

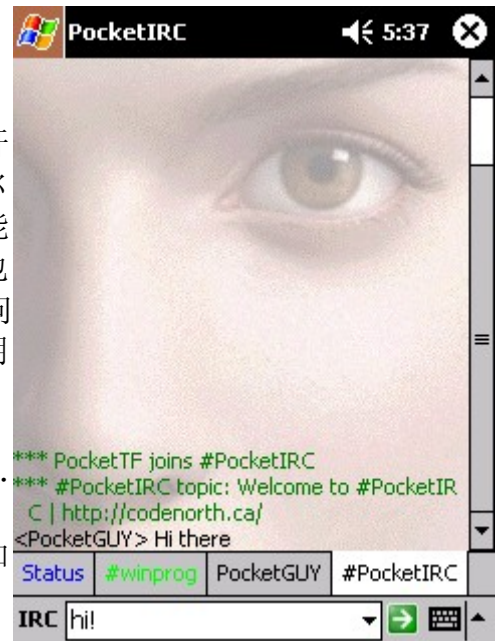
欢迎阅读 theForger's Win32 API 教程第二版(简体中文)

作者: [Brooks Miles](#)

译者: [湛宗儒](#)

本教程试图使用尽可能快和尽可能清晰的方法教你开始 Win32 API 开发. 它是以一个整体来组织的, 所以在你提问之前请从头到尾看一遍. . . 你的大多数问题很可能在文字中已被回答. 每个章节以之前的章节为基础. 我也在附录 A 中附上了一些常见的错误的解决方法. 如果你问一些在教程中已被回答的问题的话, 看起来就有点不聪明了.

- 下载[完整的范例源代码](#), 在整个教程都有对代码的引用.
- 或是下载[整个教程\(包括源代码\)](#)至你自己的计算机上慢慢看. 下到硬盘的版本可能不包括网络版本具有的一些如拼写更正之类的小规模修改.



如果你在一个别的站点上看此教程, 请访问[#winprog](#) 站点看最新的官方版本.

- [想做点什么?](#)
- [需要进一步的帮助?](#)

目錄

· 基础

1. [开始学习](#)
2. [一个简单的窗口](#)
3. [处理消息](#)
4. [理解消息循环](#)
5. [使用资源](#)
6. [菜单和图标](#)
7. [对话框, 图形界面设计者的好朋友](#)
8. [非模态对话框](#)
9. [标准控件: 按钮, 编辑框, 列表框, 静态控件](#)
10. [等等, 我还想问. . . \(对话框常见问题\)](#)

· 创建一个简单应用

1. [应用第一部分: 在运行时创建控件](#)
2. [应用第二部分: 使用文件与常用对话框](#)
3. [应用第三部分: 工具栏与状态栏](#)
4. [应用第四部分: 多文档界面](#)

· 图形设备界面

1. [位图, 设备上下文与 BitBlt](#)
2. [透明位图](#)

3.[定时器与动画](#)

4.[文本，字体与颜色](#)

· 工具与文档

1.[推荐的书与参考](#)

2.[免费的 Visual C++ 命令行工具](#)

3.[免费的 Borland C++ 命令行工具](#)

· 附表

附表 A: [常见错误的解决方法](#)

附表 B: [为什么要在学 MFC 编程之前学习 API](#)

附表 C: [关于资源文件](#)

我听某些读者说教材中的源代码在一些很旧的 Netscape 浏览器中不能正确地换行，如果你遇到此问题请参考 zip 打包下载的源代码。

想做点什么？



你可以绝对免费地使用此文档，但是把它放在互联网上的确是有些费用。如果你感觉它对你有帮助，也想回馈一些，我将很感谢你能捐赠任何数目的款项来协助支撑此网站。此页面

每月大约有 15,000 个点击，并且在一直增加：)

再说一次，你完全没有义务支付，你也不会因为支付而得到除了在此处之外的任何东西，但是你想协助的话，是很好的。．．．就点那个 PayPal 图片就行。

但愿你能享受阅读，
Brook

我想对如下几个人做出的贡献表示感谢: Yih Horng, Todd Troxell, T Frank Zvovushe, Suzanne Lorrin, Seth McCarus, Crispina Chong, John Crutchfield, Scott Johnstone, Patrick Sears, Juan Demerutis, Richard Anthony, Alex Fox, Bob Rudis, Eric Wadsworth, Chris Blume. 还有那些写信告诉我此文档有用的那些读者。我很高兴！

需要进一步的帮助？

一般地话我会免费回复求助邮件，或指出在哪里可以找到可供参考的资源。

现在我正忙于几个大项目，没有时间跟你写一些特定的范例或一定规模的软件项目。但是我愿意接项目：)

尽管[联系我](#)。

开始学习

这篇教程讲什么

这篇教程试图向你展现使用 Win32 API 写程序的一些基础知识（还有常见的扩展知识）。使用 C 语言，大多数的 C++ 工具也可以编译。事实上从大多数语言的介绍文章可知，大多数语言皆可调用 API，包括 Java，汇编语言以及 Visual Basic。但是我将不会提到这些语言的代码例子，你想使用其它语言的话，请自己参考相关资料，不过有好几个使用过此教程的人都对我说用上面的这些语言都是可行的。

本教程不会教你 C 语言，也不会教你怎么使用你喜欢的编译器（Borland C++，Visual C++，LCC-Win32，等等）。但是我将 在附表中就我对编译器的所知提供一些说明。

如果你不知道 macro 或 typedef 是什么，或 switch() 语句如何工作，那你要先回去找一本好的 C 语言的教程学习一下。

重要说明

在此文档的某些部分我将指出某些地方很重要。因为很多人在不阅读它们情况下造成理解困难，你如果不阅读，你很可能也陷入困难。第一个就是：

以 zip 打包的源代码范例不是可选可不选的！我没有把所有的代码放在教程中，只放了那些与我正在讨论问题相关的。要想知道这里的代码怎么与其它部分配合，就必须去看 zip 文件中的源代码。

好！第二个：

把整个文档看完。如果你在 读某章节遇到了问题，请耐心等待，很可能在后面一点就可以找到答案。如果你实在不能忍受这种无知的状态，请在到 IRC 频道上去提问或发出求助邮件之前至少跳过一点或在余下的文档中搜一下(是的，计算机可以搜索)。

另外一点就是一个关于话题 A 的问题很可能在关于话题 B 或 C 的讨论中得到解答，也有可能是话题 L。所以多看看，找一下。

好，东扯西拉暂时告一段落，我们来试些实际代码。

最简单的 Win32 程序

如果你是一个完全的新手，就让我们来确认一下你可以编译一个基本的 windows 程序。把下面这些代码弄到你的编译器中去编译一下，如果一切正常你就得到有史以来最简易的程序之一。

记得以 C 来编译，不是以 C++。可能没有关系，但这里的代码都是 C，在正确的轨道，行驶还是好些。大多数情况，你要做的就是 把文件的扩展名写成 .c 而不是 .cpp。如果这些话伤了你的脑筋的话，就把文件名写成 test.c 并用它就行了。

```
#include <windows.h>
```

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MessageBox(NULL, "Goodbye, cruel world!", "Note", MB_OK);
    return 0;
}
```

如果不行，首先阅读得到的任何错误提示，并在帮助文档或任何其它跟你编译器配套的文档中查找它们。确定你是以一个 Win32 GUI(不是 Console)的工程/makefile/目标来编译的。不幸的是，这一点上我也帮不了什么，对于不同的编译器(不同的人)，解决方法不同。

你可能得到一些警告说你沒有使用 WinMain()传递的那些参数。这没关系。现在我们确定你能编译一个程序了，我们来看一下代码。...

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

WinMain()是 Windows 中与 DOS 或 UNIX 的 main()的等价物。这是你的程开始执行的入口。参数如下：

HINSTANCE hInstance

程序可执行模块的句柄(内存中的.exe 文件)。

HINSTANCE hPrevInstance

在 Win32 程序中总是为 NULL。

LPSTR lpCmdLine

命令行参数组成的一个单字符串。不包括程序名字。

int nCmdShow

一个将要传递给 ShowWindow()的整数，我们在后面进行讨论。

hInstance 用作装入资源或其它的以模块为单位的任务。一个模块是一个装入到你程序的 exe 或 dll。对于本教程的大多数部分(如果不是全部的话)，我们只关心一种模块，就是 exe 模块。

hPrevInstance 在 Win16 时代曾经用作你程序的前面已经运行的实例（如果有的话）。现在已经不用了，在 Win32 中你忽略它就行了。

调用规则

WINAPI 指定调用规则并被定义为_stdcall。要是你不知道它是干什么用的，先不管它，在我们的这个教程中它对我们沒有影响。记住在这个位置我们需要它就是了。

Win32 数据类型

你会发现很多普通的关键字或类型在 windows 中有特定的定义。UINT 是 unsigned int，LPSTR 是 char*等等。... 你怎么用完全取决于你自己。你要是喜欢 char*超过了 LPSTR，那就用就是了。当然在你替换一个数据类型前你要确定你知道它是什么。

就记住一些容易记住的东西就夠了。LP 前缀代表 Long Pointer。在 Win32 中，Long 这

个部分已经是过时的概念，不要管它。要是不知道指针是什么的话，你可以 1)去找一本好的 C 语言教程，或 2)直接往下读，弄得头脑混乱。我是推荐第一种方案的，但很多人使用第二种(我也是：)。到时候别说我没有提醒你。

接下来，一个 C 接在 LP 后面表示是常量指针。LPCSTR 表示一个指向不会也不能被修改的常量字符串的指针。LPSTR 指向的就不是常量的，可以被修改。

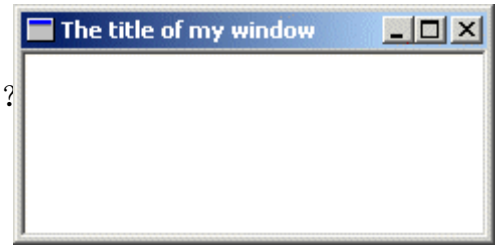
你可能还会看到一个 T 混在里面。现在不要管它，除非你打算与 Unicode 打交道，它没有其它的意义。

Copyright © 1998-2003, Brook Miles ([theForger](#)). All rights reserved.

一个简单的窗口

范例：simple_window

有时候有人在 IRC 上问：“我怎么才能实现一个窗口？”... 我觉得不是一句两句能说得清楚。虽然一旦你搞清楚你要做什么后并不难，但是你的确需要做一些事情来使显示一个窗口；这些事情不是在聊天室中一下子说得清楚的。



我总是喜欢先做一件事情然后来理解它... 所以先给出一个简单窗口的代码稍后再做解释。

```
#include <windows.h>
const char g_szClassName[] = "myWindowClass";
// Step 4: the Window Procedure
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch(msg)
    {
        case WM_CLOSE:
            DestroyWindow(hwnd);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;
    HWND hwnd;
    MSG Msg;
    //Step 1: Registering the Window Class
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = 0;
    wc.lpfnWndProc = WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
```

```

wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
wc.lpszMenuName = NULL;
wc.lpszClassName = g_szClassName;
wc.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
if(!RegisterClassEx(&wc))
{
    MessageBox(NULL, "Window Registration Failed!", "Error!",
        MB_ICONEXCLAMATION | MB_OK);
    return 0;
}
// Step 2: Creating the Window
hwnd = CreateWindowEx(
    WS_EX_CLIENTEDGE,
    g_szClassName,
    "The title of my window",
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, CW_USEDEFAULT, 240, 120,
    NULL, NULL, hInstance, NULL);
if(hwnd == NULL)
{
    MessageBox(NULL, "Window Creation Failed!", "Error!",
        MB_ICONEXCLAMATION | MB_OK);
    return 0;
}
ShowWindow(hwnd, nCmdShow);
UpdateWindow(hwnd);
// Step 3: The Message Loop
while(GetMessage(&Msg, NULL, 0, 0) > 0)
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}
return Msg.wParam;
}

```

简单来说这是你想创建一个窗口能写的最简单的程序，大致上 70 行左右的代码。如果第一个范例你通过了编译，这个也应该不成问题。

第一步:注册窗口类

一个窗口类存储关于一个窗口的消息，包括控制窗口的窗口过程，窗口的大小图标，以及背景颜色。用这种方式，你可以先注册一个窗口类，然后从它创建任意数目的窗口，而不需要一次次的指定这些参数。如果需要，大多数属性能针对单个的窗口来改变。

这里说的窗口类与 C++ 中的类是完全不同的概念。


```
const char g_szClassName[] = "myWindowClass";
```

上面的变量存储了我们窗口类的名字，马上会用来向系统注册窗口类。

```
WNDCLASSEX wc;

wc.cbSize      = sizeof(WNDCLASSEX);
wc.style       = 0;
wc.lpfnWndProc = WndProc;
wc.cbClsExtra  = 0;
wc.cbWndExtra  = 0;
wc.hInstance   = hInstance;
wc.hIcon       = LoadIcon(NULL, IDI_APPLICATION);
wc.hCursor     = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
wc.lpszMenuName = NULL;
wc.lpszClassName = g_szClassName;
wc.hIconSm     = LoadIcon(NULL, IDI_APPLICATION);
if(!RegisterClassEx(&wc))
{
    MessageBox(NULL, "Window Registration Failed!", "Error!",
        MB_ICONEXCLAMATION | MB_OK);
    return 0;
}
```

这是我们在 WinMain() 中注册我们的窗口类的代码。我们填写一个 WNDCLASSEX 结构体的成员并调用 RegisterClassEx()。

结构体的成员对窗口类的影响如下：

cbSize

结构体的大小。

style

类的式样(CS_*), 不要跟窗口式样(WS_*)混淆了.这个一般设置为 0.

lpfnWndProc

指向这个窗口类的窗口过程的指针。

cbClsExtra

配置给这个类的额外内存.一般为 0.

cbWndExtra

配置给这个类的每个窗口的额外内存.一般为 0.

hInstance

应用程序实例的句柄。（从 WinMain() 第一个参数传递来。）

hIcon

当用户按下 Alt+Tab 组合时候显示的大图标（一般为 32*32）。

hCursor

在我们的窗口上显示的光标。

hbrBackground

设置我们窗口背景颜色的背景刷子。

lpzMenuName

这个类的窗口所用的菜单资源的名字。

lpzClassName

类的名字。

hIconSm

在任务栏和窗口的左上角显示的小图标(一般为 16*16)。

如果看得不很明白先不要担心，马上会有讲解。另外要记住的一件事情是不要试图去记下这些东西。我基本上不（从来不）记结构体或函数的参数，这样做是浪费精力和更重要的时间。如果你知道要调用的函数你就去花几秒钟在你的帮助文档中查一下。要是没有帮助文档，就是想法弄到。没有它们是很郁闷的。随著你使用多次后也许你会记住那些最常用的函数的参数。

然后我们调用 RegisterClassEx()并检查是否成功，如果失败我们弹出一条提示消息并从 WinMain()函数退出程序。

第二步:创建窗口

一旦类注册完，我们即可从它创建一个窗口。你应该去查一下 CreateWindowEx()的参数列表（在你要用一个新 API 的时候你总是要这样做），但是我会在这里做个简单的介绍。

```
HWND hwnd;  
hwnd = CreateWindowEx(  
    WS_EX_CLIENTEDGE,  
    g_szClassName,  
    "The title of my window",  
    WS_OVERLAPPEDWINDOW,  
    CW_USEDEFAULT, CW_USEDEFAULT, 240, 120,  
    NULL, NULL, hInstance, NULL);
```

第一个参数（WS_EX_CLIENTEDGE）是扩展的窗口式样，这里我设置它想得到一个内部下陷的边框效果。你可以设成 0 看看效果。要习惯于试不同的值看效果。

接下来我设置了类的名字（g_szClassName），告诉系统我们要创建什么样的窗口。因为我们要从刚刚注册的类创建窗口，我们使用了该类的名字。之后我们指定了我们窗口的名字或是标题，用来显示在我们窗口的外观或是标题栏上。

我们设置的 WS_OVERLAPPEDWINDOW 是一个窗口式样参数。这里有很多东西你可以查找并做试验。在下文中将详加说明。

接下来的四个参数（CW_USEDEFAULT, CW_USEDEFAULT, 320, 240）是我们窗口的左上角的 X, Y 坐标和其宽度和高度。我把 X, Y 坐标设为 CW_USEDEFAULT 来让系统自己选择在屏幕的哪个地方来放置窗口。记住屏幕的最左边的 X 坐标为 0 并向右加；屏幕的顶部的 Y 坐标为 0 并向底加。单位是像素，这是屏幕在特定的分辨率下能显示的最小单位。

再接下来的四个（NULL, NULL, g_hInst, NULL）分别是父窗口的句柄，菜单句柄，

应用程序实例句柄，和窗口创建数据的指针。在 windows 系统中，你屏幕上的窗口是以分层次的父窗口，子窗口的形式来组织的。当你看到一个窗口中有一个按钮时候，按钮就是子窗口，包含它的窗口就是父窗口。我们的例子中，父窗口的句柄为 NULL，因为这里没有父窗口，这个是我们的主窗口或是顶层窗口。菜单也是 NULL，因为我们现在也没有菜单。实例句柄设为我们从 WinMain() 得到的第一个参数。窗口的创建数据(我们几乎没有使用)可以用来向创建的窗口发送额外数据在这里也设为 NULL。

如果你想知道神奇的 NULL 是什么，它实际上被定义为 0(零)。而且在 C 中，它是被定义为((void*)0)的，因为是用来作指针用的。所以你有可能会在把 NULL 当整数使用时会得到警告，跟你的编译器的警告级别设置有关。你可以忽略这个警告，也可以用 0 来代替。

不检查调用是否成功几乎是程序员为找不到程序究竟错在哪里而烦恼的最常发生的情况。CreateWindow() 可能会调用失败，即使你是一个用经验的程序员，原因就是可能犯错误的地方实在太多了。除非你学会了如何快速查出错误，最少你应该给自己一个机会知道是哪里出了错，一定记住检查返回值！

```
if(hwnd == NULL)
{
    MessageBox(NULL, "Window Creation Failed!", "Error!",
        MB_ICONEXCLAMATION | MB_OK);
    return 0;
}
```

在我们创建了窗口并作检查以确定我们有一个正确的句柄后，我们使用 WinMain() 最后的参数来显示窗口，再更新它以确认它在屏幕上正确地重画了自己。

```
ShowWindow(hwnd, nCmdShow);
UpdateWindow(hwnd);
```

nCmdShow 是可选的参数，你可以简单地传递 SW_SHOWNORMAL 即可。但是用从 WinMain() 传来的参数可给予运行此程序的用户选择以可视，最大化，最小化等选项。... 来引导程序的自由。你可以在 windows 快捷方式的属性中看到这些选项，参数由选项来决定。

第三步:消息循环

这是整个程序的心脏，程序中几乎所有的控制都从这个点传递。

```
while(GetMessage(&Msg, NULL, 0, 0) > 0)
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}
return Msg.wParam;
```

GetMessage() 从你应用的消息队列中取一个消息。任何时候用户移动鼠标，敲击键盘，点击你窗口的菜单，或做别的什么事，系统会产生消息并输入到你的程序的消息队列中去。调

用 GetMessage()时你请求将下一个可用的消息从队列中删除并返回给你来处理。如果队列为空，GetMessage()阻塞。如果你不熟悉这个术语，它意味著一直等待直到得到一个消息，才返回给你。


TranslateMessage()为键盘事件做一些额外的处理，如随著 WM_KEYDOWN 消息产生 WM_CHAR 消息。最后 DispatchMessage()将消息送到消息应该被送到的窗口。可能是我们的主窗口或另外一个，或一个控件，有些情况下是一个被系统或其它程序创建的窗口。这不是你需要担心的，因为我们主要关心我们得到消息并送出，系统来做后面的事以确认它到达正确的窗口。

第四步：窗口过程

如果说消息循环是程序的心脏，那个窗口过程就是程序的大脑。这里所有送到窗口的消息被处理。

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

窗口过程在每个消息到来时被调用一次，HWND 参数是消息相应的窗口的句柄。这很重要因为你可能用相同的类创建了两个或多个窗口并且它们用相同的窗口过程(WndProc())。不同点就在不同的窗口有不同的 hwnd 参数。比如我们得到 WM_CLOSE 消息我们就要销毁那个窗口。我们使用了窗口句柄作为我们得到的第一个参数，其它的窗口都不会受影响，除了那个我们想要操作的之外。

WM_CLOSE 是在我们按下关闭按钮  或按下 Alt+F4 组合时产生的。这默认会使窗口销毁，但我喜欢显式处理它，因为这是在程序退出之前做清除检查，或询问用户是否保存文件等事情的绝佳的位置。

当我们调用 DestroyWindow()系统向要销毁的窗口送出 WM_DESTROY 消息，这里是我们的窗口，并在从系统移除我们的窗口之前删除它剩下的所有的子窗口。因为这是我们的程序中唯一的窗口，我们准备好了并希望程序退出，所以我们调用了 PostQuitMessage()。这样会向消息循环发出 WM_QUIT 消息。我们不永远收不到这个消息，因为它使 GetMessage()返

回 FALSE，而且你也可以看到我们的消息循环代码中，这时候我们停止处理消息并返回最终的结果码，我们传递给 PostQuitMessage() 的 WM_QUIT 中的 wParam 部分。这个返回值只有在你的编程为被别的程序调用并且你需要一个确定的返回值时候才有真正有用。

第五步：沒有第五步

Phew（喘一口气），到这先告一段落.如果上面还没有解释清楚的话，还是先放下，也许我们在讲解更有用的程序时这些东西会变得清晰起来。

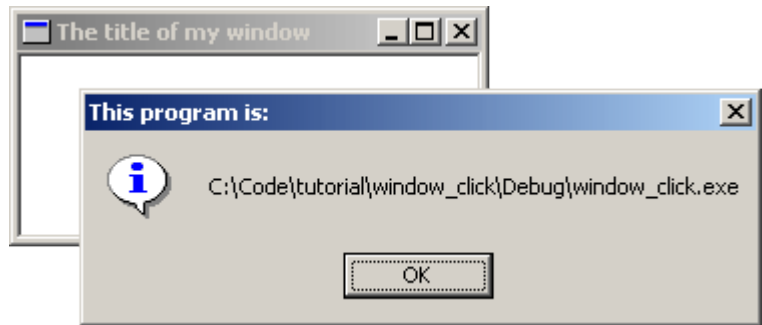
Copyright © 1998-2003, Brook Miles ([theForger](#)). All rights reserved.

处理消息

范例：window_click

不错，我们有个窗口了，不过除了 DefWindowProc() 允许它做的，如拉伸，最大化等等之外没有别的什的功能了，并不是很令人激动。

下一章节我将演示如何修改你已有的代码来加点新东西。现在我只来告诉你”处理这个消息，在这里处理...



... ”你就知道我的意思，知道怎样做且不用看一整个例子。希望如此，所以注意看:P

对于初学者，请把我们最后那个窗口编译好并确信它能工作。然后你要么就在这个例子上修改要么拷到另外一个新工程中来修改。

我们准备加个使用户点击我们的窗口时候能够显示出我们程序的名称的功能。不是很酷，就是基本的一个消息的处理。让我们看看我们的 WndProc() 中有些什么：

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

如果我们想处理鼠标的点击，我们需要添加一个 WM_LBUTTONDOWN 的处理部分（对于右键与中间键，分别是 WM_RBUTTONDOWN 与 WM_MBUTTONDOWN）。

如果你听到我或其它人提到处理消息，就是说在窗口的类的 WndProc() 中加上如下代码：

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch(msg)
```

```

{
    case WM_LBUTTONDOWN: // <-
                                // <- 这就是我们加的
        break; // <-
    case WM_CLOSE:
        DestroyWindow(hwnd);
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hwnd, msg, wParam, lParam);
}
return 0;
}

```

不同消息处理的代码的顺序基本上没有什么关系。要记住的是一定不要忘了在每个消息处理代码后写上 `break` 语句。你可以看到我们在我们的 `switch()` 中加了另外一个 `case`。现在我们要在我们的程序执行至此时发生点什么。

