№ 13 Библиотеки динамической компоновки. Экспортирование функций и классов

1. Создать Проект, в котором в одно диалоговое окно выводится строка – результат функции из DLL в другое - выводится строка – результат функции из класса DLL (функции могут совпадать, тогда при вывод дать пояснительный текст, какая из функций работает).
2. Реализовать два варианта подключения – явное (динамическая загрузка LoadLidrary) и неявное (подключение через статическую библиотеку).

Вариант	Функция
1,7,13, 19,25	Определить сумму свободного места на логических дисках
2,8,14, 20,26	Определить логические диски компьютера
3,9,15,21	Определить имя локального компьютера
4,10,16,22	Определить свободное количество физической памяти в байтах
5,11,17,23	Определить загрузку памяти в процентах
6,12,18,24	Определить MAC адрес компьютера

3. Дополнительное задание **Boost.DLL** (на оценку 9-10)

Boost - библиотеки классов, использующих функциональность языка C++ и предоставляющих удобный кроссплатформенный высокоуровневый интерфейс. Настоятельно рекомендуется посмотреть дополнительную информацию:

http://www.boost.org/

http://www.boost.org/doc/libs/1_61_0/doc/html/boost_dll.html

В настоящий момент в C++ нет кроссплатформенного решения для загрузки\выгрузки, импорта\экспорта и другой работы с DLL. С версии 1.61 в Boost появилась библиотека DLL, которая предоставляет удобный интерфейс для работы с .dll (.so) файлами. Boost не предоставляет какого-то нового механизма — это удобная обёртка над платформозависимыми решениями.

Для начала необходимо загрузить текущую версию Boost с сайта boost.org. Распакуйте скачанный архив, скомпилируйте (или скачайте уже скомпилированную библиотеку) и подключите IDE (в Visual Studio легко интегрируется). Если Вы используете Linux-based дистрибутивы, то в Package Manager есть чуть-чуть устаревшая версия пакета. Требуется версия не менее 1.61. Установка (на Debian-based) — sudo apt install libboost-dev.

Boost.DLL является header-only библиотекой, поэтому вам не нужно настраивать дополнительно линковщик. Достаточно просто включить нужный заголовочный файл в решение.

Примеры будут показаны с использованием IDE Clion, системой генерации build-скриптов Cmake, компилятор gcc 6.2.0 -std=c++14.

Для начала надо создать .so (Shared Object) файл, экспортировать в нём некоторые функции. Потом подключить .so в исполняемом файле и вызвать функцию из .so

Пример исходного кода .so

```
#include <iostream>
#include <string>
#include <boost/dll.hpp>

#define API extern "C" BOOST_SYMBOL_EXPORT

API void print(std::string param) noexcept
{
    std::cout << param << std::endl;
}
```

Макрос используется для манглинга в C-style (декорирования). Макрос раскрывается в зависимости от компилятора в нужный макрос на данной платформе.

Далее перед экспортируемой функцией просто используем данный макрос.

Код вызывающей программы:

```
#include <string>
#include <boost/dll.hpp>

int main()
{
    auto printFunc = boost::dll::import<void(std::string)>
        ("//path//to//dll",
            "print");
    printFunc(std::string("OOP"));
    return 0;
}
```

Здесь просто импортируем функцию с нужным именем из определённой DLL, и вызываем через полученный объект функцию.

Это наиболее высокоуровневый интерфейс, предоставляемый библиотекой. Если надо что-то более низкоуровневое — используйте boost::dll::shared_library.

Задание: используя Boost.DLL, реализовать отдельно .dll (.so) с заданными функциями (согласно своему варианту), подключить полученную DLL (.so) и использовать все реализованные функции. Задаётся только то, что должна делать функция — сигнатуру необходимо разработать сами.

