А.В. Леденев, И.А. Семенов, В.А. Сторожевых

Динамически загружаемые библиотеки: структура, архитектура и применение

DLL — это сокращение от Dynamic Link Library (динамически загружаемая библиотека). С формальной точки зрения DLL — 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 посвящен особенностям реализации DLL в различных средах и для различных целей.

Что такое DLL?

Особый способ оформления предполагает наличие в DLL так называемых *секций импорта* и *экспорта*. Секция экспорта указывает те идентификаторы объектов (функций, классов, переменных), доступ к которым предоставляет данная DLL. В этом случае мы говорим об экспортировании идентификаторов из DLL. В общем случае именно секция экспорта представляет особый интерес для разработчиков. Хотя ничто не мешает реализовать DLL, которая не имеет данной секции, но тем не менее выполняет полезную работу.

Относительная независимость связана с наличием/отсутствием секции импорта у DLL (т. е. секции, в которой описываются внешние зависимости данной DLL от других). Подавляющее большинство DLL (за исключением, быть может, DLL-ресурсов) импортирует функции из системных DLL (**kernel32.dll**, **user32.dll**, **gdi32.dll** и проч.). В большинстве случаев при создании проекта в его опциях автоматически проставляется стандартный набор таких библиотек. Иногда в этот список необходимо добавить DLL, требующиеся для конкретных задач (например, в случае использования библиотеки сокетов требуется дополнительно подключить библиотеку ws2_32.dll).

Исполняемый код в DLL не предполагает автономного использования. Перед тем как можно будет приступить к использованию, необходимо загрузить DLL в область памяти вызывающего процесса (т. е. DLL не может выполняться сама по себе — ей обязательно нужен клиент). Это явление носит название «*проецирование DLL на адресное пространство процесса*». И это не удивительно, если вспомнить тот факт, что процессор работает не только с регистрами, но и с адресами памяти. Поэтому каждому объекту DLL требуется свое место «под солнцем», чтобы иметь возможность быть выполненным при вызове. В конечном коде exe-файла, который генерирует компилятор, не будет инструкций процессора, соответствующих коду данной функции. Вместо этого будет сгенерирована инструкция вызова соответствующей функции (call). Так как DLL отображена на адресное пространство процесса, то код DLL будет легко доступен по call-вызову.

Итак, формально, DLL — особым образом оформленный программный компонент, доступ к исполняемому коду которого приложение получает в момент старта (DLL неявной загрузки) или в момент использования (DLL явной и отложенной загрузки).

Что же касается физического представления на диске, то разница между dll- и exe-файлами небольшая. Как динамически лин-

куемые библиотеки, так и исполняемые модули приложений в **Windows** имеют формат **Portable Executable** (PE-файл), однако нельзя «запустить» DLL-библиотеку на выполнение, как обычное приложение. «Узнать» DLL-файл можно по его сигнатуре (заголовку): признаком библиотеки является установленный флаг **IMAGE_FILE_DLL** (13-й бит) в поле **Characteristics** заголовка **IMAGE_FILE_HEADER** (эти константы описаны в файле **winnt.h**; подробная информация о формате exe- и dll-файлов — см. **MSDN «Microsoft Portable Executable and Common Object File Format Specification»**). Кроме того, в заголовках файлов динамически линкуемых библиотек указан нулевой размер стека — это связано с тем, что функции DLL используют стек вызывающего приложения. DLL-файл может содержать как инструкции (команды процессора), так и данные (разделяемые ресурсы).

В общем случае файл, являющийся динамически загружаемой библиотекой, не обязан иметь расширение **.dll**. Например, известные файлы ***.cpl** — это не что иное, как DLL, используемые апплетом панели управления; ***.ocx** — DLL, содержащие внутрипроцессные (inproc) COM-объекты.

Логическое (философское) представление DLL не имеет никаких ограничений. Удобно представлять себе DLL в виде сервера, который предлагает дополнительную функциональность приложению. Приложения, которые используют данную функциональность, являются клиентами DLL. Рисунок 1 показывает процесс взаимодействия приложения с DLL. После проецирования DLL на адресное пространство вызывающего процесса DLL становится частью этого процесса. Поэтому возможен абсолютно безболезненный вызов функций, экспортируемых DLL.

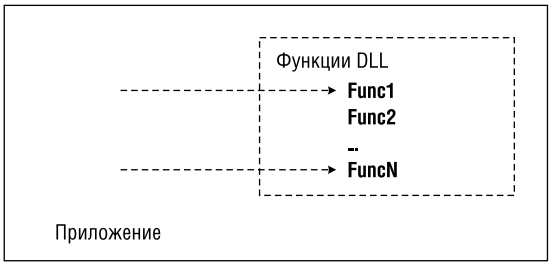

Рис. 1. Взаимодействие приложения с DLL

Зачем нужны DLL?

Свою историю DLL ведут с середины 60-х годов прошлого столетия, однако по-настоящему широкое распространение динамически линкуемые библиотеки получили после появления операционной системы **Windows**[1].

По крайней мере, доподлинно известно, что операционные системы **Windows 3.1/3.11** уже содержали программы, использующие DLL. В связи с тем, что в те времена емкости оперативной памяти и жесткого диска были значительно меньше, чем сейчас, использование DLL предоставляло ряд преимуществ.