首先我们把我们要加的代码写出来（显示我们程序的文件名）然后整合到我们的程序中去。后面的章节中我将只写出代码让你自己整合到程序中去。对我来说不用打那么多字了，对你来说也更灵活，不只是这里的程序还可以加到任何程序中去。如果你不知道怎么办，请参考样例 `zip` 文档中对应此章的部分。

```

GetModuleFileName(hInstance, szFileName, MAX_PATH);
MessageBox(hwnd, szFileName, "This program is:", MB_OK | MB_ICONINFORMATION);

```

这段代码不是自解释的，你不能把它随便敲到你原来的代码中去。我们只想在用户用鼠标点击的时候运行这段代码，所以我们把它加到我们的消息处理框架中去：

```

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch(msg)
    {
        case WM_LBUTTONDOWN:
            // BEGIN NEW CODE
            {
                char szFileName[MAX_PATH];
                HINSTANCE hInstance = GetModuleHandle(NULL);
                GetModuleFileName(hInstance, szFileName, MAX_PATH);
                MessageBox(hwnd, szFileName, "This program is:", MB_OK |
                    MB_ICONINFORMATION);
            }
            // END NEW CODE
    }
}

```

```

        break;
    case WM_CLOSE:
        DestroyWindow(hwnd);
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hwnd, msg, wParam, lParam);
}
return 0;
}

```

注意那对新的花括号`{}`。当你在`switch()`中定义变量时需要它。这本来是基础的C语言知识，但我认为我应该在此指出来，为那些理解有困难的读者著想。

如果你已经加进了代码，现在就编译它。如果正常的话，你点击窗口你就应该看到一个有.exe字样的对话框弹出来。

你可能已经注意到我们加了两个变量，`hInstance`与`szFileName`。查一下`GetModuleFileName()`你就会看到第一个参数是`HINSTANCE`。指向可执行文件（我们的程序，.exe文件）。我们哪里得到这样一个东西？`GetModuleHandle()`给出了答案。`GetModuleHandle()`的说明指出传给它一个`NULL`会返回一个创建发出调用进程的文件句柄，正是我们刚刚提到的`HINSTANCE`，是我们需要的。把这些消息集中起来我们就会得到如下的定义：

```
HINSTANCE hInstance = GetModuleHandle(NULL);
```

对于第二个参数，我们再次转向我们可以信赖的参考文档，我们看到它是”一个用来接收那个模块的文件名和路径的缓冲区的指针”而且数据类型是`LPTSTR`（要是你的文档是比较旧的话就是`LPSTR`）。因为`LPSTR`等价于`char*`，我们就定义一个字符串：

```
char szFileName[MAX_PATH];
```

`MAX_PATH`是在`<windows.h>`中定义的一个宏，用来定义Win32中文件名缓冲区的最大长度。我们也将`MAX_PATH`传给`GetModuleFileName()`以使它知道缓冲区的大小。

`GetModuleFileName()`被调用后，`szFileName`将包括一个我们的.exe文件名的字符串，以`null`结尾。我们把它传递给`MessageBox()`以一个简单的方法向用户显示。

所以如果你加了代码，编译了。如果正常，点击窗口你就会看到一个有.exe文件名的对话框弹出来。

如果不是这样，这里是完整的代码跟你的比较一下，看错误在哪里。

```
#include <windows.h>
```



```

const char g_szClassName[] = "myWindowClass";
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch(msg)
    {
        case WM_LBUTTONDOWN:
        {
            char szFileName[MAX_PATH];
            HINSTANCE hInstance = GetModuleHandle(NULL);
            GetModuleFileName(hInstance, szFileName, MAX_PATH);
            MessageBox(hwnd, szFileName, "This program is:", MB_OK | MB_ICONINFORMATION);
        }
        break;
        case WM_CLOSE:
            DestroyWindow(hwnd);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;
    HWND hwnd;
    MSG Msg;
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = 0;
    wc.lpfnWndProc = WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = g_szClassName;
    wc.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
    if(!RegisterClassEx(&wc))

```

```
{
    MessageBox(NULL, "Window Registration Failed!", "Error!",
        MB_ICONEXCLAMATION | MB_OK);
    return 0;
}
hwnd = CreateWindowEx(
    WS_EX_CLIENTEDGE,
    g_szClassName,
    "The title of my window",
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, CW_USEDEFAULT, 240, 120,
    NULL, NULL, hInstance, NULL);
if(hwnd == NULL)
{
    MessageBox(NULL, "Window Creation Failed!", "Error!",
        MB_ICONEXCLAMATION | MB_OK);
    return 0;
}
ShowWindow(hwnd, nCmdShow);
UpdateWindow(hwnd);
while(GetMessage(&Msg, NULL, 0, 0) > 0)
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}
return Msg.wParam;
}
```

理解消息循环

在写任何有实际价值的程序之前，都要理解整个消息循环和 windows 程序传递消息的结构。当前为止我们已经试了一些消息的处理，我还应该更进一步的看一下这个过程，因为你如果不理解它们的话，后面的内容将会对你很难。

什么是消息

一个消息就是一个整数。如果你到头文件中看一下（了解 API 的好地方）你就会发现这样的内容：

```
#define WM_INITDIALOG      0x0110
#define WM_COMMAND        0x0111
#define WM_LBUTTONDOWN    0x0201
```


...

等等.. 消息用来进行 windows 系统中的所有通信，至少在基本方面是这样的。如果你想要你的窗口或是控件（其实就一种特别的窗口）做点什么你就给它发个消息。如果另外一个窗口要你做什么它也发给你一个消息.. 如果发生了用户按动了键盘，移动了鼠标，点击了一个按钮之类的事件，系统就向相关的窗口发送消息。你要是就是那个被传至消息的窗口，你就要处理这些消息。

每个 windows 消息可能拥有至多两个参数，wParam 和 lParam。最初 wParam 是 16bit，lParam 是 32bit，但在 Win32 平台下两者都是 32bit。不是每个消息使用了这些参数，而且每个消息以不同的方式来使用。比如，WM_CLOSE 不使用它们任一个，所以你应该忽略它们。WM_COMMAND 消息两个都使用，wParam 有两个部分，HIWORD(wParam)中含有提示消息（如果有的话），LOWORD(wParam)含有发送消息的控件或菜单的标识号。lParam 含有发送消息的控件的 HWND（窗口的句柄）或者为 NULL，当消息不是由控件发送。

HIWORD()和 LOWORD()是 windows 定义的两个宏，用来从一个 32bit 值中分离出高字 (0xffff0000)和低字(0x0000ffff)的两字节。在 Win32 中，一个 WORD 是 16bit，所以一个 DWORD 为 32bit。

可以用 PostMessage()或 SendMessage()来发送消息。PostMessage()把消息放入消息队列再立即返回。就是说你调用了 PostMessage()后消息可能被处理了，也可能还没有被处理。SendMessage()则真接把消息送往窗口并且在窗口没有结束处理消息之前不返回。如果我们想关闭一个窗口我们可以发送一个 WM_CLOSE 消息：

PostMessage(hwnd, WM_CLOSE, 0, 0);这跟我们点击窗口顶部的 按钮一样的效果。注意 wParam 和 lParam 都为 0。这是因为我们刚才说了 WM_CLOSE 并不用它们。

对话框

一旦你开始使用对话框，你将需要向此控件发送消息以与它们通信。你可以先使用 GetDlgItem()来用 ID 得到控件的句柄然后用 SendMessage()，也可以直接用 SendDlgItemMessage()这个一步到位的函数。你给它一个窗口的句柄和一个子窗口的 ID，它就会得到子窗口的句柄并向它发送消息。SendDlgItemMessage()和类似的 API，如

GetDlgItemText()可以用在所有的窗口上面，而不是仅仅在对话框上。

什么是消息队列

打个比方，你正在处理 WM_PAINT 消息，此时突然用户在键盘上敲了一大堆的东西。这时候会怎样？你应该中止你的工作去响应键盘的输入，还是简单地把这些输入给忽略掉？错！显然两种方式都不能接受，所以我们引入了消息队列的概念，消息被发送时就被放入队列，被处理后就从队列中删除。这就保证了你不会丢掉消息，你在处理某个的时候，另外的就在队列中等待你来取走它们。

什么是消息循环

```
while(GetMessage(&Msg, NULL, 0, 0) > 0)
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}
```

1. 消息循环调用了 GetMessage()，它到你的消息队列中去看。如果队列是空的你的程序就在那里等（阻塞在那里）。
2. 当一个事件发生导致一个消息加到队列中去（比如系统注册了一次鼠标点击）GetMessage()就返回一个正的值表示有一个消息待处理，并且它把我们传递的 MSG 结构体填充。如果遇到了 WM_QUIT 它就返回 0，如果有错误发生就返回负值。
3. 我们拿到信号（在 Msg 变量中）并传给 TranslateMessage()，它进行一些额外的处理，将虚键值转为字符信息。这一步其实是可有可无的，但是有些地方会依赖这个步骤。
4. 一旦上面的工作完成了我们把消息传给 DispatchMessage()。DispatchMessage()先看信号是给哪个窗口的，再找到那个窗口的窗口过程。再调用那个过程，参数为窗口的句柄，消息，wParam 和 lParam。
5. 在你的窗口过程中，你得到了那些参数，就可以为所欲为了。如果你没有处理某些特定的消息，那你就调用 DefWindowProc()来为你做一些默认的操作（一般就是什么都不做）。
6. 一旦你完成了对消息的处理，你的窗口过程就返回了，DispatchMessage()返回了，我们就再次循环。

这是 windows 程序中一个非常重要的概念。你的窗口过程并不是由系统来神奇地调用的，实际上你在 DispatchMessage()中自己间接地调用了它。如果你愿意，你可以对消息调用 GetWindowLong()得到它的窗口过程再直接调用它！

```
while(GetMessage(&Msg, NULL, 0, 0) > 0)
{
    WNDPROC fWndProc = (WNDPROC)GetWindowLong(Msg.hwnd, GWL_WNDPROC);
    fWndProc(Msg.hwnd, Msg.message, Msg.wParam, Msg.lParam);
}
```

我对前面的例子用了这种方法，可以工作，但是有很多的方面这种方法没有注意到，比如 Unicode/ANSI 转换，时钟回调函数等等，它很可能在我们的简单试验中可以工作。所以

试下就可以了，不要在实际的代码中用它：)

注意我们用 `GetWindowLong()` 来从窗口来查找它的窗口过程。为什么我们不直接调用 `WndProc()`？因为我们的消息循环为我们程序中的所有窗口的消息服务，包括按钮，列表框这样拥有自己的窗口过程的窗口，所以我们要我们调用了正确的窗口过程。因为多个窗口可能使用一个窗口过程，第一个参数（窗口的句柄）告诉窗口过程某个消息是为那个窗口的。

你可以看到，你的应用程序的主要时间就是在消息循环这里打转，你很高兴地向那些快乐的窗口发消息让他们处理。如果你要退出程序你怎么办？因为我们使用了一个 `while` 循环，如果 `GetMessage()` 返回了 `FLASE`（就是 0），循环就结束我们就可以到 `WinMain()` 的结束点。这就是 `PostQuitMessage()` 做的工作。它向队列发送一个 `WM_QUIT` 消息，`GetMessage()` 就向 `Msg` 结构体填充数据并返回 0，而不返回正的值。这个地方，`Msg` 的 `wParam` 成员含有你传向 `PostQuitMessage()` 的数据，你可以忽略它，也可以从 `WinMain()` 返回当这个进程的结束码。

要点：`GetMessage()` 遇到错误后返回 -1。你要记住这点，说不定哪次你在这里会犯错... 即使 `GetMessage()` 被定义为返回一个 `BOOL` 值，它还是会返回 `TRUE` 和 `FLASE` 之外的值，因为 `BOOL` 被定义为 `UINT` (`unsigned int`)。下面的代码可能看起来可工作，但是不能正确处理某些情况：

```
while(GetMessage(&Msg, NULL, 0, 0))
while(GetMessage(&Msg, NULL, 0, 0) != 0)
while(GetMessage(&Msg, NULL, 0, 0) == TRUE)
```

上面的写法都是错误的！可能你注意到我在这个教程中使用了第一个，刚刚提到了，如果 `GetMessage()` 不失败，并不会出错，你代码要是正确的话是不会出错。但是你要是在读本教程的话我并不能以此做为前提，你的代码可能有很多错误，`GetMessage()` 会在某些点出错：) 这种写法我已经更正了，如果我漏了某些地方请原谅。

```
while(GetMessage(&Msg, NULL, 0, 0) > 0)
```

应该始终使用这段拥有相同的效果的代码。

我希望你对 windows 消息循环有了进一步的了解，如果没有，不用怕，你使用它们一些时间后就会更清楚了。

使用资源

你可能需要参考本教程后面的附表查看关于使用 VC++ 和 BC++ 编辑资源的更多消息。

在我们进行进一步的对资源进行讨论前我先提一点以免我要在每一段重写一次。你不需要编译此段中的内容，它仅仅是例子。

资源是你可执行文件中预定义的二进制数据。你用资源脚本来创建资源，就是以 ".rc" 结尾的那种文件。商业的编译器一般有可视化的资源编辑器让你能不用手动编辑资源脚本但有些时候你必须这样做，特别是你用的编译器没有可视化的资源编辑器的时候，或是不支持你需要的功能的时候。

不幸的是，不同的编译器以不同的方式处理资源。我将尽力对工作所需要的资源处理中的通用部分进行解释。

MSVC++ 中的资源编辑器让手工编辑资源很困难，因为它又加了一些专有的格式，而且把你手动编辑的文件弄得一团糟。通常说你不用为手动编辑 .rc 文件而操心，但是知道如何手工修改它是很有好处的。另外一件烦心的事情是 MSVC++ 总是把资源头文件命名为 "resource.h"，即使你想起一个别的名字。为简便叙述起见我暂时先遵循这一个规则，将后面的关于编译器的附表中将指出怎样改变这个规则。

首先让我们来看一个很简单的资源脚本，只有一个图标。

```
#include "resource.h"
IDI_MYICON ICON "my_icon.ico"
```

这就是整个文件了。IDI_MYICON 是资源的标识，ICON 是类型，"my_icon.ico" 是包含其的外部文件。这些应该在所有的编译器上都可以工作。

那么 "#include "resource.h" 干什么用？因为你的程序需要一种标识这个图标的方式，最好的方式就是给个一个独一无二的 ID (IDI_MYICON)。我们可以创建 "resource.h" 并在我们的资源脚本和源文件中同时包含它。

```
#define IDI_MYICON 101
```

可以看到我们给 IDI_MYICON 赋了 101 的值。我们可以忘掉那个标识符只用 101 来引用这个图标，但是 IDI_MYICON 表达的意思更清楚，而且在你拥有一大堆资源的时候也容易记忆。

现在我们来添加一个 MENU 资源：

```
#include "resource.h"
IDI_MYICON ICON "my_icon.ico"
IDR_MYMENU MENU
BEGIN
    POPUP "&File"
```



```
BEGIN
    MENUITEM "E&xit", ID_FILE_EXIT
END
END
```

这次 IDR_MYMENU 是资源的名字，MENU 是类型。看见 BEGIN 和 END 没有？有些资源编辑器或是编译器用 { 代替 BEGIN 和 } 代替 END。如果你的编译器两者都支持的话你随便选一种就是了。如果它只支持一种的话，你就需要做一些替换。

我们还添加了一个新的标识符 ID_FILE_EXIT，所以我们为了在我们的程序中使用的話，先要在资源头文件 resource.h 中添加一下。

```
#define IDI_MYICON 101
#define ID_FILE_EXIT 4001
```

在大的工程中生成这些标识并要一直保持它们的定义是一件很繁杂的事务，这就是为什么很多人使用一个可视化的资源编辑器来做这些事情。它们有时也会出点问题，比如你可能遇到很多资源项拥有相同的标识之类的问题，这时候你就可能要自己来解决它们。

现在来看下怎么在你的程序中使用资源。

```
HICON hMyIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_MYICON));
```

LoadIcon() 和很多其它的使用资源的函数的第一个参数为当前运行程序的实例的句柄（在 WinMain() 中给出，也可用像前面章节中讲的那样用 GetModuleHandle() 来获取）。第二个参数为资源的标识。

你可能在想 MAKEINTRESOURCE() 干什么用或在迷惑为什么 LoadIcon() 要吃一个 LPCTSTR 型的参数而不是 UINT 之类，当我们传给它一个标识的时候。MAKEINTRESOURCE() 所做的就是把我们的资源标识的整型数强转为 LoadIcon() 所要的 LPCTSTR 类型。这里我们可以看到标识资源的第二种方法，使用字符串。几乎没有人这样做了，所以我不详叙细节了，但是基本上如你不用 #define 为你的源定义一个整数的话那么它的名字以一个字符串来解释，在你的程序中这样来引用：

```
HICON hMyIcon = LoadIcon(hInstance, "MYICON");
```

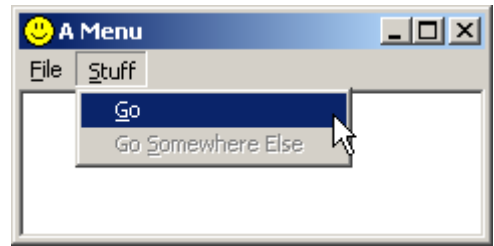
LoadIcon() 和一些别的使用资源的 API 函数能够通过检查高字来判别传给它的是一个整数还是一个指向字符串的指针。如果为 0（所有小于等于 65535 的整数）则它认为它为一个资源标识。这也把你你的标识限制在 65535 下面了，除非你的确有很多很多的资源，应该不成问题。如果不是 0 则它认为它为一个字符串的指针，它就用名字来找资源。当然除非文档中明确指明了这一点，最好不要依赖 API 的这个特性。

比如，对于菜单命令 ID_FILE_EXIT，这个特性就不能工作，因为它们只能为整型。

菜单与图标

范例：menu_one

这是一个展示如何向你的窗口添加基本的菜单的小段落。一般你使用一个准备好的菜单资源。它们在.rc 文件中，编译后链接到你的.exe 文件中。这部分对于不同的编译器很多差异，商业化的编译器拥有资源编辑器，你可用它来创建菜单，不过在这里我将向你展示.rc 的内容这样你就可以手工修改。我通常用一个.h 文件同时被我的.rc 文件和.c 文件包含。这个文件包含控件和菜单项目的标识等内容。



比如你可以从我们的那个 simple_window 的例子开始，再按照我们所说的把下面的代码加入进去。

首先是.h 文件。一般称为"resource.h"

```
#define IDR_MYMENU 101
#define IDI_MYICON 201
#define ID_FILE_EXIT 900
#define ID_STUFF_GO 9002
```

不是很多代码，但我们的菜单将非常简单。这里的名字和值由你选择。现在我们来写.rc 文件。

```
#include "resource.h"
IDR_MYMENU MENU
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "E&xit", ID_FILE_EXIT
    END
    POPUP "&Stuff"
    BEGIN
        MENUITEM "&Go", ID_STUFF_GO
        MENUITEM "G&o somewhere else", 0, GRAYED
    END
END
IDI_MYICON ICON "menu_one.ico"
```

你可能需要把这个.rc 文件添到你的项目或 makefile 中去（依你使用的工具而定）。

你也要在你的源文件（.c）中#include "resource.h"，这样你要用的菜单命令标识和菜单资源标识才会被定义。

将菜单和图标加到你的窗口的最简单的方法是当你注册你的窗口类的时候指明它们，这样：

```
wc.lpszMenuName = MAKEINTRESOURCE(IDR_MYMENU);
wc.hIcon = LoadIcon(GetModuleHandle(NULL),
    MAKEINTRESOURCE(IDI_MYICON));
wc.hIconSm = (HICON)LoadImage(GetModuleHandle(NULL),
    MAKEINTRESOURCE(IDI_MYICON), IMAGE_ICON, 16, 16, 0);
```

改变一下，看有什么效果. 你的窗口将有一个 File 和一个 Stuff 菜单项目和它们下面的项目. 假设你的.rc 文件被正确编译并链接至你的程序中去. （又要参考编译器的说明了.）

窗口的左上角和任务栏现在将显示我们定义的自定义小图标. 如果你按下 Alt+Tab，应用程序列表中将显示图标的大版本.

我用了 LoadIcon()来装入大图标因为这样简单，但它只会装入默认的分辨率为 32*32 的图标，所以为了装入小图标，我们要用 LoadImage().注意图标文件和资源可以含有个图像，这个例子中我提供了我要装入的两种大小的图像.

范例：menu_two

另外一种使用菜单资源的方式是在运行的时候创建一个（on the fly）. 这需要多一点的技巧，但增添了灵活度而且在某些时候是必需的.

你也可以不是以资源存储的图标，你可以把你的图标存为一个单独的文件并在运行的时候装入. 这样可以给你使用户自己来选择图标的选项，使用我们将在后面的章节中讲的对话框或别的什么类似的东西来选择.

