第3章 Windows的内存管理

3.1 实验一：动态链接库的建立与调用

3.1.1 实验目的

(1) 理解动态链接库的实现原理。

(2) 掌握 Windows 系统动态链接库的建立方法。

(3) 掌握 Windows 环境下动态链接库的调用方法。

3.1.2 实验准备知识：动态链接库介绍

动态链接库(Dynamic Link Library，DLL)是一个可执行模块，它包含的函数可以由Windows 应用程序调用以提供所需功能，为应用程序提供服务。

1. 动态链接库基础知识

大型的应用程序都是由多个模块组成的，这些模块彼此协作，以完成整个软件系统的工作。其中可能有些模块的功能是通用的，被多个软件系统使用。在设计软件系统时，如果将所有模块的源代码都静态编译到整个应用程序的 .exe 文件中，会产生两个问题：一是应用程序过大，运行时消耗较大的内存空间，造成系统资源的浪费；二是在修改程序时，每次程序的调整都必须编译所有的源代码，增加了编译过程的复杂度，也不利于阶段性的模块测试。

Windows 系统提供了非常有效的编程和运行环境，可以将独立的模块编译成较小的动态链接库文件，并可对这些动态链接库单独进行编译和测试。运行时，只有在主程序需要时才将动态链接库装入内存并运行。这样不仅减少了应用程序的大小及对内存的大量需求，而且使得动态链接库可以被多个应用程序使用，从而充分利用了资源。Windows 系统中的一些主要的系统功能都是以动态链接库的形式出现的，如设备驱动程序等。

动态链接库文件在 Windows 系统中的扩展名为.dll，它可以由若干函数组成，运行时被系统加载到进程的虚拟地址空间中，成为调用进程的一部分。如果与其他的动态链接库没有冲突，该文件通常映射到进程虚拟地址空间上。

2. 动态链接库入口函数

DllMain()函数是动态链接库的入口函数，当 Windows 系统加载动态链接库时调用该函数，DllMain()函数不仅在将动态链接库加载到进程地址空间时被调用，在动态链接库与进程分离时也被调用。

每个动态链接库必须有一个入口点，像用 C 语言编写其他应用程序时必须有一个

WinMain()函数一样,在 Windows 系统的动态链接库中,DllMain()是默认的入口函数。函数原型如下:

```
BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD  ul_reason_for_call,
                      LPVOID lpReserved)
{
    return TRUE;
}
```

其中,参数 hModule 为动态链接库的句柄,其值与动态链接库的地址相对应;参数 ul_reason_for_call 指明系统调用该函数的原因;参数 lpReserved 说明动态链接库是否需要动态加载或卸载。当 lpReserved 为 NULL 时表示需要动态加载,即运行时用到该动态链接库时才将其装入内存,当进程不用该动态链接库时,可使用 FreeLibrary()将动态链接库卸载;当 lpReserved 为非 NULL 时表示静态加载,进程终止时才卸载,即进程装入内存时同时将其动态链接库装入,进程终止时动态链接库与进程同时被卸载。

使用入口函数还能使动态链接库在被调用时自动做一些初始化工作,如分配额外的内存或其他资源;在撤销时做一些清除工作,如回收占用的内存或其他资源。需要做初始化或清除工作时,DllMain()函数格式如下:

```
BOOL APIENTRY DllMain (HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                      )
{
    switch (ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
          break;
    }
    return TRUE;
}
```

初始化或清除工作分以下几种情况。

(1) DLL_PROCESS_ATTACH。当动态链接库被初次映射到进程的地址空间时,系统将调用该动态链接库的 DllMain()函数,给它传递参数 ul_reason_for_call 的值 DLL_PROCESS_ATTACH。当处理 DLL_PROCESS_ATTACH 时,动态链接库应执行动态链接库函数要求的任何与进程相关的初始化工作,如动态链接库堆栈的创建等。当初始化成功时,DllMain()返回 TRUE,否则返回 FALSE,并终止整个进程的执行。

(2) DLL_PROCESS_DETACH。当动态链接库从进程的地址空间被卸载时,系统将调用该动态链接库的 DllMain()函数,给它传递参数 ul_reason_for_call 的值 DLL_PROCESS_DETACH。当处理 DLL_PROCESS_DETACH 时,动态链接库执行与进程相关的清除操作,如堆栈的撤销等。

(3) DLL_THREAD_ATTACH。当在一个进程中创建线程时,系统查看当前映射进程的地址空间中的所有动态链接库文件映像,并调用每个带有 DLL_THREAD_ATTACH 值的 DllMain()函数文件映像。这样,动态链接库就可以执行每个线程的初始化操作。新创建的线程负责执行动态链接库的所有 DllMain()函数中的代码。

当一个新动态链接库被映射到进程的地址空间时,如果该进程内已经有若干线程正在执行,那么系统将不为现有的线程调用带 DLL_THREAD_ATTACH 值的 DllMain()函数。只有当新线程创建,动态链接库被映射到进程的地址空间时,它才可以调用带有 DLL_THREAD_ATTACH 值的 DllMain()函数。另外,系统并不为主线程调用带 DLL_THREAD_ATTACH 值的 DllMain()函数。进程初次启动时映射到进程的地址空间中的任何动态链接库均接收 DLL_PROCESS_ATTACH 通知,而不是 DLL_THREAD_ATTACH 通知。

(4) DLL_THREAD_DETACH。终止线程的方法是系统调用 ExitThread()函数撤销该线程,但如果 ExitThread()函数要终止动态链接库所在的线程,系统不会立即将该线程撤销,而是取出这个即将被撤销的线程,并让它调用已映射的动态链接库中所有带有 DLL_THREAD_DETACH 值的 DllMain()函数。通知所有的动态链接库执行每个线程的清除操作,只有当每个动态链接库都完成了对 DLL_THREAD_DETACH 通知的处理时,操作系统才会终止线程的运行。

如果当动态链接库被撤销时仍然有线程在运行,那么带 DLL_THREAD_DETACH 值的 DllMain()函数就不会被任何线程调用。所以在处理 DLL_THREAD_ATTACH 时,要根据具体情况进行。

3. 动态链接库导入/导出函数

动态链接库文件中包含一个导出函数表,这些导出函数由它们的符号名和标识号被唯一地确定,导出函数表中还包含了动态链接库中函数的地址。当应用程序加载动态链接库时,通过导出函数表中各个函数的符号名和标识号找到该函数的地址。如果重新编译动态链接库文件,并不需要修改调用动态链接库的应用程序,除非改变了导出函数的符号名和其他参数。

在动态链接库源程序文件中声明导出函数的代码如下:

```
__declspec(dllexport) MyDllFunction(int x, int y);
```

其中,关键字__declspec(dllexport)表示要导出其后的函数 MyDllFunction()。如果一个动态链接库文件中的函数还需要调用其他动态链接库,此时,动态链接库文件除了导出函数外,还需要一个导入函数,声明导入函数的代码如下:

```
__declspec(dllimport) DllAdd(int x, int y);
```

