А.В. Леденёв, И.А. Семёнов, В.А. Сторожевых

Динамически загружаемые библиотеки: структура, архитектура и применение (часть 2)[1]

С формальной точки зрения Dynamic Link Library (DLL) (динамически загружаемая библиотека) — особым образом оформленный относительно независимый блок исполняемого кода. DLL используются множеством приложений. Все приложения для ОС Windows так или иначе используют динамические библиотеки.

Данный материал является продолжением первой части работы, посвященной особенностям реализации DLL в различных средах и для различных целей, опубликованной в № 2 журнала за этот год.

DLL, содержащая только ресурсы

Одним из очень полезных свойств DLL (как, впрочем, и других файлов формата **Portable Executable**) является их способность содержать в себе не только код, но и ресурсы. *Ресурсы* — это разного рода информация, необходимая программе в работе. По сравнению с данными в коде ресурсы намного быстрее и удобнее добавлять, удалять и редактировать.

Зачастую в состав программного продукта входят динамические библиотеки, не содержащие кода, а включающие исключительно ресурсы. Такие библиотеки позволяют, например, легко переводить интерфейс программы на другие языки или менять ее оформление. Для этого не нужно исправлять код программы, достаточно лишь отредактировать соответствующие ресурсы в нужной DLL. Кроме того, библиотеки DLL позволяют нескольким приложениям совместно использовать один и тот же набор ресурсов.

Наиболее распространенными типами ресурсов, хранимых в DLL, являются:

- **VERSION** — содержит информацию о версии DLL. Этот ресурс предназначен для решения проблемы «Ада DLL» (DLL Hell), однако из-за того, что он не является обязательным и никак не влияет на исполнение кода DLL, проблема решается лишь частично;
- **BITMAP** — битовый образ (картинка). Имеет огромное количество областей применения: от картинок на кнопках панели управления до персонажей игр;
- **CURSOR** — изображение курсора мыши;
- **ICON** — иконки. Применяются во многих элементах управления: в меню, списках (**List Control**), деревьях (**Tree View**) и др.;
- **MENU** — меню;
- **ACCELERATOR** — настройка «быстрых клавиш». Позволяет настраивать комбинации клавиш для доступа к различным функциям;
- **STRINGTABLE** — таблица строк. Служит для сохранения любой текстовой информации программы. Используется для быстрого перевода интерфейса программы на другие языки.

Для создания DLL, содержащих только ресурсы, необходимо указать опцию линковщика **/noentry**. Эта опция указывает линковщику, что не нужно создавать точку входа (**entry point**) в библиотеке (обычно это функция **DllMain**).

[1] Заключительная часть материала будет опубликована в № 6 журнала за этот год. — *Прим. ред.*

Кроме того, динамически загружаемая библиотека, содержащая лишь ресурсы, может быть загружена только явно (**explicitly**).

Для примера давайте создадим простую ресурсную DLL.

Visual C++ 6.0

Создадим новый проект в среде **Visual C++ 6.0**, выбрав тип приложения **«Win32 Dynamic-Link Library»**. Выбираем опцию **«An empty DLL project»** — код нам не нужен.

Добавим в динамическую библиотеку ресурсы. Для этого выберем в меню пункт **«Insert->Resource...»**. Для примера сначала добавим в DLL картинку (**bitmap**). Для этого в появившемся диалоге **«Insert Resource»** нажимаем кнопку **«Import...»** и выбираем на диске bmp-файл с картинкой.

Замечание.

В качестве ресурса в DLL можно добавить картинку (**bitmap**) с цветовым разрешением до 24 бит на цвет, но среда **Visual C++ 6.0** позволяет отображать и редактировать только ресурсы с разрешением не более 8 бит на цвет (т.е. не более 256 цветов) (рис. 1).

Таким образом, картинки с более высоким цветовым разрешением можно загружать и использовать, но нельзя просматривать и редактировать внутри среды.

Итак, картинка добавлена. Теперь нужно сохранить проект. При сохранении среда **Visual C++ 6.0** предложит выбрать имя для файла ресурсного скрипта (**resource script**). *Ресурсный скрипт* — это файл, особым образом описывающий ресурсы проекта. В нашем примере назовем ресурсный файл **dlltest.rc**. Помимо файла ресурсного скрипта, автоматически создается файл **resource.h**, описывающий индексы ресурсов. Дело в том, что в отличие от функций ресурсы импортируются исключительно по целочисленным индексам. Для добавленной нами картинки создан индекс 101:

```
#define    IDB_BITMAP1        101
```

Добавим в проект еще один ресурс — например, информацию о версии (**version info**). Для этого снова вызовем диалог **«Insert Resource»**, выберем тип ресурса **«Version»** и нажмем кнопку **«New»**. Ресурс добавляется в проект и открывается во встроенном редакторе. Здесь мы можем изменить версию DLL и ее описание. Обратите внимание: при сохранении ресурс **«Version»** записывается в ресурсный скрипт, но не записывается в **resource.h**. Дело в том, что это особый тип ресурса — его нельзя загрузить явно. Информация о версии DLL интерпретируется системой.

Другие типы ресурсов добавляются в проект и редактируются аналогично.

Замечание.

Не рекомендуется вручную редактировать файлы ресурсного скрипта и **resource.h**, не изучив досконально их формат. Если этот формат будет нарушен, среда **Visual C++ 6.0** может отказаться добавлять в проект новые ресурсы.

Созданные файлы (ресурсный скрипт и **resource.h**) следует добавить в проект,

Рис. 1. Предупреждающее сообщение среды **Visual C++**

чтобы указать среде, откуда брать описания ресурсов при сборке. Теперь необходимо собрать DLL. Пытаемся собрать проект и видим примерно такое сообщение об ошибке:

```
LINK:error LNK2001:unresolved external symbol
__DllMainCRTStartup@12

Debug/DLLTest1.dll:fatal error LNK1120:1
unresolved externals

Error executing link.exe.
```

Дело в том, что линковщик пытается найти в файлах проекта точку входа в DLL — функцию **DllMain**, но ему это не удается, потому как у нас нет ни одного файла с кодом. Да и не нужна нам эта точка входа. Для того чтобы убедить в этом линковщика, открываем **«Project->Settings…->Link»** и добавляем в **«Project Options»** ключ **«/noentry»**. Пересобираем проект — все в порядке, DLL создана.

Visual C++ 7.0

Создадим новый проект в среде **Visual Studio.NET**, выбрав тип проекта **«Win32 Project»**. В появившемся диалоге в разделе **«Application Settings»** выберем пункт **«DLL»**, а в **«Additional options»** — пункт **«Empty project»**.

Добавление ресурсов в проект происходит практически так же, как и в **Visual C++ 6.0**, за тем лишь исключением, что снято ограничение на цветовое разрешение картинок, и файлы ресурсного скрипта и **resource.h** добавляются в проект автоматически.

Для того чтобы указать линковщику, что не нужно искать точку входа в DLL — функцию **DllMain**, — выбираем **«Project->Properties...»**, на вкладке **«Linker->Advanced»** в пункте **«Resource Only DLL»** выбираем **«Yes (/noentry)»**.

C++Builder 6.0 и Delphi 6.0

Существуют два основных способа разработки ресурсных DLL в среде программирования **C++Builder**. Первый способ — использование встроенной утилиты **Image Editor**, второй — редактирование и компиляция RC-файлов вручную. У каждого из этих способов есть свои плюсы и минусы.

Создадим новый проект в среде **C++Builder**, выбрав тип проекта **DLL Wizard** (**«File->New->Other…->DLL Wizard»**). По умолчанию вновь созданный проект включает два файла: cpp-файл и res-файл. Поскольку мы создаем DLL, содержащую только ресурсы, то cpp-файл можно удалить из проекта (**«Project->Remove from Project»**). В случае **Delphi** процесс полностью аналогичен, за тем лишь исключением, что нет необходимости удалять cpp-файл из проекта.

Res-файл содержит ресурсы проекта в бинарном виде. Для его непосредственного редактирования можно использовать встроенную утилиту **Image Editor** (**«Tools->Image Editor»**) (рис. 2). Чтобы открыть res-файл, выбираем **«File->Open...»**.

Рис. 2. Окно редактора **Image Editor**

В созданном по умолчанию res-файле содержится только один ресурс — иконка проекта. Для того чтобы просмотреть/отредактировать ресурс, достаточно дважды кликнуть по его идентификатору.

Добавим в res-файл проекта новую картинку. Для этого выберем **«Resource->New->Bitmap»**, выберем параметры картинки

и нажмем OK. После изменения res-файла необходимо пересобрать проект, чтобы обновленные ресурсы попали в DLL. Такой способ редактирования ресурсов достаточно удобен, однако имеются ограничения — в **Image Editor** можно оперировать только тремя типами ресурсов: картинка (**bitmap**), иконка (**icon**) и курсор (**cursor**).

Другой способ создания res-файлов — создание rc-файла и дальнейшая его компиляция. Формат rc-файлов соответствует аналогичному формату в средах **Microsoft**, его описание легко найти в **MSDN** либо в Интернете.

Для примера создадим простой rc-файл, добавив в него файл **wav**:

```
IDR_WAV WAV "sound.wav"
```

Для успешной компиляции необходимо, чтобы файл **sound.wav** находился в той же директории, что и rc-файл. Компилировать можно при помощи утилиты **brcc32.exe**, однако намного проще просто добавить rc-файл в проект (**«Project->Add to Project...»**) — тогда он будет компилироваться автоматически.

Замечание.

При компиляции даже при отключенных установках отладки (**«Full Release»**) в DLL добавляется отладочный код. Авторам не известно, как можно отключить эту опцию.

Visual Basic

Средствами «чистого» **Visual Basic** создать ресурсную DLL невозможно. Как уже упоминалось, **Visual Basic** поддерживает создание только **ActiveX DLL**, т. е. DLL, содержащих COM-объекты (впрочем, ничто не мешает **ActiveX DLL** нести в себе ресурсы; однако речь в данном случае идет о чисто ресурсной DLL, т. е. о DLL, не содержащей исполняемого кода).

Тем не менее с помощью дополнительных утилит сторонних фирм все-таки оказывается возможным создать и ресурсную DLL. Отправим любознательного читателя на сайт http://www.vbadvance.com/.

Как получить доступ к ресурсам?

Для доступа к ресурсам, содержащимся в динамически загружаемой библиотеке, используются следующие функции:

- **FindResource**
- **FindResourceEx**
- **LoadResource**
- **LoadAccelerators**
- **LoadBitmap**
- **LoadCursor**
- **LoadIcon**
- **LoadMenu**
- **LoadString**

Итак, по порядку. Сначала необходимо найти требуемый ресурс в библиотеке. Для этого служат функции **FindResource** и **FindResourceEx**. Их прототипы представлены ниже.

```
HRSRC FindResource(
    HMODULE hModule,
    LPCTSTR lpName,
    LPCTSTR lpType
);

HRSRC FindResourceEx(
    HMODULE hModule,
    LPCTSTR lpName,
    LPCTSTR lpType,
    WORD wLanguage
);
```

где **hModule** — идентификатор модуля (например, полученный функцией **LoadLibrary**); **lpName** — как уже упоминалось выше, доступ к ресурсам, в отличие от функций DLL, осуществляется исключительно по целочисленным идентификаторам. Идентификатор можно передать через параметр **lpName** двумя способами: либо в виде строки, содержащей символ # и десятичный идентификатор ресурса (например, #254), либо с помощью макроса **MAKEINTRESOURCE** (например, **MAKEINTRESOURCE(254)**). Последний способ предпочтительнее, так как он ускоряет доступ к ресурсам;

lpType — тип ресурса. Существует множество констант для основных типов ресурсов в **Windows**. Все эти константы имеют префикс RT_, например: **RT_BITMAP** или **RT_DIALOG**. Подробнее об этих константах можно узнать в **MSDN**;
wLanguage — идентификатор языка ресурса. Для создания идентификатора используется макрос **MAKELANGID**:

```
WORD MAKELANGID(
       USHORT usPrimaryLanguage,
       USHORT usSubLanguage
);
```

Константы для **usPrimaryLanguage** и **usSubLanguage** содержатся в соответствующих разделах **MSDN**.

MAKELANGID(LANG_NEUTRAL, SUBLANG_NEUTRAL) — текущий язык исполняемого модуля;
MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT) — язык по умолчанию для текущего пользователя;
MAKELANGID(LANG_DEFAULT, SUBLANG_DEFAULT) — язык по умолчанию в системе.

Следующий шаг — загрузить найденный ресурс. Для этого служит функция **LoadResource**:

```
HGLOBAL LoadResource(
   HMODULE hModule,
   HRSRC hResInfo
);
```

где **hModule** — идентификатор модуля DLL;
hResInfo — идентификатор ресурса, возвращенный функциями **FindResource** или **FindResourceEx**.

Функция возвращает идентификатор (дескриптор) ресурса в памяти.

Для поиска и загрузки конкретных типов ресурсов используются соответствующие функции (табл. 1).

Кроме того, функция **LoadImage** позволяет загружать картинки, иконки и метафайлы.

Таблица 1

Функции для загрузки ресурсов

Тип ресурса	Функция для его загрузки
Картинка (**BITMAP**)	**LoadBitmap**
Иконка (**ICON**)	**LoadIcon**
Курсор (**CURSOR**)	**LoadCursor**
Меню (**MENU**)	**LoadMenu**
Строка (**STRING**)	**LoadString**
Горячие клавиши (**ACCELERATORS**)	**LoadAccelerators**

Подробные описания этих функций находятся в **MSDN**.

Для непосредственного доступа к двоичным данным ресурса используется функция **LockResource**:

```
LPVOID LockResource(
   HGLOBAL hResData
);
```

Эта функция фиксирует положение ресурса в памяти и возвращает указатель на его данные.

После того как работа с ресурсом завершена, его следует выгрузить. Для этого используется функция **FreeResource**:

```
BOOL FreeResource(
   HGLOBAL hgblResource
);
```

Однако эта функция сохраняется для совместимости с ранними версиями **Windows**, и **Microsoft** рекомендует использовать вместо нее следующие функции (табл. 2).

Таблица 2

Функции для выгрузки ресурсов

Тип ресурса	Функция для его выгрузки
Картинка (**BITMAP**)	**DeleteObject**
Иконка (**ICON**)	**DestroyIcon**
Курсор (**CURSOR**)	**DestroyCursor**
Меню (**MENU**)	**DestroyMenu**
Горячие клавиши (**ACCELERATORS**)	**DestroyAcceleratorTable**

Прототипы и подробные описания этих функций содержатся в **MSDN**.

Базовый адрес загрузки DLL

Базовый адрес загрузки DLL показывает положение в адресном пространстве процесса, по которому загрузчик операционной системы загрузит DLL (спроецирует на адресное пространство вызывающего процесса). Точнее, попытается загрузить. Если по запрошенному адресу имеется свободный регион памяти достаточного размера, библиотека будет загружена по заданному базовому адресу (рис. 3). Если же его нет, то загрузчик переместит библиотеку в свободный регион памяти. При этом придется только гадать, по какому адресу действительно загружена библиотека. К сожалению, невозможно заранее предсказать, что будет делать система на различных машинах.

