

Windows 程序内部运行机制

要想熟练掌握 Windows 应用程序的开发，首先需要理解 Windows 平台下程序运行的内部机制。市面上很多介绍 Visual C++ 开发的书籍，一上来就讲解 MFC，并且只讲操作不讲原理，结果使得很多初学者看完书后感觉云山雾绕。本章将深入剖析 Windows 程序的内部运行机制，为读者扫清 VC++ 学习路途中的第一个障碍，为进一步学习 MFC 程序打下基础。

1.1 API 与 SDK

我们在编写标准 C 程序的时候，经常会调用各种库函数来辅助完成某些功能；初学者使用得最多的 C 库函数就是 `printf` 了，这些库函数是由你所使用的编译器厂商提供的。在 Windows 平台下，也有类似的函数可供调用；不同的是，这些函数是由 Windows 操作系统本身提供的。

Windows 操作系统提供了各种各样的函数，以方便我们开发 Windows 应用程序。这些函数是 Windows 操作系统提供给应用程序编程的接口（Application Programming Interface），简称为 API 函数。我们在编写 Windows 程序时所说的 API 函数，就是指系统提供的函数，所有主要的 Windows 函数都在 `Windows.h` 头文件中进行了声明。

Windows 操作系统提供了 1000 多种 API 函数，作为开发人员，要全部记住这些函数调用的语法几乎是不可能的。那么我们如何才能更好地去使用和掌握这些函数呢？微软提供的 API 函数大多是有意义的单词的组合，每个单词的首字母大写，例如 `CreateWindow`，读者从函数的名字上就可以猜到，这个函数是用来为程序创建一个窗口的。其他的，例如，`ShowWindow`（用于显示窗口），`LoadIcon`（用于加载图标），`SendMessage`（用于发送消息）等，这些函数的准确拼写与调用语法都可以在 MSDN 中查找到。

你可以把 MSDN 理解为微软向开发人员提供的一套帮助系统，其中包含大量的开发文档、技术文章和示例代码。MSDN 包含的信息非常全面，程序员不但可以利用 MSDN

来辅助开发,还可以利用 MSDN 来进行学习,从而提高自己。对于初学者来说,学会使用 MSDN 并从中汲取知识,是必须要掌握的技能。

我们在程序开发过程中,没有必要去死记硬背函数的调用语法和参数信息,只要能快速地从 MSDN 中找到所需的信息就可以了,等使用的次数多了,这些函数自然也就记住了。

我们经常听人说 Win32 SDK 开发,那么什么是 SDK 呢。SDK 的全称是 Software Development Kit,中文译为软件开发包。假如现在我们要开发呼叫中心,在购买语音卡的同时,厂商就会提供语音卡的 SDK 开发包,以方便我们对语音卡的编程操作。这个开发包通常都会包含语音卡的 API 函数库、帮助文档、使用手册、辅助工具等资源。也就是说,SDK 实际上就是开发所需资源的一个集合。现在读者应该明白 Win32 SDK 的含义了吧,即 Windows 32 位平台下的软件开发包,包括了 API 函数、帮助文档、微软提供的一些辅助开发工具。



提示:API 和 SDK 是一种广泛使用的专业术语,并没有专指某一种特定的 API 和 SDK,例如,语音卡 API、语音卡 SDK、Java API、Java SDK 等。

1.2 窗口与句柄

窗口是 Windows 应用程序中一个非常重要的元素,一个 Windows 应用程序至少要有—个窗口,称为主窗口。窗口是屏幕上的一块矩形区域,是 Windows 应用程序与用户进行交互的接口。利用窗口,可以接收用户的输入,以及显示输出。

一个应用程序窗口通常都包含标题栏、菜单栏、系统菜单、最小化框、最大化框、可调边框,有的还有滚动条。本章应用程序创建的窗口如图 1.1 所示。

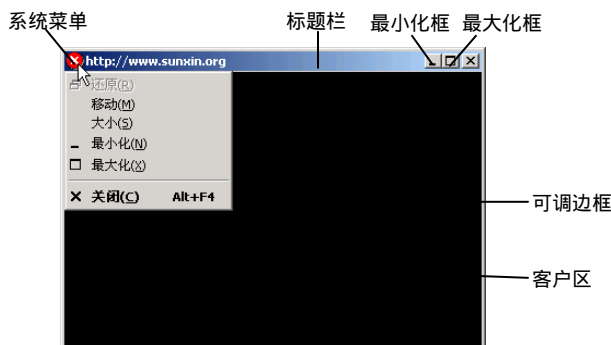


图 1.1 WS_OVERLAPPEDWINDOW 类型的窗口

窗口可以分为客户区和非客户区,如图 1.1 所示。客户区是窗口的一部分,应用程序通常在客户区中显示文字或者绘制图形。标题栏、菜单栏、系统菜单、最小化框和最大化框、可调边框统称为窗口的非客户区,它们由 Windows 系统来管理,而应用程序则主要管理客户区的外观及操作。

窗口可以有一个父窗口,有父窗口的窗口称为子窗口。除了如图 1.1 所示类型的窗口

外，对话框和消息框也是一种窗口。在对话框上通常还包含许多子窗口，这些子窗口的形式有按钮、单选按钮、复选框、组框、文本编辑框等。



提示：我们在启动 Windows 系统后，看到的桌面也是一个窗口，称为桌面窗口，它由 Windows 系统创建和管理。

在 Windows 应用程序中，窗口是通过窗口句柄（HWND）来标识的。我们要对某个窗口进行操作，首先就要得到这个窗口的句柄。句柄（HANDLE）是 Windows 程序中的一个重要的概念，使用也非常频繁。在 Windows 程序中，有各种各样的资源（窗口、图标、光标等），系统在创建这些资源时会为它们分配内存，并返回标识这些资源的标识号，即句柄。在后面的内容中我们还会看到图标句柄（HICON）、光标句柄（HCURSOR）和画刷句柄（HBRUSH）。

1.3 消息与消息队列

在传统的 C 程序中，我们调用 fopen 函数打开文件，这个库函数最终调用操作系统（提供的函数）来打开文件。而在 Windows 中，不仅用户程序可以调用系统的 API 函数，反过来，系统也会调用用户程序，这个调用是通过消息来进行的。

Windows 程序设计是一种完全不同于传统的 DOS 方式的程序设计方法。它是一种事件驱动方式的程序设计模式，主要是基于消息的。例如，当用户在窗口中画图的时候，按下鼠标左键，此时，操作系统会感知到这一事件，于是将这个事件包装成一个消息，投递到应用程序的消息队列中，然后应用程序从消息队列中取出消息并进行响应。在这个处理过程中，操作系统也会给应用程序“发送消息”。所谓“发送消息”，实际上是操作系统调用程序中一个专门负责处理消息的函数，这个函数称为窗口过程。

1. 消息

在 Windows 程序中，消息是由 MSG 结构体来表示的。MSG 结构体的定义如下（参见 MSDN）：

```
typedef struct tagMSG {  
    HWND    hwnd;  
    UINT    message;  
    WPARAM  wParam;  
    LPARAM  lParam;  
    DWORD   time;  
    POINT   pt;  
} MSG;
```

该结构体中各成员变量的含义如下：

第一个成员变量 hwnd 表示消息所属的窗口。我们通常开发的程序都是窗口应用程序，一个消息一般都是与某个窗口相关联的。例如，在某个活动窗口中按下鼠标左键，产生的按键消息就是发给该窗口的。在 Windows 程序中，用 HWND 类型的变量来标识窗口。

第二个成员变量 `message` 指定了消息的标识符。在 Windows 中，消息是由一个数值来表示的，不同的消息对应不同的数值。但是由于数值不便于记忆，所以 Windows 将消息对应的数值定义为 `WM_XXX` 宏（`WM` 是 Window Message 的缩写）的形式，`XXX` 对应某种消息的英文拼写的大写形式。例如，鼠标左键按下消息是 `WM_LBUTTONDOWN`，键盘按下消息是 `WM_KEYDOWN`，字符消息是 `WM_CHAR`，等等。在程序中我们通常都是以 `WM_XXX` 宏的形式来使用消息的。



提示：如果想知道 `WM_XXX` 消息对应的具体数值，可以在 Visual C++ 开发环境中选中 `WM_XXX`，然后单击鼠标右键，在弹出菜单中选择 `goto definition`，即可看到该宏的具体定义。跟踪或查看某个变量的定义，都可以使用这个方法。

第三、第四个成员变量 `wParam` 和 `lParam`，用于指定消息的附加信息。例如，当我们收到一个字符消息的时候，`message` 成员变量的值就是 `WM_CHAR`，但用户到底输入的是什么字符，那么就由 `wParam` 和 `lParam` 来说明。`wParam`、`lParam` 表示的信息随消息的不同而不同。如果想知道这两个成员变量具体表示的信息，可以在 MSDN 中关于某个具体消息的说明文档查看到。读者可以在 VC++ 的开发环境中通过 `goto definition` 查看一下 `WPARAM` 和 `LPARAM` 这两种类型的定义，可以发现这两种类型实际上就是 `unsigned int` 和 `long`。

最后两个变量分别表示消息投递到消息队列中的时间和鼠标的当前位置。

2. 消息队列

每一个 Windows 应用程序开始执行后，系统都会为该程序创建一个消息队列，这个消息队列用来存放该程序创建的窗口的消息。例如，当我们按下鼠标左键的时候，将会产生 `WM_LBUTTONDOWN` 消息，系统会将这个消息放到窗口所属的应用程序的消息队列中，等待应用程序的处理。Windows 将产生的消息依次放到消息队列中，而应用程序则通过一个消息循环不断地从消息队列中取出消息，并进行响应。这种消息机制，就是 Windows 程序运行的机制。关于消息队列和消息响应，在后面我们还会详细讲述。

3. 进队消息和不进队消息

Windows 程序中的消息可以分为“进队消息”和“不进队消息”。进队的消息将由系统放入到应用程序的消息队列中，然后由应用程序取出并发送。不进队的消息在系统调用窗口过程时直接发送给窗口。不管是进队消息还是不进队消息，最终都由系统调用窗口过程函数对消息进行处理。

1.4 WinMain 函数

接触过 Windows 编程方法的读者都知道，在应用程序中有一个重要的函数 `WinMain`，这个函数是应用程序的基础。当 Windows 操作系统启动一个程序时，它调用的就是该程序的 `WinMain` 函数（实际是由插入到可执行文件中的启动代码调用的）。`WinMain` 是 Windows

程序的入口点函数，与 DOS 程序的入口点函数 main 的作用相同，当 WinMain 函数结束或返回时，Windows 应用程序结束。

下面，让我们来看一个完整的 Win32 程序，该程序实现的功能是创建一个窗口，并在该窗口中响应键盘及鼠标消息，程序实现的步骤为：

- ❶ WinMain 函数的定义；
- ❷ 创建一个窗口；
- ❸ 进行消息循环；
- ❹ 编写窗口过程函数。

1.4.1 WinMain 函数的定义

WinMain 函数的原型声明如下：

```
int WINAPI WinMain(  
    HINSTANCE hInstance,           // handle to current instance  
    HINSTANCE hPrevInstance,       // handle to previous instance  
    LPSTR lpCmdLine,               // command line  
    int nCmdShow                   // show state  
);
```

WinMain 函数接收 4 个参数，这些参数都是在系统调用 WinMain 函数时，传递给应用程序的。

第一个参数 hInstance 表示该程序当前运行的实例的句柄，这是一个数值。当程序在 Windows 下运行时，它唯一标识运行中的实例（注意，只有运行中的程序实例，才有实例句柄）。一个应用程序可以运行多个实例，每运行一个实例，系统都会给该实例分配一个句柄值，并通过 hInstance 参数传递给 WinMain 函数。

第二个参数 hPrevInstance 表示当前实例的前一个实例的句柄。通过查看 MSDN 我们可以知道，在 Win32 环境下，这个参数总是 NULL，即在 Win32 环境下，这个参数不再起作用。

第三个参数 lpCmdLine 是一个以空终止的字符串，指定传递给应用程序的命令行参数。例如：在 D 盘下有一个 sunxin.txt 文件，当我们用鼠标双击这个文件时将启动记事本程序（notepad.exe），此时系统会将 D:\sunxin.txt 作为命令行参数传递给记事本程序的 WinMain 函数，记事本程序在得到这个文件的全路径名后，就在窗口中显示该文件的内容。要在 VC++ 开发环境中向应用程序传递参数，可以单击菜单【Project】 【Settings】，选择“Debug”选项卡，在“Program arguments”编辑框中输入你想传递给应用程序的参数。

第四个参数 nCmdShow 指定程序的窗口应该如何显示，例如最大化、最小化、隐藏等。这个参数的值由该程序的调用者所指定，应用程序通常不需要去理会这个参数的值。

关于 WinMain 函数前的修饰符 WINAPI，请参看下面关于__stdcall 的介绍。读者可以利用 goto definition 功能查看 WINAPI 的定义，可以看到 WINAPI 其实就是__stdcall。

1.4.2 窗口的创建

创建一个完整的窗口，需要经过下面几个操作步骤：

- ❶ 设计一个窗口类；
- ❷ 注册窗口类；
- ❸ 创建窗口；
- ❹ 显示及更新窗口。

下面的四个小节将分别介绍创建窗口的过程。完整的例程请参见光盘中的例子代码 Chapter1 目录下 WinMain。

1. 设计一个窗口类

一个完整的窗口具有许多特征，包括光标（鼠标进入该窗口时的形状）、图标、背景色等。窗口的创建过程类似于汽车的制造过程。我们在生产一个型号的汽车之前，首先要对该型号的汽车进行设计，在图纸上画出汽车的结构图，设计各个零部件，同时还要给该型号的汽车取一个响亮的名字，例如“奥迪 A6”。在完成设计后，就可以按照“奥迪 A6”这个型号生产汽车了。

类似地，在创建一个窗口前，也必须对该类型的窗口进行设计，指定窗口的特征。当然，在我们设计一个窗口时，不像汽车的设计这么复杂，因为 Windows 已经为我们定义好了一个窗口所应具有的基本属性，我们只需要像考试时做填空题一样，将需要我们填充的部分填写完整，一种窗口就设计好了。在 Windows 中，要达到作填空题的效果，只能通过结构体来完成，窗口的特征就是由 WNDCLASS 结构体来定义的。WNDCLASS 结构体的定义如下（请读者自行参看 MSDN）：

```
typedef struct _WNDCLASS {
    UINT          style;
    WNDPROC       lpfnWndProc;
    int           cbClsExtra;
    int           cbWndExtra;
    HANDLE        hInstance;
    HICON         hIcon;
    HCURSOR       hCursor;
    HBRUSH        hbrBackground;
    LPCTSTR       lpszMenuName;
    LPCTSTR       lpszClassName;
} WNDCLASS;
```

下面对该结构体的成员变量做一个说明。

第一个成员变量 style 指定这一类型窗口的样式，常用的样式如下：

■ CS_HREDRAW

当窗口水平方向上的宽度发生变化时，将重新绘制整个窗口。当窗口发生重绘时，窗口中的文字和图形将被擦除。如果没有指定这一样式，那么在水平方向上调整窗口宽度时，将不会重绘窗口。

■ CS_VREDRAW

当窗口垂直方向上的高度发生变化时,将重新绘制整个窗口。如果没有指定这一样式,那么在垂直方向上调整窗口高度时,将不会重绘窗口。


■ CS_NOCLOSE

禁用系统菜单的 Close 命令,这将导致窗口没有关闭按钮。

■ CS_DBLCLKS

当用户在窗口中双击鼠标时,向窗口过程发送鼠标双击消息。

style 成员的其他取值请参阅 MSDN。

 **知识点** 在 Windows.h 中,以 CS_开头的类样式(Class Style)标识符被定义为 16 位的常量,这些常量都只有某 1 位为 1。在 VC++ 开发环境中,利用 goto definition 功能,可以看到 CS_VREDRAW=0x0001, CS_HREDRAW=0x0002, CS_DBLCLKS =0x0008, CS_NOCLOSE=0x0200,读者可以将这些 16 进制数转换为 2 进制数,就可以发现它们都只有 1 位为 1,并且为 1 的位各不相同。用这种方式定义的标识符称为“位标志”,我们可以使用位运算操作符来组合使用这些样式。例如,要让窗口在水平和垂直尺寸发生变化时发生重绘,我们可以使用位或(|)操作符将 CS_HREDRAW 和 CS_VREDRAW 组合起来,如 style=CS_HREDRAW | CS_VREDRAW。假如有一个变量具有多个样式,而我们并不清楚该变量都有哪些样式,现在我们想要去掉该变量具有的某个样式,那么可以先对该样式标识符进行取反(~)操作,然后再和这个变量进行与(&)操作即可实现。例如,要去掉先前的 style 变量所具有的 CS_VREDRAW 样式,可以编写代码: style=style & ~ CS_VREDRAW。

在 Windows 程序中,经常会用到这种位标志标识符,后面我们在创建窗口时用到的窗口样式,也是属于位标志标识符。

第二个成员变量 lpfnWndProc 是一个函数指针,指向窗口过程函数,窗口过程函数是一个回调函数。回调函数不是由该函数的实现方直接调用,而是在特定的事件或条件发生时由另外一方调用的,用于对该事件或条件进行响应。回调函数实现的机制是:

- (1) 定义一个回调函数。
- (2) 提供函数实现的一方在初始化的时候,将回调函数的函数指针注册给调用者。
- (3) 当特定的事件或条件发生的时候,调用者使用函数指针调用回调函数对事件进行处理。

针对 Windows 的消息处理机制,窗口过程函数被调用的过程如下:

- (1) 在设计窗口类的时候,将窗口过程函数的地址赋值给 lpfnWndProc 成员变量。
- (2) 调用 RegisterClass(&wndclass)注册窗口类,那么系统就有了我们所编写的窗口过程函数的地址。
- (3) 当应用程序接收到某一窗口的消息时,调用 DispatchMessage(&msg)将消息回传给系统。系统则利用先前注册窗口类时得到的函数指针,调用窗口过程函数对消息进行处理。

一个 Windows 程序可以包含多个窗口过程函数,一个窗口过程总是与某一个特定的窗

口类相关联 (通过 WNDCLASS 结构体中的 lpfnWndProc 成员变量指定), 基于该窗口类创建的窗口使用同一个窗口过程。

lpfnWndProc 成员变量的类型是 WNDPROC, 我们在 VC++ 开发环境中使用 goto definition 功能, 可以看到 WNDPROC 的定义:

```
typedef LRESULT (CALLBACK* WNDPROC)(HWND, UINT, WPARAM, LPARAM);
```

在这里又出现了两个新的数据类型 LRESULT 和 CALLBACK, 再次使用 goto definition, 可以看到它们实际上是 long 和 __stdcall。

从 WNDPROC 的定义可以知道, WNDPROC 实际上是函数指针类型。



注意: WNDPROC 被定义为指向窗口过程函数的指针类型, 窗口过程函数的格式必须与 WNDPROC 相同。



知识点 在函数调用过程中, 会使用栈。__stdcall 与 __cdecl 是两种不同的函数调用约定, 定义了函数参数入栈的顺序, 由调用函数还是被调用函数将参数弹出栈, 以及产生函数修饰名的方法。关于这两个调用约定的详细信息, 读者可参看 MSDN。对于参数个数可变的函数, 例如 printf, 使用的是 __cdecl 调用约定, Win32 的 API 函数都遵循 __stdcall 调用约定。在 VC++ 开发环境中, 默认的编译选项是 __cdecl, 对于那些需要 __stdcall 调用约定的函数, 在声明时必须显式地加上 __stdcall。在 Windows 程序中, 回调函数必须遵循 __stdcall 调用约定, 所以我们在声明回调函数时要使用 CALLBACK。使用 CALLBACK 而不是 __stdcall 的原因是为了告诉我们这是一个回调函数。注意, 在 Windows 98 和 Windows 2000 下, 声明窗口过程函数时, 即使不使用 CALLBACK 也不会出错, 但在 Windows NT4.0 下, 则会出错。

WNDCLASS 结构体第三个成员变量 cbClsExtra: Windows 为系统中的每一个窗口类管理一个 WNDCLASS 结构。在应用程序注册一个窗口类时, 它可以让 Windows 系统为 WNDCLASS 结构分配和追加一定字节数的附加内存空间, 这部分内存空间称为类附加内存, 由属于这种窗口类的所有窗口所共享, 类附加内存空间用于存储类的附加信息。Windows 系统把这部分内存初始化为 0。一般我们将这个参数设置为 0。

第四个成员变量 cbWndExtra: Windows 系统为每一个窗口管理一个内部数据结构, 在注册一个窗口类时, 应用程序能够指定一定字节数的附加内存空间, 称为窗口附加内存。在创建这类窗口时, Windows 系统就为窗口的结构分配和追加指定数目的窗口附加内存空间, 应用程序可用这部分内存存储窗口特有的数据。Windows 系统把这部分内存初始化为 0。如果应用程序用 WNDCLASS 结构注册对话框 (用资源文件中的 CLASS 伪指令创建), 必须给 DLGWINDOWEXTRA 设置这个成员。一般我们将这个参数设置为 0。

第五个成员变量 hInstance 指定包含窗口过程的程序的实例句柄。

第六个成员变量 hIcon 指定窗口类的图标句柄。这个成员变量必须是一个图标资源的句柄, 如果这个成员为 NULL, 那么系统将提供一个默认的图标。


在为 hIcon 变量赋值时, 可以调用 LoadIcon 函数来加载一个图标资源, 返回系统分配

给该图标的句柄。该函数的原型声明如下所示：

```
HICON LoadIcon( HINSTANCE hInstance, LPCTSTR lpIconName)
```

LoadIcon 函数不仅可以加载 Windows 系统提供的标准图标到内存中，还可以加载由用户自己制作的图标资源到内存中，并返回系统分配给该图标的句柄，请参看 MSDN 关于 LoadIcon 的解释。但要注意的是，如果加载的是系统的标准图标，那么第一个参数必须为 NULL。

LoadIcon 的第二个参数是 LPCTSTR 类型，利用 goto definition 命令将会发现它实际被定义成 CONST CHAR*，即指向字符常量的指针，而图标的 ID 是一个整数。对于这种情况我们需要用 MAKEINTRESOURCE 宏把资源 ID 标识符转换为需要的 LPCTSTR 类型。

 **知识点** 在 VC++ 中，对于自定义的菜单、图标、光标、对话框等资源，都保存在资源脚本（通常扩展名为 .rc）文件中。在 VC++ 开发环境中，要访问资源文件，可以单击左边项目视图窗口底部的 ResourceView 选项卡，你将看到以树状列表形式显示的资源项目。在任何一种资源上双击鼠标左键，将打开资源编辑器。在资源编辑器中，你可以以“所见即所得”的方式对资源进行编辑。资源文件本身是文本文件格式，如果你了解资源文件的编写格式，也可以直接使用文本编辑器对资源进行编辑。

在 VC++ 中，资源是通过标识符（ID）来标识的，同一个 ID 可以标识多个不同的资源。资源的 ID 实质上是一个整数，在“resource.h”中定义为一个宏。我们在为资源指定 ID 的时候，应该养成一个良好的习惯，即在“ID”后附加特定资源英文名称的首字母，例如，菜单资源为 IDM_XXX（M 表示 Menu），图标资源为 IDI_XXX（I 表示 Icon），按钮资源为 IDB_XXX（B 表示 Button）。采用这种命名方式，我们在程序中使用资源 ID 时，可以一目了然。

WNDCLASS 结构体第七个成员变量 hCursor 指定窗口类的光标句柄。这个成员变量必须是一个光标资源的句柄，如果这个成员为 NULL，那么无论何时鼠标进入到应用程序窗口中，应用程序都必须明确地设置光标的形状。

在为 hCursor 变量赋值时，可以调用 LoadCursor 函数来加载一个光标资源，返回系统分配给该光标句柄。该函数的原型声明如下所示：

```
HCURSOR LoadCursor(HINSTANCE hInstance, LPCTSTR lpCursorName);
```

LoadCursor 函数除了加载的是光标外，其使用方法与 LoadIcon 函数一样。

第八个成员变量 hbrBackground 指定窗口类的背景画刷句柄。当窗口发生重绘时，系统使用这里指定的画刷来擦除窗口的背景。我们既可以为 hbrBackground 成员指定一个画刷的句柄，也可以为其指定一个标准的系统颜色值。关于 hbrBackground 成员的详细说明，请参看 MSDN。

我们可以调用 GetStockObject 函数来得到系统的标准画刷。GetStockObject 函数的原型声明如下所示：

```
HGDIOBJ GetStockObject( int fnObject);
```

参数 fnObject 指定要获取的对象的类型，关于该参数的取值，请参看 MSDN。

GetStockObject 函数不仅可以用于获取画刷的句柄，还可以用于获取画笔、字体和调色板的句柄。由于 GetStockObject 函数可以返回多种资源对象的句柄，在实际调用该函数前无法确定它返回哪一种资源对象的句柄，因此它的返回值的类型定义为 HGDIOBJ，在实际使用时，需要进行类型转换。例如，我们要为 hbrBackground 成员指定一个黑色画刷的句柄，可以调用如下：

```
wndclass.hbrBackground=(HBRUSH)GetStockObject(BLACK_BRUSH);
```

当窗口发生重绘时，系统会使用这里指定的黑色画刷擦除窗口的背景。

第九个成员变量 lpszMenuName 是一个以空终止的字符串，指定菜单资源的名字。如果你使用菜单资源的 ID 号，那么需要用 MAKEINTRESOURCE 宏来进行转换。如果将 lpszMenuName 成员设置为 NULL，那么基于这个窗口类创建的窗口将没有默认的菜单。要注意，菜单并不是一个窗口，很多初学者都误以为菜单是一个窗口。

第十个成员变量 lpszClassName 是一个以空终止的字符串，指定窗口类的名字。这和汽车的设计类似，设计一款新型号的汽车，需要给该型号的汽车取一个名字。同样的，设计了一种新类型的窗口，也要为该类型的窗口取个名字，这里我们将这种类型窗口的命名为“sunxin2006”，后面将看到如何使用这个名称。

2. 注册窗口类

在设计完汽车后，需要报经国家有关部门审批，批准后才能生产这种类型的汽车。同样地，设计完窗口类（WNDCLASS）后，需要调用 RegisterClass 函数对其进行注册，注册成功后，才可以创建该类型的窗口。注册函数的原型声明如下：

```
ATOM RegisterClass(CONST WNDCLASS *lpWndClass);
```

该函数只有一个参数，即上一步骤中所设计的窗口类对象的指针。

3. 创建窗口——步骤 3

设计好窗口类并且将其成功注册之后，就可以用 CreateWindow 函数产生这种类型的窗口了。CreateWindow 函数的原型声明如下：

```
HWND CreateWindow(  
    LPCTSTR lpClassName,    // pointer to registered class name  
    LPCTSTR lpWindowName,   // pointer to window name  
    DWORD dwStyle,          // window style  
    int x,                  // horizontal position of window  
    int y,                  // vertical position of window  
    int nWidth,             // window width  
    int nHeight,            // window height  
    HWND hWndParent,        // handle to parent or owner window  
    HMENU hMenu,            // handle to menu or child-window identifier  
    HANDLE hInstance,       // handle to application instance  
    LPVOID lpParam           // pointer to window-creation data  
);
```

