**第一个：FindWindow根据窗口类名或窗口标题名来获得窗口的句柄，该函数返回窗口的句柄**

函数的定义:HWND WINAPI FindWindow(LPCSTR lpClassName ,LPCSTR lpWindowName);

第一个参数填窗口的类名，第二个填窗口的标题名，其实是不需要同时填两个参数的，也就是说，你只要知道窗口的类名或窗口的标题就可以了，没有的那个就用NULL代替。

比如现在有一个窗口名为"无标题.txt - 记事本"的记事本程序。那么我就可以用上面的函数获得这个窗口的句柄，那获得了这个窗口的句柄我可以干什么呢？作用可大了，因为很多操作窗口的函数，都需要窗口句柄作参数，如移动、改变窗口大小的MoveWindow函数，在这里举个例子，大家就更能体会到这个FindWindow的用法、用处。

FindWindow例子：已知一个窗口名称，写一个程序关闭该窗口，假设当前电脑正有一个窗口名为"无标题.txt - 记事本"的记事本程序运行

#include<windows.h> //API函数的头文件  
 int main()  
 {  
 HWND wnd; //定义一个窗口句柄变量，用以存储找到的窗口句柄  
 wnd=FindWindow(NULL,"无标题.txt - 记事本"); //获得窗口名为"无标题.txt - 记事本"的窗口句柄  
 SendMessage(wnd,WM\_CLOSE,0,0); //调用SendMessage函数，发送一个WM\_CLOSE（关闭）消息给wnd窗口句柄。  
 return 0;  
 }

如果要根据窗口类名来获得窗口句柄话，只要给函数的第一个参数填类名，第二个参数窗口名填NULL，即可，用Spy++可查看窗口类名。

**第二个：SendMessage根据窗口句柄发送一个消息给窗口**

函数定义：LRESULT SendMessage（HWND hWnd，UINT Msg，WPARAM wParam，LPARAM IParam）；

第一个参数是窗口句柄，第二参数个是消息类型，下面的消息表列举了所有消息，第三，四个参数是消息附带信息，解释依赖于消息类型，比如一个字符消息（WM\_CHAR),那么第三个参数就储存有一个字符的ASCII码。

消息机制大家都应该知道吧，Windows是基于消息的系统，鼠标移动键盘按键都会产生消息。

接下来举一个例子，发送一个WM\_CHAR消息给窗口，也就是模仿键盘按键，接收消息的窗口依旧以"无标题.txt - 记事本"为例：

SendMessage例子：模仿键盘按键

#include<windows.h>  
 int main()  
 {  
 HWND wnd;  
 wnd=FindWindow(NULL,"无标题.txt - 记事本");  
 while(1)  
 {  
 SendMessage(wnd,WM\_CHAR,WPARAM('a'),0);  
 Sleep(300);  
 }  
 return 0;  
 }

呵呵上面的例子是不是没用，这是为什么呢，哪里出错了吗？错倒是没有错，只是窗口句柄有问题，消息发送给了主窗口。接收消息的窗口不对。记事本窗口界面有些有什么东西呢？菜单，编辑框， 状态栏等控件，控件也是窗口，既然是窗口，那当然它们也有窗口句柄，而在记事本里是在哪里打字的？编辑框控件里打字的嘛！所以消息应该发送编辑框控件，那如何获得记事本里编辑框控件的窗口句柄呢？用FindWindow吗？不知道编辑框窗口标题名，类名也不知道，当然也有其它方法获取编辑框窗口标题名和窗口类名，如Spy++。关于如何获得编辑框句柄，将在以后的函数中会有介绍，这里我们就用WindowFromPoint这个函数来获取，这个函数获取窗口句柄的方法比较笨，（相对于我这个例子来说），这个函数是根据什么来获取窗口句柄的呢？根据屏幕坐标点，如屏幕坐标点20，20，当前是哪个窗口占有，就返回哪个窗口的句柄。有了这个函数，我们还需要一个函数GetCursorPos获取鼠标当前位置（针对于屏幕）；

可行的例子：模仿键盘按键：

#include<windows.h>  
 int main()  
 {  
 POINT curpos; //一个可储存坐标点的结构体变量，x横坐标，y,纵坐标，如curpos.x curpos.y  
 while(1)  
 {  
 GetCursorPos(&curpos); //获取当前鼠标的位置，位置将储存在curpos里。  
 HWND hWnd = WindowFromPoint(curpos); //根据curpos所指的坐标点获取窗口句柄  
 SendMessage(hWnd,WM\_CHAR,WPARAM('g'),0); //发送一个字符（按键）消息g给当前鼠标所指向的窗口句柄  
 Sleep(300); //睡眠三百毫秒，相当于等待三分之一秒  
 }  
 }

这个程序一运行后，只要把鼠标指向要输入字符的窗口句柄，那么就相当于键盘每三分之一秒按了一个g键，试试吧！

如果这样觉得模仿键盘按键太麻烦的话，那么就用keybd\_event这个函数，这个专门用于模仿键盘按键的，关于怎么用，自己百度一搜，就知道了。既然SendMessage能模仿键盘按键的话，那也能模仿鼠标左击，右击。而此时SendMessage函数第三，四个参数的解释就是储存有鼠标左击，右击时的位置。如模仿鼠标右击，想一想，一次鼠标右击有哪几步，分别是鼠标右键按下，鼠标右键松开，如果你按下鼠标右键不松开，那它是不是鼠标右击，不是的，直到你松开鼠标右键，才能算是一次完整的鼠标右击.鼠标右键按下的消息类型是“WM\_RBUTTONDOWN”，右键松开的消息是“WM\_RBUTTONUP”，那么一次完整的鼠标右击应该是：

SendMessage(wnd,WM\_RBUTTONDOWN,0,0); //鼠标右键按下,第三，四个参数说明了鼠标按下时的位置  
 Sleep(100); //间隔100毫秒  
 SendMessage(wnd,WM\_RBUTTONUP,0,0); //鼠标右键松开

同样，也有一个专门模仿鼠标动作的函数，mouse\_event这个函数，可以模仿鼠标的移动，单击，双击等。以后会有专门介绍。

**第三个：GetCursorPos获取鼠标当前位置（屏幕）**

这个函数在SendMessage函数有介绍，这里仅举一个例子，在界面里不停的输出鼠标当前位置。

#include<windows.h>  
 #include<stdio.h>  
 int main()  
 {  
 POINT curpos;  
 while(1)  
 {  
 GetCursorPos(&curpos);  
 printf("x:%d,y:%d",curpos.x,curpos.y);  
 Sleep(300);  
 printf("\n");  
 }  
 }

**第四个：WindowFromPoint根据坐标点获得对应的窗口句柄**

在SendMessage有解释，这里仅举一个例子，鼠标指向哪个窗口，就关闭哪个窗口。

#include<windows.h>  
 int main()  
 {  
 Sleep(2500); //等待一会儿，用于把鼠标移到其它窗口上去，避免指向本身进程的窗口，关掉自己的窗口。  
 POINT curpos;  
 while(1)  
 {  
 GetCursorPos(&curpos);  
 HWND wnd=WindowFromPoint(curpos);  
 SendMessage(wnd,WM\_CLOSE,0,0);  
 Sleep(300);  
 }  
 }

**第五个MoveWindow根据窗口句柄移动窗口，改变窗口大小**

函数定义：BOOL MoveWindow( HWND hWnd, int X, int Y, int nWidth, int nHeight, BOOL bRepaint );  
hWnd是要改变大小的窗口的句柄，x,y相对于屏幕的坐标，窗口左上角的位置与之相对应，nWidth和nHeight是窗口新的宽高，bRepaint指定窗口是否重画。  
这里依旧以"无标题.txt - 记事本"为例子，改变这个窗口大小，并把窗口移到左上角去。

#include<windows.h>  
 int main()  
 {  
 HWND wnd;  
 wnd=FindWindow(NULL,"无标题.txt - 记事本");  
 MoveWindow(wnd,0,0,220,120,NULL);  
 return 0;  
 }

**第六个ShowWindow设置窗口显示状态，如隐藏，最大化，最小化**

函数定义BOOL ShowWinow(HWND hWnd,int nCmdShow);  
      SW\_HIDE：隐藏窗口并激活其他窗口。第一个参数hWnd指明了窗口句柄，第二个参数指明了窗口的状态，现在给出第二个参数常用取值范围：  
　　SW\_MAXIMIZE：最大化指定的窗口。  
　　SW\_MINIMIZE：最小化指定的窗口并且激活在Z序中的下一个顶层窗口。  
　　SW\_RESTORE：激活并显示窗口。如果窗口最小化或最大化，则系统将窗口恢复到原来的尺寸和位置。在恢复最小化窗口时，应用程序应该指定这个标志。  
      SW\_SHOW：在窗口原来的位置以原来的尺寸激活和显示窗口。  
ShowWindow例子：程序运行后，在桌面上隐藏一个指定的窗口，并在4秒后再将其显示

#include<windows.h>  
 int main()  
 {  
 HWND wnd;  
 wnd=FindWindow(NULL,"无标题.txt - 记事本");  
 ShowWindow(wnd,SW\_HIDE);  
 Sleep(5000);  
 ShowWindow(wnd,SW\_SHOW);  
 return 0;  
 }

**第七个SetCursorPos设置鼠标的位置、把鼠标移动到指定的位置**

函数定义：BOOL SetCursorPos(int x,int y);

这个函数的两个参数我想大家应该知道是什么意思吧，屏幕的坐标点。

直接看例子：

#include<windows.h>  
 int main()  
 {  
 int sec=0;  
 while(sec<200)  
 {  
 SetCursorPos(rand()%1024,rand()%768); //随机设置鼠标的位置  
 Sleep(20);  
 sec++;  
 }  
 return 0;  
 }

**第八个CopyFile复制一个文件**

如何复制一个文件，比如，我要把E盘的abb.txt的文本文件复制到d盘的zhengyong.txt,则调用语句

CopyFile("e:\\abb.txt","d:\\zhengyong.txt",FALSE);

第三个参数有以下说明：  
如果设为TRUE（非零），那么一旦目标文件已经存在，则函数调用会失败。否则目标文件会被覆盖掉。

**第九个DeleteFile删除一个文件**

如何删除一个文件，语句：DeleteFile("e\\abb.txt");既是删除  
如果目标为隐藏或只读，则无用。

**第十个CreateDirectory创建一个文件夹（目录）**

假如E盘下什么文件也没有  
CreateDirectory("e:\\aaa\\bbb",NULL);这样是错的，不能同时建两个文件，除非E盘下已经有了个aaa文件夹了。  
这样是对的CreateDirectory("e:\\aaa",NULL);

**第十一个：GetClientRect获得窗口大小**

看例子：

#include<windows.h>  
 #include<stdio.h>  
 int main(int argc, char\* argv[])  
 {  
 HWND wnd;  
 while(1)  
 {  
 wnd=FindWindow(NULL,"无标题.txt - 记事本");  
 RECT rect; //专门用来存储窗口大小  
 GetClientRect(wnd,&rect); //获取窗口大小  
 printf("%d,%d,%d,%d\n",rect.left,rect.top,rect.right,rect.bottom); //输出窗口大小，试着用鼠标改变窗口大小  
 Sleep(300);  
 }  
 }

**第十二个：GetCWindowRect获得窗口大小（相对屏幕）**

例子：

#include<windows.h>  
 #include<stdio.h>  
 int main(int argc, char\* argv[])  
 {  
 HWND wnd;  
 while(1)  
 {  
 wnd=FindWindow(NULL,"无标题.txt - 记事本");  
 RECT rect; //专门用来存储窗口大小  
 GetWindowRect(wnd,&rect); //获取窗口大小  
 printf("%d,%d,%d,%d\n",rect.left,rect.top,rect.right,rect.bottom); //输出窗口大小，试着用鼠标改变窗口大小  
 Sleep(300);  
 }  
 }

试着去找一下GetClientRect和GetWindowRect之间有什么区别；

**第十三个FindFirstFile寻找文件以及获得文件的信息**

这里举一个例子吧，列举E盘第一目录下的所有文件，包括文件夹，结合FindNextFile

#include<windows.h>  
 #include<stdio.h>  
 int main()  
 {  
 BOOL done=TRUE;  
 WIN32\_FIND\_DATA fd;  
 HANDLE hFind = FindFirstFile("e:\\\*.\*", &fd); //第一个参数是路径名，可以使用通配符，懂DOS的人应该知道吧！fd存储有文件的信息  
 while (done)  
 {  
 printf("%s\n",fd.cFileName);  
 done=FindNextFile(hFind, &fd); //返回的值如果为0则没有文件要寻了  
 }  
 return 0;  
 }

当然也可以直接找一个文件，不使用通配符，但这样有什么意义呢？，如FindFirstFile("e:\\aaa.txt",&fd);其实这个可以获取一个文件的信息，如文件是不是隐藏的，或者有没有只读属性等。

当然通过控制通配符，也可以寻找特定类型的文件，比如我只要找文本文件，那么就是这个语句FindFirstFile("e:\\\*.txt",&fd);就行了，关键看你自己灵活运用。

前面说过fd里存储有文件的信息，那怎么根据fd里面的成员判断这个文件的属性，文件是否隐藏，是不是文件夹。

fd里的dwFileAttributes存储有文件的信息，如判断是否为文件夹，只要把这个变量和FILE\_ATTRIBUTE\_DIRECTORY进行按位与运算，如果为1的话，表明为文夹件，如if(fd.dwFileAttributes&FILE\_ATTRIBUTE\_DIRECTORY==1) printf("%s是文件夹\n",fd.cFileName);

其它判断也是一样，现在给出文件的属性(常用几个）：FILE\_ATTRIBUTE\_HIDDEN（隐藏）

FILE\_ATTRIBUTE\_READONLY（只读）FILE\_ATTRIBUTE\_SYSTEM（系统）

**第十四个FindNextFile寻找文件**

参照FindFirstFile函数的例子!

**第十五个MoveFile移动文件**

如把一个盘里的文本移到另一个盘里去:MoveFile("e:\\a.txt","d:\\abc.txt");即可，意思把e盘下的a.txt移到d盘下去，并改名为abc.txt

**第十六个GetClassName根据窗口句柄获得窗口类名**

函数定义：int GetClassName(HWND hWnd, LPTSTR IpClassName, int nMaxCount)；

这种函数不需要再解释了吧，前面有太多类似的例子。

**第十七个SetFileAttributes设置文件属性**

函数定义：BOOL SetFileAttributes( LPCTSTR lpFileName, DWORD dwFileAttributes);

这个函数的第二个参数dwFileAttributes和前面讲过的WIN32\_FIND\_DATA结构里的dwFileAttributes成员相对应。假设E盘第一目录下有一个文本文件a.txt的正常文件，我要把它设为只读和隐藏那要如何做呢？在前面介绍过WIN32\_FIND\_DATA结构里dwFileAttributes成员的几个常用属性，根据这个我们知道隐藏是FILE\_ATTRIBUTE\_HIDDEN，只读是FILE\_ATTRIBUTE\_READONLY。

那么把E盘下文本文件的属性设为隐藏和只读的语句就是：

SetFileAttributes("e:\\a.txt",FILE\_ATTRIBUTE\_HIDDEN|FILE\_ATTRIBUTE\_READONLY);

(说明：这个函数同样也能设置文件夹属性）

虽然这个语句可以达到要求，但不建议用，因为会覆盖掉文件的原来属性，也就是说如果这个文件之前有系统属性（系统文件）的话，那么这个语句一旦执行后，文件就只有隐藏和只读属性了。

比如一个文件原先就有隐藏属性，依旧以a.txt为例子，那么我把它设为只读，是不是这个语句就可以呢？

SetFileAttributes("e:\\a.txt",FILE\_ATTRIBUTE\_READONLY）;这样的话，虽然文件有只读属性了，但隐藏属性却没有了。

那要如何在不覆盖掉原来的属性下，把文件设为只读呢，其实说了这么多的废话，总结起来就一句话：如何增加一个文件的属性！

前提是要获得这个文件的原有属性：获得文件的属性，在FindFirstFile函数讲过。好吧！直接看例子：

假设e盘的a.txt文件属性为隐藏，给它增加只读属性：

#include<windows.h>  
 int main()  
 {  
 WIN32\_FIND\_DATA fd;  
 FindFirstFile("e:\\a.txt",&fd);  
 fd.dwFileAttributes|=FILE\_ATTRIBUTE\_READONLY; //在原来的属性下增加只读属性  
 SetFileAttributes("e:\\a.txt",fd.dwFileAttributes); //设置文件的属性  
 return 0;  
 }

第二个例子：如何去掉一个文件的属性

（补习一下，懂的人直接跳过）

我想懂这里的按位或、按位与或者按位异或运算的人应该知道该如何去掉一个文件的属性。其实一个文件信息都是以二进制代码说明的。

比如一个八位二进制码：10000010，这里的每一位是不是只有0和1取值，不是0，就是1，正好符合一个文件属性的有无，如这个文件是隐藏的吗？只有是和不是，这样我们规定把这八位二进制码的第一位用于确定文件是否具有隐藏属性，如果为1那便是隐藏，无则没有，以此类推第二位就代表文件的只读，第三位系统。。。但要如何判断呢，或者把某一位的值改变呢，用按位运算就可以，00000010，我要把第2位的值设为0，其它位上的值保持不变，用按位异或运算即可，与00000010进行按位异或运算，但这里并不是与它本身进行运算，不管任何八位二进制数的值是多少只要与00000010进行按位异或运算，那第二位都会变成0，而其它的位保持不变。这样为了方便，我们就把00000010进行宏定义，方便记忆，这个二进制数的十进制为2。宏定义#define FILE\_ATTRIBUTE\_READONLY 2

明白了这个我们就来清除一个文件的一种属性吧！

清除一个文件的隐藏属性，假设a.txt为隐藏文件：

#include<windows.h>  
 int main()  
 {  
 WIN32\_FIND\_DATA fd;  
 FindFirstFile("e:\\a.txt",&fd); //获取文件信息  
 fd.dwFileAttributes^=FILE\_ATTRIBUTE\_HIDDEN; //在原来的属性下删除隐藏属性  
 SetFileAttributes("e:\\a.txt",fd.dwFileAttributes); //设置文件的属性  
 return 0;  
 }

如果单单只针对文件的属性进行操作的话，可以用GetFileAttributes函数获取文件的属性，该函数只一个参数，那就是文件的路径，函数返回一个DWORD值，包含文件属性信息。

**第十八个ShellExecute运行一个程序**  
函数定义:ShellExecute(HWND hwnd, LPCSTR lpOperation, LPCSTR lpFile, LPCSTR lpParameters, LPCSTR lpDirectory, INT nShowCmd);  
第一个参数hwnd是父窗口的句柄,可以为NULL,第二个参数lpOperation表示行为,第三个参数lpFile是程序的路径名,第四个参数lpParameters是给所打开程序的参数,可以为NULL,第五个参数lpDirectory可以为NULL,第六个参数nShowCmd跟ShowWindow函数的第二个参数一样,作用也一样,如果打开的程序有窗口的话,这个参数就指明了窗口如何显示.  
例如打开一个记事本:

ShellExecute(NULL,"open","NOTEPAD.EXE",NULL,NULL,SW\_SHOWNORMAL);

而且这个函数还可以指定程序打开一个属于程序本身类型的文件,假如e盘有一个a.txt文件;我调用函数运行记事本程序并打开这个文本文件.

