# 动态连接库进阶

# 目　录

# 1 简介

本文是根据“ MSDN ”中的相关资料整理，其目的在于介绍动态连接库（Dynamic Link Library—DLL）的基本概念和为 Microsoft Windows 应用程序编写 DLL 的机制。本论题仅限于使用 Visual C++6.0 编写的 32 位 Windows 应用程序。

# 2 基本概念

## 2.1 静态连接

象 C，Pascal 和 FORTRAN 这样的高级编程语言，一个程序的源代码要经过编译、与不同的库连接，然后生成可执行文件。这些库是包含预编译函数的目标文件，而这些预编译函数能用于完成一般的任务，如计算一个数据的平方根或分配内存。当这些库函数被连接到一个应用程序时，它们就变成应用程序可执行文件的一个固定部分。所有对于库函数的调用都在连接时解析所以称为静态连接。

## 2.2 动态连接

动态连接提供了一种在运行时将应用程序连接到库的机制。库驻留在它们自己的可执行文件中，并不象静态连接那样将代码复制到应用程序的可执行文件中。这些库被称为动态连接库（DLL），以强调它们是在应用程序装载和执行时连接到应用程序，而不是在连接时连接到应用程序的。当应用程序使用一个 DLL 时，操作系统将该 DLL 装载到内存，解析 DLL 中函数的引用，使它们能被应用程序调用，然后在动态连接库不再需要时，操作系统将其从内存中卸载。这种动态连接机制可由应用程序显式的执行或由操作系统隐式的执行。

## 2.3 静态连接库和 Windows　DLL 的区别

Windows　DLL 与静态连接库区别相当大。下面是两者的基本区别：

- 静态连接库驻留在扩展名为.LIB 的文件中，该文件是目标文件的集合；而动态连接库驻留在单独的可执行文件中，该文件只在需要时由 Windows 加载到内存中。
- 每个使用静态连接库的应用程序自身都一个静态连接库的拷贝。而 Windows 支持多个应用程序同时使用同一个 DLL 的拷贝。
- 静态连接库只包含代码和数据，因为它以目标文件集合的形式存储。另一方面，Windows　DLL 可以包含代码、数据和资源，如：位图、图标和光标，因为它们以可执行文件的形式存储。

- 静态连接库必须使用应用程序的数据区，而 DLL 可以（并且经常是）有它们自己的数据地址空间，它们可以（由操作系统）映射到进程的地址空间中。

## 2.4 Window 应用程序与 DLL 的区别

本节先定义一些 Windows 编程中的关键术语，然后说明这些术语相关的概念，最后解释应用程序和 DLL 之间的具体差异。

### 2.4.1 定义

让我们回顾一些 Windows 编程中常用的基本术语。

- *可执行文件*是一个文件，它具有.EXE 或.DLL 扩展名，并为应用程序或 DLL 包含了可执行代码或资源。
- *应用程序*是基于 Windows 的程序，它以.EXE 为文件的扩展名。
- *DLL* 是 Windows 动态连接库，它以.DLL 为文件的扩展名。系统 DLL 可能有.EXE 扩展名，如：User.exe、Gdi.exe、Krnl286.Exe 和 Krnl386.exe。各种设备驱动程序有.DRV 扩展名，例如：Mouse.drv 和 KeyBoard.drv。只有以.DLL 为扩展名的动态连接库才能被操作系统自动加载。如果动态连接库文件具有其它的扩展名，则程序必须通过 LoadLibrary 函数显式的加载该模块。

在进一步深入之前，我们有必要对 DLL 和应用程序如何映射到内存中有一些理解。

Win32 动态连接库的一个最显著的变化是，DLL 的代码和数据在内存中的位置。在 Win32 中，每个应用程序在它自己的 32 位线性地址空间的上下文中运行。在这种方式下，每个应用程序有它自己的私有的地址空间，在这个私有地址空间中代码只能在这个进程中编址（在 Win32 中，每个应用程序的私有地址空间只能由该应用程序的进程引用）。应用程序的所有的代码、数据、资源和动态内存都驻留在应用程序的进程中。进一步说，一个应用程序要将数据或代码编址到自己的进程以外是不可能的。正因如此，当一个 DLL 被加载时，它必须以某种方式驻留在加载它的应用程序的进程中；如果多个应用程序加载了同一个 DLL，它必须驻留在每个应用程序的进程中。

所以，为满足以上的需求（也就是说，能够被重入（对于多个线程同时可存取）和保证 DLL 只有一个拷贝物理的加载进内存），Win32 使用内存映像。通过内存映像，Win32 能够一次性将 DLL 加载到全局堆中，然后将 DLL 的地址范围映射到加载它的每一个应用程序中。图 1 描述了 DLL 如何同时映射到两个不同的应用程序的地址空间中（进程 1、进程 2）。

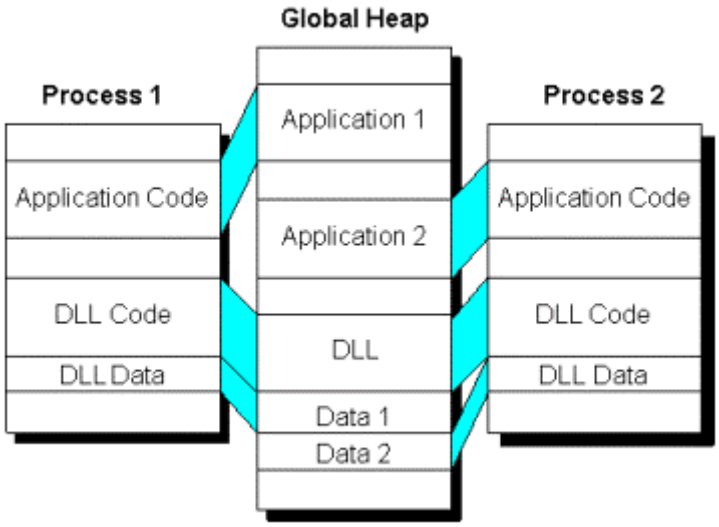


图 1 DLL 同时映射到两个不同的应用程序的地址空间中

注意，DLL 只物理的加载到全局堆一次，而两个应用程序共享 DLL 的代码和资源。

虽然为在不同应用程序中共享的 DLL 编写代码比较容易，但在如何使用 DLL 中的全局变量时仍然存在潜在的冲突。由于 DLL 数据的范围是加载该 DLL 的应用程序进程的地址空间，所以对于多线程应用程序中的每一个线程都可以存取在 DLL 数据集中的全局变量。既然每一个线程都独立于其它的线程执行，那么如果多个线程正在使用同一个全局变量，就会发生潜在的冲突。当在一个能被多个线程存取的 DLL 中使用全局变量时要警惕，应当在必要的地方使用信号量、互斥、事件或临界区。

由于 DLL 被映射到加载它的应用程序进程中，所以在执行 DLL 代码时，谁拥有当前进程可能会存在混乱。当在 DLL 的代码中调用 GetModuleHandle 函数并以空的模块名为参数时，则返回加载它的应用程序的模块句柄，而不是 DLL 的模块句柄。可以在一个静态变量中保存 DLL 模块句柄，这样以便将来存取 DLL 的资源时引用（DLL 模块句柄是传递给 DLL 入口函数的一个参数）。取得 DLL 模块句柄的另一个方法是用 GetModuleHandle 函数，参数为 DLL 的路径和文件名的有效字符串。

## 2.4.2 应用程序和 DLL 的不同

尽管 DLL 和应用程序都是可执行的程序模块，但它们是有区别的。对于最终用户，最明显的区别是 DLL 不是从程序管理器或其它 Shell 程序中直接执行的程序。从系统的角度，应用程序和 DLL 有以下两点基本的区别：

- 应用程序能在系统同时运行多个它自己的实例，而 DLL 只有一个实例。每一个应用程序有它自己的自动数据空间，但是应用程序所有的实例共享一个可执行代码和资源。另一方面，无论 DLL 被加载多少次，它只有一个实例。对于 32 位的操作系统，DLL 每一个加载将有它自己的实例映像；因此，在多个进程上共享数据段是复杂的，对于 32 位 DLL 不推荐使用。
- 应用程序有“拥有”的能力，而 DLL 不能。只有进程才能“拥有”的

能力，且只有应用程序实例是进程。在 32 位操作系统中 DLL 隶属于进程，内存可以从属于单个进程。

DLL 是独立于应用程序的程序模块。在磁盘上，它们驻留在自己特殊的可执行文件中，该文件包含代码、数据和资源（只读数据）如位图和光标。当进程加载 DLL 时，系统将 DLL 代码和数据映射到进程的地址空间中。对于 32 位的操作系统，DLL 没有成为操作系统的一部分，而是成为加载它的进程的一部分。任何由 DLL 中的函数调用产生的内存分配都引起进程地址空间中的内存分配；其它进程不能存取这块内存。由 DLL 分配的全局和静态的变量也不能在 DLL 的多个映像中共享。