参数 lpClassName 指定窗口类的名称，即我们在步骤 1 设计一个窗口类中为 WNDCLASS

的 `lpzClassName` 成员指定的名称，在这里应该设置为 “sunxin2006”，表示要产生 “sunxin2006” 这一类型的窗口。产生窗口的过程是由操作系统完成的，如果在调用 `CreateWindow` 函数之前，没有用 `RegisterClass` 函数注册过名称为 “sunxin2006” 的窗口类型，操作系统将无法得知这一类型窗口的相关信息，从而导致创建窗口失败。

参数 `lpWindowName` 指定窗口的名字。如果窗口样式指定了标题栏，那么这里指定的窗口名字将显示在标题栏上。

参数 `dwStyle` 指定创建的窗口的样式。就好像同一型号的汽车可以有不同的颜色一样，同一型号的窗口也可以有不同的外观样式。要注意区分 `WNDCLASS` 中的 `style` 成员与 `CreateWindow` 函数的 `dwStyle` 参数，前者是指定窗口类的样式，基于该窗口类创建的窗口都具有这些样式，后者是指定某个具体的窗口的样式。

在这里，我们可以给创建的窗口指定 `WS_OVERLAPPEDWINDOW` 这一类型，该类型的定义为：

```
#define WS_OVERLAPPEDWINDOW (WS_OVERLAPPED | \
                             WS_CAPTION | \
                             WS_SYSMENU | \
                             WS_THICKFRAME | \
                             WS_MINIMIZEBOX | \
                             WS_MAXIMIZEBOX)
```

可以看到，`WS_OVERLAPPEDWINDOW` 是多种窗口类型的组合，其原理和前面知识点所讲的内容是一致的。下面是这几种常用窗口类型的说明。

- `WS_OVERLAPPED`：产生一个层叠的窗口，一个层叠的窗口有一个标题栏和一个边框。
- `WS_CAPTION`：创建一个有标题栏的窗口。
- `WS_SYSMENU`：创建一个在标题栏上带有系统菜单的窗口，要和 `WS_CAPTION` 类型一起使用。
- `WS_THICKFRAME`：创建一个具有可调边框的窗口。
- `WS_MINIMIZEBOX`：创建一个具有最小化按钮的窗口，必须同时设定 `WS_SYSMENU` 类型。
- `WS_MAXIMIZEBOX`：创建一个具有最大化按钮的窗口，必须同时设定 `WS_SYSMENU` 类型。

使用 `WS_OVERLAPPEDWINDOW` 类型的窗口如图 1.1 所示。

`CreateWindow` 函数的参数 `x`, `y`, `nWidth`, `nHeight` 分别指定窗口左上角的 `x`, `y` 坐标，窗口的宽度，高度。如果参数 `x` 被设为 `CW_USEDEFAULT`，那么系统为窗口选择默认的左上角坐标并忽略 `y` 参数。如果参数 `nWidth` 被设为 `CW_USEDEFAULT`，那么系统为窗口选择默认的宽度和高度，参数 `nHeight` 被忽略。

参数 `hWndParent` 指定被创建窗口的父窗口句柄。在 1.2 节中已经介绍了，窗口之间可以有父子关系，子窗口必须具有 `WS_CHILD` 样式。对父窗口的操作同时也会影响到子窗口，表 1.1 列出了对父窗口的操作如何影响子窗口。

表 1.1 对父窗口的操作对子窗口的影响

父 窗 口	子 窗 口
销毁	在父窗口被销毁之前销毁
隐藏	在父窗口被隐藏之前隐藏，子窗口只有在父窗口可见时可见
移动	跟随父窗口客户区一起移动
显示	在父窗口显示之后显示

参数 hMenu 指定窗口菜单的句柄。

参数 hInstance 指定窗口所属的应用程序实例的句柄。

参数 lpParam：作为 WM_CREATE 消息的附加参数 lParam 传入的数据指针。在创建多文档界面的客户窗口时，lpParam 必须指向 CLIENTCREATESTRUCT 结构体。多数窗口将这个参数设置为 NULL。

如果窗口创建成功，CreateWindow 函数将返回系统为该窗口分配的句柄，否则，返回 NULL。注意，在创建窗口之前应先定义一个窗口句柄变量来接收创建窗口之后返回的句柄值。

4．显示及更新窗口

(1) 显示窗口

窗口创建之后，我们要让它显示出来，这就跟汽车生产出来后要推向市场一样。调用函数 ShowWindow 来显示窗口，该函数的原型声明如下所示：

```
BOOL ShowWindow(  
    HWND hWnd,      // handle to window  
    int nCmdShow     // show state  
);
```

ShowWindow 函数有两个参数，第一个参数 hWnd 就是在上一步骤中成功创建窗口后返回的那个窗口句柄；第二个参数 nCmdShow 指定了窗口显示的状态，常用的有以下几种。

- SW_HIDE：隐藏窗口并激活其他窗口。
- SW_SHOW：在窗口原来的位置以原来的尺寸激活和显示窗口。
- SW_SHOWMAXIMIZED：激活窗口并将其最大化显示。
- SW_SHOWMINIMIZED：激活窗口并将其最小化显示。
- SW_SHOWNORMAL：激活并显示窗口。如果窗口是最小化或最大化的状态，系统将其恢复到原来的尺寸和大小。应用程序在第一次显示窗口的时候应该指定此标志。

关于 nCmdShow 参数的详细内容请参见 MSDN。

(2) 更新窗口

在调用 ShowWindow 函数之后，我们紧接着调用 UpdateWindow 来刷新窗口，就好像我们买了新房子，需要装修一下。UpdateWindow 函数的原型声明如下：

```
BOOL UpdateWindow(  
    HWND hWnd      // handle to window
```

```
);
```

其参数 `hWnd` 指的是创建成功后的窗口的句柄。`UpdateWindow` 函数通过发送一个 `WM_PAINT` 消息来刷新窗口，`UpdateWindow` 将 `WM_PAINT` 消息直接发送给了窗口过程函数进行处理，而没有放到我们前面所说的消息队列里，请读者注意这一点。关于 `WM_PAINT` 消息的作用和窗口过程函数，后面我们将会详细讲解。

到此，一个窗口就算创建完成了。

1.4.3 消息循环

在创建窗口、显示窗口、更新窗口后，我们需要编写一个消息循环，不断地从消息队列中取出消息，并进行响应。要从消息队列中取出消息，我们需要调用 `GetMessage()` 函数，该函数的原型声明如下：

```
BOOL GetMessage(  
    LPMSG lpMsg,           // address of structure with message  
    HWND hWnd,             // handle of window  
    UINT wMsgFilterMin,     // first message  
    UINT wMsgFilterMax     // last message  
);
```

参数 `lpMsg` 指向一个消息（MSG）结构体，`GetMessage` 从线程的消息队列中取出的消息信息将保存在该结构体对象中。

参数 `hWnd` 指定接收属于哪一个窗口的消息。通常我们将其设置为 `NULL`，用于接收属于调用线程的所有窗口的窗口消息。

参数 `wMsgFilterMin` 指定要获取的消息的最小值，通常设置为 0。

参数 `wMsgFilterMax` 指定要获取的消息的最大值。如果 `wMsgFilterMin` 和 `wMsgFilterMax` 都设置为 0，则接收所有消息。

`GetMessage` 函数接收到除 `WM_QUIT` 外的消息均返回非零值。对于 `WM_QUIT` 消息，该函数返回零。如果出现了错误，该函数返回 -1，例如，当参数 `hWnd` 是无效的窗口句柄或 `lpMsg` 是无效的指针时。

通常我们编写的消息循环代码如下：

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

前面已经介绍了，`GetMessage` 函数只有在接收到 `WM_QUIT` 消息时，才返回 0。此时 `while` 语句判断的条件为假，循环退出，程序才有可能结束运行。在没有接收到 `WM_QUIT` 消息时，Windows 应用程序就通过这个 `while` 循环来保证程序始终处于运行状态。

`TranslateMessage` 函数用于将虚拟键消息转换为字符消息。字符消息被投递到调用线

程的消息队列中，当下一次调用 `GetMessage` 函数时被取出。当我们敲击键盘上的某个字符键时，系统将产生 `WM_KEYDOWN` 和 `WM_KEYUP` 消息。这两个消息的附加参数（`wParam` 和 `lParam`）包含的是虚拟键代码和扫描码等信息，而我们在程序中往往需要得到某个字符的 ASCII 码，`TranslateMessage` 这个函数就可以将 `WM_KEYDOWN` 和 `WM_KEYUP` 消息的组合转换为一条 `WM_CHAR` 消息（该消息的 `wParam` 附加参数包含了字符的 ASCII 码），并将转换后的新消息投递到调用线程的消息队列中。注意，`TranslateMessage` 函数并不会修改原有的消息，它只是产生新的消息并投递到消息队列中。

`DispatchMessage` 函数分派一个消息到窗口过程，由窗口过程函数对消息进行处理。`DispatchMessage` 实际上是将消息回传给操作系统，由操作系统调用窗口过程函数对消息进行处理（响应）。

Windows 应用程序的消息处理机制如图 1.2 所示。

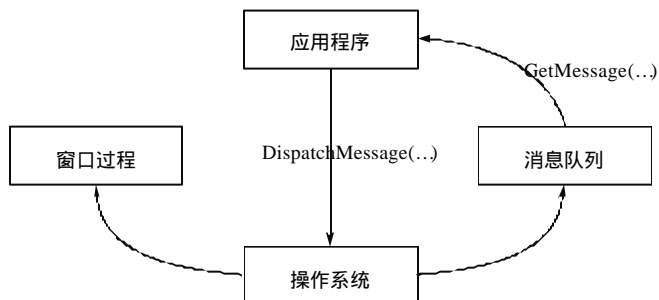


图 1.2 Windows 应用程序的消息处理机制

（1）操作系统接收到应用程序的窗口消息，将消息投递到该应用程序的消息队列中。
 （2）应用程序在消息循环中调用 `GetMessage` 函数从消息队列中取出一条一条的消息。取出消息后，应用程序可以对消息进行一些预处理，例如，放弃对某些消息的响应，或者调用 `TranslateMessage` 产生新的消息。

（3）应用程序调用 `DispatchMessage`，将消息回传给操作系统。消息是由 `MSG` 结构体对象来表示的，其中就包含了接收消息的窗口的句柄。因此，`DispatchMessage` 函数总能进行正确的传递。

（4）系统利用 `WNDCLASS` 结构体的 `lpfnWndProc` 成员保存的窗口过程函数的指针调用窗口过程，对消息进行处理（即“系统给应用程序发送了消息”）。

以上就是 Windows 应用程序的消息处理过程。



提示：

（1）从消息队列中获取消息还可以调用 `PeekMessage` 函数，该函数的原型声明如下所示：

```

BOOL PeekMessage(
    LPMSG lpMsg,           // message information
    HWND hWnd,            // handle to window
    UINT wMsgFilterMin,    // first message
    UINT wMsgFilterMax,    // last message

```

```
UINT wRemoveMsg        // removal options
);
```

前 4 个参数和 GetMessage 函数的 4 个参数的作用相同。最后 1 个参数指定消息获取的方式，如果设为 PM_NOREMOVE，那么消息将不会从消息队列中被移除；如果设为 PM_REMOVE，那么消息将从消息队列中被移除（与 GetMessage 函数的行为一致）。关于 PeekMessage 函数的更多信息，请参见 MSDN。

（2）发送消息可以使用 SendMessage 和 PostMessage 函数。SendMessage 将消息直接发送给窗口，并调用该窗口的窗口过程进行处理。在窗口过程对消息处理完毕后，该函数才返回（SendMessage 发送的消息为不进队消息）。PostMessage 函数将消息放入与创建窗口的线程相关联的消息队列后立即返回。除了这两个函数外，还有一个 PostThreadMessage 函数，用于向线程发送消息，对于线程消息，MSG 结构体中的 hwnd 成员为 NULL。

关于线程，后面我们会有专门的章节进行介绍。

1.4.4 编写窗口过程函数

在完成上述步骤后，剩下的工作就是编写一个窗口过程函数，用于处理发送给窗口的消息。一个 Windows 应用程序的主要代码部分就集中在窗口过程函数中。在 MSDN 中可以查到窗口过程函数的声明形式，如下所示：

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,           // message identifier
    WPARAM wParam,       // first message parameter
    LPARAM lParam        // second message parameter
);
```

窗口过程函数的名字可以随便取，如 WinSunProc，但函数定义的形式必须和上述声明的形式相同。



提示：系统通过窗口过程函数的地址（指针）来调用窗口过程函数，而不是名字。

WindowProc 函数的 4 个参数分别对应消息的窗口句柄、消息代码、消息代码的两个附加参数。一个程序可以有多个窗口，窗口过程函数的第 1 个参数 hwnd 就标识了接收消息的特定窗口。

在窗口过程函数内部使用 switch/case 语句来确定窗口过程接收的是什么消息，以及如何对这个消息进行处理。我们看下面的代码：

WinMain.cpp

```
.....
1. LRESULT CALLBACK WinSunProc(
```

```
2.  HWND hwnd,           // handle to window
3.  UINT uMsg,           // message identifier
4.  WPARAM wParam,       // first message parameter
5.  LPARAM lParam        // second message parameter
6. )
7. {
8.     switch(uMsg)
9.     {
10.        case WM_CHAR:
11.            char szChar[20];
12.            sprintf(szChar,"char code is %d",wParam);
13.            MessageBox(hwnd,szChar,"char",0);
14.            break;
15.        case WM_LBUTTONDOWN:
16.            MessageBox(hwnd,"mouse clicked","message",0);
17.            HDC hdc;
18.            hdc=GetDC(hwnd);
19.            TextOut(hdc,0,50,"程序员之家",strlen("程序员之家"));
20.            ReleaseDC(hwnd,hdc);
21.            break;
22.        case WM_PAINT:
23.            HDC hdc;
24.            PAINTSTRUCT ps;
25.            hdc=BeginPaint(hwnd,&ps);
26.            TextOut(hdc,0,0,"http://www.sunxin.org",strlen("http://www.sunx
in.org"));
27.            EndPaint(hwnd,&ps);
28.            break;
29.        case WM_CLOSE:
30.            if (IDYES==MessageBox(hwnd,"是否真的结束?", "message",MB_YESNO))
31.            {
32.                DestroyWindow(hwnd);
33.            }
34.            break;
35.        case WM_DESTROY:
36.            PostQuitMessage(0);
37.            break;
38.        default:
39.            return DefWindowProc(hwnd,uMsg,wParam,lParam);
40.    }
41.    return 0;
42. }
```

10~14 行代码：当用户在窗口中按下一个字符键，程序将得到一条 WM_CHAR 消息（通过调用 TranslateMessage 函数转换得到），在其 wParam 参数中含有字符的 ASCII 码值。MessageBox 函数（其用法，请读者查看 MSDN，并结合本小程序来学习）弹出一个

包含了显示信息的消息框，如果我们按下字母“a”键（注意大小写），程序将弹出如图 1.3 所示的消息框。



图 1.3 消息框

15~21 行代码：当用户在窗口中按下鼠标左键时，将产生 WM_LBUTTONDOWN 消息。为了证实这一点，我们在 WM_LBUTTONDOWN 消息的响应代码中，调用 MessageBox 函数弹出一个提示信息，告诉用户

“点击了鼠标”。接下来，我们在窗口中 (0,50) 的位置处输出一行文字。要在窗口中输出文字或者显示图形，需要用到设备描述表 (Device Context)，简称 DC。DC 是一个包含设备（物理输出设备，如显示器，以及设备驱动程序）信息的结构体，在 Windows 平台下，所有的图形操作都是利用 DC 来完成的。

关于 DC，我们可以用一个形象的比喻来说明它的作用。现在有一个美术老师，他让他的学生画一幅森林的图像，有的学生采用素描，有的学生采用水彩画，有的学生采用油画，每个学生所作的图都是森林，然而表现形式却各不相同。如果让我们来画图，老师指定了一种画法（例如用水彩画），我们就要去学习它，然后才能按照要求画出图形。如果画法（工具）经常变换，我们就要花大量的时间和精力去学习和掌握它。在这里，画法就相当于计算机中的图形设备及其驱动程序。我们要想作一幅图，就要掌握我们所平台的图形设备和它的驱动程序，调用驱动程序的接口来完成图形的显示。不同图形设备的设备驱动程序是不一样的，对于程序员来说，要掌握各种不同的驱动程序，工作量就太大了。因此，Windows 就给我们提供了一个 DC，让我们从学生的角色转变为老师的角色，只要下命令去画森林这幅图，由 DC 去和设备驱动程序打交道，完成图形的绘制。至于图形的效果，就要由所使用的图形设备来决定了。对于老师来说，只要画出的是森林图像就可以了。对于程序员来说，充当老师的角色，只需要获取 DC（DC 也是一种资源）的句柄，利用这个句柄去作图就可以了。

使用 DC，程序不用为图形的显示与打印输出分别处理了。无论显示，还是打印，都是直接在 DC 上操作，然后由 DC 映射到这些物理设备上。

第 17 行代码：定义了一个类型为 HDC 的变量 hdc。

第 18 行代码：用 hdc 保存 GetDC 函数返回的与特定窗口相关联的 DC 的句柄。为什么 DC 要和窗口相关联呢？想像一下，我们在作图时，需要有画布；利用计算机作图，窗口就相当于画布，因此，在获取 DC 的句柄时，总是和一个指定的窗口相关联。

第 19 行代码：TextOut 函数利用得到的 DC 句柄在指定的位置（x 坐标为 0，y 坐标为 50）处输出一行文字。

第 20 行代码：在执行图形操作时，如果使用 GetDC 函数来得到 DC 的句柄，那么在完成图形操作后，必须调用 ReleaseDC 函数来释放 DC 所占用的资源，否则会引起内存泄漏。

第 22~28 行代码：对 WM_PAINT 消息进行处理。当窗口客户区的一部分或者全部变为“无效”时，系统会发送 WM_PAINT 消息，通知应用程序重新绘制窗口。当窗口刚创建的时候，整个客户区都是无效的。因为这个时候程序还没有在窗口上绘制任何东西，当调用 UpdateWindow 函数时，会发送 WM_PAINT 消息给窗口过程，对窗口进行刷新。当窗口从无到有、改变尺寸、最小化后再恢复、被其他窗口遮盖后再显示时，窗口的客户区

都将变为无效，此时系统会给应用程序发送 WM_PAINT 消息，通知应用程序重新绘制。



提示：窗口大小发生变化时是否发生重绘，取决于 WNDCLASS 结构体中 style 成员是否设置了 CS_HREDRAW 和 CS_VREDRAW 标志。

第 25 行，调用 BeginPaint 函数得到 DC 的句柄。BeginPaint 函数的第 1 个参数是窗口的句柄，第二个参数是 PAINTSTRUCT 结构体的指针，该结构体对象用于接收绘制的信息。在调用 BeginPaint 时，如果客户区的背景还没有被擦除，那么 BeginPaint 会发送 WM_ERASEBKGDND 消息给窗口，系统就会使用 WNDCLASS 结构体的 hbrBackground 成员指定的画刷来擦除背景。

第 26 行，调用 TextOut 函数在 (0,0) 的位置输出一个网址 “http://www.sunxin.org”。当发生重绘时，窗口中的文字和图形都会被擦除。在擦除背景后，TextOut 函数又一次执行，在窗口中再次绘制出 “http://www.sunxin.org”。这个过程对用户来说是透明的，用户并不知道程序执行的过程，给用户的感觉就是你在响应 WM_PAINT 消息的代码中输出的文字或图形始终保持在窗口中。换句话说，如果我们想要让某个图形始终在窗口中显示，就应该将图形的绘制操作放到响应 WM_PAINT 消息的代码中。

那么系统为什么不直接保存窗口中的图形数据，而要由应用程序不断地进行重绘呢？这主要是因为图形环境中涉及的数据量太大，为了节省内存的使用，提高效率，而采用了重绘的方式。

在响应 WM_PAINT 消息的代码中，要得到窗口的 DC，必须调用 BeginPaint 函数。BeginPaint 函数也只能在 WM_PAINT 消息的响应代码中使用，在其他地方，只能使用 GetDC 来得到 DC 的句柄。另外，BeginPaint 函数得到的 DC，必须用 EndPaint 函数去释放。

29~34 行代码：当用户单击窗口上的关闭按钮时，系统将给应用程序发送一条 WM_CLOSE 消息。在这段消息响应代码中，我们首先弹出一个消息框，让用户确认是否结束。如果用户选择“否”，则什么也不做；如果用户选择“是”，则调用 DestroyWindow 函数销毁窗口，DestroyWindow 函数在销毁窗口后会向窗口过程发送 WM_DESTROY 消息。注意，此时窗口虽然销毁了，但应用程序并没有退出。有不少初学者错误地在 WM_DESTROY 消息的响应代码中，提示用户是否退出，而此时窗口已经销毁了，即使用户选择不退出，也没有什么意义了。所以如果你要控制程序是否退出，应该在 WM_CLOSE 消息的响应代码中完成。

对 WM_CLOSE 消息的响应并不是必须的，如果应用程序没有对该消息进行响应，系统将把这条消息传给 DefWindowProc 函数（参见第 39 行），而 DefWindowProc 函数则调用 DestroyWindow 函数来响应这条 WM_CLOSE 消息。

35~37 行代码：DestroyWindow 函数在销毁窗口后，会给窗口过程发送 WM_DESTROY 消息，我们在该消息的响应代码中调用 PostQuitMessage 函数（第 36 行）。PostQuitMessage 函数向应用程序的消息队列中投递一条 WM_QUIT 消息并返回。我们在 1.4.3 小节介绍过，GetMessage 函数只有在收到 WM_QUIT 消息时才返回 0，此时消息循环才结束，程序退出。要想让程序正常退出，我们必须响应 WM_DESTROY 消息，并在消息响应代码中调用

PostQuitMessage, 向应用程序的消息队列中投递 WM_QUIT 消息。传递给 PostQuitMessage 函数的参数值将作为 WM_QUIT 消息的 wParam 参数, 这个值通常用做 WinMain 函数的返回值。

38、39 行代码: DefWindowProc 函数调用默认的窗口过程, 对应用程序没有处理的其他消息提供默认处理。对于大多数的消息, 应用程序都可以直接调用 DefWindowProc 函数进行处理。在编写窗口过程时, 应该将 DefWindowProc 函数的调用放到 default 语句中, 并将该函数的返回值作为窗口过程函数的返回值。

读者可以试着将 38、39 行代码注释起来, 运行一下, 看看会有什么结果。提示: 运行之后, 在 NT4.0/Win2000 下启动任务管理器, 切换到进程标签, 查看程序是否运行。

1.5 动手写第一个 Windows 程序

到现在为止, 读者对创建一个窗口应该有了大致的印象, 但是, 光看书是不行的, 应该试着动手去编写程序。本节的内容就是教读者怎样去编写一个 Windows 窗口应用程序。完整的例程请参见光盘中的 Chapter1 目录下的 WinMain。

❶ 启动 Microsoft Visual C++6.0, 单击【File】菜单, 选择【New】菜单项, 在“Projects”选项卡下, 选择“Win32 Application”, 在右侧的“Project name:”文本框中, 输入我们的工程名 WinMain (如图 1.4 所示), 单击【OK】按钮。

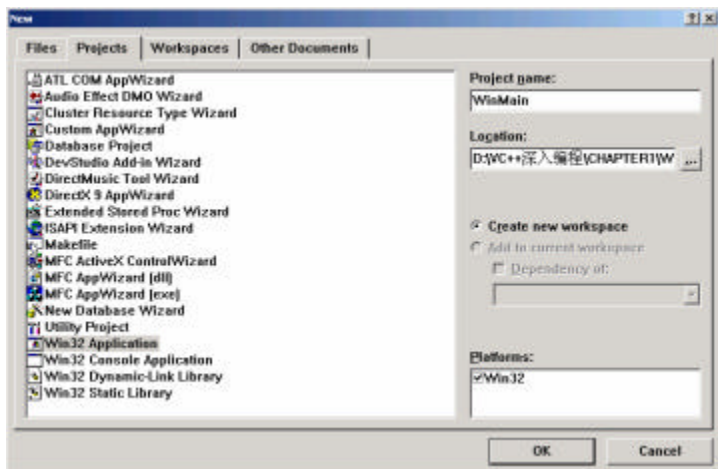


图 1.4 创建 WinMain 新工程

❷ 在 Win32 Application-Step 1 of 1 中, 选择“An empty project”(如图 1.5 所示), 单击【Finish】按钮。

❸ 出现一个工程信息窗口, 单击【OK】按钮, 这样就生成了一个空的应用程序外壳。

❹ 这样的应用程序外壳并不能做什么, 甚至不能运行, 我们还要为它加上源文件。单击【File】菜单, 选择【New】, 在“Files”选项卡下, 选择“C++ Source File”, 在右侧的“File”文本框中, 输入源文件的文件名 WinMain (如图 1.6 所示), 单击【OK】按钮。

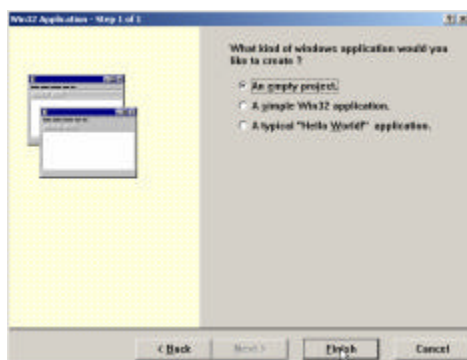


图 1.5 选择 An empty project 选项

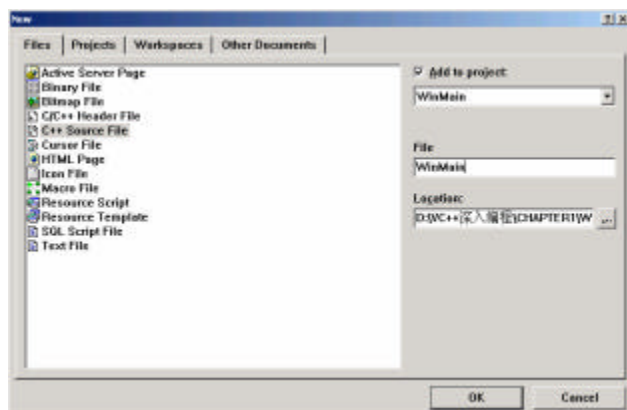


图 1.6 添加 WinMain.cpp 源文件

输入以下代码：

WinMain.cpp

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

LRESULT CALLBACK WinSunProc(
    HWND hwnd,          // handle to window
    UINT uMsg,           // message identifier
    WPARAM wParam,       // first message parameter
    LPARAM lParam        // second message parameter
);

int WINAPI WinMain(
    HINSTANCE hInstance,    // handle to current instance
    HINSTANCE hPrevInstance, // handle to previous instance
    LPSTR lpCmdLine,        // command line
    int nCmdShow            // show state
)
{
```

```

//设计一个窗口类
WNDCLASS wndcls;
wndcls.cbClsExtra=0;
wndcls.cbWndExtra=0;
wndcls.hbrBackground=(HBRUSH)GetStockObject(BLACK_BRUSH);
wndcls.hCursor=LoadCursor(NULL, IDC_CROSS);
wndcls.hIcon=LoadIcon(NULL, IDI_ERROR);
wndcls.hInstance=hInstance;    //应用程序实例句柄由 WinMain 函数传进来
wndcls.lpfnWndProc=WinSunProc;
wndcls.lpszClassName="sunxin2006";
wndcls.lpszMenuName=NULL;
wndcls.style=CS_HREDRAW | CS_VREDRAW;
RegisterClass(&wndcls);

//创建窗口，定义一个变量用来保存成功创建窗口后返回的句柄
HWND hwnd;
hwnd=CreateWindow("sunxin2006", "http://www.sunxin.org",
    WS_OVERLAPPEDWINDOW, 0, 0, 600, 400, NULL, NULL, hInstance, NULL);

//显示及刷新窗口
ShowWindow(hwnd, SW_SHOWNORMAL);
UpdateWindow(hwnd);

//定义消息结构体，开始消息循环
MSG msg;
while(GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
return msg.wParam;
}

//编写窗口过程函数
LRESULT CALLBACK WinSunProc(
    HWND hwnd,        // handle to window
    UINT uMsg,        // message identifier
    WPARAM wParam,    // first message parameter
    LPARAM lParam     // second message parameter
)
{
    switch(uMsg)
    {
        case WM_CHAR:
            char szChar[20];
            sprintf(szChar, "char code is %d", wParam);
            MessageBox(hwnd, szChar, "char", 0);
            break;
    }
}