ShellExecute(NULL,"open","NOTEPAD.EXE","e:\\a.txt",NULL,SW\_SHOWNORMAL);

这里由于记事本程序属于系统本身自带的程序,所以没有绝对路径.  
这个函数还可以打开一个网站:

ShellExecute(NULL,"open","http://www.baidu.com",NULL,NULL,SW\_SHOWNORMAL);  
 ShellExecute(NULL,"open","C:",NULL,NULL,SW\_SHOWNORMAL);

类似的函数还有WinExec，只有两个参数,它的最后一个参数跟ShellExecute函数的最后一个参数一样.  
而第一个参数则是程序路径名.举个例子:WinExce("NOTEPAD.EXE",SW\_SHOWNORMAL);  
这个函数也可以给程序传递一个文件名供要运行的程序打开,那要如何加进去呢,这里又没有第三个参数,  
方法把路径名加在NOTPEPAD.EXE的后面,要以空格来分开如:

WinExce("NOTEPAD.EXE e:\\a.txt",SW\_SHOWNORMAL);

**第十九个PlaySound播放一个WAV文件**

函数定义：BOOL PlaySound(LPCSTR pszSound, HMODULE hmod,DWORD fdwSound);

第一个参数是WAV文件的路径名，第二个参数如果不是播放MFC里以资源ID命名的文件，则可以为空，第三个参数，指明了以何种方式播放文件。注意这个函数只能播放100K以下的WAV文件。

假如E盘有个a.wav文件，下面这个例子播放这个文件：

#include<windows.h>  
 #include<mmsystem.h> //PlaySound函数的头文件  
 #pragma comment(lib, "winmm.lib") //链接库，PlaySound函数必须使用  
 int main()   
 {  
 PlaySound("e:\\19.wav",NULL,SND\_SYNC);  
 return 0;  
 }

**第二十个GetModuleFileName根据模块导入表获程序的完整路径**

函数定义：DWORD GetModuleFileName( HMODULE hModule, LPTSTR lpFilename, DWORD nSize );

关于第一个参数，将在以后的动态链接库里会有介绍，这里我们只要获得程序本身的路径，那么第一个参数可以为空。

第二个参数用以存储路径，nSize指明字符数组大小。

这个举个例子，运行后，把自身程序移动到e盘下，并改名为a.exe;

#include<windows.h>  
 int main()  
 {  
 char szAppName[128]={0};  
 GetModuleFileName(NULL,szAppName,128);  
 MoveFile(szAppName,"e:\\a.exe");  
 return 0;  
 }

**第二十一个CreateWindow创建一个窗口**

//补习懂的人直接跳过

之前API函数的例子，都是针对DOS编程的，严格来说是在windows下的仿DOS（cmd)进行编程，编写控制台应用程序大家都知道，主函数是main，那针对windows编程的主函数也是main吗？不是的，windows下的主函数（入口函数）是WinMain。在定义main主函数的时候，可以给它带两个参数，也可以不带。而WinMain函数就不能这样了，它有固定的格式，它必须带四个参数。

现给出WinMain函数的固定格式：

int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, instance LPSTR lpCmdLine,  int nCmdShow)

大家如果有兴趣可以通过其它渠道了解一下各参数的意思，现在我们只需要知道WinMain函数就是这样定义的，不理解也没关系。

知道了这个我们就来编一个WINDOWS程序吧！

因为我们是针对windows编程，所以要建一个Win32 Application工程，步骤是点击文件，然后选择新建，在弹出的对话框里选择工程，再选中Win32 Application 接着在右边的工程名称下填工程名称，名字随便取。之后点确定。接着又弹出了一个对话框，这里为了方便，我们选择“一个简单的  Win32 程序”，点完成。接着双击WinMain弹出代码编辑窗口，做完这个我们就可以打代码了。



 简单的例子如下：

#include "stdafx.h"  
 int APIENTRY WinMain(HINSTANCE hInstance,  
 HINSTANCE hPrevInstance,  
 LPSTR lpCmdLine,  
 int nCmdShow)  
 {  
 while(1)  
 Sleep(100);  
 return 0;  
 }

怎么样够简单吧，是不是觉得奇怪，怎么没有窗口，因为窗口要自己创建，不像控制台程序，只要一运行便会有窗口。虽然没有窗口，但你创建了一个进程，打开任务管理器，可以找到你所创建的那个进程，其实也没什么奇怪的，像WINDOWS本身的一些系统服务，也是只有进程，没有窗口的像spoolsv.exe,svchost.exe。

那要如何创建一个窗口呢？要创建一个窗口,就必须要向系统提供窗口的信息，如你要创建的窗口名字叫什么，窗口图标是什么，窗口大小，窗口背景色等，不然，系统怎么给你创建窗口呢？所以为了方便，VC就定义了一个结构，专门用存储窗口信息。

现给出这个结构的定义。

typedef struct \_WNDCLASS {   
 UINT style; //描述类风格  
 WNDPROC lpfnWndProc; //窗口处理函数  
 int cbClsExtra; //表示窗口类结构之后分配的额外的字节数。系统将该值初始化为0  
 int cbWndExtra; //表示窗口实例之后分配的额外的字节数。系统将该值初始化为0  
 HINSTANCE hInstance; // 应用程序实例句柄由WinMain函数传进来   
 HICON hIcon; //窗口图标句柄   
 HCURSOR hCursor; //窗口光标句柄  
 HBRUSH hbrBackground; //画刷句柄  
 LPCTSTR lpszMenuName; //窗口菜单名  
 LPCTSTR lpszClassName; //窗口类名  
 } WNDCLASS, \*PWNDCLASS;

好了，如果我们已经把窗口信息填好了，那我们要怎样把这个信息告诉系统呢，也就是把要创建窗口的信息传给系统。这里我们调用RegisterClass函数就能实现这个功能。注册完窗口，我们就要创建窗口,用CreateWindow函数就能实现，不要问为什么注册窗口后直接显示不就行了，还要搞什么创建窗口。这我也不知道，反正你只要记住这格式就行了，硬式规定的，你想创建一个窗口，就必须按这些步骤来。

好了，窗口创建了，我们就要调用ShowWindow函数显示窗口，然后用UpdateWindow函数刷新一下，确保窗口能立即显示。

以下详细实现代码：

#include "stdafx.h"  
 #include<windows.h>  
 int APIENTRY WinMain(HINSTANCE hInstance,  
 HINSTANCE hPrevInstance,  
 LPSTR lpCmdLine,  
 int nCmdShow)  
 {  
 WNDCLASS wndcls; //定义一个存储窗口信息WNDCLASS变量  
 wndcls.cbClsExtra=0; //默认为0  
 wndcls.cbWndExtra=0; //默认为0  
 wndcls.hbrBackground=(HBRUSH)GetStockObject(GRAY\_BRUSH); //背景画刷  
 wndcls.hCursor=LoadCursor(NULL,IDC\_CROSS); //十字光标  
 wndcls.hIcon=LoadIcon(NULL,IDI\_ERROR); //窗口图标  
 wndcls.hInstance=hInstance; //应用程序实例句柄由WinMain函数传进来   
 wndcls.lpfnWndProc=NULL; //窗口消息处理函数  
 wndcls.lpszClassName="windowclass"; //窗口类名  
 wndcls.lpszMenuName=NULL; //窗口菜单名，没有菜单，为NULL  
 wndcls.style=CS\_HREDRAW | CS\_VREDRAW; //窗口类型，CS\_HREDRAW和CS\_VERDRAW 表明  
 //当窗口水平方向垂直方向的宽度变化时重绘整个窗口  
 RegisterClass(&wndcls); //把窗口信息提交给系统，注册窗口类  
 HWND hwnd; //用以存储CreateWindow函数所创建的窗口句柄  
 hwnd=CreateWindow("windowclass","first windows",   
 WS\_OVERLAPPEDWINDOW,0,0,600,400,NULL,NULL,hInstance,NULL);//创建窗口  
 ShowWindow(hwnd,SW\_SHOWNORMAL); //窗口创建完了，显示它  
 UpdateWindow(hwnd); //更新窗口，让窗口毫无延迟的显示  
 return 0;  
 }

是不是出错了，内存不能读取，为什么了呢，因为你创建的窗口没有消息处理函数，windows系统当然不允许这样一个窗口存在，对按键，鼠标都没有反应，这样的窗口是没有实际意义的。  wndcls.lpfnWndProc=NULL; //窗口消息处理函数，就是前面这句，必须要填

窗口过程（消息）处理函数，那这个函数是怎样定义的呢，像WinMain一样，它也有固定的格式。

窗口过程处理函数的格式：LRESULT CALLBACK WinSunProc(HWND wnd,UINT uMsg,WPARAM wParam,LPARAM lParam)

下面的这个是一个窗口创建的完整例子：

#include "stdafx.h"  
 #include<windows.h>  
 LRESULT CALLBACK WinSunProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)  
 {  
 if(uMsg==WM\_LBUTTONDOWN) MessageBox(NULL,"kdjfkdf","Kjdfkdfj",MB\_OK); //处理鼠标按下消息，弹出消息框  
 return DefWindowProc(hwnd,uMsg,wParam,lParam); //未处理的消息通过DefWindowProc函数交给系统处理  
 }  
 int APIENTRY WinMain(HINSTANCE hInstance,  
 HINSTANCE hPrevInstance,  
 LPSTR lpCmdLine,  
 int nCmdShow)  
 {  
 WNDCLASS wndcls; //定义一个存储窗口信息WNDCLASS变量  
 wndcls.cbClsExtra=0; //默认为0  
 wndcls.cbWndExtra=0; //默认为0  
 wndcls.hbrBackground=(HBRUSH)GetStockObject(GRAY\_BRUSH); //背景画刷  
 wndcls.hCursor=LoadCursor(NULL,IDC\_ARROW); //光标  
 wndcls.hIcon=LoadIcon(NULL,IDI\_ERROR); //窗口图标  
 wndcls.hInstance=hInstance; //应用程序实例句柄由WinMain函数传进来   
 wndcls.lpfnWndProc=WinSunProc; //窗口消息处理函数  
 wndcls.lpszClassName="windowclass"; //窗口类名  
 wndcls.lpszMenuName=NULL; //窗口菜单名，没有菜单，为NULL  
 wndcls.style=CS\_HREDRAW | CS\_VREDRAW; //窗口类型，CS\_HREDRAW和CS\_VERDRAW 表明  
 //当窗口水平方向垂直方向的宽度变化时重绘整个窗口  
 RegisterClass(&wndcls); //把窗口信息提交给系统，注册窗口类  
 HWND hwnd; //用以存储CreateWindow函数所创建的窗口句柄  
 hwnd=CreateWindow("windowclass","first windows",   
 WS\_OVERLAPPEDWINDOW,0,0,600,400,NULL,NULL,hInstance,NULL); //创建窗口  
 ShowWindow(hwnd,SW\_SHOWNORMAL); //窗口创建完了，显示它  
 UpdateWindow(hwnd); //更新窗口，让窗口毫无延迟的显示  
 MSG msg; //消息结构类型  
 while(GetMessage(&msg,NULL,0,0)) //获取消息  
 {  
 //TranslateMessage(&msg); //此函数用于把键盘消息(WM\_KEYDOWN,WM\_KEYUP)转换成字符消息WM\_CHAR  
 DispatchMessage(&msg); //这个函数调用窗口过程处理函数，并把MSG里的信息处理后传给过程函数的四个参数  
 }  
 return 0;  
 }

WinSunProc函数的四个参数，分别对应着SendMessage函数四个参数，详情参见SendMessage函数参数解释。

MSG类型解释 ：

结构定义：

typedef struct tagMSG   
 {  
 HWND hwnd; //hwnd表示消息将要发送给的窗口句柄  
 UINT message; //消息类型，如WM\_WMCLOSE,WM\_CHAR,WM\_LBUTTONDOWN,参见消息表  
 WPARAM wParam; //消息附带信息，取值的意思具体依据消息类型而定  
 LPARAM lParam; //消息附带信息，取值的意思具体依据消息类型而定  
 DWORD time; //消息的发送时间，不常用  
 POINT pt; //消息发送时，鼠标所在的位置，不常用  
 }MSG;

大家试着把上面的例子运行一遍，然后关掉窗口，再运行一遍，是不是出错了，因为前一个程序虽然窗口关闭了，但进程还在运行，还记得那个循环语句吗？while(GetMessage(&msg,NULL,0,0))就是这个。只要条件成立，进程就会一直运行下去。如何让这个循环结束呢？用 PostQuitMessage(0); 这个语句就行了，参数0表示给自身窗口发送一个退出消息，当GetMessage函数接到PostQuitMessage函数发出的消息后，就会返回0值。

如在窗口过程函数中处理窗口关闭WM\_CLOSE消息:if(uMsg==WM\_CLOSE）PostQuitMessage(0); 这样只要一关闭窗口，它的进程也会结束。

接下来解释一下CreateWindow函数参数的意思,函数定义

HWND CreateWindow(LPCTSTR lpClassName, //窗口类名，应与WNDCLASS结构里的成员lpszClassName一致  
 LPCTSTR lpWindowName,, //窗口标题名  
 DWORD dwStyle， //窗口的风格，取值参见表Style  
 int x,  
 int y, //x,y表示所创建窗口左上角位置  
 int nWidth,  
 int nHeight, //nWidth,nHeight表示窗口的宽高  
 HWND hWndParent, //父窗口句柄，如果不是子窗口，这里取值为NULL  
 HMENU hMenu, //菜单句柄，没菜单的话，取NULL值  
 HANDLE hlnstance, //对应着WinMain函数的第一个参数  
 LPVOID lpParam); //NULL

表Style：（参考：百度）

WS\_BORDER：创建一个单边框的窗口。 　　  
WS\_CAPTION：创建一个有标题框的窗口（包括WS\_BODER风格）。  
WS\_CHILD：创建一个子窗口。这个风格不能与WS\_POPUP风格合用。  
WS\_CHLDWINDOW：与WS\_CHILD相同。  
WS\_CLIPCHILDREN:当在父窗口内绘图时，排除子窗口区域。在创建父窗口时使用这个风格。 　　  
WS\_CLlPBLINGS；排除子窗口之间的相对区域，也就是，当一个特定的窗口接收到WM\_PAINT消息时，WS\_CLIPSIBLINGS 风格将所有层叠窗口排除在绘图之外，只重绘指定的子窗口。如果未指定WS\_CLIPSIBLINGS风格，并且子窗口是层叠的，则在重绘子窗口的客户区时，就会重绘邻近的子窗口。  
WS\_DISABLED:创建一个初始状态为禁止的子窗口。一个禁止状态的窗口不能接受来自用户的输入信息.  
WS\_DLGFRAME:创建一个带对话框边框风格的窗口。这种风格的窗口不能带标题条。  
WS\_GROUP:指定一组控制的第一个控制。这个控制组由第一个控制和随后定义的控制组成，自第二个控制开始每个控制，具有WS\_GROUP风格，每个组的第一个控制带有WS\_TABSTOP风格，从而使用户可以在组间移动。用户随后可以使用光标在组内的控制间改变键盘焦点。 　　  
WS\_HSCROLL：创建一个有水平滚动条的窗口。 　　  
WS\_ICONIC：创建一个初始状态为最小化状态的窗口。  
与WS\_MINIMIZE风格相同。 　　  
WS\_MAXIMIZE：创建一个初始状态为最大化状态的窗口。 　　  
WS\_MAXIMIZEBOX：创建一个具有最大化按钮的窗口。该风格不能与WS\_EX\_CONTEXTHELP风格同时出现，同时必须指定WS\_SYSMENU风格。 　　  
WS\_OVERLAPPED:产生一个层叠的窗口。一个层叠的窗口有一个标题条和一个边框。与WS\_TILED风格相同。 　　WS\_OVERLAPPEDWINDOW：创建一个具有WS\_OVERLAPPED，WS\_CAPTION，WS\_SYSMENU WS\_THICKFRAME，WS\_MINIMIZEBOX，WS\_MAXIMIZEBOX风格的层叠窗口，与WS\_TILEDWINDOW风格相同。 　　WS\_POPUP；创建一个弹出式窗口。该风格不能与WS\_CHLD风格同时使用。 　　  
WS\_POPUWINDOW：创建一个具有WS\_BORDER，WS\_POPUP,WS\_SYSMENU风格的窗口，WS\_CAPTION和WS\_POPUPWINDOW必须同时设定才能使窗口某单可见。 　　  
WS\_SIZEBOX：创建一个可调边框的窗口，与WS\_THICKFRAME风格相同。 　　  
WS\_SYSMENU：创建一个在标题条上带有窗口菜单的窗口，必须同时设定WS\_CAPTION风格。 　　  
WS\_TABSTOP：创建一个控制，这个控制在用户按下Tab键时可以获得键盘焦点。按下Tab键后使键盘焦点转移到下一具有WS\_TABSTOP风格的控制。 　　  
WS\_THICKFRAME：创建一个具有可调边框的窗口，与WS\_SIZEBOX风格相同。 　　  
WS\_TILED：产生一个层叠的窗口。一个层叠的窗口有一个标题和一个边框。  
与WS\_OVERLAPPED风格相同。 　　  
WS\_TILEDWINDOW:创建一个具有WS\_OVERLAPPED，WS\_CAPTION，WS\_SYSMENU， WS\_THICKFRAME，WS\_MINIMIZEBOX，WS\_MAXMIZEBOX风格的层叠窗口。与WS\_OVERLAPPEDWINDOW风格相同。 　　  
WS\_VISIBLE创建一个初始状态为可见的窗口。 　　  
WS\_VSCROLL：创建一个有垂直滚动条的窗口。

