Windows API 编程入门教程

大家好 再次自我介绍一下 我是 beyondcode, 这次心血来潮, 计划着做一系列关于 Windows API 编程的教程，用于帮助一些在 Windows API 编程上有疑惑的，纳闷的，迷惑的新手朋友们。

先解释一些术语或名词吧 SDK 是 Software Development Kit 的简写，也就是软件开发包的意思，其中就包含了我们写程序要用到的一些头文件，库，工具，帮助文档之类的。

Windows API 编程是指调用 Windows 的接口函数来进行程序的编写，例如 MessageBox 就是一个 API 函数或者说接口函数。怎么说都可以，自己理解就行。如果你连这个都不太懂，我想也不会搜到这篇文章了吧~·

为什么做这个系列教程呢，请听我一一道来先，最近遇到一些事一些人，让我真的感觉在这方面的引导入门文章真的很是匮乏，加上 Windows SDK 头文件中那些复杂，庞大，'烦人'的宏定义与数据类型定义，对于一个新手来说(我所说的新手不单只刚接触编程的，还特指那些在其他语言领域有比较高造诣的朋友) 一个纯 SDK 写的 helloworld 程序都算是一个有些困难和挑战的任务了吧。 本着帮助别人,高兴自己的原则，我有了这个打算，当然对自己以前所学，所经历做一次回忆，也是这次计划的一部分。

声明一下，本系列教程是面向广大初次接触 WIN32 SDK 程序编写的新手朋友们的，如果你是高手，一笑而过吧~当然，除了一笑而过，也多谢你们提出指正文章中的错误，以免我误人子弟啊~~谢谢

Ok 废话不多说，进入正题，今天第一篇，讲什么？ 对于一个新人来说，第一次接触 SDK 编程或者说 API 编程，什么最迷惑你们的，我们讲它，我觉得 Windows SDK 中那'烦人'的数据类型定义和宏定义应该算这个很角色吧。。

其实微软的本意也是善良的，为了减轻程序员的负担，和为了编程的方便，才花了那么多心思与精力定义出了这么一大套数据类型与宏定义，这也是我为什么在之前说它烦人都是加上引号的原因，因为他不是真的烦人，熟练了，你不但不觉得它烦，反而离不开它了，呵呵，日久深情也就是这么来的。

呵呵 先看几个数据类型定义吧

typedef float FLOAT;

typedef long LONG;

typedef short SHORT

typedef int INT;

typedef char CHAR;

float, long, short, int, char 这几个数据类型都是大家熟悉的 C/C++的数据类型吧，微软将他们重新定义了一下，很简单，就是改变名字为大写了，这样做的目的大概是微软为了编码的方便吧，输入法大小写都不用切换了，多人性化呀 呵呵。。

再看几个数据类型定义的例子

typedef unsigned int UINT;

typedef unsigned int UINT32;

typedef signed int INT32;

typedef unsigned long DWORD;

typedef unsigned short WORD;

这些数据类型的定义就稍微有实质性作用一些了，注意观察，他们都比较短了，不用写那么长了，而且也还比较直观，如果我要定义一个无符号整形， 我就不用写 unsigned int a;

这么长了，只需 UINT a; 多简单， 多明了，所以我说其实不烦人吧。

其中 DWORD 算是 SDK 程序中可以经常看见的一个数据类型了，经常被使用，很多新手也就不明白，这是什么数据类型啊，现在看到了吧，其实就是无符号长整形 unsigned long，给他取了个外号而已··没什么技术含量，所以不用怕，程序中究竟是写 unsigned long 还是 DWORD 都看你自己心情，因为他们都代表同一种数据类型。

下面再介绍 2 个很重要的，经常被使用到的，无处不在的数据类型 WPARAM,LPARAM

先看看他们定义吧

typedef LONG_PTR LPARAM;

typedef UINT_PTR WPARAM;

先告诉你，这 2 个数据类型很重要，不是危言耸听，以后你写 SDK 程序就知道了，看他们的定义如上，有些迷糊？别，我们一步一步分析,我们分析 LPARAM。首先定义 LPARAM 为 LONG_PTR 也就是用 LPARAM 的地方也就可以写成 LONG_PTR，LONG_PTR 又是被定义成什么的呢?

typedef long LONG_PTR;

看到了吗？ 也就是 long 所以归根结底，LPARAM 就是 long 型，所有 LPARAM 型的变量，你都可以直接使用 long 数据类型代替。不过不推荐这样，至于为什么，各位思考思考呢~~

以上这些数据类型是参考 MSDN 中的说明，或者可以查看 WinDef.h 这个头文件查看这些 Windows 数据类型的定义，那么也请各位自己推推看 LARAM 和 WPARAM 的真面目吧~

各位朋友在推导的过程中可能发现 LONG_PTR 的定义是这样写的

#if defined(_WIN64)

typedef __int64 LONG_PTR;

#else

typedef long LONG_PTR;

#endif

这是什么意思呢，能看懂英文都能知道这在定义些什么,如果定义了 _WIN64 这个宏 那么就定义 LONG_PTR 为 __int64，否则定义 LONG_PTR 为 long。 很简单吧 也就是说如果 _WIN64 这个宏在前面被定义了，那么这里的 LONG_PTR 就被定义为__int64 这个在 64 位编程下的数据类型，否则就定义为 long 型，这样说应该比较好理解了吧。在这里，各位就不必深究__int64 了， 在目前的主流 32 位编程下很少使用它啦。理解就 ok 了。这样定义是微软为了程序员编写的程序能在 32 位与 64 位下都能编译而采用的伎俩。

有关这些 Windows 的数据类型，想查看他们的真面目，其实很简单，在 VC6.0,VS2008 这些集成开发环境里面，你只需要在一个数据类型上面点击右键，在弹出菜单中选择‘Goto Defination’ 或者是 ‘查看定义’就可以看到了，如果看到的还不是最终面目，在继续上面步骤。直到看到它的本质数据类型为止。通过这样，新手对于 Windows 的这些复杂的数据类型定义也就有了根本的认识，不再是迷迷糊糊，在以后的编程中也就不会出现不知道用哪种数据类型或者哪些数据类型之间可以相互转换的情况了。不过还需要多多观察与练习才是啊

~~

下面再来看一看 windows 中定义的一些宏

#define VOID void

#define CONST const

2 个最简单的宏，也是只变成大写而已，难道又是为了方便程序员不切换输入法？还真的人性化呀。

Windows SDK 中的宏定义是最庞大的，最复杂的，但也是最灵活的，为什么这样说，先不告诉你，我会在以后的系列文章中一点一点的讲解，累积，因为太多了，也比较复杂，我们就采取在需要用到的时候才讲解它，目前看来还没这个必要了解那么多，就了解上面 2 个很简单的好了，像其他如：WINAPI CALLBACK GetWindowText 这些宏现在讲了不但记不住还会增加你们的负担。,我们就在以后要用到的时候再做讲解。

到这里第一篇系列文章的内容也就差不多了。新手朋友们哪些地方迷惑的，提出来，我可以考虑是否加在后续的文章中进行解说。本 SDK 系列入门教程需要你们的支持。谢谢。

今天，开始第二篇文章，这章我准备介绍一下 Windows 平台下编程中 Unicode 编码和 ASCII 编码的相关问题。

不知道各位新手朋友们遇到这样的问题没有呢，新建一个 Windows 应用程序，调用 MessageBox 这个函数，准备让它弹出一段提示文本，可是编译器在编译的时候却报错说，不能将 const char* 或者 const char[] 转换为 const wchar_t* 之类的提示呢，很多刚接触 Windows API 编程的朋友们在这里可能就卡住了，不知如何下手解决了，其实，这就是 Unicode 编码和 ASCII 编码的问题了。我下面就会一一道来