```

时调用

```

case WM_LBUTTONDOWN:
    MessageBox(hwnd, "mouse clicked", "message", 0);
    HDC hdc;
    hdc = GetDC(hwnd);           //不能在响应 WM_PAINT 消息时调用
    TextOut(hdc, 0, 50, "程序员之家", strlen("程序员之家"));
    //ReleaseDC(hwnd, hdc);
    break;
case WM_PAINT:
    HDC hdc;
    PAINTSTRUCT ps;
    hdc = BeginPaint(hwnd, &ps); //BeginPaint 只能在响应 WM_PAINT 消息

    TextOut(hdc, 0, 0, "http://www.sunxin.org", strlen("http://www.
sunxin.org"));
    EndPaint(hwnd, &ps);
    break;
case WM_CLOSE:
    if (IDYES == MessageBox(hwnd, "是否真的结束?", "message", MB_YESNO))
    {
        DestroyWindow(hwnd);
    }
    break;
case WM_DESTROY:
    PostQuitMessage(0);
    break;
default:
    return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
return 0;
}

```

如果读者消化吸收了本章的内容，编写上述程序并不难。希望读者仔细思考一下本章所讲的内容，尽量参照每一步中所讲述的知识点，自己将程序编写出来。



提示：不少初学者在编写上述程序时，出现了下面的错误：

```

-----Configuration: WinMain- Win32 Debug-----
Compiling...
WinMain.cpp
Linking...
LIBCD.lib(crt0.obj) : error LNK2001: unresolved external symbol
_main
Debug/WinMain.exe : fatal error LNK1120: 1 unresolved externals
Error executing link.exe.

WinMain.exe - 2 error(s), 0 warning(s)

```

这主要是因为图 1.4 所示的步骤中，选错了工程（项目）类型，本章的程序应该选择“Win32 Application”，结果却选择了该类型的下面一个“Win32 Console Application”。

在这里，给读者也布置一个任务，在 MSDN 查找“Win32 Application”和“Win32 Console Application”的说明，弄清楚它们之间的区别。

出现问题的解决办法：

(1) 当然是重新建一个工程，选择正确的工程类型（相信读者看了下面的解决方法后，不会选择这一种 ☹）。

(2) 在 VC++ 集成开发环境中，单击菜单【Project】 【Settings】，选择“Link”选项卡，在最下方的“Project Options”列表框中找到“/subsystem:console”后，将其删除，或者修改为“/subsystem:windows”，单击“确定”按钮。重新编译运行程序。

1.6 消息循环的错误分析

有不少初学者学完第 1 章后，编写了下面的代码：

```
...
HWND hwnd;
    hwnd=CreateWindow(...);
...
MSG msg;
while(GetMessage(&msg,hwnd,0,0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
...
```

注意代码中以粗体显示的部分。这段代码基于这样一个想法：第 1 章的程序只有一个窗口，而我们前面说了 GetMessage 函数的 hWnd 参数是用于指定接收属于哪一个窗口的消息，于是不少人就在消息循环中为 GetMessage 函数的 hWnd 参数指定了 CreateWindow 函数返回的窗口句柄。

读者可以用上述代码中的消息循环部分替换 1.5 节代码中的消息循环部分，然后运行程序，关闭程序。你会发现你的机器变慢了，同时按下键盘上的 Ctrl + Alt + Delete 键，启动 Windows 的任务管理器，切换到“进程”选项卡，单击“CPU”项进行排序，你会发现如图 1.7 所示的情况。

从图 1.7 中可以看到，WinMain.exe 的 CPU 占用率接近 100，难怪机器“变慢了”。那么这是什么原因呢？实际上这个问题的答案在 MSDN 中就可以找到，并且就在 GetMessage 函数的说明文档中。不少初学者在遇到问题时，首先是头脑一片空白，接着就去找人求助，这种思想用在程序开发的学习中，没有什么好处。笔者经常遇到学员问问题，结果有不少问题的答案在 MSDN 关于某个函数的解释中就可看到（由于显示器的限制，有的答案需要滚动窗口才能看到 ☹）。所以在这里，笔者也建议读者遇到问题一定要记得查看 MSDN，学会使用 MSDN 并从中汲取知识，将使你受用无穷。

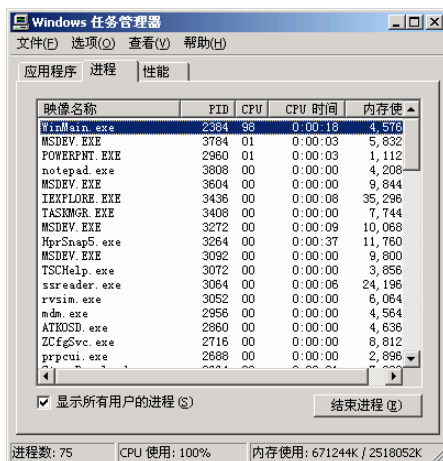


图 1.7 WinMain.exe 的 CPU 占用率接近 100

回到正题，在 1.4.3 节介绍 GetMessage 函数时，曾说过如果 hWnd 参数是无效的窗口句柄或 lpMsg 参数是无效的指针时，GetMessage 函数将返回-1。当我们关闭窗口时，调用了 DestroyWindow 来销毁窗口，由于窗口被销毁了，窗口的句柄当然也就是无效的句柄了，那么 GetMessage 将返回-1。在 C/C++语言中，非 0 即为真，由于窗口被销毁，句柄变为无效，GetMessage 总是返回-1，循环条件总是为真，于是形成了一个死循环，机器当然就“变慢了”。☺

在 MSDN 关于 GetMessage 函数的说明文档中给出了下面的代码：

```

BOOL bRet;

while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0 )
{
    if (bRet == -1)
    {
        // handle the error and possibly exit
    }
    else
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}

```

针对我们这个问题，可以修改上述代码如下：

```

...
HWND hwnd;
hwnd=CreateWindow(...);

...
MSG msg;

```



```

BOOL bRet;

while( (bRet = GetMessage( &msg, hwnd, 0, 0 )) != 0)
{
    if (bRet == -1)
    {
        // handle the error and possibly exit
        return -1;
    }
    else
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
...

```

读者可以再次运行修改后的程序，看看运行的结果。

1.7 变量的命名约定

由于 Windows 程序一般很长，编程人员在一段时间后自己都有可能忘记所定义的变量的含义。为了帮助大家记忆与区分变量，微软公司创建了 Windows 的命名约定，称之为匈牙利表示法（Hungarian notation）。匈牙利表示法提供了一组前缀字符，如表 1.1 所示，这些前缀也可以组合起来使用。

表 1.1 匈牙利表示法

前 缀	含 义
a	数组
b	布尔值（int）
by	无符号字符（字节）
c	字符（字节）
cb	字节记数
rgb	保存 RGB 颜色值的长整型
cx,cy	短整型（计算 x,y 的长度）
dw	无符号长整型
fn	函数
h	句柄
i	整数（integer）
m_	类的数据成员
n	短整型或整型
np	近指针
p	指针

续表

前 缀	含 义
l	长整型
lp	长指针
s	字符串
sz	以零结束的字符串
tm	正文大小
w	无符号整型
x,y	无符号整型（表示 x 或 y 的坐标）

1.8 小结

这一章详细介绍了 Windows 程序运行的内部机制。建议读者花点时间把本章的内容消化理解了。在学习本章内容的时候，可以结合配套光盘中的 VC 视频第 1 课一起学习。如果读者没有完全掌握本章的内容，也不要灰心，这里涉及到的一些内容，在后续章节中还会讲到。学习 VC++ 的路程是艰苦的，必须具有一定的毅力并不断努力，才有可能精通 VC++ 编程。

下面我们再次为读者总结一下创建一个 Win32 应用程序的步骤。

- ❶ 编写 WinMain 函数，可以在 MSDN 上查找并复制。
- ❷ 设计窗口类（WNDCLASS）。
- ❸ 注册窗口类。
- ❹ 创建窗口。
- ❺ 显示并更新窗口。
- ❻ 编写消息循环。
- ❼ 编写窗口过程函数。窗口过程函数的语法，可通过 MSDN 查看 WNDCLASS 的 lpfnWndProc 成员变量，在这个成员的解释中可查到。

掌握 C++

在学习 Visual C++ 6.0 编程之前，有必要复习一下 C++ 中面向对象的一些基本概念。我们知道，C++ 与 C 相比有许多优点，主要体现在封装性（Encapsulation）、继承性（Inheritance）和多态性（Polymorphism）。封装性把数据与操作数据的函数组织在一起，不仅使程序结构更加紧凑，并且提高了类内部数据的安全性；继承性增加了软件的可扩充性及代码重用性；多态性使设计人员在设计程序时可以对问题进行更好的抽象，有利于代码的维护和可重用。Visual C++ 不仅仅是一个编译器，更是一个全面的应用程序开发环境，读者可以充分利用具有面向对象特性的 C++ 语言开发出专业级的 Windows 应用程序。熟练掌握本章的内容，将为后续章节的学习打下良好的基础。

2.1 从结构到类

在 C 语言中，我们可以定义结构体类型，将多个相关的变量包装为一个整体使用。在结构体中的变量，可以是相同、部分相同，或完全不同的数据类型。在 C 语言中，结构体不能包含函数。在面向对象的程序设计中，对象具有状态（属性）和行为，状态保存在成员变量中，行为通过成员方法（函数）来实现。C 语言中的结构体只能描述一个对象的状态，不能描述一个对象的行为。在 C++ 中，对结构体进行了扩展，C++ 的结构体可以包含函数。

2.1.1 结构体的定义

下面我们看看如例 2-1 所示的程序（EX01.CPP）。

例 2-1

```
#include <iostream.h>
struct point
{
```

```
int x;
int y;
};
void main()
{
    point pt;
    pt.x=0;
    pt.y=0;
    cout<<pt.x<<endl<<pt.y<<endl;
}
```

在这段程序中，我们定义了一个结构体 `point`，在这个结构体当中，定义了两个整型的变量，作为一个点的 X 坐标和 Y 坐标。在 `main` 函数中，定义了一个结构体的变量 `pt`，对 `pt` 的两个成员变量进行赋值，然后调用 C++ 的输出流类的对象 `cout` 将这个点的坐标输出。

在 C++ 中预定义了三个标准输入输出流对象：`cin`（标准输入）、`cout`（标准输出）和 `cerr`（标准错误输出）。`cin` 与输入操作符（`>>`）一起用于从标准输入读入数据，`cout` 与输出操作符（`<<`）一起用于输出数据到标准输出上，`cerr` 与输出操作符（`<<`）一起用于输出错误信息到标准错误上（一般同标准输出）。默认的标准输入通常为键盘，默认的标准输出和标准错误输出通常为显示器。

`cin` 和 `cout` 的使用比 C 语言中的 `scanf` 和 `printf` 要简单得多。使用 `cin` 和 `cout` 你不需要去考虑输入和输出的数据的类型，`cin` 和 `cout` 可以自动根据数据的类型调整输入输出的格式。

对于输出来说，按照例 2-1 中所示的方式调用就可以了，对于输入来说，我们以如下方式调用即可：

```
int i;
cin>>i;
```



注意：在使用 `cin` 和 `cout` 对象时，要注意箭头的方向。在输出中我们还使用了 `endl`（end of line），表示换行，注意最后一个是字母 ‘`l`’，而不是数字 1。`endl` 相当于 C 语言的 ‘`\n`’，`endl` 在输出流中插入一个换行，并刷新输出缓冲区。

因为用到了 C++ 的标准输入输出流，所以我们需要包含 `iostream.h` 这个头文件，就像我们在 C 语言中用到了 `printf` 和 `scanf` 函数时，要包含 C 的标准输入输出头文件 `stdio.h`。



提示：在定义结构体时，一定不要忘了在右花括号处加上一个分号（`;`）。

我们将结构体 `point` 的定义修改一下，结果如例 2-2 所示：

例 2-2

```
struct point
{
    int x;
```

```
int y;
void output()
{
    cout<<x<<endl<<y<<endl;
}
};
```

在 point 这个结构体中加入了一个函数 output。我们知道在 C 语言中，结构体中是不能有函数的，然而在 C++ 中，结构体中是可以有函数的，称为成员函数。这样，在 main 函数中就可以以如下方式调用：

```
void main()
{
    point pt;
    pt.x=0;
    pt.y=0;
    // cout<<pt.x<<endl<<pt.y<<endl;
    pt.output();
}
```



注意：在 C++ 中，//.....用于注释一行，/*.....*/用于注释多行。

2.1.2 结构体与类

将上面例 2-2 所示的 point 结构体定义中的关键字 struct 换成 class，得到如例 2-3 所示的定义。

例 2-3

```
class point
{
    int x;
    int y;
    void output()
    {
        cout<<x<<endl<<y<<endl;
    }
};
```

这就是 C++ 中的类的定义，看起来是不是和结构体的定义很类似？在 C++ 语言中，结构体是用关键字 struct 声明的类。类和结构体的定义除了使用关键字“class”和“struct”不同之外，更重要的是在成员的访问控制方面有所差异。结构体默认情况下，其成员是公有（public）的；类默认情况下，其成员是私有（private）的。在一个类当中，公有成员是在类的外部进行访问的，而私有成员就只能在类的内部进行访问了。例如，现在设计家庭这样一个类，对于家庭的客厅，可以让家庭成员以外的人访问，我们就可以将客厅设置为 public。对于卧室，只有家庭成员才能访问，我们可以将其设置为 private。



提示：在定义类时，同样不要忘了在右花括号处加上一个分号（;）

如果我们编译例 2-4 所示的程序（EX02.CPP）：

例 2-4

```
#include <iostream.h>
class point
{
    int x;
    int y;
    void output()
    {
        cout<<x<<endl<<y<<endl;
    }
};

void main()
{
    point pt;
    pt.x=0;
    pt.y=0;
    pt.output();
}
```

将会出现如图 2.1 所示的错误提示信息，提示我们不能访问类中私有（private）的成员变量和成员函数。

```
1 : error C2248: 'x' : cannot access private member declared in class 'point'
I.CPP(4) : see declaration of 'x'
1 : error C2248: 'y' : cannot access private member declared in class 'point'
I.CPP(5) : see declaration of 'y'
1 : error C2248: 'output' : cannot access private member declared in class 'point'
I.CPP(6) : see declaration of 'output'
```

图 2.1 在类的外部访问类中私有成员变量提示出错

2.2 C++的特性

下面我们将通过具体的代码演示，给读者讲解 C++类的特性。所使用的 C++开发工具是微软公司出品的 Visual C++ 6.0，操作系统是 Windows2000 Server SP4。

- ❶ 启动 Microsoft Visual C++6.0，如图 2.2 所示。
- ❷ 单击 File 菜单，选择 New，如图 2.3 所示。
- ❸ 在 Projects 选项卡下，选择 Win32 Console Application，如图 2.4 所示。
- ❹ 在右边的 Project name：中，输入工程名 EX03，单击 OK 按钮，如图 2.5 所示。

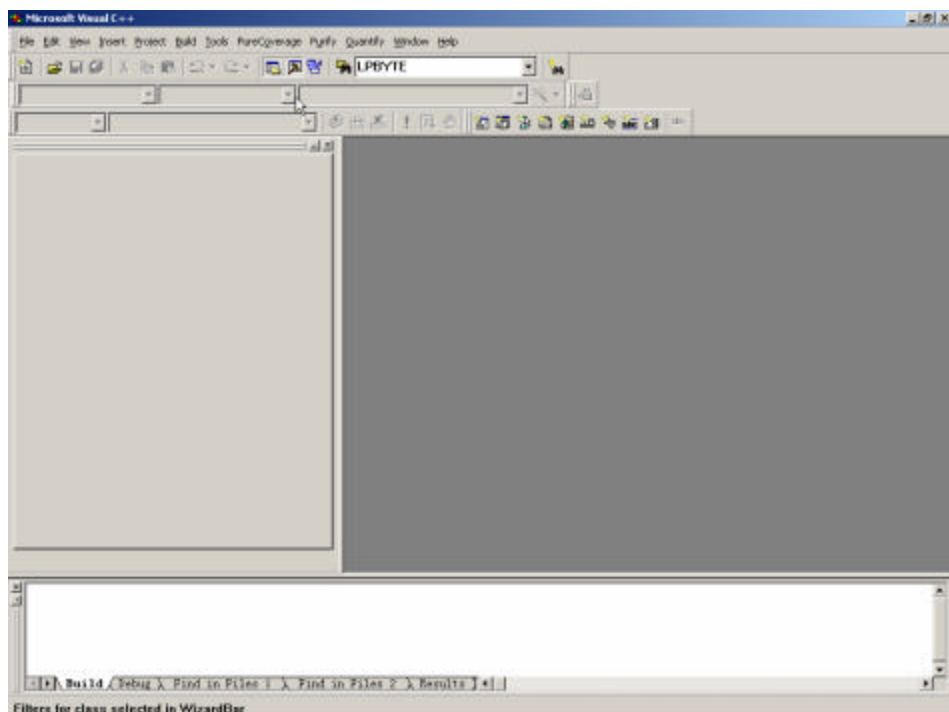


图 2.2 Microsoft Visual C++6.0 初始界面

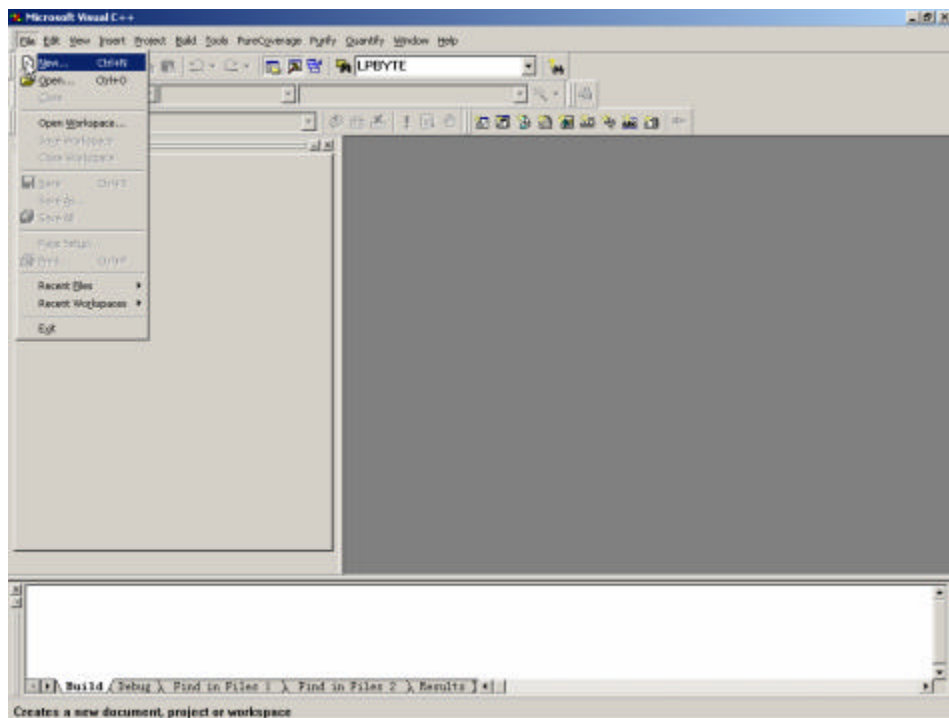


图 2.3 选择【File\New】菜单项

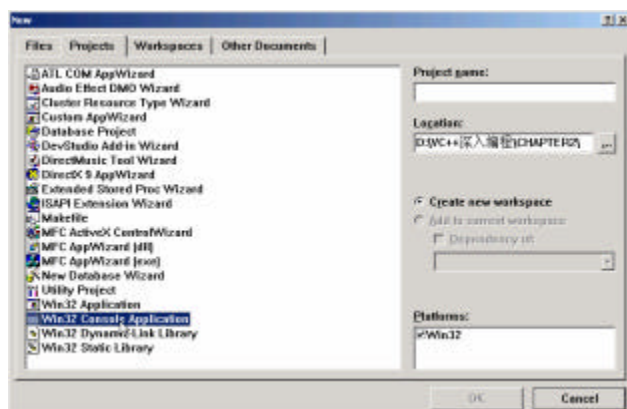


图 2.4 选择 Win32 Console Application 工程类型

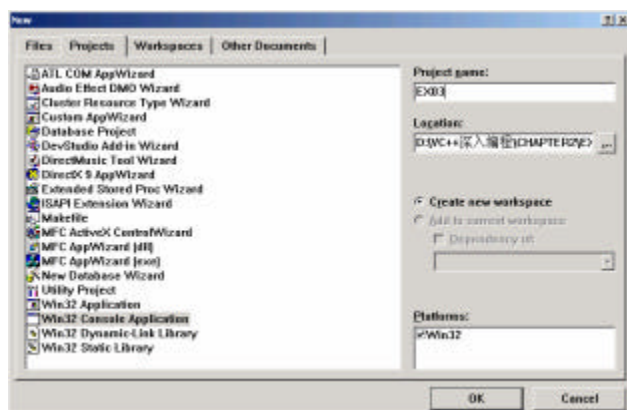


图 2.5 输入工程名

⑤ 在 Win32 Console Application-Step 1 of 1 中，选择 An empty project 单选按钮，单击【Finish】按钮，如图 2.6 所示。

⑥ 出现一个工程信息窗口，单击【OK】按钮，如图 2.7 所示，这样就生成了一个空的应用程序外壳。

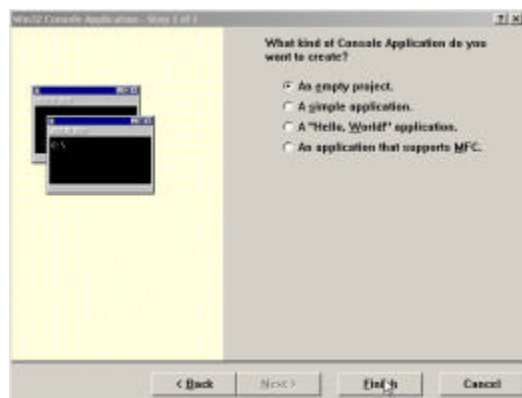


图 2.6 选择 An empty project 选项

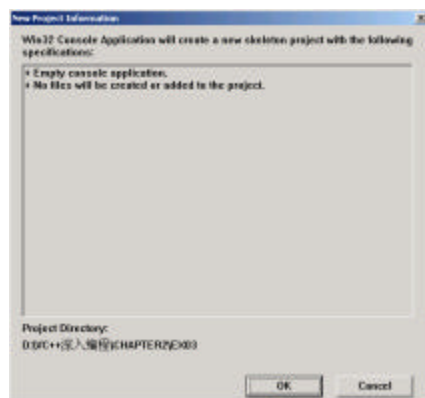


图 2.7 新工程信息

7 这样的应用程序外壳并不能做什么，甚至不能运行，我们还要为它加上源文件。单击【File】菜单，选择【New】；然后在 Files 选项卡下，选择 C++ Source File，如图 2.8 所示。

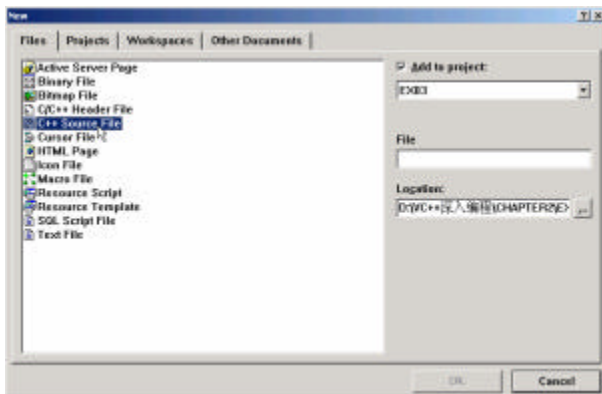


图 2.8 为程序增加 C++源文件

8 在右边的 File 文本框中，输入文件名 EX03，单击【OK】按钮，如图 2.9 所示。

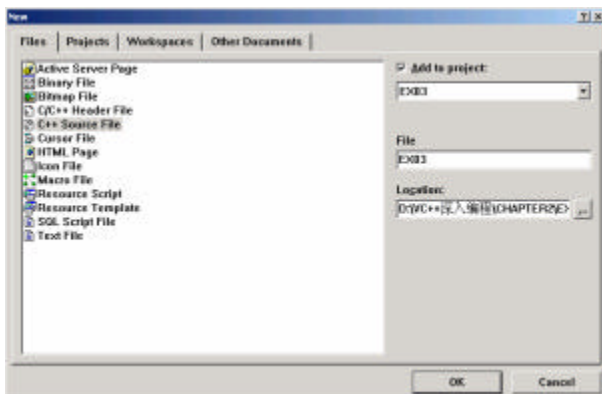


图 2.9 输入 C++源文件名称

并在 EX03.cpp 文件中输入以下代码：

例 2-5

```
#include <iostream.h>
class point
{
public:
    int x;
    int y;
    void output()
    {
        cout<<x<<endl<<y<<endl;
    }
};
```

```
void main()
{
    point pt;
    pt.output();
}
```



说明：在这一章中，我们所有的示例工程都通过上述方式创建。



提示：如果你在编译程序时出现了下面的错误，请想想错误的原因，然后参照 1.5 节给出的问题解决办法，解决下面的错误。

```
-----Configuration: EX03 - Win32 Debug-----
Compiling...
EX03.CPP
Linking...
LIBCD.lib(wincrt0.obj) : error LNK2001: unresolved external symbol _WinMain@16
Debug/EX03.exe : fatal error LNK1120: 1 unresolved externals
Error executing link.exe.
```

```
EX03.exe - 2 error(s), 0 warning(s)
```

2.2.1 类与对象

在这个程序中，我们定义了一个类 `point`，在 `main` 函数中我们定义了一个 `pt` 对象，它的类型是 `point` 这个类。C++ 语言是面向对象的语言，那么，什么是类？什么是对象呢？

类描述了一类事物，以及事物所应具有的属性，例如：我们可以定义“电脑”这个类，那么作为“电脑”这个类，它应该具有显示器、主板、CPU、内存、硬盘，等等。那么什么是“电脑”的对象呢？例如，我们组装的一台具体的电脑，它的显示器是美格的，主板是华硕的，CPU 是 Intel 的，内存是现代的，硬盘用的是希捷的，也就是“电脑”这个类所定义的属性，在我们购买的这台具体的电脑中，有了具体的值。