**第二十二个GetMessage获取窗口消息**

参照CreateWindow函数例子，以后的例子可能是在控制台下，也可能是Win32 Application，大家以后根据主函数判断该建什么工程。

**第二十三个RegisterClass注册窗口类，参照CreateWindow**

**第二十四个UpdateWindow参照CreateWindow**

**第二十五个DispatchMessage参照CreateWindow**

**第二十六个LoadCursorFromFile从磁盘加载一个光标文件，函数返回该光标句柄**

假设e盘下有一个名为a.cur的光标文件。

HCURSOR cursor //定义一个光标句柄，用于存放LoadCursorFromFile函数返回的光标句柄  
 cursor=LoadCursorFromFile("e:\\a.cur");

获得了光标句柄有什么用呢？看一下窗口类WNDCLASS里的hCursor成员，这个成员也是一个光标句柄，明白了吧！

**第二十七个CreateSolidBrush创建一个画刷，函数返回画刷句柄**

HBRUSH hbr=CreateSolidBrush(RGB(12,172,59));//三个数字分别表明RGB的颜色值，RGB根据三种颜色值返回一个COLORREF类型的值

**第二十八个LoadImage装载位图、图标、光标函数**

函数定义：HANDLE LoadImage(HINSTANCE hinst,LPCTSTR lpszName,UINT uType,int cxDesired,int CyDesired,UINT fuLoad)

这里我们只要这个函数的几个简单功能：从磁盘加载位图，从磁盘加载图标，从磁盘加载光标。所以第一个参数hinst我们不用管它，直接填NULL就行，第二个参数lpszName是图片文件所在路径名，第三个参数uType指明要加载的是什么类型的图片，

是位图（填IMAGE\_BITMAP），还是光标（填IMAGE\_CURSOR），还是图标（填IMAGE\_ICON）。第四个cxDesired和第五个参数CyDesired,指定要加载的图片的宽高（可以放大光标，或者缩小），如果加载的是位图的话，则两个参数必须为0，第六个参数fuLoad表示以何种方式加载文件，这里我们是从磁盘加载文件，所以填LR\_LOADFROMFILE;

好了，假设e盘下有一个c.cur和i.ico文件。例子：设置窗口图标和光标，还有背景色

#include "stdafx.h" //这个头文件是编译器自动生成的，不是空工程，都会有，  
 //如果是直接建C++源文件，包含这个头文件，会出错  
 #include <windows.h>   
 #include <stdio.h>   
 LRESULT CALLBACK WinSunProc(   
 HWND hwnd, // handle to window   
 UINT uMsg, // message identifier   
 WPARAM wParam, // first message parameter   
 LPARAM lParam // second message parameter   
 ); //窗口过程函数声明  
 int WINAPI WinMain(   
 HINSTANCE hInstance, // handle to current instance   
 HINSTANCE hPrevInstance, // handle to previous instance   
 LPSTR lpCmdLine, // command line   
 int nCmdShow // show state   
 )   
 {   
 //设计一个窗口类   
 WNDCLASS wndcls;   
 wndcls.cbClsExtra=0;   
 wndcls.cbWndExtra=0;   
 wndcls.hbrBackground=CreateSolidBrush(RGB(12,172,59)); //画刷  
 wndcls.hCursor=(HCURSOR)LoadImage(NULL,"e:\\c.cur",IMAGE\_CURSOR,24,24,LR\_LOADFROMFILE); //加载光标  
 wndcls.hIcon=(HICON)LoadImage(NULL,"e:\\i.ico",IMAGE\_ICON,48,48,LR\_LOADFROMFILE); //加载图标  
 wndcls.hInstance=hInstance; //应用程序实例句柄由WinMain函数传进来   
 wndcls.lpfnWndProc=WinSunProc; //定义窗口处理函数  
 wndcls.lpszClassName="windowclass";   
 wndcls.lpszMenuName=NULL;   
 wndcls.style=CS\_HREDRAW | CS\_VREDRAW;   
 RegisterClass(&wndcls);   
   
 //创建窗口，定义一个变量用来保存成功创建窗口后返回的句柄   
 HWND hwnd;   
 hwnd=CreateWindow("windowclass","first window",   
 WS\_OVERLAPPEDWINDOW,0,0,600,400,NULL,NULL,hInstance,NULL);  
 //显示及刷新窗口   
 ShowWindow(hwnd,SW\_SHOWNORMAL);   
 UpdateWindow(hwnd);  
 //定义消息结构体，开始消息循环   
 MSG msg;   
 while(GetMessage(&msg,NULL,0,0))   
 {   
 TranslateMessage(&msg);   
 DispatchMessage(&msg);   
 }   
 return msg.wParam;   
 }

//编写窗口过程函数   
 LRESULT CALLBACK WinSunProc(   
 HWND hwnd, // handle to window   
 UINT uMsg, // message identifier   
 WPARAM wParam, // first message parameter   
 LPARAM lParam // second message parameter   
 )   
 {   
 switch(uMsg)   
 {   
 case WM\_CHAR: //字符消息  
 char szChar[20];   
 sprintf(szChar,"char code is %c",wParam);   
 MessageBox(hwnd,szChar,"char",0);   
 break;   
 case WM\_LBUTTONDOWN: //鼠标左键按下消息  
 MessageBox(hwnd,"mouse clicked","message",0);   
 break;   
 case WM\_CLOSE:   
 if(IDYES==MessageBox(hwnd,"是否真的结束？","message",MB\_YESNO))  
 {   
 DestroyWindow(hwnd); //销毁窗口，并发送WM\_DESTROY消息给自身窗口  
 }   
 break;   
 case WM\_DESTROY:  
 PostQuitMessage(0);   
 break;   
 default:   
 return DefWindowProc(hwnd,uMsg,wParam,lParam);   
 }   
 return 0;   
 }

**第二十九个GetDC根据窗口句柄获取设备上下文（DC）返回DC句柄**

得到了一个窗口的设备上下文，就可以进行画图操作了，像画圆，画正方形，显示图片等函数都是要设备上下文(DC）句柄做参数的。

HDC dc //定义一个DC句柄  
  
 HWND wnd=FindWindow(NULL,"无标题.txt - 记事本"); //获取窗口句柄  
 dc=GetDC(wnd) //获取这个窗口的设备上下文

**第三十个Rectnagle在窗口中画一个矩形**

以"无标题.txt - 记事本"窗口为例，在这个窗口简单的画一个矩形

#include<windows.h>  
 void main()  
 {  
 HDC dc;  
 HWND wnd=FindWindow(NULL,"无标题.txt - 记事本");  
 dc=GetDC(wnd); //获取窗口设备上下文（DC）  
 while(1) //用循环语句重复画，是为了确保不会被窗口刷新给刷掉  
 {  
 Rectangle(dc,50,50,200,200); //画一个矩形  
 Sleep(200);  
 }  
 }

**第三十个CreateToolhelp32Snapshot给当前进程拍一个照**

HANDLE hProcessSnap=::CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS,0);  
 //记住这种格式就行了，返回的句柄，存储有进程信息，可以用Process32Firs函数找出来。

**第三十一个Process32First根据CreateToolhelp32Snapshot函数返回的句柄获取进程信息**

结合Process32Next函数使用，有点像文件寻找函数。

看完整例子：显示系统进程名，以及进程ID号

#include<windows.h>  
 #include<tlhelp32.h> //声明快照函数的头文件  
 #include<stdio.h>  
 int main()  
 {  
 PROCESSENTRY32 pe32; //进程的信息将会存储在这个结构里  
 //在使用这个结构之前，先设置它的大小  
 pe32.dwSize=sizeof(pe32);  
 //给系统内的所有进程拍一个快照  
 HANDLE hProcessSnap=::CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS,0);  
 BOOL bMore=::Process32First(hProcessSnap,&pe32); //第一次查找  
 while(bMore)  
 {  
 printf("进程名称:%s\n",pe32.szExeFile); //szExeFile是进程名  
 printf("进程ID号:%u\n\n",pe32.th32ProcessID); //th32ProcessID是进程ID号  
 bMore=::Process32Next(hProcessSnap,&pe32); //寻找下个进程，函数返回0，则没有进程可寻  
 }  
 return 0;  
 }

**第三十二个OpenProcess根据进程ID号获得进程句柄，句柄通过函数返回**

函数定义：HANDLE OpenProcess( DWORD dwDesiredAccess, BOOL bInheritHandle, DWORD dwProcessId);

第一个参数不要管它，填PROCESS\_ALL\_ACCESS，第二个参数也一样，填FALSE，那最后一个参数就是进程ID号。

**第三十三个TerminateProcess结束一个进程（需进程句柄做参数）**

该函数只有两个参数，第一个是进程句柄，第二个填0就行了。

现在给个例子：假设当前有一个进程名为abc.exe的进程正在运行，编一个程序结束它。

#include<windows.h>  
 #include<tlhelp32.h> //声明快照函数的头文件  
 int main(int argc,char \*argv[])  
 {   
 PROCESSENTRY32 pe32;  
 //在使用这个结构之前，先设置它的大小  
 pe32.dwSize=sizeof(pe32);  
 //给系统内的所有进程拍一个快照  
 HANDLE hProcessSnap=::CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS,0);  
 //遍历进程快照，轮流显示每个进程的信息  
 BOOL bMore=::Process32First(hProcessSnap,&pe32);  
 while(bMore)  
 {  
 if(strcmp("abc.exe",pe32.szExeFile)==0) //如果找到进程名为abc.exe  
 {  
 HANDLE hProcess=OpenProcess(PROCESS\_ALL\_ACCESS,FALSE,pe32.th32ProcessID); //获取句柄  
 ::TerminateProcess(hProcess,0); //结束它  
 }  
 bMore=::Process32Next(hProcessSnap,&pe32); //寻找下一个  
 }  
 return 0;  
 }

上面的这个例子，只能结束普通权限进程，如果为系统进程的话，则没有用，结束不了。在后面的提升权限函数，会有例子说明如何结束系统进程。

**第三十四个CreatePen创建一个画笔（返回画笔句柄）**

函数定义：BOOL CreatePen(int nPenStyle, int nWidth, COLORREF crColor);

第一个参数，表示是什么类型的线，取值有以下：

如创建一个画笔：HPEN pen=CreatePen(PS\_SOLID,3,RGB(255,78,99));

PS\_SOLID  画笔画出的是实线   PS\_DASH 画笔画出的是虚线（nWidth必须是1） PS\_DOT 画笔画出的是点线（nWidth必须是1）  
PS\_DASHDOT 画笔画出的是点划线（nWidth必须是1） PS\_DASHDOTDOT 画笔画出的是点-点-划线（nWidth必须是1）  
第二个参数是画笔的宽度，第三个参数是画笔的颜色，COLORREF类型可以RGB来获得如RGB(233,128,88);分别是红绿蓝。

**第三十五个CreateSolidBrush创建一个画刷**

只有一个COLORREF类型的参数

HBRUSH brush=CreateSolidBrush(RGB(22,182,111));

**第三十六个SelectObject把GDI对象选入相应的DC中**

像画笔(句柄HPEN），画刷（HBURSH），位图（HBITMAP）等都是GID对象。因为画图函数，如画圆，画矩形，画直线，它们所画出图形，默认属性都是不变的，如线的宽度。那么想要改变画出来时线的宽度，比如我想画出来的图形它的线条宽度为5（像素），那么就要创建一个宽度为5的画笔，然后再通过SelectObject函数，给这个画笔选入，就可以了.

接下举个例子:SelectObject应用

#include "stdafx.h"  
 #include<windows.h>  
 LRESULT CALLBACK WinSunProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)  
 {  
 static HPEN pen=CreatePen(PS\_SOLID,3,RGB(255,78,99)); //创建画笔  
 static HBRUSH brush=CreateSolidBrush(RGB(22,182,111)); //创建画刷  
 if(uMsg==WM\_PAINT) //窗口需要重画的时候  
 {  
 HDC hDC;   
 PAINTSTRUCT ps;   
 hDC=BeginPaint(hwnd,&ps); //BeginPaint只能在响应WM\_PAINT,不能用GetDC获取设备上下文  
 SelectObject(hDC,pen); //选入画笔  
 SelectObject(hDC,brush); //选入画刷  
 Rectangle(hDC,100,100,200,200);  
 EndPaint(hwnd,&ps);   
 }  
 else if(uMsg==WM\_CLOSE) //用户关闭了窗口  
 DestroyWindow(hwnd); //销毁窗口，并发送WM\_DESTROY消息  
 else if(uMsg==WM\_DESTROY) //如果窗口被销毁  
 PostQuitMessage(0); //让进程退出  
 return DefWindowProc(hwnd,uMsg,wParam,lParam); //未处理的消息通过DefWindowProc函数交给系统处理  
 }  
 int APIENTRY WinMain(HINSTANCE hInstance,  
 HINSTANCE hPrevInstance,  
 LPSTR lpCmdLine,  
 int nCmdShow)  
 {  
 WNDCLASS wndcls; //定义一个存储窗口信息WNDCLASS变量  
 wndcls.cbClsExtra=0; //默认为0  
 wndcls.cbWndExtra=0; //默认为0  
 wndcls.hbrBackground=(HBRUSH)GetStockObject(GRAY\_BRUSH); //背景画刷  
 wndcls.hCursor=LoadCursor(NULL,IDC\_ARROW); //光标  
 wndcls.hIcon=LoadIcon(NULL,IDI\_ERROR); //窗口图标  
 wndcls.hInstance=hInstance; //应用程序实例句柄由WinMain函数传进来   
 wndcls.lpfnWndProc=WinSunProc; //窗口消息处理函数  
 wndcls.lpszClassName="windowclass"; //窗口类名  
 wndcls.lpszMenuName=NULL; //窗口菜单名，没有菜单，为NULL  
 wndcls.style=CS\_HREDRAW | CS\_VREDRAW; //窗口类型，CS\_HREDRAW和CS\_VERDRAW 表明  
 //当窗口水平方向垂直方向的宽度变化时重绘整个窗口  
 RegisterClass(&wndcls); //把窗口信息提交给系统，注册窗口类  
 HWND hwnd; //用以存储CreateWindow函数所创建的窗口句柄  
 hwnd=CreateWindow("windowclass","first windows",   
 WS\_OVERLAPPEDWINDOW,0,0,600,400,NULL,NULL,hInstance,NULL); //创建窗口  
 ShowWindow(hwnd,SW\_SHOWNORMAL); //窗口创建完了，显示它  
 UpdateWindow(hwnd); //更新窗口，让窗口毫无延迟的显示  
 MSG msg; //消息结构类型  
 while(GetMessage(&msg,NULL,0,0)) //获取消息  
 {  
 //TranslateMessage(&msg); //此函数用于把键盘消息(WM\_KEYDOWN,WM\_KEYUP)转换成字符消息WM\_CHAR  
 DispatchMessage(&msg); //这个函数调用窗口过程处理函数，并把MSG里的信息处理后传给过程函数的四个参数  
 }  
 return 0;  
 }

**第三十七个 ReadProcessMemory根据进程句柄读取相应的一段内存（读其它进程里的内存）**

函数定义：BOOL ReadProcessMemory(HANDLE hProcess,PVOID pvAddressRemote,PVOID pvBufferLocal,DWORD dwSize,

PDWORD pdwNumBytesRead);总共四个参数

第一个参数hProcess是远程进程句柄，被读取者 。第二个pvAddressRemote是远程进程中内存地址。 从具体何处读取

pvBufferLocal是本地进程中内存地址. 函数将读取的内容写入此处 ，dwSize是要读取的字节数。要读取多少

pdwNumBytesRead是实际读取的内容（函数执行后，实际读了多少字节，将存储在该变量里）

远程进程的内存地址是什么意思呢，比如我现在定义一个变量a,int a;就是了，大家知道int型是占四个字节的，也就是说如果a变量所占的内存起始地址是0x1234,那么变量a就占用0x1234,0x1235,0x1236,0x1237这四个字节，这四个字节的内容决定了a变量的值。

好了知道了这个，我们就来举个例子，读取另一个进程里一个变量的值：需设计两个程序，一个用于读（Read)一个用于被读(BeRead);

那么要如何获得另一个进程中一个变量的地址呢?这里我们用一个简单的方法，让另一个进程自己去获取，然后输出地址值。

被读的程序代码如下：假设该进程名为：BeRead.exe

#include<stdio.h>  
 int main()  
 {  
 int a=10; //要读取的变量。  
 printf("%x\n",&a); //输出这个变量的起始地址，假设输出为12ff7c  
 while(1)  
 {  
 Sleep(1000);  
 }  
 return 0;  
 }