其中,关键字__declspec(dllimport)表示要导入其后的函数 DllAdd(),在生成动态链接库时,链接程序会自动生成一个与动态链接库相对应的导入/导出库文件(.lib 文件),下面的例子中创建了一个名为 SimpleDll 的动态链接库工程文件,在 SimpleDll 工程的 Debug 目录下,可以看到 SimpleDll.dll 和 SimpleDll.lib 两个文件,其中 SimpleDll.dll 是编译生成的动态链接库可执行文件,SimpleDll.lib 就是导入/导出库文件,该文件中包含 SimpleDll.dll 文件名和 SimpleDll.dll 中的函数名 Add()和 Sub(),SimpleDll.lib 是文件 SimpleDll.dll 的

映像文件,在进行隐式链接时要用到它。

下面是一个动态链接库程序的例子。

```
#include …
extern "C" _declspec(dllexport) int Add(int x, int y);
extern "C" _declspec(dllexport) int Sub(int x, int y);
BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD  ul_reason_for_call,
                      LPVOID lpReserved)
{
    return TRUE;
}

int Add(int x, int y)
{
  int z;
  z = x + y;
  return z;
}
int Sub(int x, int y)
{
  int z;
  z = x - y;
  return z;
}
```

应用程序要链接引用动态链接库的任何可执行模块,其 .lib 文件是必不可少的。除了创建 .lib 文件外,链接程序还要将一个输出符号表嵌入生成的动态链接库文件中。该输出符号表包含输出变量、函数和类的符号列表及函数的虚拟地址。

4. 动态链接库的两种链接方式

当应用程序调用动态链接库时,需要将动态链接库文件映射到调用进程的地址空间中。映射方法有两种:一种是应用程序的源代码只引用动态链接库中包含的符号,当应用程序运行时,加载程序隐式地将动态链接库装入进程的地址空间中,这种方法也称隐式链接;另一种方法是应用程序运行时使用 LoadLibary()显式地加载所需要的动态链接库,并显式地链接需要的输出符号表。

当进程加载动态链接库时,Windows 系统按以下搜索顺序查找并加载动态链接库。

- 应用程序的当前目录(将动态链接库文件复制到应用程序的 .exe 文件所在目录下)。
- Windows 目录下。
- Windows\System32 目录下。
- PATH 环境变量中设置的目录。
- 列入映射网络目录表中的目录。

(1) 隐式链接。如果程序员创建了一个动态链接库文件(.dll),链接程序会自动生成一个与动态链接库相对应的导入/导出库文件(.lib 文件)。该文件作为动态链接库的替代文件被编译到应用程序的工程项目中。当编译生成应用程序时,应用程序中的调用函数与

.lib 文件中导出符号名相匹配，若匹配成功，这些符号名进入生成的应用程序的可执行文件中。.lib 文件中也包含对应动态链接库文件名，但不包含路径，它同样也被放入生成的应用程序的可执行文件中。Windows 系统根据这些信息发现并加载动态链接库，然后通过符号名实现对动态链接库函数的动态链接。

在调用动态链接库的应用程序中，声明要调用的动态链接库中的函数，需要写明 extern "C"，它可以使其他编程语言访问所编写的动态链接库中的函数。下面是通过隐式链接方法调用 SimpleDll.dll 中的函数 Add()和 Sub()的方法，先创建一个控制台工程文件 CallDll，在 CallDll.cpp 文件中输入代码：

```
extern "C" _declspec(dllimport) int Add(int x, int y);
extern "C" _declspec(dllimport) int Sub(int x, int y);
int main(int argc, char * argv[])
{
    int x = 7;
    int y = 6;
    int t add = 0;
    int sub = 0;
    printf("Call Dll Now!\n");
    //调用动态链接库
    add = Add(x,y);
    sub = Sub(x,y);
    printf("7 + 6 = %d,7 - 6 = %d\n",add,sub);
    return 0;
}
```

为了能够使调用程序 CallDll.cpp 正确地调用动态链接库 SimpleDll.dll，在生成工程文件 CallDll.cpp 的可执行文件之前，先将 SimpleDll.dll 复制到工程文件 CallDll 的 Debug 目录下，将 SimpleDll.lib 复制到 CallDll.cpp 所在的目录下。然后在 Microsoft Visual Studio 的环境下的 Project Settings 对话框的 Link 选项卡中输入动态链接库的导入/导出库文件 SimpleDll.lib。

(2) 显式链接。显式链接方式更适合于集成化的开发工具，如 Visual Basic 等，使用显式链接，程序员不必再使用导入/导出文件，而直接调用 Win32 提供的 LoadLibary()函数加载动态链接库文件，调用 GetProcAddress()函数得到动态链接库中函数的内部地址，在应用程序退出之前，还应使用 FreeLibrary()释放动态链接库。

下面是通过显式链接调用动态链接库的例子。

```
#include …
int_tmain(int argc, TCHAR * argv[], TCHAR * envp[])
{   int s;
    int nRetCode = 0;

    typedef int ( * pAdd)(int x, int y);
    typedef int ( * pSub)(int x, int y);

    HMODULE hDll;
    pAdd add;
```

```
    pSub sub;

    hDll = LoadLibrary("SimpleDll.dll");          //加载动态链接库文件 SimpleDll.dll
    add = (pAdd)GetProcAddress(hDll,"Add");       //得到动态链接库中函数 Add()的内部地址
    s = add(6,2);
    sub = (pSub)GetProcAddress(hDll,"Sub");       //得到动态链接库中函数 Sub ()的内部地址
    s = sub(6,2);
    FreeLibrary(hDll);                            //释放动态链接库 SimpleDll.dll
    return nRetCode;
}
```

在上面的例子中,使用类型定义关键字 typedef 定义了指向动态链接库中相同函数原型的指针,然后通过 LoadLibrary("SimpleDll.dll")将动态链接库文件 SimpleDll.dll 加载到应用程序中,并返回当前动态链接库文件的句柄。再通过 GetProcAddress(hDll,"Add")和 GetProcAddress(hDll,"Sub")获得导入到应用程序中动态链接库的函数 Add()和 Sub()的指针。函数调用完毕后使用 FreeLibrary(hDll)卸载动态链接库文件。需要注意的是,在编译应用程序之前,要把动态链接库文件复制到应用程序所在的目录下使用显式链接方式,不需要使用相应的 .lib 文件。

5. 函数调用参数传递约定

动态链接库函数调用参数传递约定决定着函数参数传送时入栈和出栈的顺序,以及编译器用来识别函数名称的修饰约定。为了使不同的编程语言方便地共享动态链接库,函数输出时必须使用正确的调用约定。