Вариант	Функция
1	Вывести имена всех файлов в заданной директории
2	Функция, выводящая псевдослучайное число, использующая гауссово распределение.
3	Функция, перемещающая заданные файлы из заданной директории в другую заданную директорию
4	Функция, проверяющая, является ли заданный массив палиндромом
5	Функция, проверяющая, соответствуют ли все элементы массива унарному предикату
6	Вывести имена всех поддиректорий в заданной директории
7	Проверить, является ли заданный многоугольник выпуклым.
8	Проверить, пересекаются ли два отрезка
9	Найти наибольший по площади многоугольник из заданных
10	Функция, конвертирующая число в 7-ичную систему счисления из 10-ичной
11	Написать сортировку массива целых чисел [-10000, 10000] без использования операций сравнения.
12	Узнать кол-во процессов в системе
13	Найти N-ое простое число
14	Проверить, является ли число простым
15	Найти корень квадратный из числа, используя дихотомию
16	Вывести НОД двух чисел
17	Вывести НОК двух чисел.
18	Вывести текущую временную зону
19	Написать функцию, которая вернёт распарсенный

	массив данных из заданного XML файла на основе заданного тэга.
20	Написать функцию, удаляющую заданный файл в директории (если файла нет, ищем рекурсивно до глубины 2)
21	Вычислить определитель матрицы

Для справки:

GetLogicalDrives

Функция GetLogicalDrives возвращает число-битовую маску в которой храняться все доступные диски.

DWORD GetLogicalDrives(VOID);

Параметры: Эта функция не имеет параметров.

Возвращаемое значение: Если функция вызвана правильно, то она возвращает число-битовую маску в которой храняться все доступные диски (если 0 бит равен 1, то диск "A:" присутствует, и т.д.) Если функция вызвана не правильно, то она возвращает 0.

GetDriveType

Функция GetDriveType возвращает тип диска (removable, fixed, CD-ROM, RAM disk, или network drive).

UINT GetDriveType(LPCTSTR lpRootPathName);

Параметры: lpRootPathName

[in] Указатель на не нулевую стоку в которой хранится имя главной директории на диске. Обратный слэш должен присутствовать! Если lpRootPathName равно NULL, то функция использует текущую директорию.

Возвращаемое значение:

Функция возвращает тип диска. Могут быть следующие значения:

Значение	*Описание*
DRIVE_UNKNOWN	Не известный тип.
DRIVE_NO_ROOT_DIR	Не правильный путь.
DRIVE_REMOVABLE	Съёмный диск.
DRIVE_FIXED	Фиксированный диск.
DRIVE_REMOTE	Удалённый или network диск.
DRIVE_CDROM	CD-ROM диск.
DRIVE_RAMDISK	RAM диск.

GetVolumeInformation

Функция GetVolumeInformation возвращает информацию о файловой системе и дисках(директориях).

BOOL GetVolumeInformation(
LPCTSTR lpRootPathName, // имя диска(директории) [in]
LPTSTR lpVolumeNameBuffer, // название диска [out]
DWORD nVolumeNameSize, // длина буфера названия диска [in]
LPDWORD lpVolumeSerialNumber, // сериальный номер диска [out]
LPDWORD lpMaximumComponentLength, // максимальная длина фыйла [out]
LPDWORD lpFileSystemFlags, // опции файловой системы [out]
LPTSTR lpFileSystemNameBuffer, // имя файловой системы [out]
DWORD nFileSystemNameSize // длина буфера имени файл. сист. [in]
);

Возвращаемое значение: Если функция вызвана правильно, то она возвращает не нулевое значение(TRUE). Если функция вызвана не правильно, то она возвращает 0(FALSE).

GetDiskFreeSpaceEx

Функция GetDiskFreeSpaceEx выдаёт информацию о доступном месте на диске.

BOOL GetDiskFreeSpaceEx(
LPCTSTR lpDirectoryName, // имя диска(директории) [in]
PULARGE_INTEGER lpFreeBytesAvailable, // доступно для использования(байт) [out]
PULARGE_INTEGER lpTotalNumberOfBytes, // максимальный объём(в байтах) [out]
PULARGE_INTEGER lpTotalNumberOfFreeBytes // свободно на диске(в байтах) [out]
);

Возвращаемое значение: Если функция вызвана правильно, то она возвращает не нулевое значение(TRUE). Если функция вызвана не правильно, то она возвращает 0(FALSE).