这台具体的电脑就是我们“电脑”这个类的一个对象。我们还经常听到“类的实例”，什么是“类的实例”呢？实际上，类的实例和类的对象是一个概念。

对象是可以销毁的。例如，我们购买的这台电脑，它是可以被损毁的。而类是不能被损毁的，我们不能说把电脑毁掉，“电脑”类是一个抽象的概念。

2.2.2 构造函数

按下键盘上的 F7 功能键编译例 2-5 的代码，然后按下键盘上的 Ctrl+F5 执行程序，出现如图 2.10 所示的运行结果。

从图中可以看到，输出了两个很大的负数。这是因为在构造 `pt` 对象时，系统要为它的成员变量 `x` 和 `y` 分配内存空间，而在这个内存空间中的值是一个随机值，在程序中我们没有给这两个变量赋值，因此输出时就看到了如图 2.10 所示的结果。这当然不是我们所期望

的，作为一个点的两个坐标来说，应该有一个合理的值。为此，我们想到定义一个初始化函数，用它来初始化 x 和 y 坐标。这时程序的代码如例 2-6 所示，其中加灰显示的部分为新添加的代码。

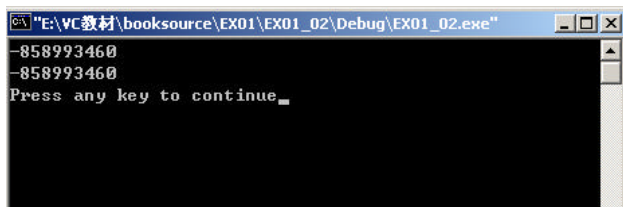


图 2.10 EX03 程序的运行结果

例 2-6

```
#include <iostream.h>
class point
{
public:
    int x;
    int y;
    void init()
    {
        x=0;
        y=0;
    }
    void output()
    {
        cout<<x<<endl<<y<<endl;
    }
};
void main()
{
    point pt;
    pt.init();
    pt.output();
}
```

然而，对于我们定义的 `init` 函数，在编写程序时仍然有可能忘记调用它。那么，能不能在我们定义 `pt` 这个对象的同时，就对 `pt` 的成员变量进行初始化呢？在 C++ 当中，给我们提供了一个构造函数，可以用来对类中的成员变量进行初始化。

C++ 规定构造函数的名字和类名相同，没有返回值。我们将 `init` 这个函数删去，增加一个构造函数 `point`。这时程序的代码如例 2-7 所示，其中加灰显示的部分为新添加的代码。

例 2-7

```
#include <iostream.h>
class point
{
```


```
public:
    int x;
    int y;
    point()    //point 类的构造函数
    {
        x=0;
        y=0;
    }
    void output()
    {
        cout<<x<<endl<<y<<endl;
    }
};

void main()
{
    point pt;
    pt.output();
}
```

在程序中, point 这个构造函数没有任何返回值。我们在函数内部对 x 和 y 变量进行了初始化, 按 F7 编译代码, 按 Ctrl+F5 执行程序, 可以看到输出结果是两个 0。

构造函数的作用是对对象本身做初始化工作, 也就是给用户初始化类中成员变量的一种方式。可以在构造函数中编写代码, 对类中的成员变量进行初始化。在例 2-7 的程序中, 当在 main 函数中执行 “point pt” 这条语句时, 就会自动调用 point 这个类的构造函数, 从而完成对 pt 对象内部数据成员 x 和 y 的初始化工作。

如果一个类中没有定义任何的构造函数, 那么 C++ 编译器在某些情况下会为该类提供一个默认的构造函数, 这个默认的构造函数是一个不带参数的构造函数。只要一个类中定义了一个构造函数, 不管这个构造函数是否是带参数的构造函数, C++ 编译器就不再提供默认的构造函数。也就是说, 如果为一个类定义了一个带参数的构造函数, 还想要无参数的构造函数, 则必须自己定义。

 **知识点** 国内很多介绍 C++ 的图书, 对于构造函数的说明, 要么是错误的, 要么没有真正说清楚构造函数的作用。在网友 backer 的帮助下, 我们参看了 ANSI C++ 的 ISO 标准, 并从汇编的角度试验了几种主流编译器的行为, 对于编译器提供默认构造函数的行为得出了下面的结论:

如果一个类中没有定义任何的构造函数, 那么编译器只有在以下三种情况, 才会提供默认的构造函数:

1. 如果类有虚拟成员函数或者虚拟继承父类 (即有虚拟基类) 时;
2. 如果类的基类有构造函数 (可以是用户定义的构造函数, 或编译器提供的默认构造函数);
3. 在类中的所有非静态的对象数据成员, 它们所属的类中有构造函数 (可以是用户定义的构造函数, 或编译器提供的默认构造函数)。

2.2.3 析构函数

当一个对象的生命周期结束时，我们应该去释放这个对象所占有的资源，这可以利用析构函数来完成。析构函数的定义格式为：`~类名()`，如：`~point()`。

析构函数是“反向”的构造函数。析构函数不允许有返回值，更重要的是析构函数不允许带参数，并且一个类中只能有一个析构函数。析构函数的作用正好与构造函数相反，析构函数用于清除类的对象。当一个类的对象超出它的作用范围，对象所在的内存空间被系统回收，或者在程序中用 `delete` 删除对象时，析构函数将自动被调用。对一个对象来说，析构函数是最后一个被调用的成员函数。

根据析构函数的这种特点，我们可以在构造函数中初始化对象的某些成员变量，为其分配内存空间（堆内存），在析构函数中释放对象运行期间所申请的资源。

例如，下面这段程序：

```
class Student
{
private:
    char *pName;
public:
    Student()
    {
        pName=new char[20];
    }
    ~Student()
    {
        delete[] pName;
    }
};
```

在 `Student` 类的构造函数中，给字符指针变量 `pName` 在堆上分配了 20 个字符的内存空间，在析构函数中调用 `delete`，释放在堆上分配的内存。如果没有 `delete[] pName` 这句代码，当我们定义一个 `Student` 的对象，在这个对象生命周期结束时，在它的构造函数中分配的这块堆内存就会丢失，造成内存泄漏。



提示：在类中定义成员变量时，不能直接给成员变量赋初值。例如：

```
class point
{
    int x=0;//错误，此处不能给变量 x 赋值。
    int y;
};
```

2.2.4 函数的重载

我们希望在构造 `pt` 这个对象的同时，传递 `x` 坐标和 `y` 坐标的值。可以再定义一个构造函数，如例 2-8 所示。

例 2-8

```
#include <iostream.h>
class point
{
public:
    int x;
    int y;
    point()
    {
        x=0;
        y=0;
    }
    point(int a, int b)
    {
        x=a;
        y=b;
    }
    void output()
    {
        cout<<x<<endl<<y<<endl;
    }
};

void main()
{
    point pt(5,5);
    pt.output();
}
```

在这个程序中，有两个构造函数，它们的函数名是一样的，只是参数的类型和个数不一样。这在 C 语言中是不允许的，而在 C++ 中上述定义是合法的，这就是 C++ 中函数的重载（**overload**）。当执行 main 函数中的 point pt(5,5) 这条语句时，C++ 编译器将根据参数的类型和参数的个数来确定执行哪一个构造函数，在这里即执行 point(int a, int b) 这个函数。

重载构成的条件：函数的参数类型、参数个数不同，才能构成函数的重载。分析以下两种情况，是否构成函数的重载。

第一种情况：(1) void output();

(2) int output();

第二种情况：(1) void output(int a, int b=5);

(2) void output(int a);

对于第一种情况，当我们在程序中调用 output() 函数时，读者认为应该调用的是哪一个函数呢？要注意：只有函数的返回类型不同是不能构成函数的重载的。

对于第二种情况，当我们在程序中调用 output(5) 时，应该调用的是哪一个函数呢？调用 (1) 的函数可以吗？当然是可以的，因为 (1) 的函数第二个参数有一个默认值，因

此可以认为调用的是第一个函数；当然也可以是调用（2）的函数。由于调用有歧义，因此这种情况也不能构成函数的重载。在函数重载时，要注意函数带有默认参数的这种情况。

2.2.5 this 指针

我们再看例 2-9 所示的这段代码（EX04.CPP）：

例 2-9

```
#include <iostream.h>
class point
{
public:
    int x;
    int y;
    point()
    {
        x=0;
        y=0;
    }
    point(int a,int b)
    {
        x=a;
        y=b;
    }
    void output()
    {
        cout<<x<<endl<<y<<endl;
    }
    void input(int x,int y)
    {
        x=x;
        y=y;
    }
};

void main()
{
    point pt(5,5);
    pt.input(10,10);
    pt.output();
}
```

我们在 point 类中定义了一个 input 函数。在这个函数中，用参数 x 和参数 y 分别给成员变量 x 和 y 进行了赋值。在 main 函数中，先调用 pt 对象的 input 函数，接收用户输入的坐标值，然后调用 output 函数输出 pt 对象的坐标值。

读者可以思考一下这段程序的运行结果，然后编译运行，看看结果和你所思考的结果是一样的吗？

有的读者可能会认为在 `input (int x, int y)` 函数中，利用形参 `x` 和形参 `y` 对 `point` 类中的成员变量 `x` 和 `y` 进行了赋值，然而事实是这样吗？因为变量的可见性，`point` 类的成员变量 `x` 和 `y` 在 `input (int x, int y)` 这个函数中是不可见的，所以，我们实际上是将形参 `x` 的值赋给了形参 `x`，将形参 `y` 的值赋给了形参 `y`，根本没有给 `point` 类的成员变量 `x` 和 `y` 进行赋值，程序运行的结果当然就是“5,5”了。

如何在 `input (int x, int y)` 这个函数中对 `point` 类的成员变量 `x` 和 `y` 进行赋值呢？有的读者马上就想到，将 `input` 函数的参数名改一下不就可以了吗？比如：将函数改为 `input (int a, int b)`，当然，这也是一种解决办法。如果我们不想改变函数的参数名，那么又如何去给 `point` 类的成员变量 `x` 和 `y` 进行赋值呢？

在这种情况下，可以利用 C++ 提供的一个特殊的指针——`this` 来完成这个工作。`this` 指针是一个隐含的指针，它是指向对象本身的，代表了对象的地址。一个类所有的对象调用的成员函数都是同一个代码段，那么，成员函数又是怎么识别属于不同对象的数据成员呢？原来，在对象调用 `pt.input (10,10)` 时，成员函数除了接收 2 个实参外，还接收到了 `pt` 对象的地址，这个地址被一个隐含的形参 `this` 指针所获取，它等同于执行 `this=&pt`。所有对数据成员的访问都隐含地被加上了前缀 `this->`。例如：`x=0`；等价于 `this->x=0`。

利用 `this` 指针，我们重写 `input (int x, int y)` 函数，结果如例 2-10 所示。

例 2-10

```
#include <iostream.h>
class point
{
public:
    int x;
    int y;
    point()
    {
        x=0;
        y=0;
    }
    point(int a,int b)
    {
        x=a;
        y=b;
    }
    void output()
    {
        cout<<x<<endl<<y<<endl;
    }
    void input(int x,int y)
    {
        this->x=x;
        this->y=y;
    }
};
```



```
void main()
{
    point pt(5,5);
    pt.input(10,10);
    pt.output();
}
```

再编译运行，此时的结果就如预期所料了。

2.2.6 类的继承

1. 继承

我们定义一个动物类，对于动物来说，它应该具有吃、睡觉和呼吸的方法。

```
class animal
{
public:
    void eat()
    {
        cout<<"animal eat"<<endl;
    }
    void sleep()
    {
        cout<<"animal sleep"<<endl;
    }
    void breathe()
    {
        cout<<"animal breathe"<<endl;
    }
};
```

我们再定义一个鱼类，对于鱼来说，它也应该具有吃、睡觉和呼吸的方法。

```
class fish
{
public:
    void eat()
    {
        cout<<"fish eat"<<endl;
    }
    void sleep()
    {
        cout<<"fish sleep"<<endl;
    }
    void breathe()
    {
        cout<<"fish breathe"<<endl;
    }
}
```

```
};
```

如果我们再定义一个绵羊类，对于绵羊来说，它也具有吃、睡觉和呼吸的方法，我们是否又重写一遍代码呢？既然鱼和绵羊都是动物，是否可以让鱼和绵羊继承动物的方法呢？在 C++ 中，提供了一种重要的机制，就是继承。类是可以继承的，我们可以基于 animal 这个类来创建 fish 类，animal 称为基类(Base Class，也称为父类)，fish 称为派生类(Derived Class，也称为子类)。派生类除了自己的成员变量和成员方法外，还可以继承基类的成员变量和成员方法。

重写 animal 和 fish 类，让 fish 从 animal 继承，代码如例 2-11 所示(EX05.CPP)。

例 2-11

```
#include <iostream.h>
class animal
{
public:
    void eat()
    {
        cout<<"animal eat"<<endl;
    }
    void sleep()
    {
        cout<<"animal sleep"<<endl;
    }
    void breathe()
    {
        cout<<"animal breathe"<<endl;
    }
};

class fish:public animal
{
};

void main()
{
    animal an;
    fish fh;
    an.eat();
    fh.eat();
}
```

虽然 fish 类没有显式地编写一个方法，但 fish 从 animal 已经继承 eat、sleep、breathe 方法，我们通过编译运行可以看到结果。

下面，我们在 animal 类和 fish 类中分别添加构造函数和析构函数，然后在 main 函数中定义一个 fish 类的对象 fh，看看在构造 fish 类的对象时，animal 类的构造函数是否被调用；如果调用，animal 类和 fish 类的构造函数的调用顺序是怎样的。完整代码如例 2-12 所示(EX06.CPP)。

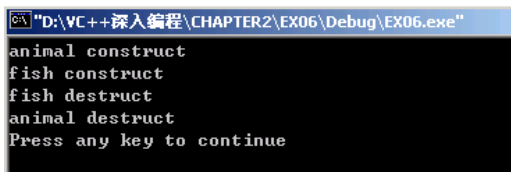
例 2-12

```
#include <iostream.h>
class animal
{
public:
    animal()
    {
        cout<<"animal construct"<<endl;
    }
    ~animal()
    {
        cout<<"animal destruct"<<endl;
    }
    void eat()
    {
        cout<<"animal eat"<<endl;
    }
    void sleep()
    {
        cout<<"animal sleep"<<endl;
    }
    void breathe()
    {
        cout<<"animal breathe"<<endl;
    }
};
class fish:public animal
{
public:
    fish()
    {
        cout<<"fish construct"<<endl;
    }
    ~fish()
    {
        cout<<"fish destruct"<<endl;
    }
};
void main()
{
    fish fh;
}
```

编译运行，出现如图 2.11 所示的结果。

可以看到当构造 fish 类的对象 fh 时，animal 类的构造函数也要被调用，而且在 fish 类的构造函数调用之前被调用。当然，这也很好理解，没有父亲就没有孩子，因为 fish 类从 animal 类继承而来，所以在 fish 类的对象构造之前，animal 类的对象要先构造。在析构

时，正好相反。



```
"D:\VC++深入编程\CHAPTER2\EX06\Debug\EX06.exe"
animal construct
fish construct
fish destruct
animal destruct
Press any key to continue
```

图 2.11 EX06.CPP 程序的运行结果

2. 在子类中调用父类的带参数的构造函数

下面我们修改一下 animal 类的构造函数，增加两个参数 height 和 weight，分别表示动物的高度和重量。代码如例 2-13 所示。

例 2-13

```
#include <iostream.h>
class animal
{
public:
    animal(int height, int weight)
    {
        cout<<"animal construct"<<endl;
    }
    ~animal()
    {
        cout<<"animal destruct"<<endl;
    }
    void eat()
    {
        cout<<"animal eat"<<endl;
    }
    void sleep()
    {
        cout<<"animal sleep"<<endl;
    }
    void breathe()
    {
        cout<<"animal breathe"<<endl;
    }
};
class fish:public animal
{
public:
    fish()
    {
        cout<<"fish construct"<<endl;
    }
    ~fish()
```

```
        {
            cout<<"fish destruct"<<endl;
        }
    };
void main()
{
    fish fh;
}
```

当我们编译这个程序时，就会出现如下错误：

EX06.CPP(29) : error C2512: 'animal' : no appropriate default constructor available

那么这个错误是如何出现的呢？当我们构造 fish 类的对象 fh 时，它需要先构造 animal 类的对象，调用 animal 类的默认构造函数（即不带参数的构造函数），而在我们的程序中，animal 类只有一个带参数的构造函数，在编译时，因找不到 animal 类的默认构造函数而出错。

因此，在构造 fish 类的对象时（调用 fish 类的构造函数时），要想办法去调用 animal 类的带参数的构造函数，那么，我们如何在子类中向父类的构造函数传递参数呢？可以采用如例 2-14 所示的方式，在构造子类时，显式地去调用父类的带参数的构造函数。

例 2-14

```
#include <iostream.h>
class animal
{
public:
    animal(int height, int weight)
    {
        cout<<"animal construct"<<endl;
    }
    ...
};
class fish:public animal
{
public:
    fish():animal(400,300)
    {
        cout<<"fish construct"<<endl;
    }
    ...
};
void main()
{
    fish fh;
}
```

注意程序中以粗体显示的代码。在 fish 类的构造函数后，加一个冒号（:），然后加上

父类的带参数的构造函数。这样，在子类的构造函数被调用时，系统就会去调用父类的带参数的构造函数去构造对象。这种初始化方式，还常用来对类中的常量（const）成员进行初始化，如下面的代码所示：

```
class point
{
public:
    point():x(0),y(0)
private:
    const int x;
    const int y;
};
```

当然，类中普通的成员变量也可以采取此种方式进行初始化，然而，这就没有必要了。

3. 类的继承及类中成员的访问特性

在类中还有另外一种成员访问权限修饰符：protected。下面是 public ,protected ,private 三种访问权限的比较：

■ public 定义的成员可以在任何地方被访问。

■ protected 定义的成员只能在该类及其子类中访问。

■ private 定义的成员只能在该类自身中访问。

对于继承，也可以有 public、protected 或 private 这三种访问权限去继承基类中的成员，例如，例 2-14 所示代码中，fish 类继承 animal 类，就是采用 public 的继承方式。如果在定义派生类时没有指定如何继承访问权限，则默认为 private。如果派生类以 private 访问权限继承基类，则基类中的成员在派生类中都变成了 private 类型的访问权限。如果派生类以 public 访问权限继承基类，则基类中的成员在派生类中仍以原来的访问权限在派生类中出现。如果派生类以 protected 访问权限继承基类，则基类中的 public 和 protected 成员在派生类中都变成了 protected 类型的访问权限。



注意：基类中的 private 成员不能被派生类访问，因此，private 成员不能被派生类所继承。

4. 多重继承

如同该名字中所描述的，一个类可以从多个基类中派生。在派生类由多个基类派生的多重继承模式中，基类是用基类表语法成分来说明的，多重继承的语法与单一继承很类似，只需要在声明继承的多个类之间加上逗号来分隔，定义形式为：

```
class 派生类名：访问权限 基类名称，访问权限 基类名称，访问权限 基类名称
{
    .....
};
```

例如 B 类是由类 C 和类 D 派生的，可按如下方式进行说明：

```
class B：public C, public D
```

```
{  
    .....  
}
```

基类的说明顺序一般没有重要的意义，除非在某些情况下要调用构造函数和析构函数时，在这样的情况下，会有一些影响。

- 由构造函数引起的初始化发生的顺序。如果你的代码依赖于 B 的 D 部分要在 C 部分之前初始化，则此说明顺序将很重要，你可以在继承表中把 D 类放到 C 类的前面。初始化是按基类表中的说明顺序进行初始化的。
- 激活析构函数以做清除工作的顺序。同样，当类的其他部分正在被清除时，如果某些特别部分要保留，则该顺序也很重要。析构函数的调用是按基类表说明顺序的反向进行调用的。

虽然，多重继承使程序编写更具有灵活性，并且更能真实地反映现实生活，但由此带来的麻烦也不小。我们看例 2-15 所示的程序 (EX07.CPP)：

例 2-15

```
1. #include <iostream.h>  
2. class B1  
3. {  
4. public:  
5.     void output();  
6. };  
7. class B2  
8. {  
9. public:  
10.    void output();  
11. };  
12. void B1::output()  
13. {  
14.     cout<<"call the class B1"<<endl;  
15. }  
16. void B2::output()  
17. {  
18.     cout<<"call the class B2"<<endl;  
19. }  
20.  
21. class A:public B1,public B2  
22. {  
23. public:  
24.     void show();  
25. };  
26. void A::show()  
27. {  
28.     cout<<"call the class A"<<endl;  
29. }  
30. void main()
```

```
31 . {  
32 .     A a;  
    a.output();           //该语句编译时会报错  
33 .     a.show();  
34 . }
```

例 2-15 的程序乍一看，好像没有错误，但是，编译时就会出错。原因何在？由第 21 行代码我们知道派生类 A 是从基类 B1 和 B2 多重继承而来的，而基类 B1 和 B2 各有一个 output() 函数，在第 33 行，当类 A 的对象 a 要使用 a.output() 时，编译器无法确定用户需要的到底是哪一个基类的 output() 函数，而产生 'A::output' is ambiguous 的错误信息，请读者注意。

2.2.7 虚函数与多态性、纯虚函数

1. 虚函数与多态性

因为鱼的呼吸是吐泡泡，和一般动物的呼吸不太一样，所以我们在 fish 类中重新定义 breathe 方法。我们希望如果对象是鱼，就调用 fish 类的 breathe() 方法，如果对象是动物，那么就调用 animal 类的 breathe() 方法。程序代码如例 2-16 所示 (EX08.CPP)。

例 2-16

```
#include <iostream.h>  
class animal  
{  
public:  
    void eat()  
    {  
        cout<<"animal eat"<<endl;  
    }  
    void sleep()  
    {  
        cout<<"animal sleep"<<endl;  
    }  
    void breathe()  
    {  
        cout<<"animal breathe"<<endl;  
    }  
};  
class fish:public animal  
{  
public:  
    void breathe()  
    {  
        cout<<"fish bubble"<<endl;  
    }  
};
```



```

void fn(animal *pAn)
{
    pAn->breathe();
}

void main()
{
    animal *pAn;
    fish fh;
    pAn=&fh;
    fn(pAn);
}

```

我们在 fish 类中重新定义了 breathe() 方法，采用吐泡泡的方式进行呼吸。接着定义了一个全局函数 fn()，指向 animal 类的指针作为 fn() 函数的参数。在 main() 函数中，定义了一个 fish 类的对象，将它的地址赋给了指向 animal 类的指针变量 pAn，然后调用 fn() 函数。看到这里，我们可能会有些疑惑，照理说，C++ 是强类型的语言，对类型的检查应该是非常严格的，但是，我们将 fish 类的对象 fh 的地址直接赋给指向 animal 类的指针变量，C++ 编译器居然不报错。这是因为 fish 对象也是一个 animal 对象，将 fish 类型转换为 animal 类型不用强制类型转换，C++ 编译器会自动进行这种转换。反过来，则不能把 animal 对象看成是 fish 对象，如果一个 animal 对象确实是 fish 对象，那么在程序中需要进行强制类型转换，这样编译才不会报错。

读者可以猜想一下例 2-16 运行的结果，输出的结果应该是“animal breathe”，还是“fish bubble”呢？

运行这个程序，你将看到如图 2.12 所示的结果。

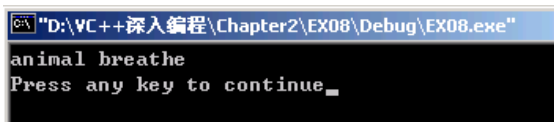


图 2.12 EX09 程序的运行结果（一）

为什么输出的结果不是“fish bubble”呢？这是因为在我们将 fish 类的对象 fh 的地址赋给 pAn 时，C++ 编译器进行了类型转换，此时 C++ 编译器认为变量 pAn 保存就是 animal 对象的地址。当在 fn 函数中执行 pAn->breathe() 时，调用的当然就是 animal 对象的 breathe 函数。

为了帮助读者更好地理解对象类型的转换，我们给出了 fish 对象内存模型，如图 2.13 所示。

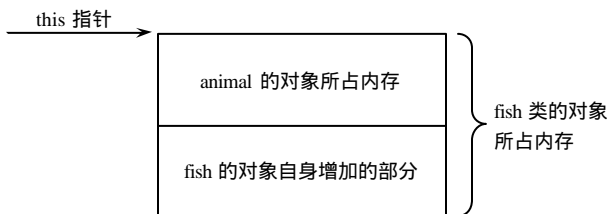


图 2.13 fish 类对象的内存模型

当我们构造 fish 类的对象时，首先要调用 animal 类的构造函数去构造 animal 类的对象，然后才调用 fish 类的构造函数完成自身部分的构造，从而拼接出一个完整的 fish 对象。当我们将 fish 类的对象转换为 animal 类型时，该对象就被认为是原对象整个内存模型的上半部分，也就是图 2.13 中的“animal 的对象所占内存”。当我们利用类型转换后的对象指针去调用它的方法时，自然也就是调用它所在的内存中的方法。因此，出现如图 2.12 所示的结果，也就顺理成章了。

现在我们在 animal 类的 breathe() 方法前面加上一个 virtual 关键字，结果如例 2-17 所示。

例 2-17

```
#include <iostream.h>
class animal
{
public:
    void eat()
    {
        cout<<"animal eat"<<endl;
    }
    void sleep()
    {
        cout<<"animal sleep"<<endl;
    }
    virtual void breathe()
    {
        cout<<"animal breathe"<<endl;
    }
};

class fish:public animal
{
public:
    void breathe()
    {
        cout<<"fish bubble"<<endl;
    }
};

void fn(animal *pAn)
{
    pAn->breathe();
}

void main()
{
    animal *pAn;
    fish fh;
    pAn=&fh;
    fn(pAn);
}
```

}

用 `virtual` 关键字申明的函数叫做虚函数。运行例 2-17 这个程序，结果调用的是 `fish` 类的呼吸方法：

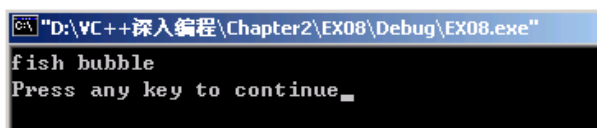


图 2.14 EX08 程序的运行结果（二）

这就是 C++ 中的多态性。当 C++ 编译器在编译的时候，发现 `animal` 类的 `breathe()` 函数是虚函数，这个时候 C++ 就会采用迟绑定（late binding）技术。也就是编译时并不确定具体调用的函数，而是在运行时，依据对象的类型（在程序中，我们传递的 `fish` 类对象的地址）来确认调用的是哪一个函数，这种能力就叫做 C++ 的多态性。我们没有在 `breathe()` 函数前加 `virtual` 关键字时，C++ 编译器在编译时就确定了哪个函数被调用，这叫做早期绑定（early binding）。

C++ 的多态性是通过迟绑定技术来实现的，关于迟绑定技术，读者可以参看相关的书籍，在这里，我们就不深入讲解了。

C++ 的多态性用一句话概括就是：在基类的函数前加上 `virtual` 关键字，在派生类中重写该函数，运行时将会根据对象的实际类型来调用相应的函数。如果对象类型是派生类，就调用派生类的函数；如果对象类型是基类，就调用基类的函数。

2. 纯虚函数

将 `breathe()` 函数申明为纯虚函数，结果如例 2-18 所示。

例 2-18

```
class animal
{
public:
    void eat()
    {
        cout<<"animal eat"<<endl;
    }
    void sleep()
    {
        cout<<"animal sleep"<<endl;
    }
    virtual void breathe() = 0;
};
```