- Экономия дискового пространства за счет многократного использования кода (**reusing**). Если приложения используют один и тот же код, нет необходимости поставлять его в коде каждого приложения. Достаточно разработать DLL.
- Экономия физической памяти (RAM) за счет загрузки в нее единственного экземпляра DLL. Именно тогда появились счетчики ссылок пользователей DLL — при каждом вызове функции ОС проверяет наличие загруженного в память экземпляра библиотеки. В случае положительного ответа счетчик ссылок пользователей данной DLL увеличивается на единицу. Если же эк-

[1] Крис Касперски («Техника сетевых атак», 2001) пишет: «Именно в **MULTICS** (1965–1969 гг.) появилась возможность динамического связывания модулей в ходе выполнения программы, более известная современному читателю по этим пресловутым DLL в **Windows**. Такой прием логически завершил эволюцию совершенствования оверлеев, обеспечив единый, унифицированный интерфейс для всех программ, позволяя сэкономить значительную часть оперативной памяти и процессорных ресурсов...»

земпляр данной DLL в памяти не обнаружен, то операционная система загружает файл в память и присваивает счетчику значение «1». Механизм разделения кода носит название **memory mapping** (отображение в память). При выгрузке DLL из памяти уменьшается значение счетчика числа пользователей, в случае равенства его нулю DLL немедленно выгружается.

• Изолирование и модификация кода DLL независимо от остального кода программы. Например, код визуализации изолируется от математической части. При изменении математического аппарата (например, при разработке нового, более быстрого алгоритма) перекомпиляция кода клиентского приложения (отвечающего за визуализацию результатов) не требуется. Этот фактор может играть значительную роль в том случае, если число клиентов достаточно велико.

Самое удивительное (время не стоит на месте!), что ранее политика **Microsoft** позволяла (и даже приветствовала) размещение DLL в системных директориях (таких как **Windows/System**, **Windows/System32**). Это порождало периодические проблемы конфликта версий при замене DLL (см. раздел «DLL Hell»). В связи с этим (а также с ростом емкости запоминающих устройств и соответствующим снижением цены на них) на данный момент политика **Microsoft** изменена на прямо противоположную — **Microsoft** настоятельно рекомендует хранить все используемые DLL в рабочем каталоге программы и лишь в случае острой необходимости пользоваться системными директориями.

Основные направления использования DLL:

• всевозможные модули расширения функциональности приложений — так называемые **plug-in** (далее приводится пример с **MatLab**, **Far** и проч.);

• локализация приложения;

• разделение объектов абстракции (функций, классов и проч.) между приложениями;

• независимость модификации кода — DLL может быть в любой момент переписана с сохранением экспортируемых интерфейсов;

• реализация определенных действий, которые можно совершить только при помощи DLL;

• хранилище ресурсов с возможностью независимого изменения этих ресурсов.

Кстати, DLL широко используются в технологии COM (а до этого — в OLE 1.0) — в качестве основы при построении так называемых **inproc**-серверов (внутрипроцессных серверов) используются DLL. Это позволяет упростить взаимодействие с приложением благодаря загрузке используемых **ActiveX** объектов в адресное пространство клиента. В этом случае накладные расходы, связанные с преодолением границ адресных пространств при передаче данных (параметров функций и т. д.) — так называемый **marshalling**, сводятся к нулю.

Все те абстракции, с которыми приходится работать в повседневной программистской жизни, могут быть внедрены в DLL — классы, объекты, таймеры, потоки, функции и проч. Однако не всегда удобно и правильно работать с этими объектами вне DLL. Связано это с тем, что не всегда логическое представление того или иного объекта может быть однозначно представлено (переведено) в физическое. Использование DLL не налагает ограничений на используемый язык (точнее, почти не налагает). Более того, как правило, DLL разрабатывается на другом языке программирования, нежели тот, который используется при ее загрузке. Приведем пример: разрабатывался математический проект, код которого реализовывался на M-языке среды **MatLab** (**Matrix Laboratory**). M-язык по своей природе является интерпретируемым языком программирования. После этого полученные алгоритмы были реализованы при помощи языка C++ и скомпилированы в DLL, которая также использовалась средой **MatLab**.

Таблица 1

Сравнение производительности исполнения кода (*.DLL vs *.M)

Задача — построение выпрямляющего функционала. Количество точек — 864 001	Среда подготовки исполняемого модуля. Время исполнения, с	
	Visual Studio 7.0	Matrix Laboratory
Функционал «Энергия»	0,44	35,15
Функционал «Длина»	0,22	34,05
Функционал «Регрессия»	31,25	148,52

Таким образом, аналогичная реализация в скомпилированном варианте дает выигрыш по скорости исполнения от пяти до ста раз! И это не предел. Разумеется, подобная эффективность может быть очень хорошо использована при наличии соответствующих возможностей. Необходимо отметить, что DLL — мощный и удобный механизм и пользоваться им надо с умом.

Что лучше: один exe-файл и никаких DLL или компактный exe-файл и много DLL?

Небезызвестный Бьерн Страуструп в своей книге «Язык программирования C++» дает после каждой главы ряд полезных советов, один из которых звучит так: «Пользуйтесь здравым смыслом...» Этот же совет можно предложить и при использовании DLL в программе.

Разумеется, не стоит каждую отдельную функцию размещать в DLL и затем экспортировать (хотя это и возможно). Со временем такой проект превратится в трудноуправляемого монстра, инициализация которого будет занимать порядочное время. Правда, экспортирование только одной функции из DLL может быть продиктовано следующей логикой: серверное приложение сканирует определенную директорию на предмет нахождения в ней файлов с определенным расширением. В случае нахождения такого файла сервер полагает, что это — DLL, которая экспортирует функцию с определенным именем (скажем, **CplApplet** или **mexFunction**), и, соответственно, он сможет ее загрузить. Таким простым образом возможно динамическое расширение функциональности приложения — именно такой подход проповедуется в случае Панели управления (**Control Panel**) и системы **MatLab** (**Matrix Laboratory**).

Не стоит также бояться использовать те плюсы, которые может принести использование DLL в программе.