关于 Unicode 和 ASCII 具体的编码是怎么的，我这里就不详细介绍了，也介绍不了，如果需要深入了解，网上有很多这方面的专门文章，我这里就只对 Unicode 编码和 ASCII 编码在 Windows 平台下的编程相关的内容进行介绍。

我们都知道 Unicode 和 ASCII 最大的区别就是 Unicode 采用 2 个字节来存储一个字符，不管是英文，汉字，还是其他国家的文字，都有能用 2 个字节来进行编码，而 ASCII 采用一个字节存储一个字符，所以对于英文的编码，那是足够的了，可是对于汉字的编码，则必须采用一些特殊的方法，用 2 个 ASCII 字符来表示一个汉字。

我们在写程序的过程中，势必要和字符打交道，要输入，获取，显示字符，到底是选用 Unicode 字符呢还是 ASCII 字符呢，这都是各位自己的权利。但为了程序的通用性和符合目前操作系统的主流趋势，Unicode 编码是被推荐的。由于 Unicode 字符要比 ASCII 字符占用的空间大一倍，编译出来的程序在体积上和占用的内存上必定要大一些，不过这并不是什么很大的问题。所以微软目前的 SDK 中保留了 2 套 API，一套用于采用 Unicode 编码处理字符的程序的编写，一套用于采用 ASCII 编码处理字符的程序的编写。 例如，我们上面提到的 MessageBox，它其实不是一个函数名，而是一个宏定义，我们先来看看它是怎么被定义的，再来讨论它。

```
#ifdef UNICODE

#define MessageBox    MessageBoxW

#else

#define MessageBox    MessageBoxA

#endif
```

看到了吗？ 很简单是不是，如果定义了 UNICODE 这个宏 那么就定义 MessageBox 为 MessageBoxW，如果没有定义 UNICODE 这个宏， 那么就定义 MessageBox 为 MessageBoxA，MessageBox 后面的 W 和 A 就是代表宽字节(Unicode)和 ASCII，这样，其实存在于 SDK 中的函数是 MessageBoxW 和 MessageBoxA 这两个函数.

MessageBox 只是一个宏而已。所以在程序中，这 3 个名字你都可以使用，只不过需要注意的是，使用 MessageBoxA 的话，那么你要注意传给它的参数，字符都必须是单字节，也就是 ASCII，在程序中就是 char，如果使用 MessageBoxW 的话，那么，字符都必须使用 Unicode，程序中就是 wchar_t。 但是这样有个非常不方便的地方那就是，如果你使用 W 后缀系列的函数的话，那么你的程序使用的字符就是 Unicode 字符编码的，但是如果你需要用这个程序的源代码编译出字符采用 ASCII 编码的程序，那么需要改动的地方就太大了。凡是涉及到字符操作的地方都需要改变。那么 ，有没有比较好的办法不做更改就可以用同样的代码编译出 ASCII 版本的程序呢。

当然有，就是我们在编程的时候尽量使用不带后缀的宏定义，如上例，就使用 MessageBox，其中的参数也不明确使用 char 还是 wchar_t 而是使用微软给我们定义的 TCHAR 字符数据类型，它的定义和上面 MessageBox 函数的定义差不多，都是根据是否定义了 UNICODE 这个宏来判断是将 TCHAR 定义为 char 还是 wchar_t，所以这样一来，这个 TCHAR 的数据类型就是可变的了，它根据工程的设置而定义为相应的最终字符类型，这样我们的程序就可以不做任何更改就可以轻松的编译出另外一个版本的了。是不是非常方便。

前面 2 篇文章纯文字的介绍比较多，因为很多是概念性的，需要理解，后面的文章我准备配合一些小示例程序，使用一些简单的 API 函数，遇到的相关的概念在一并介绍的方法进行。

所以，前 2 篇文章如果各位朋友不是很能理解，不用担心，影响不是很大，经过后面的学习，你就会慢慢的理解前面所说的内容了。

下面我罗列一些我们在 Windows 平台下编程经常使用到的和字符或字符串有关的数据类型。

char 和 wchar_t

这两个类型大家绝对不会陌生吧，一个是单字节的字符类型，一个是宽字节的字符类型(也就是 Unicode 字符)。

```
char  c = 'b';

wcha_t  wc = L'b';
```

上面我就分别定义了 2 个变量 c 和 wc ，相信第一个定义大家都看的懂，就是定一个字符变量 c，其中保存了'b'这个字符。 那么第二个呢？ 我相信还是很多人都看的懂，要是你看不懂也没关系，现在就告诉你，也是定义一个字符变量 wc， 只不过这个字符变量是 Unicode 字符变量，用 2 个字节来保存一个字符，而上面的 c 这个字符变量只有一个字节来保存，那么在'b'前面的 L 又是什么意思呢，它就表示这里的'b'这个字符是一个 Unicode 字符，所以第二个定义的意思就是将 L'b'这个 Unicode 字符保存到 wc 这个 Unicode 字符变量中。

如果我要定义一个字符数组怎么定义呢？用分别用单字节的 char 和宽字节的 wchar_t 来定义就应该是：

```
char  c[10];

wchar_t wc[10];
```

如果是要带初始化的字符数组的声明，我们来看看怎么写

```
char c[] = "beyondcode";

wchar_t wc[] = L"beyondcode";
```

看到了吗，宽字节的操作其实和单字节的字符操作一样吧，只是在前面加上 L 表示是宽字节的字符或者字符串。

上面都是属于 C/C++中的知识，并没有涉及太多 Windows 中的数据类型，那么各位朋友们在 Windows 编程中看到的满到处都是的 TCHAR,LPSTR, LPCSTR, LPWSTR, LPCWSTR, LPTSTR, LPCTSTR 这些数据类型又是怎么回事呢？别急，我们一步一步的来，最后我会联系到那上面去的。

上面的你都知道或者是理解了的话，那我们继续，除了可以声明一个字符数组，我还可以定义一个字符指针变量来指向一个字符数组，当然这个字符数组可以是 Unicode 的宽字节字符数组，也可以是单字节字符数组，如下：

```
char    c[] = "hello beyondcode"; //定义一个字符数组

wchar_t    wc[] = L"hello beyondcode"; //定义一个宽字节字符数组


char    *p = c; //定义一个字符指针，指向刚才的字符数组

wchar_t *wp = wc; //定义一个宽字节字符指针，指向刚才的宽字节字符数组
```

这样之后，我就可以通过指针来改变刚才我们定义的 2 个数组，例如：

```
p[0] = 'H';

wp[0] = L'H';
```

把上面 2 个数组的第一个字符通过指针改变成大写。这里是可以通过指针来修改的，因为我没有定义指针为常量指针，也就是没有加 const 修饰符。如果我像下面这样定义的话，那么就不能通过这些指针来改变他们所指向的数据了，而是只有读取他们。

```
const    char    *p = c;

const    wchar_t    *wp = wc;
```