再从没有添加.h 的.rc 文件的 simple_window 例子开始. 现在我们要处理 WM_CREATE 消息并向窗口添加一个菜单.

```
#define ID_FILE_EXIT 9001
#define ID_STUFF_GO 9002
```

这次把这两个标识放在你的.c 文件的首部，就在你的#include 语句下面.接下来我们向我们的 WM_CREATE 消息处理部分添加如下的代码.

```
case WM_CREATE:
{
    HMENU hMenu, hSubMenu;
    HICON hIcon, hIconSm;
    hMenu = CreateMenu();
    hSubMenu = CreatePopupMenu();
    AppendMenu(hSubMenu, MF_STRING, ID_FILE_EXIT, "E&xit");
    AppendMenu(hMenu, MF_STRING | MF_POPUP, (UINT)hSubMenu, "&File");
    hSubMenu = CreatePopupMenu();
```

```

AppendMenu(hSubMenu, MF_STRING, ID_STUFF_GO, "&Go");
AppendMenu(hMenu, MF_STRING | MF_POPUP, (UINT)hSubMenu, "&Stuff");
SetMenu(hwnd, hMenu);
hIcon = LoadImage(NULL, "menu_two.ico", IMAGE_ICON, 32, 32,
                  R_LOADFROMFILE);
if(hIcon)
    SendMessage(hwnd, WM_SETICON, ICON_BIG, (LPARAM)hIcon);
else
    MessageBox(hwnd, "Could not load large icon!", "Error", MB_OK |
                B_ICONERROR);
hIconSm = LoadImage(NULL, "menu_two.ico", IMAGE_ICON, 16, 16,
                    R_LOADFROMFILE);
if(hIconSm)
    SendMessage(hwnd, WM_SETICON, ICON_SMALL,
(LPARAM)hIconSm);
else
    MessageBox(hwnd, "Could not load small icon!", "Error", MB_OK |
                MB_ICONERROR);
}
break;

```

这样做几乎创建了和我们添加资源文件的一样的菜单。加到一个窗口的菜单会随著程序的终止而自动删除，所以我们不需要在后面来担心怎样删除。我们真的要那样做的话，可以用 `GetMenu()` 和 `DestroyMenu()`。

图标的代码相当简单，我们调用 `LoadImage()` 两次，装入了 16*16 和 32*32 两种大小的图标。我们这里不能用 `LoadIcon()` 因为它只能装入资源，而不能装入文件。我们把实例句柄参数写成了 `NULL` 因为我们不是从我们的模块装入资源，而且我们不用资源的标识而使用了我们要装入的图标文件名。最后我们传入了 `LR_LOADFROMFILE` 这个标志来指示我们需要这个函数将我们传入的字符串当作文件名而不是资源名字。

如果所有的调用都成功了我们就我在 `WM_SETICON` 的消息处理中把图标的句柄赋给我们的窗口，如果失败了则弹出一个对话框告诉我们出错了。

注意：如果我们要装入的图标文件不在程序现在的当前目录则 `LoadImage()` 调用失败。如果你用 VC++ 从 IDE 来运行程序的话，当前目录就是你工程文件所在的目录。如果你从 **Debug** 或 **Release** 目录下用文件管理器或命令行运行的话，你就需要把你的图标文件拷进去样以使你的程序找到它。如果怎样还是出错，就在你的调用中指明图标的全路径，`"c:\\path\\to\\icon.ico"`。

好的，现在我们有菜单了，我们让它做点事情。这很简单，我们就需要响应 `WM_COMMAND` 消息就行。我们也需要我们得到了哪个消息并做出相应的动作。现在我们的 `WndProc()` 应该看起来像这样。

```

LRESULT CALLBACK WndProc(HWND hwnd, UINT Message, WPARAM wParam, LPARAM lParam)

```

```

{
switch(Message)
{
    case WM_CREATE:
    {
        HMENU hMenu, hSubMenu;
        hMenu = CreateMenu();
        hSubMenu = CreatePopupMenu();
        AppendMenu(hSubMenu, MF_STRING, ID_FILE_EXIT, "E&xit");
        AppendMenu(hMenu, MF_STRING | MF_POPUP, (UINT)hSubMenu,
            "&File");
        hSubMenu = CreatePopupMenu();
        AppendMenu(hSubMenu, MF_STRING, ID_STUFF_GO, "&Go");
        AppendMenu(hMenu, MF_STRING | MF_POPUP, (UINT)hSubMenu,
            "&Stuff");
        SetMenu(hwnd, hMenu);
        hIcon = LoadImage(NULL, "menu_two.ico", IMAGE_ICON, 32, 32,
            LR_LOADFROMFILE);

        if(hIcon)
            SendMessage(hwnd, WM_SETICON, ICON_BIG, (LPARAM)hIcon);
        else
            MessageBox(hwnd, "Could not load large icon!", "Error", MB_OK |
                MB_ICONERROR);
        hIconSm = LoadImage(NULL, "menu_two.ico", IMAGE_ICON, 16, 16,
            LR_LOADFROMFILE);
        if(hIconSm)
            SendMessage(hwnd, WM_SETICON, ICON_SMALL,
                (LPARAM)hIconSm);
        else
            MessageBox(hwnd, "Could not load small icon!", "Error", MB_OK |
                MB_ICONERROR);
    }
    break;
    case WM_COMMAND:
        switch(LOWORD(wParam))
        {
            case ID_FILE_EXIT:
                break;
            case ID_STUFF_GO:
                break;
        }
        break;
    case WM_CLOSE:
        DestroyWindow(hwnd);
    break;
}

```

```

        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hwnd, Message, wParam, lParam);
    }
    return 0;
}

```

你可以看到我们把 WM_COMMAND 消息响应了，并且它还有自己的一个 switch() 在其中。这个 switch() 由 wParam 的低字来决定，因为可能 WM_COMMAND 消息包含了发送消息的菜单或控件的标识。

我们显然想要 Exit 菜单来关闭程序。所以在 WM_COMMAND, ID_FILE_EXIT 消息的处理程序中你可以使用下面的代码来做到这一点。

```
PostMessage(hwnd, WM_CLOSE, 0, 0);
```

你的 WM_COMMAND 处理代码现在看起来应该像样：

```

case WM_COMMAND:
    switch(LOWORD(wParam))
    {
        case ID_FILE_EXIT:
            PostMessage(hwnd, WM_CLOSE, 0, 0);
            break;
        case ID_STUFF_GO:
            break;
    }
    break;

```

至于 ID_STUFF_GO 要干什么就留给你来定义了。

程序文件的图标

你可能注意到了 menu_one.exe 看起来是我们给的那个当作资源的那个图标，而 menu_two.exe 则不是的，因为我们是装入的外部的文件。Windows 资源管理器只是显示程序资源中的第一个图标(由 ID 排序)，所以我们只有一个图标的时候，就显示那个了。如果我们要显示某个特定的图标，就把它加到资源中去并给定义个很小的标识。．．． 比如 1。你根本就需要使用这个图标，你完全可以给你的窗口装入完全不同的图标。

对话框：图形界面设计者的好朋友

范例：dlg_one

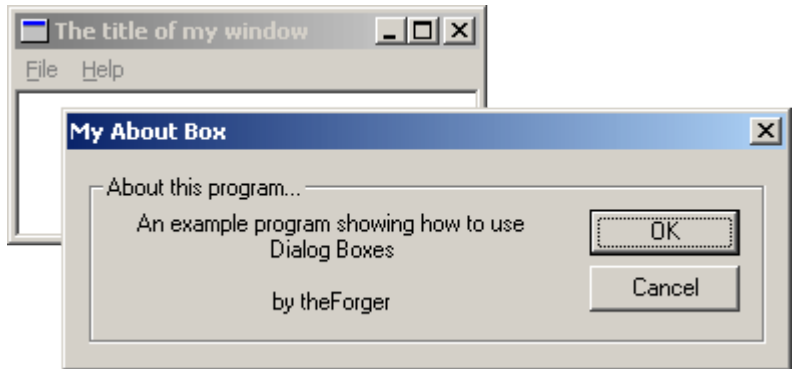
很难找到一个没有使用对话框的程序。在任何文本编辑器或其它的什么编辑器之类的东西，点 文件->打开 菜单，你就会看到一个对话框，很可能就要你选择一要打开的文件。

对话框不限于标准的打开文件框，可以是任意外观和任意功能。

对话框吸引人的地方在于它提供了一种快捷的方式来创建一个图形界面和一些默认的处理工作，大大地减少了你所需要写的代码。

要记住的一点就是对话框其实就是窗口。对话框与普通的窗口的区别在于系统为对话框作了一些额外的处理工作，比如创建并初始化控件，并处理 tab 顺序。几乎所有用于普通窗口的 API 函数都可以用于对话框。

第一步是创建对话框资源。对于任何的资源而言如何处理它们都与你用的编译器/IDE 相关。这里我向你展示.rc 文件中的对话框的文本并让你将其整合到你的项目中去。



```
IDD_ABOUT_DIALOG DISCARDABLE 0, 0, 239, 66
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "My About Box"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON "&OK",IDOK,174,18,50,14
    PUSHBUTTON "&Cancel",IDCANCEL,174,35,50,14
    GROUPBOX "About this program...",IDC_STATIC,7,7,225,52
    CTEXT "An example program showing how to use Dialog
        Boxes\r\n\r\nby theForger",
        IDC_STATIC,16,18,144,33
END
```

第一行中，IDD_AGOUTDLG 为资源的标识。DIALOG 是资源的类型，四个数字为左，顶，宽，高坐标。它们不是像素，而是基于系统的字体（由用户选择）的大小对话框单位。如果你选择了一个大的字体，对话框就大，如果选择了小点的字体，对话框就相应的小些。这一点很重要，因为要确保所有的控件用当前的字体以合适的大小显示。你可以用 MapDialogRect() 在运行的时候把对话框单位转换为像素。DISCARDABLE 告诉系统在不用此资源的时候将其交换至磁盘以节省系统资源（这是不用说的）。

第二行以 STYLE 开头然后在后面跟上用来创建此对话框的窗口风格。这些在你的帮助文档中的关于 CreateWindow() 说明中会有解释。为了使用已定义好的那些常量你需要在你的.rc 文件中加上 #include "windows.h"，或者如果你使用 VC++ 的话 winres.h, afxres.h 也可以。

如果你使用资源编辑器的话这些文件则在需要的时候自动被加上去。

CAPTION 那行应该一看就知道是什么意思。

FONT 那行指明了你想用来创建此对话框的字体的大小和名字。因为不同的人有不同的字体并可能指定不同的字体所以这行在不同的计算机上可能看起来不一样。当然你一般不用为此操心。

现在我们用控件的列表来创建此对话框

```
DEFPUSHBUTTON "&OK",IDOK,174,18,50,14
```

这行指定 OK 按钮。这里的&有点像菜单项的那个有下划线的那个字母”O”，所以用户可以按下 Alt+O 来激活这个控件（这就我上面所提到的对话框所做的默认处理的一部分）。IDOK 是控件的标识。IDOK 是已经定义好的，所以我们不需要自己用#define 来定义它了。最后的四个数字为左，顶，宽，高坐标，都是对话框单位。

这些信息太过于学术化了，所以你一般都用一个资源编辑器来创建对话框，但是知道如何用文本的方式来做这些事情还是很有必要的，尤其在你没有一个图形化的编辑器的情况下。

有两个控件的标识为 IDC_STATIC（就是-1），这表明我们永远并不需要去读写它们，所以它们并不需要一个标识。但是给它们一个标识也没有什么害处，而且资源编辑器会自动为你做一点。

静态控件中的文本中的”\r\n”是一个 CR-LF 对，表示换一个新行。

至此！把这些加到.rc 文件中去后我们要写一个对话框的过程来处理这个对话框的消息。不要担心，这不是什么很新的知识，实际上它跟我们的主窗口的窗口的过程差不多（不是完全一样）。

```
BOOL CALLBACK AboutDlgProc(HWND hwnd, UINT Message, WPARAM wParam, LPARAM lParam)
{
    switch(Message)
    {
        case WM_INITDIALOG:
            return TRUE;
        case WM_COMMAND:
            switch(LOWORD(wParam))
            {
                case IDOK:
                    EndDialog(hwnd, IDOK);
                    break;
                case IDCANCEL:
                    EndDialog(hwnd, IDCANCEL);
                    break;
            }
    }
}
```



```

    }
    break;
    default:
        return FALSE;
    }
    return TRUE;
}

```

对话框过程跟窗口过程有一些重要的差别。其中一个就是对于你不处理的消息不调用 `DefWindowProc()`。在对话框中这是自动完成的（你要是真的要自己调用的话会有问题）。

第二，通常上你不处理的消息就返回一个 `FALSE`，处理的话就是 `TRUE`，除非那个消息指明了你要返回一个别的值。注意这就是我们在上面做的，默认的就是什么都不做并返回一个 `FALSE`，而我们处理的消息就跳出 `switch()` 并返回 `TRUE`。

第三，不调用 `DestroyWindow()` 来关闭一个对话框，而调用 `EndDialog()`。第二个参数是返回给调用 `DialogBox()` 的那个地方的代码的。

最后，不处理 `WM_CREATE` 消息，取而代之，处理 `WM_INITDIALOG` 消息来做对话框出现之前的任何操作，并返回 `TRUE` 以使键盘将焦点置于点的控件之上。（你确实可以处理 `WM_CREATE` 消息，但那是在所有控件被创建之前就发送了，这样你就不能访问它们了，而在 `WM_INITDIALOG` 消息来的时候，控件已经创建好了）。

叽叽歪歪的半天了，现在创建它...

```

case WM_COMMAND:
    switch(LOWORD(wParam))
    {
        case ID_HELP_ABOUT:
        {
            int ret = DialogBox(GetModuleHandle(NULL),
                                MAKEINTRESOURCE(IDD_ABOUT), hwnd, AboutDlgProc);
            if(ret == IDOK){
                MessageBox(hwnd, "Dialog exited with IDOK.", "Notice",
                            MB_OK | MB_ICONINFORMATION);
            }
            else if(ret == IDCANCEL){
                MessageBox(hwnd, "Dialog exited with IDCANCEL.", "Notice",
                            MB_OK | MB_ICONINFORMATION);
            }
            else if(ret == -1){
                MessageBox(hwnd, "Dialog failed!", "Error",
                            MB_OK | MB_ICONINFORMATION);
            }
        }
    }
}

```

```
        break;
        // Other menu commands...
    }
break;
```

这就是我用来创建我的关于对话框的代码，你应该看出来这应该移到你的 WM_COMMAND 消息处理的代码中去，如果你对这点还不清楚，你可能需要复习关于菜单的那些章节。ID_HELP_ABOUT 是我的 Help->About 菜单项的标识。

因为我们要我们的主窗口的菜单来创建这个对话框，我们显然需要把这些代码放到我们主窗口的 WndProc() 中去，而不是对话框的过程中。

现在我存储了调用 DialogBox() 的返回值，这样你就可以观察你点击两个按钮的效果，在对话框中按 Esc, Enter 等等... 这样做也展示了如何通过对话框的返回值来判断是否调用成功，和用户用选择，或是你想从你的对话框过程中想返回给调用者的任何消息。

```
DialogBox(GetModuleHandle(NULL), MAKEINTRESOURCE(IDD_ABOUT), hwnd,
AboutDlgProc);
```

这是唯一重要的地方，你可以把这段代码放到你想要对话框出现的任何地方。IDD_ABOUT 是对话框资源的标识。hwnd 是对话框的父窗口的句柄。AboutDlgProc() 当然是用来控制对话框的对话过程。

搞定了！ Sit IDD_UBU, sit.

某些特别敏感的读者可能会问，如果 DialogBox() 直到对话关闭才返回的话，我们在它显示的时候不能处理消息，它怎么工作的？这是因为 DialogBox() 有个特点就是有自己的消息循环，所以当对话框显示的时候，我们的消息循环正在进行并且窗口处理了默认的消息循环。这个消息循环也处理类似当你按下 Tab 键时在控件之间移动键盘的焦点的事情。

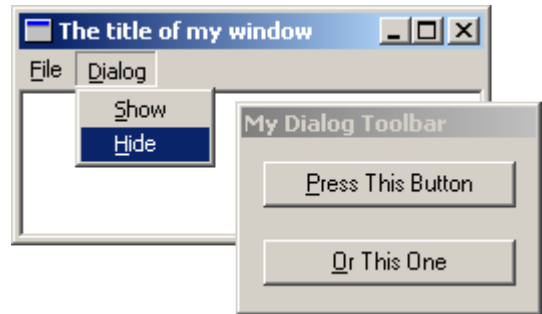
另外一个使用对话框的效果就是在对话框关闭之前你的主窗口将被禁止。有时候你就是要这种效果，有时你不是想这样，比如有时我们想要我们的对话框作为一个浮动的工具栏，这种情况下我们想要能够同时操作我们的对话框与主窗口，这也是下面的章节要讲的内容。

无模态对话框

范例: dlg_two

现在来看看 `CreateDialog()`，它是 `DialogBox()` 的姐妹函数。区别在于 `DialogBox()` 拥有自己的消息循环并且直到对话框关闭才返回，`CreateDialog()` 则更加像 `CreateWindowEx()` 创建的一个窗口，立即返回并向你的消息循环发送消息，就像是你的主窗口发的消息样。这就是所谓的无模态，而 `DialogBox()` 创建的是模态对话框。

你可以像上个例子样来创建对话框，你也需要设工具窗口的扩展风格来给你的标题栏一个典型的小些的工具栏。我创建的资源如下：



```
IDD_TOOLBAR DIALOGEX 0, 0, 98, 52
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION
EXSTYLE WS_EX_TOOLWINDOW
CAPTION "My Dialog Toolbar"
FONT 8, "MS Sans Serif"
BEGIN
    PUSHBUTTON    "&Press This Button", IDC_PRESS, 7, 7, 84, 14
    PUSHBUTTON    "&Or This One", IDC_OTHER, 7, 31, 84, 14
END
```

你可以看到资源编辑器把 `DIALOG` 换成了 `DIALOGEX` 表明我们要为我们的对话框设置扩展风格。

接下来我们想在我们的程序运行的时候创建对话框，我们想要我们的对话框可视，所以我们在 `WM_CREATE` 的消息处理中创建它。我们也需要声明一个全局变量来保持从 `CreateDialog()` 返回的窗口句柄以便在后面使用它。`DialogBox()` 不向我们返回句柄因为 `DialogBox()` 在窗口销毁的时候才返回。

```
HWND g_hToolbar = NULL;
case WM_CREATE:
    g_hToolbar = CreateDialog(GetModuleHandle(NULL),
                             MAKEINTRESOURCE(IDD_TOOLBAR),
                             hwnd, ToolDlgProc);

    if(g_hToolbar != NULL)
    {
        ShowWindow(g_hToolbar, SW_SHOW);
    }
    else
    {
        MessageBox(hwnd, "CreateDialog returned NULL", "Warning!",
                   MB_OK | MB_ICONINFORMATION);
    }
```

```
    }  
    break;
```

我们要检查返回值，这什么时候总是一个好的习惯，如果是正确的（不为 NULL）我们就用 `ShowWindow()` 来显示这个窗口，要是用 `DialogBox()`，这一步就是不必要的，因为系统为我们调用了 `ShowWindow()`。

现在我需要为我们的工具栏写一个对话过程。

```
BOOL CALLBACK ToolDlgProc(HWND hwnd, UINT Message, WPARAM wParam, LPARAM  
lParam)  
{  
    switch(Message)  
    {  
        case WM_COMMAND:  
            switch(LOWORD(wParam))  
            {  
                case IDC_PRESS:  
                    MessageBox(hwnd, "Hi!", "This is a message",  
                        MB_OK | MB_ICONEXCLAMATION);  
                    break;  
                case IDC_OTHER:  
                    MessageBox(hwnd, "Bye!", "This is also a message",  
                        MB_OK | MB_ICONEXCLAMATION);  
                    break;  
            }  
        break;  
        default:  
            return FALSE;  
    }  
    return TRUE;  
}
```

大多数适用于 `DialogBox()` 的消息处理规则也适用于 `CreateDialog()`，不调用 `DefWindowProc()`，对不处理的消息返回 `FALSE`，处理的返回 `TRUE`。

有个改变就是我们不为无模态窗口调用 `EndDialog()`，我们可以像对常规的窗口一样调用 `DestroyWindow()`。我们的例子中主窗口销毁的时候对话框才销毁。在主窗口的 `WndProc()` 中这样写...

```
case WM_DESTROY:  
    DestroyWindow(g_hToolbar);  
    PostQuitMessage(0);  
break;
```

还有一点，我们希望在任何我们希望的时候显示或隐藏我们的工具栏，所以我在菜单上

加上了两个命令来做这件事情，并这样处理：

```
case WM_COMMAND:
    switch(LOWORD(wParam))
    {
        case ID_DIALOG_SHOW:
            ShowWindow(g_hToolbar, SW_SHOW);
            break;
        case ID_DIALOG_HIDE:
            ShowWindow(g_hToolbar, SW_HIDE);
            break;
        //... other command handlers
    }
    break;
```

你现在应该可以用资源编辑器或手工的方法来创建你自己的菜单，如果还有问题（一般是这样）可以参看一个本教程提供的例子 `dlg_two`。

现在你运行你的程序的时候，你就应该可以同时访问你的对话框和主窗口。

如果你在此时运行你的程序并试图在两个按钮间切换的话，就会注意到不行，按 `Alt+P` 和 `Alt+O` 来激活按钮都不行。为什么？因为 `DialogBox()` 有自己的消息循环并默认地处理这些事件，`CreateDialog()` 却没有。但是我们可以通过在我们的消息循环中调用可以为我们做默认处理的 `IsDialogMessage()` 来自己处理它们。

```
while(GetMessage(&Msg, NULL, 0, 0))
{
    if(!IsDialogMessage(g_hToolbar, &Msg))
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
}
```

这里我们首先将消息传给 `IsDialogMessage()`，如果消息是为我们的工具栏的（由我们传进的窗口的句柄来指示），系统就作默认的处理并返回 `TRUE`。这种情况下消息已经被处理了所以我们不需要再调用 `TranslateMessage()` 或 `DispatchMessage()` 了。如果消息是为另外一个窗口的我们就照常处理。

值得提出的是 `IsDialogMessage()` 也可以用于不是对话框的窗口来给它们一些像对话框的功能。记住，对话框就是一个窗口，并且大多数（如果不是全部的话）对话框的 `API` 可以工作于任何窗口。

关于无模态的对话框讲得够多了！一个问题就是如果你有多个工具栏... 怎么办？一种解决方案就是定义一个列表（或是一个数组，一个标准模板库的 `std::list`，或类似的东西）

并在你的消息循环里循环把每个句柄传给 `IsDialogMessage()` 直至找到那个正确的，如果没有找到，就作常规的处理。这是一个泛型编程问题，不是一个 Win32 问题，就留为读者的练习吧。

Copyright © 1998-2003, Brook Miles ([theForger](#)). All rights reserved.

标准控件：按钮，编辑框，列表框

范例： `ctl_one`

我注意到我已经在前面的例子中用过按钮了，所以你们对其或多或少地应该有些了解了。但是我想我要在此例子中使用它们，我还是在标题中写了它，以求完整。

控件

对于控件要记住的一件事就是它们就是窗口。如同任何其它的窗口，他们有窗口过程，窗口类，等等... 由系统注册。任何你可以对一个窗口做的事情你都可以对一个控件做。

消息

回忆一下我们先前讲的消息循环，窗口用消息通信，你发送它们来让一个控件做点什么，当控件发生一个事件的时候它也向你发送一个通知消息。对于标准控件这个通知是一个 `WM_COMMAND` 消息，就如我们在按钮和菜单中已经见到过的一样。对于后面要讲的常见控件，它将会是 `WM_NOTIFY`。

对于不同的控件你发送相当不同的消息，每个控件都有自己的消息集合。有时一个消息可用于多个控件，但一般它们只会在一个它们专门面向的控件上工作。对于列表框(listbox)和组合框(combobox)的消息(`LB_*`和`CB_*`)这一点尤其令人伤脑筋，它们几乎完成相同的功能，但是却不能互换，我自己也经常搞混，次数多得我不不好意思承认了：)

另外一方面，像 `WM_SETTEXT` 的通用消息则为大多数控件所支持。毕竟一个控件就是一个窗口而已。

你可以用 `SendMessage()` 这个 API 来发送消息，用 `getDlgItem()` 来获取控件的句柄，或者你可以用 `SendDlgItemMessage()` 来一步完成上面的两步操作，两种方式的结果是等效的。

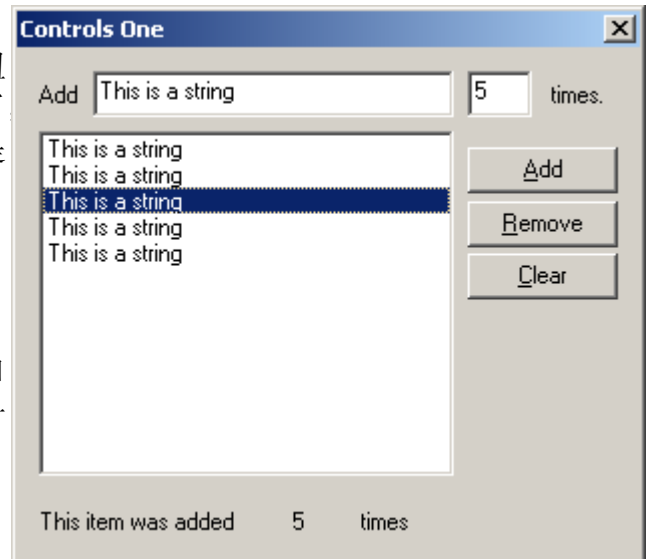
编辑框

编辑框是 Windows 系统中最常用的一个控件，用来让用户来对文本进行输入，修改，复制，等等操作。Windows 中的记事本就差不多是一个平凡的窗口再在里面嵌上一个大大的编辑框。

这就是此例中用来和编辑框接口的代码：

```
SetDlgItemText(hwnd, IDC_TEXT, "This is a string");
```

这就是用来改变控件中的文字所需的全部代码（可用于几乎所有有相关的文本值的控件，静态控件，按钮等等）。



从控件获取文本也是一样的简单，虽然多了一点点的设置要做...

```
int len = GetWindowTextLength(GetDlgItem(hwnd, IDC_TEXT));
if(len > 0)
{
    int i;
    char* buf;
    buf = (char*)GlobalAlloc(GPTR, len + 1);
    GetDlgItemText(hwnd, IDC_TEXT, buf, len + 1);
    //... do stuff with text ...
    GlobalFree((HANDLE)buf);
}
```

首先我们要为要存进来的字符串配置一点内存，它不会就返回一个指向内存中已有的字符串的指针。为了做这一点，首先我们要弄清楚为要配置多少内存。没有 `GetDlgItemTextLength()` 这个 API，但是有一个 `GetWindowTextLength()`，所以我们要自己使用 `GetDlgItem()` 获取控件的句柄。

现在我们获取了长度，就可以配置内存了。这里我添加了一个小小的检查来看是否有文本要处理，因为你很可能并不想处理一个空的字符串。．．．有时候你要这样做，那也是你自己的事。假设你有些东西要处理，我们调用 `GlobalAlloc()` 来配置一些内存。我在这里用的 `GlobalAlloc()` 等价于 `calloc()`，如果你对 DOS/UNIX 下的编程的话很熟悉的话。它分配一点内存，将其初始化为 0 并返回这段内存的指针。对于第一个参数，你可以传不同的值让它为不同的目的来做不同的操作，但在此教程中我只会用这种方式。