当一个进程首次加载 DLL 时，DLL 的使用计数变成 1。如果该进程调用 LoadLibrary 来第二次调用同一个 DLL，该库的引用计数变为 2。如果另一个进程调用 LoadLibrary 来加载被其它进程使用的 DLL，系统为 DLL 代码和数据映射到调用进程的地址空间中将 DLL 引用计数加 1。FreeLibrary 函数减少 DLL 的使用计数；如果引用计数到达 0，DLL 从进程的地址空间映射出。

# 3 动态连接库的优点

DLL 的编译和连接独立于调用它的应用程序。更新 DLL 时，不需要重新编译和连接应用程序。

如果有几个应用程序做为一个系统工作在一起，并且共享通用的 DLL，通过将通用的 DLL 替换为一个增强的版本,整个系统的性能可以被提高。修复 DLL 中的一个 BUG 就修复了所有使用它的应用程序的 BUG。同样，速度的提高或新的功能会使所有使用 DLL 的应用程序受益。

通过在多个应用程序中共享公共代码和资源的单一拷贝，DLL 可以减少内存和磁盘空间的要求。如果多个应用程序使用一个静态连接库，就会有多个不同的静态连接库的拷贝在磁盘上。如果应用程序同时运行，那么在内存中也会有多个不同的拷贝。这些同样的拷贝是冗余的并且浪费空间。如果一个 DLL 替代了静态连接库，那么无论有多少不同的应用程序使用它，只需要有一份代码和资源的拷贝。

# 4 动态连接库的不足

使用 DLL 替代静态连接库的主要缺点是，DLL 开发比较困难。

有时，实际上使用 DLLs 会增加内存的磁盘空间的使用，特别是只有一个应用程序使用该 DLL 时。这种情况发生在应用程序和 DLL 都使用相同的静态连接库函数时，如在使用 C 运行时库函数时。如果 DLL 和应用程序都有一个静态连接库函数连接到它们中，就会有两份库函数的拷贝在内存和磁盘中，这就会浪费空间。为了避免这种情况，函数的发布和函数的调用必须正确的模块化以避免重复或者使用 C 运行时库的 DLL 版本。

在图 2 中，我们有两个应用程序和一个 DLL 使用了静态连接库函数 strlen；另外，这两个应用程序使用动态连接库函数 Foo，该函数也调用了 strlen。注意

所有的三个模块都了 strlen 的代码的拷贝（共有三个静态连接库）。，而 Foo 由两个应用程序使用。

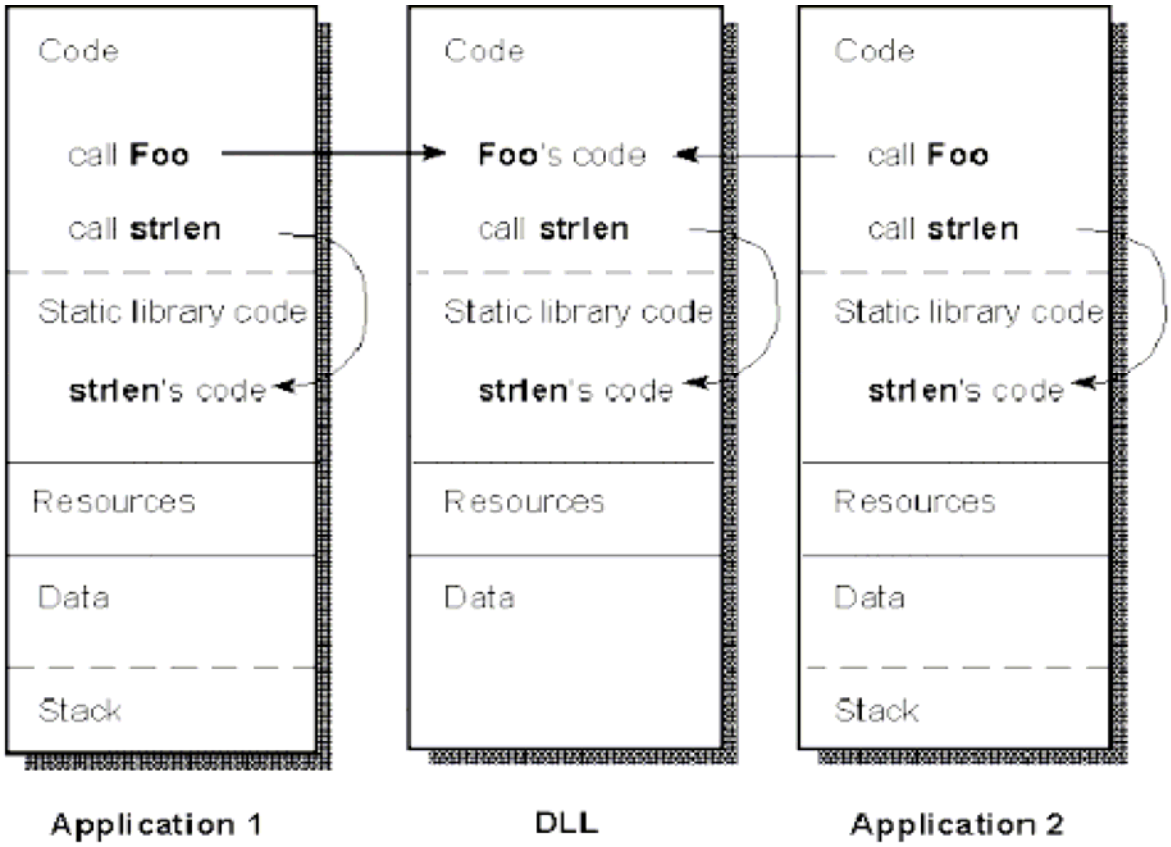


图 2 两个应用程序和一个动态连接使用一个静态连接库函数和一个动态连接库函数

# 5 动态连接库的实现

具备了以上的基本概念，我们可以开始编写动态连接库。一个 DLL 可以由操作系统隐式加载也可由应用程序显式加载。

- 隐式动态连接由操作系统在加载时执行。当操作系统加载一个应用程序或 DLL 时，它先加载所有依赖的 DLL（也就是，由应用程序或 DLL 所调用的 DLL）。
- 显式动态连接在运行时由应用程序或 DLL 通过使用 LoadLibrary 向操作系统请求调用加载 DLL。

## 5.1 实现一个 32 位的 DLL

我们先建立一个静态连接库，然后将它转换为 DLL。如果你以前没有实现过静态连接库，那么你现在将得到这个经验。我们先养成这样一个习惯，即在头文件中编写函数原型，而不是在实现文件（.c 或.cpp）中声明原型。第三方供应

商为使用他们的库的应用程序提供静态连接库或动态连接库，同时也提供应用程序所包含的的头文件。

## 5.2 建立静态连接库

我们将建立的静态连接库中有一个全局变量，局部变量和由应用程序调用的函数，一个局部函数和一个应用程序回调函数。该应用程序是一个控制台应用程序。该库的局部函数原型使用一个头文件，其它共享函数和变量使用另外一个头文件（这个头文件也可由应用程序使用）。

在 VC＋＋的 IDE 中从“ File ”菜单中选择“ New ”。选择“ Projects ”页面然后选择项目类型“ Win32 Static Library ”，点“ OK ”。如图 3

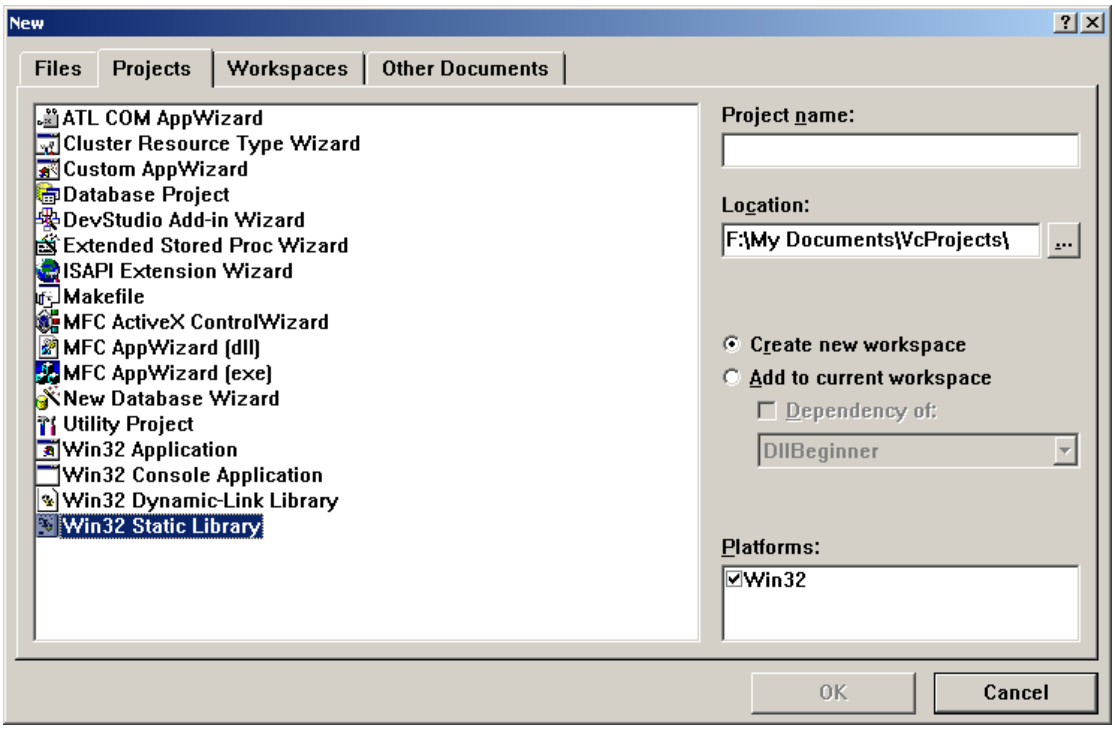


图 3 建立一个静态连接库

建立项目后，建立以下文件。

//该头文件为库输出方法和变量的声明文件。

```
#ifndef _STATIC1_H_
#define _STATIC1_H_
   int libGlobal;
   int CallAppFunc(int *ptrlib);
   void LibFuncA(char * buffer);
   int LibFuncB(int const *mainint);
#endif
```