纯虚函数是指被标明为不具体实现的虚成员函数（注意：纯虚函数也可以有函数体，但这种提供函数体的用法很少见）。纯虚函数可以让类先具有一个操作名称，而没有操作内容，让派生类在继承时再去具体地给出定义。凡是含有纯虚函数的类叫做抽象类。这种类不能声明对象，只是作为基类为派生类服务。在派生类中必须完全实现基类的纯虚函数，

否则，派生类也变成了抽象类，不能实例化对象。

纯虚函数多用在一些方法行为的设计上。在设计基类时，不太好确定或将来的行为多种多样，而此行为又是必需的，我们就可以在基类的设计中，以纯虚函数来声明此种行为，而不具体实现它。



注意：C++的多态性是由虚函数来实现的，而不是纯虚函数。在子类中如果有对基类虚函数的覆盖定义，无论该覆盖定义是否有 `virtual` 关键字，都是虚函数。

2.2.8 函数的覆盖和隐藏

1. 函数的覆盖

在上一节介绍多态性的时候，我们给出了下面的代码片段：

例 2-19

```
class animal
{
public:
    ...
    virtual void breathe()
    {
        cout<<"animal breathe"<<endl;
    }
};
class fish:public animal
{
public:
    void breathe()
    {
        cout<<"fish bubble"<<endl;
    }
};
```

在基类 `animal` 的 `breathe` 函数前添加了 `virtual` 关键字，声明该函数为虚函数。在派生类 `fish` 中重写了 `breathe` 函数，我们注意到，`fish` 类的 `breathe` 函数和 `animal` 类的 `breathe` 函数完全一样，无论函数名，还是参数列表都是一样的，这称为函数的覆盖（`override`）。构成函数覆盖的条件为：

- 基类函数必须是虚函数（使用 `virtual` 关键字进行声明）。
- 发生覆盖的两个函数要分别位于派生类和基类中。
- 函数名称与参数列表必须完全相同。

由于 C++ 的多态性是通过虚函数来实现的，所以函数的覆盖总是和多态关联在一起。在函数覆盖的情况下，编译器会在运行时根据对象的实际类型来确定要调用的函数。

2. 函数的隐藏

我们再看例 2-20 的代码：

例 2-20

```
class animal
{
public:
    ...
    void breathe()
    {
        cout<<"animal breathe"<<endl;
    }
};
class fish:public animal
{
public:
    void breathe()
    {
        cout<<"fish bubble"<<endl;
    }
};
```

你看出来这段代码和例 2-19 所示代码的区别了吗？在这段代码中，派生类 fish 中的 breathe 函数和基类 animal 中的 breathe 函数也是完全一样的，不同的是 breathe 函数不是虚函数，这种情况称为函数的隐藏。所谓隐藏，是指派生类中具有与基类同名的函数（不考虑参数列表是否相同），从而在派生类中隐藏了基类的同名函数。

初学者很容易把函数的隐藏与函数的覆盖、重载相混淆，我们看下面两种函数隐藏的情况：

（1）派生类的函数与基类的函数完全相同（函数名和参数列表都相同），只是基类的函数没有使用 virtual 关键字。此时基类的函数将被隐藏，而不是覆盖（请参照上文讲述的函数覆盖进行比较）。

（2）派生类的函数与基类的函数同名，但参数列表不同，在这种情况下，不管基类的函数声明是否有 virtual 关键字，基类的函数都将被隐藏。注意这种情况与函数重载的区别，重载发生在同一个类中。

下面我们给出一个例子，以帮助读者更好地理解函数的覆盖和隐藏，代码如例 2-21 所示。

例 2-21

```
class Base
{
public:
    virtual void fn();
};
class Derived : public Base
```

```
{
public:
    void fn(int);
};

class Derived2 : public Derived
{
public:
    void fn();
};
```

在这个例子中，Derived 类的 fn(int)函数隐藏了 Base 类的 fn()函数，Derived 类 fn(int)函数不是虚函数（注意和覆盖相区别）。Derived2 类的 fn()函数隐藏了 Derived 类的 fn(int)函数，由于 Derived2 类的 fn()函数与 Base 类的 fn()函数具有同样的函数名和参数列表，因此 Derived2 类的 fn()函数是一个虚函数，覆盖了 Base 类的 fn()函数。注意，在 Derived2 类中，Base 类的 fn()函数是不可见的，但这并不影响 fn 函数的覆盖。

当隐藏发生时，如果在派生类的同名函数中想要调用基类的被隐藏函数，可以使用“基类名::函数名（参数）”的语法形式。例如，要在 Derived 类的 fn(int)方法中调用 Base 类的 fn()方法，可以使用 Base::fn()语句。

有的读者可能会想，我怎样才能更好地区分覆盖和隐藏呢？实际上只要记住一点：函数的覆盖是发生在派生类与基类之间，两个函数必须完全相同，并且都是虚函数。那么不属于这种情况的，就是隐藏了。

最后，我们再给出一个例子，留给读者思考，代码如例 2-22 所示（EX09.CPP）。

例 2-22

```
#include <iostream.h>
class Base
{
public:
    virtual void xfn(int i)
    {
        cout<<"Base::xfn(int i)"<<endl;
    }

    void yfn(float f)
    {
        cout<<"Base::yfn(float f)"<<endl;
    }

    void zfn()
    {
        cout<<"Base::zfn()"<<endl;
    }
};
```

```
class Derived : public Base
{
public:
    void xfn(int i) //覆盖了基类的 xfn 函数
    {
        cout<<"Drived::xfn(int i)"<<endl;
    }

    void yfn(int c) //隐藏了基类的 yfn 函数
    {
        cout<<"Drived::yfn(int c)"<<endl;
    }

    void zfn()      //隐藏了基类的 zfn 函数
    {
        cout<<"Drived::zfn()"<<endl;
    }
};

void main()
{
    Derived d;

    Base *pB=&d;
    Derived *pD=&d;

    pB->xfn(5);
    pD->xfn(5);

    pB->yfn(3.14f);
    pD->yfn(3.14f);

    pB->zfn();
    pD->zfn();
}
```

2.2.9 引用

在 C++ 中，还有一个引用的概念。引用就是一个变量的别名，它需要用另一个变量或对象来初始化自身。引用就像一个人的外号一样，例如：有一个人，他的名字叫做张旭，因他在家排行老三，别人给他取了一个外号叫张三，这样，我们叫张三或张旭，指的都是同一个人。下面的代码声明了一个引用 b，并用变量 a 进行了初始化。

```
int a = 5;
int &b = a; //用&表示申明一个引用。引用必须在申明时进行初始化
```

考虑下面代码：

```
int a = 5;
int &b = a;
int c=3;
b=c;           //此处并不是将 b 变成 c 的引用，而是给 b 赋值，此时，b 和 a 的值都变成了 3
```

引用和用来初始化引用的变量指向的是同一块内存，因此通过引用或者变量可以改变同一块内存中的内容。引用一旦初始化，它就代表了一块特定的内存，再也不能代表其他的内存。

那么引用和指针变量有什么区别呢？

引用只是一个别名，是一个变量或对象的替换名称。引用的地址没有任何意义，因此 C++ 没有提供访问引用本身地址的方法。引用的地址就是它所引用的变量或者对象的地址，对引用的地址所做的操作就是对被引用的变量或者对象的地址所做的操作。指针是地址，指针变量要存储地址值，因此要占用存储空间，我们可以随时修改指针变量所保存的地址值，从而指向其他的内存。

引用和指针变量的内存模型如图 2.15 所示。

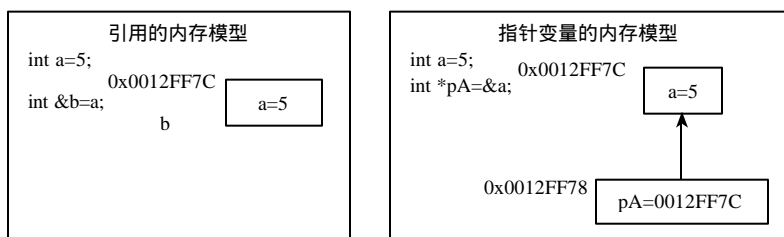


图 2.15 引用和指针变量的内存模型

在编写程序时，很少直接使用引用，即用一个变量来初始化一个引用（`int a; int &b=a`），如果这么做，通过变量和引用都可以修改同一块内存的内容，在程序中，就容易出现问題，不知道此时内存中的值到底是多少了。

引用多数用在函数的形参定义上，在调用函数传参时，我们经常使用指针传递，一是避免在实参占较大内存时发生值的复制，二是完成一些特殊的作用，例如，要在函数中修改实参所指向内存中的内容。同样，使用引用作为函数的形参也能完成指针的功能，在有些情况下还能达到比使用指针更好的效果。

下面，我们以一段程序（如例 2-23 所示）的讲解作为引用这一小节的结束。

例 2-23

```
#include <iostream.h>
//change 函数主要用来交换 a 和 b 的值
void change(int& a,int& b);

void main()
{
    int x=5;
```



```

int y=3;
cout<<"original x="<<x<<endl;
cout<<"original y="<<y<<endl;
change(x,y);    //此处如果用指针传递,则调用 change (&x, &y),这样很容易让人
                迷惑,不知道交换的是 x 和 y 的值,还是 x 和 y 的地址?此处用引用,
                可读性就比指针要好
cout<<"changed x="<<x<<endl;
cout<<"changed y="<<y<<endl;
}
/*在 change()函数的实现中,我们采用了一个小算法,完成了 a 和 b 值的交换,读者下来可以仔
细研读,细细体味一下(读者还可以采用其他的方法,当然也可以直接使用通常的实现,定义一个
临时变量,完成 a 和 b 值的交换)*/
void change(int& a,int& b)
{
    a=a+b;
    b=a-b;
    a=a-b;
}

```

2.2.10 C++类的设计习惯及头文件重复包含问题的解决

在设计一个类的时候,通常是将类的定义及类成员函数的声明放到头文件(即.h文件)中,将类中成员函数的实现放到源文件(即.cpp)中。对于 animal 类需要 animal.h 和 animal.cpp 两个文件,同样,对于 fish 类需要 fish.h 和 fish.cpp。对于 main()函数,我们把它单独放到 EX10.cpp 文件中。

往一个现有工程添加头文件(.h文件)或源文件(.cpp文件)有两种方式:一种是在打开的工程中,单击【File】 【New】,在左边的 Files 标签页下,选择 C++ Header File 或 C++ Source File,然后在右边的 File 文本框中输入头文件或源文件的文件名,如 animal.h 或 animal.cpp,单击【OK】按钮。如图 2.16 所示。

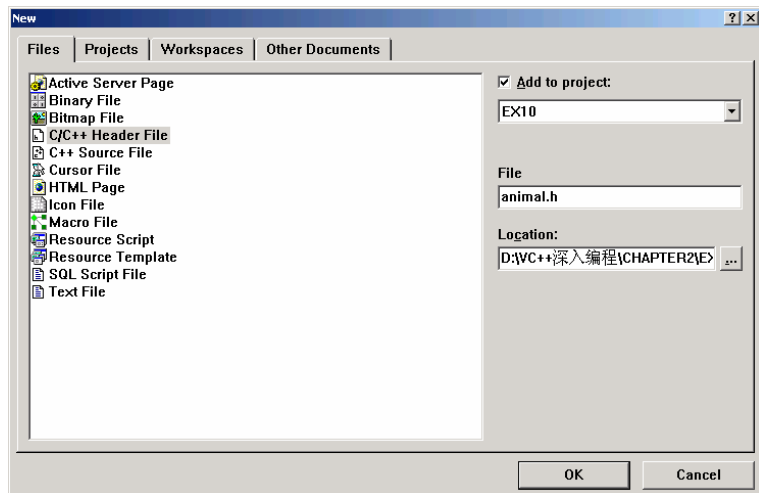


图 2.16 新建头文件或源文件

另一种方式是在 EX10 的工程目录下，单击鼠标右键，选择【新建】 【文本文档】，然后将“新建文本文档.txt”改名为“animal.h”（因.h 和.cpp 文件都是文本格式的文件），依同样的方法，建立 animal.cpp、fish.h、fish.cpp 三个文件，然后在打开的工程中，选择【Project】 【Add To Project】 【Files】，选择 animal.h、animal.cpp、fish.h、fish.cpp 这四个文件，单击【OK】按钮，如图 2.17 所示。

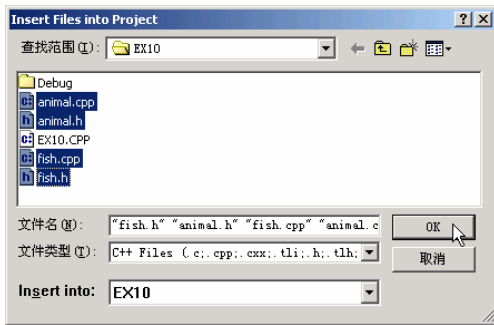


图 2.17 添加头文件和源文件到工程中

代码如例 2-24 所示。

例 2-24

animal.h

```
//在头文件中包含类的定义及类成员函数的声明
```

```
class animal
{
public:
    animal();
    ~animal();
    void eat();
    void sleep();
    virtual void breathe();
};
```

animal.cpp

//在源文件中包含类中成员函数的实现

```
#include "animal.h"
```

```
//因为在编译 animal.cpp 时,编译器不知道 animal 到底
//是什么,所以要包含 animal.h,这样,编译器就知道 animal
//是一种类型的类型
```

```
#include <iostream.h>
```

//在包含头文件时,<和"有什么区别?<和"表示编译器在搜索头文件时的顺序不同,<表示从系统目录下开始搜索,然后再搜索 PATH 环境变量所列出的目录,不搜索当前目录;"是表示先从当前目录搜索,然后是系统目录和 PATH 环境变量所列出的目录。所以如果我们知道头文件在系统目录下,就可以直接用<,这样可以加快搜索速度

```
animal::animal()
```

///`::`叫做作用域标识符，用于指明一个函数属于哪个类或一个数据成员属于哪个类。`::`前面如果不跟类名，表示是全局

```
{
    函数（即非成员函数）或全局数据
}
```

```
animal::~animal()
{
}
```

```
void animal::eat()    //注意：虽然我们在函数体中什么也没写，但仍然是实现了
                     这个函数
```

```
{
}
```

```
void animal::sleep()
{
}
```

```
void animal::breathe()    //注意，在头文件（.h 文件）中加了 virtual 后，在源文
                          件（.cpp 文件）中就不必再加 virtual 了
```

```
{
    cout<<"animal breathe"<<endl;
}
```

```
                                fish.h
#include "animal.h"            //因 fish 类从 animal 类继承而来，要让编译器知道
                                animal 是一种类型的类型，就要包含 animal.h 头文件

class fish:public animal
{
public:
    void breathe();
};
```

```
                                fish.cpp

#include "fish.h"
#include <iostream.h>
void fish::breathe()
{
    cout<<"fish bubble"<<endl;
}
```

```
                                EX10.cpp

#include "animal.h"
#include "fish.h"
void fn(animal *pAn)
{
    pAn->breathe();
}
```

```
}  
void main()  
{  
    animal *pAn;  
    fish fh;  
    pAn=&fh;  
    fn(pAn);  
}
```

现在我们就可以按下键盘上的 F7 功能键编译整个工程了，编译结果如下：

\animal.h(2) : error C2011: 'animal' : 'class' type redefinition

为什么会出现类重复定义的错误呢？请读者仔细查看 EX10.cpp 文件，在这个文件中包含了 animal.h 和 fish.h 这两个头文件。当编译器编译 EX10.cpp 文件时，因为在文件中包含了 animal.h 头文件，编译器展开这个头文件，知道 animal 这个类定义了，接着展开 fish.h 头文件，而在 fish.h 头文件中也包含了 animal.h，再次展开 animal.h，于是 animal 这个类就重复定义了。

读者可以测试一下，如果我们多次包含 iostream.h 这个头文件，也不会出现上面的错误。要解决头文件重复包含的问题，可以使用条件预处理指令。修改后的头文件如下：

	animal.h
#ifndef ANIMAL_H_H	//我们一般用#define 定义一个宏，是为了在程序中使用，使程序更加简洁，维护更加方便，然而在此处，我们只是为了判断 ANIMAL_H_H 是否定义，以此来避免类重复定义，因此我们没有为其定义某个具体的值。在选择宏名时，要选用一些不常用的名字，因为我们的程序经常会跟别人写的程序集成，如果选用一个很常用的名字（例如：x），有可能会造成一些不必要的错误
#define ANIMAL_H_H	
class animal	
{	
public:	
animal();	
~animal();	
void eat();	
void sleep();	
virtual void breathe();	
};	
#endif	

	fish.h
#include "animal.h"	
#ifndef FISH_H_H	
#define FISH_H_H	
class fish:public animal	
{	
public:	
void breathe();	

```
};
#endif
```

我们再看 EX10.cpp 的编译过程。当编译器展开 animal.h 头文件时，条件预处理指令判断 ANIMAL_H_H 没有定义，于是就定义它，然后继续执行，定义了 animal 这个类；接着展开 fish.h 头文件，而在 fish.h 头文件中也包含了 animal.h，再次展开 animal.h，这个时候条件预处理指令发现 ANIMAL_H_H 已经定义，于是跳转到 #endif，执行结束。

通过分析，我们发现在这次的编译过程中，animal 这个类只定义了一次。



提示：Windows 2000 初始安装完毕后，对于已知文件类型的扩展名是隐藏的，例如：“test.txt”这个文件，在资源浏览器中看到是“test”，在这种情况下，修改其文件名为“test.cpp”时，实际的文件名是“test.cpp.txt”，仍然是文本文件。因此在 Win2000 下做开发时，要取消“隐藏已知文件类型的扩展名”这一选项。

操作步骤：在资源浏览器（或我的电脑）中，选择菜单中的“工具->文件夹选项(O)...”，选择“查看”标签页，将滚动栏拖到底端，将“隐藏已知文件类型的扩展名”复选框中的对号(v)取消掉，单击“确定”按钮。

2.2.11 VC++程序编译链接的原理与过程

我们在 EX10 这个工程中，选择菜单中【Build】 【Rebuild All】，重新编译所有的工程文件，可以看到如下输出：

```
Deleting intermediate files and output files for project 'EX10 - Win32 Debug'.
-----Configuration: EX10 - Win32 Debug-----
Compiling...
EX10.CPP
fish.cpp
animal.cpp
Linking...
EX10.exe - 0 error(s), 0 warning(s)
```

从这个输出中，我们可以看到可执行程序 EX10.exe 的产生，经过了两个步骤：首先，C++ 编译器对工程中的三个源文件 EX10.cpp、fish.cpp、animal.cpp 单独进行编译（Compiling...）。在编译时，先由预处理器对预处理指令（#include、#define 和 #if）进行处理，在内存中输出翻译单元（一种临时文件）。编译器接受预处理的输出，将源代码转换成包含机器语言指令的三个目标文件（扩展名为 obj 的文件）：EX10.obj、fish.obj、animal.obj。注意，在编译过程中，头文件不参与编译；在 EX10 工程的 Debug 目录下，我们可以看到编译生成的 obj 文件。接下来是链接过程（Linking...），链接器将目标文件和你所用到的 C++ 类库文件一起链接生成 EX10.exe。整个编译链接的过程如图 2.18 所示。

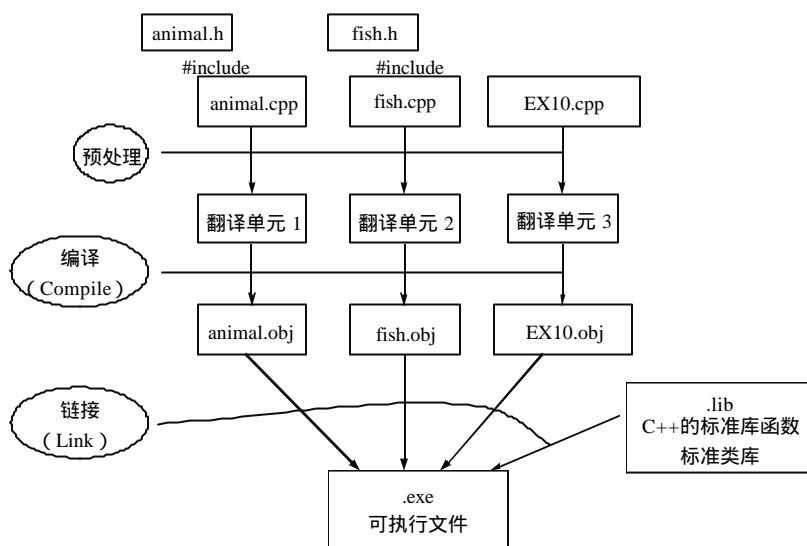


图 2.18 EX10 程序编译链接的过程

好了，到此 C++ 的知识就讲解完毕了。当然 C++ 的内容还有很多，但这一章的内容，对于我们从事 VC++ 开发已经足够了，还有部分 C++ 内容，会在后面的章节中讲解。休息一下，然后继续我们的 VC++ 之旅。

MFC 框架程序剖析

本章将剖析基于 MFC 的框架程序，探讨 MFC 框架程序的内部组织结构。MFC（Microsoft Foundation Class，微软基础类库）是微软为了简化程序员的开发工作所开发的一套 C++ 类的集合，是一套面向对象的函数库，以类的方式提供给用户使用。利用这些类，可以有效地帮助程序员完成 Windows 应用程序的开发。

3.1 MFC AppWizard

MFC AppWizard 是一个辅助我们生成源代码的向导工具，它可以帮助我们自动生成基于 MFC 框架的源代码。该向导的每一个步骤中，我们都可以根据需求来选择各种特性，从而实现定制应用程序。

下面我们就利用 MFC AppWizard 来创建一个基于 MFC 的单文档界面（SDI）应用程序。

1 启动 Microsoft Visual C++ 6.0，单击【File】菜单，选择【New】，在 Projects 选项卡下，选择 MFC AppWizard (exe)，在右侧的【Project name】文本框中，输入我们的工程名：Test，如图 3.1 所示。

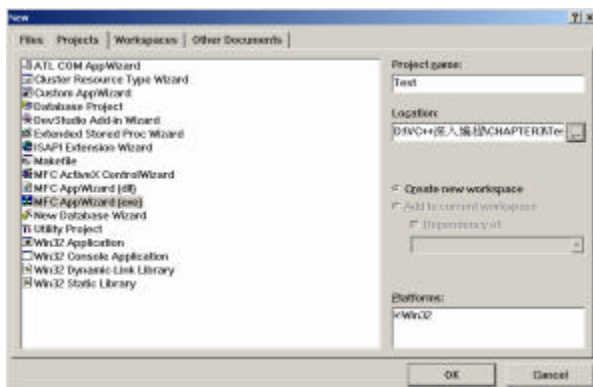


图 3.1 New 对话框

② 单击【OK】按钮，出现 MFC AppWizard-Step 1 对话框，选择 Single document 选项，如图 3.2 所示。

③ 单击【Next】按钮，出现 MFC AppWizard-Step 2 of 6 对话框，保持默认选择，如图 3.3 所示。

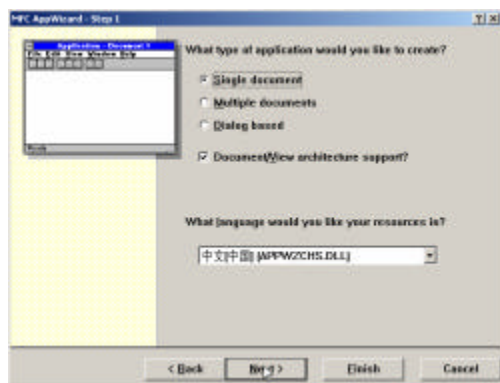


图 3.2 MFC AppWizard - Step 1 对话框

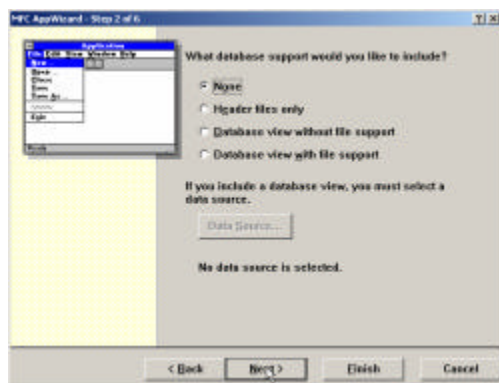


图 3.3 MFC AppWizard - Step 2 of 6 对话框

④ 单击【Next】按钮，出现 MFC AppWizard-Step 3 of 6 对话框，保持默认选择，如图 3.4 所示。

⑤ 单击【Next】按钮，出现 MFC AppWizard-Step 4 of 6 对话框，保持默认选择，如图 3.5 所示。

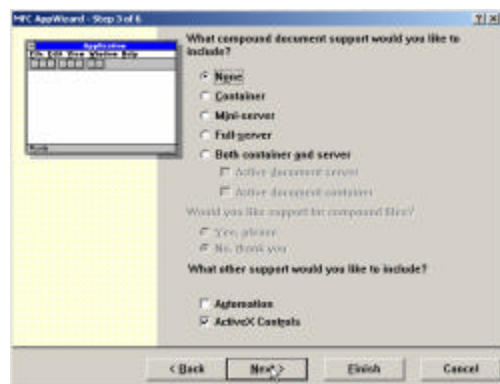


图 3.4 MFC AppWizard - Step 3 of 6 对话框

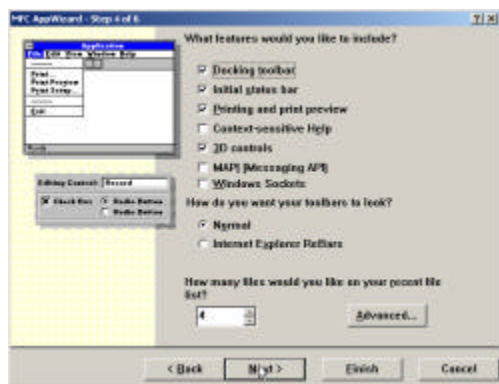


图 3.5 MFC AppWizard - Step 4 of 6 对话框

⑥ 单击【Next】按钮，出现 MFC AppWizard-Step 5 of 6 对话框，保持默认选择，如图 3.6 所示。

⑦ 单击【Next】按钮，出现 MFC AppWizard-Step 6 of 6 对话框，保持默认选择，如图 3.7 所示。

⑧ 单击【Finish】按钮，出现 New Project Information 窗口，如图 3.8 所示。这里需要确认前面几步所做的选择。如果需要修改先前步骤的选择，可以单击 Cancel 按钮返回到上一步。在这里，单击【OK】按钮，MFC AppWizard 就为我们创建一个新工程：Test。