上面将的都是 C/C++的基础知识，有点啰嗦，为了照顾新手朋友们嘛，下面我们就来看看 Windows 是怎么定义它的数据类型的

首先，定义了 CHAR, WCHAR 的这 2 个字符数据类型，就是我们上面讨论的两个字符数据类型改了一下名字而已。现在你还不昏吧··

```
typedef char    CHAR;

typedef wchar_t    WCHAR;
```

然后，用刚才定义的 CHAR, WCHAR 这 2 个字符数据类型去定义了一系列其他字符指针类

型。

typedef CHAR *LPSTR;

typedef WCHAR *LPWSTR;

这样一定义之后，LPSTR 的就是 CHAR*， 而 CHAR 又是 char， 所以 LPSTR 的本质就是 char*，也就是我们上面熟悉的不能再熟悉的字符指针， 那 LPWSTR 不用我推导，相信你也推导出来了吧。不过我还是推导一下，LPWSTR 是 WCHAR * ， WCHAR 是 wchar_t，这样 LPWSTR 就是 wchar_t* ，也就是我们上面讨论的宽字节字符指针。上面这些定义都是在 WinNT.h 这个头文件中定义的，读者朋友们有兴趣在这个头文件里面去挖掘挖掘吧，上面 2 个定义我只是提取了重要的部分，其实在里面他还定义了其他很多别名.

看了 LPSTR, LPWSTR 是怎么一回事之后，我们再接再厉，看看 LPCSTR,LPCWSTR 这 2 个数据类型又是怎么一回事呢, 老规矩，先看 windows 的定义。

typedef CONST CHAR *LPCSTR;

typedef CONST WCHAR *LPCWSTR;

和上面的比较，名字中就多了一个大写的 C，这个 C 的含义就代表是 const 修饰符,也就是我们上面所说的常量指针，指向的内容不能通过这个指针被改变，但可以读取。定义中的大写的 CONST 也是一个宏，我在第一篇文章中就讲过了，代换出来也就是 const， 所以请读者自己推导一下这两个数据类型的本质是什么。

所以，在 windows 平台下的编程过程中，凡是可以使用 char* 的地方，你都可以使用 LPSTR 来代替，凡是可以使用 wchar_t*的地方，你都可以使用 LPWSTR 来代替，至于怎么用，还是那句老话，看你个人心情，只不过 Windows 的 API 函数中关于字符串的都是使用 LP 这种数据类型。但是你还是可以给他传递 char* 或者 wchar_t* ,只要他们的本质是一样的，那怎么不可以呢~~

下面，我们来看一看一些示例。

char c = 'c'; 和 CHAR c = 'c'; 是一样的

wchar_t wc = L'w'; 和 WCHAR wc = L'w'; 是一样的

char* p 和 LPSTR p 是一样的

wchar_t* wp 和 LPWSTR wp 是一样的

再来看看动态内存分配怎么写的呢

char* p = new char[10]; //动态分配了十个字符

也可以写成

CHAR* p = new CHAR[10];

LPSTR p = new CHAR[10];

LPSTR p = new char[10];

宽字节的再来一次

wchar_t* wp = new wchar_t[10];

也可以写成下面这些形式

WCHAR* wp = new WCHAR[10];

LPWSTR wp = new WCHAR[10];

LPWSTR wp = new wchar_t[10];

上面定义的这些字符指针 p , wp 都没有用 const 修饰符，所以可以通过他们来修改他们所指向的内容。这里留给读者一个问题，怎么定义有 const 修饰符的字符指针呢，都可以用什么形式来写呢，写得越多越好哟。。

通过上面这些，我想你大概已经了解了 LPSTR, LPCSTR, LPWSTR, LPCWSTR 这四个数据类型了，他们无非就是:

LPSTR ------- char*

LPCSTR ------- const char*

LPWSTR ------- wchar_t*

LPCWSTR -------- const wchar_t*

下面我提一个问题，如果你在你的程序中使用的字符串都是通过 LPWSTR,LPCWSTR 这种宽字节(Unicode)字符指针来进行操作的，那么在 Unicode 环境下编译，完全没有问题，如果这时你需要编译一套 ASCII 版本的程序，那你会怎么办呢？　你说将用 LPWSTR 和 LPCWSTR 的地方全部换成 LPSTR 和 LPCSTR，再将字符串前面的 L 去掉就可以了，对，这是一种方法，但是!! 所有人在这里都应该知道我要说但是，这也太麻烦了吧。难道没有通用点的方法吗？ 有!! 所有人在这里也都知道我会说有，呵呵。 那就是使用微软的通用数据类型,说通用数据类型有点太专业了，其实也就那样，请听我慢慢分析来。我在上一篇文章中说过，凡是涉及字符串操作的 API 函数有 2 套，一个 A 系列的,一套 W 系列的,还有一套宏，能根据不同的工程环境定义成不同的 API 函数名。那么在字符类型上微软也使用几乎同样的技术，定义了一套宏能根据不同的工程环境定义成不同的字符数据类型。我上面就提到过的 TCHAR,LPTSTR, LPCTSTR 就是这样的类型。

首先说说 TCHAR，它是被这样定义的：

```
#ifdef　UNICODE

typedef　WCHAR　TCHAR;

#else

typedef　char TCHAR
```

看到了吗? 它也是根据 UNICODE 这个宏被定义没有，如果被定义了，那么 TCHAR 代表的数据类型就是 WCHAR， 也就是 wchar_t， 如果没被定义，那么 TCHAR 就代表的是 char

同样 LPTSTR,LPCTSTR 也是这样的，考虑到篇幅，我就只列出 LPTSTR 来给大家看看了

```
#ifdef　UNICODE

typedef　LPWSTR　LPTSTR;

#else

typedef　LPSTR LPTSTR;
```

这个是我简化了的定义，真实面目有些复杂，不过意思也是如此，有兴趣可以自己看看，在 WinNT.h 这个头文件中。下面再次解释一下上面这个 LPTSTR 的定义, 还是老样子，根据 UNICODE 这个宏被定义与否来决定怎么定义 LPTSTR ，如果是定义了 UNICODE 这个宏，表示当前工程环境是 Unicode 环境，那么 LPTSTR 就被定义为了 LPWSTR， LPWSTR 就是

我们前面所讲的 wchar_t* ，所以此时 LPTSTR 代表的数据类型就是 wchar_t* ， 如果这时的工程没有定义 UNICODE 这个宏，那么就定义 LPTSTR 为 LPSTR，而 LPSTR 就是我们前面所说的 char* ，所以这是的 LPTSTR 就代表 char*。懂了吗?各位，我都觉得自己有些啰嗦了··不好意思···

然后还有一个宏需要讲一下，由于我们使用通用数据类型，那么我事先就不知道我的源代码需要在 Unicode 下编译还是在 ASCII 环境下编译,所以如下这种情况

TCHAR tc = 'a'; 或者是 TCHAR tc = L'a'; 是否合适呢？ 前面我已经说过了字符或字符串常量前面加 L 代表这是宽字节的字符或字符串，将一个宽字节字符赋值给一个 TCHAR 数据类型的变量 tc，什么情况下是正确的呢？ 各位思考一下呢?