必须先让这个程序运行，然后根据输出的地址值，才能在下面的程序填入地址值。

读取的程序代码如下：

#include<windows.h>  
 #include<stdio.h>  
 #include<tlhelp32.h>  
 int main()  
 {  
 //先要获取进程句柄，如何获取，参照TerminateProcess函数，结束一个进程  
 HANDLE ReProcess;  
 PROCESSENTRY32 pe32;  
 pe32.dwSize=sizeof(pe32);  
 HANDLE hProcessSnap=CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS,0);  
 BOOL bMore=::Process32First(hProcessSnap,&pe32);  
 while(bMore)  
 {  
 if(strcmp(pe32.szExeFile,"BeRead.exe")==0) //如果是BeRead.exe  
 {  
 ReProcess=::OpenProcess(PROCESS\_ALL\_ACCESS,FALSE,pe32.th32ProcessID); //获取该进程句柄  
 break;  
 }  
 bMore=Process32Next(hProcessSnap,&pe32);  
 }  
 int \*ReAddress=(int \*)0x12ff7c; //要读取的内存的地址值  
 int \*p=new int;  
 unsigned long size;  
 ReadProcessMemory(ReProcess,ReAddress,p,4,&size); //读取BeRead进程的内存  
 printf("%d\n",\*p); //输出读取来的值  
 return 0;  
 }

**第三十八个WriteProcessMemory根据进程句柄写入相应的一段内存（写入其它进程里的内存）**

这个函数里的参数跟ReadProcessMemory函数参数意思一样，只不过一个是写，一个是读。

下面直接举个例子，形式跟读内存函数的例子一样。

被写的程序代码如下：假设该进程名为：BeWrite.exe

#include<stdio.h>  
 int main()  
 {  
 int a=10;  
 printf("%x\n",&a); //假设输出为12ff7c  
 while(1)  
 {   
 printf("%d\n",a); //每隔一秒输出，查看值有没有改变  
 Sleep(1000);  
 }  
 return 0;  
 }  
 写入的代码如下：  
 #include<windows.h>  
 #include<stdio.h>  
 #include<tlhelp32.h>  
 int main()  
 {  
 HANDLE ReProcess;  
 PROCESSENTRY32 pe32;  
 pe32.dwSize=sizeof(pe32);  
 HANDLE hProcessSnap=CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS,0);  
 BOOL bMore=::Process32First(hProcessSnap,&pe32);  
 while(bMore)  
 {  
 if(strcmp(pe32.szExeFile,"BeWrite.exe")==0)  
 {  
 ReProcess=::OpenProcess(PROCESS\_ALL\_ACCESS,FALSE,pe32.th32ProcessID);  
 break;  
 }  
 bMore=Process32Next(hProcessSnap,&pe32);  
 }  
 int \*ReAddress=(int \*)0x12ff7c;  
 int \*p=new int;  
 \*p=300;  
 unsigned long size;  
 WriteProcessMemory(ReProcess,ReAddress,p,4,&size);  
 return 0;  
 }

**第三十九个CreateThread创建一个线程（多线程）**

线程是什么意思呢，代码是由线程来执行的，一个程序默认只有一个线程（主线程），打个比方，线程就好比一个人，而不同功能的代码或函数就好是一件件不同的事情，如洗碗，洗衣服，擦地。一个人要把这几种事情做完，可以有好几种方案，第一种就是，洗完碗，就去洗衣服，衣服洗完了，再去擦地。第二种就是：洗一分钟碗，再去洗一分钟衣服，再去擦一分钟，然后又去洗一分钟衣服.......直到做完。好了，现在你可以再创造一个人帮你做事，创造这个人后，你就叫他洗衣服，而你就洗碗，这样两件事就可以同时被做了。而这里的创造一个人指的就是CreateThread函数。

函数定义：HANDLE CreateThread(LPSECURITY\_ATTRIBUTES lpThreadAttributes,DWORD dwStackSize,LPTHREAD\_START\_ROUTINE lpStartAddress,LPVOID lpParameter,DWORD dwCreationFlags,LPDWORD lpThreadId);

该函数有六个参数，第一个参数不用管它，填NULL，第二个参数dwStackSize用于新线程的初始堆栈大小，默认为0，第三个lpStartAddress填函数名（指标），但这个函数必须是这种固定格式的DWORD \_stdcall ThreadProc(LPVOID lpParameter),新的线程将会执行这个函数里面的代码，直到函数结束，线程死亡。第四个lpParameter是一自定义参数，用户可以通过这个参数，传递需要的类型，这个参数与线程函数的参数相对应。第五个dwCreationFlags填0表示立即执行，如果是CREATE\_SUSPENDED表示挂起，直到用ResumeThread函数唤醒。第六个lpThreadId填NULL就行了。

现举个例子，两个线程同时每隔一秒输出一个数字，也就是一秒会有两数字输出。

#include<windows.h>  
 #include<stdio.h>  
 DWORD \_stdcall ThreadProc(LPVOID lpParameter) //线程执行函数  
 {  
 int si=100;  
 while(si>0)  
 {  
 printf("子线程输出数字:%d\n",si--);  
 Sleep(1000);  
 }  
 return 0;  
 }  
 int main()  
 {  
 int mi=0;  
 CreateThread(NULL,0,ThreadProc,NULL,0,NULL); //创建一个线程，去执行ThreadProc函数  
 while(mi<100)  
 {  
 printf("主线程输出数字:%d\n",mi++);  
 Sleep(1000);  
 }  
 return 0;  
 }

**第四十个GetCurrentProcessId获得当前进程ID**

DWORD currentPID;  
 currentPID=::GetCurrentProcessId(); //返回进程ID号  
 cout<<currentPID<<endl;

**第四十一个CreateCompatibleDC创建一个兼容的内存设备上下文（DC）**

简单的来说，就是复制一个模一样的DC。就把窗口看成一幅幅图画，窗口有大有小，里面的内容也不一样（颜色值），每个像素点的颜色值可能不一样，所以就用设备上下文来描述每个窗口的信息，对于DC具体是怎样描述设备上下文的，我们暂时还不需要知道，只要了解这个概念就行了。这个窗口信息，获得一个窗口设备上下文，就用GetDC函数就行了，如HDC hDC=GetDC(hWnd);而CreateCompatibleDC的作用是根据一个设备上下文，再创建一个兼容的设备上下文，如 HDC mDC=CreateCompatibleDC(hDC)。这样mDC里的信息就跟hDC里的一样，那这有什么用呢？这个将会在后面的BitBltl输出一个位图（合并两个DC）函数里会用到。

**第四十二个GetObject获取一个对象信息（如位图，图标，光标）**

函数定义：int GetObject(HGDIOBJ hgdiobj, int cbBuffer, LPVOID lpvObject)；

第一个参数hgdiobj是对象句柄，第二个参数cbBuffer是待写入lpvObject指针指向缓存区数据大小，第三个参数lpvObject是一个指针，指向一个缓存区。

这里举一个获取位图的信息，获取位图的大小，假设E盘下有一个aa.bmp的位图文件，输出位图的宽高

#include<windows.h>  
 #include<stdio.h>  
 int main()  
 {  
 BITMAP bmInfo; //这个结构存储位图信息  
 HBITMAP bmp;  
 bmp=(HBITMAP)LoadImage(NULL,"e:\\aa.bmp",IMAGE\_BITMAP,0,0,LR\_LOADFROMFILE);  
 GetObject(bmp,sizeof(BITMAP),&bmInfo);  
 printf("位图宽：%d,位图高：%d\n",bmInfo.bmWidth,bmInfo.bmHeight);  
 return 0;  
 }

**第四十三个BitBlt在窗口输出一个位图**

其实倒不如说这个BitBlt函数是拷贝一个设备上下文（DC），或者合并两个窗口，再延伸一下，合并两个图片？也并无不可，往大了说，窗口难道不是图片吗？用截屏软件，把窗口截成图片，这样窗口便成了图片。可能有点瞎说，大家还是按照标准来吧，反正，你只要掌握这个函数就行了，而且这个概念也不会有什么影响，那就足够了。

BitBlt的作用跟把两幅图片合在一起一样，合并两幅图片。可能两幅图片大小不一样也可以合并，但合并DC就不行了，必须两个信息一样的DC才可以合并，那要如何确保两个DC一样呢？这就要用到CreateCompatibleDC函数了。

函数定义：BOOL BitBlt(HDC hdcDest,int nXDest,int nYDest,int nWidth,int nHeight,HDC hdcSrc,int nXSrc,int nYSrc,DWORD dwRop)；

第一个参数hdcDest是原DC句柄，被覆盖的DC，nXdest,nYDest,nWidth,nHeight这四个参数，指明了一个矩形，覆盖原DC哪块区域。

第六个参数hdcSrc是覆盖的DC句柄，nXSrc,nYSrc参数指明从哪里开始覆盖。（覆盖DC的左上角），第九个参数dwPop表示以何种方式覆盖。因为这里我们只要输出一个位图，所以用SRCCOPY,直接覆盖。

好了，直接举个例子，在窗口输出一副图片，假设e盘下有一个aa.bmp的位图。为了方便，我们直接在记事本窗口输出位图，先运行一个窗口名为"无标题.txt - 记事本"记事本窗口程序。

#include<windows.h>  
 #include<stdio.h>  
 int main()  
 {  
 BITMAP bmInfo; //这个结构存储位图信息  
 HBITMAP bmp;  
 bmp=(HBITMAP)LoadImage(NULL,"e:\\aa.bmp",IMAGE\_BITMAP,0,0,LR\_LOADFROMFILE);  
 GetObject(bmp,sizeof(BITMAP),&bmInfo); //获取位图信息  
 HWND wnd=FindWindow(NULL,"无标题.txt - 记事本");  
 HDC hdc=GetDC(wnd);  
 HDC memDC=::CreateCompatibleDC(hdc); //创造兼容的DC  
 SelectObject(memDC,bmp); //选入位图  
 while(1)  
 {  
 BitBlt(hdc,0,0,bmInfo.bmWidth,bmInfo.bmHeight,memDC,0,0,SRCCOPY); //输出位图  
 Sleep(200);  
   
 }  
 return 0;  
 }

下面介绍一下BitBlt函数最后一个参数的常用取值及意思。

参考（百度）  
BLACKNESS：表示使用与物理调色板的索引0相关的色彩来填充目标矩形区域，（对缺省的物理调色板而言，该颜色为黑色）。 　　  
DSTINVERT：表示使目标矩形区域颜色取反。 　　  
MERGECOPY：表示使用布尔型的AND（与）操作符将源矩形区域的颜色与特定模式组合一起。 　　MERGEPAINT：通过使用布尔型的OR（或）操作符将反向的源矩形区域的颜色与目标矩形区域的颜色合并。 NOTSRCCOPY：将源矩形区域颜色取反，于拷贝到目标矩形区域。 　　  
NOTSRCERASE：使用布尔类型的OR（或）操作符组合源和目标矩形区域的颜色值，然后将合成的颜色取反。 PATCOPY：将特定的模式拷贝到目标位图上。 　　  
PATPAINT：通过使用布尔OR（或）操作符将源矩形区域取反后的颜色值与特定模式的颜色合并。然后使用OR（或）操作符将该操作的结果与目标矩形区域内的颜色合并。 　　  
PATINVERT：通过使用XOR（异或）操作符将源和目标矩形区域内的颜色合并。 　　  
SRCAND：通过使用AND（与）操作符来将源和目标矩形区域内的颜色合并。 　　  
SRCCOPY：将源矩形区域直接拷贝到目标矩形区域。 　　  
SRCERASE：通过使用AND（与）操作符将目标矩形区域颜色取反后与源矩形区域的颜色值合并。 　　SRCINVERT：通过使用布尔型的XOR（异或）操作符将源和目标矩形区域的颜色合并。 　　  
SRCPAINT：通过使用布尔型的OR（或）操作符将源和目标矩形区域的颜色合并。 　　  
WHITENESS：使用与物理调色板中索引1有关的颜色填充目标矩形区域。（对于缺省物理调色板来说，这个颜色就是白色）

**第四十四个GetWindowText根据窗口句柄获得窗口标题名**

函数定义：int GetWindowText(HWND hWnd,LPTSTR lpString,int nMaxCount);

第一个参数hWnd是要获取窗口标题名的窗口句柄，第二个lpString是个字符串，窗口标题名，将会存储在这里面，第三个参数nMaxCount指明了第二个参数字符数组的大小。

下面结合GetCursorPos和WindowFromPoint举个例子，鼠标指向哪个窗口，就在界面显示那窗口的标题名

#include<windows.h>  
 #include<stdio.h>  
 int main()  
 {  
 char Text[256]={0};  
 HWND wnd;  
 POINT curpos;  
 while(1)  
 {  
 GetCursorPos(&curpos);  
 wnd = WindowFromPoint(curpos);  
 GetWindowText(wnd,Text,256);  
 printf("%s\n",Text);  
 Sleep(300);  
 }  
 return 0;  
 }

**第四十五个SetWindowText根据窗口句柄设置窗口标题名**

这个函数有两个参数，一个是窗口句柄，一个是标题名，这里就不需要解释了吧，直接看例子，设置一个窗口标题名，依旧以

"无标题.txt - 记事本"为例。

#include<windows.h>  
 #include<stdio.h>  
 int main(int argc, char\* argv[])  
 {  
 HWND wnd;  
 wnd=FindWindow(NULL,"无标题.txt - 记事本"); //获取窗口句柄  
 SetWindowText(wnd,"新的窗口标题"); //设置窗口标题名  
 return 0;  
 }

**第四十六个GetCurrentProcess获得当前线程句柄**

没有参数，直接调用即可，该函数返回线程句柄

**第四十七个OpenProcessToken获得一个进程的访问令牌句柄**

获得一个进程的访问令牌有什么用呢？主要是为了修改它的权限，前面在介绍结束一个进程的时候说过了，无法结束系统进程，是什么原因呢，原因是调用OpenProcess函数失败，无法获取系统进程句柄而引起的，那为什么会失败呢，权限不够，普通程序的进程没有SeDeDebug权限，而一个进程的权限是与访问令牌相关的，这样我们只要获取一个进程的访问令牌句柄，再以这个句柄为参数调用相应的函数提升进程的权限为SeDeDebug就可以获取系统进程句柄，进而结束它。

函数定义：BOOL OpenProcessToken(HANDLE ProcessHandle,DWORD DesiredAccess,PHANDLE TokenHandle)

第一个参数ProcessHandle待获取的进程句柄，第二个参数DesiredAccess操作类型，填TOKEN\_ADJUST\_PRIVILEGES就行了，

第三个TokenHandle是访问令牌句柄的指针，该参数接收句柄。

如获得本进程的访问令牌句柄：HANDLE hToken;

OpenProcessToken(GetCurrentProcess(),TOKEN\_ADJUST\_PRIVILEGES,&hToken);

**第四十七个LookupPrivilegeValue函数查看对应系统权限的特权值，返回信息到一个LUID结构体里**  
上面讲过了，进程有权限一说，那么大家也能猜到，进程权限的信息也一定存储在一个结构体里，这个结构体描述了进程权限相关的一些信息。这个结构体在这里就不具体描述了，我们所要做的，只是把一个进程权限设置成SeDeDebug就行了，所以我们只要知道TOKEN\_PRIVILEGES便是描述进程权限的结构体就可以了。而LookupPrivilegeValue函数是根据访问令牌句获取相应的权限信息吗？  
不是的。TOKEN\_PRIVILEGES结构里的Privileges[0].Luid跟这个函数所查询的东西相对应，也就是说，如果进程是SeDeDeBug权限，那Privileges[0].Luid的取值是怎样的呢？用LookupPrivilegeValue函数便可以获取其取值。  
这个函数是这样定义的：BOOL LookupPrivilegeValue(LPCTSTR lpSystemName,LPCTSTR lpName,PLUID lpLuid);  
第一个参数lpSystemName通常都填NULL，本地系统调用，第二个参数lpName填要查询的权限名，如要查询的是SeDeDebug权限则取值是SE\_DEBUG\_NAME，第三个参数lpLuid接收其取值。  
如LUID luid;LookupPrivilegeValue(NULL,SE\_DEBUG\_NAME,&luid);

**第四十八个AdjustTokenPrivileges调整一个进程的访问令牌信息（权限）**

函数定义：BOOL AdjustTokenPrivileges(HANDLE TokenHandle,BOOL DisableAllPrivileges,PTOKEN\_PRIVILEGES NewState,DWORD BufferLength,PTOKEN\_PRIVILEGES PreviousState,PDWORD ReturnLength)

第一个参数TokenHandle是令牌句柄，第二个是禁用所有权限标志，后面填FALSE就行了。第三个NewState是待刷进令牌句柄的PTOKEN\_PRIVILEGES结构信息指针，第四个BufferLength指明TOKEN\_PRIVILEGES结构大小，第五，六个参数填NULL就行了。

那么结束上面两个函数，提升一个进程权限制，让它能够结束系统进程的代码就是：

HANDLE hToken;  
 OpenProcessToken( GetCurrentProcess(), TOKEN\_ADJUST\_PRIVILEGES, &hToken );  
 TOKEN\_PRIVILEGES tp;  
 LookupPrivilegeValue( NULL, SE\_DEBUG\_NAME, &tp.Privileges[0].Luid );  
 tp.PrivilegeCount = 1; //tp里其它一些属性设置  
 tp.Privileges[0].Attributes = SE\_PRIVILEGE\_ENABLED;  
 AdjustTokenPrivileges( hToken, FALSE, &tp, sizeof( TOKEN\_PRIVILEGES ), NULL, NULL );

只上把上面的代码，加入结束普通进程例子的前面，那么就能结束系统进程了。

**第四十九个LoadLibrary加载动态链接库，返回动态链接库模块句柄**

该函数只有一个参数，那就是动态链接库的名称，如user32.dll，函数返回HMOUDLE类型的模块句柄，获得了一个动态链接库的模块句柄，就可以调用GetProcAddress函数获得模块里面的函数地址，从而调用动态链接库里的函数。