Логически отделенная библиотека, часто используемая в совместных проектах, — верный кандидат на размещение в DLL. Это позволит:

- поставлять различным клиентам приложения с различной функциональностью;
- модифицировать (оптимизировать, изменять) код DLL без повторной перекомпиляции клиентских приложений, использующих DLL. А это значит, что при модификации или исправлении ошибки достаточно изменить DLL (сама **Microsoft** часто именно так и делает: постоянно появляются пакеты обновлений **Service Pack**);
- разделить разработку большого проекта на отдельные, независимые группы;
- обеспечить легкость отладки и тестирования. В этом случае возможности, связанные с использованием нисходящих технологий проектирования, существенно повышаются;
- использовать различные языки при написании DLL и клиентского приложения. Например, клиент может быть написан при помощи m-языка (тип интерпретируемого скрипта, используемый в системе **MatLab**), а DLL — при помощи языка C++. Это позволит провести дополнительную оптимизацию кода без потери его качества.

Разумеется, каждая DLL требует определенного процесса инициализации и, соответственно, затрат на него. В случае наличия большого количества DLL процесс инициализации может занимать порядочное

время (правда, необязательно все DLL инициализировать сразу, а также необязательно вообще инициализировать DLL — ведь иногда из DLL нужно получить не код, а находящиеся в ней ресурсы — например, в случае использования DLL как хранилища локализованных строковых параметров).

В случае помещения всего кода в exe-файл дополнительная инициализация не потребуется. Правда, при этом будет происходить постоянное дублирование кода. В случае единичного проекта это не составляет проблемы. Если же таких проектов несколько, возможности повторной перекомпиляции бывают существенно затруднены (и неэффективны). Представьте себе, что было бы, если бы при каждом изменении **Microsoft** заставляла бы всех клиентов перекомпилировать «ядро» **Windows**?

Кроме того, помните, что определенные проблемы проектирования без использования DLL (например, связанные с установкой ловушек) решить вообще нельзя.

Еще можно вспомнить так называемые **ISAPI**-расширения — высокоэффективные модули, используемые при написании **Web**-приложений. Ну и, наконец, такая система, как **MatLab**, позволяет использовать собственноручно написанные и скомпилированные модули в виде DLL для повышения быстродействия работы программ. Так что использование DLL может быть продиктовано еще и вопросами оптимизации.

Подытоживая сказанное выше, можно сделать следующие выводы.

Плюсы и минусы в случае использования одного большого exe-файла:

(+) относительная быстрота инициализации — каждая DLL требовала бы отдельного процесса инициализации;

(+) все в одном файле (нет внешних зависимостей) — такой файл может поставляться без внешних библиотек;

(–) постоянное дублирование кода (за счет увеличения размера *.exe-файла);

(–) полная перекомпиляция при любом изменении;

(–) все «в одной куче»: отсутствие разделения реализации логики различных (по функциональному наполнению) объектов;

(+) идеально для небольших проектов (утилиты, тестовые приложения и проч.).

Плюсы и минусы в случае разделения EXE и DLL:

(+) физическое разделение логически независимых объектов (классов, функций и проч.). Это позволяет проводить независимую разработку и последующее тестирование подобных DLL. На момент сбора объекты будут иметь достаточно предсказуемое поведение, чтобы быть сразу же использованными другими разработчиками;

(–) возможно большее время загрузки (если не применять различного рода оптимизирующих действий — отложенную загрузку, управление предпочтительными базовыми адресами и проч.);

(+) идеально для больших проектов (повторное использование кода + отсутствие его дублирования);

(+) легкость обновления и замены.

А что использовать в каждом случае — это дело конкретной задачи.

Как создать собственную DLL

Итак, мы готовы от теории перейти к практике, т. е. к написанию собственной динамически загружаемой библиотеки. Средства для создания библиотек DLL имеются практически во всех средах разработки для **Windows**. Мы рассмотрим процесс создания DLL с использованием наиболее популярных в настоящий момент систем программирования.

Создадим простенький пример DLL — пусть она экспортирует функцию, которая будет реализовывать сверхсложный и, что самое главное, уникальный алгоритм — получать на вход два параметра, складывать их и возвращать результат.

Что для этого необходимо сделать?

Visual C++ 6.0

1. Создать проект типа **«Win32 Dynamic-Link Library»**. Для этого необходимо выбрать соответствующую опцию в левом окне, затем ввести имя проекта в окне **«Project name»** и нажать кнопку **«Ok»**.

Замечание:

Существуют также проекты типа **«Win32 Static Library»**. При компиляции проектов данного типа создаются статические (в отличие от динамических) библиотеки — имеющие расширение *.lib. Отличие статических библиотек от динамических состоит в том, что при компиляции код, заключенный в lib-файл (а там располагается именно код), помещается целиком и полностью в бинарный файл использующего его приложения. При компиляции проекта **«Win32 Dynamic-Link Library»** также создается lib-файл. Но он содержит лишь необходимую информацию для связывания адресов DLL-файла и приложения.

Как, спрашивается, отличить lib-файл статической библиотеки от lib-файла динамической? Очень просто — используйте утилиту **dumpbin** с параметром **/headers**. Для примера приведем примеры вывода этой утилиты в различных случаях (см. стр. 37).

В описаниях функций lib-файла библиотеки динамической загрузки упоминается имя файла требуемой (связанной с ними) DLL — при вызове функции происходит попытка вызова этой DLL (посредством загрузки в память и последующим процессом проецирования). Функции выше (lib-файл статической библиотеки) не имеют такого параметра — их код явно подставляется в тело вызывающего приложения при вызове.

2. Далее можно либо создать пустой проект (и самостоятельно добавить в него файлы), либо создать простой проект с заданным по умолчанию шаблоном, либо создать проект, в котором изначально будет предусмотрен экспорт функций и классов. Мы пойдем по наиболее тернистому первому пути.