请注意我在两个位置对长度加了 1，这是干什么？因为 `GetWindowTextLength()` 返回控件的字符个数中不包括末尾那个终止符。这就意味著如果我们不加 1 就配置内存来存储这个字符串的话，字符串可以放下，但是终止符就会溢出内存块，可能会破坏别的数据，导致访问冲突，或是别的什么坏的结果。在处理 windows 系统中的字符串长度时候你要格外小心，一些 API 和函数要求字符串长度包括终止符但另外一些要求不要，随时准备查文档。

如果我在这里没有讲到终止符的话，请参考一本 C 语言的课本或是基础教材，它们会讲这个。

最后我们用 `GetDlgItemText()` 来将控件的内容获取至我们刚刚配置好的缓冲区。这个调用则要求缓冲区的长度包括终止符。它的返回值是复制的字符个数，我们这里将其忽略了，但这个数字又不包括终止符。．．．搞笑吧？：)

我们全部处理完了文本后（马上就要到了），我们要我们刚刚配置的内存释放掉以免它们泄漏了，又落到 CPU 上就会把计算机搞短路。我们调用 `GlobalFree()`，传进我们的指针就可以完成这个任务了。

你可能会见到过或想像有另外一套 API，叫作 `LocalAlloc()`，`LocalFree()`，等等。．．．它们是 16 位 windows 系统的历史 API。在 Win32 平台中，`Local*` 和 `Global*` 两种内存相关函数完全是一样的。

编辑框处理数字

可以输入文本了，但是如何让用户来输入数字？这是一个很常见的任务，幸好有一个 API 来简化这项工作，它为你做所有的内存配置，并把字符串转为整型数。

```
BOOL bSuccess;  
int nTimes = GetDlgItemInt(hwnd, IDC_NUMBER, &bSuccess, FALSE);
```

GetDlgItemInt() 和 GetDlgItemText() 很相似，除了后者是把一个字符串拷贝到一个缓冲区中去，而前者是把一个字符串在内部转为整型数并把值返回给你。第三个参数是可选的，是一个指向 B O O L 型的指针。因为这个函数在失败的时候返回 0，所以没有办法来区分究竟是调用失败了还是用户输入了一个 0。如果你不介意用 0 表示一个调用失败，那尽管忽略这个参数吧。

另外一个有用的参数是编辑框控件的 ES_NUMBER 风格，这个风格可以限制只允许用户输入 0 到 9 的数字。当你只需要正数输入的时候这个功能尤其方便，其它的情况下不怎么样，因为它不让你输入任何其它的字符，包括 -（负号）、.（小数点）、/（分隔符）。

列表框

另外一个经常用的控件是列表框。这是我在这里准备讲的最后一个标准控件，因为说实话，它们不是很有趣，如果你还没有感到枯燥的话，我已经有点了：)

加一个列表项

你想对一个列表框做的第一件事就是给它加一个项。

```
int index = SendDlgItemMessage(hwnd, IDC_LIST, LB_ADDSTRING, 0, (LPARAM)"Hi there!");
```

你可以看到，这是一件很简单的任务。如果列表框指定了 LBS_SORT 风格，新项将以字母顺序加进去，否则它将被加到列表的尾部。

这条消息总是返回新添加的列表项的序号，我们可以用这个序号来对列表项进行别的一些操作，比如向其绑定一些数据。一般会是一个指向一个句含更多消息的结构体的指针，或一个你用来标识这个列表项的标识，由你定了。

```
SendDlgItemMessage(hwnd, IDC_LIST, LB_SETITEMDATA, (WPARAM)index,  
(LPARAM)nTimes);
```

通知

列表框的整个意义在于让用户从一个列表选择东西。现在我们不是很关心他们具体什么时候来做选择，比如对于我们的删除按钮，我们不需要知道什么时候选择改变了，我们只需要在用户激活这个按钮的时候检查就是了。

然而，有些时候你想马上响应，比如根据所选不同的列表项来作不同的显示或更新某些消息。为此我们要响应列表框给我们发的通知消息。本例中我们要注意 LBN_SELCHANGE，

这个通知消息告诉我们用户更改了所选的列表项目.LBN_SELCHANGE 由 WM_COMMAND 发出，不过按钮或菜单的 WM_COMMAND 消息处理(一般就是响应点击)所不同的是，列表框发出 WM_COMMAND 消息的原因很多，所以我们要做进一步的检查找出更多的消息.通知代码在 wParam 的 HIWORD 部分传递，wParam 的另外一半则告诉我们控件的 ID.

```
case WM_COMMAND:
    switch(LOWORD(wParam))
    {
        case IDC_LIST:
            // It's our listbox, check the notification code
            switch(HIWORD(wParam))
            {
                case LBN_SELCHANGE:
                    // Selection changed, do stuff here.
                    break;
            }
            break;
        // ... other controls
    }
    break;
```

从列表框中获取数据

一旦我们知道选择已被改变了，或者应用户要求，我们就需要从列表框中获取选择的结果并做点有用的处理。

本例中我们使用了一个多选列表框，所以获取所选的项的清单就要点小技巧。如果用单选框，那你就简单地发送一个 LB_GETCURSE 消息就可以获得所选项的序号。

首先我们要获得所选项的数目，据此我们才能为我们要保存的序号来配置内存。

```
HWND hList = GetDlgItem(hwnd, IDC_LIST);
int count = SendMessage(hList, LB_GETSELCOUNT, 0, 0);
```

然后根据所选的列表项数目来配置内存，并发送 LB_GETSELITEMS 去填充数组。

```
int *buf = GlobalAlloc(GPTR, sizeof(int) * count);
SendMessage(hList, LB_GETSELITEMS, (WPARAM)count, (LPARAM)buf);
// ... Do stuff with indexes
GlobalFree(buf);
```

本例中，buf[0]是第一个序号，一直到 buf[count-1]存放后面的。

你可能还想从这些所选中的列表项中，获取跟它们绑定的数据，并作一些处理.这跟当初设置它们一样简单，发送另外一个消息就是了。

```
int data = SendMessage(hList, LB_GETITEMDATA, (WPARAM)index, 0);
```

如果数据为某种类型的数值(任何 32 位的)你就可以简单地强制将其转为相应的类型。比如你可以用 HBITMAP 来代替 int 存放返回值。 . . .

```
HBITMAP hData = (HBITMAP)SendMessage(hList, LB_GETITEMDATA, (WPARAM)index, 0);
```

静态控件

跟按钮一样，静态控件也很简单，但为了完整起见，我还是把它们写在了这里。静态控件一般就是很安静，意思是它们一般不改变或做什么特别的事，主要就是用来向用户显示文本。但是你可以向其指定一个独特的标识使它稍微有些用（VC++给它们指定默认标识 IDC_STATIC，其实就是-1，表示“没有标识”），这样就可以在运行的时候设置文字来向用户动态地显示数据。

在范例代码中，我用一个静态控件来显示选择的列表项的数据，假定你一次只选一个列表项：

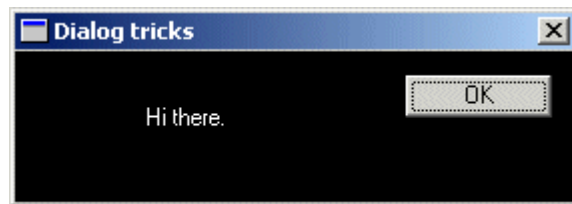
```
SetDlgItemInt(hwnd, IDC_SHOWCOUNT, data, FALSE);
```

Copyright © 1998-2003, Brook Miles ([theForger](#)). All rights reserved.

对话框常见问题(FAQ)

范例: dlg_three

要知道我这只是个教材，不是一个参考，但有些问题被问得太多了，所以我想我还是把它们也放在这里。



改颜色

一般来说，你这样做的唯一理由是你想在对话框上仿真一个连接或要达到类似的什么目的，否则你给对话框加上一大堆的颜色的后果只是让你的程序变得难看并且得刺眼睛，但是还是有得多人要这样做，不过的确有些合理的理由，所以就这样干吧：)

Windows 系统向你的对话框过程发送很多与颜色相关的消息，你通过处理它们你就可以某些地方的显示颜色。比如，要改变对话框自己的颜色，你就要处理 WM_CTLCOLORDLG 消息，要改变一个静态控件的颜色你就要处理 WM_CTLCOLORSTATIC 消息，诸如此类。

首先你创建一个画刷来画背景并将其存起来备用。WM_CTLCOLORDLG 消息和相关的消息在你程序的这个过程中来得很频繁，所以你要是每次都创建一个新的画刷的话，你可能过一会就会用完很多的 RAM 并留下得多个死刷子。这样做的话，我们就可以在对话框被销毁的时候就知道我们不再需要它，并删掉它，

```
HBRUSH g_hbrBackground = CreateSolidBrush(RGB(0,0, 0));
case WM_CTLCOLORDLG:
    return (LONG)g_hbrBackground;
case WM_CTLCOLORSTATIC:
{
    HDC hdcStatic = (HDC)wParam;
    SetTextColor(hdcStatic, RGB(255, 255, 255));
    SetBkMode(hdcStatic, TRANSPARENT);
    return (LONG)g_hbrBackground;
}
break;
```

注意我们把背景模式设为透明的那行... 如果不要这行，背景就会被你指定的刷子填充，但当你的控件写文字的时候就会被默认的背景所覆盖！把文字绘制模式设为透明就可以解决这个问题。还有一种解决方法是用 SetBkColor() 设为我们的背景刷子一样的颜色，但是我更喜欢这种解决方案。

在其它的种种的控件上改变颜色的操作步骤都是这样，在你的 Win32 参考中查一下 WM_CTLCOLOR* 消息就是了。注意编辑框控件为只读的话发的是 WM_CTLCOLORSTATIC，不是只读的话就发送 WM_CTLCOLOREDIT。

如果你有多个静态控件（或别的控件）想要设置颜色的话，那你就需要在接到消息的时候

检查控件的标识并依此操作。你在 `IParam` 中得到传递的控件的 `HWND`，并用 `GetDlgCtrlID()` 得到控件的标识。注意所有的静态控件都被资源编辑器给了一个默认的标识 `IDC_STATIC(-1)`，所以如果要把它们区分的话你就要给它们定义新的标识。

给对话框一个图标

相当简单的一个操作，只要给对话框发一个 `WM_SETICON` 消息就行了。因为 Windows 系统用两套图标，所以你要调用两次，一次为窗口角落的那个小图标，一个为当你按下 `Alt-tab` 组合时候的那个大图标。除非你有两个不同大小的图标，你可只用将相同的句柄发两次就是了。

要用默认的图标的话，你可以下面的代码：

```
SendMessage(hwnd, WM_SETICON, ICON_SMALL, (LPARAM)LoadIcon(NULL,
MAKEINTRESOURCE(IDI_APPLICATION)));
SendMessage(hwnd, WM_SETICON, ICON_BIG, (LPARAM)LoadIcon(NULL,
MAKEINTRESOURCE(IDI_APPLICATION)));
```

你要替换默认的资源为你自己的资源的时候，要记得要把 `LoadIcon()` 的 `HINSTANCE` 也改成你自己的应用实例（要是你没法从 `WinMain()` 得到的话，可以调用 `GetModuleHandle(NULL)` 来获得它）。

为什么我的组合框不工作？

一个很常见的问题是当他向对话框加了一个组合框后发现他运行程序并点击那个小箭头的时候看不到列表显示出来。这是可以理解的，因为解决方案不是很直观。

当你创建一个组合框并指定它的高度时，你实际上指定了整个高度，包括下拉列表在内，不是此控件展开的时候由系统根据所用的字体决定的高度。

比如，一个高度为 100 像素的控件，系统将控件本身设为默认的（比如为 30），当你点击小箭头的时候，下拉列表将会是 70 像素高，加在一起一共 100 像素。

如果你用 VC++ 的资源编辑器来把组合框放到你的对话框上，你就会注意到你不能把它的纵向地拉伸。除非你在编辑器中点击那个箭头，它才可以拉伸，并把焦点框改到那上面指示你正在改变下拉框的大小，这样你就可以任意地设立高度了。

其它的控件怎么办！

当然我可以把所有的控件都写个例子，但那是 MSDN 和 Petzold（译者注：Windows Programming 一书的作者）做的事情啊：)如果你实在不知道怎么用它们，你可能要把本教程的某些部分重读一下，或者去找一本更详细的书来看。

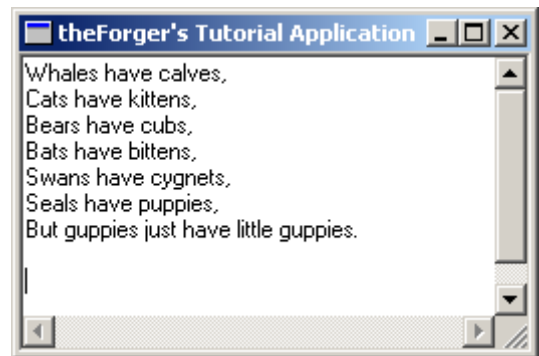
我本来想在这里给出一些 MSDN 上的一些有用链接的，但是微软好像跟我作对似的把指向 MSDN 上的页面一会换个位置或者弄得时不时不能访问。所以你就可能要去自己找它们了，看一下用户接口服务，Windows 控件，还有要翻下 Platform SDK 之类的东西。

应用 第一部分:运行时创建控件

范例: app_one

我想对于一个在运行时候创建控件的范例来讲,虽然有用,但除非真的做点什么有用的事,否则还是没什么用,所以这里我以一个文本编辑器开始并把它其做成能支持打开,编辑并保存文本文件的有用程序.

第一步,这页所说的就是简单地创建窗口和作为整个程序的核心的一个编辑框控件.



从那个简单窗口的程序框架开始,我们加上用#define 定义我们的控件标识和在我们的窗口过程中加上下面的两个消息处理的代码:

```
#define IDC_MAIN_EDIT 101
case WM_CREATE:
{
    HFONT hfDefault;
    HWND hEdit;
    hEdit = CreateWindowEx(WS_EX_CLIENTEDGE, "EDIT", "",
        WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL | ES_MULTILINE |
        ES_AUTOVSCROLL | ES_AUTOHSCROLL,
        0, 0, 100, 100, hwnd, (HMENU)IDC_MAIN_EDIT, GetModuleHandle(NULL),
        NULL);
    if(hEdit == NULL)
        MessageBox(hwnd, "Could not create edit box.", "Error", MB_OK |
            MB_ICONERROR);
    hfDefault = GetStockObject(DEFAULT_GUI_FONT);
    SendMessage(hEdit, WM_SETFONT, (WPARAM)hfDefault, MAKELPARAM(FALSE,
        0));
}
break;
case WM_SIZE:
{
    HWND hEdit;
    RECT rcClient;
    GetClientRect(hwnd, &rcClient);
    hEdit = GetDlgItem(hwnd, IDC_MAIN_EDIT);
    SetWindowPos(hEdit, NULL, 0, 0, rcClient.right, rcClient.bottom, SWP_NOZORDER);
}
break;
```

创建控件

正如创建所有其它的窗口一样，创建控件使用 `CreateWindowEx()` 这个 API。我们将我们想要的已注册的类传进去，这里是编辑框控件类，这样我们就得到一个标准的编辑框控件窗口。当用对话框来创建我们的控件时，我们一般要写下要创建的控件的清单，以便在你调用 `DialogBox()` 或 `CreateDialog()` 的时候系统可以在对话框的资源中读入控件的列表并调用 `CreateWindowEx()` 来按照在资源中定义的位置和式样来创建每个。

```
hEdit = CreateWindowEx(WS_EX_CLIENTEDGE, "EDIT", "",
    WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL | ES_MULTILINE |
    ES_AUTOVSCROLL | ES_AUTOHSCROLL,
    0, 0, 100, 100, hwnd, (HMENU)IDC_MAIN_EDIT, GetModuleHandle(NULL),
    NULL);
if(hEdit == NULL)
    MessageBox(hwnd, "Could not create edit box.", "Error", MB_OK |
        MB_ICONERROR);
```

你可以看到这个 `CreateWindowEx()` 的调用指定了很多的式样，而且在实际的编程有更加也很常见，尤其是有大堆选项的常用控件。开始的四个 `WS_*` 式样应该相当清楚，我们以我们窗口的一个子窗口来创建这个控件，我们使其可见，并且有纵向和水平的滚动条。

三个为编辑框特有的式样(`ES_MULTILINE | ES_AUTOVSCROLL | ES_AUTOHSCROLL`)指定了这个编辑框应该含有多行文本，并于你在控件的最底部和最右边打字的时候分别自动滚动。

常规的窗口式样(`WS_*`)列在 MSDN 中，自己去查下吧。而扩展的窗口式样(`WS_EX_*`) 在 MSDN 中对 `CreateWindowEx()` 的解释中有说明，那个地方你也可以找到对每个控件特有式样说明的链接。(我们例子中的 `ES_*` 就是为编辑框所特有)。

我们以控件指定了作为其父窗口的我们的窗口句柄，并为其实指定了一个为 `IDC_MAIN_EDIT` 的标识，以便在后面要在对话框上创建控件时引用该控件。位置和大小参数在这里不是很重要因为我马上要响应 `WM_SIZE` 消息来动态调整此控件以使它总位于我们的窗口中间并适合其大小。

动态创建的控件的大小调整

一般来说如果你的窗口可以调大小的话，你总是希望写点代码来动态调整你在之中创建的控件的大小以使它们总是分布合理。

```
GetClientRect(hwnd, &rcClient);
hEdit = GetDlgItem(hwnd, IDC_MAIN_EDIT);
SetWindowPos(hEdit, NULL, 0, 0, rcClient.right, rcClient.bottom, SWP_NOZORDER);
```

因为我们现在只有一个控件，工作相对比较简单。我们用 `GetClientRect()` 来获取窗口的客户区的大小，也就是说不包括边际，菜单和标题栏的那块大（当前为止）地方。这会用值填充我们的 `RECT` 结构体，`left` 和 `top` 总是为 0，所以你一般可以忽略他们。`right` 和 `bottom` 值会给你客户区的宽和高。

下一步我们用 `GetDlgItem()` 来获取我们控件的句柄，这个 API 对于对话框和常规的窗口都可以使用，并调用 `SetWindowPos()` 去移动和拉伸以填充整个客户区。你当然可以通过更改给 `SetWindowPos()` 的值来做些别的事情，比如只填充窗口的一半，让另外一半留下来放别的控件。

在运行时创建别的控件

我就不给出在运行时创建别的控件（列表框，按钮等等）的例子了，因为基本上是一样的，要是重复多了也烦：）如果你去查查 MSDN，或是在你的 Win32 参考里翻翻你就会找到创建任何其它的标准控件的所需的消息。

我们将在后面的章节中还会多次有对常见控件的操作以使你获得更多的实践。

Copyright © 1998-2003, Brook Miles ([theForger](#)). All rights reserved.

应用 第二部分:使用文件和通用对话框

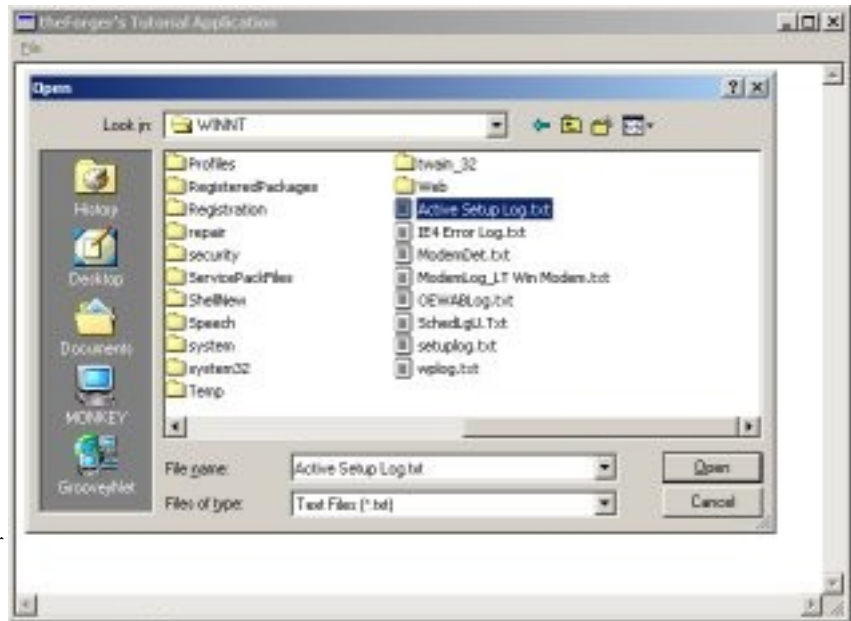
范例: app_two

通用文件对话框

要打开或保存文件的第一步就是要找到要用的文件名. . . 当然你总是可以把文件名直接写到你的程序中去, 但老实说那种做法一般不会使程序很有用.

因为这是一个如此常见的任务, 以致于有已经定义好的系统对话框可以用来允许用户来选择一个文件名. 最常用的打开和保存文件的对话框分别通过 `GetOpenFileName()` 和

`GetSaveFileName()` 来调用, 它们两个都要一个 `OPENFILENAME` 结构体作参数.



```
OPENFILENAME ofn;  
char szFileName[MAX_PATH] = "";  
ZeroMemory(&ofn, sizeof(ofn));  
ofn.lStructSize = sizeof(ofn); // SEE NOTE BELOW  
ofn.hwndOwner = hwnd;  
ofn.lpstrFilter = "Text Files (*.txt)\\0*.txt\\0All Files (*.*)\\0*.\\0";  
ofn.lpstrFile = szFileName;  
ofn.nMaxFile = MAX_PATH;  
ofn.Flags = OFN_EXPLORER | OFN_FILEMUSTEXIST | OFN_HIDEREADONLY;  
ofn.lpstrDefExt = "txt";  
if(GetOpenFileName(&ofn))  
{  
    // Do something usefull with the filename stored in szFileName  
}
```

请注意我们用了 `ZeroMemory()` 对结构体清零. 这通常是个好的习惯, 因为有些 API 对你传给的不用的参数要设为 `NULL` 很挑剔. 用这种方法就不需要对每个不用的成员进行设置了.

你要是在你的文档中查每个成员的意思并不是很容易的一件事. `LpstrFilter` 值指向一个双 `NULL` 终止符的字符串, 并且你从例子中看到有数个 `"\0"`, 包括结尾的那个. . . 编译器会在结尾加上第二个, 就像它对字符串常量通常所做的一样 (你一般不用你自己去加). 这个字符串中的空字符把其分成多个过滤器, 每个有两个部分. 第一个过滤器有描述 "Text Files (*.txt)", 通配符在这里不是必需的, 放在这里只是因为我喜欢它. 接下来的部分是第一

个过滤器的实际的通配符，” *.txt”。我们对第二个过滤器做同样的处理，除了它是一个所有文件的通用过滤器之外。你可以随你喜欢加不同的过滤器。

LpstrFile 指向我们配置来保存文件名的内存，因为文件名不能长于 MAX_PATH，所以我也把这个值当作了此内存的大小。

几个标志表明了对话框对允许用户输入已存在的文件（我们只想打开，不想创建）并隐藏了我们不打算支持的以只读方式打开的选项。最后我们提供了一个默认的文件扩展名，所以如果用户输入了“foo”，而没有找到，那么它会在最后放弃之前去试图打开“foo.txt”。

选择保存文件的代码和打开文件的代码几乎一样，除了调用 GetSaveFileName() 中我们需要改变一些标志以使选项更适合于保存文件。

```
OPENFILENAME ofn;
char szFileName[MAX_PATH] = "";
ZeroMemory(&ofn, sizeof(ofn));
ofn.lStructSize = sizeof(ofn); // SEE NOTE BELOW
ofn.hwndOwner = hwnd;
ofn.lpstrFilter = "Text Files (*.txt)\0*.txt\0All Files (*.*)\0*.*\0";
ofn.lpstrFile = szFileName;
ofn.nMaxFile = MAX_PATH;
ofn.Flags = OFN_EXPLORER | OFN_FILEMUSTEXIST | OFN_HIDEREADONLY;
ofn.lpstrDefExt = "txt";
if(GetOpenFileName(&ofn))
{
    // Do something usefull with the filename stored in szFileName
}
```

这种情下我们不再需要文件存在，但我们需要目录存在，因为我们不想去先创建它。我们也要在用户选择了一个已存在的文件时候提示他们是否想要覆盖它。

注意：MSDN 对 lStructSize 成员作了如下说明：

lStructSize

指明结构体的长度，单位为 Byte。

Windows NT 4.0: 在一个用 WINVER 和 _WIN32_WINNT >= 0x500 的定义编译的应用程序来说，用 OPENFILENAME_SIZE_VERSION_400 作它的成员。

Windows 2000/XP: 用 sizeof(OPENFILENAME) 成这个参数。

简单说来是 Windows 2000 他们对这个结构体加了一些成员，所以它的大小变了。如果上面的代码不能工作的话很可能是你的编译器所用的大小跟你的作业系统（比如 Windows 98, Windows NT 4）所期望的不同，以至调用失败。如果这样的话，试下用 OPENFILENAME_SIZE_VERSION_400 代替 sizeof(ofn)。感谢向我指出这一点的人们。

读写文件

在 windows 系统中你有好几个方式来读写文件。可以用老式的 io.h 中的 open()/read()/write()，可以用 stdio.h 中的 fopen()/fread()/fwrite()，也可以用 C++ 中的

iostream.