Microsoft Visual C++ 6.0 支持的常用函数约定主要有_stdcall 调用约定、C 调用约定和_fastcall 调用约定。

(1) _stdcall 调用约定相当于 16 位动态链接库中经常使用的 PASCAL 调用约定。在 Microsoft Visual C++ 6.0 中不再支持 PASCAL 调用约定,取而代之的是_stdcall 调用约定。两者在实质上是一致的,即调用时函数的参数自右向左通过内存栈传递,被调用函数返回时清理传送参数的内存栈。

(2) C 调用约定(用_cdecl 关键字说明)与_stdcall 调用约定在参数传递顺序上是一致的,不同的是用于传送参数的内存栈是由调用者来维护的,使用变参的函数只能使用该调用约定。

(3) _fastcall 调用约定的主要特点是快,因为它是通过寄存器来传递参数的,实际上它使用寄存器 ECX 和 EDX 传送前两个参数,剩下的参数仍自右向左通过内存栈传递,被调用函数返回时清理传送参数的内存栈。因此,对于参数较少的函数使用关键字_fastcall 可以提高其运行速度。

关键字_stdcall、_cdecl 和_fastcall 可以在函数说明时直接写在要输出的函数前面,也可以在 Microsoft Visual C++ 6.0 编译环境中进行设置。

3.1.3 实验内容

(1) 在 Windows 环境下建立一个动态链接库。

(2) 使用隐式调用法调用动态链接库。

(3) 使用显式调用法调用动态链接库。

3.1.4 实验要求

掌握动态链接库的建立和调用方法。在 Microsoft Visual C++ 6.0 环境下建立一个动态链接库,并分别使用隐式和显示方式将其调用,从而体会使用动态链接库的优点。

3.1.5 实验指导

(1) 启动安装好的 Microsoft Visual C++ 6.0 后,在 Microsoft Visual C++ 6.0 环境下选择 File→New 命令,然后在 Projects 选项卡中选择 Win32 Dynamic-Link Library 选项建立一个动态链接库工程文件,在 Project name 文本框中输入工程文件名,在 Location 文本框中输入工程文件名所在路径,然后单击 OK 按钮,如图 3-1 所示。

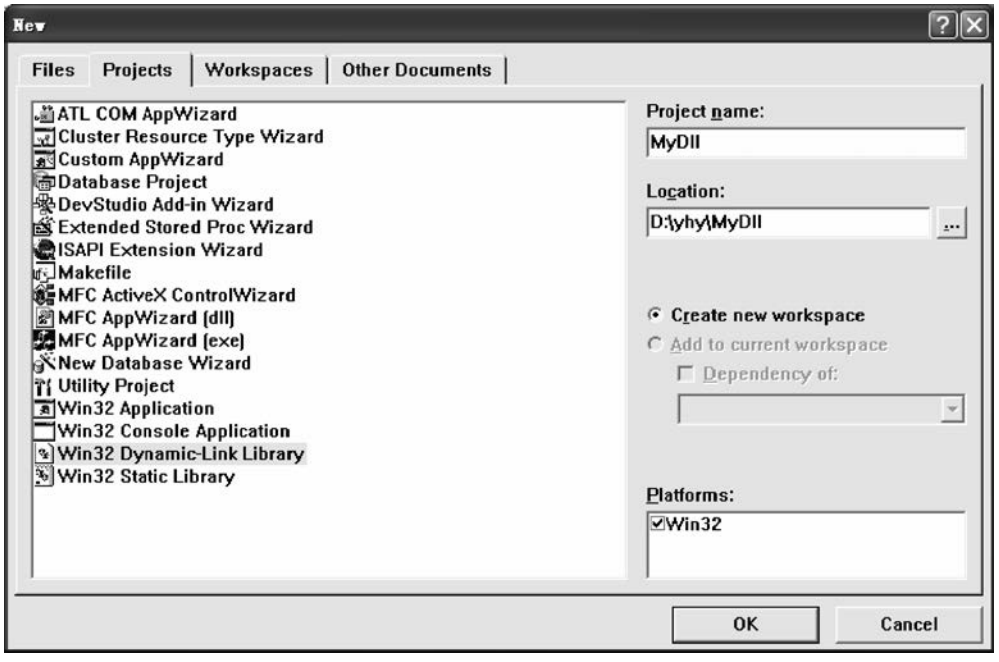

图 3-1 动态链接库工程文件的建立

(2) 接下来选择动态链接库的类型,有 3 种选择,分别是 An empty DLL project、A simple DLL project 和 A DLL that exports some symbols,如图 3-2 所示,选中所需单选按钮后单击 Finish 按钮即可。如果选中 An empty DLL project 单选按钮,接下来打开的 Microsoft Visual C++ 6.0 编辑环境是一个空白文件,用户需要在该文件中增加动态链接库的入口函数 DllMain()。

(3) 如果选中图 3-2 中的 A simple DLL project 单选按钮,将会打开图 3-3 所示的 Microsoft Visual C++ 6.0 编辑环境,该环境下会有一个 DllMain 的动态链接库入口函数,用户直接使用该函数即可。