如果当前工程是 Unicode 环境，那么 TCHAR 数据类型就是 wchar_t 的宽字节类型，所以 tc 就是宽字节字符变量，那么上面第二个赋值语句就是正确的，而第一个就是错误的。

如果反过来，当前的工程是 ASCII 环境，那么 TCHAR 代表的是 char 这种数据类型，那么第一个赋值语句就是正确的，而第二个就是错误的了。

分析了这么多，我就是要讲一个宏 _T(), 只要将字符或者字符串常量放在_T()这个宏里面，那么这个宏就能根据当前的环境决定是否在字符或字符串前面加 L，如下面：

TCHAR tc = _T('A');

如果这么写，在不需要改写源代码的情况下，就可以编译出 Unicode 和 ASCII 两套程序

而只需要改变工程的环境而已。

这篇文章的内容大概就这么多了，关于后续文章的内容安排，我会适当采纳各位朋友的留言来进行安排。

在这里我介绍的是 Windows 平台下的和字符串操作有关的数据类型，至于 MFC 中的 CString 类，c++标准库中的 string，我就不做讲解了。

大家好，还是我 beyondcode，再次见面，前面介绍的那么多'理论知识'，你们都懂了吗？就算还没有彻底领悟，但至少还是有那么一点意识了吧，知道有那么一回事了吧。这一篇我打算通过一个小小小例子，来回忆一下我们以前介绍的相关知识，如 Windows 的数据类型，特别是和字符和字符串操作相关的数据类型，还有就是 Unicode 和 ASCII 在 API 函数上的具体体现。

另外，SDK 编程交流群已经建立，很多朋友踊跃参加，系列文章和群的发展离不开你们。群号:81543028。

Ok，我们正式开始，我打算从一个简单的 SDK 程序开始，别怕，就几行代码而已·· /* BY beyondcode */

```
#include <windows.h>
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
MessageBoxA( NULL, "Hello beyondcode", "Title", MB_OK );
return 0;
}
```

复制代码程序你已经看到了，这恐怕就是一个最简单的带窗口的 SDK 程序了吧，如果你能写出代码行数比这个还少，又带窗口显示字符串的 SDK 程序，欢迎交流，呵呵，开个玩笑。

程序倒是简单，可是我还是要问一问，这个程序，你通过观察我在字符串的处理，还是在 API 函数的调用，还是主函数的参数写法，你能看出什么问题呢?.....对，就是我全部明确指出是单字节版本的，WinMain 的第三个参数是 LPTSTR 类型，调用的 MessageBox 是带 A 后缀的单字节版本，字符串常量"Hello beyondcode"和"Title"都没有使用 L 前缀。那么第二个问题来了，如果我告诉你我现在的工程环境是 使用 Unicode 字符集 (工程使用的字符集可以在 【项目】->工程属性 弹出的属性页中的 【配置属性】中的【常规】左边的【字符集】中设置)，那么我上面的程序能正常通过编译吗？当然能，因为我已经试过了，不信你也可以试试，可是为什么呢？这是因为我指定的参数和函数需要的参数都是单字节版本的，也就是说他们相互匹配。要是我这里将 MessageBoxA 改成 MessageBoxW 呢？就会出错吧，因为 MessageBoxW 的第二个，和第三个参数是需要 LPCWSTR，通过上一篇学习，我们知道也就是 const wchar_t*， 而我给出的两个字符串常量却没有用 L 前缀.也就是说他们是单字节的，传给宽字节版本的 MessageBoxW 当然就类型不匹配了啊，所以就通不过编译了吧。

通过上面的学习，我再出一个问题，如果我此时的工程环境是使用 Unicode 字符集，而这里我不用 MessageBoxA，也不用 MessageBoxW，而是用 MessageBox，其他的都不变，结果会怎么样呢？ 不能理解的可以加群讨论哟~~~

好了，单字节版本的程序，我们已经看到了，我们再来看看我们怎么才能把它改成宽字节版本的呢?

其实需要改的地方不多，也就 5 处 WinMain 改成 wWinMain， WinMain 的第三个参数改成 LPWSTR，MessageBoxA 改成 W，两个字符串常量加 L 就 ok 了。 /* BY beyondcode */

```
#include <windows.h>
int WINAPI wWinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR lpCmdLine, int nShowCmd )
{
MessageBoxW( NULL, L"Hello Beyondcode", L"Title", MB_OK );
return 0;
}
```

复制代码如果我想写一个代码比较通用的版本，也就是可以不用改动代码，就能编译出 Unicode 和 ASCII 的两个版本的程序，我应该怎么写呢？ 其实就是我上一篇重点讨论的，凡是涉及到字符串的都不明确指出是 Unicode 还是 ASCII 版本的，调用的 API 函数凡是涉及到字符串参数的都不明确指出调用是 A 后缀的还是 W 后缀的函数，而是调用没有后缀的函数，如上面的 MessageBox，这样就能写出代码比较通用的程序了。那么我们现在来把我们上面的程序改一改，让它通用 /* BY beyondcode */

```
#include <windows.h>
#include <tchar.h>
int WINAPI _tWinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPTSTR lpCmdLine, int nShowCmd )
{
MessageBox( NULL, _T("Hello Beyondcode"), _T("Title"), MB_OK );
return 0;
}
```

复制代码 WinMain 被改成了_tWinMain ,_tWinMain 也是一个宏，根据 UNICODE 这个宏被设置与否而被定义成 WinMain 或 wWinMain，和 LPTSTR 是一样的，这里还需要注意的是要包含 tchar.h 这个头文件，因为_tWinMain 和_T()这些宏是被定义在里面的。经过上面我们就写出了第一个 SDK 的可以编译出两个版本的比较通用的程序代码了。是不是有点成就感了呢。。

下面，我们继续在上面的程序中加一些功能，让它计算 1 到 10 的和，然后把结果显示给我们看，这个地方，很多 SDK 初学者就不知所措了，因为一个和是一个整数，怎么显示这个整数给我们呢，通过对话框? MessageBox，可是 MessageBox 显示的是字符串。而我们这里又不是控制台程序可以使用 printf 之类的格式化输出函数来输出数字，也不能使用 cout 之类的 C++对象来输出，那我们怎么办呢？通过对话框来显示结果是不错的选择，但是对话框需要的是字符串，那我们就把我们的结果格式化到一个字符串里面，然后传送给 MessageBox 让它显示出来。那么就需要用到格式化字符串函数，下面我们就介绍 wsprintf 这个函数#ifdef

UNICODE

```
#define wsprintf    wsprintfW
#else
#define wsprintf    wsprintfA
#endif // !UNICODE
```

复制代码说它是函数，是不确切的。因为它实际是一个宏,根据环境被定义成不同的函数名 wsprintfW 或者 wsprintfA， 而我们为了程序的通用性，直接使用 wsprintf，传递的参数凡是涉及到字符串常量的我们都是用_T()宏，字符串指针的我们都使用 LPTSTR 和 LPCTSTR。

下面我就先贴出添加了功能的程序代码，然后在做分析，你可以先不看分析，自己看一看代码，不懂的猜一猜它的意思。/* BY beyondcode */

```
#include <windows.h>
#include <tchar.h>
int WINAPI _tWinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPTSTR lpCmdLine, int nShowCmd )
{
int sum = 0;
for( int i = 1; i<=10; i++ )
sum += i;
TCHAR strSum[256] = { 0 };
wsprintf( strSum, _T("%d"), sum );
MessageBox( NULL, strSum, _T("Title"), MB_OK );
return 0;
}
```