**第五十个GetProcAddress根据模块句柄获取相应的函数地址**

提到GetProcAddress函数，不得不讲一下怎么设计一个动态链接库，这里我们就以自己设计动态链接库作为GetProcAddress函数的例子。

动态链接库里的函数相对于头文件的函数有什么优势呢？更节省内存，相对于比较常用的函数而已。如果在一个程序里，调用一个头文件里的函数的话，那不管如何，函数的代码就会被复制一份到当前程序里，所以，当有十几个程序调用同一个函数的时候，这个函数在内存中所占用的空间，就会有十几份，分别在各自调用的进程内存空间里，而动态链接库的函数，只在内存中有一份空间（公用空间）如果哪个程序要用到这个函数的话，只要给这个函数的地址，进程就可以跑到这个空间执行函数，那么如何获取函数地址呢，用GetProcAddress函数就行了。

下面我们就自己设计一个动态链接库，点“文件->新建->工程",然后选中“Win32 Dynamic-Link Library”,再在右边给工程取一个名，点确定。接着弹出了一个对话框，询问希望创建什么类型，我们选择第二个“一个简单的DLL工程”，点完成->确定.然后单击右边的“+”号，很小的一个，接着下面会出现一个Globals的"+"号，单击该加号，然后再双击DllMain函数，进入代码编辑区，在这里编写代码，这里已经有了一些代码了，编译器自动生成的。那个DllMain函数，便是动态链接库的主函数。在程序进程加载动态链接的时候，进程会自动调用DllMain函数，也就是说会自动执行DllMain函数里的代码，也就是说，如果哪程序执行了这个语句“LoadLibrar("user32.dll")",那么执行这个语句的进程，便会自动执行user32.dll里的DllMain函数。如果是主线程加载动态库的话，那么该DllMain函数里的代码会被执行两次，分别是加载的时候执行一次，调用FreeLibrary函数释放或程序结束自动释放动态链接库的时候执行一次，至于是什么原因导致DllMain函数被调用，DllMain函数的第二个参数ul\_reason\_for\_call说明了原因，它有四个取值，代表了四个原因。分别是：

DLL\_PROCESS\_ATTACH（进程加载），DLL\_THREAD\_ATTACH （线程加载）

DLL\_THREAD\_DETACH（线程释放），DLL\_PROCESS\_DETACH（进程释放）

因为这里我们只要设计一个动态链接函数，所以便不用管DllMain函数，DllMain函数将会在介绍CreateRemoteThread（创建一个远程线程）函数的时候讲到，所以我们只要在DllMain函数外定义一个函数就行了。

那么在动态链接库是如何定义函数呢？如果函数不需要导出的话，则跟普通函数定义没什么两样，导出是什么意思，就是可以用GetProcAddress函数获取地址的函数。那导出的函数要如何定义呢？  
只要在函数前面加上extern "C" \_\_declspec(dllexport)就行了，声明导出函数，防止函数重命名。那么接下来就举个例子。

动态链接里的代码：

#include "stdafx.h"  
 BOOL APIENTRY DllMain( HANDLE hModule,  
 DWORD ul\_reason\_for\_call,  
 LPVOID lpReserved  
 )  
 {  
 return TRUE;  
 }  
 extern "C" \_\_declspec(dllexport) int Add(int a,int b)  
 {  
 return a+b;  
 }

点编译执行，然后就会弹出一个调试对话框，直接点取消，接着便生成了动态链接库DLL，然后到你的工程里把后缀名为dll的文件找到，

位置在MyProject\"你的工程名"\Debug下。接着把这个文件复制到要调用的工程下，或者直接复制C:\windows\system32目录下。

假设这个文件名为"sss.dll",那么要调用里面的Add函数便是如下代码：

HMODULE hmod=::LoadLibrary("sss.dll"); //获取sss.dll的模块，加载sss.dll动态链接库  
 typedef int (\*pAdd)(int a,int b); //定义一个对应的函数型，以便识别  
 pAdd add=(pAdd)GetProcAddress(hmod,"Add"); //获取hmod模块里的Add函数地址  
 int a=add(3,5); //调用模块里的Add函数

**第五十一个SetWindowsHookEx安装一个钩子**

WINDOWS是基于消息的系统，鼠标移动，单击，键盘按键，窗口关闭等都会产生相应的消息，那么钩子是什么意思呢，它可以监控一个消息，比如在一个窗口里单击了一下，首先获得这个消息的，不是应用程序，而是系统，系统获取这个消息后，就去查看这个消息是在哪个窗口产生的，找到窗口后，再把消息投递到相应程序里的消息队列里，这之间有一个传递过程，那么钩子的作用就是在消息到达应用程序之前截获它，钩子可以关联一个函数（钩子处理函数），也就是说，如果对一个进程安装了一个钩子，进程再接收到相应在消息之前，会先去执行钩子所关联的函数，

先来看一下这个函数定义：

HHOOK WINAPI SetWindowsHookEx(int idHook,HOOKPROC lpfn,HINSTANCE hmod,DWORD dwThreadId)

第一个参数idHook指明要安装的钩子类型，如WH\_KEYBOARD(键盘钩子），WH\_MOUSE(鼠标钩子），第二个参数是钩子处理函数的地址，该函数必须是这种固定的格式：LRESULT WINAPI HookProc(int nCode,WPARAM wParam,LPARAM lParam)

第三个参数hmod是钩子函数所在模块的句柄，第四个参数dwThreadId是线程ID，待监视消息的ID，如果为0，则为全局钩子，监视所有消息

好，接下来我们举一个例子，钩子类型为WH\_KEYBOARD,全局钩子。截获键盘按键消息，并扔掉该消息，让键盘失灵。

由于是装的是全局钩子，所以钩子处理函数必须放在动态链接库里。那么我们就设计一个动态链接库吧。

现给出动态链接库的所有代码：(KeyDll.dll)

#include "stdafx.h"  
 #include<windows.h>  
 BOOL APIENTRY DllMain( HANDLE hModule,  
 DWORD ul\_reason\_for\_call,  
 LPVOID lpReserved  
 )  
 {  
 return TRUE;  
 }  
 HMODULE WINAPI ModuleFromAddress(PVOID pv) //该函数根据内存地址，获得其所在的模块句柄  
 {  
 MEMORY\_BASIC\_INFORMATION mbi;  
 VirtualQuery(pv,&mbi,sizeof(mbi));  
 return (HMODULE)mbi.AllocationBase;  
 }  
 LRESULT CALLBACK HookKey(int nCode,WPARAM wParam,LPARAM lParam)  
 {  
 return TRUE; //返回真，扔掉该消息  
 }  
 extern "C" \_\_declspec(dllexport) void SetHook(void)  
 {  
 SetWindowsHookEx(WH\_KEYBOARD,HookKey,ModuleFromAddress(HookKey),0);  
 }

生成dll文件后，把它复制到相应的目录下去。

再新建一个工程，调用用动态链接库里的函数，代码如下：

#include<windows.h>  
 int main()  
 {  
 HMODULE hMod=LoadLibrary("KeyDll.dll");  
 typedef void(\*pSetHook)(void);  
 pSetHook SetHook=(pSetHook)GetProcAddress(hMod,"SetHook");  
 SetHook();  
 while(1)  
 {  
 Sleep(1000); //避免程序结束，自动释放动态链接库  
 }  
 return 0;  
 }

这样当按下了一个键后，接收该按键消息的进程，会先去执行钩子处理函数，然后再处理消息，而钩子处理函数的几个参数说明了按键的详细信息，如按了哪个键，是按下（KEYDOWN）还是松开（KEYUP）。如果有兴趣的话，把上面那钩子处理函数的代码换成下面这个

LRESULT CALLBACK HookKey(int nCode,WPARAM wParam,LPARAM lParam)  
 {  
 char sz[25];  
 sprintf(sz,"%c",wParam); //这个函数头文件#include<stdio.h>  
 MessageBox(NULL,sz,sz,MB\_OK);  
 return FALSE;  
 }

每按下一个键，就会弹出一个提示框，并输出所按下的键，只对字符键有用。

**第五十二个SHGetFileInfo获取一个文件的各项信息（文件关联图标，属性等）**  
函数定义： DWORD SHGetFileInfo(LPCSTR pszPath, DWORD dwFileAttributes, SHFILEINFOA FAR \*psfi, UINT cbFileInfo, UINT uFlags);  
pszPath是文件的路径，dwFileAttributes一般取0，如果想要获取文件夹信息的话，则取值为FILE\_ATTRIBUTE\_DIRECTORY，psfi是一个SHFILEINFO结构的指针，该结构存储文件信息，定义如下：

typedef struct \_SHFILEINFOA  
 {  
 HICON hIcon; // 文件关联图标句柄  
 int iIcon; // 系统图标列表索引  
 DWORD dwAttributes; // 文件的属性  
 CHAR szDisplayName[MAX\_PATH]; // 文件的路径名  
 CHAR szTypeName[80]; // 文件的类型名，如是bmp文件，还是执行文件exe，或者其它  
 } SHFILEINFO;

第四个参数cbFileInfo指明SHFILEINFO结构的大小，填sizoef(SHFILEINFO);  
最后一个参数uFlags指定获取文件的什么信息，可选取值如下：（对应着SHFILEINFO里的成员）

SHGFI\_ICON; //获得图标　　  
 SHGFI\_DISPLAYNAME; //获得显示名　　  
 SHGFI\_TYPENAME; //获得类型名　　  
 SHGFI\_USEFILEATTRIBUTES; //获得属性　　  
 SHGFI\_LARGEICON; //获得大图标　　  
 SHGFI\_SMALLICON; //获得小图标　　  
 SHGFI\_PIDL; // pszPath是一个标识符

比如，我只要获取文件图标，那么参数填SHGFI\_LARGEICON就行了。如果又想获取文件关联的图标，又想获取文件类型名，那么就是  
SHGFI\_LARGEICON|SHGFI\_TYPENAME;  
函数例子：

SHFILEINFO sfi;  
 SHGetFileInfo("e:\\aa.bmp",0,&sfi,sizeof(sfi),  
 SHGFI\_ICON|SHGFI\_LARGEICON|SHGFI\_USEFILEATTRIBUTES|SHGFI\_TYPENAME);

接着可以用DrawIcon函数画出文件关联图标：该函数定义：BOOL DrawIcon（HDC hDC，int X，int Y, HICON hlcon );

**第五十三个RegCreateKeyEx在注册表里创建一个子键，或获取一个子键的句柄**

在这里我们先来了解一下注册表的基本概念，打开运行对话框，输入regedit，然后回车，便打开了注册表编辑器，首先映入眼前的，便是五个根键  
HKEY\_CLASSES\_ROOT  
HKEY\_CURRENT\_USER  
HKEY\_LOCAL\_MACHINE  
HKEY\_USER  
HKEY\_CURRENT\_CONFIG

在根键下面便是主键了，如HKEY\_CURRENT\_CONFIG根键下有两个主键，分别是Software和System（可能会不一样）,那么主键下面是什么呢，对了，就是跟 RegCreateKeyEx函数相关的子键，子键下面就是具体的键值项了，但也可以又是子键。键值有五种可选类型，分别是：字符串值（REG\_SZ)，二进制值（REG\_BINARY），DWORD值（REG\_DWORD），多字符串值（REG\_MULTI\_SZ）和可扩充字符值（REG\_EXPAND\_SZ）。键值项还有其它信息，它的名称，数据。

了解了上面这些东西，接着就来了解下RegCreateKeyEx函数的各个参数吧,先来看一下函数定义：

LONG RegCreateKeyEx (  
 HKEY hKey, //根键句柄，指明要在哪个根键下创建子键，填根键名既可  
 LPCSTR lpSubKey, //子键名，包含完整路径名  
 DWORD Reserved,. //一般取0  
 LPSTR lpClass, //一般取NULL  
 DWORD dwOptions, //创建子键时的选项,可选值REG\_OPTION\_NON\_VOLATILE，REG\_OPTION\_VOLATILE，这里取0既可  
 REGSAM samDesired, //打开方式，填KEY\_ALL\_ACCESS，在任何情况都行。  
 LPSECURITY\_ATTRIBUTES lpSecurityAttributes, //指定继承性,还是取0  
 PHKEY phkResult, //子键对应句柄，待创建或打开的子键句柄将存储在该句柄里  
 LPDWORD lpdwDisposition //打开还是创建子键,对应REG\_CREATED\_NEW\_KEY和REG\_OPENED\_EXISTING\_KEY  
 );

在这里举一个例子，以便我们能更好的理解该函数。

在HKEY\_CURRENT\_CONFIG根键下的Software主键里创建一个名为MySelf的子键。

#include<windows.h>  
 int main()  
 {  
 HKEY hroot; //子键句柄  
 DWORD dwDisposition; //对应着最后一个参数  
 RegCreateKeyEx(HKEY\_CURRENT\_CONFIG,"Software\\MySelf",0,NULL,0,KEY\_ALL\_ACCESS,NULL,&hroot,&dwDisposition);  
 return 0;  
 }

**第五十四个RegSetValueEx根据子键句柄在其下创建或修改一个键值**

函数定义：

LONG RegSetValueEx(  
 HKEY hKey, // 子键句柄  
 LPCTSTR lpValueName, // 键值名称，如果提供的子键下没有该名称，则创建  
 DWORD Reserved, // 保留，填0  
 DWORD dwType, // 键值类型，  
 CONST BYTE \*lpData, // 键值的数据  
 DWORD cbData // 键值的数据的大小  
 );

接着我们以增加开机自启动为例，来看一下函数是如何创建一个键值的，我们知道，像程序添加开机自启动一般都在

HKEY\_LOCAL\_MACHINE\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run下添加一个键值，键值类型为二进制（REG\_SZ），而键值的数据就为要自启动程序的路径。

假设e盘下有一个AutoRun.exe的应用程序，让电脑开机时自动运行它。

#include<windows.h>  
 int main()  
 {  
 HKEY hroot; //子键句柄  
 DWORD dwDisposition;  
 RegCreateKeyEx(HKEY\_LOCAL\_MACHINE,"SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run",0,  
 NULL,0,KEY\_ALL\_ACCESS,NULL,&hroot,&dwDisposition);  
 RegSetValueEx(hroot,"AutoRun",0,REG\_SZ,(BYTE \*)"e:\\AutoRun.exe",sizeof("e:\\AutoRun.exe"));  
 return 0;  
 }

**第五十五个RegDeleteValue根据子键句柄删除其下的一个键值**

这里直接举一个例子，删除RegSetValueEx函数创建的键值

#include<windows.h>  
 int main()  
 {  
 HKEY hroot; //子键句柄  
 DWORD dwDisposition;  
 RegCreateKeyEx(HKEY\_LOCAL\_MACHINE,"SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run",0,  
 NULL,0,KEY\_ALL\_ACCESS,NULL,&hroot,&dwDisposition);  
 RegDeleteValue(hroot,"AutoRun"); //删除子键下名为AutoRun的键值  
 return 0;  
 }

**第五十六个RegQueryValueEx根据子键句柄获取一个键值数据，类型。**

函数定义：

LONG RegQueryValueEx (  
 HKEY hKey, //根键句柄  
 LPCWSTR lpValueName, //键值名称  
 LPDWORD lpReserved, //预留，填0  
 LPDWORD lpType, //接收键值类型  
 LPBYTE lpData, //接收键值数据  
 LPDWORD lpcbData //接收数据的大小  
 );

例子，获取RegSetValueEx函数创建的键值的类型，数据

int main()  
 {  
 char Data[52];  
 DWORD Size,Type;  
 HKEY hroot; //子键句柄  
 DWORD dwDisposition;  
 RegCreateKeyEx(HKEY\_LOCAL\_MACHINE,"SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run",0,  
 NULL,0,KEY\_ALL\_ACCESS,NULL,&hroot,&dwDisposition); //获取根键句柄  
 RegQueryValueEx(hroot,"AutoRun",0,&Type,(BYTE \*)Data,&Size); //获取AutoRun的信息  
 printf("键值名称：AutoRun ");  
 switch(Type)  
 {  
 case REG\_SZ:printf("键值类型：REG\_SZ");break;  
 case REG\_BINARY:printf("键值类型:REG\_BINARY");break;  
 case REG\_DWORD:printf("键值类型:REG\_DWORD");break;  
 case REG\_MULTI\_SZ:printf("键值类型:REG\_MULTI\_SZ");break;  
 case REG\_EXPAND\_SZ:printf("键值类型:REG\_EXPAND");break;  
 }  
 printf(" 键值数据：%s %d\n",Data,Size);  
 return 0;  
 }

**第五十七个RegEnumValue根据子键句柄返回对应索引的键值信息（名称，数据，类型，子键下第一个键值索引为0，以此类推，函数成功执行返回ERROR\_SUCCESS）**

函数定义：

LONG RegEnumValue (  
 HKEY hKey, //子键句柄  
 DWORD dwIndex, //键值索引  
 LPWSTR lpValueName, //接收键值名称，字符数组  
 LPDWORD lpcbValueName, //指明数组大小  
 LPDWORD lpReserved, //预留，0  
 LPDWORD lpType, //键值类型，填NULL，不获取  
 LPBYTE lpData, //键值数据，填NULL，不获取  
 LPDWORD lpcbData //接收数据的大小，如果键值数据那项参数为NULL，则该项也为NULL  
 );

例子：输出Run下的所有键值名

#include<windows.h>  
 #include<stdio.h>  
 int main()  
 {  
 char Name[52];  
 int Index=0;  
 DWORD dwSize=52;  
 DWORD Size,Type;  
 HKEY hroot; //子键句柄  
 DWORD dwDisposition;  
 RegCreateKeyEx(HKEY\_LOCAL\_MACHINE,"SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run",0,  
 NULL,0,KEY\_ALL\_ACCESS,NULL,&hroot,&dwDisposition); //获取根键句柄  
 while(RegEnumValue(hroot,Index,Name,&dwSize,NULL,NULL,NULL,NULL)==ERROR\_SUCCESS)  
 {  
 printf("%s\n",Name);  
 Index++; //索引从0开始每次自增一，函数如果执行失败，则索引已到头  
 }  
 return 0;  
 }