但是在 Windows 系统中这些不同的方式最后都是调用的 Win32 API 函数，这也是我要在这里用的。如果你已经对使用另外一种方法来做文件 IO 很熟悉的话，这是很容易适应的，或者你想用你自己的方法来读写文件也可以。

打开文件，用 `OpenFile()` 或 `CreateFile()`。微软推荐只用 `CreateFile()` 因为 `OpenFile()` 现在已经过时了。`CreateFile()` 是一个很灵活的函数，提供了用来打开文件的很多的控制。

读

打个比方你用 `GetOpenFileName()` 允许用户选择文件。...

```
BOOL LoadTextFileToEdit(HWND hEdit, LPCTSTR pszFileName)
{
    HANDLE hFile;
    BOOL bSuccess = FALSE;
    hFile = CreateFile(pszFileName, GENERIC_READ, FILE_SHARE_READ, NULL,
        OPEN_EXISTING, 0, NULL);
    if(hFile != INVALID_HANDLE_VALUE)
    {
        DWORD dwFileSize;
        dwFileSize = GetFileSize(hFile, NULL);
        if(dwFileSize != 0xFFFFFFFF)
        {
            LPSTR pszFileText;
            pszFileText = GlobalAlloc(GPTR, dwFileSize + 1);
            if(pszFileText != NULL)
            {
                DWORD dwRead;
                if(ReadFile(hFile, pszFileText, dwFileSize, &dwRead, NULL))
                {
                    pszFileText[dwFileSize] = 0; // Add null terminator
                    if(SetWindowText(hEdit, pszFileText))
                        bSuccess = TRUE; // It worked!
                }
                GlobalFree(pszFileText);
            }
        }
        CloseHandle(hFile);
    }
    return bSuccess;
}
```

有一个完整的函数来将一个文本文件读入到编辑框中。它将编辑框的句柄和要读入的文件的名字作为参数。这个很严格的函数有一大堆的出错检查，文件 IO 的确是容易出错的地

方， 所以你要记得检错。

注意变量 dwRead.我们在 ReadFile()中不需要， 但这个参数必须提供， 否则就要出错。

对 createFile()的调用中， GENERIC_READ 意即我们只想有读的权限。

FILE_SHARE_READ 的意思是如果其它的程序也在我们打开的时候打开它也可以， 但是它们也要是只读才行， 它们要是在我们读的时候在对其进行写操作就不行。 OPEN_EXISTING 意思是我们只打开已经存在的文件， 不要创建它， 也不要覆盖它。

一旦我们打开了文件并检查了 CreateFile()调用成功后， 我们再来检查文件的大小以便知道我们要配置多少内存来读入整个文件。 我们于是配置内存， 并检查配置的成功与否， 再用 ReadFile()把文件从硬盘读到内存中来。 API 函数不会知道什么文本文件之类的消息所以它们不能读入文本的一行， 或在我们的字符串后加上一个 NULL 终止符。 这就是为什么我们要额外地配置一个字节以便我们在读入整个文件后可以自己加上一个 NULL， 这样我们就可以把我们的整个缓冲区当作一个单字符串当为参数传给 SetWindowText()。

一旦所有的操作都成功了我们就把成功变量设为 TRUE， 并在函数的末尾处作一些清除工作， 在函数最终返回到调用者之前释放配置的内存并关闭文件的句柄。

写

```
BOOL SaveTextFileFromEdit(HWND hEdit, LPCTSTR pszFileName)
{
    HANDLE hFile;
    BOOL bSuccess = FALSE;
    hFile = CreateFile(pszFileName, GENERIC_WRITE, 0, NULL,
        CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    if(hFile != INVALID_HANDLE_VALUE)
    {
        DWORD dwTextLength;
        dwTextLength = GetWindowTextLength(hEdit);
        // No need to bother if there's no text.
        if(dwTextLength > 0)
        {
            LPSTR pszText;
            DWORD dwBufferSize = dwTextLength + 1;
            pszText = GlobalAlloc(GPTR, dwBufferSize);
            if(pszText != NULL)
            {
                if(GetWindowText(hEdit, pszText, dwBufferSize))
                {
                    DWORD dwWritten;
                    if(WriteFile(hFile, pszText, dwTextLength, &dwWritten,
                        NULL))
                        bSuccess = TRUE;
                }
            }
        }
    }
}
```



```
        GlobalFree(pszText);
    }
}
CloseHandle(hFile);
}
return bSuccess;
}
```

与读文件十分类似，写文件的函数只有一点不同。首先在调用 `CreateFile()` 的时候我们要求有读权限，而且文件应该总是新创建（如果已经存在则它将会被打开并清除），如果不存在则它会被以常规属性创建。

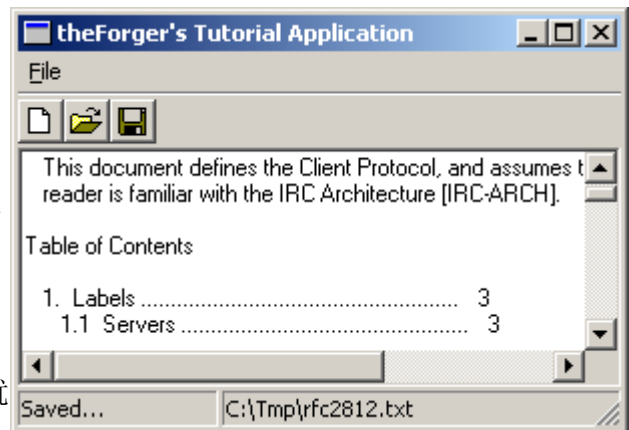
接下来我们要获取编辑框所需要的内存缓冲区，因为这是数据源。一旦我们配置了内存，我用 `GetWindowText()` 从编辑框获取字符串并用 `WriteFile()` 写入文件。与 `ReadFile()` 类似，返回的值是实际被写入的字节数目，不过我们一般不用这个返回值。

应用第三部分:工具栏和状态栏

范例: app_three

关于通用控件的重要说明

对于所有的通用控件,你要用它们之前都要调用 `InitCommonControls()`.还要`#include <comctl.h>`以便使用函数与一些所必须的通用控件的申明与定义.你还需要在链接设置中加上 `comctl32.lib`, 如果它不在那里的话. 注意 `InitCommonControls()`是个旧 API, 为了使用更多的功能你可以使用 `InitCommonControlsEx()`(就是 `InitCommonControlSex()`), 在使用很多最近才有的通用控件的时候,你也必须要用这个函数.这个地方因为我沒有用什么高级功能,所以就用 `InitCommonControls()`就夠了,也简单些.



工具栏

你可以用 `CreateToolbarEx()`创建一个工具栏,但我在这里不这样用.第一件事情就是要实际地创建一个工具栏. . .

```
hTool = CreateWindowEx(0, TOOLBARCLASSNAME, NULL, WS_CHILD | WS_VISIBLE,
0,0, 0, 0,
hwnd, (HMENU)IDC_MAIN_TOOL, GetModuleHandle(NULL), NULL);
```

夠简单了吧, `TOOLBARCLASSNAME` 是在通用控件的头文件中定义的常量. `hwnd` 是父窗口,也就是你要放工具栏的窗口. `IDC_MAIN_TOOL` 是一个标识,在后面如果需要的话,可以用 `GetDlgItem()`和它来获最这个工具栏的 `HWND`.

```
// Send the TB_BUTTONSTRUCTSIZE message, which is required for
// backward compatibility.
SendMessage(hTool, TB_BUTTONSTRUCTSIZE, (WPARAM)sizeof(TBBUTTON), 0);
```

需要这个消息来让系统算出你用的是什麼版本的通用控件库.因为新版本加了一些新东西到这个结构体中去了,所以有了它的大小就知道你可以用它的哪些功能.

工具栏按钮

基本的工具栏上按钮图片有两个来源,标准按钮是 `comctl32` 提供的,用户定义的是你自己创建的. **注意:** 按钮和图片是分别加到工具栏上去的. . . 先加一些图片,再加一些按钮,最后告诉它哪个按钮用哪个图片.

添加标准按钮

现在我们创建了一个工具栏了,我们要向它加一些按钮.最常用的图片就在通用控件库中,所以我们不需要对每个程序来创建,添加一遍就可以使用了.

首先我们申明一个 TBBUTTON 和 TBADDBITMAP.

```
TBBUTTON tbb[3];  
TBADDBITMAP tbab;
```

然后我们向工具栏添加标准的图片，就用通用控件库中定义好的图片表. . .

```
tbab.hInst = HINST_COMMCTRL;  
tbab.nID = IDB_STD_SMALL_COLOR;  
SendMessage(hTool, TB_ADDBITMAP, 0, (LPARAM)&tbab);
```

现在我们装入了我们的图片，我们可以添加一些按钮，使用它们. . .

```
ZeroMemory(tbb, sizeof(tbb));  
tbb[0].iBitmap = STD_FILENEW;  
tbb[0].fsState = TBSTATE_ENABLED;  
tbb[0].fsStyle = TBSTYLE_BUTTON;  
tbb[0].idCommand = ID_FILE_NEW;  
tbb[1].iBitmap = STD_FILEOPEN;  
tbb[1].fsState = TBSTATE_ENABLED;  
tbb[1].fsStyle = TBSTYLE_BUTTON;  
tbb[1].idCommand = ID_FILE_OPEN;  
tbb[2].iBitmap = STD_FILESAVE;  
tbb[2].fsState = TBSTATE_ENABLED;  
tbb[2].fsStyle = TBSTYLE_BUTTON;  
tbb[2].idCommand = ID_FILE_SAVEAS;  
SendMessage(hTool, TB_ADDBUTTONS, sizeof(tbb)/sizeof(TBBUTTON),  
(LPARAM)&tbb);
```

这里我们用标准图片来分别添加一个 New, Open 和 Save As 按钮，这样做一般是需要的，因为用户已经习惯了它们，知道它们是干什么的。

每个图片在图片表中的编号在通用控件头文件中定义好了，并在 MSDN 中有说明。

我们给每个按钮配置一个标识(ID_FILE_NEW 等等. . .)，跟相应的菜单项的标识是一模一样的。这些按钮可以如同相应的菜单项产生一样的 WM_COMMAND 消息，所以不用额外的处理了！如果我们添加了一个不跟相应的菜单对应的按钮，我们只需要搞个新的标识并在 WM_COMMAND 中加一点处理代码就行了。

可能你会在想我传给 TB_ADDBUTTONS 的这个怪怪的 wParam 是干什么的。它是来计算在数组 tbb 中的按钮数目这样我们就不用写个固定的值。如果我写个 3 这里可能是正确的，但是如果我一会加了一个按钮我就要把它改为 4，这样做起来就很不好. . . 你总是一个地方的变动引起尽可能少的别的地方的变动。比如如果 sizeof(TBBUTTON) 是 16 字节(我自己做个比喻，其实是跟平台相关)这样我们有三个按钮的话 sizeof(tbb) 就是 16*3 或 48. 然后 48/16 就是我们的按钮数目，3。

状态栏

在程序中经常和工具栏在一起的还有状态栏， 在窗口的底部显示消息的一个小方块。使用起来是很容易的，创建它们。 . . .

```
hStatus = CreateWindowEx(0, STATUSCLASSNAME, NULL,  
    WS_CHILD | WS_VISIBLE | SBARS_SIZEGRIP, 0, 0, 0, 0,  
    hwnd, (HMENU)IDC_MAIN_STATUS, GetModuleHandle(NULL), NULL);
```

然后设置你需要的分段数目（可选）。如果你不设置的话，它就只有一个分段，使用整个状态栏，你可以如其它的很多控件一样用 `SetWindowText()` 来设置它的文字或从其获取文字。要是有多段的话，你就需要给出每个分段的宽度，并用 `SB_SETTEXT` 来设置每个分段的文字。

为了设置宽度，我们声明一个整数的数组， 每个值分别是各个分段的以像素为单位的宽度。如果你想要某个分段用剩下的所有的宽度，就把它的宽度设为-1 就行。

```
int statwidths[] = {100, -1};  
SendMessage(hStatus, SB_SETPARTS, sizeof(statwidths)/sizeof(int),  
    (LPARAM)statwidths);  
SendMessage(hStatus, SB_SETTEXT, 0, (LPARAM)"Hi there :)");
```

这里的 `wParam` 又是用来计算中数组中有多少元素的。一旦我们完成了添加分段，我们对第一个(序号为 0)进行设置看看效果。

合适地设置大小

跟菜单不一样，工具栏和状态栏是在父窗口的客户区域内的独立的控件。这样一来，我们如果还是用原来的 `WM_SIZE` 处理代码，它们就会覆盖掉我们在前面的章节添加的编辑框控件。这是个显然需要更正的理由。 . . . 在 `WM_SIZE` 的处理中，我们把工具栏和状态栏移到一个位置，并把它们的高度和宽度从客户区域减去以使我们可以让编辑框填充剩下的空间。 . . .

```
HWND hTool;  
RECT rcTool;  
int iToolHeight;  
HWND hStatus;  
RECT rcStatus;  
int iStatusHeight;  
HWND hEdit;  
int iEditHeight;  
RECT rcClient;  
// Size toolbar and get height  
hTool = GetDlgItem(hwnd, IDC_MAIN_TOOL);  
SendMessage(hTool, TB_AUTOSIZE, 0, 0);  
GetWindowRect(hTool, &rcTool);
```

```
iToolHeight = rcTool.bottom - rcTool.top;
// Size status bar and get height
hStatus = GetDlgItem(hwnd, IDC_MAIN_STATUS);
SendMessage(hStatus, WM_SIZE, 0, 0);
GetWindowRect(hStatus, &rcStatus);
iStatusHeight = rcStatus.bottom - rcStatus.top;

// Calculate remaining height and size edit
GetClientRect(hwnd, &rcClient);
iEditHeight = rcClient.bottom - iToolHeight - iStatusHeight;
hEdit = GetDlgItem(hwnd, IDC_MAIN_EDIT);
SetWindowPos(hEdit, NULL, 0, iToolHeight, rcClient.right, iEditHeight,
SWP_NOZORDER);
```

不幸的是，这段代码有点长，但是很简单．．． 工具栏可以自己调整自己的位置当向它发送 TB_AUTOSIZE 消息的时候，如果发送 WM_SIZE 的时候状态栏也可以．（通用控件库不以一致性好而闻名．）

Copyright © 1998-2003, Brook Miles ([theForger](#)). All rights reserved.

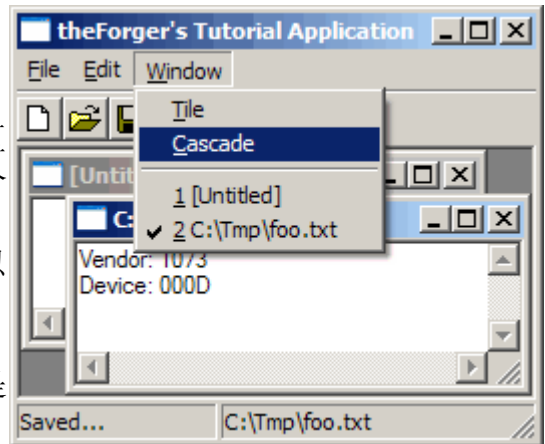
应用第四部分:多文档界面

范例: app_four

MDI 概述

首先来点背景知识...每个窗口都有个客户区域,大多数程序在这里画图,放置控件等等...客户区域跟窗口不是分开的两部分,它仅仅是窗口中的一个特定的小些的区域.有时一个窗口的所有部分都是客户区域,有时一点也没有,有时候客户区域小一点以给菜单,标题,滚动栏等等留出位置.

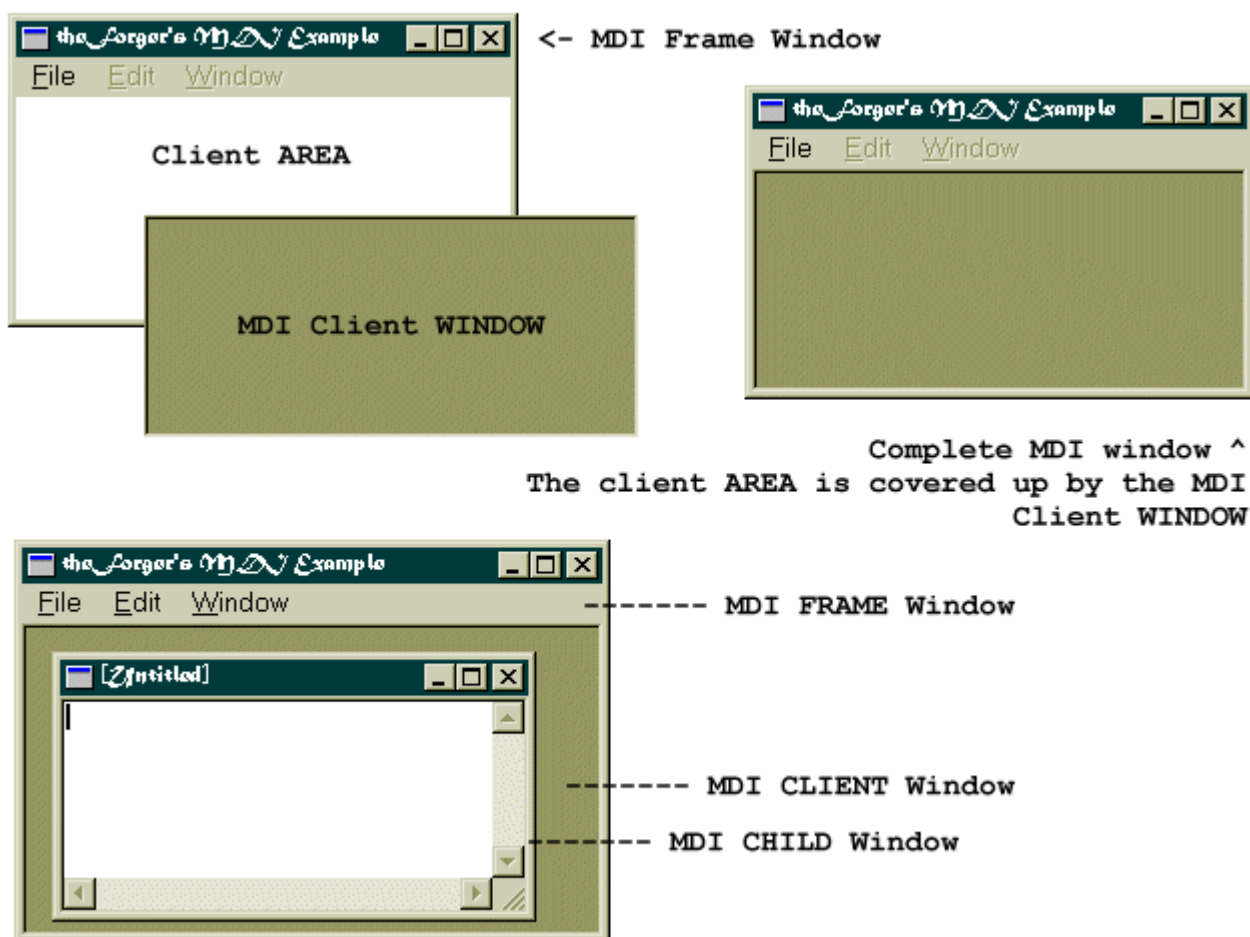
在多文档术语中,主窗口被称为框架,这可能是你在 SDI (单文档界面) 程序中唯一的窗口.在 MDI 中还有一个额外的窗口,称为 MDI 客户窗口,它是你的框架窗口的一个子窗口.跟客户区域不同,它是一个完全独立的窗口,它有自己的客户区域,还可能有一点像素的宽度用作边框.你永远不用直接处理 MDI 客户窗口的消息,它被预定义的窗口类"MDIClient".你可以通过消息操作 MDI 客户区域并让其跟它包含的窗口来通信.



当它到你实际显示文档或显示别的什么东西的窗口时,你就向 MDI 客户窗口发送一个消息告诉它创建一个你指定的类型的新窗口.新窗口是以一个 MDI 客户窗口的子窗口来创建的,而不是你的框架窗口的子窗口.新窗口是一个 MDI 子窗口. MDI 子窗口是 MDI 客户窗口的一个子窗口, MDI 窗口又是 MDI 框架窗口的一个子窗口(搞昏了吧?) 更糟的是, MDI 子窗口还可能有自己的子窗口, 比如本段中的样例的编辑框控件.

你要写两个(或更多)的窗口过程.. 一个如以前一样,为我们的主窗口(即框架窗口),一个是 MDI 子窗口. 你可能有多种类型的子窗口,这种情况下,你可能给每个类型写个窗口过程.

如果我用 MDI 客户窗口这些东西把你完全搞混了,下面的这个方框图可能可以解释得清楚些:



MDI 基础

MDI 需要一些程序在细节上作一些改动， 所以请把这章节仔细地看完. . . 要是你的 MDI 程序不能工作或是行为怪异，很可能是你漏掉了一些与普通程序不同的地方.