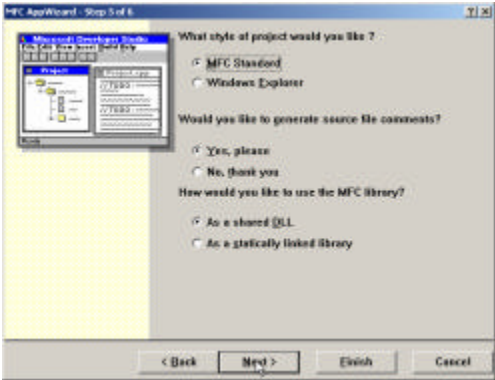


图 3.6 MFC AppWizard - Step 5 of 6 对话框

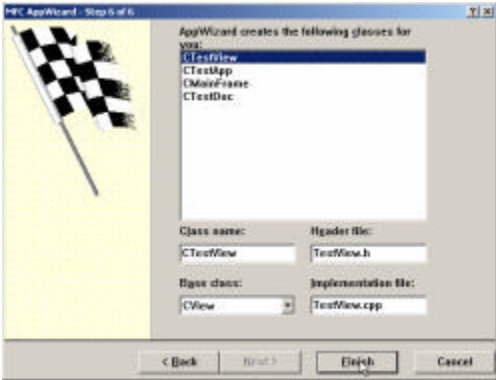


图 3.7 MFC AppWizard - Step 6 of 6 对话框

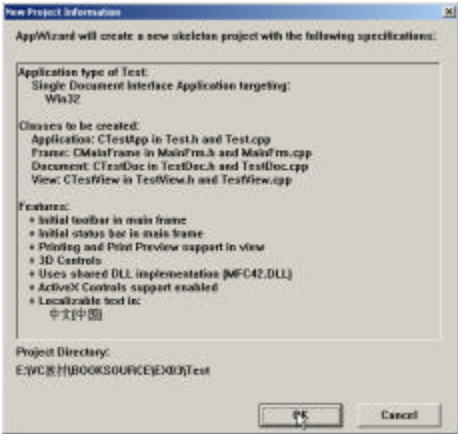


图 3.8 New Project Information 对话框

现在，按下 F7 键编译程序，接着按下 Ctrl+F5 键运行程序，可以看到如图 3.9 所示的运行结果。

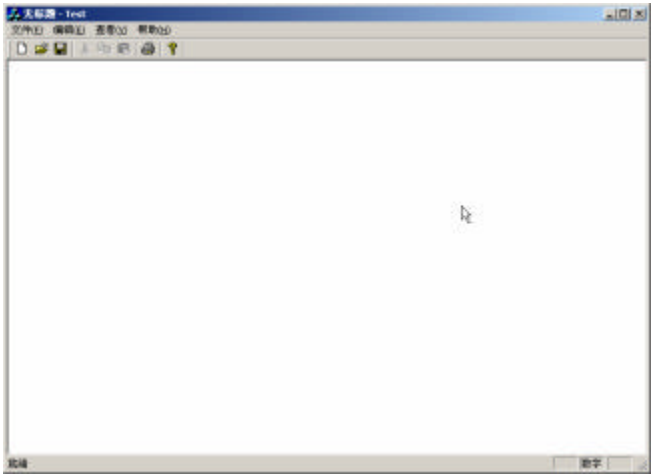


图 3.9 AppWizard 创建的 SDI 程序运行结果

在这个程序中，我们自己没有编写任何代码，就生成了一个带有标题栏，具有最小化框、最大化框，具有系统菜单和一个可调边框的应用程序。这个程序和我们在第 1 章中所创建的程序类似，但比后者多了菜单栏、工具栏以及状态栏。这一切都是通过 MFC AppWizard 生成的。

3.2 基于 MFC 的程序框架剖析

MFC 库是开发 Windows 应用程序的 C++ 接口。MFC 提供了面向对象的框架，程序开发人员可以基于这一框架开发 Windows 应用程序。MFC 采用面向对象设计，将大部分的 Windows API 封装到 C++ 类中，以类成员函数的形式提供给程序开发人员调用。

下面我们看一下 MFC AppWizard 帮助我们生成的这些代码。单击左边工作区窗格中的 ClassView（类视图）标签页，可以看到如图 3.10 所示的五个类。

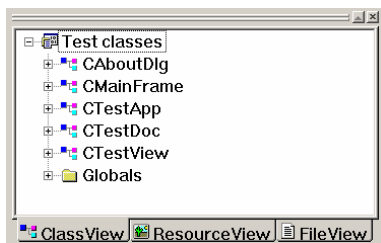


图 3.10 ClassView 标签页



提示：如果要查看某个类提供的信息，可以在 ClassView 标签页上单击该类前面的“+”符号，即可展开该类，显示该类拥有的函数和属性。

在 MFC 中，类的命名都以字母“C”开头，就像 Delphi 中类名以 T 开头，Oracle 的类名以 O 开头一样，当然，这并不是必需的，这只是一种约定。对于一个单文档应用程序（即我们在创建工程时第二步所选的 Single document），都有一个 CMainFrame 类，和一个以“C+工程名+App”为名字类、一个以“C+工程名+Doc”为名字类、一个以“C+工程名+View”为名字类。作为读者，在刚接触 MFC 的程序时，一定要逐步熟悉 MFC AppWizard 所生成的这几个类，以及类中的代码。这样才能在阅读程序时，知道哪些类、哪些代码是向导生成的，哪些类、哪些代码是我们自己编写的。

在 ClassView 标签页中的类名上双击，右边工作区窗格就会打开定义该类的头文件。我们可以发现 ClassView 标签页中的这五个类都有一个基类，例如，CTestView 派生于 CView；CMainFrame 派生于 CFrameWnd.....这些基类都是 MFC 中的类，可以查看一下这些基类的帮助信息。



提示：如果想查看某个类或函数的帮助，可以把当前光标放在该类或函数所在位置，然后按 F1 键，即可打开 MSDN 中相应帮助。在 MSDN 帮助页中每个类的说明页底部都有一个“Hierarchy Chart”超链接，单击此链接，即可看到整个 MFC 类的组织结构图。

图 3.11 是 MFC 类组织结构图中的一部分，可以发现 CFrameWnd 是由 CWnd 派生的。另外，也可以发现从 CWnd 派生的还有 CView 类。这就说明这个程序中的 CMainFrame 类和 CTestView 类追本溯源有一个共同的基类：CWnd 类。CWnd 类是 MFC 中一个非常重要的类，它封装了与窗口相关的操作。

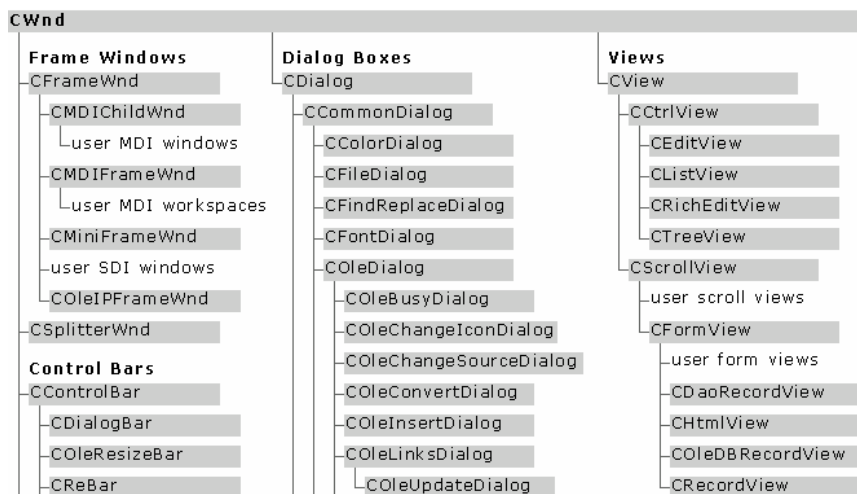


图 3.11 部分 MFC 类组织结构图

3.2.1 MFC 程序中的 WinMain 函数

读者还记得我们在第 2 章中讲述的创建 Win32 应用程序的几个步骤吗？当时，我们介绍 Win32 应用程序有一条很明确的主线：首先进入 WinMain 函数，然后设计窗口类、注册窗口类、产生窗口、注册窗口、显示窗口、更新窗口，最后进入消息循环，将消息路由到窗口过程函数中去处理。遵循这条主线，我们在写程序时就有了一条很清晰的脉络。

但在编写 MFC 程序时，我们找不到这样一条主线，甚至在程序中找不到 WinMain 函数。可以在当前 Test 工程中查找 WinMain 函数，方法是在 VC++ 开发环境中单击【Edit】菜单，选择【Find in Files...】菜单项，并在弹出的查找对话框中“Find What:”文本框内输入“WinMain”，单击【Find】按钮，结果当然是找不到 WinMain 函数。读者可以在这个工程中，再查找一下 WNDCLASS、CreateWindow 等，你会发现仍然找不到。那么是不是 MFC 程序就不需要 WinMain 函数，不需要设计窗口类，不需要创建窗口了呢？当然不是。我们之所以看不见这些，是因为微软在 MFC 的底层框架类中封装了这些每一个窗口应用程序都需要的步骤，目的主要是为了简化程序员的开发工作，但这也给我们在学习和掌握 MFC 程序时造成了很多不必要的困扰。

为了更好地学习和掌握基于 MFC 的程序，有必要对 MFC 的运行机制，以及封装原理有所了解。在第 1 章就讲述了 WinMain 函数是所有 Win32 程序的入口函数，就像 DOS 下的 main 函数一样。我们创建的这个 MFC 程序也不例外，它也有一个 WinMain 函数，但这个 WinMain 函数是在程序编译链接时，由链接器将该函数链接到 Test 程序中的。

在安装完 Microsoft Visual Studio 6.0 后，在安装目录下（将 Microsoft Visual Studio 6.0

安装到了 D:\Program Files 下), 微软提供了部分 MFC 的源代码, 我们可以跟踪这些源代码, 来找出程序运行的脉络。机器上 MFC 源代码的具体路径为 D:\Program Files\Microsoft Visual Studio\VC98\MFC\ SRC ,读者可以根据这个目录结构在自己机器上查找相应的目录。找到相应的目录后, 在资源浏览器的工具栏上选择“搜索”。然后在搜索窗口的“包含文字”文本框中输入“WinMain”, 单击“立即搜索(S)”按钮, 搜索结果如图 3.12 所示。

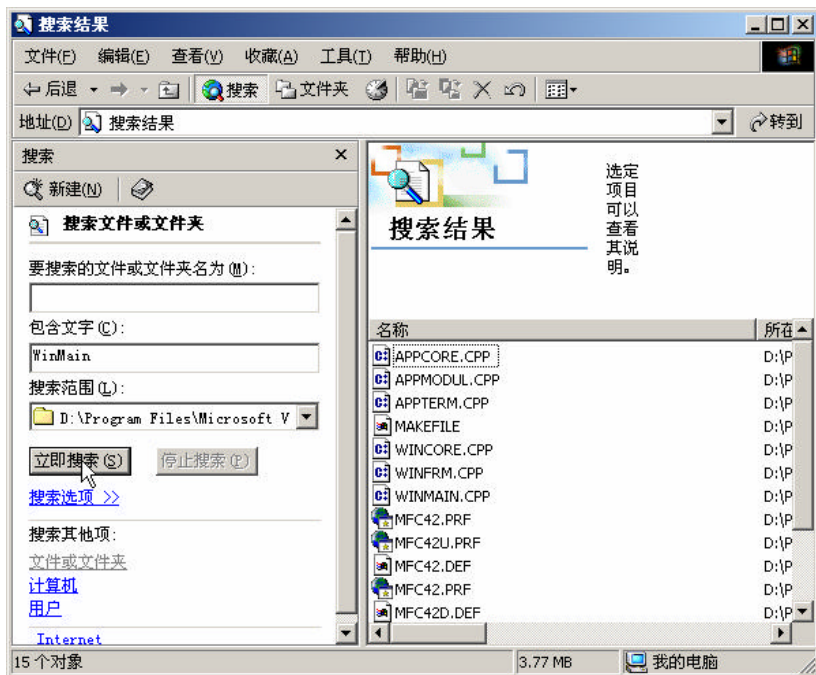


图 3.12 包含“WinMain”文字的搜索结果

我们只需要查看后缀名为 CPP 的源文件即可, 实际上, WinMain 函数在 APPMODUL.CPP 这个文件中。保持 Test 工程的打开状态, 然后双击 APPMODUL.CPP 即可在 VC++ 环境中打开该文件, 在其中可以找到如例 3-1 所示的这段代码。

例 3-1

```
extern "C" int WINAPI
_tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPCTSTR lpCmdLine, int nCmdShow)
{
    // call shared/exported WinMain
    return AfxWinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow);
}
```

WinMain 函数找到了。现在我们可以看看 Test 程序是否会进入这个 WinMain 函数。在 WinMain 函数中按下 F9 键设置一个断点, 然后按下 F5 键调试运行当前程序。我们发现程序确实运行到该断点处停了下来, 如图 3.13 所示。这说明 Test 这个 MFC 程序确实有 WinMain 函数, 在程序编译链接时, WinMain 函数就成为该程序的一部分。

```

// export WinMain to force linkage to this module

extern int AFXAPI AfxWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPTSTR lpCmdLine, int nCmdShow);

extern "C" int WINAPI
_tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPTSTR lpCmdLine, int nCmdShow)
{
    // call shared/exported WinMain
    return AfxWinMain(hInstance, hPrevInstance, lpCmdLine, nCmdShow);
}

// initialize app state such that it points to this module's core state

BOOL AFXAPI AfxInitialize(BOOL bDLL, DWORD dwVersion)
{
    AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
    pModuleState->m_bDLL = (BYTE)bDLL;
    ASSERT(dwVersion <= _MFC_VER);
    UNUSED(dwVersion); // not used in release build
#ifdef _AFXDLL
    pModuleState->m_dwVersion = dwVersion;
#endif
}

```

图 3.13 程序运行到 WinMain 断点处

但这个_tWinMain 函数和第 1 章所讲的 WinMain 函数有些不同,让我们先看看这个函数的定义。读者可以在_tWinMain 上单击鼠标右键,从弹出的快捷菜单中选择【Go To Definition Of _tWinMain】菜单项,光标就会定位到_tWinMain 函数的定义处,代码如例 3-2 所示,从中我们可以发现_tWinMain 实际上是一个宏,展开之后就是 WinMain 函数。

例 3-2

```

#define _tmain      main
#define _tWinMain   WinMain
#ifdef _POSIX_
#define _tenviron   environ
#else
#define _tenviron   _environ
#endif
#define __targv     __argv

```

1. theApp 全局对象

找到了 WinMain 函数,那么它是如何与 MFC 程序中的各个类组织在一起的呢?也就是说,MFC 程序中的类是如何与 WinMain 函数关联起来的呢?

双击 ClassView 标签页中的 CTestApp 类,跳转到该类的定义文件(Test.h)中。可以发现 CTestApp 派生于 CWinApp 类,后者表示应用程序类。我们在 ClassView 标签页中打开 CTestApp 类前面的“+”符号,双击该类的构造函数,就跳转到该类的源文件(Test.cpp)中。在 CTestApp 构造函数处设置一个断点,然后调试运行 Test 程序,将发现程序首先停在 CTestApp 类的构造函数处,继续运行该程序。这时程序才进入 WinMain 函数,即停在先前我们在 WinMain 函数中设置的断点处。

在我们通常的理解当中,WinMain 函数是程序的入口函数。也就是说,程序运行时首

先应该调用的是 WinMain 函数，那么这里为什么程序会首先调用 CTestApp 类的构造函数呢？看一下 CTestApp 的源文件，可以发现程序中定义了一个 CTestApp 类型的全局对象：theApp。代码如下。

```
////////////////////////////////////  
// The one and only CTestApp object  
CTestApp theApp;
```



提示：MFC 程序的全局变量都放置在 ClassView 标签页的 Globals 分支下，展开该分支即可看到程序当前所有的全局变量。双击某个全局变量，即可定位到该变量的定义处。

我们在这个全局对象定义处设置一个断点，然后调试运行 Test 程序，将发现程序执行的顺序依次是：theApp 全局对象定义处、TestApp 构造函数，然后才是 WinMain 函数。

为了更好地解释这一过程，我们再新建一个 Win32 控制台工程。单击【File】菜单，选择【New】菜单项，在 Projects 选项卡下，选择 Win32 Console Application 类型，在右侧的 Project name 文本框中输入工程名：main，并将程序放置到适当的位置（即设置 Location 的内容），如图 3.14 所示。

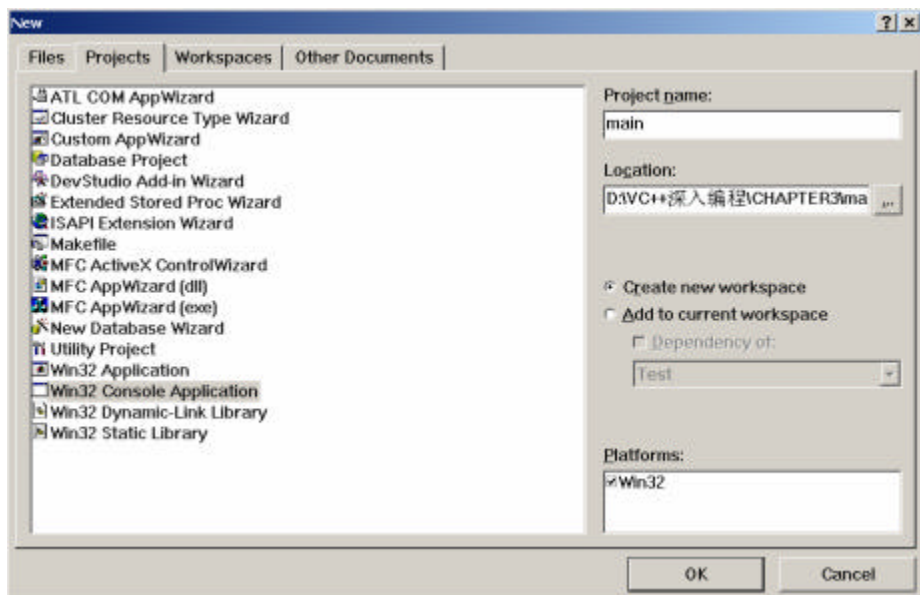


图 3.14 新建 Win32 控制台应用

单击【OK】按钮，进入“Win32 Console Application”向导，选择一个空工程即可，如图 3.15 所示。单击【Finish】按钮，向导就自动生成一个空的 Win32 控制台应用框架。

接着为这个 main 工程新建一个源文件，方法是单击【File】菜单，选择【New】命令，在弹出的【New】对话框中选择【Files】选项卡，然后选择 C++ Source File 项，并在右侧的【File】文本框中输入源文件名：main，如图 3.16 所示。

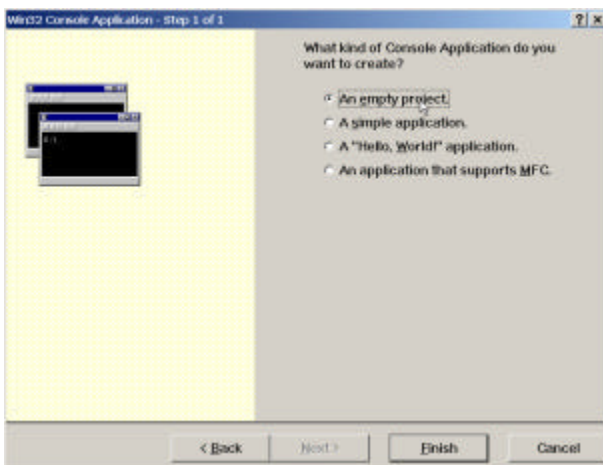


图 3.15 Win32 Console Application 向导

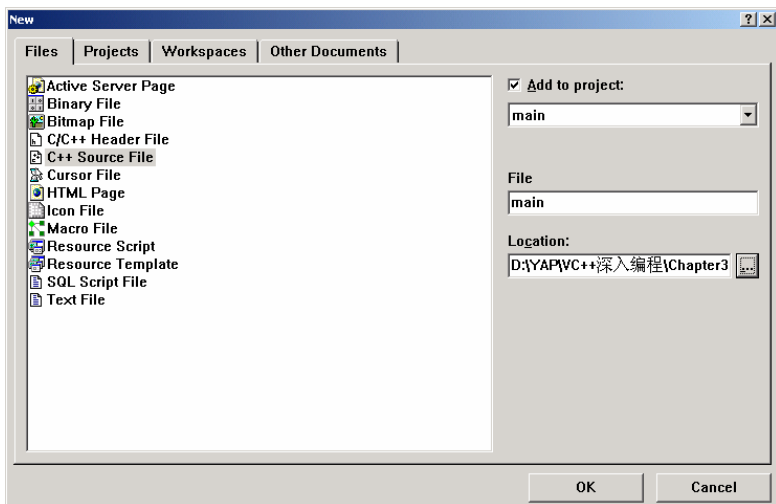


图 3.16 新建一个源文件

接下来，在 main.cpp 文件中输入如例 3-3 所示的代码。

例 3-3

```
#include <iostream.h>

int a=6;

void main()
{
    cout<<a<<endl;
}
```

上述代码非常简单，首先定义了一个 int 类型的全局变量 a，并给它赋了一个初值 6。然后定义了一个 main 函数，该函数所做的工作就是将全局变量 a 的值输出到标准输出 cout

上。因为使用了标准输出，所以需要包含相应的头文件：iostream.h，这是 C++ 中的标准输入输出流头文件。

我们在 main 函数处设置一个断点，调试运行该程序，将会发现程序在进入 main 函数时，a 的值已经是 6 了。也就是说，在程序入口 main 函数加载时，系统就已经为全局变量或全局对象分配了存储空间，并为它们赋了初始值。



小技巧：在程序运行过程中，如果想要查看某个变量的当前值，方法一是把鼠标移到该变量上，停留片刻，VC++ 就会弹出一个窗口，此窗口中显示了该变量的当前值，如图 3.17 所示。

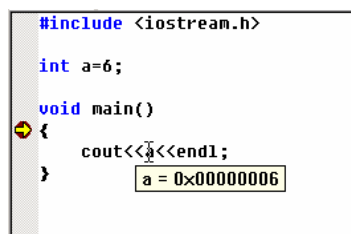


图 3.17 显示当前变量取值的小窗口

方法二是利用 VC++ 提供的调试窗口来查看变量的当前值。操作步骤是单击 View 菜单，选择 Debug Windows 选项，在下拉菜单中选择 Variables 菜单项，即可显示变量窗口，如图 3.18 所示。该窗口显示了程序当前上下文的一些重要变量的当前值。

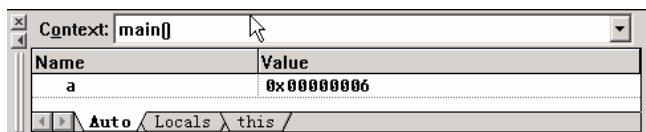


图 3.18 Variables 窗口

接下来，把全局变量 a 换成一个全局对象，看看结果如何。修改如例 3-3 所示的代码，新定义一个 CPoint 类，并定义该类的一个全局变量 pt，结果如例 3-4 所示。

例 3-4

```
1. #include <iostream.h>
```

```
2. //int a=6;
```

```
3. class CPoint
```

```
4. {
```

```
5. public:
```

```
6. CPoint()
```

```
7. {
```

```
8. }
```

```
9. };
```

```
10. CPoint pt;
```



```
11. void main()  
12. {  
13. //  cout<<a<<endl;  
14. }
```

设置三个断点：CPoint 构造函数处（第 6 行代码处）、pt 全局对象定义处（第 10 行代码处）和 main 函数定义处（第 12 行代码处）。选择调试运行 main 函数，将会看到程序代码执行的先后顺序。这时我们将发现 main 程序首先到达 pt 全局对象定义处（第 10 行代码处）；继续运行程序，程序到达 CPoint 类的构造函数（第 6 行代码处）；再继续运行程序，程序到达 main 函数处（第 12 行代码处）。由此可见，无论全局变量，还是全局对象，程序在运行时，在加载 main 函数之前，就已经为全局变量或全局对象分配了内存空间。对一个全局对象来说，此时就会调用该对象的构造函数，构造该对象，并进行初始化操作。

至此，读者应该明白了先前创建的 Test 程序的运行顺序，也就是为什么全局变量 theApp 的构造函数会在 WinMain 函数之前执行了。那么，为什么要定义一个全局对象 theApp，让它在 WinMain 函数之前执行呢？该对象的作用是什么呢？

先关闭 main 工程，返回 Test 程序，并使其处于编辑状态。在前面介绍 Win32 SDK 应用程序时，曾经讲过应用程序的实例是由实例句柄（WinMain 函数的参数 hInstance）来标识的。而对 MFC 程序来说，通过产生一个应用程序类的对象来惟一标识应用程序的实例。每一个 MFC 程序有且仅有一个从应用程序类（CWinApp）派生的类。每一个 MFC 程序实例有且仅有一个该派生类的实例化对象，也就是 theApp 全局对象。该对象就表示了应用程序本身。

我们在第 2 章中阐述了子类构造函数的执行过程，当一个子类在构造之前会先调用其父类的构造函数。因此 theApp 对象的构造函数 CTestApp 在调用之前，会调用其父类 CWinApp 的构造函数，从而就把我们程序自己创建的类与 Microsoft 提供的基类关联起来了。CWinApp 的构造函数完成程序运行时的一些初始化工作。

下面让我们看看 CWinApp 类构造函数的定义。像前面搜索“WinMain”函数那样，找到 Microsoft 提供的 CWinApp 类定义的源文件：appcore.cpp，并在编辑环境中打开，其中 CWinApp 构造函数的代码如例 3-5 所示。

例 3-5

```
CWinApp::CWinApp(LPCTSTR lpszAppName)  
{  
    if (lpszAppName != NULL)  
        m_pszAppName = _tcsdup(lpszAppName);  
    else  
        m_pszAppName = NULL;  
  
    // initialize CWinThread state  
    AFX_MODULE_STATE* pModuleState = _AFX_CMDTARGET_GETSTATE();  
    AFX_MODULE_THREAD_STATE* pThreadState = pModuleState->m_thread;  
    ASSERT(AfxGetThread() == NULL);  
    pThreadState->m_pCurrentWinThread = this;
```

```

ASSERT(AfxGetThread() == this);
m_hThread = ::GetCurrentThread();
m_nThreadId = ::GetCurrentThreadId();

// initialize CWinApp state
ASSERT(afxCurrentWinApp == NULL); // only one CWinApp object please
pModuleState->m_pCurrentWinApp = this;
ASSERT(AfxGetApp() == this);

// in non-running state until WinMain
m_hInstance = NULL;
m_pszHelpFilePath = NULL;
m_pszProfileName = NULL;
m_pszRegistryKey = NULL;
m_pszExeName = NULL;
m_pRecentFileList = NULL;
m_pDocManager = NULL;
m_atomApp = m_atomSystemTopic = NULL;
m_lpCmdLine = NULL;
m_pCmdInfo = NULL;

// initialize wait cursor state
m_nWaitCursorCount = 0;
m_hcurWaitCursorRestore = NULL;

// initialize current printer state
m_hDevMode = NULL;
m_hDevNames = NULL;
m_nNumPreviewPages = 0; // not specified (defaults to 1)

// initialize DAO state
m_lpfndaoTerm = NULL; // will be set if AfxDaoInit called

// other initialization
m_bHelpMode = FALSE;
m_nSafetyPoolSize = 512; // default size
}

```

上述 CWinApp 的构造函数中有这样一句代码：

```
pModuleState->m_pCurrentWinApp = this;
```

根据 C++ 继承性原理，这个 this 对象代表的是子类 CTestApp 的对象，即 theApp。同时，可以发现 CWinApp 的构造函数有一个 LPCTSTR 类型的形参：lpzAppName。但是我们程序中 CTestApp 的构造函数是没有参数的。在第 2 章介绍 C++ 编程知识时，曾经介绍，如果基类的构造函数带有一个形参，那么子类构造函数需要显式地调用基类带参数的构造函数。那么，为什么我们程序中的 CTestApp 构造函数没有这么做呢？

我们知道,如果某个函数的参数有默认值,那么在调用该函数时可以传递该参数的值,也可以不传递,直接使用默认值即可。我们可以在例 3-5 所示代码中的 CWinApp 类名上单击鼠标右键,利用【Go to Definition of CWinApp】命令,定位到 CWinApp 类的定义处,代码如例 3-6 所示。