复制代码怎么样，也还不算复杂吧，计算 1 到 10 的那部分不用我讲了吧，最后的结果存放在 sum 这个变量里，我们现在的目的就是要让它显示在 MessageBox 弹出的对话框上面。

首先我们定义一个字符数组，我使用的是通用类型 TCHAR，然后把它全部初始化为 0。接着调用格式化字符函数 wsprintf，它的第一个参数是 LPTSTR 类型的，指定经过格式化的字符串存放的地方，第二个参数是指定以什么格式来格式化后面的数据，这里我们要格式化一个整数，所以指定%d，这个和 printf 这些函数是一样的， 后面的参数就是我们要格式化的数据了。

这里还有一点可能有些新手朋友们不太懂，那就是，wsprintf 要求的第一个参数是 LPTSTR，而我传递的是一个 TCHAR 的数组名，这里我就在啰嗦一次咯，我们假设我们的环境是 UNICODE 的，那么 LPTSTR 相当于什么类型呢？上一篇就讲过的啊，就是 wchar_t* 就是

宽字符指针,而我们知道数组名就是代表这个数组元素类型的指针，那么这里 TCHAR 数组的元素类型就是 TCHAR，在 Unicode 环境下，TCHAR 就是 wchar_t 也就是说 strSum 代表的是 wchar_t 类型的指针，也就是 wchar_t*，所以看到了吗，他们是一样的类型。

通过上面的 wsprintf 函数的调用 strSum 这个字符数组中就包含了计算结果的字符串表示，然后我们通过 MessageBox 讲这个字符数组中的内容显示出来。在这里 MessageBox 的第二个参数类型是 LPCTSTR，也就是 const wchar_t*, 而我们上面说过我们的 strSum 代表的是 wchar_t*，这里同样可以传递给它又是为什么呢？这是因为阿，这里 strSum 在传递给 MessageBox 的时候就隐式转换成了 const wchar_t*了。

有关格式化字符串的函数还有如下，详细用法各位可以查看 MSDN，和上面所介绍的都差不多

sprintf 单字节版本的 C/C++库函数

swprintf 宽字节版本的 C/C++库函数

而我们上面的 wsprintf 和上面两个函数看起来很相似，大家不要搞混淆了啊，wsprintf 最前面的 w 不是代表 Wide，宽字节的意思了，而是 Windows 的 W，代表是 windows 的 API 函数了，其实它是一个宏这在上面已经说过了，真正的 API 函数其实是 wsprintfA 和 wsprintfW 这两个，在不严格的情况下通常我们也说 wsprintf 是函数，只要大家懂就行了~

OK, 这一篇文章就到这里了，源代码就这么点，我也不上传了，各位有兴趣自己敲一下。下一篇，我讲会就怎么自己创建窗口进行介绍，谢谢大家的支持。
上一篇，讲了一个简单的 SDK 程序的多种版本的编写，弹出了一个窗口，显示了我们计算 1 到 10 的结果，计算的程序不是重点，重点在于，一：让大家认识到 Unicode 版本的程序和 ASCII 版本的程序在编程方面的区别，以及怎么样编写出通用代码的程序。二：怎么样运用 API 或者 c++库函数格式化非字符数据到一个字符串中显示出来。

不过，那个相当简单的程序，还算不上是一个正儿八经的 SDK 程序，也就是说还不是一个纯爷们儿,因为我们并亲自完成一个 SDK 程序的经典步骤。而是调用了一个 MessageBox API 函数，这个函数虽然使用简单，但是在它的内部，那可是相当复杂啊~~~。怎么个复杂法，具体的我不知道，但是我知道的是一个 SDK 程序的经典步骤它是都用到了的，什么是编写 SDK 程序的经典步骤呢?新手朋友们听好了哦，现在我就告诉你。

第一步：注册窗口类

第二步：创建窗口

第三步：消息循环

第四步：编写窗口消息处理函数

上面我所说的，听起来都比较专业，下面我就解释一下，什么是注册窗口类呢?注册窗口类就是使用一个 窗口类结构体(WNDCLASSEX) 来描述一类窗口，这类窗口具有相同的属性，也就是你在结构体 WNDCLASSEX 中指定的那些值。只要是用这个窗口类创建的窗口都具有这些特性。至于 WNDCLASS 能描述哪些特性，下面会具体讲，这里你只要了解是用一个名叫 WNDCLASSEX 的结构体来描述一个窗口的类别。

创建窗口应该比较好理解吧，就是创建一个具体的窗口，好像是一句废话嘛。也就是说这个窗口是根据一个窗口类而创建的，不是凭空而造的。意思是你要创建一个窗口，那么必须要有一个已经注册的窗口类。

对于前两步，我打一个比方，就好比你要造一辆车，那们第一步首先是干什么？当然是设计图纸啦，图纸上就有说明这种车有哪些特性。然后第二步才是根据这个图纸来创建一个具体的看得见的车。所以我上面说的注册窗口类就好比设计窗口的图纸，然后就是根据这个窗口的图纸来创建一个具体的窗口。都说成这样了，应该明了了吧~~

至于消息循环，就是创建的窗口随时都有可能发生很多事情，那么发生的这些事情怎么通知你呢？比如窗口最小化了，窗口大小改变了，怎么通知你呢？ 其实就就是通过消息循环不断的取得窗口所发生的事情， 然后以消息的形式发送给我们后面要介绍的窗口消息处理函数。

消息处理函数呢就是我们程序员负责编写代码对具体的消息进行具体的处理，当然你也可以不处理，交给系统的默认处理函数来处理。

对于这两步，我也打一个比方。消息循环就好比汽车的一个总传感器，它源源不断的将汽车内部所发生的事情以消息的形式通过仪表板传达给开车的人，开车的人根据具体的事情而采取具体的操作，当然你也可以不操作，无动于衷，对于 windows 消息来说，不操作倒没有什么，而对于开车的人来说，不操作的后果就不好说了。 在这里，这个总传感器就相当于 SDK 程序的消息循环，不断的发送消息，而开车的人就相当于窗口消息处理函数，负责处理各种消息。明白了吧，还不明白的话就看看下面的具体的程序吧，也需还有最后一丝希望可以让你恍然大悟。

讲了正儿八经的 SDK 程序的经典步骤后，我们进入正式的代码阶段，通过代码结合上面所讲理论进一步巩固知识。我讲逐步讲解并逐步编写一个自己注册窗口类，创建窗口，带消息循环，并自己编写消息处理过程的程序。
首先给出程序框架/* BY beyondcode */

```
#include <windows.h>
#include <tchar.h>
LRESULT CALLBACK WinMessageProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam );
```

```
int WINAPI _tWinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPTSTR lpCmdLine, int nShowCmd )
{
return 0;
}
```

复制代码第一个函数声明，返回类型为 LRESULT，本质经查看是 long，然后函数调用约定 CALLBACK 和 WINAPI 是一样的，都是__stdcall，说明函数调用相关的约定，不必深究。至于为什么用 CALLBACK 是为了意思显而易见，表示是回调函数，什么是回调函数？也就是系统负责调用的，不必你亲自调用的函数，所以你在你的程序里是看不到调用 WinMessageProc 这个函数的代码的，你只负责编写它的代码，至于调用，系统会在有消息的时候自动调用它。WinMessageProc 的参数类型和个数是规定好了的，不然系统怎么知道怎么掉用，所以不能更改。