//该头文件声明库的局部函数。

```
#ifndef _STATIC2_H_
#define _STATIC2_H_
   int LibLocalA(int x,int y);
#endif

//库的实现文件  Static.c

#include <stdio.h>
#include "static1.h"
#include "static2.h"

libGlobal = 20;

void LibFuncA(char * buffer)
{
     int i = 0;
     printf("在库中打印串\n%s\n\n",buffer);
     printf("i 的值为：",i);
}

int LibFuncB(int const * mainint)
{
     int returnvalue;
     int localint;
     int libvalue = 3;
     localint = *mainint;
     returnvalue = LibLocalA(localint,libGlobal);
     libvalue = CallAppFunc(&libvalue);
     printf("打印从应用程序的回调函数的返回值:%d\n",libvalue);
     return returnvalue;
}

int LibLocalA(int x,int y)
{
     return(x + y);
}
```

现在建立这个库的调试版。你会发现在项目的目录下创建一个 Debug 目录。在该目录下，有刚建立的名为“ 库名.Lib ”的静态库。“ 库名 ”是你早先为项目指定的名字。这个库被称为“ 目标库 ”，其中包含了库中函数的目标代码。如果你想知道库函数和变量是在库中是如何安排的，你可以在 DOS 命令提示下使用 Dumpbin 命令来查看，使用方法为：

DUMPBIN /ALL  libname.lib

上述命令的输出结果给出了函数的修饰名字，它对于解决连接器问题如

LNK2001 号错误（不可解析的外部函数）是很有用的。你也可以得到使用的缺省库信息，它对于解决 LNK2005 错误（符号已在其它库中字义）是很有用的。在该库上运行以上命令，查看使用的缺省库和函数及变量是如何列出（或修饰）的。DUMPBIN 的另一个有用的开关是/EXPORTS 和/IMPORTS。

我们现在准备建立应用程序，该程序将使用刚建立的静态库。使用下列头文件和源文件。注意，我们使用的头文件与库所使用的头文件相同。在你选择的目录中创建源文件（最好与你创建库所使用的目录不同）复制头文件 Static.h 到该目录下。创建一个新的“ Win 32 Console Application ”类型的应用程序。为了将库包含到你的项目，可以通过 Projects 菜单中的“ Add to project -files ”项加载到项目中或在 “ Build- Settings-Link-Input-Ojbect/Library Modules ”编辑框中提供库的完整路径（图 4 在应用程序中使用静态连接库）。在调试模式下建立项目。

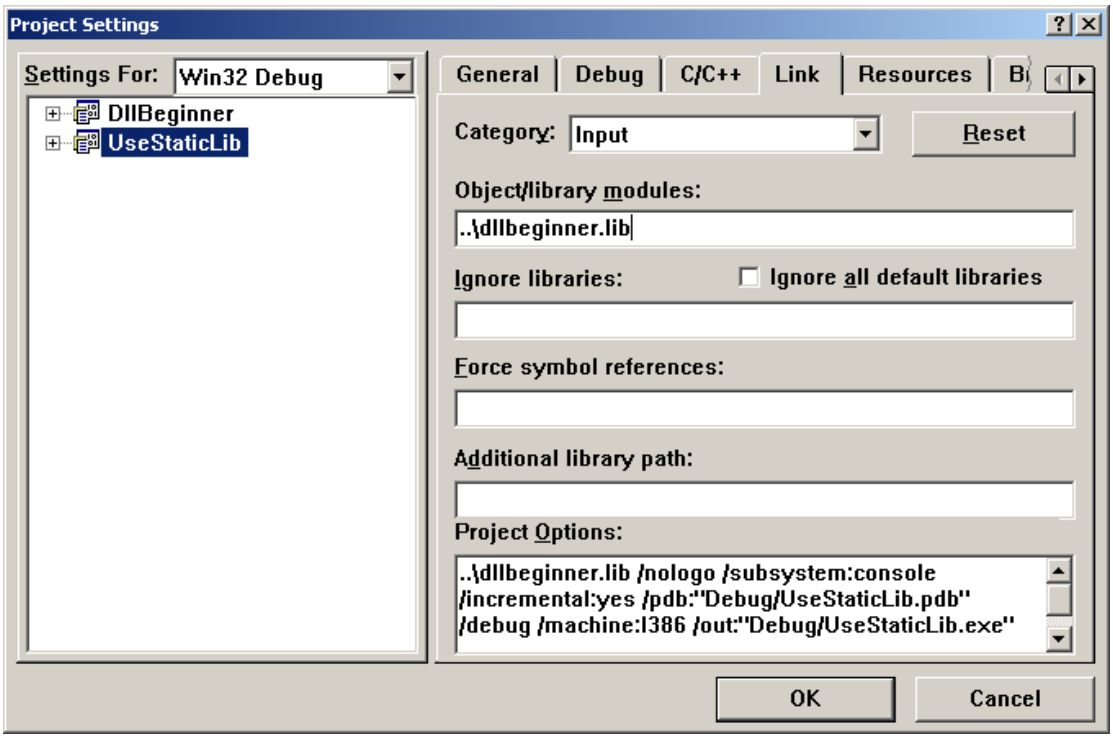


图 4 在应用程序中使用静态连接库

项目建立完成后，对它进行调试，看一看我们是否能够执行应用程序和库的所有函数。我们不设置断点而是按 F5（Debug Go），由于它是一个很小的程序，我们可以按 F11(Step into)键。它将带我们进入主程序的花括号内。从该处开始，我们可以按 F11 执行所有的代码，包括库函数。我们可以通过 ATL+TAB 键在控制台命令行窗口和 VC＋＋ IDE 的代码调试窗口之间切换来验证输出结果。当你到达主函数的最后一个花括号时，你可以按 F5 终止调试或从“ Debug ”菜单中选择“ Stop Debug ”(或 SHIFT+F5)。可以通过这种方式来设置断点，即，将光标停在将要设置断点的源代码行上，点工具条上的手形图标（或按 F9）－－个红点将出现在该行，表示断点。按 F5 应用程序将在断点处停止执行。

注意，应用程序可以使用库的发行版来建立。可以通过发行版方式建立该库，然后将库加到应用程序中来验证，应用程序应在调试模式下建立。如果现在调试程序代码，你会发现在库函数被调用时，它不会完全执行所有代码，取而代之的是，库的函数正确执行，你得到了预期的执行结果。尽管我们可以对 libGlobal

变量声明时没有用 extern 关键字，但是对于在模块外部声明的变量使用 extern 关键字是个好习惯。

我们用 C 代码实现了静态库并用 C 应用程序来调用。同一个静态库也可以由 C＋＋程序来调用。但是，如果我们将应用程序源代码的名称由 AppSource.c 改为 appsource.cpp，然后重新建立项目，就会得到下面的连接错误,这是因为 C＋＋修饰了变量和函数的名字。

```
appsource.obj : error LNK2001: unresolved external symbol "int __cdecl LibFuncB(int const *)"
(?libfuncB@@YAHPBH@Z)
appsource.obj : error LNK2001: unresolved external symbol "void __cdecl LibFuncA(char *)"
(?libfuncA@@YAXPAD@Z)
```

图 5 连接错误

通过以如下方式声明函数和变量，我们可以命令编译器使用 C 的名称修饰方法。
extern “C” functionname/variablename;
我们可以将多个声明组成一组。修改的头文件如下所示，建立并测试程序。

```
//该头文件为库输出方法和变量的声明文件。

#ifndef _STATIC1_H_
#define _STATIC1_H_
#ifdef __cplusplus
extern "C"{
#endif

   extern int libGlobal;
   int CallAppFunc(int *ptrlib);
   void LibFuncA(char * buffer);
   int LibFuncB(int const *mainint);

#ifdef __cplusplus
}
#endif

#endif
```