其实也还可以扩充一下，可以像msconfig程序那样列出当前计算机的所有开机自启动程序，当然，注册表也不只就前面的那一个子键下可以添加自启动程序，在HKEY\_CURRENT\_USER\\Software\\Microsoft\\Windows\\CurrentVersion\\Run下也可以添加，所以这些子键都需要去查看，更多添加自启动程序的子键可以到百度里去搜一下，大家如果掌握前面那几个注册表操作函数，可以结合起来试着做一个可以添加，查看，删除开机自启动程序的小程序。

**第五十八个ExitWindowsEx关机，重启，注销计算机函数**

这个函数只有两个参数，后一个参数为系统预留，填0就可以了，而第一个参数则，指明关机，还是重启，或注销，可选值如下：

EWX\_LOGOFF//注销    EWX\_REBOOT//重启 NT系统中需SE\_SHUTDOWN\_NAME 特权 EWX\_SHUTDOWN//关机，需权限。

例子：关闭计算机，由于需要SE\_SHUTDOWN\_NAME权限，所以我们得先提升权限，代码如下：

#include<windows.h>  
 int main()  
 {  
 HANDLE hToken;  
 TOKEN\_PRIVILEGES tkp;  
 OpenProcessToken(GetCurrentProcess(),TOKEN\_ADJUST\_PRIVILEGES|TOKEN\_QUERY,&hToken);  
 LookupPrivilegeValue(NULL,SE\_SHUTDOWN\_NAME,&tkp.Privileges[0].Luid);  
 tkp.PrivilegeCount=1;  
 tkp.Privileges[0].Attributes=SE\_PRIVILEGE\_ENABLED;  
 AdjustTokenPrivileges(hToken,FALSE,&tkp,0,(PTOKEN\_PRIVILEGES)NULL,0);  
 ::ExitWindowsEx(EWX\_SHUTDOWN,0);  
 return 0;  
 }

**第五十九个VirtualAllocEx在其它的进程中分配内存空间**

函数定义：

LPVOID VirtualAllocEx(  
 HANDLE hProcess, //进程句柄，将会在该进程句柄相关的进程分配空间  
 LPVOID lpAddress, //默认为系统指定，填NUL  
 DWORD dwSize, //分配多大的内存  
 DWORD flAllocationType, //填MEM\_COMMIT  
 DWORD flProtect //指定分配的内存属性，为PAGE\_READWRITE，内存可读写  
 );

函数返回分配的内存首地址，

**第六十个CreateRemoteThread创建一个远程线程（在其它进程中创建线程）**

函数定义：

HANDLE WINAPI  
 CreateRemoteThread(HANDLE hProcess, //进程句柄，函数将在这个进程句柄关联的进程创建线程  
 LPSECURITY\_ATTRIBUTES lpThreadAttributes,  
 DWORD dwStackSize,  
 LPTHREAD\_START\_ROUTINE lpStartAddress,  
 LPVOID lpParameter,  
 DWORD dwCreationFlags,  
 LPDWORD lpThreadId  
 );

这个函数比CreateThread函数多了一个参数，就是这个函数的第一个hProcess（函数在该进程里创建线程），后面的六个参数跟第三十九个函数CreateThread的六个参数一样，这里就不再解释了。

例子：远程线程注入

创建一个远程线程，就必须得有一个线程函数供线程执行，而线程函数又不能在其它程序里。那要怎么办呢？大家看一下线程函数的定义，和LoadLibrary函数的定义，它们的定义相似，都是只有一个参数，而且每个程序都能调用LoadLibrary函数，这样我们便能把LoadLibrary函数作为线程函数。这样创建的线程就会去执行LoadLibrary函数。因而我们就有了一次让其它程序调用LoadLibrar函数的机会，并还可以指定LoadLibrary函数的参数（通过创建远程线程函数传递）。前面在动态链接库提到，一个程序如果调用LoadLibrary函数，它都会自动去执行相应动态链接库里的DllMain函数，所以我们自己可以编写一个动态链接库，在DllMain函数里写入想要其它程序执行的代码。再通过CreateRemoteThread函数在其它程序创建一个线程去执行LoadLibary加载我们已经编写好的动态链接库，这样就可以让其它程序执行我们的代码了。这里还有一个问题，CreateRemoteThread函数传递过去的参数，因为要供注入的那个程序访问，所以参数数据所存储的空间不能在调用CreateRemoteThread函数的程序里。必须调用VirtualAllocEx函数，在注入程序里分配一个空间，把数据（动态链接库的名称）存在里面，而新分配空间的首地址则作为CreateRemoteThread函数的参数传过去。这样注入程序访问的是自己的地址空间。

远程线程注入：

假设动态链接库为“ReCode.dll”它的代码如下：

#include<windows.h>  
 BOOL APIENTRY DllMain( HANDLE hModule,  
 DWORD ul\_reason\_for\_call,  
 LPVOID lpReserved  
 ) //DllMain函数，只要加载这个动态链接库的程序，都会跑来执行这个函数  
 { //在这里填让其它程序执行的代码  
 while(1)  
 {  
 MessageBox(NULL,"aaaa","aaaa",MB\_OK); //简单的让其它程序每隔3秒弹出一个提示框  
 Sleep(3000);  
 }  
 return TRUE;  
 }

编译运行，然后把生成的“ReCode.dll”文件复制到c:\\windows\\system23下去。

注入线程的代码：

//选择ctfmon.exe（输入法管理）作为我们要注入进线程的程序  
 #include<windows.h>  
 #include<tlhelp32.h>  
 #include<stdio.h>  
 int main()  
 {   
 char DllName[25]="ReCode.dll";  
 HANDLE hProcess; //用于存储ctfmon.exe的进程句柄  
 //先提升进程权限，使其能获取任何进程句柄，并对其进行操作  
 HANDLE hToken;  
 OpenProcessToken( GetCurrentProcess(), TOKEN\_ADJUST\_PRIVILEGES, &hToken );  
 TOKEN\_PRIVILEGES tp;  
 LookupPrivilegeValue( NULL, SE\_DEBUG\_NAME, &tp.Privileges[0].Luid );  
 tp.PrivilegeCount = 1;  
 tp.Privileges[0].Attributes = SE\_PRIVILEGE\_ENABLED;  
 AdjustTokenPrivileges( hToken, FALSE, &tp, sizeof( TOKEN\_PRIVILEGES ), NULL, NULL );  
 ////////////////////////////////////////////////////////////////////////////  
 //Process32First和Process32Next函数结合（寻找）获取ctfmon.exe进程ID号  
 //再调用OpenProcess函数根据进程ID获得进程句柄  
 PROCESSENTRY32 pe32; //进程相关信息存储这个结构里  
 pe32.dwSize=sizeof(pe32);  
 //给系统内的所有进程拍一个快照  
 HANDLE hProcessSnap=::CreateToolhelp32Snapshot(TH32CS\_SNAPPROCESS,0);  
 BOOL bMore=::Process32First(hProcessSnap,&pe32);  
 while(bMore)  
 {  
 if(strcmp("ctfmon.exe",pe32.szExeFile)==0) //如果找到进程名为ctfmon.exe  
 hProcess=OpenProcess(PROCESS\_ALL\_ACCESS,FALSE,pe32.th32ProcessID); //获取句柄  
 bMore=::Process32Next(hProcessSnap,&pe32); //寻找下一个  
 }  
   
//在ctfmon进程中分配空间  
 LPVOID lpBuf=VirtualAllocEx(hProcess,NULL,strlen(DllName),MEM\_COMMIT, PAGE\_READWRITE );  
 DWORD WrSize;  
 //把DllName里的数据写入到分配的空间里  
 WriteProcessMemory(hProcess, lpBuf, (LPVOID)DllName, strlen(DllName), &WrSize);  
//创建远程线程  
 CreateRemoteThread(hProcess,NULL,0,(LPTHREAD\_START\_ROUTINE)LoadLibraryA,lpBuf,0,NULL);  
 return 0; //程序使命完成，结束  
 }

当然,给一个程序安装钩子,也可以让指定的应用程序加载特定的动态链接库,但要了解,加载动态链接库的是是应用程序的主程序,你总不能让应用程序不干它自己的事,而来一直执行DllMain函数里的代码吧!而且即使这样,当安装钩子的程序退出或卸载钩子的时候，那么被系统强迫加载动态链接库的程序，也会自动释放动态链库，退出DllMain函数。如此，那就没有办法了吗？，办法肯定是有的，用CreateThread函数。当其它程序主线程执行DllMain函数的时候，使其调用CreateThread再创建一个线程，就行了

**第六十一个GetWindowThreadProcessId根据窗口句柄获得对应进程ID号,和线程ID号**

函数只有两个参数，第一个参数是窗口句柄，第二个参数是一个DOWRD类型的指针,函数返回线程ID

如:

DWORD ThreadId,ProcessId;  
 ThreadId=GetWindowThreadProcessId(wnd,&ProcessId);

**第六十二个EnumWindows枚举当前正运行的所有主窗口，不包括子窗口**

调用这个函数，还必须定义一个回调函数，它的格式是这样的：BOOL CALLBACK lpEnumFunc(HWND wnd, LPARAM lParam);

EnumWindows函数有两个参数，第一个就是回调函数的地址，另一个是自定义参数，对应着回调函数的第二个参数。

每枚举一次窗口，这个回调函数就会被执行一次，而获得的窗口句柄，就会传递给回调函数，对应着回调函数的第一个参数。直到枚举完所有窗口。而在回调用函数里，返回真表示继续枚举，返回假则停止枚举。

例子：枚举窗口

#include<windows.h>  
 #include<stdio.h>  
 BOOL CALLBACK lpEnumFunc(HWND hwnd, LPARAM lParam);  
 int main()  
 {  
 ::EnumWindows(lpEnumFunc,0);  
 return 0;  
 }  
 BOOL CALLBACK lpEnumFunc(HWND wnd, LPARAM lParam)  
 {  
 char WindowName[52]={0};  
 GetWindowText(wnd,WindowName,sizeof(WindowName)); //根据窗口句柄获得窗口名  
 printf("窗口句柄:%d 窗口名称:%s\n,",wnd,WindowName);  
 //可以在这里加个判断当前是否有一个窗口正在运行  
   
 //如if(strcmp(WindowName,"Windows 任务管理器")==0) return 0;结束枚举，  
 return 1;  
 }

**第六十三个MessageBox弹出一个消息提示框**

int MessageBox(  
 HWND hWnd , //指明消息框是属于哪个窗口的，可以为NULL  
 LPCSTR lpText, //消息提示具体内容  
 LPCSTR lpCaption, //提示框窗口标题  
 UINT uType); //指明提示框类型，按钮，图标

这里只说下uType的常用可选值：  
按钮：  
MB\_ABORTRETRYIGNORE      终止 重试  忽略  
MB\_OK                                     确定  
MB\_OKCANCEL                       确定  取消  
MB\_RETRYCANCEL                重试  取消  
MB\_YESNO                              是     否  
MB\_YESNOCANCEL                是     否  取消  
图标：  
MB\_ICONWARNING       感叹号  
MB\_ICONINFORMATION   提示i  
MB\_ICONQUESTION      问号  
MB\_ICONSTOP          红X号  
按钮和图标可以结合用，如：

MessageBox(NULL,"该内存不能访问","出错",MB\_OK|MB\_ICONSTOP);

系统模式：MB\_SYSTEMMODAL

函数的返回值确定用户选择了哪个按钮，正面给各按钮的定义：  
IDABORT    “放弃”按钮  
IDCANCEL “取消”按钮  
IDIGNORE  “忽略”按钮  
IDNO       “否”按钮  
IDOK        确定  
IDRETRY     重试  
IDYES       是  
判断返回值是否与其相等即可。

**第六十四个GetForegroundWindow获得当前激活的窗口句柄**

函数没参数，调用即返回最前窗口句柄

这里举一个例子：每当切换窗口的时候弹出一个消息提示框

#include<windows.h>  
 int main()  
 {  
 char WindowName[52];  
 HWND wnd,Orgwnd;  
 wnd=Orgwnd=NULL;  
 while(1)  
 {  
 wnd=GetForegroundWindow();  
 if(wnd!=Orgwnd)  
 { GetWindowText(wnd,WindowName,sizeof(WindowName));  
 MessageBox(NULL,WindowName,"切换窗口到",MB\_OK);  
 Orgwnd=wnd;  
 }  
 Sleep(200);  
 }  
 }

**第六十五个GetTopWindow根据窗口句柄获得其下第一子窗口句柄（如果有）**

用过MFC的人都知道，在对话编辑区，如果要为控件排序的话，就按CTRL+D显示出每个控件的顺序，如下图：



而GetTopWindow函数获取的就是控件顺序为1的窗口句柄。

例子：改变一个主窗口下的第一子窗口的显示内容（前提得它有），这里就以上面那个abc对话框为例：

#include<windows.h>  
 int main()  
 {  
 HWND wnd;  
 wnd=FindWindow(NULL,"abc");  
 HWND FirstWnd=GetTopWindow(wnd);  
 SetWindowText(FirstWnd,"first");  
 return 0;  
 }

执行效果：



**第六十六个GetNextWindow根据子窗口句柄获得下一个或上一个同级的窗口句柄（返回NULL，函数执行失败）**

函数有两个参数，第一个是子窗口句柄，第二个参数指明寻找上一个，还是一下个窗口句柄，值：GW\_HWNONEXT（下一个），GW\_HWNDPREV（上一个）。比如子窗口句柄在主窗口的顺序为3，那么获取的是顺序为2或顺序为3的窗口句柄（具体取决于第二个参数），函数返回获得的窗口句柄.这样GetNextWindow结合GetTopWindow函数就可以遍历一个主窗口里的所有子窗口了。

例子：遍历一个窗口里的所有子窗口，以上面的abc窗口为例

#include<stdio.h>  
 #include<windows.h>  
 int main()  
 {  
 char Name[52];  
 HWND wnd;  
 wnd=FindWindow(NULL,"abc");   
 wnd=GetTopWindow(wnd);  
 while(wnd!=NULL)  
 {  
 GetWindowText(wnd,Name,sizeof(Name));  
 printf("窗口句柄:%d,名称:%s\n",wnd,Name);  
 wnd=GetNextWindow(wnd,GW\_HWNDNEXT); //GW\_HWNDNEXT获取下一个  
 }  
 return 0;  
 }

**第六十七个InvalidateRect发送一个WM\_PAINT消息给窗口（刷新窗口）**

函数定义：

BOOL InvalidateRect(  
 HWND hWnd , //要刷新窗口的句柄  
 CONST RECT \*lpRect, //刷新的范围  
 BOOL bErase //重画为TRUE  
 );

例子：在SetTimer函数里会举例

**第六十八个SetTimer设置一个定时器（每隔一段时间执行一次定时器函数）**

函数定义：

UINT SetTimer(  
 HWND hWnd , //窗口句柄  
 UINT nIDEvent, //定时器ID号，为了能设置多个定时器  
 UINT uElapse, //时间，指明间隔多久执行一次定时器函数，单位：毫秒  
 TIMERPROC lpTimerFunc); //定时器回调函数的地址

定时器函数的固定格式：VOID CALLBACK TimerProc(HWND hwnd,UINT uMsg,UINT\_PTR idEvent, DWORD dwTime);

例子：在用w,a,s,d键控制一个矩形移动的同时，一个相同的矩形自动移动。

#include<windows.h>  
 LRESULT CALLBACK WinSunProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam);//函数声明  
 VOID CALLBACK TimerProc(HWND hwnd,UINT uMsg,UINT\_PTR idEvent, DWORD dwTime);  
 int APIENTRY WinMain(HINSTANCE hInstance,  
 HINSTANCE hPrevInstance,  
 LPSTR lpCmdLine,  
 int nCmdShow)  
 {  
 WNDCLASS wndcls; //定义一个存储窗口信息WNDCLASS变量  
 wndcls.cbClsExtra=0; //默认为0  
 wndcls.cbWndExtra=0; //默认为0  
 wndcls.hbrBackground=(HBRUSH)GetStockObject(GRAY\_BRUSH); //背景画刷  
 wndcls.hCursor=LoadCursor(NULL,IDC\_ARROW); //光标  
 wndcls.hIcon=LoadIcon(NULL,IDI\_ERROR); //窗口图标  
 wndcls.hInstance=hInstance; //应用程序实例句柄由WinMain函数传进来   
 wndcls.lpfnWndProc=WinSunProc; //窗口消息处理函数  
 wndcls.lpszClassName="windowclass"; //窗口类名  
 wndcls.lpszMenuName=NULL; //窗口菜单名，没有菜单，为NULL  
 wndcls.style=CS\_HREDRAW | CS\_VREDRAW; //窗口类型，CS\_HREDRAW和CS\_VERDRAW 表明  
 //当窗口水平方向垂直方向的宽度变化时重绘整个窗口  
 RegisterClass(&wndcls); //把窗口信息提交给系统，注册窗口类  
 HWND hwnd; //用以存储CreateWindow函数所创建的窗口句柄  
 hwnd=CreateWindow("windowclass","first windows",   
 WS\_OVERLAPPEDWINDOW,0,0,600,400,NULL,NULL,hInstance,NULL); //创建窗口  
 ShowWindow(hwnd,SW\_SHOWNORMAL); //窗口创建完了，显示它  
 UpdateWindow(hwnd); //更新窗口，让窗口毫无延迟的显示  
 SetTimer(hwnd,1,200,(TIMERPROC)TimerProc); //设置定时器  
 MSG msg; //消息结构类型  
 while(GetMessage(&msg,NULL,0,0)) //获取消息  
 {  
 TranslateMessage(&msg); //此函数用于把键盘消息(WM\_KEYDOWN,WM\_KEYUP)转换成字符消息WM\_CHAR  
 DispatchMessage(&msg); //这个函数调用窗口过程处理函数，并把MSG里的信息处理后传给过程函数的四个参数  
 }  
 return 0;  
 }  
 VOID CALLBACK TimerProc(HWND hwnd,UINT uMsg,UINT\_PTR idEvent, DWORD dwTime) //定时器函数  
 {  
 static int x=0,y=0;  
 x+=15;  
 if(x>500)  
 {y+=15;x=0;}  
 HDC dc=GetDC(hwnd);  
 Rectangle(dc,x,y,x+30,y+30);  
 }  
 LRESULT CALLBACK WinSunProc(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)  
 { static int x=0,y=0;  
 switch(uMsg)  
 {  
 case WM\_CHAR:  
 {  
 if(wParam=='a') x-=10;  
 else if(wParam=='d') x+=10;  
 else if(wParam=='w') y-=10;  
 else if(wParam=='s') y+=10;  
 RECT rect;  
 GetClientRect(hwnd,&rect);  
 ::InvalidateRect(hwnd,&rect,TRUE); //发送WM\_PAINT消息，刷新窗口  
 }  
 break;  
 case WM\_PAINT:  
 HDC dc;   
 PAINTSTRUCT ps;   
 dc=BeginPaint(hwnd,&ps); //BeginPaint只能在响应WM\_PAINT,不能用GetDC获取设备上下文  
 Rectangle(dc,x,y,x+30,y+30);  
 break;  
 case WM\_CLOSE: //用户关闭了窗口  
 DestroyWindow(hwnd); //销毁窗口，并发送WM\_DESTROY消息  
 break;  
 case WM\_DESTROY: //如果窗口被销毁  
 PostQuitMessage(0); //让进程退出  
 break;  
 }  
 return DefWindowProc(hwnd,uMsg,wParam,lParam); //未处理的消息通过DefWindowProc函数交给系统处理  
 }