3. Добавьте в проект следующие файлы (см. стр. 38).

Замечание:

Будьте осторожны со встраиваемыми (**inline**) функциями в DLL. Опасность заключается в том, что код таких функций явно подставляется в код приложения-клиента. Это не вызывает проблем до тех пор, пока не произойдет изменения кода библиотеки. В этом случае придется перекомпилировать код приложения, использующего конкретную DLL. Как правило, код встраиваемых функций тривиален и подвержен очень редким изменениям. Но знать об этом, тем не менее, нужно.

4. И не забываем определить идентификатор **XDLL_EXPORTS** в настройках проекта — **«Projects settings->C++->General->Preprocessor definitions»**. В случае создания проекта DLL VC++ выполняет эту операцию автоматически. После успешной компиляции данного проекта мы можем убедиться, что в директории вывода объектных файлов появились два файла — **XDll6.dll** и **XDll6.lib**. Первый из них представляет собой непосредственно DLL. Посмотрим, что нам выдаст **dumpbin** относительно **XDll6.dll**:

```
Microsoft (R) COFF/PE Dumper Version 7.00.9466
Copyright (C) Microsoft Corporation. All rights
reserved.

Dump of file XDll6.dll
File Type: DLL
Section contains the following exports for XDll6.dll
00000000 characteristics
3F50EECB time date stamp Sat Aug 30 22:36:59 2003
    0.00 version
       1 ordinal base
       1 number of functions
       1 number of names

ordinal hint RVA name
      1 0 00001005 ?getSum@@YAHHH@Z
Summary
    4000 .data
    1000 .idata
    2000 .rdata
    2000 .reloc
   28000 .text
```

Lib-файл статической библиотеки

```
...
SECTION HEADER #3
      .text name
          0 physical address
          0 virtual address
         25 size of raw data
        486 file pointer to raw data (00000486 to 000004AA)
          0 file pointer to relocation table
        4AB file pointer to line numbers
          0 number of relocations
          3 number of line numbers
   60501020 flags
            Code
            COMDAT; sym= "int __cdecl getSum(int,int)" (?getSum@@YAHHH@Z)
            16 byte align
            Execute Read
SECTION HEADER #5
      .text name
          0 physical address
          0 virtual address
         28 size of raw data
        53F file pointer to raw data (0000053F to 00000566)
          0 file pointer to relocation table
        567 file pointer to line numbers
          0 number of relocations
          3 number of line numbers
   60501020 flags
            Code
            COMDAT; sym= "int __cdecl getSum(int,int,int)" (?getSum@@YAHHHH@Z)
            16 byte align
            Execute Read
...
```

Lib-файл динамической библиотеки

```
...
Version       : 0
Machine       : 14C (x86)
TimeDateStamp : 3F54CD9E Tue Sep 02 21:04:30 2003
SizeOfData    : 0000001B
DLL name      : XDll6.dll
Symbol name   : ?getSum@@YAHHH@Z (int __cdecl getSum(int,int))
Type          : code
Name type     : name
Hint          : 0
Name          : ?getSum@@YAHHH@Z

Version       : 0
Machine       : 14C (x86)
TimeDateStamp : 3F54CD9E Tue Sep 02 21:04:30 2003
SizeOfData    : 0000001C
DLL name      : XDll6.dll
Symbol name   : ?getSum@@YAHHHH@Z (int __cdecl getSum(int,int,int))
Type          : code
Name type     : name
Hint          : 1
Name          : ?getSum@@YAHHHH@Z
...
```

Файл **XDll.h**

```
#ifndef __XDLL_H
#define __XDLL_H

/* Символ XDLL6_EXPORTS по умолчанию определен в проекте (см. Настройки проекта->C/C++->General->
Preprocessor Definitions). При этом все экспортируемые идентификаторы предваряются символом XDLL_API.
Что это дает?
В случае определения XDLL6_EXPORTS в проекте XDLL_API определяется как экспортируемый объект; в случае
же отсутствия такого определения будет получен импортируемый объект.
Таким образом, один и тот же заголовочный файл может быть использован и в DLL-проекте, и в проекте, кото-
рый будет использовать данную DLL! Без каких-либо изменений. Очень удобно, не правда ли? */

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

/* Каждый экспортируемый идентификатор предваряем __declspec(dllexport).
Эта директива позволяет линкеру определить, что данный идентификатор следует экспортировать из DLL. При
этом создается специальный lib-файл, который содержит все экспортируемые идентификаторы из модуля. Так-
же экспортируемые объекты заносятся в раздел экспорта DLL — это можно проверить при помощи утилиты
dumpbin.exe. */

XDLL_API int getSum(const int n1,const int n2);
#endif // __XDLL_H
```

Файл **XDll.cpp**

```
#include "XDll.h"
XDLL_API int getSum(const int n1,const int n2)
{
   return n1 + n2;
}
```

Увидели знакомую уже информацию? Да, только имя экспортируемой функции имеет, мягко говоря, странный вид. Ну ничего, с этим мы разберемся позже. Пока нам это не будет мешать.

Файл **XDll6.lib** представляет собой список (в особом формате) экспортируемых идентификаторов. Наличие такого файла позволяет проводить так называемую неявную загрузку DLL — в этом случае компилятор и линкер по содержимому файла **XDll6.lib** могут автоматически получить всю информацию, необходимую для правильного разрешения адресов при вызове функции. Но об этом немного позже.

Visual C++ 7.0

Для VC++ 7.0 процесс во многом повторяется. Итак, что нужно сделать:

1. Выбрать тип приложения — **«Win32 Project»**. Ввести название проекта — **«XDll»**.