## 5.3 建立动态连接库

### 5.3.1 隐式动态连接

隐式动态连接又称加载时动态连接，这需要调用模块在连接时连接 DLL 的导入库。

#### 5.3.1.1 使用__declspec 属性

现在我们把建立的静态库转换为动态连接库（DLL）。我们可以使用 dllexport 属性建立动态连接库，或使用单独的模块定义文件(.DEF 文件)建立动态连接库。这两种方法我们都将讨论。首先，我们将讨论使用 dllexport 属性实现 DLL。使用__dllimport 属性能够提高效率，它能向导入代码一样导入数据和对象。因为它们是属性不是关键字，所以 dllexport 和 dllimport 必须与__declspec 关键字联在一起使用。使用__declspec 的语法为：

__declspec(attribute)variable-delcaration

例如：下面使用 dllexport 属性定义了一个输出的整数。

__declspec(dllexport) int Sum = 0;

现在，让我们使用以上语法转换已建立的静态库使用的头文件中的原型声明。转换后的 DLL 的头文件看起来象下面的样子。现在，我们将回调函数在 DLL 中注掉；也就是说，我们将不从 DLL 调用一个在应用程序中的函数。删除在源代码中的回调函数。

```
//使用属性的动态连接库的头文件。
#ifndef _STATIC1_H_
#define _STATIC1_H_
   __declspec(dllexport) extern int libGlobal;
   __declspec(dllexport)void LibFuncA(char * buffer);
   __declspec(dllexport)int LibFuncB(int const *mainint);
#endif
```

现在建立一个新的项目(图 6 建立动态连接库)，给出它的名字，选择其类型为“ Dynamin-LinkLibrary ”，将源文件和头文件插入到该项目中，然后在调试模式下建立。现在在“ Debug ”目录下有两个文件 LibName.lib 和 Libname.dll，其中 LibName 是项目的名称。“ LibName.Lib ”作为我们刚建立 DLL（LibName.dll）的导入库。该文件不包含任何代码，只包含与应用程序连接所需要的函数引用和其它声明。函数的目标代码是名为 LibName.dll 的文件。

现在我们可以建立应用程序了。应用程序的源文件和头文件将与建立静态库时所用的源代码和头文件相同。这次，应用程序的头文件与在 DLL 中的头文件不一样了，因为这个文件是我们在静态库中使用的。我们可以给头文件一个新的名字，以示区别，对这个应用程序，命名为 Usedllattr.h

我们再建立一个控制台项目，将源文件加入到项目中。向前面一样，我们在“ Projects – Add To Project –Files ”加入导入库的全路径，或在“ Build – Settings – Link – Input _ Object/Library Modules ”编辑框中加入导入库全路径。建立该项目。

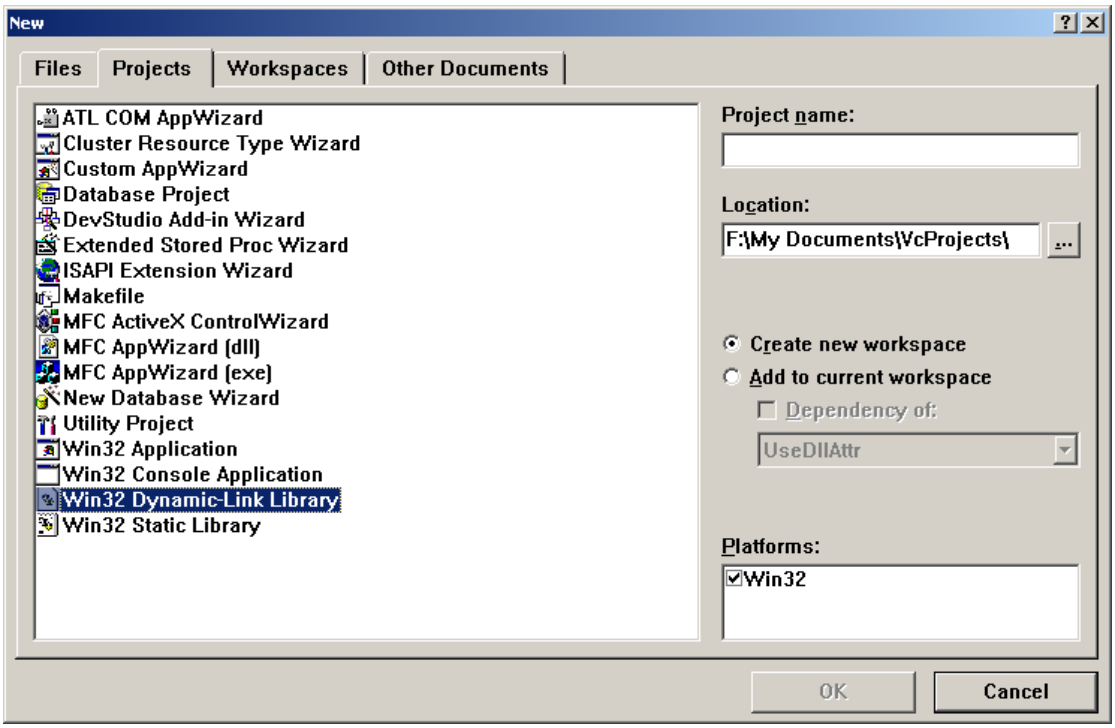


图 6 建立动态连接库

以下是该项目的实现文件：

```
#include <stdio.h>
#include "UseDllAttr.h"

int mainglobal;
void main(void)
{
    int libreturn;
    char buf[] = "在库中打印该串文字";
    printf("我们启动应用程序\n");
    LibFuncA(buf);
    printf("增加库中的全局变量并在应用程序中打印它：%d",++libGlobal);
    mainglobal = 10;
    libreturn = LibFuncB(&mainglobal);
    printf("打印库函数返回值:%d\n",libreturn);
    printf("调用程序结束\n");
}
```

### 5.3.1.2 调试

当应用程序建立时，我们可以按 F11 开始调试。但是这次，我们在消息框中得到了一个错误信息（图 7 错误信息），指示动态连接库没有在指定的路径中。这表示操作系统在试图从指定目录中加载动态连接库，它查找的目录有当前目

录，执行文件的目录，Windows 系统目录，Windows 目录和所有在环境变量中指定的目录。这意味着我们需要将建立的动态连接库复制到这些目录其中的一个目录中。

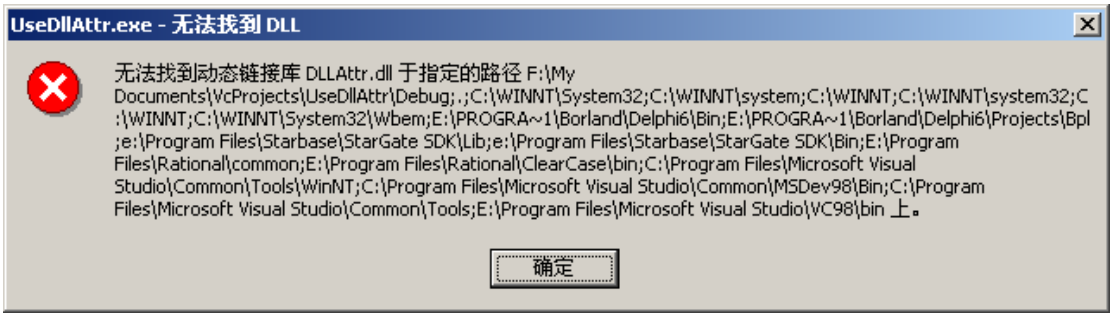


图 7 错误信息

我们将动态连接库移到应用程序的“ Debug ”目录中。现在按 F11，启动调试器，进入应用程序。结果，能进入到动态连接库的源文件中调试。除了在应用程序中输出 libGlobal 变量我们得到 1（这个值在静态库中是 21）以外，其它都正常工作。这个结果与静态库的例子相比有所不同。这是因为静态库和调用它的应用程序有相同的数据区而现在应用程序有自己的数据区，DLL 也有它自己的数据区。所以 libGlobal 被应用程序处理自己的全局变量，同样 DLL 中 libGlobal 被 DLL 处理为 DLL 自己的全局变量。所以我们从 DLL 中导入 libGlobal 的目的没有实现。要使它能够按既定的目的运行，我们需要修改在应用程序头文件中 libGlobal 变量声明。修改后的头文件如下所示：

```
#ifndef _USEDLLATTR_H_
#define _USEDLLATTR_H_

__declspec(dllimport)int libGlobal;
void LibFuncA(char * buffer);
int    LibFuncB(int const * mainint);

#endif
```

这就意味着从 DLL 导出数据类型或对象时，应用程序向上面一样需要使用__declspec(dllimport)属性。再次建立程序并调试。这次一切工作都正常。

上面的调试会话是从应用程序的工作区开始的。我们也可以从 DLL 的项目工作区开始调试。方法是，在 DLL 函数中设置断点。然后在 DLL 的 Build 菜单的“ Settings – debug-general-executable for debug session ”框中输入调用动态连接库的应用程序路径（图 8 设置在 DLL 中调试）。通过按 F5 启动调试器，调试器会停在 DLL 中的断点处。你甚至可以在应用程序没有调试信息的情况下调试 DLL。从而象 Word，Excel 这些可以使用用户自定义 DLL 的应用程序的调试就可以使用这种技术。如果没有输入应用程序的全路径，那么会出现一个提示框要求你输入。

