialog box template, and attach it to this **CWnd** object. When a control is dynamically subclassed, windows messages will route through the **CWnd** message map and call message handlers in the **CWnd** class first. Messages that are passed to the base class will be passed to the default message handler in the control.

This method attaches the Windows control to a **CWnd** object and replaces the **WndProc** and **AfxWndProc** functions of the control. The function stores the old **WndProc** in the location returned by the **CWnd::GetSuperWndProcAddr**method.

翻译：

该方法动态子类化一个从对话框模板创建的控件，然后将它与一个CWnd对象（记为A）关联。当一个控件被动态子类化后，Windows消息将会根据CWnd消息地图路由并首先响应CWnd对象A的消息响应函数。被路由到基类的消息将会被该控件的默认消息处理函数处理。

该方法将一个Windows控件和一个CWnd对象相关联，并替换了这个控件原来的WndProc和AfxWndProc函数。这个函数储存了原先的WndProc的地址，该地址由CWnd：：GetSuperWndProcAddr返回。

好！那么，该函数是怎样替换掉这个控件的原先WndProc和AfxWndProc函数的呢？在SubclassDlgItem函数中我们发现它返回的是SubclassWindow(hWndControl)这个函数的执行结果。

继续查看MSDN：

This method dynamically subclasses a window and attach it to this **CWnd** object. When a window is dynamically subclassed, windows messages will route through the **CWnd** message map and call message handlers in the**CWnd** class first. Messages that are passed to the base class will be passed to the default message handler in the window.

This method attaches the Windows CE control to a **CWnd** object and replaces the**WndProc** and **AfxWndProc** functions of the window.

The function stores a pointer to the old **WndProc** in the **CWnd** object.

**发现**：SubclassWindow与SubclassDlgItem的MSDN说明惊人的相似。可见，SubclassDlgItem函数功能的实现是通过SubclassWindow实现的。那么，对于上面的问题等于没有任何发现。现在查看SubclassWindow源代码：

BOOL CWnd::SubclassWindow(HWND hWnd)

{

    if (!Attach(hWnd))

       return FALSE;

    // allow any other subclassing to occur

    PreSubclassWindow();

    // now hook into the AFX WndProc

1） **WNDPROC\* lplpfn = GetSuperWndProcAddr();**

2） **WNDPROC oldWndProc = (WNDPROC)::SetWindowLong(hWnd, GWL\_WNDPROC,**

**(DWORD)AfxGetAfxWndProc());**

    ASSERT(oldWndProc != (WNDPROC)AfxGetAfxWndProc());

    if (\*lplpfn == NULL)

3）    **\*lplpfn = oldWndProc;**   // the first control of that type created

    // 省略无关代码

    … …

    return TRUE;

}

在K这段代码之前，先回顾一下“MFC消息的起点”

**MFC消息起点和流动：**

    Windows消息怎样从产生到响应函数收到该消息？

**消息的起点**

    不管MFC是什么机理，其本质还是对Windows编程进行了整合封装，仅此而已！对Windows系统来说都是一样的，它都是不断地用GetMessage（或其他）从消息队列中取出消息，然后用DispatchMessage将消息发送到窗口函数中去。在“窗口类的诞生”中知道，MFC将所有的窗口处理函数都注册成DefWndProc，那么，是不是MFC将所有的消息都发送到DefWndProc中去了呢？答案是“不是”，**而是都发送到AfxWndProc函数中去了**（您可以回想一下前面我们查看MSDN是提到的AfxWndProc）。那么，MFC为什么要这样做呢？那就查看MFC代码，让它来告诉我们：

BOOL CWnd::CreateEx(……)

{

……

PreCreateWindow(cs);

**AfxHookWindowCreate(this);**

HWND hWnd = ::CreateWindowEx(……);

……

}

void AFXAPI AfxHookWindowCreate(CWnd \*pWnd)

{

……

pThreadState->m\_hHookOldCbtFilter =

**::SetWindowsHookEx(WH\_CBT,\_AfxCbtFilterHook,NULL,::GetCurrentThreadId());**

……

}

\_AfxCbtFilterHook(int code, WPARAM wParam, LPARAM lParam)

{

……

if(!afxData.bWin31)

{

**\_AfxStandardSubclass((HWND)wParam);**

}

……

}

void AFXAPI \_AfxStandardSubclass(HWND hWnd)

{

……

**oldWndProc =**

**(WNDPROC)SetWindowLong(hWnd,GWL\_WNDPROC,(DWORD)AfxGetAfxWndProc()**);

}

WNDPROC AFXAPI AfxGetAfxWndProc()