Что произойдет, если две библиотеки DLL или более имеют одинаковый базовый адрес загрузки? Очевидно, загрузчик не сможет поместить все библиотеки в одну и ту же область памяти. Поэтому загрузчик ОС загрузит по запрошенному адресу только одну библиотеку, а остальные переместит в свободные регионы.

Обратите внимание на значение поля **«Image Base»** на рис. 3 — это и есть базовый адрес загрузки.

Как изменить базовый адрес загрузки DLL

Существует два способа изменения базового адреса загрузки DLL.

Первый — использование утилиты **rebase.exe** из **Platform SDK**. Эта утилита имеет множество различных параметров, но лучше всего вызывать ее, задав ключ командной строки **/b**, базовый адрес загрузки

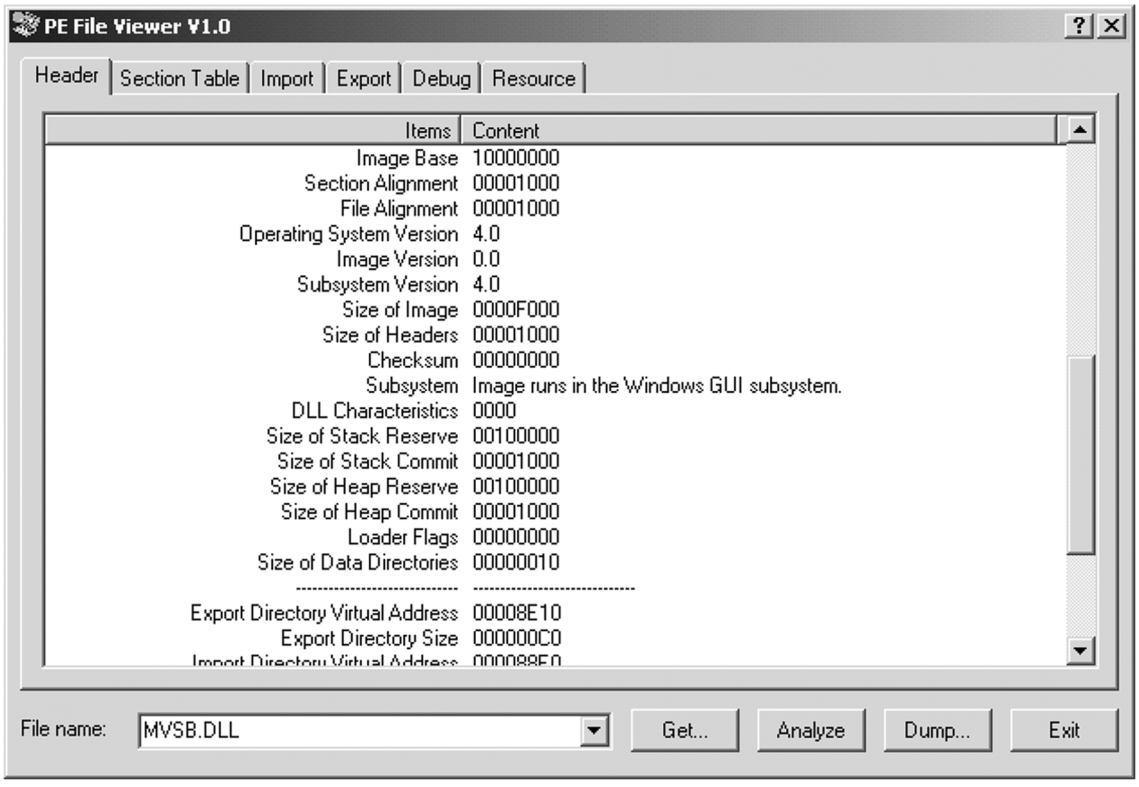

Рис. 3. Пример получения информации о базовом адресе загрузки DLL

и имя соответствующей DLL. Если в командной строке утилиты **rebase.exe** задаются сразу несколько имен DLL, то их базовые адреса будут изменены так, что они будут последовательно загружены вплотную друг к другу начиная с заданного адреса. Пример командной строки:

```
rebase.exe /b 0x12000000 MyDLL.dll
```

Второй способ изменения адреса загрузки DLL — явно задать адрес загрузки при компоновке. В **Visual Basic** этот адрес задается в поле **«DLL Base Address»** на вкладке **«Compile»**, в **Borland C++ Builder** и **Delphi** — в поле **«Image base»** вкладки **«Linker»**, а в **Visual C++** — в поле **«Base Address»** на вкладке **«Link»** или же параметром командной строки компоновщика **link.exe** после ключа **/BASE** (рис. 4).

Зачем нужно изменять базовый адрес загрузки DLL? Для этого есть два достаточно веских основания.

Первое заключается в том, что при известном адресе загрузки DLL облегчается поиск причины ошибки при сбое в приложении. Как правило, в подобном случае система выводит окно с маловразумительным сообщением типа «Инструкция по адресу 0x105C0F23 попыталась обратиться по адресу 0x00000010: память не может быть read».

Если ваше приложение загружает десяток-полтора DLL, то как по адресу ошибки **0x105C0F23** установить, в какой именно библиотеке возник сбой, если все они имеют один и тот же базовый адрес загрузки (**0x10000000**)? Как сказано выше, в такой ситуации загрузчик операционной системы переместит каждую DLL в свободный регион памяти, но как вы сможете установить, какая именно DLL была отображена на адрес, например, **0x105C0000**? Совсем иначе обстоит дело с поиском ошибки, если вы знаете, какая именно DLL должна загружаться

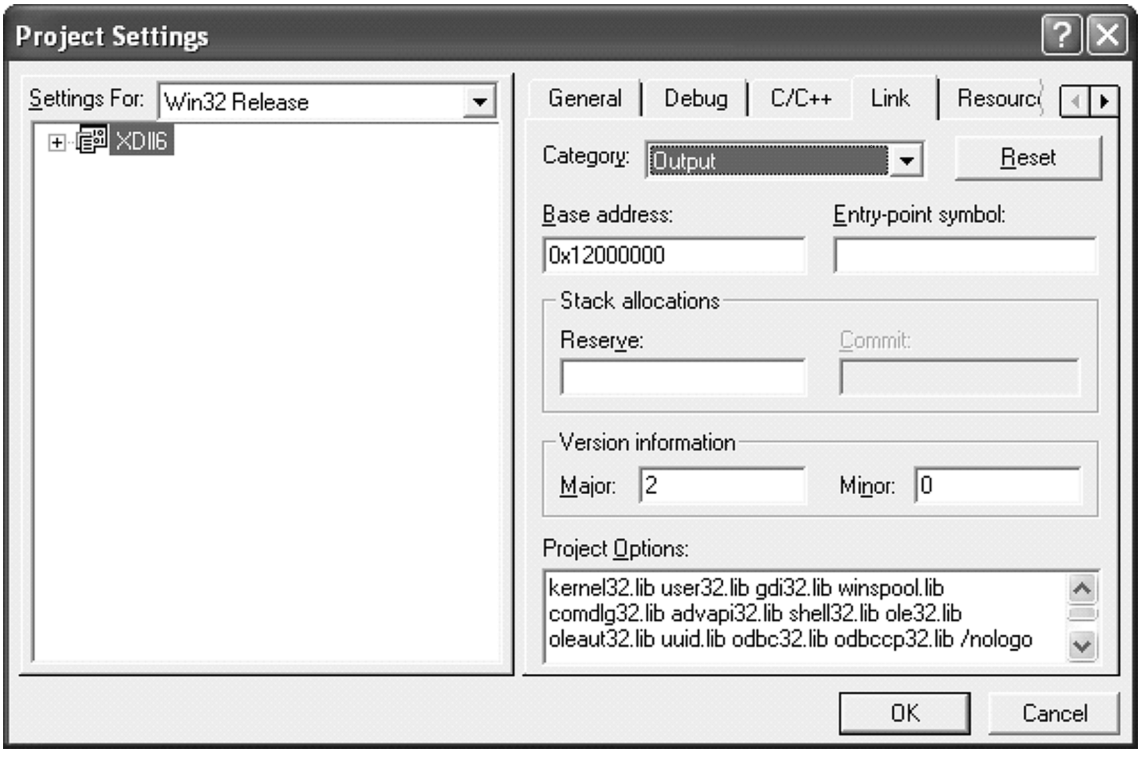

Рис. 4. Настройка базового адреса загрузки DLL в **Visual C++**

по базовому адресу **0x105C0000**, — найти причину сбоя будет намного легче.

Второе основание заключается в том, что перемещение DLL замедляет загрузку и запуск приложения. Во время перемещения загрузчик операционной системы должен считать из соответствующей секции DLL нужную для этого информацию, обойти все участки кода, которые обращаются к адресам внутри DLL, и изменить их, поскольку теперь библиотека находится в другой области памяти. Если в приложении несколько конфликтов адресов загрузки, то запуск приложения может замедлиться более чем вдвое.

Порядок поиска файла DLL при загрузке

Когда загрузчик операционной системы пытается подключить (спроецировать) файл динамически загружаемой библиотеки на адресное пространство процесса, он проводит поиск DLL-файла в каталогах в строго определенной последовательности:

1) каталог, содержащий EXE-файл;
2) текущий каталог процесса;
3) системный каталог **Windows** (например, **«C:\Windows\System32»**);
4) основной каталог **Windows** (например, **«C:\Windows»**);
5) каталоги, указанные в переменной окружения **PATH**.

Если на любом из этих шагов необходимый файл динамической библиотеки найден, загрузчик прекращает дальнейший поиск и подключает найденный файл к адресному пространству процесса. Важно понимать, что порядок поиска не зависит от того, каким образом DLL подключается к адресному пространству процесса — используется неявное связывание или же явный вызов функции **LoadLibrary**. Порядок поиска файла DLL остается всегда одним и тем же.

Когда разрабатывались первые версии **Windows**, оперативная намять и дисковое пространство были крайне дефицитным ресурсом, так что **Windows** была рассчитана на предельно экономное их использование — с максимальным разделением между потребителями. В связи с этим **Microsoft** рекомендовала размещать все модули, используемые многими приложениями (например, библиотеку C/C++ и DLL, относящиеся к MFC), в системном каталоге **Windows**, где их можно было легко найти.

Однако со временем это вылилось в серьезную проблему: программы установки приложений то и дело перезаписывали новые системные файлы старыми или не полностью совместимыми (см. об этом раздел «DLL Hell и конфликты версий»). Из-за этого уже установленные приложения переставали работать.

В связи с этим **Microsoft** сменила свою позицию на прямо противоположную: теперь она настоятельно рекомендует размещать все файлы приложения в своем каталоге и ничего не трогать в системном каталоге **Windows**. Тогда ваше приложение не нарушит работу других программ, и наоборот. А ваши версии необходимых для вашего приложения динамических библиотек заведомо будут найдены и подключены к адресному пространству вашего процесса раньше, чем соответствующие библиотеки из системного или общего каталогов **Windows**.

Как изменить порядок поиска?

Существует два способа изменения порядка поиска файла DLL при загрузке.

Первый из них применяется при явной загрузке DLL, он заключается в вызове функции **LoadLibraryEx** с флагом **LOAD_WITH_ALTERED_SEARCH_PATH**. Этот флаг изменяет алгоритм, используемый функцией **LoadLibraryEx** при поиске DLL-файла. Обычно поиск осуществляется так, как описано выше. Однако если данный флаг установлен, функция ищет файл, просматривая каталоги в следующем порядке:

1) каталог, заданный в параметре **pszDLLPathName** функции **LoadLibraryEx**;
2) текущий каталог процесса;
3) системный каталог **Windows**;

4) основной каталог **Windows**;

5) каталоги, перечисленные в переменной окружения **PATH**.

Второй способ применяется как для неявно загружаемых, так и для явно загружаемых DLL. Он заключается в создании особого строкового параметра в ключе реестра:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\
Control\Session Manager\KnownDLLs
```

В этом ключе реестра описываются так называемые известные DLL (**known DLLs**) (рис. 5). Они ведут себя точно так же, как и любые другие DLL — с той лишь разницей, что система всегда ищет их в одном и том же каталоге. Ключ содержит набор параметров, имена которых совпадают с именами известных DLL. Значения этих параметров представляют собой строки, идентичные именам параметров, но дополненные расширением .dll. Когда вы вызываете **LoadLibrary** или **LoadLibraryEx**, каждая из них сначала проверяет, указано ли имя DLL вместе с расширением .dll. Если нет — поиск DLL ведется по обычным правилам.

Если расширение .dll указано, функция его отбрасывает и ищет в разделе реестра **KnownDLLs** параметр, имя которого совпадает с именем DLL. Если его нет — вновь применяются обычные правила поиска. А если параметр есть, система считывает его значение и пытается загрузить заданную в нем DLL. При этом система ищет DLL в каталоге, на который указывает значение, связанное с параметром реестра **DllDirectory**. По умолчанию в **Windows 2000** и **Windows XP** параметру **DllDirectory** присваивается значение **%SystemRoot%\System32**. Таким образом, для изменения порядка поиска файла DLL (например, **MyOwnDLL.dll**) описанным способом необходимо:

• создать в описанном ключе реестра строковый параметр с именем **MyOwnDLL**;

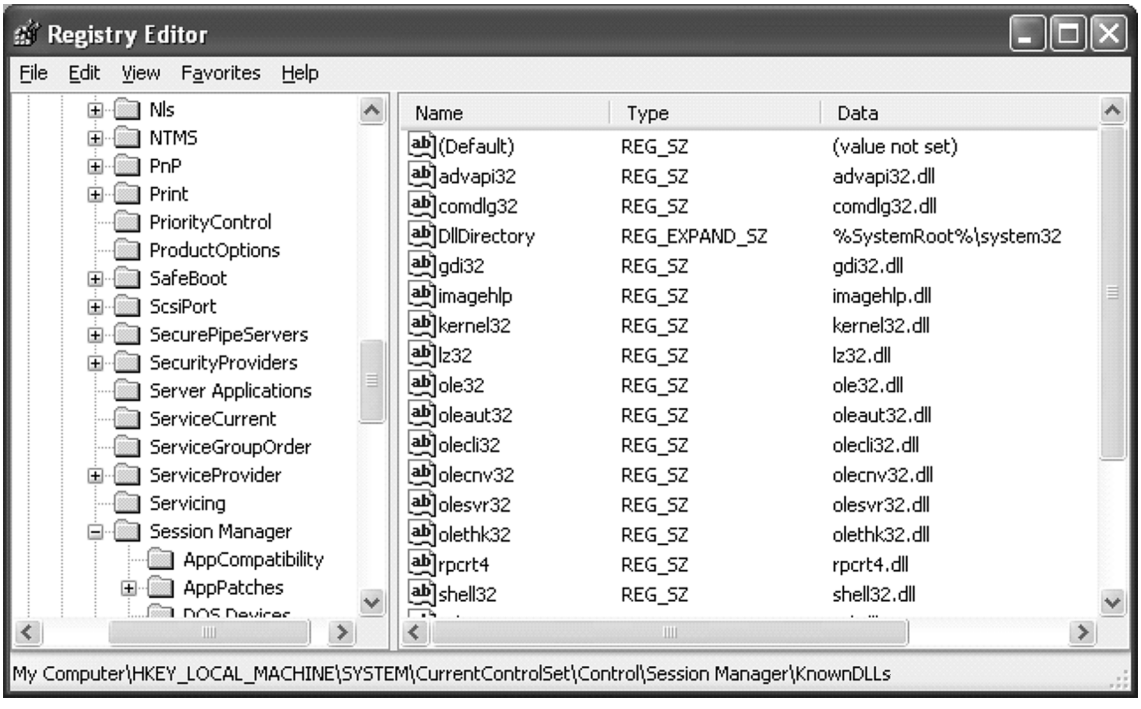

Рис. 5. Пример списка известных DLL

• установить значение этого параметра, например, в **SomeOtherDLL.dll**.

Теперь, если написать код:

```
HMODULE hDll = LoadLibrary("MyOwnDLL.dll");
```

то загрузчик ОС загрузит библиотеку **SomeOtherDLL.dll** из каталога, например, **C:\Windows\System32**.

DLL Hell и конфликты версий

Ничто не постоянно, и то, что сегодня кажется самым современным, уже завтра устаревает и отмирает. Не избегают этой судьбы и динамически загружаемые библиотеки. Однако DLL после смерти не попадают в свой ад (**DLL Hell**). Туда попадает программист, который их использует.

Представьте себе такую ситуацию. Вы разрабатываете библиотеку, реализующую некую полезную функциональность (ПФ). Вашу библиотеку замечают, оценивают, и многие разработчики ПО начинают ею пользоваться. Отпраздновав свой успех, вы начинаете думать: а что бы еще такого хорошего поместить в библиотеку?

Вам приходит блестящая идея, и вы выпускаете версию 2.0 своей библиотеки, которая содержит уже две полезные функциональности: ПФ-1 и ПФ-2. Конечно же вы рассылаете всем своим клиентам обновленную версию, а они, в свою очередь, выкладывают у себя на сайтах соответствующие обновления.

Но представим себя на месте пользователя. Он купил два программных продукта и по очереди устанавливает их на свой компьютер. В первую очередь он ставит программу, которая поставляется с новой библиотекой версии 2.0. Все работает, и все счастливы. А затем пользователь ставит еще одну программу устаревшей версии, которая использует библиотеку версии 1.0. И эта программа перезаписывает новенькую библиотеку 2.0 ее устаревшей версией 1.0. А теперь угадайте: что будет, когда первая программа попытается обратиться к вашей библиотеке за ПФ-2? Результат будет плачевный.

Суть проблемы в том, что в **Windows** отсутствует системное средство контроля версий библиотек. Так что проверить, какая из двух DLL старше, практически не представляется возможным. Конфликты различных версий библиотек DLL и все связанные с этим проблемы получили название **DLL Hell** («Ад DLL»).

Компания **Microsoft** попыталась решить эту проблему, введя ресурс **VERSIONINFO** в структуру DLL. Однако это достаточно слабое решение.

Во-первых, это не решает проблем со старыми библиотеками, созданными до появления указанного ресурса.

Во-вторых, ответственность за проверку версии DLL по-прежнему лежит на разработчике, и, если он забудет в программе проверить версию библиотеки, снова возникнут проблемы.

Наконец, не редким является случай, когда именно более новые версии библиотек являются источниками ошибок. В таком случае попытка подсунуть программе старую и надежную DLL окончится крахом.

Функция DllMain

DLL по аналогии с консольными программами на C++ имеют точку входа **DllMain**. Это функция, вызываемая системой всякий раз, когда происходит одно из четырех событий:

- присоединение DLL к процессу;
- присоединение DLL к потоку;
- отсоединение DLL от потока;
- отсоединение DLL от процесса.

Замечание.

Проводя дальнейшие аналогии, вы можете подумать, что существует аналог **wDllMain**, который будет работать в Unicode-версии библиотеки? Это совершенно неправильная аналогия. Unicode-версии **DllMain** не существует и существовать не должно, потому что функция **DllMain** не требует приема параметров типа TCHAR (и его производных).

Функция **DllMain** обязана называться именно так! Не стоит ошибаться в наименовании данной функции. Несмотря на то что DLL — это аббревиатура, соответствующее название функции должно быть представлено в виде **DllMain** (и никак иначе — с точным соблюдением регистра). Такое название используется по умолчанию внутри реализации функции **_DllMainCRTStartup**, которая, в свою очередь, определяется ключом **/entry** компоновщика. В общем случае можно реализовать собственную версию **_DllMainCRTStartup**, которая будет вызывать по умолчанию функцию с отличным от **DllMain** названием.

Замечание.

Если вы ошибетесь при написании названия **DllMain**, компилятор посчитает, что вы не реализовали данную функцию, и подставит вам в код DLL версию по умолчанию.

Требования, которые необходимо соблюдать для функции **DllMain**:

- использование соглашения вызова **stdcall**: если вы определите другое соглашение вызова (**calling convention**), то получите предупреждение от компилятора, который принудительно назначит требуемое соглашение;
- соответствие прототипа функции **DllEntryPoint**.

Рекомендуется все же оставить назначение точки входа линкеру, чтобы корректно совершить инициализацию CRT и статических объектов C++. По умолчанию линкер использует для DLL точку входа, именуемую **_DllMainCRTStartup**.

Благодаря посылке уведомлений DLL можно удостовериться, что происходит с вашей DLL в тот или иной момент времени (и происходит ли что-нибудь вообще). Кроме этого, подобные уведомления могут понадобиться для различного рода инициализирующих действий — например, для создания дополнительных потоков, запуска таймера и пр.

В **DllMain** не стоит выполнять различного рода «сложных» инициализирующих действий. На момент отображения библиотека, к которой происходит обращение, может быть еще не инициализирована (ведь DLL инициализируются в определенном порядке). И вызов функции приведет к краху приложения! Не стоит также использовать вызов **LoadLibrary** внутри **DllMain**, так как это может привести к неразрешимому циклу при инициализации DLL.

Те же рекомендации относятся и к вызову **FreeLibrary** при отсоединении DLL от адресного пространства вызывающего процесса, так как это может привести к ситуации, когда код динамически загружаемой библиотеки используется после исполнения кода ее завершения.

Примеры действий, которые безопасно совершать в функции **DllMain**:

- управление TLS (**thread local storage** — локальное хранилище потока);
- создание объектов ядра;
- доступ к файловым дескрипторам.

Библиотечные функции, обращения к которым не стоит совершать в функции **DllMain**:

- функции **User32.dll**;
- **Shell**-функции;
- функции COM;
- RPC-функции;
- функции **Windows Sockets**;
- функции, в которх происходит вызов библиотек, описанных выше.

Несмотря на то что в **DllMain** не запрещается создавать и инициализировать объекты синхронизации, сам процесс синхронизации не стоит совершать в функции **DllMain**, потому что вызовы **DllMain** синхронизируются системой в единую последовательность (пока не завершится один вызов **DllMain**, не будет начат другой). Ожидание установки объекта в сигнальное состояние может привести к состоянию **deadlock** (состояние взаимной блокировки потоков).

Если ваша DLL требует каких-либо сложных видов инициализации, предоставьте для этих целей отдельную функцию (наподобие **WSAStartup** в случае библиотеки **Windows Sockets**). Точно таким же образом можно завести и соответствующую функцию для освобождения ресурсов (**WSARelease**).

В общем случае вы не обязаны реализовывать данную функцию (в отличие от функции **main**), если на то нет особой необходимости. Как было упомянуто ранее, компилятор сам позаботится о том, чтобы предоставить вашей библиотеке реализацию функции **DllMain** по умолчанию (см. раздел «Предоставляемая компилятором версия DllMain по умолчанию»).

А теперь более подробно поговорим об уведомлениях, приходящих в **DllMain** от системы.

Как система работает с DLL

Система уведомляет DLL об определенных событиях, происходящих с библиотекой. Существует четыре стандартных сообщения, уведомления о которых приходят в функцию **DllMain**.

Уведомление DLL_PROCESS_ATTACH

Данное уведомление присылается всякий раз, когда система присоединяет указанную DLL к адресному пространству вызывающего процесса. Это происходит либо неявным (при старте приложения или при вызове функции из DLL отложенной загрузки) или явным (посредством вызова функции **LoadLibrary**) образом.

Как вы уже знаете, явление присоединения DLL к процессу называется «проецирование DLL на адресное пространство вызывающего процесса». Дело в том, что одно из преимуществ DLL, благодаря которому они до сих пор не сдают свои устойчивые позиции, — это экономия оперативной памяти посредством разделения кода между несколькими вызывающими процессами. Таким образом, получается, что при одновременном существовании нескольких экземпляров приложений, которые пользуются услугами одной и той же DLL, в ОП находится лишь один экземпляр данной DLL. Осуществляется это благодаря механизму виртуальных адресов, отображению файлов в памяти и счетчику ссылок на DLL.

Механизм виртуальных адресов позволяет приложению перенастроить адреса функций, вызываемых из DLL, таким образом, что все они будут ссылаться на одни и те же участки физической ОП.

Счетчики ссылок необходимы для того, чтобы правильно определить моменты действительной загрузки и выгрузки DLL. Первоначально система, получив запрос на загрузку DLL в ОП, проверяет, нет ли такой DLL среди уже загруженных в ОП. Если такой DLL нет, то она действительно загружается в ОП, а счетчик ссылок (т. е. количество клиентов, которые используют данную DLL) становится равным единице.

Если же DLL уже находится в ОП, то DLL с диска повторно не загружается. Происходит настройка виртуальных адресов в приложении таким образом, чтобы они ссылались на нужные участки физической ОП, в которых находится исполняемый код DLL. Счетчик ссылок клиентов увеличивается на единицу (рис. 6).

В связи с тем что в общем случае загрузка DLL в ОП может занять много времени (не говоря уже о том, что держать в памяти одинаковые страницы с кодом — непозволительная роскошь), подобный механизм может сослужить (и делает это уже давно) хорошую службу. Кроме того, рассмотренные в предыдущих разделах способы оптимизации позволяют определенным образом снизить затраты на инициализацию (всех интересующихся отсылаем к статье MSDN «Optimizing DLL Load Time Performance»).

В любом случае (есть DLL в ОП или нет) функции **DllMain** посылается уведомление **DLL_PROCESS_ATTACH** о присоединении данной DLL к адресному пространству вызывающего процесса. Конечно, если один и тот же процесс (конкретный экземпляр приложения) вызовет несколько раз функцию **LoadLibrary**, то дополнительных вызовов

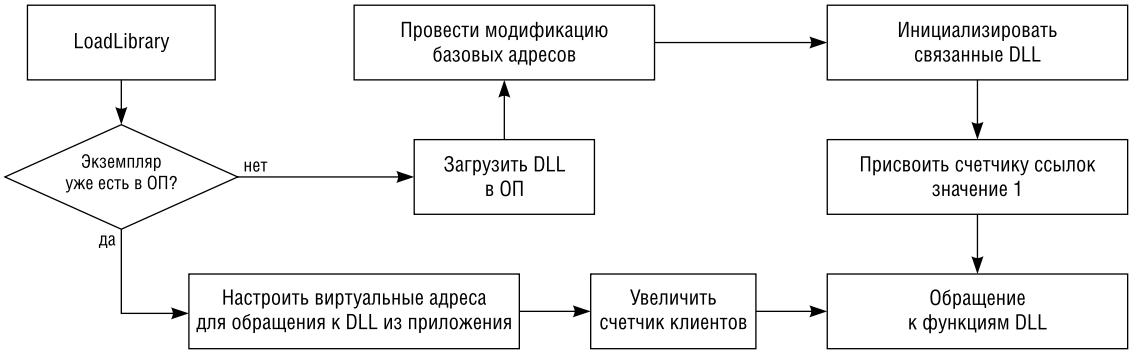

Рис. 6. Процесс загрузки DLL

DllMain с параметром **DLL_PROCESS_ATTACH** не произойдет, как не будет происходить и дополнительного процесса проецирования.

Функция **DllMain** сообщает об успешности выполнения инициализирующих действий путем возврата значения **TRUE**. Если DLL по каким-то причинам не может провести успешную инициализацию, то функция **DllMain** должна вернуть значение **FALSE**. В этом случае приложение в ответ на вызов **LoadLibrary** получит значение **HINSTANCE**, равное **NULL**. Использование неявной или отложенной загрузки приведет к появлению на экране соответствующего сообщения об ошибке, после чего приложение будет немедленно завершено.

Замечания.

1. Несмотря на то что механизм DLL обеспечивает разделение кода между различными процессами, участки данных (переменные, константы, статические переменные) НЕ разделяются между приложениями (т. е. страницы, содержащие данные, имеют атрибут «копирование при записи», вследствие чего любая запись приводит к немедленному копированию этой страницы в ОП). Если вы, конечно, об этом явно не попросите компилятор: для этого необходимо поместить ту или иную переменную в разделяемый сегмент памяти, который доступен нескольким экземплярам EXE или DLL, настроив соответствующие атрибуты. **Visual C++** позволяет это сделать при помощи директивы **#pragma section**.

2. С константами не всегда дело обстоит именно так. Компилятор помещает константы либо в секцию **.rdata** (так любит делать VC++), которая разделяется всеми экземплярами загруженной библиотеки — точно так же, как секция **.code**, либо в секцию **.code** (а так любит делать компилятор Паскаля/Дельфи).

3. Случай ошибочной инициализации DLL отложенной загрузки может быть обработан посредством механизма исключений.

Уведомление DLL_PROCESS_DETACH

Данное уведомление сообщает DLL, что она готова к отсоединению от адресного пространства вызывающего процесса. В этом случае производятся, как правило, различного рода действия по освобождению ресурсов (если они требуются) — уничтожение потоков, остановка таймеров, освобождение памяти и пр.

Как вы уже знаете, механизм подсчета ссылок отрабатывает, и здесь — в случае отсоединения DLL от клиента (в нашем случае — вызывающего процесса) — счетчик ссылок уменьшается на единицу. По достижении им нулевого значения DLL выгружается из оперативной памяти (рис. 7).

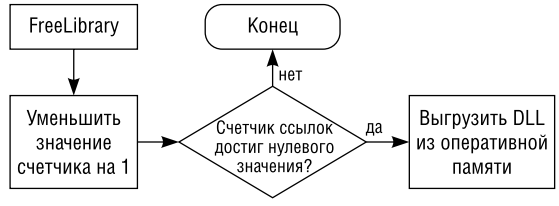

Рис. 7. Процесс выгрузки DLL

Прежде чем перейти дальше, необходимо четко уяснить: уведомления **DLL_PROCESS_ ATTACH** и **DLL_PROCESS_ DETACH** приходят в **DllMain** всегда, независимо от того, выгружается она физически из ОП или нет. Правда, существуют случаи, когда уведомление **DLL_PROCESS_DETACH** может и не прийти, — это происходит при использовании функций семейства **TerminateProcess**. В этом случае DLL не получит уведомления об отсоединении от адресного пространства вызывающего процесса, а следовательно, не сможет произвести корректную очистку всех используемых ресурсов. В дальнейшем это может привести к непредсказуемым результатам работы не только конкретного приложения, но и всей системы в целом. Это еще один довод в пользу того, чтобы НИКОГДА не использовать **TerminateProcess** для завершения приложения.

Уведомление DLL_THREAD_ATTACH

Данное уведомление присылается системой в том случае, если DLL находится в области видимости отдельного потока. Можете понимать это сообщение аналогично тому, что и **DLL_PROCESS_ATTACH**. Только **DLL_PROCESS_ATTACH** присылается для первичного (**main**) потока приложения, а **DLL_THREAD_ATTACH** — для всех остальных. При этом если поток появился до того, как DLL были спроецированы, то для них такие сообщения не присылаются.

Когда может потребоваться обработка подобного уведомления? Например, в том случае, если DLL необходима инициализация, связанная с появлением конкретного потока.

Уведомление DLL_THREAD_DETACH

Это уведомление сообщает DLL о выходе потока из области видимости. Сообщение может и не прийти, если для завершения потока используется функция **TerminateThread**. Как и раньше, аналогия с первичным потоком сохраняется — для него присылается уведомление **DLL_PROCESS_DETACH**, для всех остальных потоков присылается уведомление **DLL_THREAD_DETACH**.

Обработка этого уведомления требуется также только в том случае, если необходима конкретная реакция DLL на завершение работы какого-либо из рабочих потоков.

Обобщая сказанное выше, приведем простой пример, который поможет понять, когда какое уведомление будет приходить, а когда — нет. Допустим, у нас есть DLL отложенной загрузки, из которой экспортируется полюбившаяся вам функция **getSum**:

```
int main()
{
    ...
    // (1)
    ...
    const int res = getSum(10, 20);
    ...
    // (2)
    ...
    const BOOL b = __FUnloadDelayLoadedDLL2
    ("XDll6.dll");
    ...
    // (3)
    ...
}
```

Так как мы работаем с DLL отложенной загрузки, инициализация DLL будет происходить лишь в момент вызова функции **getSum**. Именно тогда DLL будет спроецирована на адресное пространство нашего процесса. В это время DLL получит уведомление **DLL_ PROCESS_ATTACH**. Если **DllMain** вернет значение **TRUE**, свидетельствующее об успешности выполнения инициализирующих действий, то работа приложения продолжится — будет вызвана функция **getSum** с параметрами 10 и 20.

Если бы мы не использовали дополнительных средств предварительной выгрузки DLL, то только при завершении нашего приложения DLL получила бы уведомление **DLL_PROCESS_DETACH**. Если же подобные действия в приложении осуществлены (в нашем случае — при помощи вызова **__FUnloadDelayLoadedDLL2**), то выгрузка DLL будет произведена в момент вызова

этой функции, а следовательно, DLL сразу же получит уведомление **DLL_PROCESS_DETACH**.

В любом случае функция **DllMain** проведет необходимые действия по очистке используемых ресурсов, после чего DLL будет отсоединена от адресного пространства.

Немного усложним наш пример. Предположим, мы работаем с дополнительными (дочерними) потоками, используя функции семейства **CreateThread**. Если дополнительный поток создается в разделе (1) (согласно приведенному выше листингу), то уведомление **DLL_THREAD_ATTACH** библиотека не получит (так как DLL отложенной загрузки еще не спроецирована на адресное пространство). Если же инициализация потока происходит в разделе (2), то в этом случае DLL отложенной загрузки уже спроецирована на адресное пространство, а следовательно, для нее поступит уведомление **DLL_THREAD_ATTACH**.

Такая же ситуация наблюдается и при завершении потока. Если поток завершается в момент времени (2), то DLL обязательно получит уведомление **DLL_THREAD_DETACH**, что поможет ей совершить необходимые действия (если они требуются), связанные с завершением конкретного потока. Если же поток завершается в момент времени (3), то уведомление **DLL_THREAD_DETACH** не будет отослано нашей DLL — его просто некому отправлять, так как DLL уже отсоединена от адресного пространства нашего процесса.

Упорядочивание вызовов функции DllMain в приложении

Обратите внимание, что система упорядочивает вызовы функции **DllMain**. Это значит, что в многопоточном приложении при подключении к адресному пространству процесса DLL (не важно, выполняется такое подключение с помощью явного или неявного связывания) система упорядочивает вызовы функции **DllMain** со значениями аргумента **DLL_PROCESS_ATTACH** и **DLL_THREAD_ATTACH** таким образом, что выполнение одним из потоков функции **DllMain** вызовет блокировку других потоков до тех пор, пока выполняющий эту функцию поток не выйдет из функции **DllMain**. Затем выполнение функции **DllMain** будет предоставлено другому потоку, и так далее. Аналогичная картина наблюдается при вызовах функции **DllMain** со значениями аргумента **fdwReason**, равными **DLL_THREAD_DETACH** и **DLL_PROCESS_DETACH**. Помните: в многопоточном приложении функцию **DllMain** потоки исполняют по очереди и никогда — несколько потоков одновременно! Это может стать причиной зависания вашего приложения, если в функции **DllMain** вы используете ожидающие функции семейства **WaitFor...** с объектами синхронизации (мьютексами, семафорами и пр.). Поскольку в тот момент, когда один из потоков вашего приложения исполняет функцию **DllMain** и ожидает наступления некоего события в функции **WaitFor...**, другие потоки «заморожены» в ожидании завершения исполнения этим потоком кода **DllMain**. Ожидаемое событие никогда не наступит, и ваш процесс оказывается «замороженным навсегда». Происходит так называемая взаимная блокировка потоков (**deadlock**). Помните об этом, если решите в функции **DllMain** использовать одну из ожидающих функций **WaitFor...**!

Как приложение выгружает DLL?

При выгрузке DLL приложение вызывает (неявно) функцию **DllMain** с передачей в параметре **fdwReason** значения **DLL_PROCESS_DETACH**. Обрабатывая это сообщение, библиотека должна провести всю завершающую очистку и освободить все захваченные ресурсы (если они еще не освобождены), поскольку после возврата из функции **DllMain** система немедленно отключит DLL от адресного пространства процесса и выгрузит код и данные библиотеки из памяти. Попытки последующего доступа к коду и данным библиотеки будут возбуждать исключения.

При неявном связывании (**implicit linking**), если DLL выгружается при завершении процесса, функцию **DllMain** исполняет поток,

вызвавший функцию **ExitProcess**, — обычно это первичный поток приложения. Если же DLL выгружается в результате явного вызова функции **FreeLibrary**, то код функции **DllMain** исполняет поток, вызвавший функцию **FreeLibrary**. Возврат из функции **FreeLibrary** не происходит до завершения функции **DllMain**.

Примеры написания функции DllMain

Ну, что ж, как водится, после изучения теории наступает время практики. Попрактикуемся немного и мы.

Для начала рассмотрим прототип функции **DllMain**:

```
BOOL WINAPI DllMain
    (
    // базовый адрес (handle) DLL; отметим, что
    // значение HINSTANCE может использоваться
    // в качестве HMODULE
    HANDLE hModule,
    // код одного из четырех рассмотренных
    // выше уведомления
    DWORD fdwReason,
    // параметр, позволяющий определить различные
    // типы инициализации и выгрузки DLL
    LPVOID lpvReserved
    );
```

А теперь приведем пример написания своей функции **DllMain**.

VC++ 6.0/7.0

```
// DllMain
BOOL APIENTRY DllMain(HANDLE hModule, DWORD
fdwReason, LPVOID lpReserved)
{
   switch (fdwReason)
   {
      case DLL_PROCESS_ATTACH:
           // действия, связанные с присоединением
           // к адресному пространству вызывающего
           // процесса
           break;
      case DLL_THREAD_ATTACH:
           // действия, связанные с появлением
           // в области видимости еще одного
           // дочернего потока
           break;
      case DLL_THREAD_DETACH:
           // действия, связанные с выходом
           // дочернего потока из области
           // видимости DLL
           break;
      case DLL_PROCESS_DETACH:
           // действия, связанные с отсоединением
           // DLL от адресного пространства
           // вызывающего процесса
           break;
   }
   return TRUE; // сообщим, что все прошло успешно
}
```

Как видите, ничего сложного нет. При создании проекта DLL помощник автоматически подготавливает вам подобную заготовку. Если какие-либо уведомления обрабатывать не имеет смысла, их можно исключить. А все остальное определяется конкретно поставленной задачей.

C++Builder

Функция **DllMain** определяется точно таким же образом, как описано выше. Не забывайте, правда, что среда **C++ Builder** позволяет определять точку входа в DLL одним из двух способов — посредством либо функции **DllMain** (так называемой **«Visual C++ style»**) либо функции **DllEntryPoint**.

Ниже приведен аналогичный пример с функцией **DllEntryPoint**.

```
// DllEntryPoint
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned
long reason, void* lpReserved)
{
   switch (reason)
   {
      ...
   }
   return TRUE; // сообщим, что все прошло успешно
}
```

Delphi 6.0

Создадим для этого собственную функцию **DllMain**. Кстати, в **Delphi** функция, получающая уведомления от системы, не обязана называться именно так — конкретное

имя определяет программист. К сожалению, по умолчанию каркас приложения не содержит ничего подобного, так что вам придется немного поработать руками.

```
uses
   Windows;
...
//работает с исправленным System.pas и соответственно
//перекомпилированным впоследствии System.dcu
procedure DllMain(Reason: Integer);
begin
   case Reason of
      DLL_PROCESS_ATTACH:
      //действия, связанные с присоединением
      //к адресному пространству вызывающего
      //процесса
      DLL_PROCESS_DETACH:
      //действия, связанные с отсоединением DLL
      //от адресного пространства вызывающего
      //процесса
      DLL_THREAD_ATTACH:
      //действия, связанные с появлением в области
      //видимости еще одного дочернего потока
      DLL_THREAD_DETACH:
      //действия, связанные с выходом дочернего
      //потока из области видимости DLL
   end;
end;
```

Теперь необходимо зарегистрировать нашу функцию, чтобы RTL **Delphi** знала, какую именно функцию необходимо вызывать. В случае **Visual C++** это решается четкой установкой правила, что функция обязана иметь название **DllMain**. Здесь же такого правила нет.

Что ж, займемся регистрацией.

```
begin
   ...
   //должны явно вызвать!
   DllMain(DLL_PROCESS_ATTACH);
   //работает... если подправить System.pas
   //и исправить там одну маленькую ошибку
   DLLProc := DllMain;
   ...
end.
```

Глобальной переменной **DLLProc** (типа **TDLLProc**) необходимо присвоить адрес вызываемой процедуры (в нашем случае — **DllMain**). После этого **DllMain** будет вызываться в случае необходимости посылки сообщения.

Следует сказать о двух особенностях исполнения данного кода.

Так как секция **begin... end** по своей сути является уведомлением о присоединении DLL к адресному пространству вызывающего процесса, второй раз уведомление **DLL_PROCESS_ATTACH** посылаться не будет, следовательно, мы обязаны сделать это самостоятельно.

Кроме того, в процессе исследования написанного кода обнаружилась удивительная особенность: код **DllMain** не вызывался (если не учитывать факт явного вызова данной процедуры с параметром **DLL_PROCESS_ATTACH**). Как оказалось (спасибо за эту информацию Форуму на **http://delphi.mastak.ru**), **Delphi6 Enterprise** содержит ошибку в модуле **System.pas**.

Замечание.

По заверениям коллег по цеху, **Delphi 5** этой ошибки не содержит. Так что все сказанное ниже справедливо для версии 6.0 (без установленного **Update Pack**).

Рассмотрим этот момент немного подробнее — думаем, подобные приемы исследований не раз пригодятся вам.

Для начала приведем фрагмент кода процедуры **_StartLib** модуля **System.pas**. Именно эта процедура отвечает за своевременный вызов зарегистрированной функции.

```
procedure _StartLib;
asm
   { -> EAX InitTable     }
   {      EDX Module      }
   {      ECX InitTLS }
   {      [ESP+4] DllProc }
   {      [EBP+8] HInst   }
   {      [EBP+12] Reason }
...
   { Call any DllProc     }
   // сохраняем значение ECX
```

```
   PUSH    ECX
   // загружаем значение из стека [ESP+4] -
   // оно там действительно находилось до тех пор,
   // пока не было оператора PUSH ECX; а теперь
   // там лежит код возврата из _StartLib
   MOV     ECX,[ESP+4]
   // значение равно nil?
   TEST    ECX,ECX
   JE      @@noDllProc
   // нет, значит, вызвать DllProc с двумя
   // параметрами
   MOV     EAX,[EBP+12]
   MOV     EDX,[EBP+16]
   // вызов
   CALL    ECX
   @@noDllProc: // нет вызова DllProc
   POP     ECX
...
```

В описании процедуры сказано, что параметр **DllProc** содержится по адресу **[ESP+4]**. Это действительно так, но... до вызова **PUSH ECX**. Таким образом, фрагмент кода должен выглядеть примерно так:

```
...
   { Call any DllProc }
   // сохраняем значение ECX
   PUSH    ECX
   // загружаем значение из стека [ESP+8] (!!!) -
   // там лежит значение DllProc
   MOV     ECX,[ESP+8]
   // значение равно nil?
   TEST    ECX,ECX
   JE      @@noDllProc
   // нет, значит, вызвать DllProc с двумя
   // параметрами
   MOV     EAX,[EBP+12]
   MOV     EDX,[EBP+16]
   // вызов
   CALL    ECX
   @@noDllProc: // нет вызова DllProc
   POP     ECX
...
```

Почему этот код замечательно работает, несмотря на явную ошибку? (Имеется в виду, что при этом не рушится вся система, а единственным недостатком является игнорирование зарегистрированной функции.)

Для начала представим себе весь механизм вызова: функция **_StartLib** вызывается из **_InitLib**.