MDI 客户窗口

在我们创建我们的 MDI 窗口之前我们要对我们窗口过程中的默认消息处理地方作一些修改. . . 因为我们要创建一个要包含一个 MDI 客户窗口的框架窗口，我们要改变 DefWindowProc() 为 DefFrameProc() 调用，后者加了一些框架窗口的一些特定的消息处理.

default:

```
return DefFrameProc(hwnd, g_hMDIClient, msg, wParam, lParam);
```

下一步是创建 MDI 客户窗口自己，作为我们框架窗口的一个子窗口. 我们依旧在 WM_CREATE 的处理中如下操作:

```
CLIENTCREATESTRUCT ccs;
ccs.hWindowMenu = GetSubMenu(GetMenu(hwnd), 2);
ccs.idFirstChild = ID_MDI_FIRSTCHILD;
g_hMDIClient = CreateWindowEx(WS_EX_CLIENTEDGE, "mdiclient", NULL,
    WS_CHILD | WS_CLIPCHILDREN | WS_VSCROLL | WS_HSCROLL | WS_VISIBLE,
    CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
```



```
hwnd, (HMENU)IDC_MAIN_MDI, GetModuleHandle(NULL), (LPVOID)&ccs);
```

菜单句柄是 MDI 客户窗口用来对每个它创建的窗口添加一个相应的项的弹出菜单的句柄，允许用户在这个菜单中来选择他们想要激活的窗口，我们马上就来为这个事件添加功能代码。这个例子中它为第三个弹出(序号为 2)因为我在 File 后加了 Edit 和 Window。

ccs.idFirstChild 是用来作为客户窗口添至窗口菜单的项的第一个标识的数字...你想要使其跟你的自己的菜单标识区别开来，这样可以处理你的菜单命令并将 Windows 菜单命令传至 DefFrameProc()作处理。在例子中我指定了一个定义为 50000 的标识，高到我确定我的菜单命令标识不会超过它。

现在我们还要在我们的 WM_COMMAND 处理代码中添加一些东西来使这个菜单工作正常：

```
case WM_COMMAND:
    switch(LOWORD(wParam))
    {
        case ID_FILE_EXIT:
            PostMessage(hwnd, WM_CLOSE, 0, 0);
            break;
        // ... handle other regular IDs ...
        // Handle MDI Window commands
        default:
        {
            if(LOWORD(wParam) >= ID_MDI_FIRSTCHILD)
            {
                DefFrameProc(hwnd, g_hMDIClient, msg, wParam, lParam);
            }
            else
            {
                HWND hChild = (HWND)SendMessage(g_hMDIClient,
                    WM_MDIGETACTIVE, 0, 0);
                if(hChild)
                {
                    SendMessage(hChild, WM_COMMAND, wParam, lParam);
                }
            }
        }
    }
    break;
```

我加了一个 default:来处理所有我没有直接处理的消息并检查它是否超过或对于 ID_MDI_FIRSTCHILD。如果是，那么是用户点击了一个 Window 菜单项我们就把消息送给 DefFrameProc()来处理。

如果不是我们的 Window 标识我们就获取句柄来激活子窗口并把消息传给它处理。这样可使你将责任推给子窗口来完成一些特定的动作，并允许不同的子窗口按照需要以不同的方式来处理消息。这个例子中我只处理了我们的框架窗口过程中的全局命令消息，并将影响特定子窗口或文档的消息发送给子窗口自己来处理。

因为我们以最后一个例子来作为基础来构建的，所以调整 MDI 客户窗口大小的代码跟那个例子中调整编辑框的大小的代码一样，负责处理工具栏和状态栏的大小与位置以使它们不会覆盖 MDI 客户窗口。

我们也要把我们的消息循环改一点...

```
while(GetMessage(&Msg, NULL, 0, 0))
{
    if (!TranslateMDISysAccel(g_hMDIClient, &Msg))
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
}
```

我们加了一个额外的步骤(TranslateMDISysAccel())，是用来检查预定义好的一些加速键的，Ctrl+F6 切换到下个窗口，Ctrl+F4 关闭子窗口等等。如果你不添加这些检查的话，你就要被你的用户抱怨没有提供他们已经熟悉的标准功能，或者你要另外手动实现这些功能。子窗口类除了我们的主窗口（框架窗口）之外，我们还要为我们所要的每个类型的子窗口创建一个新的窗口类。比如你可能需要一个来显示文字，一个显示图形或图片。这个例子中我们只创建一种子窗口类型，具备就像我们在前面的章节的那个编辑器程序一样的功能。

```
BOOL SetupMDIChildWindowClass(HINSTANCE hInstance)
{
    WNDCLASSEX wc;
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = MDIChildWndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_3DFACE+1);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = g_szChildClassName;
    wc.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
    if(!RegisterClassEx(&wc))
    {
        MessageBox(0, "Could Not Register Child Window", "Oh Oh...",
```

```

        MB_ICONEXCLAMATION | MB_OK);
    return FALSE;
}
else
    return TRUE;
}

```

这基本上跟我们的普通的框架窗口的注册一样，没有特别的 MDI 的特征。我们把菜单设为 NULL，并设置指向我们马上要写的子窗口过程的窗口过程。

MDI 子窗口过程

MDI 子窗口的窗口过程跟一般的窗口过程只有一些区别。首先默认的消息传给 DefMDIChildProc() 而不是 DefWindowProc()。

这个例子中，我们还要在不需要的时候禁止 Edit 和 Window 菜单（就是因为这件事情看起来很酷，所以要做），所以我们处理 WM_MDIACTIVE 并根据我们的窗口是否被激活来使能或禁止它们。如果有多种类型的子窗口，这里你就可以放一些代码使程序根据不同的激活窗口来改变菜单或工具栏或是修改程序的一些别的方面。

为了更完整，我们可以也禁止 Close 和 Save 文件菜单，因为如果没有窗口它们也没有什么好处。我在资源中把它们默认地都禁止了所以我不需要再加一些代码在应用引导的时候来做这件事情。

```

LRESULT CALLBACK MDIChildWndProc(HWND hwnd, UINT msg, WPARAM wParam,
LPARAM lParam)
{
    switch(msg)
    {
        case WM_CREATE:
        {
            HFONT hfDefault;
            HWND hEdit;
            // Create Edit Control
            hEdit = CreateWindowEx(WS_EX_CLIENTEDGE, "EDIT", "",
                WS_CHILD | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL |
                ES_MULTILINE |
                ES_AUTOVSCROLL | ES_AUTOHSCROLL,
                0, 0, 100, 100, hwnd, (HMENU)IDC_CHILD_EDIT,
                GetModuleHandle(NULL), NULL);
            if(hEdit == NULL)
                MessageBox(hwnd, "Could not create edit box.", "Error", MB_OK |
                    MB_ICONERROR);
            hfDefault = GetStockObject(DEFAULT_GUI_FONT);
            SendMessage(hEdit, WM_SETFONT, (WPARAM)hfDefault,
                MAKELPARAM(FALSE,

```

```

        0));
    }
    break;
case WM_MDIACTIVATE:
{
    HMENU hMenu, hFileMenu;
    UINT EnableFlag;
    hMenu = GetMenu(g_hMainWindow);
    if(hwnd == (HWND)lParam)
    {
        //being activated, enable the menus
        EnableFlag = MF_ENABLED;
    }
    else
    {
        //being de-activated, gray the menus
        EnableFlag = MF_GRAYED;
    }
    EnableMenuItem(hMenu, 1, MF_BYPOSITION | EnableFlag);
    EnableMenuItem(hMenu, 2, MF_BYPOSITION | EnableFlag);
    hFileMenu = GetSubMenu(hMenu, 0);
    EnableMenuItem(hFileMenu, ID_FILE_SAVEAS, MF_BYCOMMAND |
        EnableFlag);
    EnableMenuItem(hFileMenu, ID_FILE_CLOSE, MF_BYCOMMAND |
        EnableFlag);
    EnableMenuItem(hFileMenu, ID_FILE_CLOSEALL, MF_BYCOMMAND |
        EnableFlag);
    DrawMenuBar(g_hMainWindow);
}
break;
case WM_COMMAND:
    switch(LOWORD(wParam))
    {
        case ID_FILE_OPEN:
            DoFileOpen(hwnd);
            break;
        case ID_FILE_SAVEAS:
            DoFileSave(hwnd);
            break;
        case ID_EDIT_CUT:
            SendDlgItemMessage(hwnd, IDC_CHILD_EDIT, WM_CUT, 0, 0);
            break;
        case ID_EDIT_COPY:
            SendDlgItemMessage(hwnd, IDC_CHILD_EDIT, WM_COPY,
                0, 0);
            break;
        case ID_EDIT_PASTE:

```

```

        SendDlgItemMessage(hwnd, IDC_CHILD_EDIT, WM_PASTE, 0, 0);
        break;
    }
    break;
case WM_SIZE:
{
    HWND hEdit;
    RECT rcClient;
    // Calculate remaining height and size edit
    GetClientRect(hwnd, &rcClient);
    hEdit = GetDlgItem(hwnd, IDC_CHILD_EDIT);
    SetWindowPos(hEdit, NULL, 0, 0, rcClient.right, rcClient.bottom,
        SWP_NOZORDER);
}
return DefMDIChildProc(hwnd, msg, wParam, lParam);
default:
    return DefMDIChildProc(hwnd, msg, wParam, lParam);
}
return 0;
}

```

我把 File Open 和 Save as 都当作命令来处理，DoFileOpen()和 DoFileSave()跟前面的章节中编辑框控件的标识几乎一样，并且添加了将 MDI 子窗口的标题设为文件名的功能。

编辑命令很简单，因为里面的编辑框本身就支持它们，我们只需要告诉它做什么就行。

记得我提到过有些事情你如果不记住的话你的程序就会行为怪异吗？注意到我在 WM_SIZE 的处理末尾调用了 DefMDIChildProc()没有，这点很重要不然系统就没有机会为这个消息作它自己的处理。你可以在 MSDN 中查下 DefMDIChildProc()处理的消息，记得在这些消息处理的后面加上它。

创建和销毁窗口

MDI 子窗口不是直接创建的，而是我们向客户窗口发送一个 WM_MDICREATE 消息告诉它我们要创建一个以 MDICREATESTRUCT 的某些成员指定的什么样的一个子窗口。你可以在你的文档中查一下这个结构体的成员，都相当简单。WM_MDICREATE 消息处理函数的返回值就是新创建的窗口的句柄。

```

HWND CreateNewMDIChild(HWND hMDIClient)
{
    MDICREATESTRUCT mcs;
    HWND hChild;
    mcs.szTitle = "[Untitled]";
    mcs.szClass = g_szChildClassName;
    mcs.hOwner = GetModuleHandle(NULL);
}

```

```

mcs.x = mcs.cx = CW_USEDEFAULT;
mcs.y = mcs.cy = CW_USEDEFAULT;
mcs.style = MDIS_ALLCHILDSTYLES;
hChild = (HWND)SendMessage(hMDIClient, WM_MDICREATE, 0, (LONG)&mcs);
if(!hChild)
{
    MessageBox(hMDIClient, "MDI Child creation failed.", "Oh Oh...",
        MB_ICONEXCLAMATION | MB_OK);
}
return hChild;
}

```

在 MDICREATESTRUCT 中我这里没有用的一个很有用的成员是 IParam. 这个成员用来向子窗口发送任何 32bit 的值（比如一个指针）以传递任何你选择的自定义消息。在你子窗口的 WM_CREATE 消息处理中，WM_CREATE 的 IParam 值将指向一个 CREATESTRUCT。这个结构的 lpCreateParams 成员将指向你随 WM_MDICREATE 发送的 MDICREATESTRUCT。所以要在子窗口中访问 IParam，你要在子窗口的窗口过程中做些事情...

```

case WM_CREATE:
{
    CREATESTRUCT* pCreateStruct;
    MDICREATESTRUCT* pMDICreateStruct;
    pCreateStruct = (CREATESTRUCT*)IParam;
    pMDICreateStruct = (MDICREATESTRUCT*)pCreateStruct->lpCreateParams;
    /*
    pMDICreateStruct now points to the same MDICREATESTRUCT that you
    sent along with the WM_MDICREATE message and you can use it
    to access the IParam.
    */
}
break;

```

如果你不想麻烦额外的两个指针就可以就这样一步到位访问 IParam ((MDICREATESTRUCT*)((CREATESTRUCT*)IParam)->lpCreateParams)->IParam 现在我们可以我们的框架窗口中实现菜单上的文件命令了：

```

case ID_FILE_NEW:
    CreateNewMDIChild(g_hMDIClient);
    break;
case ID_FILE_OPEN:
{
    HWND hChild = CreateNewMDIChild(g_hMDIClient);
    if(hChild)
    {

```

```

        DoFileOpen(hChild);
    }
}
break;
case ID_FILE_CLOSE:
{
    HWND hChild = (HWND)SendMessage(g_hMDIClient, WM_MDIGETACTIVE, 0, 0);
    if(hChild)
    {
        SendMessage(hChild, WM_CLOSE, 0, 0);
    }
}
break;

```

我们也可以为我们的 Window 菜单提供一些默认的 MDI 处理，因为 MDI 本身就支持这样所以要做的事情不是很多。

```

case ID_WINDOW_TILE:
    SendMessage(g_hMDIClient, WM_MDITILE, 0, 0);
break;
case ID_WINDOW_CASCADE:
    SendMessage(g_hMDIClient, WM_MDICASCADE, 0, 0);
break;

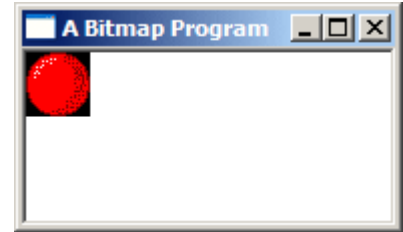
```


位图，设备上下文和 BitBlt

范例：bmp_one

GDI

MS Windows 跟 DOS 比起来，真正比较伟大的一件事就是你不用知道你用的什么显示硬件就可以显示图形。与 DOS 不同的是，Windows 提供了一套 API 就作图形设备接口(Graphics Device Interface)，或称之为 GDI。GDI 用一套通用的图形对象来向屏幕，内存甚至是打印机绘图。



设备上下文

GDI 跟一个跟设备上下文（Device Context，DC）的概念联系得很紧密，由 HDC 这个数据类型来代表（Handle to Device Context，设备上下文的句柄）。一个 HDC 大概就是一个你可以向其绘图的句柄；它可以代表整个屏幕，一个窗口的客户区域，一个存在内存中的位图，或是一个打印机。比较好的一点就是你甚至不需要知道它是指向哪里，你可以用一种方法来使用它，在你写自定义的绘图函数时候你就可以把它用于任何上述设备而为每个设备作修改。

HDC 跟大多数的 GDI 对象一样有点生硬，就是说你不能直接访问它的数据。．．．但是你可以将其传给可以操作它的很多 GDI 函数，比如绘图，获取它的消息，或对其作出一些修改。

例如，如果你想在窗口上绘图，首先你要用 GetDC() 来获取代表这个窗口的 HDC，然后你就可以用任何以 HDC 为参数的 GDI 函数比如 BitBlt() 来绘图，TextOut() 来输出文字，LineTo() 来画线等等。

位图

位图可以如前面章节的例子中的图标一样来装入，有一个 LoadBitmap() 来完成大多数的基本操作比如简单地装入一个位图资源，还有 LoadImage() 可用来从一个 *.bmp 文件装入位图，就像从图标中一样。

GDI 有个不方便的地方是你不能直接从向位图对象（HBITMAP 类型）绘图。记住绘图操作已经被设备上下文抽象，所以要对一个位图对象绘图，你首先要创建一个内存 DC，然后用 SelectObject() 把 HBITMAP 选入其中。最后的效果就是这个 HDC 指向的“设备”就是内存中的位图，当你操作这个 HDC 时，操作实际上是在那个位图上起了效果。我提到的，有一个很方便的方法，只要你可写为向一个 HDC 绘图写代码你就可以它其用在一个窗口 DC 或一个内存 DC 上而不需要任何的检查和改动。

你自己是不能操作在内存中的位图数据的。你可以用设备独立位图（Device independent Bitmaps，DIB）来这样做，并且你甚至可以把 GDI 和一般的操作一起用在 DIB 上。但是这里，因为已经超出了我们这个简单的教程的范围所以我们现在只讨论简单的 GDI 操作本身。

GDI 泄漏

一旦完成了对一个 HDC 的操作，释放它是很重要的(具体怎么样做取决于你是怎么样得到它的，这点我们马上将要谈到).GDI 对象是有限的. 在 windows95 之前的版本，它们不限是难以致信的有限而且是在整个系统中共享，所以如果一个程序用了过多的话，别的程序就根本不能绘任何东西!幸运的是现在的情况已经不这样了，而且你可以在 Windows2000 和 XP 中用很多个的资源，直至情况变得很糟...但是释放 GDI 对象是很容易被忘记的，在 Windows 9x 下你的程序可以很快地用完 GDI 资源.理论上讲在 NT(NT/2K/XP)系统中你不应该可能把系统的 GDI 资源用完，但是还是存在极端情况，或者你就是不经意地冒险。

如果你的程序在刚开始的几分钟运行正常然后开始奇怪地绘图或根本就不画图了，这就表明你很可能泄漏了 GDI 资源.值得你注意要释放的 GDI 对象不仅仅只有 HDC， 但是把位图，字体之类的资源在程序的整个运行周期中保存基本上是安全的，这样做比每次需要就装入一次要有效率些。

还有，一个 HDC 一次只能保存一种类型的对象（位图，字体，笔...），当你选入一个新的进去的时候它将原来的返回. 合理地处理这个对象是很重要的.如果你完全忽略它，它将在内存中堆积并导致 GDI 泄漏.当一个 HDC 被创建，它也和某些默认选入它的对象一起创建... 当它们返回至你的时候最好将它们存起来，并在你完成了 HDC 绘图你就要把它们存回去.这不仅仅只是把你自己的对象从 HDC 删除（这是个好事情）也可以将默认的对象合理地放在当你释放或销毁这个 HDC 的时候可以看到的方（这是非常好的）。

重要更新：不是所有的对象默认地选入 HDC，你可以看看 MSDN 找出哪些不是这样的。因为我先前不确定 HBITMAP 是否是它们中的一个，因为看起来没有关于这个的确定的文档，并且样例(即使是微软自己的)经常忽略默认的位图.自从几年前原来的教程被写作后，我确定了事实上有个默认的位图需要释放.这个消息是由 Shaun Ivory 提供的，一个微软的工程师，也是我在 #winprog 上的一个朋友。

事实上原来微软写的一个屏幕保护程序有一个 bug，后来被发现就是因为默认的位图对象没有被替换或销毁，它就慢慢地用完了 GDI 资源。警惕！这是一个很容易犯的错误。

显示位图

好了，开始谈正事。对于一个窗口最简单的绘图操作发生于对 WM_PAINT 消息的处理中.当你的窗口从最小化或是从一个覆盖它的窗口恢复到被显示， Windows 向你的窗口发送 WM_PAINT 消息来让你知道你的窗口需要重新绘它的内容.当你在屏幕上绘图时候，并不是永久地绘上去，只是在它被别的什么东西覆盖它的之前存在，而且在适当的时候你需要重新绘它。

```
HBITMAP g_hbmBall = NULL;
case WM_CREATE:
    g_hbmBall = LoadBitmap(GetModuleHandle(NULL),
                          MAKEINTRESOURCE(IDB_BALL));
    if(g_hbmBall == NULL)
        MessageBox(hwnd, "Could not load IDB_BALL!", "Error", MB_OK |
                      MB_ICONEXCLAMATION);
```

```
break;
```

首先要做的当然是装入位图，对于一个位图资源来说这很简单，跟装入其它的什么资源没有什么很大的差异。然后我们开始绘图。．．

```
case WM_PAINT:
{
    BITMAP bm;
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hwnd, &ps);
    HDC hdcMem = CreateCompatibleDC(hdc);
    HBITMAP hbmOld = SelectObject(hdcMem, g_hbmBall);
    GetObject(g_hbmBall, sizeof(bm), &bm);
    BitBlt(hdc, 0, 0, bm.bmWidth, bm.bmHeight, hdcMem, 0, 0, SRCCOPY);
    SelectObject(hdcMem, hbmOld);
    DeleteDC(hdcMem);
    EndPaint(hwnd, &ps);
}
break;
```

获取窗口 DC

在开始的时候我们定义一些变量。注意第一个是一个位图，而不是一个 HBITMAP。BITMAP 是一个保存实际的 GDI 对象 HBITMAP 的有关消息的一个结构体。我们需要一种得到 HBITMAP 的高度和宽度的方法，所以我们使用 GetObject() 这个函数，跟它的名字相反，它并不能真正地获取一个对象，而是获取一个已经存在的对象的相关消息。如果是 "GetObjectInfo" 这个名字还更恰切些。GetObject() 可以对不同种类的 GDI 对象使用，具体地是通过第二个参数，即结构体的大小来区分的。

PAINTSTRUCT 是一个结构体，保存有要绘图的窗口和要实际绘图的内容相关的消息。对于大多数简单的任务，你可以简单地忽略它包含的消息，但调用 BeginPaint() 时要用到它。BeginPaint() 这个函数，如同它的名字一样被特意设计来处理 WM_PAINT 消息的。当不是处理一个 WM_PAINT 消息的时候，你可以使用 GetDC()，这个我们马上会在我们的定时动画样例中看到...但是在 WM_PAINT 消息的处理中，BeginPaint() 与 EndPaint() 的使用是很重要的。

BeginPaint() 向我们返回一个 HDC，代表我们传给它的 HWND，就是 WM_PAINT 消息来处理的那个句柄。我们对这个 HDC 所做的任何操作将立即在屏幕上显示出来。

为位图创建一个内存 DC

与同我在上面提到的，为了在位图上或用位图来绘图，我们要在内存中创建一个 DC。．．这里最简单的方法是用 CreateCompatibleDC() 与我们已经有的那个来创建。这样做我们就会得到一个与我们的窗口的 HDC 的颜色深度和显示属性相兼容的内存 DC。

现在我们调用 SelectObject() 将这个位图选入这个 DC 来，小心地存储默认的位图这样我

们才能在后面来替换它时候不引起 GDI 对象泄漏。

绘图

一旦我们将这个位图的大小得到并填入到 BITMAP 结构体中去，我们就可以调用 BitBlt()来将这个图像从内存 DC 拷贝到窗口 DC 中去，以在屏幕上显示。还是老话，你可以在 MSDN 中查它的每个参数，但简单地讲，它们是：目标，位置与大小，源与源位置，最后是光栅操作(Raster Operation, ROP 代码)，这是用来指明怎样复制的。在本例中我们只想简单地把源进行复制，不搞花里胡哨。

BitBlt()可能是所有 Win32 API 中最 Happy 的一个函数，任何想在 Windows 下写点游戏或是别的图形应用的人都会喜欢它。也可能是我第一个记住了所有的参数的 API 函数。

清除

这个时候位图应该已经在屏幕上了，我们要自己来做清洁工作。要做的第一件事情是把内存 DC 恢复到我们获取它时候的状态，意思就是要用我们存储的默认的位图来替换我们的位图。下一步就可以用 DeleteDC()来把它们一起删除。

最后我们用 EndPaint()来释放从 BeginPaint()得到的窗口 DC。

销毁一个 HDC 有时有点令人迷惑，因为根据你获得它的方式，至少有三种方法来完成这个操作。

下面是一些常用的获取一个 DC，和怎样在使用完释放的方法。

- GetDC() -- ReleaseDC()
- BeginPaint() -- EndPaint()
- CreateCompatibleDC() -- DeleteDC()

最后在我们程序终止的时候，我们想把我们配置的所有的资源释放掉。从技术上讲这不是绝对地需要，因为现代的 Windows 平台在你的程序终止的时候已经可以很智能地把你的所有东西释放掉，但是把你自己的东西管理好是很有好处的，因为如果你偷懒不去删除它们，它们的习惯就会变得很松散。而且毫无质疑的，尤其在一些老的版本的 Windows 中还有很多的 bug，并不能完全清除你的 GDI 对象，如果你自己不完成这一点的话。

```
case WM_DESTROY:  
    DeleteObject(g_hbmBall);  
    PostQuitMessage(0);  
break;
```

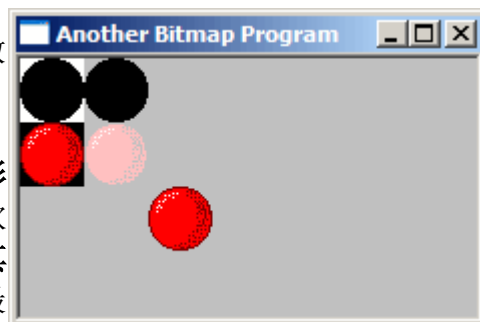
透明位图

范例: bmp_two

透明

给位图加上透明效果很简单,除了我们要显示透明效果的彩色图像外,还要用到它的黑白色的掩图.

以下是要正确显示我们的这种效果的条件:第一,彩色的图像的我们要显示为透明的地方要都是黑色的.其次掩图中我们要显示为透明效果的地方要都是白的,并在其它的地方都是黑色的.彩色图像和它的掩图在例图中的最左边显示.



BitBlt 操作

我们怎么样做到透明效果的?首先我们将 BitBlt() 的最后一个参数设为 SRCAND 来显示掩图,然后再把其设为 SRCPAINT 用 BitBlt() 来显示彩图.结果就是我们想要透明的地方并没有改变而其它的地方如通常一样显示.

```
SelectObject(hdcMem, g_hbmMask);
BitBlt(hdc, 0, 0, bm.bmWidth, bm.bmHeight, hdcMem, 0, 0, SRCAND);
SelectObject(hdcMem, g_hbmBall);
BitBlt(hdc, 0, bm.bmHeight, bm.bmWidth, bm.bmHeight, hdcMem, 0, 0,
SRCPAINT);
```

相当简单吧?有点幸运,但是有个问题...掩图从哪里来?大致上有两个方法来获取掩图...

- 就用你创建彩图的绘图工具来自己创建,当你使用得不多的话这样做也是合理的.这样的话你就把你创建的掩图加入资源并用 LoadBitmap() 来装入它.
- 在你程序运行的时候来创建,把你原来的图片中的一种颜色选为''透明''色,创建一个这种颜色存在的地方为白色的,其它的地方都为黑色的图片.

因为第一种方法没有什么可以说的,如果你愿意的话就可以采取这种方法.第二种牵涉一些 BitBlt() 的技巧,所以我来演示一下这种方法.

创建掩图

最简单的方法,就是遍历彩图中的每个像素,检查它的值并把相应的像素设为掩图中的黑或白...但是 SetPixel() 绘图起来很慢,并且这也是不太实际.