Замечание:

Существуют дополнительные типы проектов — так называемые **«MFC DLL»**. От обычных проектов они отличаются тем, что предполагают использование в той или иной мере ресурсов библиотеки MFC. В данной статье реализация таких DLL не рассматривается, хотя, основываясь на базовых понятиях, легко можно реализовать DLL данного типа.

2. Уточнить тип проекта — **«DLL»**.

После завершения последовательности действий появится каталог с проектом, в котором будет находиться ряд файлов (если оставить неотмеченным флажок **«Создать пустой проект»**):

• **XDll.cpp** — файл исходного кода библиотеки. На основе исходного кода и строится файл библиотеки DLL.

• **XDll.h** — заголовочный файл экспортируемых идентификаторов. Именно этот файл должен поставляться с DLL, чтобы обеспечить удобство и простоту использования DLL в других проектах.

3. Далее необходимо точно так же модифицировать существующие (или добавить новые, если не создавать каркаса DLL автоматически) файлы заголовка и исходного кода **XDll.h** и **XDll.cpp** соответственно.

4. И, конечно же, не забыть определить идентификатор **XDLL_EXPORTS** (**«C/C++->Preprocessor->Preprocessor Definitions»**) — в случае создания проекта DLL VC++ выполняет эту операцию автоматически.

Borland C++Builder 6.0

1. Для создания нового проекта DLL нужно выбрать команду **«File->New->Other...»**, затем в окне из списка выбрать **«DLL Wizard»**. После этих манипуляций появляется окно настройки некоторых свойств будущей DLL.

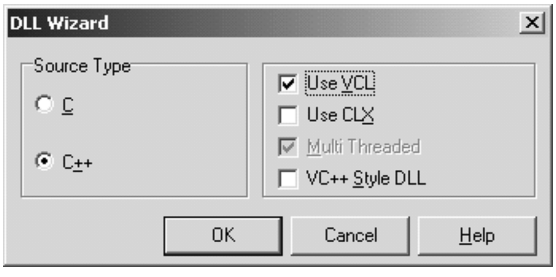

Рис. 2. Окно настройки свойств DLL

«Source type» определяет, какой язык будет использоваться для написания DLL: «чистый» C или C++.

«Use VCL» и **«Use CLX»** указывают линкеру, нужно ли подключать к DLL библиотеки VCL и CLX. **«Multi Threaded»** указывает, будет ли создаваемая библиотека создавать дополнительные потоки (**threads**).

И, наконец, **«VC++ Style DLL»** указывает компилятору, какая функция будет точкой входа в DLL. Если снять этот флажок, то точкой входа в DLL будет функция

```
int WINAPI DllEntryPoint(HINSTANCE hinst,
unsigned long reason, void* lpReserved)
```

Если этот флажок установлен, входной функцией будет

```
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD
fwdreason, LPVOID lpvReserved)
```

Такой вариант введен для совместимости кода с компиляторами **Microsoft**.

2. После выбора этих свойств создается шаблон файла библиотеки DLL. За исключением комментариев, он имеет примерно такой вид:

```
#include <windows.h>

int WINAPI DllEntryPoint(HINSTANCE hinst,
unsigned long reason, void* lpReserved)
{
   return 1;
}
```

Теперь сюда можно добавить экспортируемую функцию. Делается это практически так же, как и в **Visual C++**:

```
extern __declspec(dllexport) int getSum (const
int n1, const int n2);

int getSum(const int n1, const int n2)
{
   return n1 + n2;
}
```

Borland Delphi 6.0

Процесс создания DLL:

1. Выбрать пункт меню **«File->New->Other->DLL Wizard»**. Помощник создаст необходимые текстовые файлы для компиляции проекта типа **«DLL»**.

2. Подправим немного созданный мастером файл, чтобы придать ему требуемую форму:

```
library XDll;
{$R *.res}
////////////////////////////////////////
function getSum(const n1: integer; const n2:
integer): integer;
begin
   Result := n1 + n2;
end;
////////////////////////////////////////
exports
   getSum;
begin
end.
```

Замечание:

Синтаксис **Object Pascal** (**Delphi**) подразумевает возможность определения функции в следующем формате (обратите внимание на явное использование ключевого слова **export**):

```
function getSum(const n1: integer; const n2:
integer): integer; stdcall; export;
var
...
begin
...
end;
```

Указание ключевого слова **export** никоим образом не влияет на компоновку в 32-битных приложениях. Данная директива оставлена для совместимости. Все экспортируемые идентификаторы определяются исключительно в специально предназначенной для этого секции **exports**.

В общем и целом структура проекта типа **«DLL»** очень сильно напоминает структуру обычного приложения **Delphi** — только вместо ключевого слова **program** используется ключевое слово **library**, что позволяет компилятору однозначно идентифицировать тип проекта как динамически загружаемую библиотеку.

Экспорт из DLL обеспечивается при помощи ключевого слова **exports**, вслед за которым следует перечисление необходимых идентификаторов (функций и переменных). В данном случае мы, как и прежде, экспортируем функцию **getSum**.

Замечание:

Если используется экспорт функций из DLL, которые каким-то образом связаны с объектами типа **string**, необходимо использовать модуль **ShareMem**. Подробнее об этом можно почитать в разделе **«Shared-memory manager (Windows only)»** справочной службы **Delphi**.