例 3-6

```
class CWinApp : public CWinThread
{
    DECLARE_DYNAMIC(CWinApp)
public:

    // Constructor
    CWinApp(LPCTSTR lpszAppName = NULL);    // app name defaults to EXE name

    .....
```

从例 3-6 所示代码中,可以看到 CWinApp 构造函数的形参确实有一个默认值(NULL),这样,在调用 CWinApp 类的构造函数时,就不用显式地去传递这个参数的值。

2. AfxWinMain 函数

当程序调用了 CWinApp 类的构造函数,并执行了 CTestApp 类的构造函数,且产生了 theApp 对象之后,接下来就进入 WinMain 函数。根据前面例 3-1 所示代码,可以发现 WinMain 函数实际上是通过调用 AfxWinMain 函数来完成它的功能的。

知识点 Afx 前缀的函数代表应用程序框架 (Application Framework) 函数。应用程序框架实际上是一套辅助我们生成应用程序的框架模型。该模型把多个类进行了一个有机的集成,可以根据该模型提供的方案来设计我们自己的应用程序。在 MFC 中,以 Afx 为前缀的函数都是全局函数,可以在程序的任何地方调用它们。

我们可以采取同样的方式查找定义 AfxWinMain 函数的源文件,在搜索到的文件中双击 WINMAIN.CPP,并在其中找到 AfxWinMain 函数的定义代码,如例 3-7 所示。

例 3-7

```
int AFXAPI AfxWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPCTSTR lpCmdLine, int nCmdShow)
{
    ASSERT(hPrevInstance == NULL);

    int nReturnCode = -1;
    CWinThread* pThread = AfxGetThread();
    CWinApp* pApp = AfxGetApp();

    // AFX internal initialization
    if (!AfxWinInit(hInstance, hPrevInstance, lpCmdLine, nCmdShow))
        goto InitFailure;
```

```

// App global initializations (rare)
if (pApp != NULL && !pApp->InitApplication())
    goto InitFailure;

// Perform specific initializations
if (!pThread->InitInstance())
{
    if (pThread->m_pMainWnd != NULL)
    {
        TRACE0("Warning: Destroying non-NULL m_pMainWnd\n");
        pThread->m_pMainWnd->DestroyWindow();
    }
    nReturnCode = pThread->ExitInstance();
    goto InitFailure;
}
nReturnCode = pThread->Run();

InitFailure:
#ifdef _DEBUG
    // Check for missing AfxLockTempMap calls
    if (AfxGetModuleThreadState()->m_nTempMapLock != 0)
    {
        TRACE1("Warning: Temp map lock count non-zero (%ld).\n",
            AfxGetModuleThreadState()->m_nTempMapLock);
    }
    AfxLockTempMaps();
    AfxUnlockTempMaps(-1);
#endif

    AfxWinTerm();
    return nReturnCode;
}

```

在例 3-7 所示的代码中，AfxWinMain 首先调用 AfxGetThread 函数获得一个 CWinThread 类型的指针，接着调用 AfxGetApp 函数获得一个 CWinApp 类型的指针。从 MFC 类库组织结构图(读者可以按照前面介绍的方法在 MSDN 中找到该结构图)中，可以知道 CWinApp 派生于 CWinThread。例 3-8 是 AfxGetThread 函数的源代码，位于 THREDCORE.CPP 文件中。

例 3-8

```

CWinThread* AFXAPI AfxGetThread()
{
    // check for current thread in module thread state
    AFX_MODULE_THREAD_STATE* pState = AfxGetModuleThreadState();
    CWinThread* pThread = pState->m_pCurrentWinThread;

```

```

// if no CWinThread for the module, then use the global app
if (pThread == NULL)
    pThread = AfxGetApp();

return pThread;
}

```

从例 3-8 所示代码中可以发现，AfxGetThread 函数返回的就是 AfxGetApp 函数的结果。因此，AfxWinMain 函数中的 pThread 和 pApp 这两个指针是一致的。

AfxGetApp 是一个全局函数，定义于 AFXWIN1.INL 中：

```

_AFXWIN_INLINE CWinApp* AFXAPI AfxGetApp()
{ return afxCurrentWinApp; }

```

而 afxCurrentWinApp 的定义位于 AFXWIN.H 文件中，代码如下：

```

#define afxCurrentWinApp AfxGetModuleState()->m_pCurrentWinApp

```

我们返回去看看前面例 3-5 所示的 CWinApp 构造函数代码，就可以知道 AfxGetApp 函数返回的是在 CWinApp 构造函数中保存的 this 指针。对 Test 程序来说，这个 this 指针实际上指向的是 CTestApp 的对象 theApp。也就是说，对 Test 程序来说，pThread 和 pApp 所指向的都是 CTestApp 类的对象，即 theApp 全局对象。

3. InitInstance 函数

再回到例 3-7 所示的 AfxWinMain 函数，可以看到在接下来的代码中，pThread 和 pApp 调用了三个函数（加灰显示的代码行），这三个函数就完成了 Win32 程序所需要的几个步骤：设计窗口类、注册窗口类、创建窗口、显示窗口、更新窗口、消息循环，以及窗口过程函数。pApp 首先调用 InitApplication 函数，该函数完成 MFC 内部管理方面的工作。接着，调用 pThread 的 InitInstance 函数。在 Test 程序中，可以发现从 CWinApp 派生的应用程序类 CTestApp 也有一个 InitInstance 函数，其声明代码如下所示。

```

virtual BOOL InitInstance();

```

从其定义可以知道，InitInstance 函数是一个虚函数。根据类的多态性原理，可以知道 AfxWinMain 函数这里实际调用的是子类 CTestApp 的 InitInstance 函数（读者可以在此函数处设置一个断点，并调试运行程序以验证一下）。CTestApp 类的 InitInstance 函数定义代码如例 3-9 所示。

例 3-9

```

BOOL CTestApp::InitInstance()
{
    AfxEnableControlContainer();

    // Standard initialization
    // If you are not using these features and wish to reduce the size
    // of your final executable, you should remove from the following
    // the specific initialization routines you do not need.

```

```

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

    // Change the registry key under which our settings are stored.
    // TODO: You should modify this string to be something appropriate
    // such as the name of your company or organization.
    SetRegistryKey(_T("Local AppWizard-Generated Applications"));

    LoadStdProfileSettings(); // Load standard INI file options (including
MRU)

    // Register the application's document templates. Document templates
    // serve as the connection between documents, frame windows and views.

    CSingleDocTemplate* pDocTemplate;
    pDocTemplate = new CSingleDocTemplate(
        IDR_MAINFRAME,
        RUNTIME_CLASS(CTestDoc),
        RUNTIME_CLASS(CMainFrame),           // main SDI frame window
        RUNTIME_CLASS(CTestView));
    AddDocTemplate(pDocTemplate);

    // Parse command line for standard shell commands, DDE, file open
    CCommandLineInfo cmdInfo;
    ParseCommandLine(cmdInfo);

    // Dispatch commands specified on the command line
    if (!ProcessShellCommand(cmdInfo))
        return FALSE;

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

    return TRUE;
}

```

3.2.2 MFC 框架窗口

1. 设计和注册窗口

有了 WinMain 函数，根据创建 Win32 应用程序的步骤，接下来应该是设计窗口类和注册窗口类了。MFC 已经为我们预定义了一些默认的标准窗口类，只需要选择所需的窗口类，然后注册就可以了。窗口类的注册是由 AfxEndDeferRegisterClass 函数完成的，该函

数的定义位于 WINCORE.CPP 文件中。其定义代码较长，由于篇幅所限，在这里仅列出部分代码，如例 3-10 所示。

例 3-10

```

BOOL AFXAPI AfxEndDeferRegisterClass(LONG fToRegister)
{
    .....

    // common initialization
    WNDCLASS wndcls;
    memset(&wndcls, 0, sizeof(WNDCLASS)); // start with NULL defaults
    wndcls.lpfnWndProc = DefWindowProc;
    wndcls.hInstance = AfxGetInstanceHandle();
    wndcls.hCursor = afxData.hcurArrow;

    .....

    // work to register classes as specified by fToRegister, populate
    fRegisteredClasses as we go
    if (fToRegister & AFX_WND_REG)
    {
        // Child windows - no brush, no icon, safest default class styles
        wndcls.style = CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
        wndcls.lpszClassName = _afxWnd;
        if (AfxRegisterClass(&wndcls))
            fRegisteredClasses |= AFX_WND_REG;
    }
    if (fToRegister & AFX_WNDOLECONTROL_REG)
    {
        // OLE Control windows - use parent DC for speed
        wndcls.style |= CS_PARENTDC | CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
        wndcls.lpszClassName = _afxWndOleControl;
        if (AfxRegisterClass(&wndcls))
            fRegisteredClasses |= AFX_WNDOLECONTROL_REG;
    }
    .....
    if (fToRegister & AFX_WNDMDIFRAME_REG)
    {
        // MDI Frame window (also used for splitter window)
        wndcls.style = CS_DBLCLKS;
        wndcls.hbrBackground = NULL;
        if (_AfxRegisterWithIcon(&wndcls, _afxWndMDIFrame, AFX_IDI_STD_
MDIFRAME))
            fRegisteredClasses |= AFX_WNDMDIFRAME_REG;
    }
    if (fToRegister & AFX_WNDFRAMEORVIEW_REG)
    {
        // SDI Frame or MDI Child windows or views - normal colors
        wndcls.style = CS_DBLCLKS | CS_HREDRAW | CS_VREDRAW;
        wndcls.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
    }
}

```

```

        if (_AfxRegisterWithIcon(&wndcls, _afxWndFrameOrView, AFX_IDI_STD
_FRAME))
            fRegisteredClasses |= AFX_WNDFRAMEORVIEW_REG;
    }
    .....
}

```

从例 3-10 所示代码可知，`AfxEndDeferRegisterClass` 函数首先判断窗口类的类型，然后赋予其相应的类名（`wndcls.lpszClassName` 变量），这些类名都是 MFC 预定义的。之后就调用 `AfxRegisterClass` 函数注册窗口类，后者的定义也位于 `WINCORE.CPP` 文件中，代码如例 3-11 所示。

例 3-11

```

BOOL AFXAPI AfxRegisterClass(WNDCLASS* lpWndClass)
{
    WNDCLASS wndcls;
    if (GetClassInfo(lpWndClass->hInstance, lpWndClass->lpszClassName,
        &wndcls))
    {
        // class already registered
        return TRUE;
    }

    if (!::RegisterClass(lpWndClass))
    {
        TRACE1("Can't register window class named %s\n",
            lpWndClass->lpszClassName);
        return FALSE;
    }

    if (afxContextIsDLL)
    {
        AfxLockGlobals(CRIT_REGCLASSLIST);
        TRY
        {
            // class registered successfully, add to registered list
            AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
            LPTSTR lpszUnregisterList = pModuleState->m_szUnregisterList;
            // the buffer is of fixed size-- ensure that it does not overflow
            ASSERT(lstrlen(lpszUnregisterList) + 1 +
                lstrlen(lpWndClass->lpszClassName) + 1 <
                _countof(pModuleState->m_szUnregisterList));
            // append classname + newline to m_szUnregisterList
            lstrcat(lpszUnregisterList, lpWndClass->lpszClassName);
            TCHAR szTemp[2];
            szTemp[0] = '\n';
            szTemp[1] = '\0';

```



```

        lstrcat(lpszUnregisterList, szTemp);
    }
    CATCH_ALL(e)
    {
        AfxUnlockGlobals(CRIT_REGCLASSLIST);
        THROW_LAST();
        // Note: DELETE_EXCEPTION not required.
    }
    END_CATCH_ALL
    AfxUnlockGlobals(CRIT_REGCLASSLIST);
}

return TRUE;
}

```

从例 3-11 所示代码可知，AfxRegisterClass 函数首先获得窗口类信息。如果该窗口类已经注册，则直接返回一个真值；如果尚未注册，就调用 RegisterClass 函数注册该窗口类。读者可以看出这个注册窗口类函数与第 2 章介绍的 Win32 SDK 编程中所使用的函数是一样的。



小技巧：如果在当前工程文件中查找某个函数或字符串，可以利用工具栏上的“Find in Files”工具按钮或 Edit 菜单下的 Find in Files 命令；如果在当前文件中查找某个函数或字符串，可以使用 Ctrl+F 快捷键或 Edit 菜单下的 Find 命令。

我们创建的这个 MFC 应用程序 Test，实际上有两个窗口。其中一个是 CMainFrame 类的对象所代表的应用程序框架窗口。该类有一个 PreCreateWindow 函数，这是在窗口产生之前被调用的。该函数的默认实现代码如例 3-12 所示。

例 3-12

```

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    if( !CFrameWnd::PreCreateWindow(cs) )
        return FALSE;
    // TODO: Modify the Window class or styles here by modifying
    // the CREATESTRUCT cs

    return TRUE;
}

```

从其代码可知，该函数首先调用 CFrameWnd 的 PreCreateWindow 函数。后者的定义位于源文件 WINFRM.CPP 中，代码如例 3-13 所示。

例 3-13

```

BOOL CFrameWnd::PreCreateWindow(CREATESTRUCT& cs)
{

```

```

    if (cs.lpszClass == NULL)
    {
        VERIFY(AfxDeferRegisterClass(AFX_WNDFRAMEORVIEW_REG));
        cs.lpszClass = _afxWndFrameOrView; // COLOR_WINDOW background
    }

    if ((cs.style & FWS_ADDTOTITLE) && afxData.bWin4)
        cs.style |= FWS_PREFIXTITLE;

    if (afxData.bWin4)
        cs.dwExStyle |= WS_EX_CLIENTEDGE;

    return TRUE;
}

```

我们发现该函数中调用了 `AfxDeferRegisterClass` 函数，读者可以在 `AFXIMPL.H` 文件中找到后者的定义，定义代码如下：

```
#define AfxDeferRegisterClass(fClass) AfxEndDeferRegisterClass(fClass)
```

由其定义代码可以发现，`AfxDeferRegisterClass` 实际上是一个宏，真正指向的是 `AfxEndDefer-RegisterClass` 函数。根据前面介绍的内容，我们知道这里完成的功能就是注册窗口类。

在 `CMainFrame` 类的 `PreCreateWindow` 函数处设置一个断点，调试运行 Test 程序，将会发现程序在调用 `theApp` 全局对象和 `WinMain` 函数之后，到达此函数处。由此，我们知道 MFC 程序执行的脉络也是在 `WinMain` 函数之后，窗口产生之前注册窗口类的。

2. 创建窗口

按照 Win32 程序编写步骤，设计窗口类并注册窗口类之后，应该是创建窗口了。在 MFC 程序中，窗口的创建功能是由 `CWnd` 类的 `CreateEx` 函数实现的，该函数的声明位于 `AFXWin.h` 文件中，具体代码如下所示。

```

BOOL CreateEx(DWORD dwExStyle, LPCTSTR lpszClassName,
               LPCTSTR lpszWindowName, DWORD dwStyle,
               int x, int y, int nWidth, int nHeight,
               HWND hWndParent, HMENU nIDorHMenu, LPVOID lpParam = NULL);

```

其实现代码位于 `WINCORE.CPP` 文件中，部分代码如例 3-14 所示。

例 3-14

```

BOOL CWnd::CreateEx(DWORD dwExStyle, LPCTSTR lpszClassName,
                    LPCTSTR lpszWindowName, DWORD dwStyle,
                    int x, int y, int nWidth, int nHeight,
                    HWND hWndParent, HMENU nIDorHMenu, LPVOID lpParam)
{
    // allow modification of several common create parameters
    CREATESTRUCT cs;

```

```

        cs.dwExStyle = dwExStyle;
        cs.lpszClass = lpszClassName;
        cs.lpszName = lpszWindowName;
        cs.style = dwStyle;
.....
        if (!PreCreateWindow(cs))
        {
            PostNcDestroy();
            return FALSE;
        }

        AfxHookWindowCreate(this);
        HWND hWnd = ::CreateWindowEx(cs.dwExStyle, cs.lpszClass,
            cs.lpszName, cs.style, cs.x, cs.y, cs.cx, cs.cy,
            cs.hwndParent, cs.hMenu, cs.hInstance, cs.lpCreateParams);
.....
    }

```

在 MFC 底层代码中, CFrameWnd 类的 Create 函数内部调用了上述 CreateEx 函数。而前者又由 CFrameWnd 类的 LoadFrame 函数调用。读者可以自行跟踪这一调用过程。

CFrameWnd 类的 Create 函数的声明也位于 AFXWin.h 文件中, 具体代码如下所示。

```

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

```

其定义位于在 WINFRM.CPP 文件中, 部分代码如例 3-15 所示。

例 3-15

```

BOOL CFrameWnd::Create(LPCTSTR lpszClassName,
                      LPCTSTR lpszWindowName,
                      DWORD dwStyle,
                      const RECT& rect,
                      CWnd* pParentWnd,
                      LPCTSTR lpszMenuName,
                      DWORD dwExStyle,
                      CCreateContext* pContext)
{
.....
    if (!CreateEx(dwExStyle, lpszClassName, lpszWindowName, dwStyle,
        rect.left, rect.top, rect.right - rect.left, rect.bottom - rect.top,
        pParentWnd->GetSafeHwnd(), hMenu, (LPVOID)pContext))
    {

```

```

        TRACE0("Warning: failed to create CFrameWnd.\n");
        if (hMenu != NULL)
            DestroyMenu(hMenu);
        return FALSE;
    }
    .....
}

```

CFrameWnd 类派生于 CWnd 类。从上述声明代码可知，CWnd 类的 CreateEx 函数不是虚函数。另外，CFrameWnd 类中也没有重写这个函数。根据类的继承性原理，CFrameWnd 类就继承了 CWnd 类的 CreateEx 函数。因此，例 3-15 所示 CFrameWnd 类的 Create 函数内调用的实际上就是 CWnd 类的 CreateEx 函数。读者可以在这两个函数的定义处都设置断点，然后调试运行 Test 程序以验证这一点。

再回到例 3-14 所示 CWnd 类的 CreateEx 函数实现代码中，可以发现该函数中又调用了 PreCreateWindow 函数，后者是一个虚函数。因此，这里实际上调用的是子类，即 CMainFrame 类的 PreCreateWindow 函数。之所以在这里再次调用这个函数，主要是为了在产生窗口之前让程序员有机会修改窗口外观，例如，去掉窗口的最大化按钮等，PreCreateWindow 函数的参数就是为了实现这个功能而提供的。该参数的类型是 CREATESTRUCT 结构，我们可以把这个结构体与 CreateWindowEx 函数的参数作一个比较，图 3.19 是 CREATESTRUCT 结构和 CreateWindowEx 函数声明的一个对比，注意左边结构体成员与右边函数参数的对应关系。

typedef struct tagCREATESTRUCT {	HWND CreateWindowEx(
LPVOID lpCreateParams;	DWORD dwExStyle, // extended window style
HANDLE hInstance;	LPCTSTR lpClassName, // pointer to registered class name
HMENU hMenu;	LPCTSTR lpWindowName, // pointer to window name
HWND hwndParent;	DWORD dwStyle, // window style
int cy;	int x, // horizontal position of window
int cx;	int y, // vertical position of window
int y;	int nWidth, // window width
int x;	int nHeight, // window height
LONG style;	HWND hwndParent, // handle to parent or owner window
LPCSTR lpszName;	HMENU hMenu, // handle to menu, or child-window identifier
LPCSTR lpszClass;	HINSTANCE hInstance, // handle to application instance
DWORD dwExStyle;	LPVOID lpParam // pointer to window-creation data
} CREATESTRUCT;);

图 3.19 CREATESTRUCT 结构和 CreateWindowEx 函数定义的对比

可以发现，CREATESTRUCT 结构体中的字段与 CreateWindowEx 函数的参数是一致的，只是先后顺序相反而已。同时，可以看到 PreCreateWindow 函数的这个参数是引用类型。这样，在子类中对此参数所做的修改，在其基类中是可以体现出来的。再看看前面例 3-14 所示 CWnd 类的 CreateEx 函数代码，如果在子类的 PreCreateWindow 函数中修改了 CREATESTRUCT 结构体的值，那么，接下来调用 CreateWindowEx 函数时，其参数就会发生相应的改变，从而就会创建一个符合我们要求的窗口。



知识点 MFC 中后缀名为 Ex 的函数都是扩展函数。

3. 显示窗口和更新窗口

在 Test 程序的应用程序类 (CTestApp) 中有一个名为 m_pMainWnd 的成员变量。该变量是一个 CWnd 类型的指针, 它保存了应用程序框架窗口对象的指针。也就是说, 是指向 CMainFrame 对象的指针。在 CTestApp 类的 InitInstance 函数实现内部有如下两句代码。

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

这两行代码的功能是显示应用程序框架窗口和更新这个窗口。

3.2.3 消息循环

至此, 注册窗口类、创建窗口、显示和更新窗口的工作都已完成, 就该进入消息循环了。CWinThread 类的 Run 函数就是完成消息循环这一任务的, 该函数是在 AfxWinMain 函数中调用的, 调用形式如下 (位于例 3-7 所示 AfxWinMain 函数实现代码的符号 处) 所示。

```
pThread->Run();
```

CWinThread 类的 Run 函数的定义位于 THREDCORE.CPP 文件中, 代码如例 3-16 所示。

例 3-16

```
// main running routine until thread exits
int CWinThread::Run()
{
    ASSERT_VALID(this);

    // for tracking the idle time state
    BOOL bIdle = TRUE;
    LONG lIdleCount = 0;

    // acquire and dispatch messages until a WM_QUIT message is received.
    for (;;)
    {
        // phase1: check to see if we can do idle work
        while (bIdle &&
            !::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE))
        {
            // call OnIdle while in bIdle state
            if (!OnIdle(lIdleCount++))
                bIdle = FALSE; // assume "no idle" state
        }

        // phase2: pump messages while available
        do
        {
            // pump message, but quit on WM_QUIT
```

```

        if (!PumpMessage())
            return ExitInstance();

        // reset "no idle" state after pumping "normal" message
        if (IsIdleMessage(&m_msgCur))
        {
            bIdle = TRUE;
            lIdleCount = 0;
        }

        } while (::PeekMessage(&m_msgCur, NULL, NULL, NULL, PM_NOREMOVE));
    }

    ASSERT(FALSE); // not reachable
}

```

该函数的主要结构是一个 for 循环，该循环在接收到一个 WM_QUIT 消息时退出。在此循环中调用了 PumpMessage 函数，该函数的部分定义代码如例 3-17 所示。

例 3-17

```

BOOL CWinThread::PumpMessage()
{
    ASSERT_VALID(this);

    if (!::GetMessage(&m_msgCur, NULL, NULL, NULL))
    {
        .....
        return FALSE;
    }
    .....
    // process this message
    if (m_msgCur.message != WM_KICKIDLE && !PreTranslateMessage(&m_msgCur))
    {
        ::TranslateMessage(&m_msgCur);
        ::DispatchMessage(&m_msgCur);
    }
    return TRUE;
}

```

可以发现，这与前面第 2 章中讲述的 SDK 编程的消息处理代码是一致的。

3.2.4 窗口过程函数

现在已经进入了消息循环，那么 MFC 程序是否也把消息路由给一个窗口过程函数去处理呢？回头看看例 3-10 所示的 AfxEndDeferRegisterClass 函数的源程序，其中有这样一句代码（符号 所在那行代码）。

```
wndcls.lpfnWndProc = DefWindowProc;
```

这行代码的作用就是设置窗口过程函数，这里指定的是一个默认的窗口过程：DefWindowProc。但实际上，MFC 程序并不是把所有消息都交给 DefWindowProc 这一默认窗口过程来处理的，而是采用了一种称之为消息映射的机制来处理各种消息的。关于该机制将在后面的内容中详细介绍。

至此，我们就了解了 MFC 程序的整个运行机制，实际上与 Win32 SDK 程序是一致的。它同样也需要经过：设计窗口类（只不过 MFC 程序中已经预定义了一些窗口类，我们可以直接使用），注册窗口类，创建窗口，显示并更新窗口，消息循环。

再次调试运行 Test 程序，把 MFC 程序的运行过程再梳理一遍。

- 首先利用全局应用程序对象 theApp 启动应用程序。正是产生了这个全局对象，基类 CWinApp 中的 this 指针才能指向这个对象。如果没有这个全局对象，程序在编译时不会出错，但在运行时就会出错。
- 调用全局应用程序对象的构造函数，从而就会先调用其基类 CWinApp 的构造函数。后者完成应用程序的一些初始化工作，并将应用程序对象的指针保存起来。
- 进入 WinMain 函数。在 AfxWinMain 函数中可以获取子类（对 Test 程序来说，就是 CTestApp 类）的指针，利用此指针调用虚函数：InitInstance，根据多态性原理，实际上调用的是子类（CTestApp）的 InitInstance 函数。后者完成应用程序的一些初始化工作，包括窗口类的注册、创建，窗口的显示和更新。期间会多次调用 CreateEx 函数，因为一个单文档 MFC 应用程序有多个窗口，包括框架窗口、工具条、状态条等。
- 进入消息循环。虽然也设置了默认的窗口过程函数，但是，MFC 应用程序实际上是采用消息映射机制来处理各种消息的。当收到 WM_QUIT 消息时，退出消息循环，程序结束。

3.2.5 文档/视类结构

前面已经提到，我们创建的 MFC 程序除了主框架窗口以外，还有一个窗口是视类窗口，对应的类是 CView 类，CView 类也派生于 CWnd 类。框架窗口是视类窗口的一个父窗口，它们之间的关系如图 3.20 所示。主框架窗口就是整个应用程序外框所包括的部分，即图中粗框以内的内容；而视类窗口只是主框架窗口中空白的地方。

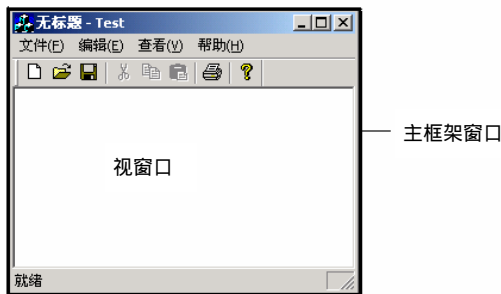


图 3.20 主框架窗口和视窗口之间的关系

可以看到 Test 程序中还有一个 CTestDoc 类,它派生于 CDocument 类。其基类是 CCmdTarget,而后者又派生于 CObject 类,从而,可以知道这个 CTestDoc 类不是一个窗口类,实际上它是一个文档类。

MFC 提供了一个文档/视 (Document/View) 结构,其中文档就是指 CDocument 类,视就是指 CView 类。Microsoft 在设计基础类库时,考虑到要把数据本身与它的显示分离开,于是就采用文档类和视类结构来实现这一想法。数据的存储和加载由文档类来完成,数据的显示和修改则由视类来完成,从而把数据管理和显示方法分离开来。文档/视结构是 MFC 程序的一个重点,后面章节将详细介绍此内容,读者应很好地掌握。

我们回头看看如例 3-9 所示 CTestApp 类的 InitInstance 函数实现代码,可以看到其中定义了一个单文档模板对象指针 (符号所示处的 pDocTemplate 变量)。该对象把文档对象、框架对象、视类对象有机地组织在一起,程序接着利用 AddDocTemplate 函数把这个单文档模板添加到文档模板中,从而把这三个类组织成为一个整体。