GlobalMemoryStatus

Функция GlobalMemoryStatus возвращает информацию о используемой системой памяти.

```
VOID GlobalMemoryStatus(
LPMEMORYSTATUS lpBuffer // указатель на структуру MEMORYSTATUS
);

typedef struct _MEMORYSTATUS {
DWORD dwLength; // длина структуры в байтах
DWORD dwMemoryLoad; // загрузка памяти в процентах
SIZE_T dwTotalPhys; // максимальное количество физической памяти в байтах
SIZE_T dwAvailPhys; // свободное количество физической памяти в байтах
SIZE_T dwTotalPageFile; // макс. кол. памяти для программ в байтах
SIZE_T dwAvailPageFile; // свободное кол. памяти для программ в байтах
SIZE_T dwTotalVirtual; // максимальное количество виртуальной памяти в
байтах
SIZE_T dwAvailVirtual; // свободное количество виртуальной памяти в байтах
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

Возвращаемое значение: Эта функция не возвращает параметров

GetComputerName, GetUserNameA
Функция GetComputerName возвращает NetBIOS имя локального компьютера.

```
BOOL GetComputerName(
LPTSTR lpBuffer, // имя локального компьютера( длина буфера равна
MAX_COMPUTERNAME_LENGTH + 1 ) [out]
LPDWORD lpnSize // размер буфера ( лучше поставить
MAX_COMPUTERNAME_LENGTH + 1 ) [out/in]
);
```

Функция GetUserName возвращает имя текущего узера.

```
BOOL GetUserName(
LPTSTR lpBuffer, // имя юзера( длина буфера равна UNLEN + 1 ) [out]
LPDWORD nSize // размер буфера ( лучше поставить UNLEN + 1 ) [out/in]
);
```

Возвращаемые значения: Если функции вызваны правильно, то они возвращают не нулевое значение(TRUE). Если функции вызваны не правильно, то они возвращают 0(FALSE).

GetSystemDirectory, GetTempPath, GetWindowsDirectory, GetCurrentDirectory

Функция GetSystemDirectory возвращает путь к системной директории.

```
UINT GetSystemDirectory(
LPTSTR lpBuffer, // буфер для системной директории [out]
UINT uSize // размер буфера [in]
);
```

Возвращаемое значение: Эта функция возвращает размер буфера для системной директории не включая нулевого значения в конце, если она вызвана правильно. Если функция вызвана не правильно, то она возвращает 0.

Теория

Введение

Динамически подключаемые библиотеки (dynamic-link libraries, DLL) — краеугольный камень операционной системы Windows, начиная с самой первой ее версии. В DLL содержатся все функции Windows API. Три самые важные DLL: Kernel32.dll (управление памятью, процессами и потоками), User32.dll (поддержка пользовательского интерфейса, в том числе функции, связанные с созданием окон и передачей сообщений) и GDI32.dll (графика и вывод текста). В Windows есть и другие DLL, функции которых предназначены для более специализированных задач. Например, в AdvAPI32.dll содержатся функции для защиты объектов, работы с реестром и регистрации событий, в ComDlg32.dll ~ стандартные диалоговые окна (вроде File Open и File Save), а ComCrl32 dll поддерживает стандартные элементы управления. Вот лишь некоторые из причин, по которым нужно применять DLL:

• Расширение функциональности приложения. DLL можно загружать в адресное пространство процесса динамически, что позволяет приложению, определив, какие действия от него требуются, подгружать нужный код.

• Возможность использования разных языков программирования. У Вас есть выбор, на каком языке писать ту или иную часть приложения Так, пользовательский интерфейс приложения Вы скорее всего будете создавать на Microsoft Visual Basic, но прикладную логику лучше всего реализовать на C++. Программа на Visual Basic может загружать DLL, написанные на C++, Коболе, Фортране и др.

• Более простое управление проектом. Если в процессе разработки программного продукта отдельные его модули создаются разными

группами, то при использовании DLL таким проектом управлять гораздо проще. Однако конечная версия приложения должна включать как можно меньше файлов

• Экономия памяти. Если одну и ту же DLL использует несколько приложений, в оперативной памяти может храниться только один ее экземпляр, доступный этим приложениям. Пример — DLL-версия библиотеки C/C++ Ею пользуются многие приложения. Если всех их скомпоновать со статически подключаемой версией этой библиотеки, то код таких функций, как sprintf, strcpy, malloc и др., будет многократно дублироваться в памяти Но если они компонуются с DLL-версией библиотеки C/C++, в памяти будет присутствовать лишь одна копия кода этих функций, что позволит гораздо эффективнее использовать оперативную память.

• Разделение ресурсов. DLL могут содержать такие ресурсы, как шаблоны диалоговых окон, строки, значки и битовые карты (растровые изображения). Эти ресурсы доступны любым программам

• Упрощение локализации. DI,L нередко применяются для локализации приложений. Например, приложение, содержащее только код без всяких компонентов пользовательского интерфейса, может загружать DLL с компонентами локализованного интерфейса

• Решение проблем, связанных с особенностями различных платформ. В разных версиях Windows содержатся разные наборы функций. Зачастую разработчикам нужны новые функции, существующие в той версии системы, которой они пользуются. Если Ваша версия Windows не поддерживает эти функции, Вам не удастся запустить такое приложение: загрузчик попросту откажется его запускать.

Согласование интерфейсов

При использовании собственных библиотек или библиотек независимых разработчиков придется обратить внимание на согласование вызова функции с ее прототипом.

Если бы мир был совершенен, то программистам не пришлось бы беспокоиться о согласовании интерфейсов функций при подключении библиотек ґ все они были бы одинаковыми. Однако мир далек от совершенства, и многие большие программы написаны с помощью различных библиотек без C++.

По умолчанию в Visual C++ интерфейсы функций согласуются по правилам C++. Это значит, что параметры заносятся в стек справа налево, вызывающая программа отвечает за их удаление из стека при выходе из функции и расширении ее имени. Расширение имен (name mangling) позволяет редактору связей различать перегруженные функции, т.е. функции

с одинаковыми именами, но разными списками аргументов. Однако в старой библиотеке С функции с расширенными именами отсутствуют.

Хотя все остальные правила вызова функции в С идентичны правилам вызова функции в С++, в библиотеках С имена функций не расширяются. К ним только добавляется впереди символ подчеркивания (_).

Если необходимо подключить библиотеку на С к приложению на С++, все функции из этой библиотеки придется объявить как внешние в формате С:

```
extern "C" int MyOldCFunction(int myParam);
```

Объявления функций библиотеки обычно помещаются в файле заголовка этой библиотеки, хотя заголовки большинства библиотек С не рассчитаны на применение в проектах на С++. В этом случае необходимо создать копию файла заголовка и включить в нее модификатор extern "C" к объявлению всех используемых функций библиотеки. Модификатор extern "C" можно применить и к целому блоку, к которому с помощью директивы #tinclude подключен файл старого заголовка С. Таким образом, вместо модификации каждой функции в отдельности можно обойтись всего тремя строками:

```
extern "C"
        {
                #include "MyCLib.h"
        }
```

В программах для старых версий Windows использовались также соглашения о вызове функций языка PASCAL для функций Windows API. В новых программах следует использовать модификатор winapi, преобразуемый в _stdcall. Хотя это и не стандартный интерфейс функций С или С++, но именно он используется для обращений к функциям Windows API. Однако обычно все это уже учтено в стандартных заголовках Windows.

Загрузка неявно подключаемой DLL

При запуске приложение пытается найти все файлы DLL, неявно подключенные к приложению, и поместить их в область оперативной памяти, занимаемую данным процессом. Поиск файлов DLL операционной системой осуществляется в следующей последовательности:

- Каталог, в котором находится EXE-файл.
- Текущий каталог процесса.
- Системный каталог Windows.

Если библиотека DLL не обнаружена, приложение выводит диалоговое окно с сообщением о ее отсутствии и путях, по которым осуществлялся поиск. Затем процесс отключается.

Если нужная библиотека найдена, она помещается в оперативную память процесса, где и остается до его окончания. Теперь приложение может обращаться к функциям, содержащимся в DLL.

Динамическая загрузка и выгрузка DLL

Вместо того, чтобы Windows выполняла динамическое связывание с DLL при первой загрузке приложения в оперативную память, можно связать программу с модулем библиотеки во время выполнения программы (при таком способе в процессе создания приложения не нужно использовать библиотеку импорта). В частности, можно определить, какая из библиотек DLL доступна пользователю, или разрешить пользователю выбрать, какая из библиотек будет загружаться. Таким образом можно использовать разные DLL, в которых реализованы одни и те же функции, выполняющие различные действия. Например, приложение, предназначенное для независимой передачи данных, сможет в ходе выполнения принять решение, загружать ли DLL для протокола TCP/IP или для другого протокола.

Загрузка обычной DLL

Первое, что необходимо сделать при динамической загрузке DLL, - это поместить модуль библиотеки в память процесса. Данная операция выполняется с помощью функции ::LoadLibrary, имеющей единственный аргумент имя загружаемого модуля. Соответствующий фрагмент программы должен выглядеть так:

```
HINSTANCE hMyDll;
        if((hMyDll=::LoadLibrary("MyDLL"))==NULL) {
/* не удалось загрузить DLL */ }
        else {
/* приложение имеет право пользоваться функциями DLL через hMyDll */
}
```

Стандартным расширением файла библиотеки Windows считает .dll, если не указать другое расширение. Если в имени файла указан и путь, то только он будет использоваться для поиска файла. В противном случае Windows будет искать файл по той же схеме, что и в случае неявно подключенных DLL, начиная с каталога, из которого загружается exe-файл, и продолжая в соответствии со значением PATH.

Когда Windows обнаружит файл, его полный путь будет сравнен с путем библиотек DLL, уже загруженных данным процессом. Если обнаружится тождество, вместо загрузки копии приложения возвращвется дескриптор уже подключенной библиотеки.

Если файл обнаружен и библиотека успешно загрузилась, функция ::LoadLibrary возвращает ее дескриптор, который используется для доступа к функциям библиотеки.

Перед тем, как использовать функции библиотеки, необходимо получить их адрес. Для этого сначала следует воспользоваться директивой typedef для определения типа указателя на функцию и определить переменую этого нового типа, например:

```
   // тип PFN_MyFunction будет объявлять указатель на функцию,
          // принимающую указатель на символьный буфер и выдающую
значение типа int
          typedef int (WINAPI *PFN_MyFunction)(char *);

          PFN_MyFunction pfnMyFunction;
```

Затем следует получить дескриптор библиотеки, при помощи которого и определить адреса функций, например адрес фунции с именем MyFunction:

```
   hMyDll=::LoadLibrary("MyDLL");

pfnMyFunction=(PFN_MyFunction)::GetProcAddress(hMyDll,"MyFunction");

          int iCode=(*pfnMyFunction)("Hello");
```

Адрес функции определяется при помощи функции ::GetProcAddress, ей следует передать имя библиотеки и имя функции. Последнее должно передаваться в том виде, в котором эксаортируется из DLL.

Можно также сослаться на функцию по порядковому номеру, по которому она экспортируется (при этом для создания библиотеки должен использоваться def-файл, об этом будет рассказано далее):

```
pfnMyFunction=(PFN_MyFunction)::GetProcAddress(hMyDll,
                                        MAKEINTRESOURCE(1));
```

После завершения работы с библиотекой динамической компоновки, ее можно выгрузить из памяти процесса с помощью функции ::FreeLibrary:

```
          ::FreeLibrary(hMyDll);
```

Загрузка MFC-расширений динамических библиотек

При загрузке MFC-расширений для DLL (подробно о которых рассказывается далее) вместо функций LoadLibraryи FreeLibrary используются функции AfxLoadLibrary и AfxFreeLibrary. Последние почти идентичны функциям Win32 API. Они лишь гарантируют дополнительно, что структуры MFC, инициализированные расширением DLL, не были запорчены другими потоками.

Ресурсы DLL

Динамическая загрузка применима и к ресурсам DLL, используемым MFC для загрузки стандартных ресурсов приложения. Для этого сначала необходимо вызвать функцию LoadLibrary и разместить DLL в памяти. Затем с помощью функции AfxSetResourceHandle нужно подготовить окно программы к приему ресурсов из вновь загруженной библиотеки. В противном случае ресурсы будут загружаться из файлов, подключенных к выполняемому файлу процесса. Такой подход удобен, если нужно использовать различные наборы ресурсов, например для разных языков.

Замечание. С помощью функции LoadLibrary можно также загружать в память исполняемые файлы (не запускать их на выполнение!). Дескриптор выполняемого модуля может затем использоваться при обращении к функциям FindResource и LoadResource для поиска и загрузки ресурсов приложения. Выгружают модули из памяти также при помощи функции FreeLibrary.

Пример обычной DLL и способов загрузки

Приведем исходный код динамически подключаемой библиотеки, которая называется MyDLL и содержит одну функцию MyFunction, которая просто выводит сообщение.

Сначала в заголовочном файле определяется макроконтстанта EXPORT. Использование этого ключевого слова при определении некоторой функции динамически подключаемой билиотеке позволяет сообщить компоновщику, что эта функция доступна для использования другими программами, в результате чего он заносит ее в библилтеку импорта. Кроме этого, такая функция, точно так же, как и оконная процедура, должна определяться с помощью константы CALLBACK:

```
    ------------------------------MyDLL.h-----------------------------
------
              #define EXPORT extern "C"__declspec (dllexport)
              EXPORT int CALLBACK MyFunction(char *str);
```

Файл библиотеки также несколько отличается от обычных файлов на языке C для Windows. В нем вместо функции WinMain имеется функция DllMain. Эта функция используется для выполнения инициализации, о чем будет рассказано позже. Для того, чтобы библиотека осталась после ее загрузки в памяти, и можно было вызывать ее функции, необходимо, чтобы ее возвращаемым значением было TRUE:

```
    -------------------------------- MyDLL.c-------------------------
-------------

        #include <windows.h>
        #include "MyDLL.h"


        int WINAPI DllMain(HINSTANCE hInstance, DWORD fdReason, PVOID
pvReserved)
        {
                return TRUE;
        }
        EXPORT int CALLBACK MyFunction(char *str)
        {
                MessageBox(NULL,str,"Function from DLL",MB_OK);
                return 1;
        }
```

После трансляции и компоновки этих файлов появятся два файла Ї MyDLL.dll (сама динамически подключаемая библиотека) и MyDLL.lib (ее библиотека импорта).

Пример неявного поключения DLL приложением

Приведем теперь исходный код простого приложения, которое использует функцию MyFunction из библиотеки MyDLL.dll:

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


        int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance,
                        LPSTR lpCmdLine, int nCmdShow)
        {
                int iCode=MyFunction("Hello");
                return 0;
        }
```

Эта программа выглядит как обычная программ для Windows, чем она в сущности и является. Тем не менее, следует обратить внимание, что в исходный ее текст помимо вызова функции MyFunction из DLL-библиотеки включен и заголовочный файл этой библиотеки MyDLL.h. Также необходимо на этапе компоновки приложения подключить к нему библиотеку импорта MyDLL.lib (процесс неявного подключения DLL к исполняемому модулю).

Чрезвычайно важно понимать, что сам код функции MyFunction не включается в файл MyApp.exe. Вместо этого там просто имеется ссылка на файл MyDLL.dll и ссылка на функцию MyFunction, которая находится в этом файле. Файл MyApp.exe требует запуска файла MyDLL.dll.

Заголовочный файл MyDLL.h включен в файл с исходным текстом программы MyApp.c точно так же, как туда включен файл windows.h. Включение библиотеки импорта MyDLL.lib для компоновки аналогично включению туда всех библиотек импорта Windows. Когда програма MyApp.exe работает, она подключается к библиотеке MyDLL.dll точно так же, как ко всем стандартным динамически подключаемым библиотекам Windows.

Пример динамической загрузки DLL приложением

Приведем теперь полностью исходный код простого приложения, которое использует функцию MyFunction из библиотеки MyDLL.dll, используя динамическую загрузку библиотеки:

```
#include <windows.h>
typedef int (WINAPI *PFN_MyFunction)(char *);


int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow)
{
        HINSTANCE hMyDll;
        if((hMyDll=LoadLibrary("MyDLL"))==NULL) return 1;


        PFN_MyFunction pfnMyFunction;

pfnMyFunction=(PFN_MyFunction)GetProcAddress(hMyDll,"MyFunction");
        int iCode=(*pfnMyFunction)("Hello");


        FreeLibrary(hMyDll);
        return 0;
}
```

Создание DLL

Теперь, познакомившись с принципами работы библиотек DLL в приложениях, рассмотрим способы их создания. При разработке приложении функции, к которым обращается несколько процессов, желательно размещать в DLL. Это позволяет более рационально использовать память в Windows.

Проще всего создать новый проект DLL с помощью мастера AppWizard, который автоматически выполняет многие операции. Для простых DLL, таких как рассмотренные в этой главе, необходимо выбрать тип проекта Win32 Dynamic-Link Library. Новому проекту будут присвоены все необходимые параметры для создания библиотеки DLL. Файлы исходных текстов придется добавлять к проекту вручную.

Если же планируется в полной мере использовать функциональные возможности MFC, такие как документы и представления, или намерены создать сервер автоматизации OLE, лучше выбрать тип проекта MFC AppWizard (dll). В этом случае, помимо присвоения проекту параметров для подключения динамических библиотек, мастер проделает некоторую дополнительную работу. В проект будут добавлены необходимые ссылки на библиотеки MFC и файлы исходных текстов, содержащие описание и реализацию в библиотеке DLL объекта класса приложения, производного от CWinApp.