{

……

**return &AfxWndProc;**

}

仔细分析上面的代码，发现MFC在创建窗口之前，通过**AfxHookWindowCreate**设置了钩子（这样有消息满足所设置的消息时，系统就发送给你设置的函数，在这里就是**\_AfxCbtFilterHook**），这样每次创建窗口时，该函数就将窗口函数修改成AfxWndProc。至于为什么要这样做？那是因为包含3D控件和兼容MFC2.5。

   消息的起点都是AfxWndProc，所有消息都被发送到AfxWndProc中，然后在从AfxWndProc流向各自的消息响应函数。AfxWndProc的作用就和车站的作用是一样的，人们都要先到车站来，然后流向各种的目的地。那么自己的目的地在哪，只有自己才会知道。当然对于消息，也只有它自己才会知道要去哪！而“这种只有自己知道”在MFC中反映为“MFC根据不同类型的消息设置不同的消息路由路径，然后不同类型的消息走自己的路就OK了(计算机嘛！自己当然不会知道，要用算法嘛！）”。

**消息的流动**

LRESULT CALLBACK **AfxWndProc**(…….)

{

……

**return AfxCallWndProc(pWnd,hWnd,nMsg,wParam,lParam);**

}

LRESULT AFXAPI AfxCallWndProc(……)

{

……

lResult = **pWnd->WindowProc(nMsg,wParam,lParam);**

……

}

LRESULT CWnd::WindowProc(……)

{

……

**if(!OnWndMsg(message,wParam,lParam,&lResult))**

**lResult = DefWindowProc(message,wParam,lParam);**

……

}

BOOL CWnd::OnWndMsg(……)

//该函数原来太过庞大，为了只表达意思，将其改造如下

{

    ……

    if(message == WM\_COMMAND)

         OnCommand(wParam,lParam);

    if(message == WM\_NOTIFY)

         OnNotify(wParam,lParam,&lResult);

**//每个CWnd类都有它自己的消息地图**

**pMessage = GetMessageMap();**

**//在消息地图中查找当前消息的消息处理函数**

for(; pMessageMap!=NULL; pMessageMap = pMessageMap->pBaseMap)

{

        if((lpEntry**=AfxFindMessageEntry**(pMessageMap->lpEntries,

                    message,0,0))!=NULL)

         break;

}

(this->\*(lpEntry->pnf))(……);//调用消息响应函数

}

AFX\_MSGMAP\_ENTRY AfxFindMessageEntry(……)

{

……

while(lpEntry->nSign!=AfxSig\_end)

{

       if(lpEntry->nMessage==nMsg&&lpEntry->nCode==nCode&&nID>=lpEntry->nID

              &&nID<=lpEntry->nLastID)

       {

             return lpEntry;

       }

       lpEntry++;

}

……

}

仔细分析上面的代码，发现消息的路由关键在于OnCmdMsg函数。OnCmdMsg或者WM\_COMMAND消息调用OnCommand()，或者为WM\_NOTIFY消息调用OnNotify()。它将没有被处理的消息都是为**窗口消息**。OnCmdMsg()搜索类的消息映射，以便找到一个能处理窗口消息的处理函数。

    这样我们就找到了消息的处理函数了。

    回顾完这些知识，我们回到SubclassWindow的实现代码中。

    我们发现了三个关键的语句：

1） **WNDPROC\* lplpfn = GetSuperWndProcAddr();**

2） **WNDPROC oldWndProc = (WNDPROC)::SetWindowLong(hWnd, GWL\_WNDPROC,**

**(DWORD)AfxGetAfxWndProc());**

3）                **\*lplpfn = oldWndProc;**

对这三个语句进行分析：

1）            获取原窗口处理过程地址，随便说一下是怎么获得的：窗口通过CreateEx创建，在调用CreateEx中又调用了CreateWindowEx，调用该函数后将原来的窗口处理函数地址保存在了**窗口类的成员函数m\_pfnSuper**中了。

2）            看到了没：用SetWindowLong将窗口处理函数改为AfxGetWndProc，根据前面的分析，它会调用AfxWndProc，再通过OnCmdMsg进行消息的路由。结合本例，CSuperEdit有它自己的消息地图，WM\_CHAR消息路由时就会找了CSuperEdit类它自己的OnChar()函数，这样子类化的目的----封装自己的消息处理函数----就达到了。

3）            当然CSuperEdit只定义了一部分自己的消息处理函数，大部分还是要由原来的函数（CWnd）完成，所以要保存原来函数地址，这句代码就完成此功能。

至此，我们关于子类化的来龙去脉就搞清楚了。

补充：我们在用ClassWizard将一个控件与一个自定义的类型关联起来后，我们并没有添加像xxx.SubclassDlgItem(xxx,this);这样的代码，但是也实现了以上的功能？原因是ClassWizard已经为我们实现了上面的关联，其原理是一样的。