**第六十九个RegisterHotKey注册一个热键**

函数定义：

BOOL RegisterHotKey(  
 HWND hWnd ,  
 int id,  
 UINT fsModifiers,  
 UINT vk);

第一个参数hWnd表明热键消息（HOT\_KEY）发送给哪个窗口，为NULL表明直接把消息投递给调用这个函数进程的消息队列。

第二个参数可以自定取值，取值范围0xC000-0xFFFF,这个参数是为了程序能同时拥有多个热键而存在。

第三个参数fsModifiers的可选取值如下：MOD\_ALT（Alt键），MOD\_CONTROL（Ctrl键），MOD\_SHIFT（Shift键），MOD\_WIN（‘田’图标键）  
最一个参数是一个ASCII值，指明具体和哪个非系统键组合。  
如QQ的热键ctrl+alt+z，注册这个热键的语句是RegisterHotKey(NULL,0x0001,MOD\_CONTROL|MOD\_ALT,‘Z’)  
如QQ的截图热键 RegisterHotKey(NULL,0x0001,MOD\_CONTROL|MOD\_ALT,'A')

例子：按下ctrl+alt+x热键，弹出消息提示框，询问是否要退出。

//#include "stdafx.h" 新建空工程，不需要该头文件  
 #include<windows.h>  
 int APIENTRY WinMain(HINSTANCE hInstance,  
 HINSTANCE hPrevInstance,  
 LPSTR lpCmdLine,  
 int nCmdShow)  
 {  
 RegisterHotKey(NULL,0x0001,MOD\_CONTROL|MOD\_ALT,'X'); //注册热键  
 MSG msg;  
 while(GetMessage(&msg,NULL,0,0)) //从消息队伍获取消息  
 {  
 if(msg.message==WM\_HOTKEY) //热键消息  
 {  
 if(MessageBox(NULL,"你确定要退出程序吗？","热键提示",MB\_YESNO|MB\_SYSTEMMODAL)==IDYES)  
 {  
 UnregisterHotKey(NULL,0x0001); //卸载热键  
 break;  
 }  
 }  
 }  
 return 0;  
 }

记得要退出程序

**第七十个StretchBlt在窗口输出一个位图**

这个函数比BitBlt多了两个参数，那就是源目标DC的宽高，像BitBlt函数，只有目标DC的宽高。

有了这两个参数的加入，StretchBlt函数功能就比BitBlt函数强大了许多，它可以缩小或放大图片，可以把一张图片上的任意矩形区域覆盖到另一张图片上的任意区域。

函数语句：StretchBlt(hdc,0,0,bmInfo.bmWidth,bmInfo.bmHeight,memDC,0,0,50,50,SRCCOPY);

具体用法参考BitBlt函数。

**第七十一个TextOut根据设备DC在窗口输出文字**

函数定义:

BOOL TextOut(HDC hdc, // 设备DC  
 int nXStart, // 窗口x坐标  
 int nYStart, // 窗口y坐标，字符串左上角是将是x,y  
 LPCTSTR lpString, // 字符串  
 int cbString // 字符串中字符的个数  
 );

例子：在窗口输出文字，为了方便，这里依旧在"无标题.txt - 记事本",窗口里输出文字

#include<windows.h>  
 #include<string.h>  
 int main()  
 {  
 char Text[52]="从坐标点50,50开始输出文字";  
 HWND wnd=FindWindow(NULL,"无标题.txt - 记事本");  
 HDC dc=GetDC(wnd);  
 SetTextColor(dc,RGB(255,0,0)); //设置文字颜色  
 while(1)  
 {  
 TextOut(dc,50,50,Text,strlen(Text));  
 Sleep(200);  
 }  
 return 0;  
 }

**第七十二个DrawText根据设备DC在窗口的一个矩形区输出文字。**

int DrawTextW(  
 HDC hDC, //设备DC  
 LPCWSTR lpString, //字符串  
 int nCount, //字符串的个数  
 LPRECT lpRect, //指明一个矩形区  
 UINT uFormat); //输出格式

uFormat的常用取值

值                             说明   
DT\_BOTTOM              将正文调整到矩形底部。此值必须和DT\_SINGLELINE组合。   
DT\_CENTER               使正文在矩形中水平居中。   
DT\_VCENTER             使正文在矩形中垂直居中。   
DT\_END\_ELLIPSIS      对于显示的文本，如果结束的字符串的范围不在矩形内，它会被截断并以省略号标识。   
如果一个字母不是在字符串的末尾处超出了矩形范围，它不会被截断并以省略号标识。   
字符串不会被修改，除非指定了DT\_MODIFYSTRING标志。   
T\_WORD\_ELLIPSIS, DT\_PATH\_ELLIPSIS和DT\_END\_ELLIPSIS不能和此参数一起使用   
DT\_LEFT 正文左对齐。   
T\_RIGHT 正文右对齐。   
DT\_RTLREADING 当选择进设备环境的字体是希伯来文或阿拉伯文字体时，为双向正文安排从右到左的阅读顺序都是从左到右的。   
DT\_TOP 正文顶端对齐（仅对单行）。   
DT\_WORDBREAK 断开字。当一行中的字符将会延伸到由lpRect指定的矩形的边框时，此行自动地在字之间断开。一个回车一换行也能使行折断。   
DT\_WORD\_ELLIPSIS 截短不符合矩形的正文，并增加省略号。

**第七十三个GetLogicalDriveStrings获取系统分区信息**

函数定义：

DWORD GetLogicalDriveStrings(  
 DWORD nBufferLength, //指明lpBuffer参数大小  
 LPSTR lpBuffer //分区信息将会存储在这个参数，格式是“分区NULL分区NULL分区NULL NULL”两个NULL结尾  
 //假设当前电脑有C，D，E分区，那字符串的存储格式是 "C:\\\0D:\\\0E:\\\0\0"; ('\\'转义字符\)  
 );

例子：枚举当前磁盘所有分区

#include<windows.h>  
 #include<stdio.h>  
 int main()  
 {  
 char \*szBuffer=new char[52];  
 GetLogicalDriveStrings(52,szBuffer);  
 while(\*szBuffer!='\0')  
 {  
 printf("%s\n",szBuffer);  
 szBuffer+=strlen(szBuffer)+1;  
 }  
 return 0;  
 }

**第七十四个GetDiskFreeSpaceEx获取一个分区（盘符）的信息（已用空间，总大小，可用空间）**

这个函数必须用ULARGE\_INTEGER联合类型来存储磁盘使用信息。因为要获取磁盘的已用空间，总大小，可用空间，所以我们必须定义三个ULARGE\_INTEGER类型变量来存储这三个信息。而具体信息就存储在ULARGE\_INTEGER类型的QuadPart成员变量（该成员占八位字节）

如获取C盘的信息：ULARGE\_INTEGER dwAvailable,dwFree,dwTotal;

GetDiskFreeSpaceEx("c:\\",&dwAvailable,&dwTotal,&dwFree);//获取分区信息

下面结合GetLogicalDriveStrings举个例子：获取当前磁盘所有分区信息并输出

#include<windows.h>  
 #include<stdio.h>  
 int GetSpace(char \*P);  
 int main()  
 {  
 char \*szBuffer=new char[52];  
 GetLogicalDriveStrings(52,szBuffer);  
 while(\*szBuffer!='\0')  
 {  
 printf("%s ",szBuffer);  
 GetSpace(szBuffer);  
 szBuffer+=strlen(szBuffer)+1;  
 }  
 return 0;  
 }  
 int GetSpace(char \*Par)  
 {  
 ULARGE\_INTEGER dwAvailable,dwFree,dwTotal;  
 double fDwa,fDwf,fDwt;  
 char szSize[128]={0};  
 int Size;  
 GetDiskFreeSpaceEx(Par,&dwAvailable,&dwTotal,&dwFree); //获取分区信息  
 Size=dwTotal.QuadPart/1024/1024; //获取总大小  
 fDwt=Size/1024.0;  
 Size=dwAvailable.QuadPart/1024/1024; //获取已用空间  
 fDwa=Size/1024.0;  
 Size=dwFree.QuadPart/1024/1024; //获取可用空间  
 fDwf=Size/1024.0;  
 sprintf(szSize,"总大小:%0.2fGB 已用空间:%0.2fGB 可用空间:%0.2fGB",fDwt,fDwa,fDwf);  
 printf("%s\n",szSize);  
 return 0;  
 }

**第七十五个WritePrivateProfileString修改或创建一个INI文件里的数据**

INI文件的内容一般由节名，键名，键值组成，先来看一下INI文件的结构，打开一个INI文件,我们可能会看到以下内容

////////////////////////////////////////////////////////////////////////////////////

[gmy\_p]  
exist\_p=0  
linux\_p=

[boot]  
a20=0  
a21=0

///////////////////////////////////////////////////////////////////////////////////

上面的内容中[gmy\_p]和[boot]便是INI文件的节名，节名包含键名，和键值。一个INI文件可以有多个节名.

那么哪些是键名，那些是键值呢，在“=”左边的是键名，而在右边的就是键值，键值可以为NULL。

好了，看一下WritePrivateProfileString的函数的定义：

BOOL WritePrivateProfileString(  
 LPCWSTR lpAppName, //节名  
 LPCWSTR lpKeyName, //键名  
 LPCWSTR lpString, //修改的数据  
 LPCWSTR lpFileName //INI文件名  
 );

如果要修改键值，那么要提供哪些信息呢，首先，必须要知道INI文件的路径（lpFileName)，要修改的键值是在哪个节名下(lpAppName)，以及具体是哪个键名(lpKeyName)，还有修改的键值数据(lpString).

比如我要把之前INI文件里节名为gmy\_p下的键名exist\_p的键值改为100(假设这个文件的路径为d:\gho.ini).

那么就是语句：WritePrivateProfileString("gmy\_p","exist\_p","100","d:\\gho.ini");

WritePrivateProfileString函数功能不止于此，当函数提供的INI文件名，节名，键名不存在时，那么函数就会创建他们。这样，我们就可以用这个函数创建一个INI文件，或在一个INI文件里创建一个节名，或在一个节名下创建一个键名。

如：WritePrivateProfileString("ZhengYong","QQ","\*\*\*980073","d:\\Info.ini");

**第七十六个GetPrivateProfileString获取一个INI文件里的数据**

函数定义：

DWORD GetPrivateProfileStringW(  
 LPCWSTR lpAppName, //节名  
 LPCWSTR lpKeyName, //键名  
 LPCWSTR lpDefault, //默认值，填0既可  
 LPWSTR lpReturnedString, //接收数据的缓存区（字符串）  
 DWORD nSize, //指明缓存区的大小  
 LPCWSTR lpFileName //INI文件名  
 );

例子获取一个键值：假设D盘下有一个名为Info.ini文件，它的内容如下：

////////////////////////////////////////////////////////////////////////////

[ZhengYong]  
QQ=\*\*\*980073  
Age=100  
Weight=65kg  
[LiYang]  
QQ=\*\*\*990129  
Age=22  
Weight=55kg

///////////////////////////////////////////////////////////////////////////

如果我想要获取节名为"ZhengYong"下的键名QQ的键值，那么就是：

#include<windows.h>  
 #include<stdio.h>  
 int main()  
 {  
 char KeyValue[252];  
 ::GetPrivateProfileString("ZhengYong","QQ",0,KeyValue,252,"d:\\Info.ini");  
 printf("%s\n",KeyValue);  
 return 0;  
 }

同WritePrivateProfileString类似，如果提供的节名，或键名为NULL，则获取当前所有的节名或键名。跟分区信息存储格式一样，字符串里，多个节名，或键名以'\0'间隔，字符串最终以两个'\0'结束。

例子：枚举ZhengYong节名下的所有键名：

#include<windows.h>  
 #include<stdio.h>  
 int main()  
 {  
 char Data[252];  
 ::GetPrivateProfileString("ZhengYong",NULL,0,Data,252,"d:\\Info.ini");  
 char \*pKey=Data;  
 while(\*pKey!='\0')  
 {  
 printf("%s\n",pKey);  
 pKey+=strlen(pKey)+1;  
 }  
 return 0;  
 }

那么枚举节名只要在上面的例子中，把函数的节名参数设为NULL就行了，如：

GetPrivateProfileString(NULL,NULL,0,Data,252,"d:\\Info.ini");

大家可以用这个函数编一个读取INI文件内容的程序，以便更好的掌握这个函数。记得把接收数据的缓存区设置大一点。

**第七十七个GetSystemMetrics获得特定窗口的高宽度**

该函数只有一个参数，常用取值如下：

SM\_CXSCREEN 屏幕宽度

SM\_CYSCREEN屏幕高度

SM\_CXFULLSCREEN窗口客户区宽度

SM\_CYFULLSCREEN窗口客户区高度

SM\_CYMENU菜单栏高度

SM\_CYCAPTION//标题栏高度

SM\_CXBORDER窗口边框宽度

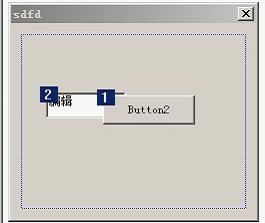
SM\_CYBORDER窗口边框高度

例子：获取屏幕分辨率（桌面宽高度）

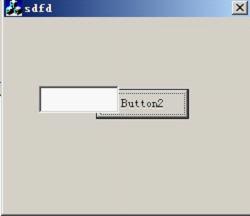
#include<windows.h>  
 #include<stdio.h>  
 int main()  
 {  
 int ScreenX=::GetSystemMetrics(SM\_CXSCREEN);  
 int ScreenY=::GetSystemMetrics(SM\_CYSCREEN);  
 printf("屏幕分辨率:%dX%d\n",ScreenX,ScreenY);  
 return 0;  
 }

**第七十八个SetWindowPos设置一个窗口的大小和它的Z序**

窗口的Z序是什么意思呢？用过MFC的人应该都清楚，在对话框编辑区按CTRL+D就会显示出每个控件的顺序。如下图：



设置控件的顺序有什么用呢，大家看到上面两个控件有什么特别的吗？对了，两个控件正好有一部分重叠，这时候问题就来了，重叠的部分显示的是那个窗口呢，或者说是以什么来确定显示哪个窗口，我想大家也应该猜到了，是以控件的顺序来确定的。顺序较大的会被显示。这个程序运行如下图如示：



明白窗口的Z序了，我们就来看一下这个函数的参数及其意思。

函数定义：

BOOL SetWindowPos(  
 HWND hWnd, //要设置的窗口句柄  
 HWND hWndInsertAfter,  
 int X,  
 int Y, //X,Y指明窗口左上角的位置  
 int cx, //窗口宽度  
 int cy, //窗口高度  
 UINT uFlags);

第二个参数hWndInsertAfter的常用取值：

HWND\_BOTTOM：  
将窗口置于Z序的底部.  
HWND\_NOTOPMOST：如果窗口在Z序顶部，则取消顶部位置，如果不是，则该参数无效  
HWND\_TOP:将窗口置于Z序的顶部。  
HWND\_TOPMOST:将窗口置于Z序的顶部。窗口当前未被激活，也依然是顶部位置

最后一个参数uFlags可以是Z序中hWnd的前一个窗口句柄的或以下常用取值：

SWP\_HIDEWINDOW;隐藏窗口

SWP\_SHOWWINDOW：显示窗口

SWP\_NOMOVE指明X,Y参数无效

SWP\_NOSIZE指明CX ,CY参数无效

SWP\_NOZORDER指明hWndInsertAfter参数无效

例子：设置一个窗口像PPS和任务栏那样，总在最前显示。

以"无标题.txt - 记事本"窗口为例

#include<windows.h>  
 int main()  
 {  
 HWND wnd=::FindWindow(NULL,"无标题.txt - 记事本");  
 ::SetWindowPos(wnd,HWND\_TOPMOST,0,0,0,0,SWP\_SHOWWINDOW|SWP\_NOMOVE|SWP\_NOSIZE);  
 return 0;  
 }

例子2：全屏一个窗口

依旧以"无标题.txt - 记事本"窗口为例：

#include<windows.h>  
 int main()  
 {  
 int ScreenX=::GetSystemMetrics(SM\_CXSCREEN);  
 int ScreenY=::GetSystemMetrics(SM\_CYSCREEN);  
 HWND wnd=::FindWindow(NULL,"无标题.txt - 记事本");  
 ::SetWindowPos(wnd,HWND\_TOPMOST,0,0,ScreenX,ScreenY,SWP\_SHOWWINDOW);  
 return 0;  
 }