Иногда удобно сначала создать проект типа MFC AppWizard (dll) в качестве тестового приложения, а затем ґ библиотеку DLL в виде его составной части. В результате DLL в случае необходимости будет создаваться автоматически.

Функция DllMain

Большинство библиотек DLL ґ просто коллекции практически независимых друг от друга функций, экспортируемых в приложения и используемых в них. Кроме функций, предназначенных для экспортирования, в каждой библиотеке DLL есть функция DllMain. Эта функция предназначена для инициализации и очистки DLL. Она пришла на смену функциям LibMain и WEP, применявшимся в предыдущих версиях Windows. Структура простейшей функции DllMain может выглядеть, например, так:

```
    BOOL WINAPI DllMain (HANDLE hInst,DWORD dwReason, LPVOID
IpReserved)
        {
```

```
                BOOL bAllWentWell=TRUE;
                switch (dwReason)
                {
                   case DLL_PROCESS_ATTACH:
// Инициализация процесса.
                                break;
                   case DLL_THREAD_ATTACH: // Инициализация потока.
                                break;
                   case DLL_THREAD_DETACH: // Очистка структур потока.
                                break;
                  case DLL_PROCESS_DETACH: // Очистка структур
процесса.
                                break;
                }
                if(bAllWentWell)       return TRUE;
                else                   return FALSE;
        }
```

Функция DllMain вызывается в нескольких случаях. Причина ее вызова определяется параметром dwReason, который может принимать одно из следующих значений.

При первой загрузке библиотеки DLL процессом вызывается функция DllMain с dwReason, равным DLL_PROCESS_ATTACH. Каждый раз при создании процессом нового потока DllMainO вызывается с dwReason, равным DLL_THREAD_ATTACH (кроме первого потока, потому что в этом случае dwReason равен DLL_PROCESS_ATTACH).

По окончании работы процесса с DLL функция DllMain вызывается с параметром dwReason, равным DLL_PROCESS_DETACH. При уничтожении потока (кроме первого) dwReason будет равен DLL_THREAD_DETACH.

Все операции по инициализации и очистке для процессов и потоков, в которых нуждается DLL, необходимо выполнять на основании значения dwReason, как было показано в предыдущем примере. Инициализация процессов обычно ограничивается выделением ресурсов, совместно используемых потоками, в частности загрузкой разделяемых файлов и инициализацией библиотек. Инициализация потоков применяется для настройки режимов, свойственных только данному потоку, например для инициализации локальной памяти.

В состав DLL могут входить ресурсы, не принадлежащие вызывающему эту библиотеку приложению. Если функции DLL работают с ресурсами DLL, было бы, очевидно, полезно сохранить где-нибудь в укромном месте дескриптор hInst и использовать его при загрузке ресурсов из DLL. Указатель lpReserved зарезервирован для внутреннего использования Windows. Следовательно, приложение не должно претендовать на него. Можно лишь проверить его значение. Если библиотека DLL была загружена динамически,

оно будет равно NULL. При статической загрузке этот указатель будет ненулевым.

В случае успешного завершения функция DllMain должна возвращать TRUE. В случае возникновения ошибки возвращается FALSE, и дальнейшие действия прекращаются.

Замечание. Если не написать собственной функции DllMain(), компилятор подключит стандартную версию, которая просто возвращает TRUE.

Экспортирование функций из DLL

Чтобы приложение могло обращаться к функциям динамической библиотеки, каждая из них должна занимать строку в таблице экспортируемых функций DLL. Есть два способа занести функцию в эту таблицу на этапе компиляции.

Метод __declspec (dllexport)

Можно экспортировать функцию из DLL, поставив в начале ее описания модификатор __declspec (dllexport) . Кроме того, в состав MFC входит несколько макросов, определяющих __declspec (dllexport), в том числе AFX_CLASS_EXPORT, AFX_DATA_EXPORT и AFX_API_EXPORT.

Метод __declspec применяется не так часто, как второй метод, работающий с файлами определения модуля (.def), и позволяет лучше управлять процессом экспортирования.