再解释一下这四个参数吧，第一个参数是一个窗口的句柄，也就是告诉你，是哪个窗口的消息，第二个参数是消息的类型，告诉你是什么消息，第三个和第四个参数是这个消息所带的一些额外的但是必须的数据。你在窗口消息处理函数中只使用他们就可以了，他们的值都是系统传递进来的。你只是根据他们来判断是哪个窗口的什么消息，并且获取该消息的额外参数信息。

有了程序框架，我们来第一步，注册一个窗口类//注册一个名叫 MyWindowClass 的窗口类

```
WNDCLASSEX wc;
wc.cbSize = sizeof( wc );
wc.style = CS_VREDRAW | CS_HREDRAW;
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
wc.hCursor = LoadCursor( NULL, IDC_ARROW );
wc.hIcon = LoadIcon( NULL, IDI_APPLICATION );
wc.hIconSm = LoadIcon( NULL, IDI_APPLICATION );
wc.hInstance = hInstance;
wc.lpfnWndProc = WinMessageProc;
wc.lpszMenuName = NULL;
wc.lpszClassName = _T("MyWindowClass");
if( !RegisterClassEx( &wc ) )
{
MessageBox( NULL, _T("注册窗口类出错"), _T("出错"), MB_OK );
return 0;
}
```

复制代码上面的 WNDCLASS 的各个成员值我就不一一介绍是什么含义，MSDN 上面讲的非常清楚，我只讲一两个比较重点的，第一个 lpszClassName 这个成员，我们给它指定的是 _T("MyWindowClass")这个值，这是指定这个窗口类的名字是什么，因为下面的创建窗口会用到这个名字。

lpfnWndProc 这个成员是 WinMessageProc 这个函数，这是指定这个窗口类所创建的窗口的消息处理函数是哪一个，我们这里指定的是 WinMessageProc。其他的参数我就不啰嗦了，各位不懂的 MSDN 一下或者在群里来交流一下。

指定了这个窗口类有哪些特性后就完了?当然没有，没有注册怎么使用啊，所以还需要注册，注册调用 RegisterClassEx 这个 API 函数，将刚才的 WNDCLASS 变量的地址传给它就可以进行注册了，如果注册失败，返回值为零，成功的话返回值为非零。

注册了窗口类，我们来第二步，创建一个窗口。代码如下：// 根据上面注册的一个名叫 MyWindowClass 的窗口类创建窗口

```
HWND newWind = CreateWindowEx( 0L, _T("MyWindowClass"), _T("beyondcode"), WS_OVERLAPPEDWINDOW, 0, 0, 200, 200, NULL, NULL, hInstance, NULL );
if( NULL==newWind )
{
MessageBox( NULL, _T("创建窗口出错"), _T("出错"), MB_OK );
return 0;
}
ShowWindow( newWind, nShowCmd );
UpdateWindow( newWind );
```

复制代码可见，创建窗口用 CreateWindowEx 这个 API 函数，它的第一个参数是扩展样式，我们这里不设置扩展样式，所以传递 0L，第二个参数就是窗口类的名字，我们这里指定我们上面已经注册了的那个名叫 MyWindowClass 的窗口类，第三个参数是窗口的标题，随便设置，第四个参数是窗口的样式，我们这里设置的是 WS_OVERLAPPEDWINDOW，一般主窗口都用这个样式，就是有最大化，最小化框，有标题栏，有系统菜单。。具体的可以参见 MSDN，第五个，六个，七个，八个参数分别指定窗口的初始坐标和长宽，第九个参数指定父窗口是哪个，这里没有父窗口，所以传递 NULL，第十个参数指定菜单的句柄，我们这里不设置菜单，所以传递 NULL,第十一个是应用程序句柄，用 WinMain 传递进来的那个 hInstance 参数，第十二个参数表示额外数据，不设置，所以为 NULL。

这个 API 函数有点复杂，不过用熟悉了也就不觉得了。这样我们就创建了一个窗口，返回值是一个窗口的句柄，如果是 NULL 的话，说明创建窗口失败了，如果不是 NULL 的话，说明成功了。

光创建成功了还不行，如果你不显示和更新它，你还是看不到它，所以需要调用 2 个 API 函数，ShowWindow 和 UpdateWindow，参数就是刚才创建成功的那个窗口的句柄，至于 ShowWindow 的第二个参数是指显示的类型，是最大化显示呢还是最小化显示呢，不过在程序中第一次调用 ShowWindow 必须使用 WinMain 所传递进来的参数的第四个参数的值。这是 MSDN 上说的~

窗口创建成功了，下面一步是消息循环了，消息循环说起来复杂，其实代码挺简单的，而且基本格式固定，如下：//消息循环

```
MSG msg;
while( GetMessage( &msg, NULL, 0, 0 ) )
{
TranslateMessage( &msg );
DispatchMessage( &msg );
}
```

复制代码看到了吗? 一直在一个循环里面，一直调用 GetMessage，只要 GetMessage 所取得的消息不是 WM_QUIT 的话，那么 GetMessage 的返回值就不是 0，那么循环就一直进行。在循环内部，将 GetMessage 取得的消息传递给 TranslateMessage 和 DispatchMessage 两个 API 函数进行处理.其中 DispatchMessage 就是将消息发送给了对应的窗口的窗口消息处理函数进行处理。至于 TranslateMessage 呢，则进行一些消息的转换，可以先不深究。

最后就是编写窗口消息处理函数的代码了，你需要处理那些消息，那么你就编写处理那些消息的代码，对于你不处理的消息，则统统交给一个叫 DefWindowProc 的 API 函数进行默认的处理。

```
LRESULT CALLBACK WinMessageProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam )
{
switch ( msg )
{
case WM_DESTROY:
{
PostQuitMessage( 0 );
break;
}
default:
return DefWindowProc( hwnd, msg, wParam, lParam );
}
return 0;
}
```

复制代码这里我们只处理了 WM_DESTROY 这个消息，这个消息是在窗口被销毁的时候发送给窗口消息处理函数的，在窗口处理函数中，我们判断这个消息是不是 WM_DESTROY，如果是，就调用 PostQuitMessage 这个 API 函数，如果是其他消息，我们就不管，将参数全部传递给 DefWindowProc 这个函数进行处理。

而 PostQuitMessage 这个 API 函数的功能就是发送一个 WM_QUIT 的消息。而我们前面说到过， 在消息循环中 GetMessage 一旦取得 WM_QUIT 这个消息，就返回值为 0，那么消息循环也就结束了，进而整个程序也就结束了，如果在这里我们处理 WM_DESTORY 函数，但是不调用 PostQuitMessage,那么结果会怎样呢，读者朋友们思考一下~~

好了，到这里，这个什么功能也没有的 SDK 程序也就完了，它只显示一个带有标题栏的可最大化，最小化的窗口，除了能够关闭它，你几乎不能进行其他任何操作，因为我们除了处理窗口销毁这个消息，其他任何消息我们都没处理。

这篇文章中的源代码在 VS2008，windows 7 平台下编写并完成编译运行。

我在 API 入门系列之四 -一个相当简单的 SDK 程序 中讲到了通过调用 MessageBoxAPI 函数来弹出一个对话框，那你有没有想过，我们一句简单的代码背后所隐藏的细节是怎么的呢，那时候让你了解这些未免还早了些，不过现在时机到了，通过一些基本的 API 函数调用来实现一个自己的对话框其实也不是很难，那么这一篇文章就是这个目的。并顺带介绍一些基本 API 的应用，由于前面五篇文章的铺垫，我相信你对 SDK 的程序的大致结构和相关字符处理都有所了解了，所以在这篇文章中的这些知识点，我就不再多说，以免有些人觉得我实在太过啰嗦。实在疑惑的，可以参看前面的文章。