**第七十九个CreateFile创建一个文件，或打开一个文件用于读写,函数返回文件句柄**

函数定义：

HANDLE CreateFile(  
 LPCSTR lpFileName, //文件名  
 DWORD dwDesiredAccess, //指明对文件进行何种操作，是要读它（GENERIC\_READ）还是要写入（GENERIC\_WRITE）  
 DWORD dwShareMode, //指明文件可以同时被多个程序读写吗？FILE\_SHARE\_READ可以同时读，FILE\_SHARE\_WRITED可以同时写  
 LPSECURITY\_ATTRIBUTES lpSecurityAttributes, //指向一个SECURITY\_ATTRIBUTES结构的指针，一般为NULL  
 DWORD dwCreationDisposition, //安全属性，指明以何种方式打开或创建文件  
 DWORD dwFlagsAndAttributes, //指明文件的属性，隐藏？只读？系统文件？为NULL表示默认属性  
 HANDLE hTemplateFile //如果不为零，则指定一个文件句柄。新文件将从这个文件中复制扩展属性   
 );

第五个参数dwCreationDisposition的常用取值及意思

TRUNCATE\_EXISTING 将现有文件缩短为零长度，清空文件的内容，文件必须已经存在

CREATE\_ALWAYS 创建一个文件，如果文件已经存在，则覆盖它

CREATE\_NEW 创建文件，如果文件已经存在，则函数执行失败

OPEN\_ALWAYS打开文件，如果文件不存在，则创建它

OPEN\_EXISTING 打开文件，文件必须存在。

第六个参数dwFlagsAndAttributes的常用取值及其意思

FILE\_ATTRIBUTE\_NORMAL 默认属性

FILE\_ATTRIBUTE\_HIDDEN 隐藏

FILE\_ATTRIBUTE\_READONLY 只读

FILE\_ATTRIBUTE\_SYSTEM 系统文件

**第八十个ReadFile根据文件句柄，从文件中读取一段数据**

函数定义：

BOOL WINAPI ReadFile(  
 HANDLE hFile, //文件句柄  
 LPVOID lpBuffer, //接收文件数据的缓存区  
 DWORD nNumberOfBytesToRead, //指明读取多少数据(字节）  
 LPDWORD lpNumberOfBytesRead, //实际读取数据  
 LPOVERLAPPED lpOverlapped //一般为NULL，如文件打开时指定了FILE\_FLAG\_OVERLAPPED，该参才有具体取值。  
 );

例子：读取txt文件的内容，假设E盘下有一个名a.txt的文件，文件内容为123456789

#include<windows.h>  
 #include<stdio.h>  
 int main()  
 {  
 char Text[25]={0};  
 DWORD dwSize;  
 HANDLE FileHandle=CreateFile("e:\\a.txt",GENERIC\_READ,0,NULL,OPEN\_EXISTING  
 ,FILE\_ATTRIBUTE\_NORMAL ,NULL); //获取文件句柄  
 ReadFile(FileHandle,Text,15,&dwSize,NULL); //从文件中读取15个字节  
 printf("内容：%s 实际读入字节：%d\n",Text,dwSize);  
 return 0;  
 }

**第八十一个WriteFile根据文件句柄，写入一段数据到文件中**

函数定义：

BOOL WriteFile(  
 HANDLE hFile, //文件句柄  
 LPCVOID lpBuffer, //该缓存区的数据将要写入到文件里  
 DWORD nNumberOfBytesToWrite, //指明写入多少数据  
 LPDWORD lpNumberOfBytesWritten, //实际写入数据  
 LPOVERLAPPED lpOverlapped //一般为NULL  
 );

例子：在E盘创建一个名为aa.txt的文件，并向其写入数据

#include<windows.h>  
 #include<stdio.h>  
 int main()  
 {  
 char Text[25]="123456789";  
 DWORD dwSize;  
 HANDLE FileHandle=CreateFile("e:\\aa.txt",GENERIC\_WRITE,0,NULL,CREATE\_ALWAYS,  
 FILE\_ATTRIBUTE\_NORMAL,NULL);  
 ::WriteFile(FileHandle,Text,9,&dwSize,0);  
 return 0;  
 }

**第八十二个SetFilePointer移动一个文件指针的位置**

移动一个文件指针的位置有什么用呢，作用是读取一个文件里指定位置的数据，比如我只要读取文件中第四个字节到第七个字节这一段的数据，用SetFilePointer函数就可以完成。

函数定义：

DWORD SetFilePointer(  
 HANDLE hFile, //文件句柄  
 LONG lDistanceToMove, //移动字节，负数表示反向移动  
 PLONG lpDistanceToMoveHigh, //为了支持超大文件而存在，一般为NULL  
 DWORD dwMoveMethod //从哪里开始移动，FILE\_BEGIN 从文件开始处开始移动，FILE\_CURRENT当前位置，FILE\_END文件末尾  
 );

例子：假设E盘下有一个名为a.txt的文件，内容为"123456789",读取该文件第四个字节到第七个字节的数据

#include<stdio.h>  
 int main()  
 {  
 char Text[25]={0};  
 DWORD dwSize;  
 HANDLE FileHandle=CreateFile("e:\\a.txt",GENERIC\_READ,0,NULL,OPEN\_ALWAYS,  
 FILE\_ATTRIBUTE\_NORMAL,NULL);  
 SetFilePointer(FileHandle,3,NULL,FILE\_BEGIN);  
 ReadFile(FileHandle,Text,4,&dwSize,NULL);  
 printf("%s\n",Text);  
 return 0;

例子2：从文件中第四个字节开始写入数据，被新数据所占位置的数据会被覆盖掉，依旧以上面a.txt文件为例子

#include<windows.h>  
 #include<stdio.h>  
 int main()  
 {  
 char Text[25]={"abcd"};  
 DWORD dwSize;  
 HANDLE FileHandle=CreateFile("e:\\a.txt",GENERIC\_WRITE,0,NULL,OPEN\_ALWAYS,  
 FILE\_ATTRIBUTE\_NORMAL,NULL);  
 SetFilePointer(FileHandle,3,NULL,FILE\_BEGIN);  
 WriteFile(FileHandle,Text,4,&dwSize,NULL);  
 return 0;  
 }

则写入后a.txt文件里的数据为123abcd89

如果要在文件的末尾添加数据，就用这个语句：SetFilePointer(FileHandle,0,NULL,FILE\_END);

**第八十三个GetFileSize获取一个文件的大小**

函数定义：

DWORD GetFileSize(  
 HANDLE hFile, //文件句柄  
 LPDWORD lpFileSizeHigh //一般为NULL  
 );

如获取a.txt文件的大小：

#include<windows.h>  
 #include<stdio.h>  
 int main()  
 {  
 DWORD FileSize;  
 HANDLE FileHandle=CreateFile("e:\\a.txt",GENERIC\_WRITE,0,NULL,OPEN\_ALWAYS,  
 FILE\_ATTRIBUTE\_NORMAL,NULL);  
 FileSize=GetFileSize(FileHandle,NULL);  
 printf("%d字节\n",FileSize);  
 return 0;  
 }

**第八十四个SetTextColor根据窗口输出文本颜色**

第一个参数是设备DC，第二个参数是一个COLORREF类型的颜色值，可用RGB进行转换。

**第八十五个SetBkColor设置背景颜色**

参数跟SetTextColor函数一样，第一个DC，第二个COLORREF

**第八十六个GetWindowDC获取整个窗口设备上下文DC**

像GetDC获取的只是客户区DC，不能对窗口标题栏，状态栏等进行操作。该函数用法跟GetDC一样，仅区域不一样。

例子：在一个窗口的标题栏输出文字

#include<windows.h>  
 int main()  
 {  
 HWND wnd=FindWindow(NULL,"无标题.txt - 记事本");  
 HDC dc=GetWindowDC(wnd);  
 SetTextColor(dc,RGB(255,0,0)); //文本颜色设置为红色  
 ::SetBkColor(dc,RGB(0,0,255)); //文本背景颜色设置为蓝色  
 while(1)  
 {  
 TextOut(dc,0,0,"123456",6);  
 Sleep(200);  
 }  
 return 0;  
 }

**第八十七个GetDesktopWindow获取桌面窗口句柄**

该函数没有参数，调用返回桌面窗口句柄

例子：

#include<windows.h>  
 int main()  
 {   
 HWND DeskWnd=GetDesktopWindow(); //获取桌面窗口句柄  
 HDC DeskDC=GetWindowDC(DeskWnd); //获取DC  
 HBRUSH brush=::CreateSolidBrush(RGB(255,0,0)); //红色画刷  
 SelectObject(DeskDC,brush); //选入画刷  
 while(1)  
 {  
 ::Rectangle(DeskDC,50,50,200,200);  
 Sleep(200);  
 }  
 return 0;  
 }

**第八十八个CreateCompatibleBitmap根据DC创造一个兼容的位图**

该函数需与CreateCompatibleDC函数配合使用

函数第一个参数是窗口DC，第二，三参数指明窗口宽高，函数返回位图句柄(HBITMAP）

创建一个兼容的位图是什么意思呢？就好比给HBITMAP分配内存以及指定这位图相关的一些信息（跟DC相关的信息），如位图的宽高，数据大小，但此时数据没有具体取值。就好比一个字符串，我已经知道字符串大小了，但却不知道字符串具体是什么：

如：

char \*p;  
 p=new char[15]; //知道字符串大小为15

但此时p所指向的缓存区，没有具体取值。

而用CreateCompatibleBitmap函数，创建的位图，只是一个空壳子。数据没有赋值，那要怎样给数据赋值呢？

首先得把这个位图句柄选入一个DC（该DC必须为CreateCompatibleDC函数创建的）里，然后再用BitBlt函数具体给数据赋值就行了。

例子：实时获取屏幕图像

为了方便，在记事本窗口输出图像，自己就不创建窗口了（打开"无标题.txt - 记事本")

#include<windows.h>  
 int main()  
 {   
 HWND TextWnd=FindWindow(NULL,"无标题.txt - 记事本");  
 HDC TextDC=GetDC(TextWnd);  
 HWND DeskWnd=::GetDesktopWindow();  
 RECT DeskRC;  
 ::GetClientRect(DeskWnd,&DeskRC);  
 HDC DeskDC=GetDC(DeskWnd);  
 HBITMAP DeskBmp=::CreateCompatibleBitmap(DeskDC,DeskRC.right,DeskRC.bottom);  
 HDC memDC;  
 memDC=::CreateCompatibleDC(DeskDC);  
 SelectObject(memDC,DeskBmp);  
 while(1)  
 {  
 StretchBlt(memDC,0,0,DeskRC.right,DeskRC.bottom,DeskDC,0,0,DeskRC.right,DeskRC.bottom,SRCCOPY);  
 RECT TextRC;  
 GetClientRect(TextWnd,&TextRC);  
 ::StretchBlt(TextDC,0,0,TextRC.right,TextRC.bottom,memDC,0,0,DeskRC.right,  
 DeskRC.bottom,SRCCOPY);  
 Sleep(300);  
 }  
 return 0;  
 }

**第八十九个GetDIBits从一个兼容位图里获取位图数据**  
先来分析一下位图文件信息结构，一个位图由以下四部分组成：

位图文件头（BITMAPFILEHEADER）//占14字节

位图信息头（BITMAPINFOHEADER）//占40字节

调色板（LOGPALLETE）//如果真彩位图，那该部分没有，直接是位图数据

实际位图数据

而GetDIBits函数获取的就是实际位图数据这一部分了。

接来看一下BITMAPFILEHEADER这个结构以及它成员的意思和取值

typedef struct tagBITMAPFILEHEADER {  
 WORD bfType; //表示文件类型，值必须为0x4d42  
 DWORD bfSize; //文件的大小  
 WORD bfReserved1; //保留，必须为0  
 WORD bfReserved2; //保留，必须为0  
 DWORD bfOffBits; //位图前三部分所占的字节，真彩色位图为54  
 } BITMAPFILEHEADER;

上面的成员，只有bfSize的取值不确定，其它都一样，也就是说，每个真彩位图，这几个成员取值都是一样的.下面的例子可以说明。

读取一个真彩位图的文件信息头。

#include<windows.h>  
 #include<stdio.h>  
 int main()  
 {  
 BITMAPFILEHEADER bfh;  
 HANDLE hFile=CreateFile("e:\\aa.bmp",GENERIC\_READ,0,NULL,OPEN\_EXISTING,  
 FILE\_ATTRIBUTE\_NORMAL,0);  
 DWORD dwSize;  
 ReadFile(hFile,(void \*)&bfh,sizeof(BITMAPFILEHEADER),&dwSize,0);  
 printf("bfType:%x\n",bfh.bfType);  
 printf("bfSize:%d\n",bfh.bfSize);  
 printf("bfReserved1:%d\n",bfh.bfReserved1);  
 printf("bfReserved2:%d\n",bfh.bfReserved2);  
 printf("bfOffbits:%d\n",bfh.bfOffBits);  
 return 0;  
 }

再来看一下BITMAPINFOHEADER这个结构以及它成员的意思和取值

typedef struct tagBITMAPINFOHEADER{  
 DWORD biSize; //本结构大小，为40  
 LONG biWidth; //位图的宽度，以像素为单位  
 LONG biHeight; //位图的高度，以像素为单位  
 WORD biPlanes; //目标设备的级别，必须是1  
 WORD biBitCount; //每个像素所占的位数，24表示真彩位图  
 DWORD biCompression; //位图压缩类型，一般为BI\_RGB（未经压缩）  
 DWORD biSizeImage; //实际位图数据这部分的所占用的字节数  
 LONG biXPelsPerMeter; //指定目标设备水平分辨率，单位像素/米，为0  
 LONG biYPelsPerMeter; //指定目标垂直分辨率真，单位像素/米，为0  
 DWORD biClrUsed; //指定目标设备实际用到的颜色数，如果该值为0，则用到的颜色数为2的biBitCount方  
 DWORD biClrImportant; //表示图像中重要的颜色数，如果为0，则所有颜色都是重要的。  
 } BITMAPINFOHEADER;

调色板（LOGPALLETE）由于大部分都是针对真彩位图操作，此部分略过

GetDIBits函数定义：

int GetDIBits(  
 HDC hdc, //位图兼容的DC  
 HBITMAP hbmp, //位图句柄  
 UINT uStartScan, //从哪行开始扫描  
 UINT cScanLines, //扫描多少行数据  
 LPVOID lpvBits, //接收数据的缓存区  
 LPBITMAPINFO lpbi, //真彩位图，此处填文件信息头就行了  
 UINT uUsage //真彩位图填DIB\_RGB\_COLORS，表示由R,G,B三色直接构成  
 );

例子：截屏，并把屏幕图片保存成位图

#include<windows.h>  
 void ScreenSnap(HBITMAP hBitmap,char \*bmpPath,HDC dc);  
 int main()  
 {   
 HWND DeskWnd=::GetDesktopWindow(); //获取桌面窗口句柄  
 RECT DeskRC;  
 ::GetClientRect(DeskWnd,&DeskRC); //获取窗口大小  
 HDC DeskDC=GetDC(DeskWnd); //获取窗口DC  
 HBITMAP DeskBmp=::CreateCompatibleBitmap(DeskDC,DeskRC.right,DeskRC.bottom); //兼容位图  
 HDC memDC=::CreateCompatibleDC(DeskDC); //兼容DC  
 SelectObject(memDC,DeskBmp); //把兼容位图选入兼容DC中  
 BitBlt(memDC,0,0,DeskRC.right,DeskRC.bottom,DeskDC,0,0,SRCCOPY); //拷贝DC  
 ScreenSnap(DeskBmp,"d:\\Screen.bmp",DeskDC);  
 return 0;  
 }  
 void ScreenSnap(HBITMAP hBitmap,char \*bmpPath,HDC dc)  
 {  
 BITMAP bmInfo;  
 DWORD bmDataSize;  
 char \*bmData; //位图数据  
 GetObject(hBitmap,sizeof(BITMAP),&bmInfo); //根据位图句柄，获取位图信息  
 bmDataSize=bmInfo.bmWidthBytes\*bmInfo.bmHeight; //计算位图数据大小  
 bmData=new char[bmDataSize]; //分配数据  
 BITMAPFILEHEADER bfh; //位图文件头  
 bfh.bfType=0x4d42;  
 bfh.bfSize=bmDataSize+54;  
 bfh.bfReserved1=0;  
 bfh.bfReserved2=0;  
 bfh.bfOffBits=54;  
 BITMAPINFOHEADER bih; //位图信息头  
 bih.biSize=40;  
 bih.biWidth=bmInfo.bmWidth;  
 bih.biHeight=bmInfo.bmHeight;  
 bih.biPlanes=1;  
 bih.biBitCount=24;  
 bih.biCompression=BI\_RGB;  
 bih.biSizeImage=bmDataSize;  
 bih.biXPelsPerMeter=0;  
 bih.biYPelsPerMeter=0;  
 bih.biClrUsed=0;  
 bih.biClrImportant=0;  
 ::GetDIBits(dc,hBitmap,0,bmInfo.bmHeight,bmData,(BITMAPINFO \*)&bih,DIB\_RGB\_COLORS);//获取位图数据部分  
 HANDLE hFile=CreateFile(bmpPath,GENERIC\_WRITE,0,NULL,CREATE\_ALWAYS,  
 FILE\_ATTRIBUTE\_NORMAL,0); //创建文件  
 DWORD dwSize;  
 WriteFile(hFile,(void \*)&bfh,sizeof(BITMAPFILEHEADER),&dwSize,0); //写入位图文件头  
 WriteFile(hFile,(void \*)&bih,sizeof(BITMAPINFOHEADER),&dwSize,0); //写入位图信息头  
 WriteFile(hFile,(void \*)bmData,bmDataSize,&dwSize,0); //写入位图数据  
 ::CloseHandle(hFile); //关闭文件句柄  
 }