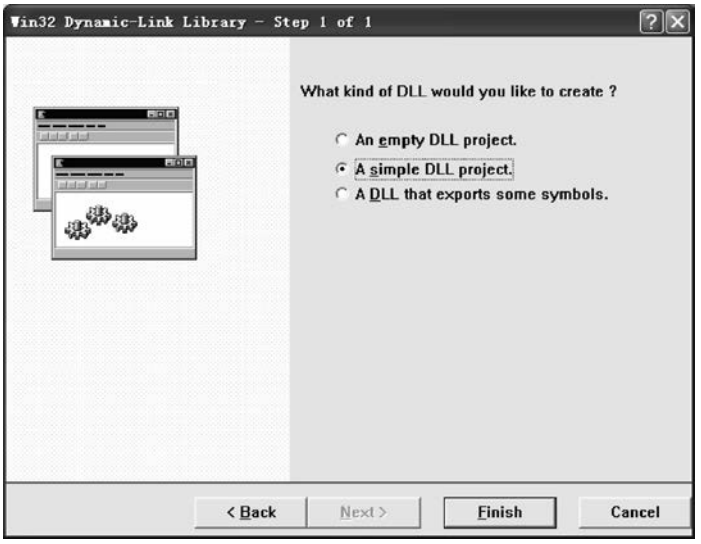

图 3-2　选择动态链接库的类型

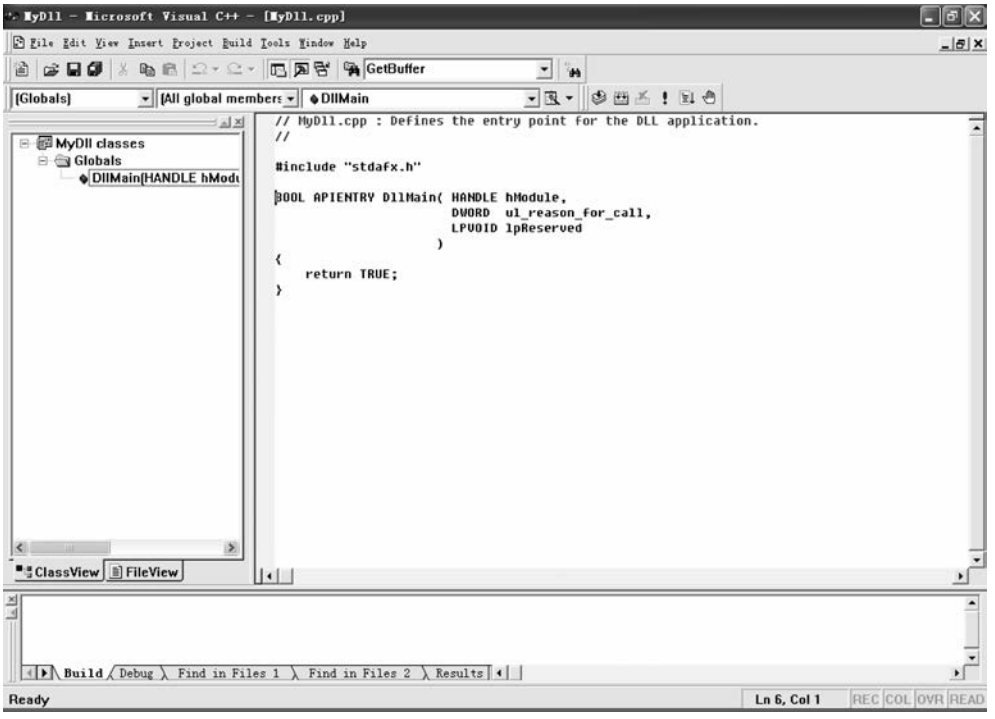

图 3-3　选择简单动态链接库编辑环境

(4) 如果选中图 3-2 中的 A DLL that exports some symbols 单选按钮,将会打开图 3-4 所示的 Microsoft Visual C++ 6.0 编辑环境,该环境下会有一个 DllMain 的动态链接库入口函数。

(5) 在图 3-3 或图 3-4 所示的工程环境的 .cpp 文件中就可以编写动态链接库的函数

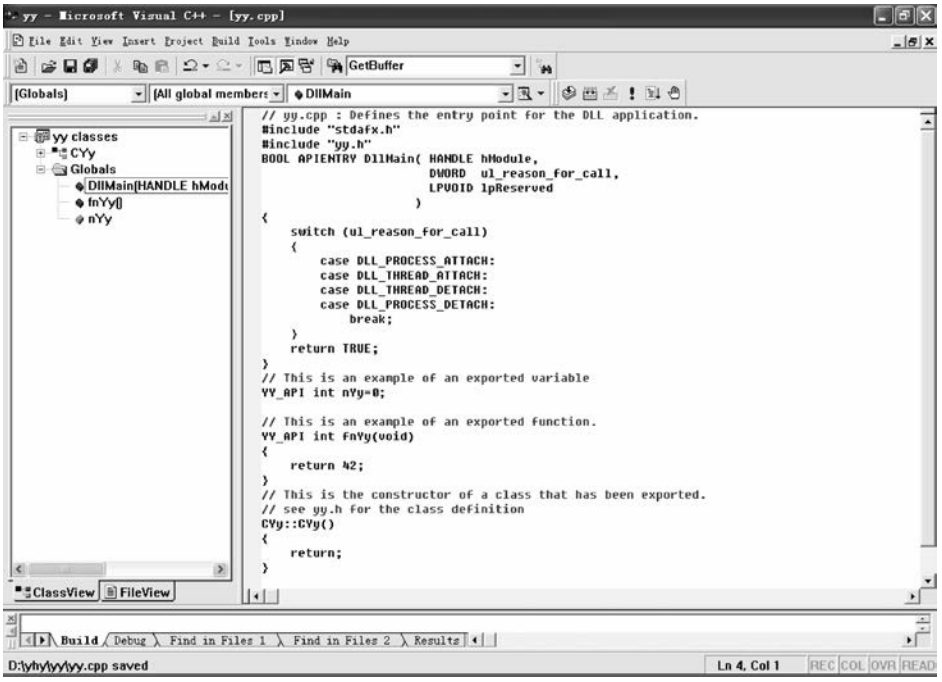

图 3-4　选择输出符号的动态链接库编辑环境

了。然后使用 Microsoft Visual C++提供的编译、链接工具可生成动态链接库，如图 3-5 所示，至此动态链接库建立完毕。

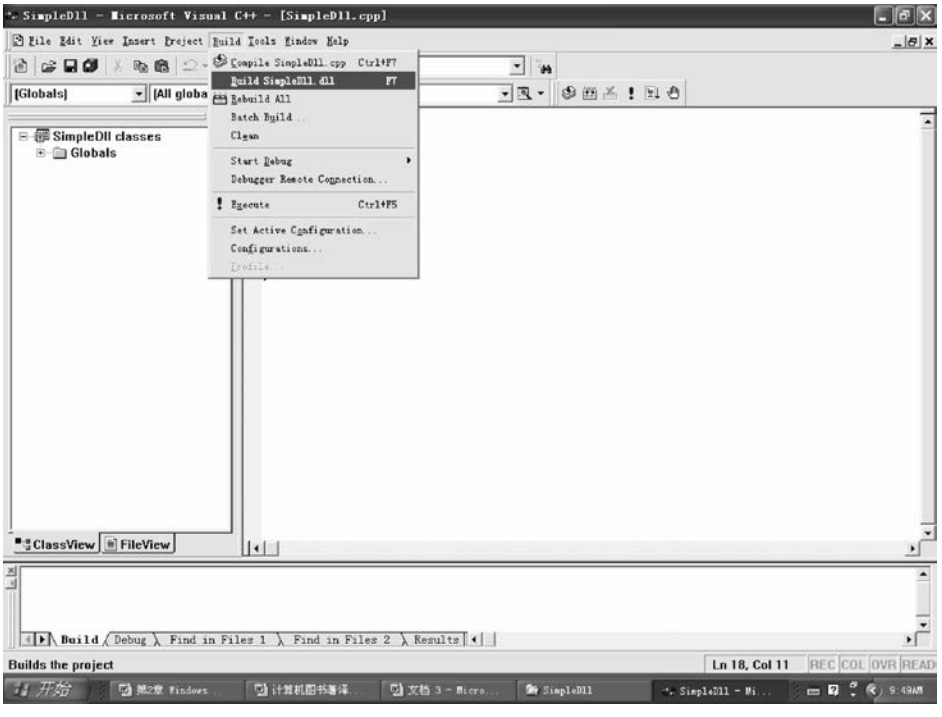

图 3-5　生成动态链接库 .dll 文件

(6) 如果想查看动态链接库的导入/导出函数,可以使用 Microsoft Visual Studio 提供的 DumpBin.exe 应用程序(带有-exports 开关),能够看到动态链接库的输出符号表的内容,图 3-6 所示的是动态链接库 SimpleDll.dll 的输出结果。