Файлы определения модуля

Синтаксис файлов с расширением .def в Visual C++ достаточно прямолинеен, главным образом потому, что сложные параметры, использовавшиеся в ранних версиях Windows, в Win32 более не применяются. Как станет ясно из следующего простого примера, .def-файл содержит имя и описание библиотеки, а также список экспортируемых функций:

MyDLL.def

```
LIBRARY          "MyDLL"
DESCRIPTION    'пример DLL-библиотеки


EXPORTS
    MyFunction    @1
```

В строке экспорта функции можно указать ее порядковый номер, поставив перед ним символ @. Этот номер будет затем использоваться при обращении к GetProcAddress (). На самом деле компилятор присваивает порядковые номера всем экспортируемым объектам. Однако способ, которым он это делает, отчасти непредсказуем, если не присвоить эти номера явно.

В строке экспорта можно использовать параметр NONAME. Он запрещает компилятору включать имя функции в таблицу экспортирования DLL:

```
MyFunction    @1 NONAME
```

Иногда это позволяет сэкономить много места в файле DLL. Приложения, использующие библитеку импортирования для неявного подключения DLL, не «заметят¬ разницы, поскоьку при неявном подключении порядковые номера используются автоматически. Приложениям, загружающим библиотеки DLL динамически, потребуется передавать в GetProcAddress порядковый номер, а не имя функции.

При использовании вышеприведенного def-файл описания экспортируемых функций DLL-библиотеки может быть,например, не таким:

```
#define EXPORT extern "C" __declspec (dllexport)
EXPORT int CALLBACK MyFunction(char *str);
```

а таким:

```
extern "C" int CALLBACK MyFunction(char *str);
```

Экспортирование классов

Создание .def-файла для экспортирования даже простых классов из динамической библиотеки может оказаться довольно сложным делом. Понадобится явно экспортировать каждую функцию, которая может быть использована внешним приложением.

Если взглянуть на реализованный в классе файл распределения памяти, в нем можно заметить некоторые весьма необычные функции. Оказывается, здесь есть неявные конструкторы и деструкторы, функции, объявленные в макросах MFC, в частности _DECLARE_MESSAGE_MAP, а также функции, которые написанные программистом.

Хотя можно экспортировать каждую из этих функций в отдельности, есть более простой способ. Если в объявлении класса воспользоваться

макромодификатором AFX_CLASS_EXPORT, компилятор сам позаботится об экспортировании необходимых функций, позволяющих приложению использовать класс, содержащийся в DLL.

Память DLL

В отличие от статических библиотек, которые, по существу, становятся частью кода приложения, библиотеки динамической компоновки в 16-разрядных версиях Windows работали с памятью несколько иначе. Под управлением Win 16 память DLL размещалась вне адресного пространства задачи. Размещение динамических библиотек в глобальной памяти обеспечивало возможность совместного использования их различными задачами.

В Win32 библиотека DLL располагается в области памяти загружающего ее процесса. Каждому процессу предоставляется отдельная копия "глобальной" памяти DLL, которая реинициализируется каждый раз, когда ее загружает новый процесс. Это означает, что динамическая библиотека не может использоваться совместно, в общей памяти, как это было в Winl6.

И все же, выполнив ряд замысловатых манипуляций над сегментом данных DLL, можно создать общую область памяти для всех процессов, использующих данную библиотеку.

Допустим, имеется массив целых чисел, который должен использоваться всеми процессами, загружающими данную DLL. Это можно запрограммировать следующим образом:

```
#pragma data_seg(".myseg")
        int sharedlnts[10] ;
        // другие переменные общего пользования
        #pragma data_seg()
        #pragma comment(lib, "msvcrt" "-SECTION:.myseg,rws");
```

Все переменные, объявленные между директивами #pragma data_seg(), размещаются в сегменте .myseg. Директива #pragma comment () ґ не обычный комментарий. Она дает указание библиотеке выполняющей системы С пометить новый раздел как разрешенный для чтения, записи и совместного доступа.

Полная компиляция DLL

Если проект динамической библиотеки создан с помощью AppWizard и .def-файл модифицирован соответствующим образом этого достаточно. Если же файлы проекта создаются вручную или другими способами без помощи AppWizard, в командную строку редактора связей следует включить

параметр /DLL. В результате вместо автономного выполняемого файла будет создана библиотека DLL.

Если в .def-файле есть строка LIBRART, указывать явно параметр /DLL в командной строке редактора связей не нужно.