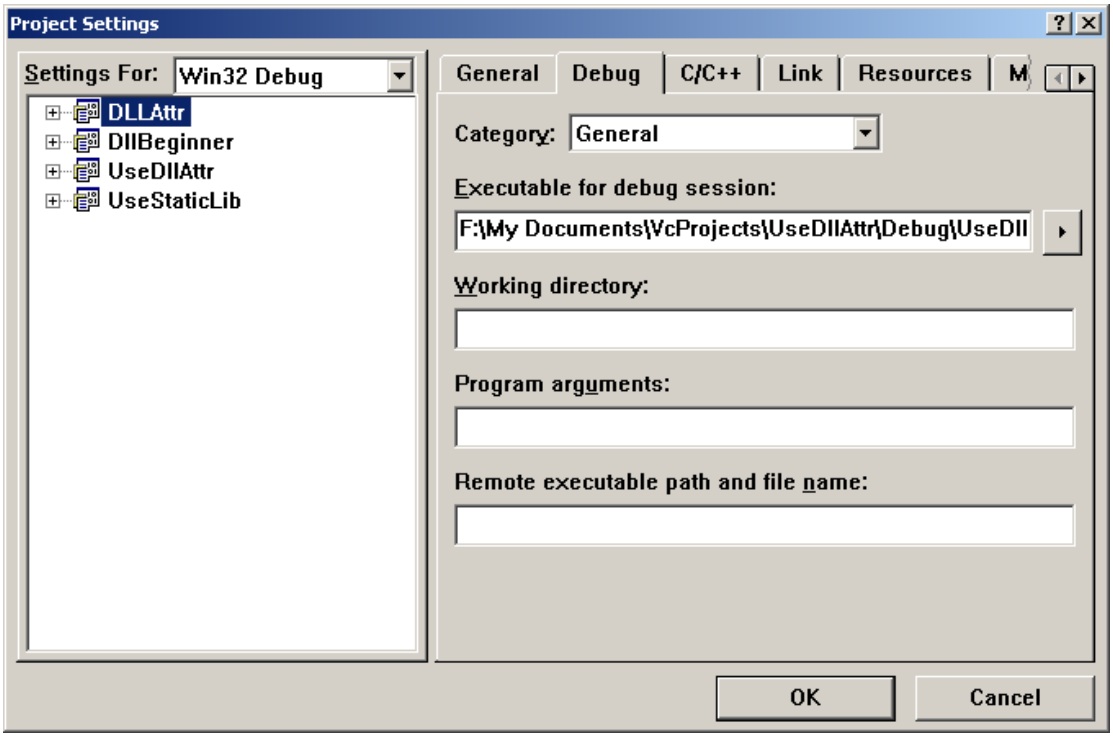


图 8 设置在 DLL 中调试

### 5.3.1.3 使用__declspec(dllimport)

在应用程序的头文件中，我们没有为 DLL 的导出函数使用__declspec(dllimport)属性，但是这样做会提高代码的效率。如果这样做，修改后的头文件如下所示 ：

```
__declspec(dllimport) int libglobal;
__declspec(dllimport) void libfuncA(char * buffer);
__declspec(dllimport) int    libfuncB(int const * mainint);
```

### 5.3.1.4 使用相同的头文件

让 DLL 和应用程序一样使用相同的头文件会怎样呢？在 DLL 源文件的最顶端加入#define MAKE_A_DLL 后试用下面的头文件。

```
#ifdef MAKE_A_DLL
#define LINKDLL __declspec(dllexport)
#else
#define LINKDLL __declspec(dllimport)
#endif

LINKDLL int libglobal;
LINKDLL void libfuncA(char * buffer);
LINKDLL int    libfuncB(int const * mainint);
```

这样应用程序和动态连接库可以使用相同的头文件了。

#### 5.3.1.5 使用 DEF 文件

前面我们提到，可以通过使用 DEF 文件而不使用__declspec(dllexport)来实现 DLL。有几种情况你必须使用 DEF 来建立 DLL。我们将在后面讨论这些情况。但是你必须记住在从 DLL 中导入数据时，客户程序必须使用_declspec(dllimport)。

我们先使用 DEF 文件建立一个 DLL。这需要从头文件中清除所有的 declspec 指示字，加入一个模块定义文件。以下是这两个文件：

```
//头文件
   int libGlobal;
   void LibFuncA(char * buffer);
   int LibFuncB(int const *mainint);
;DEF 文件
LIBRARY      DllDef
EXPORTS
   libGlobal
   LibFuncA
   LibFuncB
```

在 EXPORTS 段中列出了所有导出的声明。

LIBRARY 语句定义了这个模块定义文件所属的 DLL。它必须是文件中的第一个语句。在 LIBRARY 语句中指出的名字是 DLL 导出库的标识。将 DEF 文件加入到 DLL 项目中，建立 DLL。

建立应用程序时，我们必须修改库的头文件，使其作为应用程序的头文件，包含在应用程序中。所有的都和前面一样，注意，要为数据成员使用 declspec 指示符（这是必须的）。

```
#ifndef _USEDLLATTR_H_
#define _USEDLLATTR_H_

__declspec(dllimport)int libGlobal;
void LibFuncA(char * buffer);
int    LibFuncB(int const * mainint);

#endif
```

建立程序然后运行并调试，得到预期的执行结果。

DEF 文件也可以用于调用一个不同名字的导出函数。那样的话，我们可以实现这样一个 DLL，在该 DLL 中的函数可通过其原名调用或别名调用。在给出这样的一个例子之前，我想介绍一下在函数中使用_cdecl 和_stdcall 调用惯例的意义。

_cdecl 是 C 和 C＋＋程序的缺省调用惯例。因为栈的清除工作是由调用者来完成的，它可以执行 vararg 函数。__cdecl 调用惯例创建的可执行文件大小比__stdcall 调用惯例创建的可执行文件的大，因为它要求每个函数调用包含栈清除代

码。C 的名称修饰方案为用下划线__来修饰这个函数名。对于下面的函数声明，修饰的名字为__Myfunc，在这里使用__cdecl 是可选的。

int __cdecl Myfucn(int a,double b)；

__stdcall 调用惯例用于调用 Win32 API 函数。被调用者清除栈，所以编译器自动地将变量函数（vararg）转换成__cdecl。C 名字修饰方案是以下划线（__）后跟@＃nn 函数名修饰，其中＃nn 是函数变量表中变量的字节数，以十进制表示，是 4 的倍数或是栈中变量所需要的字节数。下面的函数声明，修饰后的名字为_Myfunc@12。

int __stdcll Myfunc(int a ，double b)

对于__cdecl 和__stdcall，参数在栈中是由右到左传递的。在 windows.h 头文件中，WINAPI，PASCAL，和 CALLBACK 都定义为__stdcall。现在，应当在以前使用 PASCAL 或__far__pascal 的地方使用 WINAPI。

函数的 PASCAL 名字修饰是将函数名转换成大写字母。象 VisualBasic 等语言遵循 PASCAL 习惯。这样一来，上面的 Myfunc 函数将在 PASCAL 修饰下的名字为 MYFUNC。VB 程序调用 C 的 DLL 时，我们可以使用 DEF 文件来用不同的名字调用函数。下面例子显示了如何使用 DEF 文件达到目的。这个例子中的调用程序是一个 C 程序。

```
//实现文件
#include "pascal.h"

double __stdcall MyFunc(int a, double b)
{
    return a*b;
}

int CdeclFunc(int a)
{
    return 2*a;
}
//头文件 pascal.h
//Header file   pascal.h
#ifndef _PASCAL_H_
#define _PASCAL_H_
  extern "C" double __stdcall MyFunc(int a, double b);
  extern "C" int   CdeclFunc(int a);     //缺省调用方式
#endif
；DEF 文件
LIBRARY PASCALDLL
EXPORTS
  MYFUNC = _MyFunc@12 ;Pascal 调用
  _MyFunc@12 ;在 C 中的__stdcall 调用
  CdeclFunc   ;在 C 中的__cdecl 调用
  CMyFunc = CdeclFunc ;以不同的名字调用
```

现在建立新的控制台应用程序，将如下的 CPP 文件加入到项目中，并将 pascaldll.lib 加入到项目中。

```
//应用程序实现文件
#include <iostream.h>
#include "pascal2.h"
#include "pascal.h"

void main( void )
{
    int x = 3;
    double y = 2.3;
    int a;
    double b;
    b = MyFunc(x,y);
    cout <<"这是 B="<<b<<endl;
    a = CdeclFunc(x);
    cout <<"a = "<<a<<endl;
}

//头文件    pascal2.h
extern "C" int CdeclFunc(int a);
```