Как и прежде (несмотря на то, что мы создали DLL посредством продукта компании **Borland**, а не **Microsoft**), при помощи утилиты **dumpbin** можно убедиться в правильности создания DLL и наличии требуемых экспортируемых идентификаторов:

```
Microsoft (R) COFF/PE Dumper Version 7.00.9466
Copyright (C) Microsoft Corporation. All rights
reserved.

Dump of file XDll.dll

File Type: DLL

Section contains the following exports for XDll.dll

 00000000 characteristics
        0 time date stamp Thu Jan 01 03:00:00 1970
     0.00 version
        1 ordinal base
        1 number of functions
        1 number of names

  ordinal hint  RVA      name
        1   0   00001F2C getSum

Summary
     1000 .edata
     1000 .idata
     1000 .reloc
     1000 .rsrc
     1000 BSS
     1000 CODE
     1000 DATA
```

Обратите внимание на небольшое различие в списке секций DLL в случае создания ее компилятором **VC++** и **Delphi** (раздел **Summary**). Несмотря на отличие в названиях секций, в большинстве своем они призваны выполнять одну и ту же задачу. На-

пример, соответствующие разделы **.text** и **.code** содержат исполняемый код DLL.

Дополнительную информацию о секциях DLL можно получить при помощи **dumpbin** с ключом **/headers**.

Используя расширенный синтаксис, можно назначить совершенно другое экспортируемое имя функции, а также указать для нее порядковое число (вспоминаете поле **«ordinal»** в выводе **dumpbin**?).

Предположим, нам необходимо создать дополнительный псевдоним для нашей функции **getSum** с явным указанием ее порядкового номера. Тогда раздел **exports** необходимо переписать в следующем виде:

```
exports
   getSum index 3 name 'myGetSum',
   getSum;
```

Вывод **dumpbin** в этом случае претерпит небольшие изменения:

```
ordinal hint  RVA        name
   1      0   00001F2C   getSum
   3      1   00001F2C   myGetSum
```

В секции **begin ... end** можно определить код, который будет выполняться при проецировании DLL на адресное пространство вызывающего процесса.

Microsoft Visual Basic 6.0

Visual Basic 6.0, как и многие другие системы программирования, позволяет использовать механизм DLL. При этом можно:

- создавать полноценные динамически загружаемые библиотеки функций средствами VB;
- использовать вызовы внешних DLL из вашего приложения;
- использовать готовые библиотеки **Win32 API**.

Многие разработчики считают VB ненадежной «игрушкой», которая не имеет эффективных средств для программирования полноценных приложений. Конечно, иногда это является правдой, но только не по отношению к созданию динамических библиотек. С помощью VB можно быстро и эффективно скомпоновать библиотеку. Но хватит рекламы! Давайте поговорим конкретно о теме нашего разговора.

Прежде всего, хотелось бы заметить, что VB не является полноценным объектно-ориентированным языком, но все же он поддерживает классы. При создании проекта типа **«ActiveX DLL»** создается класс, который имеет новое свойство (у обычного класса его нет) — **«Instancing»**. Этот параметр по умолчанию установлен в значение 5 (**«Multiuse»**). Не забывайте, что одна DLL может содержать в себе несколько объектов (классов). Приведем описание часто используемых параметров свойства **Instancing** (табл. 2).

Мы настоятельно рекомендуем самостоятельно разобрать каждый их этих пунктов, так как это пригодится в будущем. Конечно, обычно создаются классы именно со значением этого свойства **«MultiUse»**.

Таблица 2

Часто используемые параметры свойства Instancing

Значение	Имя	Описание
1	**Private**	Класс доступен только в данном проекте
2	**PublicNotCreatable**	Можно обращаться к экземплярам класса только после их создания другим приложением
3	**SingleUse**	Каждое приложение связывается с новой копией класса
5	**MultiUse**	Каждое приложение связано с единственной копией класса

Стоит подробнее остановиться на установке этого параметра в **«SingleUse»**. Если проект имеет несколько **SingleUse**-классов — становится трудно предсказать поведение динамической системы. Если несколько (например, три) приложений состыковываются с разными компонентами (классами) из DLL — происходит неразбериха. Иногда динамическая система может связать два класса первой «копии» библиотеки с первым приложением, а для третьего приложения-клиента запустить отдельную «копию» DLL. Хотя краха системы может и не быть, зато точно получится не то, что ожидалось! Поэтому разумнее оставить этот параметр в покое (по умолчанию).

Если класс является публичным (не приватным, а, например, **MultiUse**), доступным становится еще одно свойство — **«Persistable»** (дословно — «сохраняемый»). Если данное свойство установлено в **TRUE** — можно использовать методы **ReadProperties/WriteProperties/InitProperties**. До 6-й версии VB программисту необходимо было самому «реализовывать» (имеется в виду не полноценная ООП-реализация) некоторые классы, теперь же VB сам автоматически имплементирует классы **IPersistStream/IPersist** (что и является базисами технологии COM). Для чего же все это нужно? Например, необходимо извлекать кусочки информации и работать с полями класса как с БД. Именно в таких случаях, например, и вызывается метод **WriteProperties**. Когда несколько приложений обмениваются данными с одной DLL, свойство **Persistable** должно быть установлено в «1». Поэтому мы рекомендуем всегда ставить этот параметр в «1», правда это может привести к крахам системы и взломам. Но поскольку в данный момент мы не заботимся о безопасности — смело прислушайтесь к этому совету. В сам проект, как и обычно, можно добавлять модули и формы. По сути, библиотечный проект практически ничем не отличается от обычного, за исключением небольших изменений.

Сам по себе VB является системой без контроля потоков (т. е. разработчик вообще не имеет дела с потоками), что несколько упрощает программирование, но ограничивает в средствах. Несмотря на это, при создании DLL-проекта появляется возможность указать в диалоговом окне **«Project Properties»** потоковую модель проекта. Имеется лишь два варианта выбора — **«Apartment model»** и **«Single-thread model»**. Давайте подробнее рассмотрим каждый из этих пунктов:

1. Апартаментная модель — каждый объект находится в своем индивидуальном потоке, но имеет «глобальную» информацию. Глобальная информация — это, например, **Public**-переменные, поэтому тщательно ограничивайте их для сохранения памяти.
2. Однопоточная модель — все объекты находятся в одном потоке.