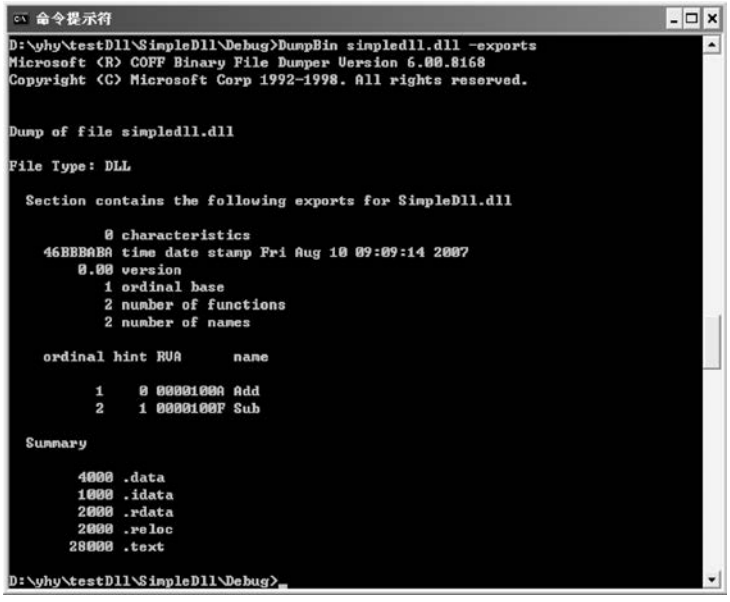

图 3-6 使用 DumpBin.exe 查看输出动态链接库的导出函数

也可以使用 Microsoft Visual Studio 提供的可视化工具 Dependency Walker,查看动态链接库的导出函数信息,如图 3-7 所示。

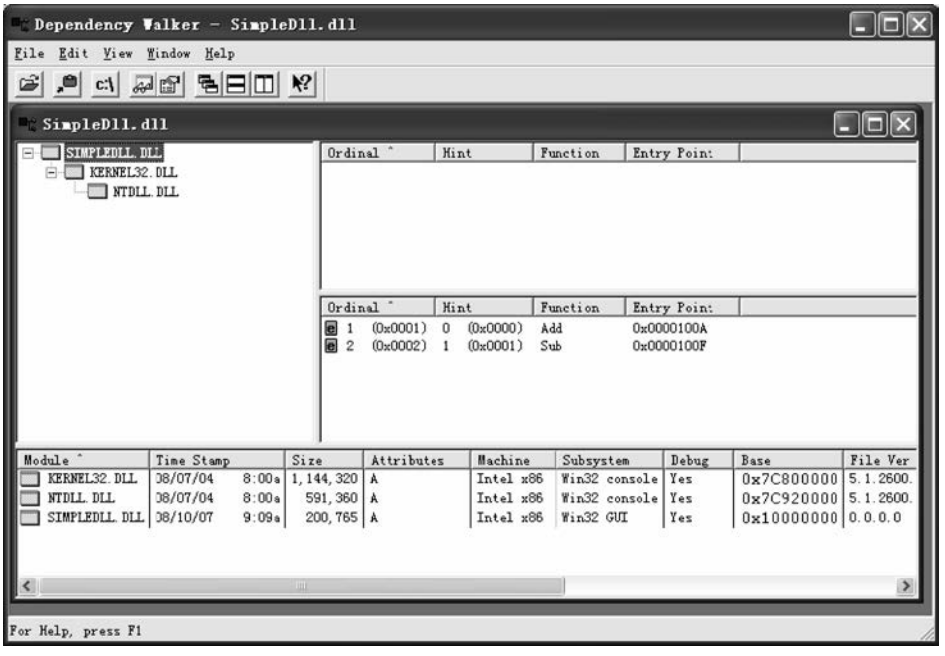

图 3-7 使用 Dependency Walker 查看动态链接库的导出函数

(7) 下面讨论如何调用该动态链接库。如果采用隐式链接法调用动态链接库,首先建立一个控制台工程文件 CallDll,用此文件中的 CallDll.cpp 调用前面建立好的动态链接库 SimpleDll.dll。在生成控制台工程文件 CallDll 的可执行文件之前,先将 SimpleDll.dll 复制到工程文件 CallDll 的 Debug 目录下,将 SimpleDll.lib 复制到 CallDll.cpp 所在的目录下。然后在 Microsoft Visual Studio 的环境下,选择 Project→Settings 命令,在 Link 选项卡中的 Project Options 文本框中输入动态链接库的导入/导出库文件 SimpleDll.lib,如图 3-8 所示。

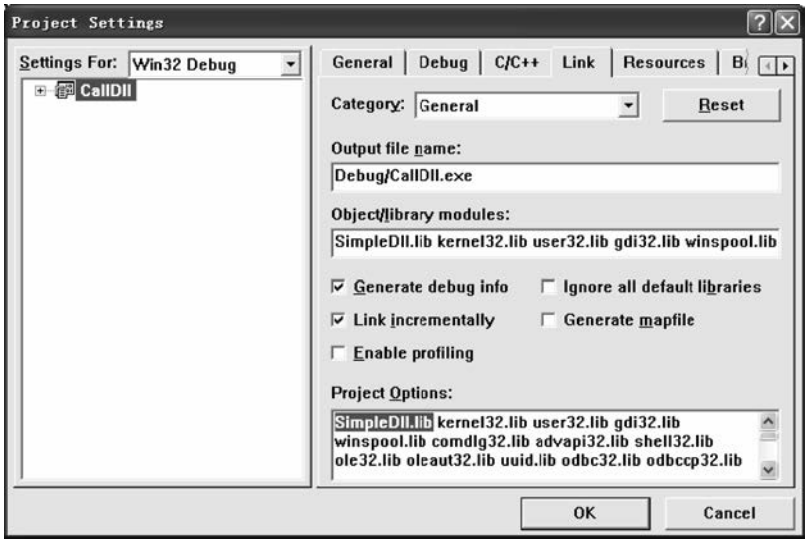

图 3-8　应用程序中输入动态链接库的导入/导出库文件 SimpleDll.lib

(8) 如果需要设置函数调用参数传递约定,可在 Microsoft Visual C++ 6.0 编译环境中进行设置。选择 Project→Settings 命令,在 C/C++选项卡中的 Category 下拉列表框中选择 Code Generation 选项,然后可在 Calling convention 下拉列表框中选择所需要的关键字,如图 3-9 所示,这里选择的调用约定是_cdecl * 。