建立程序并运行，一切如预料的一样。

DEF 文件另一个有用的特征是，使用序号和 NONAME 标记。将上面的 DEF 做如下改动，并重新建立 DLL。序号是任意的十进制数，通常用于标示 DLL 中函数的编号，它在调用 GetProcAddress 函数时，提供一个简短的得到 DLL 函数地址方法。

```
; DEF file for the    DLL

LIBRARY    PascalDll
EXPORTS
  MYFUNC =_MyFunc@12 ; 用于 Pascal 调用的别名  MYFUNC
  _MyFunc@12          @1          ;   在 C 中以__stdcall 调用
    CdeclFunc             @2 NONAME ; 在 C 中以__cdecl 调用
    CMyFunc = CdeclFunc                ; 用不同的名字调用  CdeclFunc
```

NONAME 标记表示你不想将该函数的名字输出，只输出序号值。这时连接器不会把函数的名字加入到 DLL 的函数表中。这意味着应用程序不能通过以函数名为参数调用 GetProcAddress 函数，来取得该 DLL 导出函数的地址。

指定 NONAME 标记并不能将 DLL 中的函数名从导入库中删除。否则，我们就不能隐式的使用函数名将应用程序与 DLL 连接。

建立 DLL 后，重新建立应用程序，应用程序运行正常。

使用 NONANME 标记的优点是，用序号替代字符串函数名。这样对于包含大量函数的 DLL 来说能节省空间。

### 5.3.1.6 导出类

在 DLL 中导出 C＋＋类与我们上面导出函数和数据一样简单，你可以使用 declspec 属性或 DEF 文件。如果要导出一个派生类，也必须要导出它的基类。导出嵌入类也是一样的。以下代码展示了使用_declspec(dllexport)导出类和类成员的方法。

```
//动态连接库
#include <stdio.h>

    class DLLClass
        {
        public:
          // 导出成员函数
          __declspec( dllexport ) void functionA( void ) {
                printf("\nIn Function A of the exported function");
          }
        };

    // 导出类
    class __declspec( dllexport) ExportDLLClass
        {
        public:
          void functionB(void) {
              printf("\nIn Function B of the exported class");
          }
        };

    // 导出类实例
    __declspec(dllexport) DLLClass test;

//调用
#include <stdio.h>

    class DLLClass
        {
        public:
          // imported member function
          __declspec( dllimport ) void functionA( void );
        };

    class __declspec( dllimport) ExportDLLClass
        {
        public:
```

```
    void functionB(void);
};

__declspec( dllimport ) DLLClass test;

void main(void)
{
    ExportDLLClass TestClass;

    test.functionA();
    TestClass.functionB();
}
```

当跨 DLL 和 EXE 创建和删除类时，如果 new/delete 不匹配就会产生运行时错误。解决这个问题的方法是：

- 如果在你的代码中没有使用虚析构函数，那么就将对象的构造函数和析构函数定义成内联函数 inline，并将实际的工作封装辅助函数中。
- 重载 DLL 类的 new 和 delete 操作符，这样 new 和 delete 操作就会在 DLL 内部完成。

#### 5.3.1.7 导出模板类和函数

VC＋＋不支持导出模板类和函数，原因是因为模板类本身只有在实例化时才有具体的代码和数据。

### 5.3.2 显式动态连接

只有准备好在应用程序中调用一个 DLL 函数时，才使用显式动态连接，显示动态连接又称为运行时动态连接。当完成该函数调用时，就可以将 DLL 从内存中卸载。这样可以在应用程序运行时节约内存空间，可以调用更多的函数或加载其它的 DLL。显式动态连接所用到的两个函数是 LoadLibrary 和 FreeLibrary。使用 GetProcAddress 来得到被调用函数的地址。GetProcAddress 可以函数名或函数在 DEF 文件中的序号为参数。GetProcAddress 使用函数序号为参数，比使用函数名字符串为参数更快地在 DLL 函数表中查到相应的函数。如果在 DEF 文件中使用了 NONAME 标记，那么 GetProcAddress 方法必须用序号为参数。在通过使用序号得到函数的地址时，应当使用 MAKEINTRESOURCE 宏。下面的例子显示了如何使用该方法在应用程序中实现这种调用。注意，在调用 DLL 的应用程序实现文件中没有包含 Pascal.h 和 Pascal2.h 两个头文件，这是由于 DLL 是显式调用的，应用程序不与 DLL 的导入库连接。但是，DLL 必须在 LoadLibray 所要求的特定目录中。在我们下面的例子中，我们复制 pascaldll.dll 到应用程序的 EXE 目录中。为应用程序使用下列代码，建立应用程序。

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

```
void main(void)
{
    typedef int (*lpFunc1)(int);//
    typedef    double (__stdcall *lpFunc2)(int,double) ;//
    HINSTANCE hLibrary;
    lpFunc1 Func1,Func2;
     lpFunc2 Func3;
    int x = 3;
    double y = 2.3;
    int a,c;
    double b;
    hLibrary = LoadLibrary("pascaldll.dll");
    if (hLibrary != NULL)
    {
        Func1  =  (lpFunc1)  GetProcAddress(hLibrary,"CMyFunc");//按名称调
用
        if (Func1 != NULL)
            a = (Func1)(x);
        else
            cout<<"Func1 函数调用错误"<<endl;
    Func2  =  (lpFunc1)GetProcAddress(hLibrary,MAKEINTRESOURCE(2));//
按序号调用
        if(Func2 != NULL)
            c = (Func2)(a*x);
        else
            cout<<"Func2 函数调用错误"<<endl;
Func3= (lpFunc2) GetProcAddress(hLibrary,MAKEINTRESOURCE(1));
        if(Func3 != NULL)
            b = (Func3)(x,y);
        else
            cout <<"Func3 调用错误"<<endl;


    }
    else
        cout <<"动态连接调用错误"<<endl;
    cout <<"b="<<b<<endl;
    cout <<"a = "<<a<<endl;
    cout <<"c = "<< c<< endl;
    FreeLibrary(hLibrary);

}
```

运行应用程序，显示的输出与预想的一样。

当调试时，除非 LoadLibrary 加载动态连接库，否则不能在 DLL 中任何函数

处设置断点。为了避免这种情况，从“ Project ”菜单中打开“ Setting ”对话框，单击“ Debug ”页面，从“ Category ”下拉列表框中选择“ Additional DLLs ”，然后在“ Local Name ”域中输入动态加载的 DLL 名（图 9 设置 DLL 调试）。现在当你通过按 F5 或 F11 启动调试器时，你会看到调试输出窗口中显示“ Loaded symbols for mydallname.dll ”。现在你可以在 DLL 的源文件中设置断点了。