首先我先大概的列出我们为完成这个任务所要用到的一些 API 函数他

SetWindowText　　设置窗口的标题

GetClientRect 得到窗口客户区的大小信息

GetWindowLongPtr 通过窗口句柄得到和窗口的相关联信息

CreateWindowEx 创建窗口

BeginPaint 得到窗口的设备句柄

EndPaint 释放窗口的设备句柄

DrawText 通过设备句柄在窗口上画出文字

好了，就这些函数，我们就能自己实现有一个确定按钮并在确定按钮正上方显示提示信息的简易的对话框了，不过这个对话框可是我们一句一句代码自己实现的哟~~还是比较有成就感吧~

程序的大体框架呢还是我们上一篇文章中的框架，注册窗口类，创建一个主窗口，消息循环，窗口消息处理函数。不过要我们需要在窗口消息处理函数中添加一些代码来完成我们需要的功能。在什么地方添加呢？ 上个程序，我们只处理 WM_DESTROY 这个消息。对于这个消息我不再做讲解，不懂的或者忘记了的可以自己 MSDN 或者看上一篇文章，这里我们要添加对两个消息的处理代码，首先是 WM_CREATE，这个消息会在一个窗口被创建的时候被发送到窗口消息处理函数，如果一些事情需要在一个窗口刚被创建的时候执行，那么通过处理 WM_CREATE 最合适不过啦，代码如下：

```
case WM_CREATE:
{
RECT rctClient; //用来存放主窗口客户区大小信息
const int buttonWidth = 80; //按钮的宽
const int buttonHeight = 25; /按钮的高
GetClientRect( hwnd, &rctClient ); //得到主窗口客户区的大小信息
HINSTANCE hInst = (HINSTANCE)GetWindowLongPtr( hwnd, GWLP_HINSTANCE );
HWND hButton = CreateWindowEx( 0L, _T("button"), _T(" 确 定 "), WS_VISIBLE | WS_CHILD , rctClient.right/2-buttonWidth/2, rctClient.bottom/2-buttonHeight/2, buttonWidth, buttonHeight, hwnd, (HMENU)2, hInst, NULL );
SetWindowText( hwnd, _T("自定义对话框") );
break;
}
```

复制代码在 WM_CREATE 消息的处理中，我们就用到了 GetClientRect，它的第一个参数是窗口的句柄，你想要获取哪个窗口的客户区大小，你就将传递哪个窗口的句柄，第二个参数是一个 RECT 结构的指针，我上面定义了一个 rctClient 变量，然后这里把这个变量的地址传递给 GetClientRect 的第二个参数，让它将所得到的窗口的大小信息保存到这个变量里面。这个函数的具体用法，读者朋友们还可以自己参考 MSDN，如果函数调用成功，那么 rctClient 这个结构体变量中就存放了这个窗口的大小信息了。

然后，我定义了两个整形常量 buttonWidth, buttonHeight 用来保存我们需要创建的按钮的宽和高。

再然后我调用 GetWindowLongPtr 这个函数获取和窗口有关的信息，这里获取的是窗口所属的应用程序实例的句柄，也就是 WinMain 函数所传递进来的第一个参数。在得到这些需要的信息之后，我们就开始着手子窗体的创建了，这里我们要创建的是一个按钮，按钮也是一个窗体，所以也需要窗口类，我们并没有写按钮的窗口类进行注册，那么这个窗口类由谁来

注册呢？其实是有系统创建并注册了按钮的窗口类，窗口类的名字是 button，所以我们这里只管用这个窗口类来创建窗口就是了，我们创建主窗口是用的 WS_OVERLAPPEDWINDOW 这个窗口样式，如果是创建一个子窗口，那么我们需要指定 WS_CHILD ,如果我们需要创建的窗口能显示出来，那么需要指定 WS_VISIBLE 这个窗口样式，并且还需要指定创建的窗口所属的父窗口的句柄，如上代码所示。其中第五个参数到第八个参数是该按钮的坐标位置和宽度高度的信息，因为我们需要将该按钮创建在主窗口的中央，所以有一系列的计算，具体是怎么计算的，就请各位自己仔细根据上面的代码进行思考了，如果还是有些疑惑，请与我讨论或者加入 SDK 编程（81543028）群进行讨论交流。

创建完了按钮子窗口，我们还需要将我们的主窗口的标题设置为我们想要的，可以通过 SetWindowText 这个 API 函数来完成，第一个参数就是要设置的窗口的句柄，这里为主窗口，所以是我们窗口消息处理函数传递进来的第一个参数 hwnd，第二个参数就是一个字符串指针，指向一个以零结尾的字符串。这里我们就直接将一个字符串常量的首地址传递给它。就完成了主窗口的标题设置。

经过上面这些步骤，我们已经在主窗体的中央显示了一个按钮了，并且把主窗口的标题设置为我们自己需要的，但是还要一个问题需要解决，那就是在按钮的正上方显示一串提示文本，怎么来完成呢，这就是我们下面要讲的。

要在主窗口的按钮的正上方显示提示文本信息，就需要得到主窗口的设备句柄，然后通过该设备句柄调用 GDI 函数 DrawText 来完成。由于该提示文本需要在每次窗口进行更新的时候绘出，所以我们需要处理 WM_PAINT 消息来达到这个目的。下面还是先看代码： case WM_PAINT:

{

const int buttonWidth = 80;

const int buttonHeight = 25;

const int textHeight = 25;

```
PAINTSTRUCT ps;

HDC hdc = BeginPaint( hwnd, &ps );

RECT rctClient,rctText;

GetClientRect( hwnd, &rctClient );

rctText.left = rctClient.left;

rctText.right = rctClient.right;

rctText.top = rctClient.bottom/2 - buttonHeight -textHeight;

rctText.bottom = rctClient.bottom/2 - buttonHeight;

DrawText( hdc, _T("Beyondcode"), _tcslen( _T("Beyondcode")), &rctText, DT_CENTER |
DT_SINGLELINE | DT_VCENTER );
```

```
EndPaint( hwnd, &ps );


break;


}
```

复制代码首先定义了三个整形常量 buttonWidth，buttonHeight 指示刚才创建的按钮的大小，textHeight 指示要显示在文本的矩形框的高度，矩形框的宽度和主窗口的宽度一直，所以就没定义了，然后 PAINTSTRUCT 是 BeginPaint 和 EndPaint 这两个函数会用到的一个结构体类型，用它定义了一个结构体变量 ps，并在调用 BeginPaint 和 EndPaint 的时候将它的地址传递给他们的第二个参数。获取一些相关和绘图有关的信息。不过我们这里不会用到，所以就不做详细解释，可以查看 MSDN。