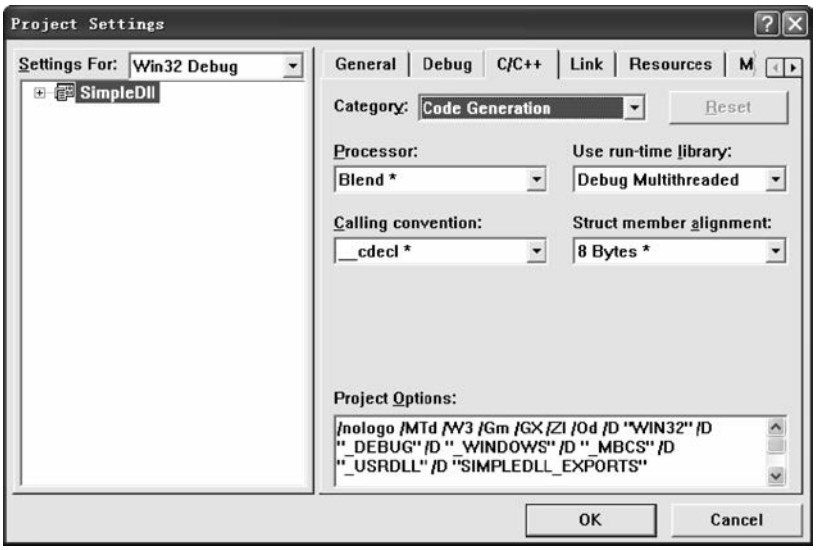

图 3-9　设置函数调用参数传递约定

3.1.6 实验总结

该实验完成了动态链接库的建立和调用。函数 Add()和 Sub()在动态链接库文件 SimpleDll.cpp 中,分别完成两个整数的相加和相减。而调用该动态链接库的程序文件是 CallDll.cpp,该程序的运行结果如图 3-10 所示。

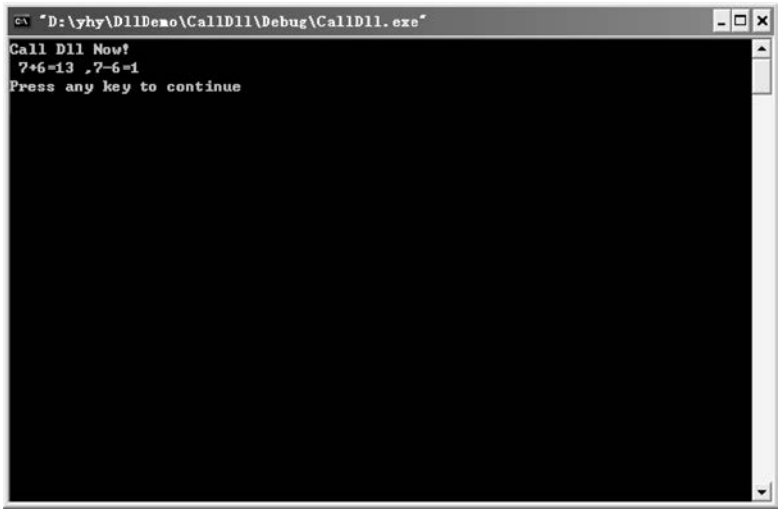

图 3-10 动态链接库被调用结果

3.1.7 源程序

```
//SimpleDll.cpp: Defines the entry point for the DLL application

#include "stdafx.h"
extern "C"_declspec(dllexport) int Add(int x,int y);
extern "C"_declspec(dllexport) int Sub(int x,int y);
BOOL APIENTRY DllMain (HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                      )
{
    return TRUE;
}

int Add(int x,int y)
{
    int z;
    z = x + y;
    return z;
}

int Sub(int x,int y)
{
```

```
    int z;
    z = x - y;
    return z;
}

//隐式调用动态链接库的程序
//CallDll.cpp: Defines the entry point for the console application

#include "stdafx.h"

extern "C" _declspec(dllimport) int Add(int x,int y);
extern "C" _declspec(dllimport) int Sub(int x,int y);
int main(int argc, char * argv[])
{
    int x = 7;
    int y = 6;
    int add = 0;
    int sub = 0;
    printf("Call Dll Now!\n");
    add = Add(x,y);
    sub = Sub(x,y);
    printf("7 + 6 = %d,7 - 6 = %d\n",add,sub);
    return 0;
}

//显式调用动态链接库的程序
//CallDllAddress.cpp: Defines the entry point for the console application

#include "stdafx.h"
#include "CallDllAddress.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////////////////////////////////////////////////////////////////////
//The one and only application object
CWinApp theApp;
using namespace std;

int _tmain(int argc, TCHAR * argv[], TCHAR * envp[])
{
    int s;
    int nRetCode = 0;

    typedef int ( * pAdd)(int x,int y);
    typedef int ( * pSub)(int x,int y);

    HMODULE hDll;
```

```
    pAdd add;
    pSub sub;

    hDll = LoadLibrary("SimpleDll.dll");
    if(hDll == NULL)
    {
        printf("LoadLibrary Error....\n");
        return nRetCode;
    }
    else printf("LoadLibrary Success....\n");

    add = (pAdd)GetProcAddress(hDll,"Add");
    s = add(6,2);
    printf("6 + 2 = %d\n",s);

    sub = (pSub)GetProcAddress(hDll,"Sub");
    s = sub(6,2);
    printf("6 - 2 = %d\n",s);
    FreeLibrary(hDll);
    return nRetCode;
}
```

3.1.8 实验展望

本实验介绍了在 Microsoft Visual C++ 6.0 环境下建立与调用动态链接库的方法,使用动态链接库除了可以节省内存空间、实现代码共享之外,还可以实现多种编程语言书写的程序相互调用,读者在完成上述实验的基础上,可以通过自学完成用 Visual Basic 语言或 Java 编写的程序调用 Visual C++建立的动态链接库,从中体会多种语言编程给程序员带来的方便。

3.2 实验二:系统内存使用统计

3.2.1 实验目的

(1) 了解 Windows 内存管理机制,理解页式存储管理技术。
(2) 熟悉 Windows 内存管理基本数据结构。
(3) 掌握 Windows 内存管理基本 API 的使用。

3.2.2 实验准备知识:相关数据结构及 API 函数介绍

1. 相关系统数据结构说明

系统结构 MEMORYSTATUS 中包含当前物理内存和虚拟内存信息,使用函数 GlobalMemoryStatus()可以将这些信息存储在结构 MEMORYSTATUS 中。

结构原型:

```
typedef struct_MEMORYSTATUS {
```

```
DWORD dwLength;                    //MEMORYSTATUS 结构大小
DWORD dwMemoryLoad;                //内存利用率
DWORD dwTotalPhys;                 //物理内存大小
DWORD dwAvailPhys;                 //空闲物理内存大小
DWORD dwTotalPageFile;             //页文件大小
DWORD dwAvailPageFile;             //空闲页文件大小
DWORD dwTotalVirtual;              //虚拟地址空间大小
DWORD dwAvailVirtual;              //空闲虚拟地址空间大小
} MEMORYSTATUS, * LPMEMORYSTATUS;
```