3.2.6 帮助对话框类

我们可以发现 Test 程序还有一个 CAboutDlg 类,从其定义可知,其基类是 CDialog 类,该类的派生层次结构如图 3.21 所示,由此可知后者又派生于 CWnd 类。因此,CAboutDlg 类也是一个窗口类。其主要作用是为用户提供一些与程序有关的帮助信息,例如版本号等。该类是一个无关紧要的类,可有可无。在程序运行时,通过单击【帮助\关于 Test...】菜单命令可以显示相应的帮助窗口。其操作命令及运行结果分别如图 3.22 和图 3.23 所示。

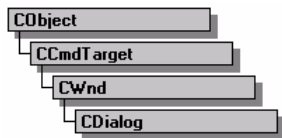


图 3.21 CDialog类的继承结构层次图

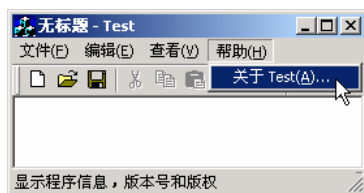


图 3.22 打开帮助窗口类的操作



图 3.23 帮助窗口

3.3 窗口类、窗口类对象与窗口

3.3.1 三者之间关系

很多开发人员都将窗口类、窗口类的对象和窗口之间的关系弄混淆了。为了使读者能更好地理解它们之间的关系,下面我们将模拟 CWnd 类的封装过程。首先新建一个 Win32

Application 类型的工程，取名为“WinMain”。在随后的向导窗口中选择创建一个空工程（即选择 an empty project 选项）。接着为该工程新建一个源文件 WinMain.cpp。在该文件中，首先新建一个类 CWnd，然后为其定义创建窗口函数（CreateEx）、显示窗口函数（ShowWindow）和更新窗口函数（UpdateWindow）三个函数，并定义一个成员变量（m_hWnd）。具体代码如例 3-18 所示。

例 3-18

```
class CWnd
{
public:
    BOOL CreateEx(DWORD dwExStyle,          // extended window style
                  LPCTSTR lpClassName,      // registered class name
                  LPCTSTR lpWindowName,     // window name
                  DWORD dwStyle,            // window style
                  int x,                     // horizontal position of window
                  int y,                     // vertical position of window
                  int nWidth,                // window width
                  int nHeight,              // window height
                  HWND hWndParent,           // handle to parent or owner window
                  HMENU hMenu,              // menu handle or child identifier
                  HINSTANCE hInstance,      // handle to application instance
                  LPVOID lpParam);          // window-creation data

    BOOL ShowWindow(int nCmdShow);
    BOOL UpdateWindow();

public:
    HWND m_hWnd;
};
```



小技巧：这些函数的参数可以参照 MSDN 中相应 MFC 函数的定义，然后直接复制这些参数即可。



提示：因为 SDK 函数数量很多，程序员记忆负担很重。MFC 中使用的大部分函数名与相应的 SDK 函数名相同，这样做的目的就是为了方便程序员，减轻记忆负担。程序员只需要记忆两者中的一个就可以了。

接下来完成这三个函数的定义，代码如例 3-19 所示。

例 3-19

```
BOOL CWnd::CreateEx(DWORD dwExStyle,          // extended window style
                    LPCTSTR lpClassName,      // registered class name
                    LPCTSTR lpWindowName,     // window name
                    DWORD dwStyle,            // window style
                    int x,                     // horizontal position of window
                    int y,                     // vertical position of window
                    int nWidth,                // window width
```

```

        int nHeight,           // window height
        HWND hWndParent,      // handle to parent or owner window
        HMENU hMenu,          // menu handle or child identifier
        HINSTANCE hInstance,  // handle to application instance
        LPVOID lpParam)       // window-creation data
    {
        m_hWnd::CreateWindowEx(dwExStyle, lpClassName, dwStyle, x, y,
                                nWidth, nHeight, hWndParent, hMenu, hInstance,
                                lpParam);
        if(m_hWnd!=NULL)
            return TRUE;
        else
            return FALSE;
    }

    BOOL CWnd::ShowWindow(int nCmdShow)
    {
        return ::ShowWindow(m_hWnd, nCmdShow);
    }

    BOOL CWnd::UpdateWindow()
    {
        return ::UpdateWindow(m_hWnd);
    }

```

其中，我们定义的 CWnd 类的 CreateEx 函数需要完成创建窗口的工作，这可以利用 Win32 提供的 SDK 函数：CreateWindowEx 函数来实现。该函数返回一个句柄，标识它所创建的窗口。这里，我们就可以利用已定义的 CWnd 类的成员变量 m_hWnd 来保存这个窗口句柄。因为我们定义的 CreateEx 函数返回值是个 BOOL 型，所以应该判断一下这个窗口句柄。根据其值是否为空来决定函数是返回 TRUE 值，还是 FALSE 值。

读者应注意的是，在实际开发时，应该初始化 m_hWnd 变量，这可以在构造函数中实现，给它赋一个初值 NULL。这里我们只是为了演示 CWnd 类是如何与窗口关联起来的，因此就不进行初始化工作了。

接下来定义 ShowWindow 函数的实现。同样，需要调用 Platform SDK 函数，即 ShowWindow 来完成窗口的显示。为了区分这两个同名函数，在调用这个 Platform SDK 函数时，前面加上作用域标识符（即::）。这种以“::”开始的表示方法表明该函数是一个全局函数，这里表示调用的 ShowWindow 函数是 Platform SDK 函数。因为 CreateEx 函数已经获取了窗口句柄并保存到 m_hWnd 成员变量中，所以，ShowWindow 函数可以直接把这个句柄变量作为参数来使用。



提示：读者在定义自己的成员函数时，如果调用的 API 函数名与自己的函数名不同，那么该 API 函数名前可以加也可以不加“::”符号，编译器会自动识别 API 函数。但是如果当前定义的成员函数与内部调用的 API 函数名相同，那么后者前面必须加“::”符号，否则程序在编译或运行时就会出错。

我们自己定义的 `UpdateWindow` 函数的实现比较简单，直接调用 SDK 函数：`UpdateWindow` 完成更新窗口的工作。

从例 3-19 所示代码可知，我们定义的 `CWnd` 类的后两个函数（`ShowWindow` 和 `UpdateWindow`）内部都需要一个窗口句柄，即需要知道对哪个窗口进行操作。

现在我们就实现了一个窗口类：`CWnd`。但我们知道如果要以类的方式来完成窗口的创建、显示和更新操作，那么首先还需要编写一个 `WinMain` 函数。读者并不需要记忆这个函数的写法，只要机器上有 MSDN 就可以了，在 MSDN 中找到该函数的帮助文档，直接复制其定义即可。这里，我们只是想讲解在这个函数内部所做的工作，并不是真正的实现，因此只是写出其主要的代码，如例 3-20 所示。

例 3-20

```
int WINAPI WinMain(
    HINSTANCE hInstance,      // handle to current instance
    HINSTANCE hPrevInstance,  // handle to previous instance
    LPSTR lpCmdLine,          // command line
    int nCmdShow               // show state
)
{
    //首先是设计窗口类，即定义一个 WNDCLASS，并为相应字段赋值。
    WNDCLASS wndcls;
    wndcls.cbClsExtra=0;
    wndcls.cbWndExtra=0;
    .....
    //注册窗口类
    RegisterClass(&wndcls);

    //创建窗口
    CWnd wnd;
    wnd.CreateEx(...);

    //显示窗口
    wnd.ShowWindow(SW_SHOWNORMAL);

    //更新窗口
    wnd.UpdateWindow();
    //接下来就是消息循环，此处省略
    .....
    return 0;
}
```

请读者回想一下第 1 章中我们利用 SDK 编程时为创建窗口、显示窗口和更新窗口所编写的代码（如例 3-21 所示），并比较例 3-20 和例 3-21 这两段代码的区别。

例 3-21

```
HWND hwnd;
hwnd=CreateWindowEx();
```

```
::ShowWindow(hwnd, SW_SHOWNORMAL);  
::UpdateWindow(hwnd);
```

我们可以发现，SDK 程序中多了一个 `HWND` 类型的变量 `hwnd`。该变量用来保存由 `CreateWindowEx` 函数创建的窗口句柄，并将其作为参数传递给随后的显示窗口操作（`ShowWindow` 函数）和更新窗口操作（`UpdateWindow` 函数）。而我们自定义的实现代码中，`CWnd` 类定义了一个 `HWND` 类型的成员变量：`m_hWnd`，用于保存这个窗口句柄。首先 `CWnd` 类的 `CreateEx` 函数创建窗口，并将该窗口句柄保存到该成员变量，接着调用 `CWnd` 类的 `ShowWindow` 函数显示窗口时，就不需要再传递这个句柄了，因为它已经是成员变量，该函数可以直接使用它。`CWnd` 类的 `UpdateWindow` 函数也是一样的道理。

许多程序员在进行 MFC 程序开发时，容易混淆一点：认为这里的 `CWnd` 类型的 `wnd` 这个 C++ 对象所代表的就是一个窗口。因为在实践中，他们看到的现象是：当 C++ 窗口类对象销毁时，相应的窗口也就没了。有时正好巧合，当窗口销毁时，C++ 窗口类对象的生命周期也到了，从而也销毁了。正因为如此，许多程序员感觉 C++ 窗口类对象就是窗口，窗口就是这个 C++ 窗口类对象。事实并非如此。读者可以想像一下，如果我们关闭了一个窗口，这个窗口就销毁了，那么该窗口对应的 C++ 窗口类对象销毁了没有呢？当然没有。当一个窗口销毁时，它会调用 `CWnd` 类的 `DestroyWindow` 函数，该函数销毁窗口后，将 `CWnd` 成员变量：`m_hWnd` 设为 `NULL`。

C++ 窗口类对象的生命周期和窗口的生命周期不是一致的。当一个窗口销毁时，与 C++ 窗口类对象没有关系，它们之间的纽带仅仅在于这个 C++ 窗口类内部的成员变量：`m_hWnd`，该变量保存了与这个 C++ 窗口类对象相关的那个窗口的句柄。

另一方面，当我们设计的这个 C++ 窗口类对象销毁的时候，与之相关的窗口是应该销毁的，因为它们之间的纽带（`m_hWnd`）已经断了。另外，窗口也是一种资源，它也占据内存。这样，在 C++ 窗口类对象析构时，也需要回收相关的窗口资源，即销毁这个窗口。

因此，读者一定要注意：C++ 窗口类对象与窗口并不是一回事，它们之间惟一的关系是 C++ 窗口类对象内部定义了一个窗口句柄变量，保存了与这个 C++ 窗口类对象相关的那个窗口的句柄。窗口销毁时，与之对应的 C++ 窗口类对象销毁与否，要看其生命周期是否结束。但 C++ 窗口类对象销毁时，与之相关的窗口也将销毁。在我们定义的这个 `WinMain` 程序（例 3-20 所示代码）中，当程序运行到 `WinMain` 函数的右大括号（`}`）时，该函数内部定义的 `Wnd` 窗口类对象的生命周期也就结束了。

这是我们自己定义的 `CWnd` 类，那么 MFC 提供的 `CWnd` 类是不是这样实现的呢？读者在 MSDN 中查看 MFC 提供的 `CWnd` 类，将会发现该类确实定义了一个数据成员：`m_hwnd`，用来保存与之相关的窗口的句柄。因为 MFC 中所有的窗口类都是由 `CWnd` 类派生的，于是，所有的窗口类（包括子类）内部都有这样的一个成员用来保存与之相关的窗口句柄。所以，读者不能认为我们前面创建的 MFC 程序 `Test` 中的 `CMainFrame` 类和 `CTextView` 类的对象就是一个窗口。

3.3.2 在窗口中显示按钮

为了更好地理解窗口类、窗口类对象和窗口之间的关系，我们接下来实现在窗口中显

示一个按钮这一功能，仍在已有的 Test 程序中实现。首先需要创建一个按钮类对象，按钮对应的 MFC 类是 CButton 类，其继承层次结构如图 3.24 所示，从而可以得知 CButton 类派生于 CWnd 类。

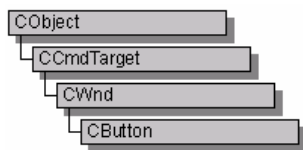


图 3.24 CButton 类的继承层次结构

在 MFC 提供的资源类中，有些类的对象的构造（包括对象构造与初始化）直接通过其构造函数就可以完成。也就是说，这些对象的构造函数包含这个对象的初始化操作。但有些对象的产生除了调用构造函数外，还需要调用其他一些函数来进行初始化的工作，然后才能使用该对象。

对于一个 CButton 对象，在定义之后就可以使用了。但是作为一个窗口类对象，即 CWnd 对象，如果在构造之后还需要产生这个窗口的话，还需要调用 CreateEx 函数来完成初始化工作。也就是说，如果要显示一个按钮的话，在定义这个 CButton 类对象之后，即调用 CButton 类的构造函数之后，还需要调用 CButton 的 Create 函数创建这个按钮窗口，从而把按钮窗口与 CButton 对象关联起来。

CButton 的 Create 函数声明如下。

```

BOOL Create( LPCTSTR lpszCaption, DWORD dwStyle, const RECT& rect, CWnd*
pParentWnd, UINT nID );
  
```

各个参数的意义如下所述。

■ lpszCaption

指定按钮控件的文本。

■ dwStyle

指定按钮控件的风格。按钮控件不仅具有按钮风格类型，还具有窗口风格类型。多种风格类型可以通过位或操作加以组合。

■ rect

指定按钮控件的大小和位置。该参数是 RECT 结构体类型，通过指定左上角和右下角两个点的坐标定义一个矩形。结构体也是一种特殊的类，所以可以用类 CRect 来构造一个 RECT 结构体。

■ pParentWnd

指定按钮控件的父窗口。这是一个 CWnd 类型的指针。MFC 中不再通过窗口句柄，而是通过一个与窗口相关的 C++ 窗口类对象指针来传递窗口对象。

■ nID

指定按钮控件的标识。

为了在框架窗口上产生一个按钮控件，显然应该是在框架窗口产生之后，再创建该按钮控件，否则没有地方放置它。窗口创建时都会产生 WM_CREATE 消息，CMainFrame 类

提供一个 OnCreate 函数，该函数就是用来响应这条窗口创建消息的。该函数的默认实现代码如例 3-22 所示。

例 3-22

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

    if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE
| CBRSTOP
        | CBRSGRIPPER | CBRSTOOLTIPS | CBRSTFLYBY | CBRSSIZE_DYNAMIC)
||
        !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;        // fail to create
    }

    if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
            sizeof(indicators)/sizeof(UINT)))
    {
        TRACE0("Failed to create status bar\n");
        return -1;        // fail to create
    }

    // TODO: Delete these three lines if you don't want the toolbar to
    // be dockable
    m_wndToolBar.EnableDocking(CBRSTALIGN_ANY);
    EnableDocking(CBRSTALIGN_ANY);
    DockControlBar(&m_wndToolBar);

    return 0;
}
```

从例 3-22 所示代码可知，CMainFrame 类的 OnCreate 函数首先调用基类 CFrameWnd 的 OnCreate 函数，创建一个窗口，然后创建工具条（m_wndToolBar）和状态栏（m_wndStatusBar）对象。我们可以在该函数的最后完成按钮的创建工作，即在 return 语句之前添加例 3-23 所示代码中加灰显示的代码。

例 3-23

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    .....
    CButton btn;
    btn.Create("按钮", WS_CHILD | BS_DEFPUSHBUTTON, CRect(0, 0, 100, 100), this,
```

```
123);
```

```
    return 0;
```

```
}
```

其中,将该按钮的名称设置为“按钮”,其位置由 `CRect(0,0,100,100)` 这一矩形确定, ID 号为 123。前面已经讲过,按钮控件不仅具有按钮风格类型,还具有窗口风格类型,因此,在按钮的 `Create` 函数中指定该按钮具有 `WS_CHILD` 窗口风格类型,同时还具有 `BS_DEFPUSHBUTTON` 按钮风格类型,即下按按钮风格。

另外,我们知道每个对象都有一个 `this` 指针,代表对象本身。为了使按钮控件的父窗口就是框架窗口,这里可以直接将代表 `CMainFrame` 对象的 `this` 指针作为参数传递给按钮的 `Create` 函数。

编译并运行 `Test` 程序,但发现按钮并没有显示出来。问题的原因有两个:一是这里定义的 `btn` 对象是个局部对象,当执行到 `OnCreate` 函数的右大括号 `{}` 时,该对象的生命周期就结束了,就会发生析构。前面已经讲过,如果一个窗口与一个 C++ 窗口类对象相关联,当这个 C++ 对象生命周期结束时,该对象在析构时通常会把与之相关联的窗口资源进行回收。这就是说,当执行到例 3-22 所示的 `OnCreate` 函数的右大括号时,刚刚创建的 `btn` 窗口就被与之相关的 C++ 对象销毁了。因此,不能将这个按钮对象定义为一个局部对象。解决方法是将其定义为 `CMainFrame` 类的一个成员变量,可以将其访问权限定义为 `private` 类型以实现信息隐藏。

有多种方法可以定义一个类的成员变量,可以直接在该类的定义中添加成员变量定义代码,也可以利用 VC++ 提供的工具来定义。后者的方法是:在 `ClassView` 标签页中的类名上单击鼠标右键,从弹出的快捷菜单上选择【Add member variable...】菜单命令,将弹出 `Add Member Variable` 对话框。通常,在定义类的成员变量名称时都以“`m_`”为前缀,表明这个变量是类的一个成员变量。在添加成员变量对话框的 `Variable Type` (变量类型) 文本框中输入变量类型 `CButton`, `Variable Name` (变量名称) 文本框中输入按钮对象名称 `m_btn`,并为其选择 `private` 类型的访问权限,如图 3.25 所示。

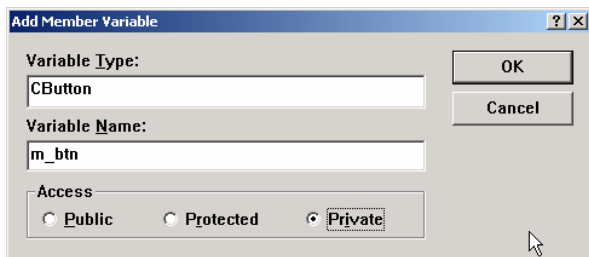


图 3.25 Add Member Variable 对话框

然后单击对话框上的【OK】按钮,即可以在 `CMainFrame` 类的头文件中看到新成员变量的定义,代码如下:

```
private:
    CButton m_btn;
```

修改例 3-23 所示 CMainFrame 类 OnCreate 函数中创建按钮的代码，删除局部按钮对象的定义，并将按钮创建函数的对象名称改为 m_btn，结果如例 3-24 所示。

例 3-24

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    .....
    m_btn.Create("按钮", WS_CHILD | BS_DEFPUSHBUTTON, CRect(0,0,100,100),
this,123);

    return 0;
}
```

再次运行 Test 程序，将会发现按钮还没有出现。这一问题的第二个原因就是在一个窗口创建完成之后，应该将这个窗口显示出来。因此，需要在调用 Create 函数之后再添加一条窗口显示代码，如例 3-25 所示。

例 3-25

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    .....
    1. m_btn.Create("按钮", WS_CHILD | BS_DEFPUSHBUTTON, CRect(0,0,100,100),
this,123);
    2. m_btn.ShowWindow(SW_SHOWNORMAL);

    return 0;
}
```

再次运行 Test 程序，这时就可以看到按钮出现了，如图 3.26 所示。



图 3.26 在框架窗口中显示按钮

根据运行结果，我们可以看到该按钮显示在工具栏上了，这是因为按钮当前的父窗口是 CMainFrame 类窗口，即主框架窗口。该窗口中，标题栏和菜单都位于非客户区，而工具栏位于它的客户区（关于窗口的客户区和非客户区的内容将在下一章讲解）。我们程序中的按钮是在主框架窗口的客户区出现的，并且其位置由 CRect(0,0,100,100) 参数指定，说明其左上角就是其父窗口客户区的(0,0)点，因此，该按钮就在程序的菜单下、工具栏上显示出来了。

读者可以设想一下，如果我们改在 CTestView 类中创建这个按钮，会是什么样的结果

呢？首先，我们把 CMainFrame 中创建按钮的代码（即上述例 3-25 所示代码中第 1 行和第 2 行代码）注释起来，然后为 CTestView 类定义一个 CButton 类型的成员变量 m_btn。但是接下来，我们发现 CTestView 类中没有 OnCreate 函数。我们知道，Windows 下的程序都是基于消息的，无论 MFC 程序，还是 SDK 程序都是这样的。既然窗口在创建时都会产生一个 WM_CREATE 消息，那么就可以让 CTestView 响应这个消息，也就是为这个类添加 WM_CREATE 消息的处理函数。

在 VC++ 中，为一个类添加某个消息的处理函数的方法是：在 ClassView 标签页上，在该类名上单击右键，从弹出的快捷菜单上选择【Add Windows Message Handler...】菜单命令，这时将弹出如图 3.27 所示的添加消息处理函数的窗口。

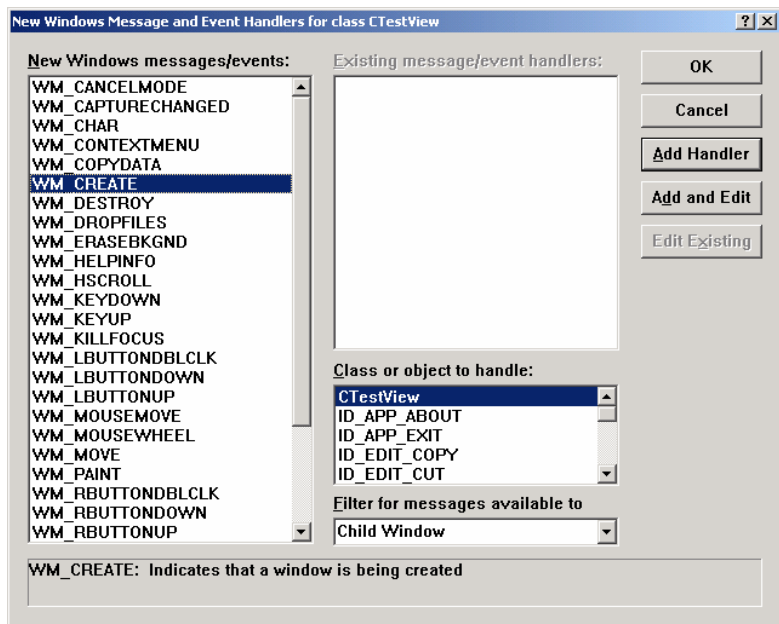


图 3.27 添加消息处理函数的窗口

在该窗口左边的 Windows 消息列表中找到并选中 WM_CREATE 消息，然后单击 Add Handler 按钮，接着再单击 Edit Existing 按钮，或者在选中需要处理的消息之后，直接单击 Add and Edit 按钮。这时，就为 CTestView 类添加了 WM_CREATE 消息的处理函数 OnCreate，并且光标将定位于该函数的定义处。我们就在该函数的尾部添加显示按钮的代码，与 CMainFrame 中的代码相同，可以直接复制过来，结果如例 3-26 所示。

例 3-26

```
int CTestView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    // TODO: Add your specialized creation code here
    m_btn.Create("按钮", WS_CHILD | BS_DEFPUSHBUTTON, CRect(0, 0, 100, 100),
```

```
this,123);  
    m_btn.ShowWindow(SW_SHOWNORMAL);  
  
    return 0;  
}
```

编译并运行 Test 程序，结果如图 3.28 所示。

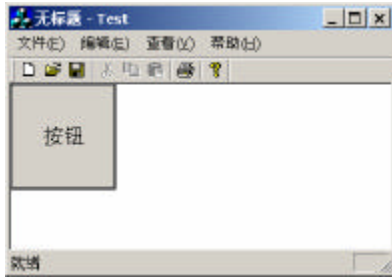


图 3.28 在视窗口中显示按钮

我们可以看到按钮显示出来了，但位置发生了变化。因为这时给按钮的 Create 函数传递的 this 指针指向的是 CTestView 类的对象，因此，这时按钮的父窗口就是视类窗口，所以按钮在视窗口的客户区中显示。如果这时仍想让按钮的父窗口为 CMainFrame 类窗口，即视类窗口的父窗口，可以调用 GetParent 函数来获得视类的父窗口对象的指针，并将该指针传递给按钮的 Create 函数。这时的 CTestView 类 OnCreate 函数定义代码如例 3-27 所示。

例 3-27

```
int CTestView::OnCreate(LPCREATESTRUCT lpCreateStruct)  
{  
    if (CView::OnCreate(lpCreateStruct) == -1)  
        return -1;  
  
    // TODO: Add your specialized creation code here  
    m_btn.Create("按钮",WS_CHILD | BS_DEFPUSHBUTTON, CRect(0,0,100,100),  
GetParent(), 123);  
    m_btn.ShowWindow(SW_SHOWNORMAL);  
  
    return 0;  
}
```

运行 Test 程序，读者会发现按钮的位置与在 CMainFrame 中创建按钮的位置一样，可见按钮的位置与其父窗口有关，而不是与创建它的代码所在的类有关。

另外，如果想在创建按钮之后立即显示，可以将其窗口风格指定为 WS_VISIBLE，这时，就不需要再调用 ShowWindow 函数了。即此时按钮的创建和显示只需要下面这一条代码即可：

```
m_btn.Create("按钮",WS_CHILD | WS_VISIBLE | BS_DEFPUSHBUTTON, CRect(0,0,
```

```
100,100), GetParent(),123);
```



小技巧：Windows 中很多函数名都是一些有意义的单词的组合，并且每个单词的首字母大写。例如，如果想要得到某个类的父窗口，我们可以猜想这个函数名应该是 Get 再加上 ParentWindow 这样的。打开 MSDN 的索引标签页，键入 GetParentWindow，发现没有这个函数，但有一个 GetParent 函数。打开这个函数，发现就是我们所要的函数。在编程时，通过这种方法，可以快速找到所需要的函数。

本例中，我们选择的是 BS_DEFPUSHBUTTON 按钮风格类型，读者可以试着使用其他类型的风格，例如 BS_AUTORADIOBUTTON、BS_CHECKBOX 等，看看结果如何。

通过这个 CButton 对象的创建，希望读者能更好地理解 C++窗口类对象和窗口之间的关系。当我们将按钮窗口销毁，它所对应的 m_btn 这个 C++对象并没有销毁，因为它是 CTestView 类的一个成员变量，它的生命周期与 CTestView 对象是一致的。只要 CTestView 对象没有销毁，该按钮对象就一直存在，在程序中仍可以访问这个对象。

另外，我们发现在调用 CButton 的 ShowWindow 函数时，也没有传递一个窗口句柄，因为 CButton 类是 CWnd 类的子类，因此，它已有一个用于保存窗口句柄的成员变量 m_hwnd。这样，CButton 的成员函数可以直接使用这个变量，并不需要再传递窗口句柄了。

另一点需要注意的是，按钮的父窗口不同，其显示位置也会有所差异。

最后，我们在写程序时，如果不知道某个函数的名称，可以凭感觉利用单词的组合来拼写，通过这种方法一般都能在 MSDN 中找到需要的函数。

3.4 本章小结

本章主要剖析了 MFC 框架的运行机制，可以发现在其框架内部也有与 Win32 SDK 程序相应的操作，包括设计窗口类、注册窗口类、创建窗口、显示和更新窗口、消息循环，以及窗口处理过程，只不过它使用的是一个默认的窗口处理函数。当然，MFC 最终的消息处理是利用消息映射来完成的，这将在后面的章节中介绍。另外，本章还介绍了窗口类的封装过程。我们发现很多窗口类的函数调用都不再需要传递窗口句柄了，因为它们都在内部维护了一个窗口句柄成员变量。