注意，BeginPatin 这个函数会返回一个设备句柄，然后我们就可以通过这个设备句柄进行绘图，显示文字也是一种绘图，在绘图完毕后，我们需要调用 EndPaint 这个函数释放刚才得到的哪个设备句柄，也就是是刚才哪个设备句柄无效。而所有的绘图操作，都必须在 BeginPaint 和 EndPatin 这两个函数之间完成。如上面，通过参数 hdc 调用 DrawText 这个函数，因为获取的 hdc 是通过 hwnd 这个窗口句柄的，所以这里所有的绘图都会显示在 hwnd 这个句柄所代表的窗口上，也就是主窗口。rctText 是显示文本的矩形的信息，它的大小和位置是通过按钮的大小和当前主窗口的大小信息计算出来的，具体的计算代码中已经写的很清楚了，如有疑惑的可以和我交流交流。 然后还要说的一个就是 DT_CENTER 和 DT_VCENTER 这两个标志表示在刚才那个矩形框中的水平中央和垂直中央显示我们的文本，DT_SINGLELINE 就是指示单行显示。

最后留给大家一个问题，以供大家思考，上面的程序中，当你改变窗口的大小的时候，就会出现问题，按钮就不会再位于主窗口的中央了，怎么解决呢？我提示一下吧,处理 WM_SIZE 这个消息。好了，留下这个任务给大家，试试吧~~让按钮随时随地位于主窗口的中央。

上一篇《自己实现 MessageBox》中我们基本已经实现了一个对话框了，可以在中央显示自己的文字，并且显示一个确定按钮，可是，上一篇完的时候我留下了一个问题，那就是那个确定按钮并不会根据窗口的大小的改变而改变。那么我们怎么来解决这个问题呢?

我给出了提示可以通过处理 WM_SIZE 来完成这个目的。那么今天我们就来完成这一遗留的问题。所要使用到的新的 API 函数也不多，就两个，要处理的消息也就两个 WM_SIZE 和 WM_COMMAND

GetDlgItem

MoveWindow

这么两个，至于其他的 API 函数，都是我们以前接触过的，如果你忘记了，可以自己复习一下使用方法。

首先，我们先看 WM_SIZE 消息处理函数是怎么写的 case WM_SIZE:

{

```
const int buttonWidth = 80;

const int buttonHeight = 25;

int buttonx, buttony;

RECT rctClient;

GetClientRect( hwnd, &rctClient );

buttonx = rctClient.right/2 - buttonWidth/2;

buttony = rctClient.bottom/2 - buttonHeight/2;
```

```
HWND hButton = GetDlgItem( hwnd, 2 );

MoveWindow( hButton, buttonx, buttony, buttonWidth, buttonHeight, TRUE );

}

break;
```

复制代码由于这篇文章内容比较少，那么我就可以详细的介绍一下 WM_SIZE 这个消息处理函数中实现按钮始终保持居中的代码。WM_SIZE 这个消息是当一个窗口的 size 也就是大小被改变后而被发送到该窗口的消息处理函数的。我们在这里通过截获 WM_SIZE 就可以在每次窗口大小被改变的时候进行一些处理，我们这里的处理就是将该窗口上的一个子窗口，也就是那个确定按钮移动到中央。

首先，我们还是老规矩，定义 buttonWidth,和 buttonHeight 两个常整形来存放确定按钮的长和高。然后定义了两个整形变量 buttonx，和 buttony 用来存放后面通过计算得到的确定按钮的左上角的坐标位置。

然后定义了一个 RECT 结构体用来保存后面通过 GetClientRect API 函数获取的窗口的长宽，其中 rctClient 中 right 就保存了窗口的长，bottom 就保存了窗口的高。我们为了让按钮保持在主窗口的中央，那么我们就需要让按钮的左上角的 x 坐标位置在主窗口的长的一半再减去按钮的长的一半的位置。高也是一样的原理。所以 buttonx = rctClient.right/2 - buttonWidth/2; buttony = rctClient.bottom/2 - buttonHeight/2; 这两句就是根据当前主窗口的长和高计算按钮应该在的位置。

计算完成后，我们就只需要移动按钮就可以了。可是移动按钮之前，我们需要获得按钮的句柄，这个句柄怎么获得呢，有很多中方法，这里我就用 GeDlgItem 这个 API 函数来获取，它需要两个参数，第一个参数是一个主窗口的句柄，这里我们就传递按钮的主窗口的句柄 hwnd，第二个参数是按钮的一个标识符，因为我们在前一篇文章中用 CreateWindowEx 创建子窗口的时候给按钮指定的标识符是 2，所以这里我们就传递 2，那么这样 GetDlgItem 返回的就是这个按钮的句柄了。

得到了句柄后，我们就需要用 MoveWindow 来移动这个子窗口按钮，到我们需要的位置了。第一个参数是这个字窗口的句柄，也就是我们上面获得的句柄，第二个参数和第三个参数是移动到的 x，y 坐标。这里我们传递 buttonx 和 buttony，第四个和第五个是移动的窗口的长和高，如果同时还需要改变窗口的长和高，那么这里也可以传递改变后的长和高的值，我们

这里只移动位置，不改变大小，所以就传递 buttonWidth 和 buttonHeight。最后一个参数是一个 BOOL 型的，指示是否需要重绘，这里传递 TRUE，也就是让它在移动后进行重绘。

好了，现在，当你改变主窗口的大小的时候，里面的确定按钮也会跟着改变位置而达到始终保持在主窗体的中央。

可是还有一个问题就是，当我们点击按钮的时候，程序没有任何的反映， MessageBox 的确定按钮被点击的时候一般都会关闭当前对话框，所以我们这里也需要实现当用户点击确定按钮的时候，将我们的主窗体关闭。那么怎么来实现呢。

在实现之前，我首先要讲一讲，子窗体是怎么通知他们的父窗体的，比如说按钮被点击的时候是怎么通知他们的父窗体的。其实一般就是通过 WM_COMMAND 来通知的，例如我点击这个确定按钮，那么在这个确定按钮的窗口消息处理函数中就会向它的父窗体的窗口消息处理函数发送一条 WM_COMMAND 消息，并且 WM_COMMAND 消息的 wParam 参数的低 16 包含的就是一个标识符，指示是哪个子窗体发送的这条消息。至于 wParam 的高 16 和 lParam 包含的是些什么信息，就请各位自己查阅 MSDN 了，这里我们不会用到，也就不做讲解了。

所以我们要处理在子窗体上发生的事情，就需要在父窗体的消息处理函数中截获 WM_COMMAND 消息，并进行处理。那么这里的 WM_COMMAND 消息处理也很简单，如下 case WM_COMMAND:

```
{

if( LOWORD(wParam)==2 )

{

DestroyWindow( hwnd );

}
```

}

break;

复制代码就是用 LOWORD 这个宏来取出 wParam 的低 16 位，并且判断是不是 2，也就是判断是不是确定按钮的标识符，如果是，就表示确定按钮上发生了事件，具体的事件我们就没做过细的判断了，一般来说都是指被点击。所以我们就进行处理，调用 DestroyWindow 这个 API 来销毁主窗体。就达到了我们的目的了。

怎么样，比较简单吧~

经过 7 篇 API 入门系列文章的介绍，我想你对 WIN32 API 编程的一般流程还是有了一个大概的了解了吧。以及对于 windows 的数据类型，字符编码方面。API 的使用方面，消息的处理方面，因为都不会陌生了吧·

所以我后面的文章，对于细节就不会这么细了，对于一个 API 函数，如果参数不是很复杂，我也不会做过多的解释了。而只是说明一下它的作用。至于细节，各位就应该养成 MSDN 的习惯了。