Не путайте потоковую модель и свойство **«Instancing»**. Это различные понятия. Вообще, такое свойство введено потому, что DLL может быть связана с приложениями, написанными на других языках. Как известно, такие программы могут быть многопоточными клиентами DLL, поэтому, если многопоточное приложение связывается с однопоточной библиотекой, все «общение» происходит через один главный поток этого приложения. Использование однопоточной библиотеки в этом случае безопаснее, но намного медленнее.

Использование апартаментной модели дает такую выгоду:

- запросы к объектам из разных потоков клиента (приложения) к DLL автоматически синхронизируются;
- блокировка потоков (автоматическая) обеспечивает надежность и безопасность;
- достигается эффективная работа такой системы за счет соответствия потокам DLL потоков EXE.

На вкладке **«General»** диалогового окна **«Project Properties»** (там, где мы указывали потоковую модель) присутствует флажок

«Unattended Execution». Дело в том, что бывают ситуации, когда работу с DLL нужно «скрыть от глаз пользователя» — например, не показывать сообщения об ошибках. Часто это необходимо при работе на сервере или на выделенной машине. Именно в этих случаях и нужно ставить данный флажок: он означает, что на экран не будет выводиться какая-либо информация (диалоговое окно, сообщение об ошибке и проч.). Такие DLL часто выполняют служебные функции по обработке данных и т. д. В случае возникновения непредвиденной ситуации она будет зафиксирована в журнале (лог-файле).

Каждый класс, помещаемый в DLL, которая разрабатывается на **Visual Basic**, имеет два стандартных события (функции, если использовать терминологию языка C) — **Initialize()** и **Terminate()**. Первое вызывается при инициализации класса библиотеки, второе — при его деинициализации.

Блок инициализации — это своеобразный аналог exe-процедуры **main()/DllMain()**. Весь основной код может содержаться именно здесь (хотя никто не запрещает использовать функции и процедуры, другие вызовы и т. д. Поэтому иногда полезно создавать библиотеки, а не модули функций). Код, содержащийся в этом блоке, будет вызван позже оператором **set ... new**.

Приведем пример: допустим, мы имеем DLL, которая просто выводит готовую форму на экран (соответственно, имеем форму и вызывающий ее код). Вот этот код:

```
Private Sub Class_Initialize()
   Form1.Show
End Sub
```

Осталось просто скомпилировать DLL в какой-нибудь каталог. Перед компиляцией не забудьте полностью указать свойства проекта (вкладка **«MAKE»** — версия библиотеки, ее имя и т. д.) и протестировать вашу библиотеку. При компиляции VB создает 3 файла с расширениями **.dll/.lib/.exp**. Первый файл и является нужной нам динамической библиотекой.

Осталось только вызвать нашу динамическую библиотеку из внешнего приложения, что мы и сделаем несколько позже.

Какой компилятор выбрать для создания своих DLL?

Ответ: тот, который наиболее подходит для конкретных задач. При компиляции EXE данный ответ следовало бы дать в таком вот виде: «...и тот, который может дать достаточный уровень производительности». Для DLL потребуется еще одно уточнение: и тот, который поддерживает импорт/экспорт с заданными условиями. **Visual C++ 6.0/7.0**, **Builder C++**, **Delphi**, **MASM**, **Visual Basic** — все они поддерживают создание и/или использование DLL в той или иной степени.

Вообще, процесс построения и реализации проекта может носить следующий характер:

- наиболее критичные по времени, по скорости работы модули выполняются в виде, удобном для всестороннего использования (посредством DLL, COM и проч.). Возможно, при этом будут использованы различного рода оптимизирующие методы (в том числе построение кода на низкоуровневых машинозависимых языках системы — например, **assembler**). Вполне вероятно, что сам процесс построения исполняемого кода может носить «консольный» характер — работа с компилятором в командной строке и проч.;
- графическая подсистема реализуется на любом удобном для данной предметной области языке программирования и трансляторе, который обеспечен требуемыми характеристиками поддержки подсистемы разработки графического интерфейса пользователя (**GUI**, **graphical user interface**) — **Visual Basic**, **MatLab** и проч. При этом среда построения GUI может не предоставлять средства генерации высокоэффективного исполняемого кода. Даже, скорее всего, эта среда будет работать интерпретирующим образом — в таких системах упор делается на удобство работы пользователя

и скорость построения интерфейса, а не на быстроту исполнения (например, такой подход проповедуется в среде **MatLab**). Как правило, такие среды обеспечивают богатый высокоуровневый API для построения мощной системы визуализации результатов. Кроме того, они же обеспечивают определенный интерфейс для взаимодействия с кодом, написанным на других языках программирования, — чтобы обеспечить дополнительную эффективность исполнения. А также умеют преобразовывать интерпретируемый скрипт в скомпилированный и готовый к исполнению код, который работает в десятки раз быстрее.

Замечание:

По предыдущему опыту можно сказать, что однажды даже пришлось осуществлять взаимодействие реализаций исполняемого кода сразу в трех различных средах — **Builder C++**, **Visual C++** и **Delphi**. **Builder C++** использовался для быстрого и удобного построения GUI интерфейса. Использование **Delphi** было продиктовано существованием готовой системы генерации кода на языке **Object Pascal**. А код, написанный на **Visual C++**, осуществлял взаимодействие между GUI-интерфейсом и библиотекой динамической загрузки, скомпилированной **Delphi**, — упор на VC++ был сделан в связи с наличием удобной системы документации по объектам ядра, а также с достаточно эффективной системой генерации кода.