用 BitBlt() 来把彩图转为黑白色则有效率得多.如果你用 BitBlt() 带上 SRCCOPY 参数把保存有一个彩图的 HDC 写向一个保存有一个黑白图片的 HDC,它就会检查彩图中设为背景色的颜色,并把这种颜色的像素都设为白,任何不是背景色的其它像素都会设为黑色.

这对我们来说是个很好的功能，因为我们需要做的就是把我们彩图中要透明效果的颜色指定为背景色，并把彩图用 BitBlt() 从彩图绘向掩图就行了。注意这只能对单色的掩图有效（黑和白）... 就是说每个像素的颜色深度为 1 个 bit 的位图。如果你对一个只有黑白像素的彩图这样操作，但是位图的深度大于 1 个 bit (比如 16 或是 24bit) 就不行。

记住上面的我们要成功的首要条件没有？就是我们要透明效果的地方都要是黑色的。因为我们用于此例的位图已经满足这个条件了，不需要再来怎么操作了，但是如果你把这里的代码用于另外一个位图相使一个不同的颜色为透明效果（一般是粉红色）那我们就要进行第二步的操作，就是用我们刚刚创建的掩图来把原图变一点，以使我们要透明效果的地方都为黑色。别的什么地方也是黑色也不要紧，因为掩图上它们不是白色的，它们就不会为透明的。我们可以用 BitBlt() 带上 SRCINVERT 参数来把新的掩图写向原彩图，把它的掩图中为白色的地方设为黑色。

这里可能有点复杂，所以就用一个小函数来为我们完成这个操作，如下面：

```
HBITMAP CreateBitmapMask(HBITMAP hbmColour, COLORREF crTransparent)
{
    HDC hdcMem, hdcMem2;
    HBITMAP hbmMask;
    BITMAP bm;
    // Create monochrome (1 bit) mask bitmap.
    GetObject(hbmColour, sizeof(BITMAP), &bm);
    hbmMask = CreateBitmap(bm.bmWidth, bm.bmHeight, 1, 1, NULL);
    // Get some HDCs that are compatible with the display driver
    hdcMem = CreateCompatibleDC(0);
    hdcMem2 = CreateCompatibleDC(0);
    SelectBitmap(hdcMem, hbmColour);
    SelectBitmap(hdcMem2, hbmMask);
    // Set the background colour of the colour image to the colour
    // you want to be transparent.
    SetBkColor(hdcMem, crTransparent);
    // Copy the bits from the colour image to the B+W mask... everything
    // with the background colour ends up white while everything else ends up
    // black... Just what we wanted.
    BitBlt(hdcMem2, 0, 0, bm.bmWidth, bm.bmHeight, hdcMem, 0, 0, SRCCOPY);
    // Take our new mask and use it to turn the transparent colour in our
    // original colour image to black so the transparency effect will
    // work right.
    BitBlt(hdcMem, 0, 0, bm.bmWidth, bm.bmHeight, hdcMem2, 0, 0, SRCINVERT);
    // Clean up.
    DeleteDC(hdcMem);
    DeleteDC(hdcMem2);
    return hbmMask;
}
```

注意：这个函数用了 `SelectObject()` 来暂时地把我们传向一个 HDC 的位图选入。一个位图不能同时被选入多个 HDC，所以要确认这个位图在你调用这个函数的时候没有被选入另外一个 HDC，否则就会出错。现在有了这个可爱的小函数，我们就装入一个原图的时候立马上就可以创建一个掩图了：

```
case WM_CREATE:
    g_hbmBall = LoadBitmap(GetModuleHandle(NULL),
                           MAKEINTRESOURCE(IDB_BALL));
    if(g_hbmBall == NULL)
        MessageBox(hwnd, "Could not load IDB_BALL!", "Error", MB_OK |
                    MB_ICONEXCLAMATION);
    g_hbmMask = CreateBitmapMask(g_hbmBall, RGB(0, 0, 0));
    if(g_hbmMask == NULL)
        MessageBox(hwnd, "Could not create mask!", "Error", MB_OK |
                    MB_ICONEXCLAMATION);
    break;
```

第二个参数当然是我们要在原图中显示为透明效果的颜色，这个地方为黑色。

这些是怎么工作的啊？

...你可能会问这句话。如果你编了这么久的 C 或是 C++ 程序的话，你应该熟悉这些二进制的操作如 OR，XOR，AND，NOT 等等。我不打算完全讲解这些东西，但是我要说说我怎样在这个例子中用它们的。如果我的解释不够清楚(好像就是这样的)，去复习一下二进制操作的知识可能就明白了。理解不是现在使用它们的必要条件，你如果想完全信任它们你也可以略过理解这一步。

SRCAND

`BitBlt()` 的 SRCAND 光栅操作或称为 ROP 代码意味著用 AND 来组合位。就是说：只有源和目的的位都是 1 结果才为 1。我们用这个操作来把我们的彩图中的那些有颜色部分在掩图中都设为黑色。我们要颜色的地方在掩图中都是黑色(在二进制上都为 0)，然后掩图中所有的为黑色的像素在结果中被设为 0，所以也是黑色。所有用 AND 跟 1 操作的值都不会受影响，如果原来是 1 就得出 1，原来是 0 就得出 0... 所以我们的掩图中为白色的部分在 `BitBlt()` 调用之后完全没有被影响。结果就是样例图片中的右上角的那个图像。

SRCPAINT

SRCPAINT 是用的 OR 操作，所以如果两者之一或两者都为 1，则结果就为 1。我们对彩图进行这个操作。当我们的彩图的黑色部分(透明部分)用 OR 来跟目的数据组合，结果就是没有影响，因为任何值跟 0 进行 OR 运算的数都会不受影响。

但是，我们彩图的其它部分不是黑色的，如果目的数据也不是黑色的，我们就得到源与目的色彩的混合，结果就是我们样例图片第二行的第二个球。这就是我们首先要用掩图把我们要留为彩色的地方设为黑色的全部理由，所以我们用 OR 来操作彩图时，彩色的像素不会跟任何其它的像素混合。

SRCINVERT

这是用来把我们源彩图中的要透明效果的地方设为黑色的 XOR 操作(如果不是黑色的话).用掩图中的黑像素与目的像素组合对目的像素不会有任何影响，而用掩图中的白色像素（我们设一个特定的背景色来生成的）与背景色的像素来组合就会把它取消掉，把它变为黑色。

这就是一点关于彩图与单色的操作把戏，把我的头想得都有点痛了，但是怎么说还是有它的意义的．．．说真的。

范例

在与章配合的 `bmp_two` 工程中的样例代码包含了可以绘出本章开始的样例图片的程序。先是用 `SRCCOPY` 参数把掩图与原彩图原样画出来，再分别用 `SRCAND` 和 `SRCPAINT` 对两个图片操作，最后把它们合成最终结果。

本范例中把背景色设为灰色以使透明效果更明显，要是用白色或是黑色作背景色的话就很难看出来是否真的起了效果。

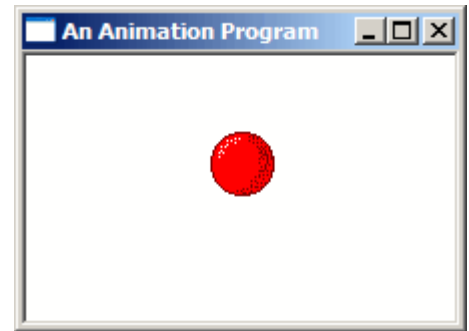
定时器与动画

范例：anim_one

设置

在我们开始动画之前，我们要设置一个可以存储在移动之间的球的位置的结构。这个结构将用来存储球现在的位置与大小，和其增量，就是我们要求其在每帧之间移动的距离。

一旦我们声明了我们的结构体类型，我们同时也要声明一个此结构体的全局实例。我们现在只有一个球，如果我们想要使一堆的球来动画的话，就要声明一个数组或是其它的什么容器（比如 C++ 中的链表）来方便地保存它们。



```
const int BALL_MOVE_DELTA = 2;
typedef struct _BALLINFO
{
    int width;
    int height;
    int x;
    int y;
    int dx;
    int dy;
}BALLINFO;
BALLINFO g_ballInfo;
```

这里我们也定义了一个常量 BALL_MOVE_DELTA，表示我们要将这个球在每次更新时移动多远的距离。我们之所以要把增量保存在一个 BALLINFO 结构体中，是因为想让移动球至上下左右四个方向的操作为独立的过程，BALL_MOVE_DELTA 只是为了叫起来方便，如果需要，我们可以在后面的程序中进行更改。

现在我们要在装入位图后对此结构体进行初始化：

```
BITMAP bm;
GetObject(g_hbmBall, sizeof(bm), &bm);
ZeroMemory(&g_ballInfo, sizeof(g_ballInfo));
g_ballInfo.width = bm.bmWidth;
g_ballInfo.height = bm.bmHeight;
g_ballInfo.dx = BALL_MOVE_DELTA;
g_ballInfo.dy = BALL_MOVE_DELTA;
```

球于左上角开始运动，根据 BALLINFO 中的 dx 和 dy 向右和下的方向移动。

设置定时器

最简单的向窗口程序添加一个定时器的方法是用 SetTimer()，但不是最好的方法，而且

不推荐在真实或是纯粹的游戏中使用，然而在我们的样例中的这样简单的动画中还是没问题的.如果要更好的请在 MSDN 查阅一下 `timeSetEvent()`，那个更精确些。

```
const int ID_TIMER = 1;
ret = SetTimer(hwnd, ID_TIMER, 50, NULL);
if(ret == 0)
    MessageBox(hwnd, "Could not SetTimer!", "Error", MB_OK |
        MB_ICONEXCLAMATION);
```

这里我们声明了一个定时器的标识以在后面引用它（终止它）并在我们的主窗口的 `WM_CREATE` 消息处理代码中来设置它.每次时间到了，它就向我们的窗口发送一个 `WM_TIMER` 消息，并把定时器标识放于 `wParam` 中.因为我们只有一个定时器所以不需要标识，在有多个定时器的时候你就需要它们来作区分。

我们把定时器的间隔设为 50 毫秒，大致上每秒钟可以数 20 帧.之所以说大致上，因为我说过，`SetTimer()`有点不精确，但是这里不是很关键的代码，这里那里的一点毫秒的误差并不是什么致命的问题。

在 `WM_TIMER` 来实现动画

现在我们在得到 `WM_TIMER` 消息的时候我们要计算出球的新位置并在更新的位置再绘制一次。

```
case WM_TIMER:
{
    RECT rcClient;
    HDC hdc = GetDC(hwnd);
    GetClientRect(hwnd, &rcClient);
    UpdateBall(&rcClient);
    DrawBall(hdc, &rcClient);
    ReleaseDC(hwnd, hdc);
}
break;
```

我把更新和绘图的代码分别放于两个单独的函数中。这样做的好处就是可以让我们在每个 `WM_TIMER` 或 `WM_PAINT` 消息中来绘制球而不会有重复的代码，注意我们在两个地方获取 `HDC` 的方法是不同的，所以最好是把这些代码留在消息处理中并把结果传到 `DrawBall()`函数中去。

```
void UpdateBall(RECT* prc)
{
    g_ballInfo.x += g_ballInfo.dx;
    g_ballInfo.y += g_ballInfo.dy;
    if(g_ballInfo.x < 0)
    {
        g_ballInfo.x = 0;
```

```

        g_ballInfo.dx = BALL_MOVE_DELTA;
    }
    else if(g_ballInfo.x + g_ballInfo.width > prc->right)
    {
        g_ballInfo.x = prc->right - g_ballInfo.width;
        g_ballInfo.dx = -BALL_MOVE_DELTA;
    }
    if(g_ballInfo.y < 0)
    {
        g_ballInfo.y = 0;
        g_ballInfo.dy = BALL_MOVE_DELTA;
    }
    else if(g_ballInfo.y + g_ballInfo.height > prc->bottom)
    {
        g_ballInfo.y = prc->bottom - g_ballInfo.height;
        g_ballInfo.dy = -BALL_MOVE_DELTA;
    }
}

```

这都是些基本的数学运算，我们把增量值加到 x 坐标来移动这个球。如果球到了客户区域外，就把它拉回来范围内并把增量值改为相反方向的值以使球在边缘”弹”回来了。

```

void DrawBall(HDC hdc, RECT* prc)
{
    HDC hdcBuffer = CreateCompatibleDC(hdc);
    HBITMAP hbmBuffer = CreateCompatibleBitmap(hdc, prc->right, prc->bottom);
    HBITMAP hbmOldBuffer = SelectObject(hdcBuffer, hbmBuffer);
    HDC hdcMem = CreateCompatibleDC(hdc);
    HBITMAP hbmOld = SelectObject(hdcMem, g_hbmMask);
    FillRect(hdcBuffer, prc, GetStockObject(WHITE_BRUSH));
    BitBlt(hdcBuffer, g_ballInfo.x, g_ballInfo.y, g_ballInfo.width,
           g_ballInfo.height, hdcMem, 0, 0, SRCAND);
    SelectObject(hdcMem, g_hbmBall);
    BitBlt(hdcBuffer, g_ballInfo.x, g_ballInfo.y, g_ballInfo.width,
           g_ballInfo.height, hdcMem, 0, 0, SRCPAINT);
    BitBlt(hdc, 0, 0, prc->right, prc->bottom, hdcBuffer, 0, 0, SRCCOPY);
    SelectObject(hdcMem, hbmOld);
    DeleteDC(hdcMem);
    SelectObject(hdcBuffer, hbmOldBuffer);
    DeleteDC(hdcBuffer);
    DeleteObject(hbmBuffer);
}

```

这里的绘图操作跟先前几个例子中的几乎是一样的，就是从 BALLINFO 结构体中取了球的位置与大小。但是还有一个重要的差异...

双缓冲

当对你的窗口的 HDC 直接做绘图操作的时候，非常有可能的是屏幕会在你完成操作前就更新了。．．．比如在你绘完掩图，还没有在顶上绘彩图，用户就会在你有机会绘彩色之前看到一个黑色背景的闪烁。你的计算机愈是慢，你的绘图操作愈多，闪烁就愈明显，很有可能看起来就像一个大的乱污点。

这是相当郁闷的，但我们可以通过先在内存中完成所有的绘图操作，再把完成的杰作用一个 `BitBlt()` 拷入屏幕，这样屏幕就是直接从旧的图像直接更新至完全新的图像，而不再留下可见的单个操作。

为了这样做，我们在内存中创建一个暂时的 `HBITMAP`，跟我们要绘向屏幕的那个区域一模一样大小。我们也需要一个 HDC 以能向这个位图使用 `BitBlt()`。

```
HDC hdcBuffer = CreateCompatibleDC(hdc);
HBITMAP hbmBuffer = CreateCompatibleBitmap(hdc, prc->right, prc->bottom);
HBITMAP hbmOldBuffer = SelectObject(hdcBuffer, hbmBuffer);
```

现在我们就有一个向内存绘图的地方了，所有的绘图操作作用 `hdcBuffer` 而不是 `hdc`(窗口的)，这样在我们完成之前所有的结果都存于内存中的位图。我们现在就可以一步到位地把整个拷向窗口。

```
BitBlt(hdc, 0, 0, prc->right, prc->bottom, hdcBuffer, 0, 0, SRCCOPY);
```

就这样，清除 HDC 和 `HBITMAP` 的操作还是如往常一样。

更快的双缓冲

这个例子中我在每个帧都创建和销毁了用于双缓冲的位图，这样做的目的是我想能够拉伸窗口的大小，所以总是创建一个新缓冲比在每次窗口的位置改变的时候都要跟踪并且要重新决定缓冲区的大小。如果创建一个全局的缓冲位图，要么不允许窗口拉伸，要么在窗口拉伸的时候只拉伸这个位图，而不是每次创建一个新的，就会有效率得多。如果你要为一个游戏或别的什么程序作优化的话，你就自己试一下。

终止定时器

当我们的窗口被销毁的时候，将我们用的所有的资源进行释放是个好主意，这里就要包括我们设置的定时器了。要停止它，我们简单地把我们创建时用的标识传入 `KillTimer()` 就行了。

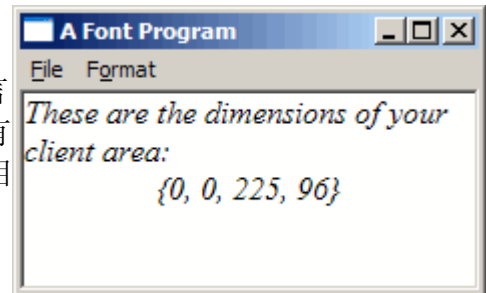
```
KillTimer(hwnd, ID_TIMER);
```

文本与字体

范例: font_one

装入字体

Win32 GDI 在处理不同的绘图类型, 风格式样, 语言和字符集的能力向来为人称道. 但是在对字体的处理上有个例外, 尤其是对初学者来说更是如此. 主要的跟字体相关的 API 函数, `CreateFont()` 有 14 个参数来指定高度, 式样, 线宽, 字体族, 和其它的一些属性.



幸好, 它并不像看起来那么难以理解, 很多这里的工作牵涉到处理一些敏感的默认参数值. 除了其中的 2 个之外, 所有的 `CreateFont()` 有参数都可以设为 0 或是 NULL, 这样系统就会用一些使你看到一般的效果的默认字体.

`CreateFont()` 创建一个 `HFONT`, 即一个内存中的逻辑字体的句柄. 这个句柄所包含的数据可以用 `GetObject()` 获取到一个 `LOGFONT` 结构体中, 就像 `HBITMAP` 可以填充 `BITMAP` 结构体一样.

`LOGFONT` 的成员跟 `CreateFont()` 的几乎一模一样, 为了方便你也可以用 `CreateFontIndirect()` 从一个已有的 `LOGFONT` 直接创建一个字体. 这样是相当简便的, 当你只有改变某些已存在的字体句柄的参数来创建一个新字体时候. 用 `GetObject()` 填充一个 `LOGFONT`, 按照你的需要来改变其中的某些成员, 再用 `CreateFontIndirect()` 创建一个新字体.

```
HFONT hf;
HDC hdc;
long lfHeight;

hdc = GetDC(NULL);
lfHeight = -MulDiv(12, GetDeviceCaps(hdc, LOGPIXELSY), 72);
ReleaseDC(NULL, hdc);
hf = CreateFont(lfHeight, 0, 0, 0, 0, TRUE, 0, 0, 0, 0, 0, 0, "Times New
                Roman");

if(hf)
{
    DeleteObject(g_hfFont);
    g_hfFont = hf;
}
else
{
    MessageBox(hwnd, "Font creation failed!", "Error", MB_OK |
                MB_ICONEXCLAMATION);
}
```

这就是用来创建样例图片中字体的代码.这是 Times New Roman 字体, 12Point 宽, 并有 *Italics* 式样.italics 的标志是 CreateFont()的第 6 个参数, 你可以看到我们将其设为了 TRUE.我们要创建字体的名字是最后一个参数.

有一点要说明的是代码中用来指定字体大小的值, CreateFont()的 lfHeight 参数.一般来说人们习惯于用線宽来指定字体的大小, 10 号, 12 号, 等等...但 CreateFont()卻不接受線宽指定大小的方法, 它接受逻辑单位, 在你的屏幕和打印机上是不同的, 甚至在打印机之间, 屏幕之间也是不同的.

这种情况的原因就是不同的设备的分辨率是相当不同的. . . 打印机可以轻易达到 600 或 1200dpi, 而屏幕达到 200 就很不错了. . . 如果在屏幕和打印机上用同样大小的字体, 你可能甚至看不清单个的字母.

我们要做的就是将所要線宽的字体转为设备上接近的逻辑大小. 本例中设备为屏幕, 所以我们获取屏幕的 HDC, 并用 GetDeviceCaps()来获取每个英寸它能显示多个像素, 再传给 MSDN 慷慨提供的 MulDiv()函数去来把我们的 12 号字体转为 CreateFont()需要的正确的逻辑大小.. 我们把这个值在 lfHeight 存起来并当第一个参数传给 CreateFont().

默认字体

当你首次用 GetDC()来获取你窗口的 HDC 时, 系统为你把默认字体选入其中, 一般来说不是很好看. 要得到一个可以接受的字体最简单的方法 (不用麻烦地调用 CreatFont()了) 就是调用 GetStockObject()来获取一个 DEFAULT_GUI_FONT.

这属于系统资源所以你想得到多少都可以, 不用担忧会引起内存泄漏, 在不需要的时候可以调用 DeleteObject()对其进行删除, 这样做的好处就是你不需要来释放一个字体的一直跟蹤它是从 CreateFont()或是 GetStockObject()获取的.

绘文本

现在我们有了一个比较 cute 的字体了, 我们怎样在屏幕上绘一点文本? 这里假设我们不用编辑框或是靜态控件.

你基本上有两个选择, TextOut()和 DrawText().TextOut()简单些, 但是选项少些并且不为你做单词換行和对齐等操作.

```
char szSize[100];
char szTitle[] = "These are the dimensions of your client area.";
HFONT hfOld = SelectObject(hdc, hf);
SetBkColor(hdc, g_rgbBackground);
SetTextColor(hdc, g_rgbText);
if(g_bOpaque)
{
    SetBkMode(hdc, OPAQUE);
}
else
```



```

{
    SetBkMode(hdc, TRANSPARENT);
}
DrawText(hdc, szTitle, -1, prc, DT_WORDBREAK);
wsprintf(szSize, "{%d, %d, %d, %d}", prc->left, prc->top, prc->right, prc->bottom);
DrawText(hdc, szSize, -1, prc, DT_SINGLELINE | DT_CENTER | DT_VCENTER);
SelectObject(hdc, hfOld);

```

我们所做的第一件事就是用 `SelectObject()` 来获取我们要在我们的 HDC 中使用的字体并为绘图作准备. 在选入下一个字体之前所有的文本操作都用这个字体.

下一步我们设置文本和背景颜色. 设置背景颜色并不会让整个背景都成这个颜色, 它只影响某些用背景颜色来绘图的操作 (绘文本是其中之一). 这也跟当前的背景模式有关. 如果为 `OPAQUE` (默认), 那么所有的文本绘制都会填入一个背景色的方框. 如果设为 `TRANSPARENT`, 那么文本绘制就不会随著背景色一起绘制, 原来在后面有什么就还是什么, 这种情况下背景色没有效果.

现在我们用 `DrawText()` 来绘制文本, 我们传入要用的 HDC 和要绘的字符串. 第三个参数是字符串的长度, 但我们传入了 -1 因为 `DrawText()` 很聪明可以自己算出文本的长度. 第四个参数我们传入了 `prc`, 指向客户 RECT 的指针. `DrawText()` 将会根据你给的其它的参数来在这个方框中进行绘制.

在第一次调用中, 我们指定了 `DT_WORDBREAK`, 默认地把文本左对齐, 并把所绘的文本自动地在方框的边缘换行. . . 非常有用.

第二次调用中, 我们只打印了一个单行而没有换行, 并且我们要它在水平和纵向的方向上都是居中的 (`DrawText()` 只会在单行绘制的时候这样做).

客户区重绘

关于范例的一个说明. . . 当 `WNDCLASS` 被注册的时候我设置了 `CS_VREDRAW` 和 `CS_HREDRAW` 式样. 这会导致如果窗口被拉伸的时候整个客户区被重绘, 默认地只是重绘改变的部分. 这样看起来, 当拉伸你的窗口时候, 居中的文本就不如你所想像的那么更新, 有点糟糕.

选择字体

一般讲, 任何牵涉到字体的程序都要使用户可以自己选字体显示的颜色和大小.