参数说明如下。

(1) dwLength：MEMORYSTATUS 数据结构的大小，单位为字节。

(2) dwMemoryLoad：当前内存利用率，取值范围为 0～100％，0 表示内存没有被使用，100％表示内存全部被使用。

(3) dwTotalPhys：物理内存的总字节数。

(4) dwAvailPhys：可用物理内存的字节数。

(5) dwTotalPageFile：页文件的总字节数。页文件是虚拟内存系统占用的磁盘空间。

(6) dwAvailPageFile：页文件中可用字节数。

(7) dwTotalVirtual：用户模式下调用进程可以访问的虚拟地址空间总字节数。

(8) dwAvailVirtual：用户模式下调用进程虚拟地址空间中未提交和未保留的内存总字节数，即可用虚拟地址空间大小。

2. 相关 API 函数介绍

(1) 获取系统物理内存和虚拟内存使用信息。

原型：

```
VOID GlobalMemoryStatus(
                        LPMEMORYSTATUS lpBuffer          //指向 MEMORYSTATUS 数据结构
                       );
```

参数说明如下。

lpBuffer：指向 MEMORYSTATUS 数据结构的指针，函数 GlobalMemoryStatus()将内存的当前信息存储在该结构中。

返回值：

该参数没有返回值。

(2) 保留或提交某一段虚拟地址空间。函数 VirtualAlloc()可以在调用进程的虚拟地址空间中保留或提交若干页面。保留意味着这段虚拟地址不能被使用，当提交时，这段虚拟地址才真正被分配给该进程。

原型：

```
LPVOID VirtualAlloc(
                    LPVOID lpAddress,                //待分配空间的起始位置
                    DWORD dwSize,                    //待分配空间的大小
                    DWORD flAllocationType,          //分配类型
                    DWORD flProtect                  //存取保护的类型
                   );
```

参数说明如下。

① lpAddress：待分配空间的起始位置。若该值为NULL，系统将为其分配一个合适的起始地址，否则用户要指定一个准确的起始地址。

② dwSize：待分配空间的大小。如果参数lpAddress不为NULL，则待分配空间范围为lpAddress～lpAddress＋dwSize。

③ flAllocationType：分配类型，可以为表3-1所示标志的任意组合。

表3-1　标志描述

标　志	描　述
MEM_COMMIT	提交，即在内存或磁盘页文件中分配物理内存
MEM_RESERVE	保留进程的虚拟地址空间，而不分配物理内存。保留的地址空间在没有被释放之前不能使用，因此使用Malloc()和LocalAlloc()这样的操作再次申请使用该地址空间被视为无效。被保留的地址空间可随后使用VirtualAlloc()函数提交

④ flProtect：指定存取保护的类型。若虚拟地址空间已经被提交，则在指定表3-2中的任何一个属性时，同时也要一起指定PAGE_GUARD(页保护)和PAGE_NOCACHE(页无缓存)这两个属性。存取保护位的类型如表3-2所示。

表3-2　存取保护位的类型

标　志	描　述
PAGE_READONLY	被提交的虚拟地址空间只读
PAGE_READWRITE	被提交的虚拟地址空间可读/写
PAGE_EXECUTE	被提交的虚拟地址空间可执行
PAGE_EXECUTE_READ	被提交的虚拟地址空间可执行、可读
PAGE_EXECUTE_READWRITE	被提交的虚拟地址空间可执行、可读/写
PAGE_GUARD	保护
PAGE_NOACCESS	不允许访问
PAGE_NOCACHE	无缓存

返回值：

如果函数调用成功，则返回值为已分配虚拟地址空间的起始地址。如果函数调用失败，则返回值为NULL。若要得到更多的错误信息，可调用GetLastError()函数。

(3) 释放或注销某一段虚拟地址空间。函数VirtualFree()用于释放或注销某一段虚拟地址空间。

原型：

```
BOOL VirtualFree(
                LPVOID lpAddress,          //待分配空间的起始位置
                DWORD dwSize,              //待分配空间的大小
                DWORD dwFreeType           //释放操作类型
               );
```

参数说明如下。

① lpAddress：待释放空间的起始位置。如果dwFreeType值为MEM_RELEASE，该参数必须使用VirtualAlloc()函数返回的地址。

② dwSize：待释放空间的大小。如果 dwFreeType 值为 MEM_RELEASE，该参数必须为 0，否则待释放空间的范围为 lpAddress～lpAddress＋dwSize。

③ dwFreeType：释放类型。它可以为表 3-3 所示标志的任意组合。

表 3-3　释放类型

标　志	描　述
MEM_DECOMMIT	注销提交，如果注销一个没有提交的虚拟地址空间，也不会导致失败，即提交或没有提交的虚拟地址空间都可以注销
MEM_RELEASE	释放保留的虚拟地址空间，如果使用该标志，参数 dwSize 必须为 0，否则函数调用失败

返回值：

如果函数调用成功，则返回值为非零。

如果函数调用失败，则返回值为零。若要得到更多的错误信息，可调用 GetLastError()函数。

(4) 分配内存空间。

原型：

```
void * malloc(size_t size);
```

参数说明如下。

size：要分配内存大小，单位为 B(字节)。

返回值：

该函数返回分配内存空间 void 类型的指针。如果函数返回 NULL，说明没有有效的内存空间可供分配。

(5) 释放内存空间。

原型：

```
void free(void * memblock);
```

参数说明如下。

* memblock：要释放的内存地址。

返回值：

无。

用法举例：

```
/* MALLOC.C: This program allocates memory with
 * malloc, then frees the memory with free.
 */

#include <stdlib.h>          /* For_MAX_PATH definition */
#include <stdio.h>
#include <malloc.h>

void main(void)
{
    char * string;
```

```
    /* Allocate space for a path name */
    string = malloc(_MAX_PATH);                    //分配内存空间
    if(string == NULL)
        printf("Insufficient memory available\n");
    else
    {
        printf("Memory space allocated for path name\n");
        free(string);                              //释放内存空间
        printf("Memory freed\n");
    }
}
```

3.2.3 实验内容

使用 Windows 系统提供的函数和数据结构显示系统存储空间的使用情况,当内存和虚拟存储空间变化时,观察系统显示变化情况。

3.2.4 实验要求

能正确使用系统函数 GlobalMemoryStatus()和数据结构 MEMORYSTATUS 了解系统内存和虚拟存储空间使用情况,会使用 VirtualAlloc()函数和 VirtualFree()函数分配和释放虚拟存储空间。

3.2.5 实验指导

在 Microsoft Visual C++ 6.0 环境下选择 Win32 Console Application 建立一个控制台工程文件,由于内存分配、释放及系统存储空间使用情况函数均是 Microsoft Windows 操作系统的系统调用函数,因此选中 An application that supports MFC 单选按钮,如图 3-11 所示。