```
_InitLib:
   <действия 1>
   call _StartLib
   <действия 2>
   ret // from _InitLib
_StartLib:
   <действия 3>
   <фрагмент>
   <действия 4>
   ret // from _StartLib
```

Как известно, стек растет в сторону меньших адресов. Предположим, что мы вызвали процедуру с двумя параметрами типа **integer**. После выполнения инструкции **call _StartLib** стек будет иметь примерно следующий вид (рис. 8).

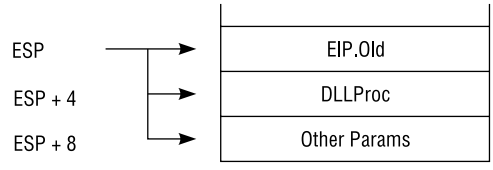

Рис. 8. Стек вызовов после **call _StartLib**

Замечание.

Расположение параметров **Value1** и **Value2** зависит от используемой модели вызова — **calling conventions**.

Для нашего случая **EIP** будет указывать на адрес возврата из процедуры **_StartLib**, а [ESP+4] — на адрес переменной **DllProc**. Если бы забыть про использование **PUSH ECX**, то все замечательно работало бы. Но значение регистра **ECX** сохранено в стеке, поэтому картина немного изменится (рис. 9).

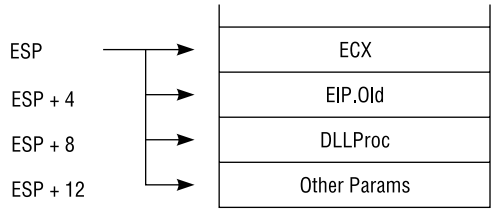

Рис. 9. Стек вызовов после **PUSH ECX**

Теперь **ESP** будет указывать на сохраненное значение регистра **ECX**, адрес возврата будет доступен по **[ESP+4]** и т. д. Что происходит дальше?

```
...
   MOV    ECX,[ESP+4]
  // значение равно nil?
   TEST   ECX,ECX
   JE @@noDllProc
   // нет, значит, вызвать DllProc с двумя
   // параметрами
   MOV    EAX,[EBP+12]
   MOV    EDX,[EBP+16]
   CALL   ECX // вызов
...
```

Мы загружаем (по ошибке) адрес возврата из функции **_StartLib**! Так как он не равен нулю, то мы загружаем в регистры два параметра и возвращаемся из функции (по команде **CALL EIP.Old**).

Таким образом, мы вернемся на фрагмент кода в функции **_InitLib**, который обозначен **<действия 4>**. Эти действия выполняются, после чего происходит выполнение команды **RET**.

Но куда она нас вернет? Опять в функцию **_StartLib** (так как в стеке лежит значение **EIP.Old**) на фрагмент **<действия 4>**! После чего мы покинем эту функцию раз и навсегда. При этом произойдет (и это совсем удивительно) корректная очистка стека!

Проверить вышесказанное можно, если сгенерировать отладочные файлы DCU для модулей **System.pas** и **SysInit.pas**, а затем в режиме отладки пройтись по коду указанных процедур.

Что необходимо сделать для исправления?

1. Переименовываем старый файл **System.dcu** в **System_.dcu** (для истории).

2. Исправляем ошибку в **System.pas** (как показано выше).

3. Идем в директорию **«...\Program Files\Borland\Delphi6\Source\Rtl\»**, в которой находится готовый **makefile** для построения библиотек RTL. Создаем временную поддиректорию в этом каталоге с именем **«Lib»** (или можно подправить переменную **LIB** файла **makefile** для указания директории вывода).

4. Запускаем **make.exe**.

Замечание.

Для создания модулей с отладочной информацией используйте **make.exe — DDEBUG**.

5. Затем копируем полученный файл **System.dcu** из временной директории **Lib** на свое законное место (в **«...\Program Files\Borland\Delphi6\Lib\»**).

6. Проверяем — работает!

Или установите **Update Pack 2**, в котором данная ошибка уже была исправлена!

Замечание.

Не используйте вариант ExitProc для DLL-приложений (он просто не будет работать!) — несмотря на то, что обратное утверждается в документации **Delphi** в разделе **«Library initialization code»**.

Этот способ оставлен для совместимости с предыдущими версиями **Delphi**, поэтому его не стоит использовать и в EXE-приложениях. Вместо этого лучше воспользоваться секциями **initialization/finalization** модуля.

Предоставляемая компилятором версия DllMain по умолчанию

В случае если в вашем коде явно не присутствует какое-либо упоминание **DllMain** (например, вы ошиблись при написании правильного названия: написали что-нибудь вроде **"DLLMain"**; код при этом скомпилируется просто замечательно, но соответствующая функция инициализации вызываться не будет), компилятор вставит в вашу библиотеку некоторый код по умолчанию.

В общем случае компилятор добавляет следующий код (и в этом мы сейчас убедимся):

1) код инициализации CRT;

2) код вызова функции **DllMain** с соответствующими уведомлениями.

Как было сказано ранее, вас не всегда будет интересовать реализация функции **DllMain** в связи с отсутствием в ее необходимости. То же самое произойдет и в том случае, если вы ошибетесь в наименовании данной функции. Что произойдет страшного? Ничего особенного. Просто компилятор сам позаботится о включении кода **DllMain** в DLL.

В общем случае сгенерированный код будет выглядеть примерно так:

```
/* DllMain — функция-заглушка для DLL, скомпонованных с версией 3 C Run-Time Library.
Назначение:
Эта процедура вызывается _DllMainCrtStartup в том случае, когда пользователь не побеспокоился обеспечить собственную реализацию DllMain.
В случае использования LIBC.LIB и MSVCRT.LIB CRTL не требует получения дополнительных уведомлений о присоединении к потоку и отсоединении от него. Поэтому эти уведомления могут быть игнорированы в общем случае (см. соответствующий вызов DisableThreadLibraryCalls). */

BOOL WINAPI DllMain
    (
        HANDLE hDllHandle,
        DWORD dwReason,
        LPVOID lpreserved
    )
{
#if !defined (_MT) || defined (CRTDLL)
    if ( dwReason == DLL_PROCESS_ATTACH && !
    _pRawDllMain )
        DisableThreadLibraryCalls(hDllHandle);
#endif /* !defined (_MT) || defined (CRTDLL) */
    return TRUE;
}
```

Как правило, блоки инициализации/деинициализации и проверки вынесены в разделы **#ifdef/#define/#else/#endif** в целях оптимизации.

При этом вызов **DllMain** произойдет из функции **_DllMainCRTStartup** примерно таким образом (см. **dllcrt0.c**) (см. пример 1).

Изучение подобного кода очень полезно в познавательных целях.

Во-первых, как следует из приведенных комментариев, использование **_DllMainCRTStartup** является предпочтительным (но необязательным) условием определения точки входа — это определяется ключом **/entry** компоновщика (**Project->Settings->Link->Project Options** в случае VC++ 6.0 и **Project->Options->Linker->Advanced->Entry Point** в случае VC++ 7.0).

Во-вторых, если вдруг понадобится реализовать собственную версию **_DllMainCRTStartup**, вы всегда сможете взять в качестве исходного образца приведенный код. Что поможет, в свою очередь, не забыть вызвать **_CRT_INIT**.

Альтернативная реализация может понадобиться, например, в том случае, если надоест поддержка RTL. Это позволит сэкономить пару десятков килобайт сгенерированного размера DLL. Но тогда придется забыть об использовании функций из стандартного набора CRTL (таких как **printf**) и быть очень внимательным при использовании статических и глобальных объектов — об автоматическом вызове конструкторов и деструкторов в этом случае стоит забыть! Вот здесь следует сделать небольшое уточнение.

Зачем нам нужна инициализация CRT? Не проще ли отказаться он нее совсем? Рассмотрим следующий вариант кода, находящегося в DLL:

```
class X
{
   public:
      X()
      {
           // код конструирования объекта
           ...
      }
      ~X()
      {
           // код уничтожения объекта
           ...
      }
} _x;
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID /*lpReserved*/)
{
   ...
}
...
```

Пример 1

```
...
*********************************************************************************************/
* Замечание.
*     Эта процедура является предпочтительной для определения точки входа в DLL. _CRT_INIT также
*     может быть использована в качестве точки входа. Конечно, вы можете использовать свою точку
*     входа и вызвать _CRT_INIT из нее, но делать это все же не рекомендуется.
*********************************************************************************************/

BOOL WINAPI _DllMainCRTStartup(hDllHandle, DWORD dwReason, LPVOID lpreserved)
{
   BOOL retcode = TRUE;

   /* Прежде чем реагировать на уведомление об отсоединении DLL от адресного пространства, необходимо
убедиться, что до этого пришло уведомление о проецировании данной DLL на адресное пространство вызывающего процесса */

   if ((dwReason == DLL_PROCESS_DETACH) && (__proc_attached == 0))
      // Если ничего подобного не было, то вернем FALSE
      return FALSE;

   if (dwReason == DLL_PROCESS_ATTACH || dwReason == DLL_THREAD_ATTACH)
   {
      if   (_pRawDllMain)
           retcode = (*_pRawDllMain)(hDllHandle, dwReason, lpreserved);

      if   (retcode)
           retcode = _CRT_INIT(hDllHandle, dwReason, lpreserved);

      if   (!retcode)
           return FALSE;
}
retcode = DllMain(hDllHandle, dwReason, lpreserved);

/* Пользовательская DllMain вернула FALSE, следовательно, необходимо очистить структуры CRTL. Сделать
это можно, если вызвать _CRT_INIT еще раз, имитировав посылку уведомления DLL_PROCESS_DETACH. Отметим,
что это приведет также к сбросу флага __proc_attached, так что очистка не вызовет повторной (реальной)
посылки уведомления об отсоединении*/

   if ((dwReason == DLL_PROCESS_ATTACH) && !retcode )
      _CRT_INIT(hDllHandle, DLL_PROCESS_DETACH, lpreserved);

   if ((dwReason == DLL_PROCESS_DETACH) || (dwReason == DLL_THREAD_DETACH) )
   {
      if   (_CRT_INIT(hDllHandle, dwReason, lpreserved) == FALSE )
           retcode = FALSE ;

      if   (retcode && _pRawDllMain )
           retcode = (*_pRawDllMain)(hDllHandle, dwReason, lpreserved);
   }
   return retcode;
}
```

Как вы знаете, создание подобных глобальных и статических объектов должно производиться ДО выполнения стартового кода **DllMain** (точно так же, как в консольных приложениях это происходит до старта функции **main**). Это обеспечивается следующим механизмом.

Несмотря на то что функция **DllMain** — стартовая, точкой входа в DLL является одна из функций библиотеки C++ RTL (в нашем случае **_DllMainCRTStartup**). Именно она отвечает за создание подобных объектов (вызывая с соответствующими параметрами функцию **_CRT_INIT**). Лишь после этого происходит вызов **DllMain**. Убедиться в этом можно, поставив контрольную точку в конструкторе/деструкторе класса и в функции **DllMain**. Это позволит удостовериться и в том, что вызов конструктора происходит из **_CRT_INIT**, а вызов деструктора... также из функции **_CRT_INIT**, но когда она вызывается с параметром **DLL_PROCESS_DETACH**.

Таким образом, в самом общем случае образовалась следующая цепочка вызовов (рис. 10).

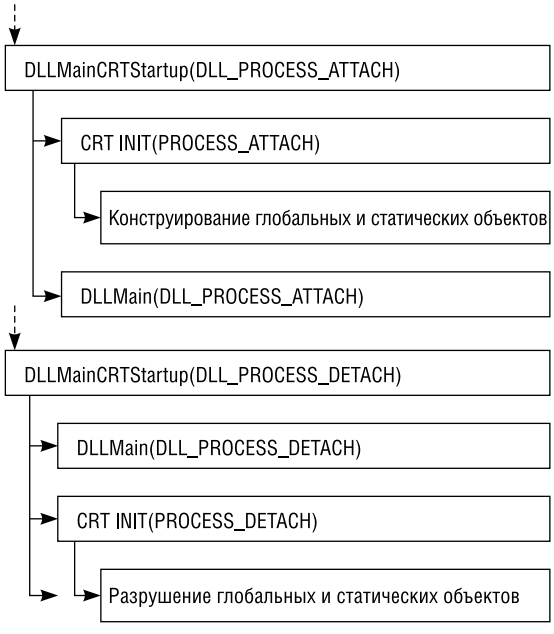

Рис. 10. Последовательность внутренних вызовов **CRTL**

В случае многопоточного приложения среди указанных элементов появятся соответствующие (промежуточные) вызовы функций **_CRT_INIT** и **DllMain** с уведомлениями **DLL_THREAD_ATTACH** и **DLL_THREAD_DETTACH**.

Таким образом, благодаря использованию CRT намного облегчается жизнь конечного программиста. **Visual C++** предоставляет реализацию функций инициализации по умолчанию, от которых пользователь может отказаться, обеспечив собственную реализацию соответствующих функций **_DllMainCRTStartup** и **DllMain**.

Как отладить свою DLL

Отладка DLL проводится почти так же, как и отладка приложений. Разница заключается в том, что система не может непосредственно запустить DLL, — необходимо, чтобы какое-нибудь приложение загрузило DLL и «подтолкнуло» ее. Поэтому при отладке DLL следует указать запускающее приложение.

VC++ 6.0:

- Откройте окно **Project Settings** (Alt-F7) (рис. 11).
- Установите параметр **«Executable for debug session»** вкладки **«Debug»**.

VC++ 7.0:

- Откройте свойства проекта.
- Установите параметр **«Command»** вкладки **«Debugging»**.

Затем можно расставить точки останова и запустить DLL для отладки по команде **Go** (F5). Отладчик среды запустит приложение, дождется загрузки отлаживаемой DLL и остановит выполнение при достижении первой точки останова. После этого можно отлаживать DLL как обычное приложение.

C++ Builder и Delphi:

Для отладки в этих средах необходимо указать имя исполняемого приложения: **Run->Parameters->Host Application** (рис. 12).

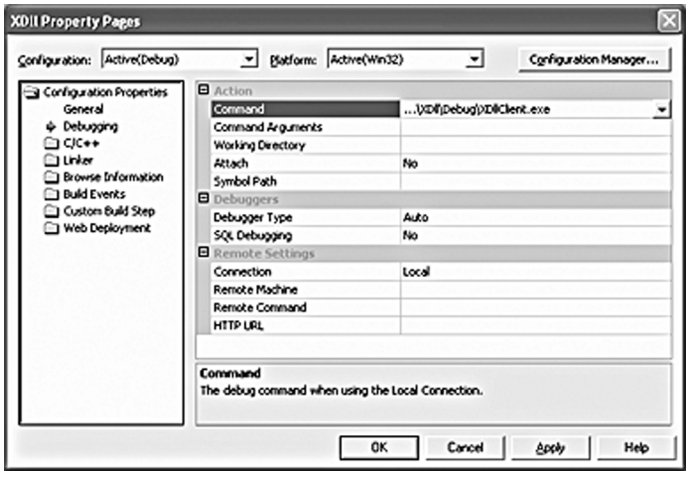

Рис. 11. Отладка DLL (**Visual C++**)

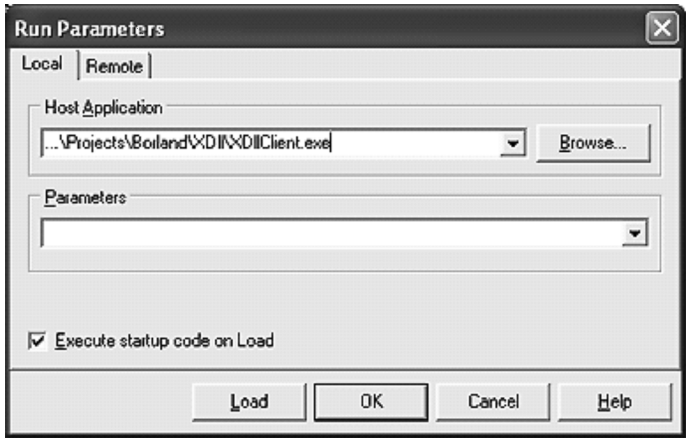

Рис. 12. Отладка DLL (**C++ Builder, Delphi**)

При этом необходимо, чтобы:

а) файл DLL был доступен используемому приложению;

б) исходные файлы проекта также должны быть доступны (а это не всегда выполняется, если вы используете параметр **Project Options->Directories/Conditionals->Output Directory**).

Проще всего добиться одновременного выполнения указанных выше условий: скопировать файл *.exe в папку с проектом DLL.

Экспорт и импорт

Различные способы экспорта

Для того чтобы можно было обратиться из приложения к функциональности некоторой DLL, сама DLL должна каким-либо образом сообщить, что она готова предоставить услуги в виде определенных экспортируемых функций с определенным интерфейсом (список и тип параметров, тип занятия и освобождения стека). А приложение соответственно должно сообщить, что готово воспользоваться именно этими услугами, а не какими-нибудь другими.

Напомним основные определения.

Экспортирование идентификаторов — процесс предоставления базовой функциональности DLL (функций, переменных).

Импортирование идентификаторов — процесс использования функциональности других DLL-приложений. Сразу оговоримся: импортированием занимаются не только exe-приложения. Как правило, написанная DLL сама зависит от других библиотек, а следовательно имеет раздел импорта. Так что не пугайтесь, если вдруг обнаружите у какой-нибудь DLL такой раздел. Ничего удивительного в этом нет.

Итак, DLL (под DLL следует понимать конкретного разработчика, предоставляющего данную DLL, потому что сама по себе библиотека делать, разумеется, ничего не может) должна:

1) определить прототипы экспортируемых функций (например, в виде h-файлов). В случае несоответствия цепочки «язык исполнения равен языку написания» такой способ не всегда подходит. Что же делать? В общем случае импортировать в приложение функцию можно и без h-файла, но прототип данной функции знать все равно надо. Один из способов сделать это — прочитать документацию, поставляемую с DLL, или немного покопаться в исходных кодах. Например, указанная выше система **MatLab** предоставляет множество различного рода математических операций, работа с которыми происходит посредством DLL. Несмотря на то что данные DLL документированы не очень хорошо (сделано это, по всей видимости, специально!), разобраться с принципами их работы достаточно просто — большая часть методов поставляется с исходными кодами: в виде h- и cpp-файлов;

2) определить реализацию экспортируемых функций. В общем случае DLL — средство, которое позволяет предоставить конкретный машинный код без публикации исходных текстов программ (правда, дизассемблеры пока никто не отменял!). Например, это один из принципов, которые исповедует COM: сокрытие деталей реализации;

3) заявить линкеру о своем желании экспортировать все или часть объявленных прототипов. О том, как это сделать, речь пойдет ниже.

Со стороны приложения для использования DLL (и импортирования соответствующих функций) требуется:

1) объявить прототипы использованных функций. Это можно сделать либо с помощью включения h-файла, либо (если такой файл отсутствует) прописав их в явном виде, при этом перед названием функции указать ключевое слово **__declspec(dllimport)** или **extern**.

Замечание.

Особой разницы в использовании способов экспортирования (их мы рассмотрим целых четыре — см. ниже), как правило, нет. Но с импортированием ситуация несколько иная. В документации утверждается, что использование ключевого слова **__declspec(dllimport)** предпочтительнее, так как это позволяет компилятору создавать более эффективный код. Знание о том, что конкретный идентификатор будет экспортироваться из DLL, помогает ему в этом. На основе объявления **extern** сделать подобное предположение (изначально) затруднительно!

2) предоставить *.lib-файл в случае неявной или отложенной загрузки. При использовании явной загрузки такая информация не требуется;

3) предоставить в область видимости приложения DLL-файл. Под областью видимости следует понимать «Алгоритм отыскания DLL».

Теперь поговорим подробнее о способах экспорта.

Желание что-нибудь экспортировать из DLL можно претворить в жизнь несколькими способами:

- использованием **__declspec**;
- использованием DEF-файла;
- использованием **#pragma**;
- использованием специальных настроек проекта.

Замечание.

Кроме того, можно комбинировать различные варианты: например, использовать технику **#pragma** совместно с определением DEF-файла. Если не хотите лишних проблем, делать это рекомендуется ИСКЛЮЧИТЕЛЬНО в исследовательских целях!

Ключевое слово **__declspec** с параметрами **dllimport** и **dllexport** (кроме этих параметров имеются и другие) — это расширение синтаксиса языка C++ от **Microsoft**. Введены они для обеспечения удобного экспорта и импорта функций, данных и объектов из/в DLL.

Замечание.

Для поддержания совместимости среда C++ Builder также поддерживает данные ключевые слова.

Как правило, значения **dllimport** и **dllexport** применяются вместе — в одном h-файле. При этом происходит настройка таким образом, чтобы обеспечить:

а) экспортирование в случае включения h-файла в клиентский проект DLL;

б) импортирование в случае включения h-файла в проект DLL.

При этом применяется конструкция примерно следующего вида:

```
#ifdef    XDLL6_EXPORTS
#define   XDLL_API __declspec(dllexport)
#else
#define   XDLL_API __declspec(dllimport)
#endif    // XDLL_EXPORTS
```

Затем происходит объявление всех необходимых идентификаторов с предварительным модификатором **XDLL_API**. Например, так:

```
//////////////////////////////////////////
XDLL_API int getSum(const int n1, const int n2);
//////////////////////////////////////////
// class CSummator
class XDLL_API CSummator
{
public:
   CSummator(const int n = 0);
   ~CSummator();

   int Add(const int n);

   virtual int GetBalance();
   static int GetDevilSum();
   static int m_DevilSum;
private:
   int m_N;
};
//////////////////////////////////////////
extern XDLL_API int g_N;
```

Так как в проекте DLL обязательно определяется символ **XDLL6_EXPORTS**, все идентификаторы определяются как **__declspec (dllexport)**. В других проектах-клиентах (которые не задумываются — и правильно делают — об этом символе) те же идентификаторы определяются уже как **__declspec (dllimport)**.

Замечание.

Использование конструкции **__declspec (dllexport)** почти всегда требует обязательного применения модификатора **extern "C"**. Поскольку в противном случае компилятор искажает («декорирует») имена экспортируемых функций, и правильный экспорт/импорт функций по имени становится невозможным. Если DLL и исполняемый файл приложения (*.EXE) компилируются разными компиляторами, линкер не сможет собрать исполняемый файл, использующий DLL.

Существуют другие способы экспортирования. Один из них — работа посредством DEF-файла. В этом случае разработчик

DLL предоставляет так называемый файл определений (**define** — определять), в котором последовательно перечисляются различные секции. Нас особенно заинтересует секция **EXPORTS** — в ней перечисляются все экспортируемые идентификаторы. Каждый идентификатор должен располагаться на отдельной строке. При этом число появлений таких секций не ограничено. Можно определить внешнюю функцию с отличным именем от внутреннего, т. е., с точки зрения пользователя, обеспечить некоторого рода «переименование» функции (например, чтобы избавиться от декорирования). Также при определении идентификатора могут использоваться необязательные ключевые слова **PRIVATE** и **DATA**. Параметр **PRIVATE** заставляет не генерировать информацию о данном идентификаторе в lib-файле. Параметр **DATA** применяется для указания того, что идентификатор — это экспортируемая переменная.

Здесь же можно указать порядковый номер, который будет присвоен экспортируемому идентификатору. Это обеспечит совпадение порядковых номеров функций при расширении функциональности DLL (и избавит пользователя от разных неприятных сюрпризов!).

Например:

```
LIBRARY   TESTLIB.DLL
EXPORTS
   MyFunc             @1     ; MyFunc
```

В данном DEF-файле определяются имя DLL (**TESTLIB.Dll**) и имена экспортируемых идентификаторов — в данном примере это **MyFunc**. Значение @1 — это так называемый ординал, или порядковый номер экспортируемой функции. Связывание по ординалу выполняется быстрее, чем по имени, поскольку загрузчику не надо выполнять сравнение символьных литералов. Однако **Microsoft** настоятельно рекомендует использовать во всех вновь разрабатываемых приложениях исключительно связывание по имени. В этом случае если функция **GetProcAddress** не сможет отыскать в таблице имен нужное имя импортируемой функции, то гарантированно вернет **NULL**, сигнализируя о неуспешном поиске. В случае же связывания по ординалу, если функция **GetProcAddress** не сможет отыскать в таблице нужный ординал, то возвращаемое значение не определено (может быть, **NULL**, а может, и нет), это может послужить источником трудно обнаруживаемых ошибок в приложении.

Вернемся к каноническому примеру со сложением. Допустим, мы хотим экспортировать из DLL два идентификатора — **getSum** и **g_N**:

Файл **XDll6.h**:

```
#ifndef __XDLL_H
#define __XDLL_H
//////////////////////////////////////////
int getSum(const int n1, const int n2);
//////////////////////////////////////////
extern int g_N;
#endif // __XDLL_H
```

Файл **XDll6.cpp**:

```
//////////////////////////////////////////
//
int getSum(const int n1, const int n2)
{
   const int n = n1 + n2;
   g_N = n;
   return n;
}
//////////////////////////////////////////
// g_N
int g_N = -1;
```

Воспользуемся для этого DEF-файлом.

VC++ 6.0.

Создадим простой текстовый файл (**File->New->Text File**), озаглавим его **XDll6.def**.

VC++ 7.0.

Выбираем **Project Options->Add New Item->DEF File**.

Напишем там следующие строчки:

Файл **XDll6.def**.

```
EXPORTS
   getSum
   g_N
```

После компиляции проекта воспользуемся вновь услугами **dumpbin**. Информация на экране свидетельствует о том, что мы действительно экспортируем из DLL два идентификатора (причем во вполне удобочитаемом виде!):

```
...
 ordinal hint  RVA        name
    1      0   00076DB8   g_N
    2      1   00001253   getSum
...
```

Замечание.

После создания файла **XDll6.def Visual Studio** автоматически «понимает», что необходимо этот файл использовать как файл определений. Если вы хотите явно указать системе, какой файл нужно использовать в качестве файла определений, необходимо проделать следующие шаги:

Для VC++ 6.0:

Добавить в командную строку линкера (**Link->Project Options**) строку вида **/def:"XDll6.def"**.

Для VC++ 7.0:

Присвоить полю **«Linker->Input->Module Definition File»** необходимое значение — **XDll.def**.

В VC++ 6.0 автоматическое «распознавание» DEF-файла происходит и в случае простого подключения его к файлам проекта: **Project->Add to Project->Files** (**Files Of Type: Definition Files**).

На самом деле существует еще пара способов сказать, что мы хотим что-то экспортировать. Воспользуйтесь ключом **/export** в командной строке линкера или директивой **#pragma**.

VC++ 6.0:

Добавьте в поле **Link->Project Options** параметр **/export** для каждого экспортируемого идентификатора.

Для нашего случая необходимо добавить строку вида **«/export:getSum /export:g_N»**.

VC++ 7.0:

Добавьте в поле **Linker->CommandLine->Additional Options** параметр **/export** для каждого экспортируемого идентификатора.

#pragma вариант выглядит так:

```
...
#pragma comment(linker, "/export:getSum=
?getSum@@YAHHH@Z")
#pragma comment(linker, "/export:g_N=?g_N@@3HA")
////////////////////////////////////////////
int getSum(const int n1, const int n2)
{
...
```

Как видите, этот способ обладает определенным недостатком по сравнению с использованием командной строки линкера — вам придется указать декорированные имена для правильного экспортирования. Все бы ничего, но для начала их придется как-нибудь узнать (например, воспользовавшись **__declspec(dllexport)**-методом). Но тогда зачем нужен этот метод? Впрочем, раз он есть — значит, о нем стоило сказать.

Рассмотрим преимущества и недостатки каждого из четырех разобранных выше методов экспорта.

Вариант 1: **__declspec(dllexport)**

(–) Изобретение **Microsoft**, что приводит к отсутствию стандарта на использование этого ключевого слова. Следовательно, перекомпиляция проекта в другой среде потребует дополнительных модификаций написанного h-файла.

(+) Простота использования как в проекте DLL, так и в клиентских приложениях. С помощью механизма директив условной компиляции (**#ifdef - #define - #else - #endif**) достаточно просто обеспечить определенную гибкость для использования одного h-файла в различных приложениях.

(+) Очевидность случая экспортирования функции по одному взгляду на ее определение в h-файле.

Вариант 2: **DEF-файл**

(–) Наличие дополнительного файла в каждом проекте.

(+) Возможность изоляции определения параметров экспортируемых функций в специализированный файл. При переносе файлов из одного проекта в другой опытный программист не забудет скопировать соответствующий файл определений.

(+) Возможность настройки дополнительных параметров экспортируемых объектов (например, порядковых номеров) — это обеспечит совместимость работы приложений, использующих порядковые номера для экспортирования даже в случае применения новых версий DLL.

Вариант 3: использование **#pragma**

(–) Отсутствие стандарта (**«implementation-specific»**) на использование параметров директивы. Каждая реализация компилятора определяет собственный набор параметров.

(+) Простота использования в других проектах. Текст директивы будет автоматически распознан линкером при компиляции любого другого проекта, содержащего данный файл.

(–) Перемешивание исходного кода с параметрами их определений. Подобные финты не всегда может предсказать человек, который впоследствии будет использовать код такой библиотеки (таким человеком можете оказаться вы сами через несколько лет!).

(–) Не всегда среда правильно отслеживает необходимость полной перекомпиляции (как это обычно происходит при изменении каких-либо настроек проекта). В случае возникновения непонятных ошибок приходится делать **«Rebuild all»**, но об этом иногда забываешь (зато вспоминаешь всех родственников разработчиков от **Microsoft**).

(–) Необходимость определения декорированных имен идентификаторов.

Вариант 4: настройки линкера **/export**

(–) Необходимость дополнительной настройки при использовании исходного кода в других проектах.

(+) Простота использования.

Экспорт и импорт классов и переменных

Основными объектами, с которыми, как правило, происходит работа посредством DLL, являются функции. Кроме того, не запрещается экспортировать классы и переменные.

Замечания.

1. Как известно, «класс» — понятие исключительно программиста и, если хотите, компилятора. Линкер при окончательной компоновке исполняемого модуля уже работает с объектами гораздо более низкой иерархии (адреса памяти, регистры процессора). Поэтому класс растворяется в коде и перестает быть тем классом, к которому мы привыкли.

Таким образом, об экспорте классов из DLL говорить не совсем корректно. Единственное, что можно экспортировать из DLL, — это методы класса. Но как же можно говорить о методах класса, если самого класса не существует? Чтобы устранить данную неоднозначность, компилятор и линкер прибегают к определенным хитростям: при экспорте класса экспортируются соответствующие методы. А при импорте данные методы вновь группируются и становятся (только для программиста! — для удобства) тем классом, к которому мы привыкли. Данную особенность надо учитывать при явной загрузке DLL.

Как было сказано выше, если вы используете неявную или отложенную загрузку, то определенная часть нагрузки (так называемая «бóльшая половина») ложится на сам компилятор. Именно ему предстоит самостоятельно заниматься разрешением имен и определением их местоположения.

Кстати, для различия простых функций и методов класса введено специальное соглашение вызова функций — **__thiscall**. Функция с таким соглашением получает на вход еще один (скрытый) параметр, позволяющий идентифицировать конкретный экземпляр класса, с которым происходит работа.

2. Отложенная загрузка не предполагает наличия в DLL экспортированных переменных!

Итак, вновь модифицируем код нашей DLL: не будем уходить при объяснении материала от уникального алгоритма, который был разработан и внедрен в предыдущих разделах. Теперь добавим к знакомой нам функции **getSum** класс **CSummator**, который будет заниматься аналогичной задачей. Только здесь будет одна особенность: чтобы продемонстрировать возможность экспорта переменной из DLL, будем возвращать накопленный результат в экспортированной (глобальной) переменной.

Замечание.

В реальной жизни делать так строго не рекомендуется. Техника применения ООП позволяет избежать подобного рода глобальных переменных, благодаря этому удается достичь так называемой сильной связности в приложении — когда все связи в приложении четко регламентируются посредством интерфейсов предоставляемых классов.

Именно для демонстрации этого мы решили сделать так, как НЕ НАДО делать на практике!

Кроме того, экспортирование классов из DLL является СЕРЬЕЗНОЙ ОШИБКОЙ проектирования интерфейса DLL. Помните об этом по ходу чтения данного раздела!

Реализация данного класса тривиальна и не должна вызывать у вас проблем с пониманием.

Visual C++

Таким образом, h-файл у нас будет в виде:

Файл **XDll6.h**.