如同获取打开与保存文件名字的通用对话框样, 有一个选择字的通用对话框. 有点古怪, 叫作 `ChooseFont()` 和一个叫 `CHOOSEFONT` 的结构体一起用, 为你选择默认的开始字体, 并返回用户最终的选择结果.

```

HFONT g_hfFont = GetStockObject(DEFAULT_GUI_FONT);
COLORREF g_rgbText = RGB(0, 0, 0);
void DoSelectFont(HWND hwnd)
{

```

```

CHOOSEFONT cf = {sizeof(CHOOSEFONT)};
LOGFONT lf;
GetObject(g_hFont, sizeof(LOGFONT), &lf);
cf.Flags = CF_EFFECTS | CF_INITTOLOGFONTSTRUCT | CF_SCREENFONTS;
cf.hwndOwner = hwnd;
cf.lpLogFont = &lf;
cf.rgbColors = g_rgbText;
if(ChooseFont(&cf))
{
    HFONT hf = CreateFontIndirect(&lf);
    if(hf)
    {
        g_hFont = hf;
    }
    else
    {
        MessageBox(hwnd, "Font creation failed!", "Error", MB_OK |
                    MB_ICONEXCLAMATION);
    }
    g_rgbText = cf.rgbColors;
}
}

```

这个调用中的 `hwnd` 就是你要作为调用这个字体对话框的父窗口。

最简单使用这个对话框的方式就是和一个已有的 `LOGFONT` 结构体一起使用，一般来讲就是你当前使用的 `HFONT` 的那个。我们把这个结构体中的 `lpLogFont` 成员指向我们刚刚填充过我们当前消息的 `LOGFONT`，并加上 `CF_INITTOLOGFONTSTRUCT` 标志以使 `ChooseFont()` 知道用这个成员。`CF_EFFECTS` 告诉 `ChooseFont()` 要允许用户选择颜色和underline和删除线的属性。

更怪的是，黑体和斜体不算效果，而被看作字体的一部分，事实上有些字体只有黑体和斜体。如果你想检查或是阻止用户选择一个黑体或斜体的字体你可以在用户作出选择后分别检查 `LOGFONT` 中的 `lfWeight` 和 `lfItalic` 成员。你就可以在此提醒用户更改选择或是做出别的什么操作对成员变量进行一些更改，再去调用 `CreateFontIndirect()`。

字体的颜色没有跟 `HFONT` 关联，所以要单独存储，`CHOOSEFONT` 结构体中的 `rgbColors` 成员就可以用来传入初始的颜色，也可以用来获取新的颜色。

`CF_SCREENFONTS` 指示我们要把字体设计为在屏幕上显示，而不是在打印机上使用。有些字体都支持，有些只支持一种。根据你想要用字体的目的，还有很多用来限制用户选择字体的各种标志，可以在 MSDN 中查到。

选择颜色

为了使用户可以选择字体的颜色，或是让他为所有的东西指定一个颜色，还有一个

ChooseColor()通用对话框. 这是样例程序中用来允许用户来选择背景色的代码.

```
COLORREF g_rgbBackground = RGB(255, 255, 255);
COLORREF g_rgbCustom[16] = {0};
void DoSelectColour(HWND hwnd)
{
    CHOOSECOLOR cc = {sizeof(CHOOSECOLOR)};
    cc.Flags = CC_RGBINIT | CC_FULLOPEN | CC_ANYCOLOR;
    cc.hwndOwner = hwnd;
    cc.rgbResult = g_rgbBackground;
    cc.lpCustColors = g_rgbCustom;
    if(ChooseColor(&cc))
    {
        g_rgbBackground = cc.rgbResult;
    }
}
```

这应该相当直观, 我们这里的 `hwnd` 也是对话框的父窗口. `CC_RGBINIT` 参数说用了要用我们用 `rgbResult` 成员传入的颜色来开始, 这个成员也是我们可以在对话框关闭后获取用户选择的颜色的地方.

`g_rgbCustom` 是 16 个 `COLORREF` 的数组, 用来存储用户在对话框中决定的放入自定义格子中的值. 你可以把这些值存在一些比如注册表的位置, 不然当你程序关闭的时候它们就丢失了. 这个参数不是可选的 (要填).

控件字体

还有, 你可能想使你的窗口或是对话框上的控件来在某个时候更改字体. 这如我们在前面的例子中用 `CreateWindow()` 来创建控件一样. 如同系统默认用的窗口控件一样, 我们用 `WM_SETFONT` 来设置一个新的字体的句柄 (来自于 `GetStockObject()`) 以便在控件中用. 你也可以对你用 `CreateFont()` 创建的字体用这个方法. 把字体句柄传入 `wParam` 并把 `lParam` 设为 `TRUE` 让控件重绘就行了.

我曾在前面的例子中做过这样的操作, 但是值得在这再提一下, 因为关系紧密, 也比较短:

```
SendDlgItemMessage(hwnd, IDC_OF_YOUR_CONTROL, WM_SETFONT,
(WPARAM)hFont, TRUE);
```

`hFont` 当然是你要用的 `HFONT`, `IDC_OF_YOUR_CONTROL` 就是你要更改字体的那个控件的标识.

推荐的书与参考文档

书

如果你在学习的时候想让线上的人尊敬你，你就需要找本好书来学。这里我们提供一些学习的方向和一些必要的解释，不要把它看作图书馆的说明和手把手的讲授。

你还可以在[#Winprog Store](#)找到更多的推荐书籍与链接。

[Programming Windows](#)

Charles Petzold. 这是讲 Win32 API 的书。如果你想写只用 API 的程序（也是本教程所讲授的），你就需要这本书。

[Programming Windows with MFC](#)

Jeff Prosise. 如果你想探究一下 MFC（在完全熟悉 Win32 API 后），这本书就是为你准备的。如果你不喜欢 MFC 但是想找个 windows 开发的工作，那就一定要看看，开卷有益。

[Programming Applications for Windows](#)

Jeffrey Richter. 不是新手看的书，如果你需要跟管理进程，线程，动态链接库，Windows 内存管理，异常处理和系统 Hook 之类的事情打交道，那就需要看这本书。

[Visual C++ Windows Shell Programming](#)

Dino Esposito. 为那些对可视化和用户友好性感兴趣的人所写，这本书涵盖了为 windows shell 写插件，有效地管理文件的拖拉操作，客制化任务栏和资源管理器，加上一大堆的其它技巧。非常值得所有写 Windows 下 GUI 程序的人一读。

[Network Programming for Microsoft Windows](#)

关于网络编程的最新消息，包括 NetBIOS，邮件槽和管道，当然还有永远很重要的 Windows sockets 编程，winsock2 和底层的 raw sockets 都有。还包括 2000，CE 等不同的 windows 平台。

链接

[MSDN Online](#)

这个站点包括了所有能想得到的 Microsoft 的技术，包括完全的 Win32 API 和 MFC 文档。如果没有随你的编译器工具(比如 VC++)附送的话，这个完全免费的在线站点就向你提供所有需要的消息。如果你问一个可以在 MSDN 一下就搜到答案的问题，别人就会很轻视你。

[#winprog homepage](#)

查看 FAQ 和商店中的说明。

免费的 Visual C++ 命令行工具

如何得到它们

Microsoft 已经偷偷地随著 .NET Framework SDK 一起发布了免费的命令行编译器与链接工具。Framework SDK 包括了所有你需要的 .NET 开发的东西 <C# 编译器等等. . . > 包括命令行编译器 `cl.exe`，虽然是为了 .NET Framework 的，但其实跟标准的 VC++ 中的就是一样的。[.NET Framework SDK](#)

因为这是 .NET SDK，所以并不包含 Win32 API 开发所需的头文件和库，那包含于平台 SDK 中。瞧瞧！平台 SDK 也是免费的。其实只需要 Core SDK，但是要是想要别的组件，尽管当下来就是了。

[平台 SDK](#)

如同奖励一样，要是下载了平台 SDK 的文档（我强烈推荐）你就会拥有一全套的最新的 Win32 参考，这个比 MSDN 在线好用多了。

记得在两个 SDK 中都要选上 Register Environment Variables 这个选项，不然的话你就要自己设置一下 PATH 与其它的环境变量，这些工具才能在命令行中使用。

使用它们

因为已经有很全面的文档了，而且 MSDN 中也说得很清楚，你需要自己来 RTFM（译者注：Read The F*%king Manual）学习 VC++ 编译器与其它的工具了。为了让你上路，这里给出最简单的编译一个程序的步骤. . .

编译一个 console 应用：

```
cl foo.c
```

编译一个简单的 windows 应用，比如我们教程中的那些范例：

```
rc dlg_one.rc
```

```
cl dlg_one.c dlg_one.res user32.lib
```

免费的 Borland C++ 命令行工具

如何得到它们

对于任何想学习 windows 开发的人，Borland 向大家提供了它的免费的命令行工具。爽吧！没有好看的 IDE 或资源编辑器，但是免费的你还挑剔个什么呢，我在这里要说的是这个编译器要比其它的什么 LCC-Win32（这个连 C++ 都不支持）啊，gcc，mingw，cygwin，djgpp 之类的质量好得多。（译者注：Borland 在我翻译的时候已分离了它的开发工具部门。）

仔细的读一下 readme 文档来自己熟悉它

Borland C++ 5.5（译者注：原文这个位置是失效连接，因为这个东西不推荐使用了，感兴趣更炫的是，它还带着自己在网上找找吧）。有一个 debugger！我不用这个，所以关于它我不能帮什么忙，但是有总比没有好吧。如果你在 DOS 时代对 Turbo C++ 很熟悉了的话，这个就是你的新伙伴了。

有时不知道什么原因 IE 不能下载这个文件，所以如果点击它没有工作的话，就右键点击它保存快捷方式再用你常用的 FTP 客户端程序来下载它。

Turbo Debugger（同上，自己找吧。）

还有一点，它还有一个包含完整 Win32 API 参考的帮助文件。这个有点年头了，但是总体来说还是很精确的，也比 MSDN 在线好用，除非你要查一些 API 函数的最新变动(要是还在看我们这个教程的话，不会很需要的)。我是经常用它的。

Win32 API Reference

使用它们

基本的命令

如果你要编译一个单文件的程序(比如 simple_window.c)，那么你可以如下的命令：

```
bcc32 -tW simple_window.c
```

这个 -tW 选项指定这是一个 Win32 GUI 应用，而不是默认的 Console 应用。你可以在这个命令的尾部加上其它的文件以把多个文件编入一个单个的 .exe 程序。

链入资源

这是很多命令行用户头痛的地方，毫无疑问，Borland 就是要尽其所能把在应用程序中链入资源做得困难些，资源编辑器 brc32 不再像它之前的版本样自己把已编译的资源链入 .exe 文件。如果你不带参数运行 brc32 想得到使用方法帮助，它也是列出一个选项将 .exe 链接关闭，但是好像没有什么方法重新打开。

我试过很多的命令与选项的组合，但是还是没有找到什么方法把一个 .res 文件加到一个用上述方法生成的 .exe 文件中去。很不爽，因为我找到解决这个问题的解决方案非常复杂。

但是有一个比较简单的方法...

BC++ 现在有一个用于资源包含的替换的方法，就是在程序中用一个 #pragma(一个非标准的预处理指令，要是编译器不支持的话就直接忽略)。

```
#pragma resource "app_name.res"
```

把这句代码放入你的 main.c 或是 .cpp 文件中就会使编译器自动链入这个由你的 .rc 文件生成的 .res 文件。（.res 文件就像资源的 .obj 文件）

使用 #pragma 可以让你如上所述的简单地编译程序，但还是要先用 brc32 来编译你的 .rc 文件。如果你还是想用我在教程中的 makefile 中用的那些命令行选项，继续读下去。 . . .

困难的方式

这就是用来编译包括资源文件在内的 dlg_one 样例的命令。

```
bcc32 -c -tW dlg_one.c  
ilink32 -aa -c -x -Gn dlg_one.obj c0w32.obj,dlg_one.exe,,import32.lib cw32.lib,,dlg_one.res
```

酷吧？bcc32 的 -c 选项意味著仅仅编译，不链入一个 .exe 文件。-x -Gn 选项是取消链接器生成的一些额外的文件，你一般不需要这些东西。

这种方法的唯一的缺点就是因为我们手动的指定链接命令，我们需要包括默认的库和目标文件，而在正常情况下编译器是会为我们做这个工作的。如上，我指定了一个普通的 windows 应用大致所需要的文件。

要简化这个步骤的话，最好是写个 makefile。这里我提供了一个通用的，应该可以用于本教程中的所有样例，你也可以改动它以适应你自己的任何程序。

```
APP    = dlg_one  
EXEFILE = $(APP).exe  
OBJFILES = $(APP).obj  
RESFILES = $(APP).res  
LIBFILES =  
DEFFILE =  
.AUTODEPEND  
BCC32 = bcc32  
ILINK32 = ilink32  
BRC32 = brc32  
CFLAGS = -c -tWM- -w -w-par -w-inl -W -a1 -Od  
LFLAGS = -aa -V4.0 -c -x -Gn  
RFLAGS = -X -R  
STDOBJ = c0w32.obj  
STDLIBS = import32.lib cw32.lib  
$(EXEFILE) : $(OBJFILES) $(RESFILES)  
    $(ILINK32) $(LFLAGS) $(OBJFILES) $(STDOBJ), $(EXEFILE), \  
        $(LIBFILES) $(STDLIBS), $(DEFFILE), $(RESFILES)  
  
clean:  
    del *.obj *.res *.tds *.map
```

你只需要更改这个文件前 6 行的相关信息。

Copyright © 1998-2003, Brook Miles ([theForger](#)). All rights reserved.

常见问题的解决方案

- [Error LNK2001: unresolved external symbol _main](#)
- [Error C2440: cannot convert from 'void*' to 'HICON__ *' \(or similar\)](#)
- [Fatal error RC1015: cannot open include file 'afxres.h'](#)
- [Error LNK2001: unresolved external symbol InitCommonControls](#)
- [Dialog does not display when certain controls are added](#)

Error LNK2001: unresolved external symbol _main

一个未解析外部符号错误发生于:一个地方调用了一个外部函数,但是链接器在你所有模块和你链入的所有库中都找不到这个函数.

这种情况可能有两种原因或其中之一. 要么你想写一个 Win32 GUI 应用(或是一个非 console 的应用)但是卻将其以 console 应用来编译...或是你真正的想要编译一个 console 应用但没有正确地写或是编译 main() 函数.

一般来说第一种情况居多,如果你在 VC++中创建工程时指定了 Win32 Console 工程类型你就会得到这个错误. 用 BC++的命令行工具来编译的时候没有指定正确的选项使其以一个 Win32 GUI 应用来编译,它就会以默认的 console 应用来编译,这样也会得到这个错误.

修复方法

如果是用的 VC++就重新创建你的工程并选择 Win32 应用工程类型(不是 Console).

如果用的是 BC++命令行编译器,加上 -tW 选项指明它是一个 windows 应用.

Error C2440: cannot convert from 'void*' to 'HICON__ *' (or similar)

如果你在编译本教程中的代码,意味著你在以 C++代码在编译它. 这些代码是为 bcc32 和 VC++的 C 编译器写的,所以并不能精确地以 C++编译,因为 C++的类型转换的规则更严格些, C++需要你显示地转换.

VC++(大多数编译器也是这样)会自动将一个 .cpp 文件以 C++代码来编译, .c 的文件就以 C 代码来编译. 如果你把教程中的代码放入一个 .cpp 文件,这应该就是错误的产生原因.

如果你不是在编译这个教程中的代码,那我就不能确定是否如上述,有可能的确存在一个要解决的错误. 需要你自己来判断能不安全地做强制转换以消除这个错误, 否则的话,你就有可能把一个变量转到不能转到的地方.

修复方法

如果想用 C,就把你的 .cpp 文件改为 .c 就是了. 再就是加一个简单的强转,教程中的代码以 C++编译的话只需要这个修改.

例如,在 C 中这样可以:

```
HBITMAP hbmOldBuffer = SelectObject(hdcBuffer, hbmBuffer);
```

但在 C++ 中就需要一个强转：

```
HBITMAP hbmOldBuffer = (HBITMAP)SelectObject(hdcBuffer, hbmBuffer);
```

Fatal error RC1015: cannot open include file 'afxres.h'.

非常奇怪的是，VC++ 在资源文件中加入了 `afxres.h`，就算你没有用 MFC 的工程类型也是如此，而且这个文件只有你装了 MFC 才会有。这个郁闷的文件并不是必需的，所以要修正这个错误，可以用一个编辑器打开 `.rc` 文件并用 `winres.h` 替换掉两个 `afxres.h` 就行了。（注意应该有两个，两个都要换）。

Error LNK2001: unresolved external symbol InitCommonControls

你没有链入这个 API 被定义的 `comctl32.lib`。这个库不是默认加入的，所以要么在你的命令行中加进去，要么在 VC++ 的工程设置中的 Link 属性项中加入。

Dialog does not display when certain controls are added

ListView, TreeView, Hotkey, Progress Bar, 等控件被归类为通用控件，因为它们随著 `comctl32` 加入 windows，于 Windows95 之前并不存在。而 Button, Edit, ListBox 等等控件毫无疑问也很通用，但不是“通用控件”，一般称为“标准控件”。

如果把一个通用控件加入到一个对话框后不能显示的话，很可能是在你运行对话框之前调用 `InitCommonControls()` 失败，或许肯定是这样的。调用它的最好的地方是 `WinMain()` 的开始。在 `WM_INITDIALOG` 就太晚了，因为对话框会在这之前就创建失败，从而使它永远没有机会被调用。

有些人 and 文档可能告诉你 `InitCommonControls()` 已经过时了，你应该调用 `InitCommonControlsEx()`。如果想那样做的话也是可以的。但 `InitCommonControls()` 就是简单点而已，并没有一点错误。

为什么你应该在学习 MFC 之前学习 API 编程

争论

很多人到 IRC 上问”哪个好些，MFC 或 API？”然后有很多人说”MFC 不行”或”API 差劲”，有的是因为他们在童年的心理创伤，有的是因为别人这样说。

标准的争议话题有：

- API 太难
- MFC 太令人困惑
- API 写起来太多代码
- MFC 太臃肿
- API 编程没有助手
- MFC 设计糟糕
- API 不是面向对象的
- MFC 踢了我的狗
- API 抢了我女友

还有很多...

我的答案

我的观点（虽然不是独创的）就是，你应该用合适的工具来完成合适的工作。

首先先来对 API 和 MFC 给一个清晰的说明。API 是 Application Programming Interface 的缩写，但是在 Windows 编程的前提下，就是特指的 Windows API，是应用程序与 windows 系统打交道的最底层接口。驱动当然有更低的层数，和一套不同的调用接口，但对于绝大多数 windows 开发这不是一个要考虑的问题。MFC 是一套类库(Class Library)，是一堆已写好的 C++ 类，用来减少用 API 编程需要写的代码量。它也向应用程序引入了一种（有争议地）面向对象的框架，你可以利用这个框架也可以忽略它，一般是初学者使用了它，因为这个框架并非不是针对写一个 MP3 播放器，IRC 客户端，或是游戏的。

任何一个程序，不管它是用 MFC，Delphi，Visual Basic，Perl 或是其它的你能想到的什么语言或是 Framework 所写的，都是建构在 API 之上的。很多情况下，这种关系是隐藏的，所以你不用跟 API 直接打交道，运行时和支持的库为你做了这些工作。有些人问道，”MFC 可以做什么什么，API 可以不？”答案是 MFC 只能做 API 能做到的，因为前者建于后者之上。但是自己用 API 来写就比用已经写好的 MFC 类需要多得多的代码。

那么什么算合适的 Framework？对于初学者来讲，那些刚刚学编程的人，我坚信你应该先用 API 直到你对 Windows 应用工作的工作原理比较熟悉，理解消息循环，GDI，控件甚至还有多线程和 socket 背后的基本机制。这样你就可以理解所有 windows 应用程序的基本模块，就可以把这些常识用于 MFC，Visual Basic，或你在后面的工作中选择使用的任何 Framework。这也是很重要的一点，因为这些其它的 Framework 并不是支持所有 API 的功能，因为它们已经支持了很多了，不能够也不必要支持所有的神秘的细节，有些是大多数人并不会使用的。这样你就可以在最终要用的时候自己加上，你不能依赖 Framework 来为你完成

这些工作，如果你不理解 API 这些工作就很让你头痛。

但是，MFC 不是更简单些吗？一般看来，在它为你做的那些通用的工作上它是比较简单，也减少了你需要实际写的代码量。但是在你不理解你要写的代码时，或是不明白那些代码如何为你工作的时候，更少的代码并不意味着更简单。一般情况下，初学者用向导生成他们的应用但是不理解生成的大多数代码，就要花一堆的时间来试图弄清楚要在哪里加点代码或是改点什么来完成一个特定的任务。如果你用 API 或是 MFC 自己写你的程序，你就会明白所有的这些代码，因为就你把它们放在那里的，你就只会用你清楚的功能。

另外一个重要的因素就是大多数刚刚学 Win32 API 的人还没有一个很扎实的 C++ 基础。同时来试图理解 windows 编程和学习 C++ 很可能是一个很让你苦恼的任务。虽然不是可能，但是比起你已经熟悉了 C++ 或是 API 后来说是要花长得多的时间才有可能有所收获。

所以说起来...

结论就是你要先学习 API 编程直至你很熟悉它了，再开始来试 MFC。如果它看起来的确有意义，也可以节省你的时间，那就使用它。

然而，有很重要的一点... 如果不懂 API 又在用 MFC，还想问点什么问题的话，得到了用 API 陈述的答案（比如“用 WM_CTLCOLORSTATIC 消息提供的 HDC”），那你就傻眼了，因为你不知道怎么把 API 话题翻成 MFC，然而自己又陷入了困境，别人也烦你，说你为什么不在用 MFC 之前学点需要的知识。

我个人更喜欢用 API 编程，因为更适合我，但是如果要我写个数据库的前端或是一个有很多 ActiveX 控件的程序，我就要严肃地思考一下 MFC 了，因为它可以帮我节省很多代码，否则我就要自己重新发明一遍了。

关于资源文件的几点说明

啊!

我在把我的主要的开发环境从 Borland C++ 转向微软的 Visual C++ 后最痛恨的一件事就是 VC++ 处理资源脚本的方式。

在 BC++ 中我可以自由地控制 .rc 文件的布局和内容，而且在使用资源编辑器时，只有我在编辑器中作过修改的地方才在资源文件中被修改。令我十分沮丧的是，VC++ 的资源编辑器会完全地把你的 .rc 文件进行改写，并可能毁掉或是忽略掉你自己作的任何修改。

刚开始我是非常的郁闷，但是我基本上学会处理这种情况了，过了一段时间后就至少没有那么差劲了，因为我一般不是完全手工写我的资源文件，只是在不能编辑器完成的时候作一点小小的修改。

兼容性

我试图把这个教程中的所有样例做得能不做修改地在 VC++ 和 BC++ 中都能正确地编译。在原来的版本的教程中，我用 Borland 的命名规则来命名头文件，即 project_name.h。但是在 VC++ 中这个头文件默认总是叫作 resource.h，所以为了简便我在这个教程中作了一些调整修改，反正对 BC++ 没有什么影响的。

对于那些好奇的人，他们可以修改资源的名称，在 VC++ 中可以手动修改 .rc 文件并在两个位置替换名称，一个是 #include 的地方，另外一个包含于一个 TEXTINCLUDE 资源之中。

另外一个问题就是 VC++ 默认地需要在 .rc 文件中包含 afxres.h，而 BC++ 自动地定义了所有必要的预处理宏所以不需要这个包含。还有一件事就是 afxres.h 只存在于那些装了 MFC 的机器上，而不是所有的情况如此，就算是你只创建一个只需要总是存在的 winres.h 的 API 应用时也是这样。

因为我用 VC++ 并用它的资源编辑器，所以我的解决方案是：稍微地改了下每个自动生成的 .rc 文件以包含下面这些东西：

```
#ifndef __BORLANDC__
#include "winres.h"
#endif
```

改之前默认的通常是这样的：

```
#include "afxres.h"
```

对于使用 VC++ 的那些读者，你们可以在 IDE 中的 "View>Resource Includes" 菜单下面找到改变这里的文本的选项。一般的实际工作中是不需要这样做的，我这里只是想让这些样例同时能在 BC++ 和 VC++ 编译。

对于使用 BC++ 的读者，我对在 .rc 文件中加上这些额外由 VC++ 编辑器生成的东西说声对不起，但是它们不会影响任何事情的。

在 BC++ 下编译资源

我试了好久也没有发现一个在 BC++ 中简单的编译一个包含资源文件的程序，最后只有转向一种非优化的方式，你可以在教程包含的 makefile 中找到这种分配。你也可以参阅[免费的 Borland C++ 命令行工具](#)。

Copyright © 1998-2003, Brook Miles ([theForger](#)). All rights reserved.