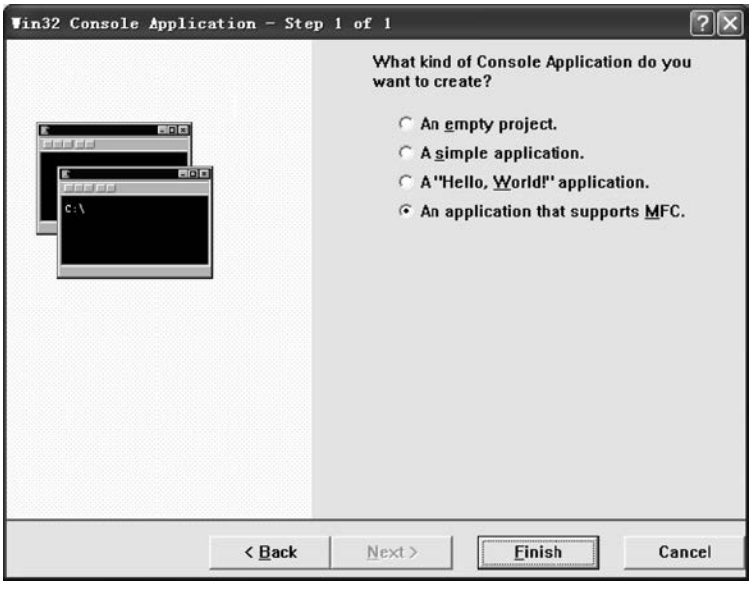

图 3-11 建立一个 MFC 支持的控制台应用程序

3.2.6 实验总结

观察图 3-12 所示程序的运行情况，可以看出以下几点。

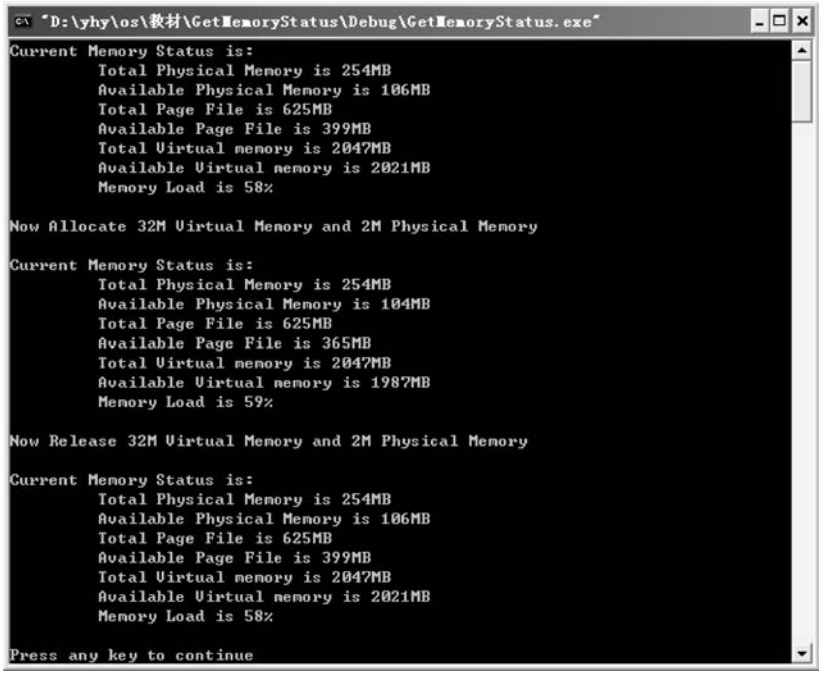

图 3-12 分配内存和虚拟存储空间前后情况

(1) 程序开始运行时，显示的可用物理内存为 106MB，可用页文件大小为 399MB，可用虚拟内存为 2021MB。

(2) 当分别使用函数 VirtualAlloc()和 malloc()分配了 32MB 虚拟内存和 2MB 物理内存后，系统显示可用物理内存为 104MB，可用页文件大小为 365MB，可用虚拟内存为 1987MB。

(3) 当分别使用函数 VirtualFree()和 free()释放了 32MB 虚拟内存和 2MB 物理内存后，系统的显示情况又恢复了(1)的情况。

3.2.7 源程序

```
//GetMemoryStatus.cpp: Defines the entry point for the console application

#include "stdafx.h"
#include "GetMemoryStatus.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

```
void GetMemSta(void);

//////////////////////////////////////////////////////////////////////////
//The one and only application object
CWinApp theApp;
using namespace std;

int_tmain(int argc, TCHAR * argv[], TCHAR * envp[])
{
    int nRetCode = 0;
    LPVOID BaseAddr;
    char * str;

    GetMemSta();
    printf("Now Allocate 32M Virtual Memory and 2M Physical Memory\n\n");
    BaseAddr = VirtualAlloc(NULL,                    //分配虚拟内存
        1024 * 1024 * 32,
        MEM_RESERVE|MEM_COMMIT,
        PAGE_READWRITE);
    if (BaseAddr == NULL)printf("Virtual Allocate Fail\n");
    str = (char * )malloc(1024 * 1024 * 2);          //分配内存
    GetMemSta();
    printf("Now Release 32M Virtual Memory and 2M Physical Memory\n\n");
    if (VirtualFree(BaseAddr,0,MEM_RELEASE) == 0)    //释放虚拟内存
        printf("Release Allocate Fail\n");
    free(str);                                       //释放内存
    GetMemSta();

    return nRetCode;
}

void GetMemSta(void)                                 //统计内存的状态
{
    MEMORYSTATUS MemInfo;
    GlobalMemoryStatus(&MemInfo);
    printf("Current Memory Status is: \n");
    printf("\t Total Physical Memory is %dMB\n",MemInfo.dwTotalPhys/(1024 * 1024));
    printf("\t Available Physical Memory is %dMB\n",MemInfo.dwAvailPhys/(1024 * 1024));
    printf("\t Total Page File is %dMB\n",MemInfo.dwTotalPageFile/(1024 * 1024));
    printf("\t Available Page File is %dMB\n",MemInfo.dwAvailPageFile/(1024 * 1024));
    printf("\t Total Virtual memory is %dMB\n",MemInfo.dwTotalVirtual/(1024 * 1024));
    printf("\t Available Virtual memory is %dMB\n",MemInfo.dwAvailVirtual/(1024 * 1024));
    printf("\t Memory Load is %d%% \n\n",MemInfo.dwMemoryLoad);
```

3.2.8 实验展望

从实验结果可以看出,虚拟内存的变化有些误差,请解释原因是什么。如果想了解更详细的虚拟内存情况,如锁定(VirtualLock)、解锁(VirtualUnlock)及保护方式(VirtualProtect)等,又要如何设计程序来实现呢?