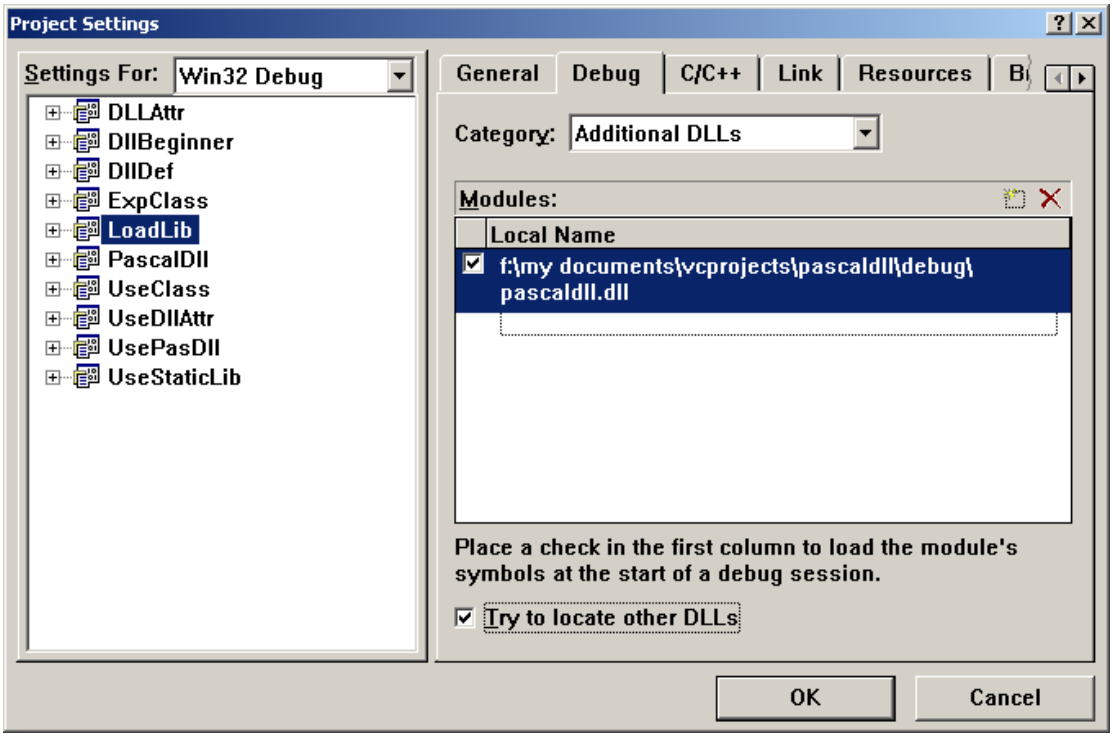


图 9 设置 DLL 调试

### 5.3.3 DLL 入口点

前面我们提到了 DLL 的入口函数，目前在我们的 DLL 实现中，并没有包含这个函数，但 DLL 仍能正常运行。这是因为 C/C++缺省提供了一个 DLL 入口函数，这个函数称为 DllMain。编译器在 DLL 代码中查找入口函数，如果没有发现，就使用缺省的 DllMain 函数。这个函数报告了 DLL 的初始化和终止。C/C++运行时库的实际名字为__DllMainCRTStartUp，它初始化了 C/C++运行时库（调用__CRT__INIT 函数）然后调用 DllMain。你可以在自己的代码中提供自己的 DllMain 函数。在这种情况下，你需要做执行__DllMainCRTStartUp 函数的工作（正确的初始化 C/C++运行时库）和用连接开关/ENTRY：FunctionName 或，如果你使用 VisualC++开发环境，就可以从 Project 菜单中打开 Setting 对话框，单击 Link 页面，从 Category 下拉列表中选择 OutPut，并在 Entry-point symbol 框输入函数名（图 10 设置 DLL 入口函数）。入口函数有三个参数，一个模块句柄，一个调用原因（有四种可能的原因）和一个保留的参数，它返回 1 表示成功。下面的代码显示 DllMain 的基本框架。你可以将下面的代码加入到前面实现的 DLL 中，但在执行时不会看到什么不同，因为它与缺省的 DllMain 没有什么区别。

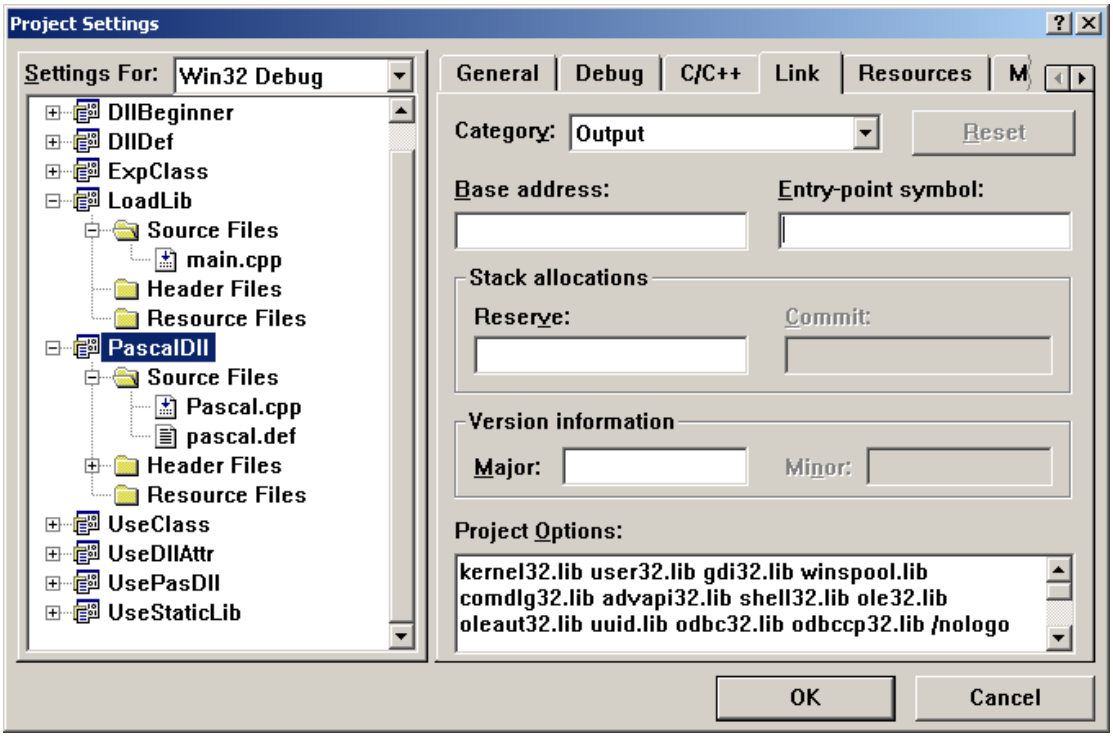


图 10 设置 DLL 入口函数

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

BOOL WINAPI DllMain(HANDLE hmodule, DWORD fdwreason, LPVOID lpReserved)
{
    switch (fdwreason)
    {
    case DLL_PROCESS_ATTACH:
        break;
    case DLL_THREAD_ATTACH:
        break;
    case DLL_THREAD_DETACH:
        break;
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}


double __stdcall MyFunc(int a, double b)
{
    return a*b;
```

```
}

int CdeclFunc(int a)
{
    return 2*a;
}
```

hModule 参数保存包含了 DLL 的实例句柄。可以将该参数保存在全局变量中，这样你可以因加载资源时（如 LoadString）使用它。

fdwreason 参数指示操作系统调用 DllMain 的原因。这四个原因分别是：DLL_PROCESS_ATTACH，DLL__THREAD__ATTACH，DLL__THREAD__DETACH 和 DLL__PROCESS__DETACH。如果每次 DLL 是由一个新进程调用，那么，DllMain 是因为 DLL_PROCESS_ATTACH 调用。如果在这个进程中的一个线程调用了 LoadLibrary 加载 DLL，那么，DllMain 是因为 DLL_THREAD_ATTACH 调用。在线程中调用 FreeLibrary 时，DllMain 的 fdwreason 参数为 DLL_TREAD_DETACH。当一个应用程序释放 DLL，DllMain 因为 DLL_PROCESS_DETACH 调用。

LpReserved 参数是一个保留参数，对于一般的进程退出通常传递 NULL，如调用 FreeLibrary，而并非调用 ExitProcess。

让我们写一些代码来说明 DllMain 函数的使用方法。我们修改 Pascldll.cpp 文件，并重新建立该 DLL。修改后的文件如下：

```
#include <windows.h>

#include "pascal.h"

HANDLE dllHandle;


BOOL WINAPI DllMain(HANDLE hmodule, DWORD fdwreason, LPVOID lpReserved)
{
     dllHandle = hmodule;
    switch (fdwreason)
    {
    case DLL_PROCESS_ATTACH:
        {
            MessageBox(NULL,"应用程序加载","",MB_OK);
             break;
        }
    case DLL_THREAD_ATTACH:
        MessageBox(NULL,"应用程序线程加载","",MB_OK);
        break;
    case DLL_THREAD_DETACH:
        MessageBox(NULL,"应用程序线程释放","",MB_OK);
```

```
            break;
        case DLL_PROCESS_DETACH:
            MessageBox(NULL,"应用程序释放","",MB_OK);
            break;
        }
        return TRUE;
    }

    double __stdcall MyFunc(int a, double b)
    {
        return a*b;
    }

    int CdeclFunc(int a)
    {
        return 2*a;
    }
```

用以下代码调用修改后的 DLL，在该代码中创建了一个线程。应用程序正常执行，并显示相应的信息。

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

    #pragma comment (lib,"LIBCMTD.LIB")//

    void firstthread(void * dummy);
    void main(void)
    {
        typedef int (*lpFunc1)(int);
        typedef    double (__stdcall *lpFunc2)(int,double) ;
        HINSTANCE hLibrary;
        lpFunc1 Func1,Func2;
          lpFunc2 Func3;
        int x = 3;
        double y = 2.3;
        int a,c;
        double b;
        hLibrary = LoadLibrary("pascaldll.dll");
        if (hLibrary != NULL)
        {
            Func1 = (lpFunc1) GetProcAddress(hLibrary,"CMyFunc");
            if (Func1 != NULL)
                a = (Func1)(x);
```

```
        else
            cout<<"Func1 函数调用错误"<<endl;
    Func2 = (lpFunc1)GetProcAddress(hLibrary,MAKEINTRESOURCE(2));
        if(Func2 != NULL)
            c = (Func2)(a*x);
        else
            cout<<"Func2 函数调用错误"<<endl;
    Func3 = (lpFunc2) GetProcAddress(hLibrary,MAKEINTRESOURCE(1));
        if(Func3 != NULL)
            b = (Func3)(x,y);
        else
            cout <<"Func3 调用错误"<<endl;


    }
    else
        cout <<"动态连接调用错误"<<endl;
    cout <<"应用程序中 b="<<b<<endl;
    cout <<"应用程序中 a = "<<a<<endl;
    cout <<"应用程序中 c = "<< c<< endl;
    _beginthread(firstthread,0,NULL);
    Sleep(10000L);
    cout<<"退出应用程序"<<endl;
    FreeLibrary(hLibrary);
}

void firstthread(void * dummy)
{
    typedef int (*lpFunc1)(int);
    typedef    double (__stdcall *lpFunc2)(int,double) ;
    HINSTANCE hLibrary;
    lpFunc1 Func1,Func2;
     lpFunc2 Func3;
    int x = 10;
    double y = 2.3;
    int a,c;
    double b;
    hLibrary = LoadLibrary("pascaldll.dll");
    if (hLibrary != NULL)
    {
        Func1 = (lpFunc1) GetProcAddress(hLibrary,"CMyFunc");
        if (Func1 != NULL)
            a = (Func1)(x);
        else
```

```
            cout<<"Func1 函数调用错误"<<endl;
        Func2 = (lpFunc1)GetProcAddress(hLibrary,MAKEINTRESOURCE(2));
            if(Func2 != NULL)
                c = (Func2)(a*x);
            else
                cout<<"Func2 函数调用错误"<<endl;
        Func3 = (lpFunc2) GetProcAddress(hLibrary,MAKEINTRESOURCE(1));
            if(Func3 != NULL)
                b = (Func3)(x,y);
            else
                cout <<"Func3 调用错误"<<endl;


        }
        else
            cout <<"动态连接调用错误"<<endl;
        cout <<"线程中 b="<<b<<endl;
        cout <<"线程中 a = "<<a<<endl;
        cout <<"线程中 c = "<< c<< endl;
        FreeLibrary(hLibrary);
    }
```