```
#ifndef __XDLL_H
#define __XDLL_H

#ifdef XDLL6_EXPORTS
#define XDLL_API __declspec(dllexport)
#else
#define XDLL_API __declspec(dllimport)
#endif // XDLL_EXPORTS
////////////////////////////////////////////
XDLL_API int getSum(const int n1, const int n2);
////////////////////////////////////////////
// class CSummator
class XDLL_API CSummator
{
public:
   CSummator(const int n = 0);
   int Add(const int n);
private:
   int m_N;
};
////////////////////////////////////////////
extern XDLL_API int g_N;
#endif // __XDLL_H
```

Соответствующий код реализации примет вид (файл **XDll.cpp**):

```
////////////////////////////////////////////
// CSummator
CSummator::CSummator(const int n):
   m_N(n)
{
   g_N = m_N;
}
int CSummator::Add(const int n)
{
   m_N += n;
   g_N = m_N;
   return m_N;
}
////////////////////////////////////////////
//
XDLL_API int getSum(const int n1, const int n2)
{
```

```
   return n1 + n2;
}
//////////////////////////////////////////
// g_N
XDLL_API int g_N = -1;
```

Итак, что же получится в результате компиляции данного проекта? Обратимся вновь к услугам утилиты **dumpbin**. Среди прочей информации будет примерно следующее:

```
...
 ordinal  hint   RVA      name
     1      0    000010F0 ??0CSummator@@QAE@H@Z
     2      1    00001172 ??4CSummator@@QAEAAV0
                          @ABV0@@Z
     3      2    000011D6 ??_FCSummator@@QAEXXZ
     4      3    0000134D ?Add@CSummator@@QAEHH@Z
     5      4    00076DB8 ?g_N@@3HA
     6      5    00001262 ?getSum@@YAHHH@Z
...
```

Попробуем дедекорировать эти загадочные имена посредством **undname.exe -f** (пример 2).

Замечание.

undname.exe для версии VC++ 6.0. (**«...\Microsoft Visual Studio\Common\Tools\»**) требует ключа **-f**, чтобы производить полную дедекорацию имен для всех указываемых идентификаторов (без этого ключа полная информация выводится только для членов класса). Эта же утилита в VC++ 7.0 (**«...\Microsoft Visual Studio .NET\Vc7\bin\»**) работает без ключей — полная информация выводится всегда!

Теперь мы наглядно убедились, что наш класс действительно был разбит при экспорте на функции (представляющие методы этого класса). Компилятор решил сделать немножко больше (впрочем, так он поступает всегда!), поэтому вы можете также обнаружить реализацию конструктора по умолчанию и копирующего конструктора — см. так называемое «правило большой четверки» (стандарт языка C++, раздел 12).

Теперь попробуем воспользоваться предоставленной нам функциональностью в каком-либо приложении. Модифицируем текст файла **main.cpp**:

Файл **main.cpp**:

```
#pragma comment(lib, "xdll6.lib")

#include "../XDll6/XDll.h"
#include <iostream>

int main()
{
   const int res = getSum(10, 20);
   /* используем функцию так, словно мы сами ее
написали */
   std::cout<"getSum(10,20):"<res<std::endl;

   /* а теперь сделаем то же самое посредством
класса */
   CSummator sum(10);
   sum.Add(20);

   /* получим накопленный результат через экспор-
тируемую переменную */
   std::cout < "g_N: " < g_N < std::endl;

   return 0;
}
```

Пример 2 (**undname.exe -f**)

```
...
??0CSummator@@QAE@H@Z == public: __thiscall CSummator::CSummator(int)
??4CSummator@@QAEAAV0@ABV0@@Z == public: class CSummator &
   __thiscall CSummator::operator=(class CSummator const &)
??_FCSummator@@QAEXXZ == public: void
   __thiscall CSummator::`default constructor closure'(void)
?Add@CSummator@@QAEHH@Z == public: int __thiscall CSummator::Add(int)
?g_N@@3HA == int g_N
?getSum@@YAHHH@Z == int __cdecl getSum(int,int)
...
```

Если вы запустите пример на исполнение, получите вполне ожидаемый результат:

```
getSum(10, 20): 30
g_N: 30
```

Как было показано выше, осуществить «неявное» использование класса действительно очень просто. Давайте попробуем сделать то же самое при помощи явной загрузки.

Для облегчения этой задачи избавимся от декорирования имен при компиляции. Один из способов — добавить такие строки в файл **XDll6.cpp** (см. пример 3).

После отмены декорирования **dumpbin** выдаст нам следующую информацию:

```
ordinal hint  RVA      name
    1      0  000010F5 ??0CSummator@@QAE@H@Z
    2      1  00001096 ??1CSummator@@QAE@XZ
    3      2  00001177 ??4CSummator@@QAEAAV0@
                       ABV0@@Z
    4      3  000011DB ??_FCSummator@@QAEXXZ
    5      4  00001352 ?Add@CSummator@@QAEHH@Z
    6      5  00076DB8 ?g_N@@3HA
    7      6  00001267 ?getSum@@YAHHH@Z
    8      7  00001352 CSummatorAdd
    9      8  000010F5 CSummatorConstructor
   10      9  00001096 CSummatorDestructor
   11      A  00076DB8 g_N
   12      B  00001267 getSum
```

Этот способ предполагает, что теперь из DLL экспортируются два идентификатора одной и той же функции (это очевидно, если изучить значения поля RVA). Можно пользоваться любым из них. Кстати, данный подход обладает одним преимуществом — использование такой DLL удобно как в случае неявной загрузки (линкер будет использовать декорированные имена), так и в случае явной (программист, скорее всего, будет использовать относительно «нормальные» имена).

Использование переменной тривиально:

```
int main()
{
    /* явным образом проецируем DLL на адресное
пространство нашего процесса */
    HMODULE hModule = LoadLibrary("xdll6.dll");
    /* проверяем успешность загрузки */
    _ASSERT(hModule != NULL);

    /* определяем при помощи typedef новый тип -
указатель на вызываемую функцию.
    Очень важно знать типы и количество аргументов,
а также тип возвращаемого результата */
    typedef int (*PGetSum)(const int, const int);
    /* пытаемся получить адрес функции getSum */
    PGetSum pGetSum = (PGetSum)GetProcAddress
    (hModule, "getSum");
    /* проверяем успешность получения адреса */
    _ASSERT(pGetSum != NULL);

    /* используем функцию так, словно мы сами ее
написали */
    const int res = pGetSum(10, 20);

    int*pg_N= (int*)GetProcAddress(hModule,"g_N");
    _ASSERT(pg_N != NULL);
    /* использование экспортируемой переменной */
    std::cout < "pg_N: " < *pg_N < std::endl;

    /* выгружаем библиотеку из памяти */
    BOOL b = FreeLibrary(hModule);
    /* проверяем корректность выгрузки */
    _ASSERT(b);

    return 0;
}
```

Пример 3 (файл **XDll6.cpp**)

```
#pragma comment(linker, "/export:CSummatorConstructor=??0CSummator@@QAE@H@Z")
#pragma comment(linker, "/export:CSummatorDestructor=??1CSummator@@QAE@XZ")
#pragma comment(linker, "/export:CSummatorAdd=?Add@CSummator@@QAEHH@Z")
#pragma comment(linker, "/export:g_N=?g_N@@3HA")
#pragma comment(linker, "/export:getSum=?getSum@@YAHHH@Z")
////////////////////////////////////////////////////////////////////////////////
// CSummator
CSummator::CSummator(const int n):
...
```

Как видно из примера, адрес экспортируемой переменной получается при помощи все той же функции **GetProcAddress** (хотя ее название говорит несколько о другом). Благодаря избавлению от декорирования мы можем использовать вполне «человеческое» имя в виде **«g_N»** при обращении к этой переменной.

Delphi 6.0

Теперь посмотрим, что сможет сделать **Delphi** при работе с экспортируемыми переменными. Модифицируем код используемой DLL, чтобы он приобрел следующий вид:

```
library XDll;
{$R *.res}
////////////////////////////////////////////
var
   g_N: integer;
////////////////////////////////////////////
function getSum(const n1: integer; const n2:
integer): integer;
var
   res: integer;
begin
   res := n1 + n2;
   // сохраняем результат в экспортируемой
   // переменной
   g_N := res;
   Result := res;
end;
////////////////////////////////////////////
exports
   getSum,
   g_N;
begin
end.
```

Замечание.

К сожалению, **Delphi 6.0** не позволяет использовать экспортированную из DLL переменную при помощи неявной загрузки. Если вам требуется подобная функциональность, придется использовать BPL (**Borland Package Library**).

В случае явной загрузки код приложения можно представить в виде:

```
program XDllClient;
{$APPTYPE CONSOLE}
// явная загрузка
uses
   Windows;
var
   hModule: THandle;
   // объявляем переменную типа "указатель на
   // функцию"
   pGetSum: function(constn 1, n2: integer):
   integer;
   n: integer;
   pg_N: ^integer;
begin
   hModule := LoadLibrary('xdll.dll');
   assert(hModule  0, 'Can''t load DLL!');
   pGetSum := GetProcAddress(hModule, 'getSum');
   assert(@pGetSum  nil, 'Can''t find the getSum
   function!');
   n := pGetSum(10, 20);
   WriteLn('n = ', n);
   pg_N := GetProcAddress(hModule, 'g_N');
   assert(pg_N  nil, 'Can''t find the pg_N
   variable!');
   WriteLn('g_N = ', pg_N^);
   WriteLn;
   WriteLn('Press any key...');
   ReadLn;
   FreeLibrary(hModule);
end.
```

Как видим, благодаря функции **GetProcAddress** можно получить необходимый виртуальный адрес для работы с экспортированной переменной. Использование такой переменной в программе тривиально.

Таким образом, **Delphi** избавляет нас от многих проблем, связанных с применением DLL (в том числе связанных с декорированием имен).

С классом дело обстоит несколько сложнее. Как вы знаете, классы в C++ характеризуются двумя важными с точки зрения программирования функциями — конструктором и деструктором. В C++ (в отличие

Таблица 3

Процесс работы с объектом класса

Требуемое действие	Действие компилятора
Создать объект	1. Выделение памяти под объектом — размер этой памяти всегда можно узнать при помощи оператора **sizeof**. Указатель на эту память характеризуется параметром **this**. 2. Автоматическая генерация кода вызова функции конструктора. Так как память под переменные члены была выделена в п.1, действия конструктора будут корректны. Конструктору также передается параметр **this**
Вызвать метод	Генерация кода вызова функции с передачей ей параметра **this** — именно он характеризует область памяти, связанной с конкретным объектом
Уничтожить объект	1. Генерация кода вызова функции деструктора. Деструктор также получает параметр **this**. 2. Уничтожение памяти, выделенной под объект. Уничтожается ровно **sizeof** байт памяти. Далее любые ссылки при помощи **this** будут некорректны

от **Object Pascal**) вы не обязаны помнить о преднамеренном вызове конструктора — компилятор всегда автоматически генерирует код для его вызова при создании объекта. Кроме того, он также генерирует код вызова деструктора при необходимости уничтожить объект. Все это происходит прозрачно для программиста, так что он всегда может быть уверен, что конструктор будет вызван при инициализации объекта, а деструктор — при его уничтожении (финализации).

Алгоритм этого процесса представлен в табл. 3.

Кроме того, каждому члену-функции обязательно передается неявный параметр **this**, характеризующий конкретный объект этого класса.

Замечание.

Ценные идеи по реализации данного процесса были почерпнуты с сайта http://www.rsdn.ru.

С учетом сделанных рассуждений должно получиться примерно следующее (см. пример 4).

Как видите, приходится несколько раз «жульничать». Во-первых, надо выделить память под предполагаемый объект — мы делаем это, объявляя массив **char** (на стеке или в куче). Только после этого можем вызвать конструктор. Для этого надо объявить указатель на член-функцию. Но так как **GetProcAddress** возвращает нам **FARPROC**, мы должны «притвориться», что работаем именно с этим типом. После выполнения требуемой работы надо вызывать деструктор — делается это аналогичным способом.

Ну, и конечно же при работе с динамической памятью не забываем вызывать **delete[ ]**.

Замечание.

Давайте немного поразмышляем на тему, почему мы не можем сделать так же, как при обычном получении указателя на функцию. Дело в том, что стандарт языка C++ запрещает явные приведения типов, когда в этом приведении участвуют указатели на члены-функции, т. е. обычным (явным) приведением типов мы не имеем права из обычного указателя на функцию получить указатель на член-функцию. Но не имеем права не значит, что не можем. Для этого мы приводим не правую часть выражения, а левую! Приводим к указателю на функцию и по полученному адресу записываем адрес получаемой функции. Только и всего. А после этого выражения опять работаем с **pMemFunc** как с указателем на член-функцию.

Пример 4

Visual C++

Для объекта, создаваемого на стеке:

```
...
   /* используем класс: ну, что ж, вы сами этого хотели... */
   /* 1: получаем адрес конструктора */
   // использование typedef исключительно для красоты - см. ниже пример без typedef
   typedef void (CSummator::*PConstructor)(int);
   // объявляем указатель на член-функцию
   PConstructor pConstructor = NULL;
   // "обманываем" компилятор
   *(FARPROC*)&pConstructor = GetProcAddress(hModule, "CSummatorConstructor");
   _ASSERT(pConstructor != NULL);

   /* 2: создаем (вручную) объект данного класса */
   // объект на стеке
   char p_ch1[sizeof(CSummator)];
   CSummator& sum1 = *(CSummator*)p_ch1;

   /* 3: теперь мы можем вызвать конструктор для данного объекта! */
   (sum1.*pConstructor)(10);

   /* 4: теперь можем использовать метод созданного класса! */
   int (CSummator::*pAdd)(int);
   *(FARPROC*)&pAdd = GetProcAddress(hModule, "CSummatorAdd");
   _ASSERT(pAdd != NULL);
   (sum1.*pAdd)(20);

   /* 5: и не забываем конечно же вызвать деструктор! */
   void (CSummator::*pDestructor)();
   *(FARPROC*)&pDestructor = GetProcAddress(hModule, "CSummatorDestructor");
   _ASSERT(pDestructor != NULL);
   (sum1.*pDestructor)();
...
```

Можно также выделить объект в куче:

```
...
   /* используем класс: ну, что ж, вы сами этого хотели... */
   /* 1: получаем адрес конструктора */
   void (CSummator::*pConstructor)(int);
   *(FARPROC*)&pConstructor = GetProcAddress(hModule, "CSummatorConstructor");
   _ASSERT(pConstructor != NULL);

   /* 2: создаем (вручную) объект данного класса */
   // объект в куче
   char* p_ch2 = new char[sizeof(CSummator)];
   CSummator* pSum2 = (CSummator*)p_ch2;

   /* 3: теперь мы можем вызвать конструктор для данного объекта! */
   (pSum2->*pConstructor)(10);

   /* 4: теперь можем использовать метод созданного класса! */
   int (CSummator::*pAdd)(int);
   *(FARPROC*)&pAdd = GetProcAddress(hModule, "CSummatorAdd");
   _ASSERT(pAdd != NULL);
   (pSum2->*pAdd)(20);

   /* 5: и не забываем конечно же вызвать деструктор! */
   void (CSummator::*pDestructor)();
   *(FARPROC*)&pDestructor = GetProcAddress(hModule, "CSummatorDestructor");
   _ASSERT(pDestructor != NULL);
   (pSum2->*pDestructor)();

   /* и не забываем освобождать кучу */
   delete[] p_ch2;
...
```

Остались нерассмотренными два важных вопроса:

1) работа с виртуальными функциями;
2) работа со статическими функциями и переменными.

Предположим, мы добавили в описание нашего класса следующие строки:

```
//////////////////////////////////////////
// class CSummator
class XDLL_API CSummator
{
public:
   ...
   virtual int GetBalance();

   static int GetDevilSum();
   static int m_DevilSum;
   ...
};
```

и соответствующим образом их реализовали (см. пример 5).

Обратите внимание на закомментированную директиву **#pragma** — это не опечатка (почему так сделано — см. ниже).

Сначала по поводу статических данных и переменных. Так как эти функции не требуют скрытого указателя **this**, можно работать с ними точно так же, как и с обычными функциями, не являющимися членами класса (см. пример 6).

С виртуальными функциями дело обстоит несколько иначе.

Здесь немного поговорим о примерах кода, описанных выше. Почему приходится применять дополнительный код, связанный с получением адреса метода класса из DLL при использовании явной загрузки? Да потому, что при указании вызова метода компилятор сразу же вставляет код вызова такой функции (с указанием виртуального адреса). Но линкер при компоновке не может разрешить этот адрес, так как он его попросту не знает (ведь у него нет lib-файла)! Поэтому приходится явно получать

Пример 5

```
...
//#pragma comment(linker, "/export:CSummatorGetBalance=?GetBalance@CSummator@@UAEHXZ")
#pragma comment(linker, "/export:CSummatorGetDevilSum=?GetDevilSum@CSummator@@SAHXZ")
#pragma comment(linker, "/export:CSummatorm_DevilSum=?m_DevilSum@CSummator@@2HA")

int CSummator::GetDevilSum()
{
   return 666;
}
int CSummator::GetBalance()
{
   return m_N;
}
int CSummator::m_DevilSum = 666;
...
```

Пример 6

```
...
   /* и со статическими членами-данными и членами-функциями */
   int* pm_DevilSum = (int*)GetProcAddress (hModule, "CSummatorm_DevilSum");
   _ASSERT(pm_DevilSum != NULL);
   int (*pGetDevilSum)();
   const int n2 = *pm_DevilSum; // вызов статической функции
   (FARPROC&) pGetDevilSum = GetProcAddress (hModule, "CSummatorGetDevilSum");
   _ASSERT(pGetDevilSum != NULL);
   const int n3 = pGetDevilSum(); // вызов статического члена-данного
...
```

этот виртуальный адрес при помощи **GetProcAddress**.

Виртуальные функции ведут себя по-другому. Их вызовы компилятор разрешает несколько иначе — при помощи таблицы виртуальных методов (**vtbl**), указатель на которую вы также найдете в списке декорированных имен. Таким образом, при вызове этой функции компилятор должен вставить не ее виртуальный адрес, а ее смещение в таблице виртуальных вызовов (которая заполняется после вызова конструктора класса). Значит, никаких проблем с разрешением адресов возникнуть не может. Следовательно, виртуальные функции можем использовать точно так же, как и обычно:

```
...
   /* поиграем немного с виртуальной функцией */
   const int n1 = sum1.GetBalance();
...
```

Встретив подобный код, компилятор сгенерирует код вызова функции по ее смещению в таблице (так называемая косвенная адресация) на основе определения класса. Именно поэтому нам даже не придется заботиться о декорировании имен этой функции.

Delphi 6.0

К сожалению, и здесь эта среда разработки покинет нас на произвол судьбы. Для экспорта классов потребуется вновь обратить взоры в сторону BPL.

Следует помнить о следующих ограничениях, которые накладываются на экспорт классов из DLL в среде **Delphi**:

1. Вызов методов класса возможен только посредством таблицы виртуальных методов (**vtbl**). Следовательно, все используемые извне методы должны быть объявлены с модификатором **virtual**. Почему это работает? Дело в том, что при объявлении виртуального метода класса любой вызов этого метода осуществляется не напрямую (т. е. посредством перехода по конкретному адресу), а через специальную таблицу — компилятор в месте вызова метода генерирует обращение к этой таблице по «номеру» этого метода к ячейке, в которой лежит истинный адрес метода. Это позволяет, не зная истинного адреса функции на момент старта приложения, (автоматически) вычислить его при обращении к методу класса.

2. Экземпляры класса должны быть созданы внутри DLL. Этот пункт следует из первого требования.

3. Декларации класса как в DLL, так и в приложении должны быть объявлены в одном и том же порядке. Это обеспечит корректность генерируемого компилятором кода обращения к **vtbl** со стороны как DLL, так и приложения.

4. Нет возможности унаследовать класс от класса, заключенного внутри DLL.

Чтобы удовлетворялись одновременно перечисленные 4 требования, необходимо создать заголовочный файл объявления нашего класса **TSummator** в следующем виде (см. пример 7).

Как видим, директивы препроцессора позволяют автоматически исключать (или, наоборот, включать) часть кода в случае выполнения/невыполнения указанных условий.

В случае определения символа **__SUMMATORLIB** (это осуществляется в коде DLL) получим следующее определение класса (см. пример 8).

Если же такой символ определен не будет (это должно быть справедливо для любого клиента, использующего наш класс), получим определение класса вида (см. пример 9).

Техника использования заголовочных файлов очень широко применяется в языках C/C++. В данном случае **inc**-файл позволяет обеспечить удовлетворение всех требований, указанных выше.

Клиенту будут доступны исключительно виртуальные методы **TSummator.Add** и **TSummator.GetSum**. Создание класса также придется осуществлять внутри кода DLL — для этого специально выделим отдельную экспортируемую функцию.

Пример 7 (файл **Summator.inc**)

```
type
   TSummator = class(TObject)
   public
{$IFDEF __SUMMATORLIB}
      constructor Create(const n: integer);
      destructor Destroy; override;
{$ENDIF}

      function Add(const n: integer): integer; virtual; stdcall;
          {$IFNDEF __SUMMATORLIB}abstract;{$ENDIF}
      function GetSum(): integer; virtual; stdcall;
          {$IFNDEF __SUMMATORLIB}abstract;{$ENDIF}
{$IFDEF __SUMMATORLIB}
   private
      m_N: integer;
{$ENDIF}
   end;
```

Пример 8

```
type
   TSummator = class(TObject)
   public
      constructor Create(const n: integer);
      destructor Destroy; override;

      function Add(const n: integer): integer; virtual; stdcall;
      function GetSum(): integer; virtual; stdcall;
   private
      m_N: integer;
   end;
```

Пример 9

```
type
   TSummator = class(TObject)
   public

      function Add(const n: integer): integer; virtual; stdcall; abstract;
      function GetSum(): integer; virtual; stdcall; abstract;
   end;
```

Выполнение пункта 3 (эквивалентный порядок объявления виртуальных методов) обеспечивается за счет использования одного заголовочного файла как для DLL, так и для клиента. «А нельзя ли обойтись без дополнительных файлов?», — спросите вы. Можно, но при этом вероятность совершить ошибку будет значительно выше.

Объявление методов с модификатором **abstract** делает невозможным их дальнейшее переопределение — при этом компилятор не требует наличия их явного определения в коде приложения.

С **INC**-файлом разобрались. Теперь рассмотрим определение методов (см. пример 10).

Таким образом, определяем символ **__SUMMATORLIB** и лишь после этого подключаем **INCLUDE**-файл. Затем реализуем конструктор и деструктор, а также члены-методы **AddSum** и **GetSum**. Думаем, особых сложностей с пониманием методов их реализации возникнуть не должно.

Чтобы удовлетворить требованиям пункта 2, мы также экспортируем из DLL специальную функцию конструирования объекта — **InitSummator**, которая занимается вызовом конструктора объекта **TSummator** и возвращает ссылку на него.

Теперь есть все необходимое, чтобы попробовать использовать этот класс в приложении. Рассмотрим пример.

Для начала создадим в приложении вспомогательный модуль, в котором будут описаны соответствующие экспортируемые функции.

Пример 10 (файл **XDll.dpr**)

```
{
   Следующее определение (директива #DEFINE) необходимо ТОЛЬКО в теле DLL; благодаря этому класс
TSummator раскрывается по-разному в DLL и приложении, которое ее использует.
   В случае приложения мы получим определение в виде:

   type
     TSummator = class(TObject)
          public
              function Add(const n: integer): integer; virtual; stdcall; abstract;
     end;

   В коде DLL мы имеем "полнофункциональную" версию с конструктором и деструктором!
}
{$DEFINE __SUMMATORLIB}
{*.inc-файлы - аналоги header-файлов в языках C/C++}
{$I Summator.inc}

constructor TSummator.Create(const n: integer);
begin
   inherited Create;
   m_N := n;
end;

destructor TSummator.Destroy();
begin
   inherited Destroy;
end;

function TSummator.Add(const n: integer): integer;
begin
   // Inc(m_N, n);
   m_N := m_N + n;
   Add := m_N;
end;

function TSummator.GetSum: integer;
begin
   Result := m_N;
end;

{далее определяем функцию конструирования объекта TSummator}
function InitSummator(const n: integer): TSummator; stdcall;
begin
   InitSummator := TSummator.Create(n);
end;
```

Файл **XDllFuncUnit.pas**.

```
unit XDllFuncUnit;
interface
{$I ..\XDll\Summator.inc}
   // объявление функции в интерфейсе модуля
   function InitSummator(const n: integer):
   TSummator; stdcall;
implementation
   // эта функция импортируется из DLL
   function InitSummator(const n: integer):
   TSummator; external '..\XDll\XDll.dll';
end.
```

Здесь подключаем тот же **INC**-файл, который использовался в DLL, чтобы обеспечить корректность декларации **vtbl** объекта **TSummator**. Объявляем импортируемую функцию создания объекта **InitSummator**.

Текст главного приложения предстанет в следующем виде:

```
program XDllClient;
var
   n: integer;
   sum: TSummator;
begin
   // создаем объекта класса TSummator
   sum := InitSummator(30);
   // пример использования методов
   // импортированного класса
   sum.Add(20);
   sum.Add(30);
   WriteLn('sum.GetSum() = ', sum.GetSum());
   WriteLn;
   WriteLn('Press any key...');
   ReadLn;
end.
```

Инициализируем объект с указанием стартовой суммы, затем производим двойное суммирование. Таким образом, после старта приложения на экране вы увидите заветный результат **'sum.GetSum() = 80'**.

С одной стороны, использование экспортированных классов ничем не отличается от применения обычных функций (особенно в случае помощи со стороны компилятора). С другой стороны, в этом заключается большая опасность.

Старайтесь не прибегать к экспорту классов из DLL. Кроме того, что экспорт классов является серьезной ошибкой проектирования DLL, возникающие неочевидные ошибки с несоответствием в моделях управления памятью могут поставить крест на применении такой DLL в дальнейшем.

Экспорт классов может быть приемлем в случае использования DLL в одной конкретной среде (как в случае MFC). Как было показано выше, среда разработки накладывает на экспортированные методы класса типичные для нее отпечатки при экспорте членов-функций. Подобные правила могут разительно отличаться от тех, что будет использовать другая среда пользователя, применяющего такую DLL. Поэтому экспорт классов из DLL, имеющих общее применение, является дурным тоном и, как правило, не применяется.

Использование экспортированных классов может привести к неожиданным проблемам и в случае эксплуатации одной и той же среды. Подобные ошибки обычно связаны с использованием динамической памяти в классах (как это происходит в STL) — при этом ошибки переключения контекста кучи (**heap**) не всегда могут быть адекватно учтены пользователем конкретных классов. Если такие классы целиком находятся в области видимости приложения (т. е. статически линкуются вместе с ним), подобные проблемы возникнуть не могут — все классы используют одну и ту же область динамической памяти. Если же классы экспортируются из DLL, то они вполне могут использовать собственный локальный **heap**, а программа пользователя может об этом даже и не подозревать. В дальнейшем это приведет к неадекватному поведению программы и к ее аварийному завершению в связи с нарушением доступа к памяти.

Список литературы

1. *Рихтер Дж.* Windows для профессионалов. Создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows. М.: Русская редакция; СПб.: Питер, 2001.

2. Стандарт языка C++. International Standart ISO/IEC 14882. Programming Languages — C++. First Edition 1998-09-11.