# 6 应用程序共享 DLL 数据

在某些情况下，DLL 要与其它的 DLL 或同一个 DLL 的不同映像共享数据。DLL 将被映射到加载它的应用程序的地址空间中，而该地址是私有的，因此有两种方法来完成在 DLL 中共享数据，一是创建共享数据段，一是使用内存映像文件。在这里推荐使用后者。

## 6.1 使用共享数据段

每个 32 位的应用程序都运行在自己的私有地址空间中，每一个 DLL 所映射到的应用程序都有自己的数据集。每个 DLL 或 EXE 都是由段的集合组成，通常每个段都由一个“ . ”开始（该符号是可选的），每段都有下面属性：READ, WRITE, SHARED, 和 EXECUTE。

在使用共享数据段时，需要共享数据的 DLL 可以在其源文件中加入#pragma 预处理命令，用以创建一个包含共享数据的共享数据段。

```
#pragma data_seg(".shared")
int iSharedVar = 0;
#pragma data_seg()
```

第一行指示编译器，将在该段声明的数据放入“ .shared ”数据段。因此 iShareVar 变量被存储在“ .shared ”数据段（注意数据段命名不要大于 8 个字符）。

缺省情况下，数据是不共享的。注意，你必须初始化命名数据段中的所有数据，data_seg()只适用于初始化的数据。第三行#pragma data_seg()将位置重新设置到缺省数据段中。

如果一个应用程序对共享数据段中的变量做出的任何修改，都会反映给该动态连接库的其它映像，所以当在应用程序或 DLL 中处理共享数据段中的变量时应当特别小心。

最后，你必须通知连接器，在你定义的数据区中的变量是可共享的，通知的方法是修改你的.DEF 文件，使之包括一个 SECTIONS 段；或是在你的项目的连接器设置中指定/SECTION:.shared,RWS，以下是一个在.DEF 中的 SECTIONS 代码

```
SECTIONS
    .shared      READ WRITE SHARED
```

有的编译器允许在代码中设置连接器开关，这样可以当代码复制到其它工程时，连接器开关也被复制到相应的工程中。在代码中使用连接器开关的方法是在#pragma data_seg()后面加入下面的语句：

```
#pragma comment(linker, "/SECTION:.shared,RWS")
```

切记，不要在“ ”之间加入任何多余的字符，否则，会引起连接器对该指示字的误解。

以下代码显示了如何使用共享数据段：

```
//
#ifndef _SHARE_H_
#define _SHARE_H_
   extern "C" void SetVar(int a);
   extern "C" int    GetVar();      //
#endif
//
;模块定义文件

LIBRARY ShareData
EXPORTS
   GetVar
   SetVar
//实现文件
#include "share.h"
#pragma data_seg(".SHARED")
int iSharedVar = 0;
#pragma data_seg()


#pragma comment(linker,"/SECTION:.SHARED,RWS")

void SetVar(int a)
{
     iSharedVar = a;
```

```
}

int GetVar()
{
    return iSharedVar;
}
```

## 6.2 使用内存映像文件

在动态连接库中使用内存映像文件（Memory-Mapped File），可以为加载该 DLL 的进程提供共享数据。在 DLL 中使用内存映像文件的方法是在 DLL 的入口函数中创建文件映射，只要 DLL 被加载，这块内存就固定下来。在 DLL 的入口函数中实现文件映射的方法是：

- 调用 CreateFileMapping 得到映射文件对象的句柄，第一个加载该 DLL 的进程创建文件映像对象，随后的进程打开已有　映像文件对象句柄。
- 调用 MapViewOfFile，映射一个视图到虚拟地址空间中，这样可以使进程能够存取该共享内存。

以下代码显示了如何在 DLL 中使用内存映像文件：

```
;模块定义文件
LIBRARY MapFileDll
EXPORTS
   GetMemData
   SetMemData
//头文件
#include <windows.h>
extern "C" VOID GetMemData(LPTSTR, DWORD );
extern "C" VOID SetMemData(LPTSTR );

//实现文件
#include <memory.h>
#include "MapFile.h"

#define SHMEMSIZE 4096 //共享内存大小

static LPVOID lpvShareMem = NULL; //  指向共享内存的指针

BOOL WINAPI DllMain(HINSTANCE hinstDLL,DWORD fdwReason,LPVOID lpvReserved)
{
    HANDLE hMapObject = NULL;    //  文件映像句柄
    BOOL fInit, fIgnore;
    switch (fdwReason)
    {
```

```
    case DLL_PROCESS_ATTACH:

        // 创建文件映像对象

        hMapObject = CreateFileMapping(
            (HANDLE) 0xFFFFFFFF, //
            NULL,                          //
            PAGE_READWRITE,          //
            0,                               //
            SHMEMSIZE,                   //
            "dllmemfilemap");       //
        if (hMapObject == NULL)
            return FALSE;

        // 第一个进程创建文件映像

        fInit = (GetLastError() != ERROR_ALREADY_EXISTS);

        // 得到文件映像内存的指针。

        lpvShareMem = MapViewOfFile(
            hMapObject,         //
            FILE_MAP_WRITE, //
            0,                        //
            0,                        //
            0);                       //
        if (lpvShareMem == NULL)
            return FALSE;

        // 初始化共享内存

        if (fInit)
            memset(lpvShareMem, '\0', SHMEMSIZE);

        break;

    case DLL_THREAD_ATTACH:
        break;

    case DLL_THREAD_DETACH:
        break;

    case DLL_PROCESS_DETACH:
```

```
                fIgnore = UnmapViewOfFile(lpvShareMem);    //解映像
                fIgnore = CloseHandle(hMapObject); //关闭映像文件句柄
                break;

            default:
               break;
        }

    return TRUE;
    UNREFERENCED_PARAMETER(hinstDLL);
    UNREFERENCED_PARAMETER(lpvReserved);
}

//设置共享内存的内容

VOID SetMemData(LPTSTR lpszBuf)
{
    LPTSTR lpszTmp;
    lpszTmp = (LPTSTR) lpvShareMem;
    while (*lpszBuf)
        *lpszTmp++ = *lpszBuf++;
    *lpszTmp = '\0';
}

// GetSharedMem gets the contents of shared memory.

VOID GetMemData(LPTSTR lpszBuf, DWORD cchSize)
{
    LPTSTR lpszTmp,lpszTmp1;

    lpszTmp = (LPTSTR) lpvShareMem;
   lpszTmp1 = lpszBuf;
    while (--cchSize) //*lpszTmp &&
        *lpszTmp1++ = *lpszTmp++;
    *lpszTmp1 = '\0';
}
```

# 7 DLL 基址冲突

无论是什么进程，所有系统 DLL 通常从相同虚拟地址加载。有时可能会发生 DLL 基址冲突，当 DLL 加载时会出现一面的信息：

LDR: Dll xxxx.DLL base 10000000 relocated due to collision with yyyy.DLL。

用 VC++建立的所有 DLL 的基址是 0x1000000(HEX)。我们可以通过 link -dump -headers your.dll 命令来查看 DLL 的基址。使用连接器的 /BASE:ADDRESS 开关修改基址。你使用的地址范围是 0x100000000<ADDRESS<0x60000000。

# 8 结论

本文介绍了 DLL 的基本概念，并用 VC＋＋写了几个简单易懂的代码。本文的目的是突破初学者对于编写 DLL 时的障碍，希望读者能从中得到帮助。关于 DLL 的其它高级用法如 TLS 和 HOOK 函数的编写，请读者查阅相关资料。

# 9 参考资料

Debabrata Sarma “DLLs for Beginners”（MSDN Library Archive）

“Wrong Operator Delete Called for Exported Class”（MSDNKnowledge Base）

“Using Shared Memory in a Dynamic-Link Library”（MSDN）