Как правило, каждый из компиляторов налагает определенные требования на имена объектов, формируемых и понимаемых компилятором.

Замечание:

Здесь и далее термин «объект» употребляется в контексте «сущность», а не «объект класса».

Значит ли это, что нельзя использовать DLL, написанную, скажем, с помощью **Builder C++**, в **Visual C++**? Ответим так: нельзя, если не прибегнуть к различного рода ухищрениями и уловкам.

Среди нюансов можно отметить:

• Декорирование имен... точнее, дедекорирование имен. **«Decorate»** в переводе с английского означает «украшать», «награждать». Этот процесс происходит из-за того, что разные компании используют различные способы декорирования имен при экспорте объектов. Например, VC++ 6.0 добавляет к имени символы '?', '@' и кучу другой маловразумительной информации. Какова же причина, по которой это было сделано? Пока лишь достаточно знать, что декорирование имеет своей целью сохранение дополнительной информации о функции (название класса, типы передаваемых аргументов и возвращаемого значения, стратегия вызова и проч.). При избавлении от декорированных имен не всегда возможно использование неявных средств загрузки (основанных на подключении lib-файла, из которого распаковывается вся необходимая для связывания идентификаторов информация). Типичный выход — использовать явную загрузку. При этом необходимо строго соблюдать внешний (типы и количество передаваемых параметров, порядок их размещения в стеке, тип возвращаемого результата) и внутренний (стратегия освобождения стека) интерфейсы функции.

• Экспорт классов и других сложных типов. Здесь возникают две существенные проблемы (а точнее, три):

1. Описание сложных типов очень сильно зависит от стандарта языка — разумеется, что **class** в **Object Pascal** отличается от своего собрата в C++. Ведь класс — понятие скорее из человеческой жизни, компилятор затем «разбирает» этот класс на множество функций, с которыми, собственно, и происходит работа при исполнении кода. Процессор понимает машинную инструкцию **call**, не заботясь о том, является ли эта функция членом класса или обычной функцией. А вот

компилятор обязан позаботиться о том, чтобы правильно преобразовать вызов метода класса в вызов «обычной» функции (как правило, это осуществляется благодаря добавлению особого скрытого параметра **this**, в других языках называемого также **Self** или **Me**, к аргументам функции). Разумеется, различные компиляторы по-разному выполняют задачу «разбиения» класса на функции. Процесс разбиения будет разобран в разделе «Экспорт и импорт классов и переменных».

2. Предопределенные «стандартные» типы для одной среды и языка вполне резонно могут не являться таковыми для другого. Как известно, тип **string** в **Pascal** совсем не соответствует тому, что используется программистами в C++ (**std::string**). Необходимо ограничить работу со сложными типами данных при их экспорте из DLL.
3. Проблемы управления памятью. Данная тема обсуждается в разделе «Использование STL в DLL». Подобные проблемы могут быть связаны с несоответствием в моделях управления динамической памятью в различных языках (занятия и освобождения) — например, **New()** из **Pascal** совсем не соответствует **delete** из C++. Поэтому в операциях, связанных с динамической памятью, надо проявлять особую осторожность: если есть функция DLL, выделяющая память, значит, должна быть функция, которая «знает», как ее правильно освободить. Более того, проблемы с памятью могут возникнуть даже при соответствии языков — например, в случае различного способа связывания с RTL (**runtime library**) в коде DLL и exe-приложения.

А что касается темы «Выбор компилятора», то на основе всего сказанного следует сделать вывод: выбор компилятора должен осуществляться на основе задач, которые стоят перед конкретным разработчиком, а также условий, которые налагают эти задачи.

Кроме того, каждая среда предоставляет, как правило, различную функциональность для разработчика — эти условия также необходимо учитывать при выборе компилятора. Ведь создание новых языков программирования призвано обеспечить более эффективное решение задач определенного класса.

Как приложение загружает DLL

Как уже известно из предыдущих разделов, модуль DLL может определять два вида функций — внутренних (**internal**) и внешних (**exported**). Экспортируемые функции могут быть вызваны другими модулями (в том числе другими DLL). Внутренние функции могут быть использованы лишь кодом самой DLL.

Динамически линкуемые библиотеки обеспечивают путь для модульного подхода при создании приложений. Кроме того, написанный однажды код может быть в дальнейшем использован другими приложениями. Любые вносимые изменения (не затрагивающие изменения внешнего интерфейса DLL) не будут отражаться на работе использующего данный код приложения.

Для того чтобы приложение могло использовать экспортируемые функции (либо экспортируемые переменные), DLL должна быть отображена (спроецирована, загружена) на адресное пространство использующего ее процесса. Существуют три основных способа проецирования DLL на адресное пространство, которые и будут рассмотрены ниже.

Неявная загрузка

При неявной загрузке DLL проецируется на адресное пространство вызывающего процесса (загружается) при его создании. Если по какой-либо причине неявная загрузка DLL завершается неудачно, загруз-

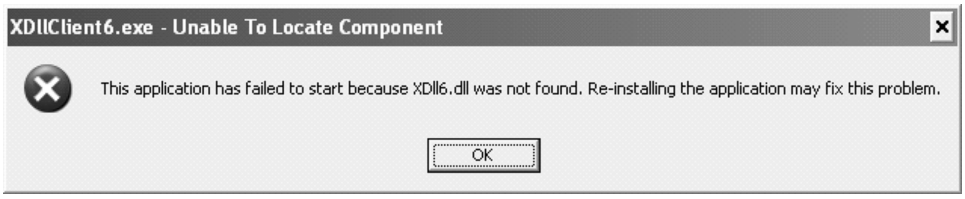

Рис. 3. Сообщение об ошибке для Windows XP

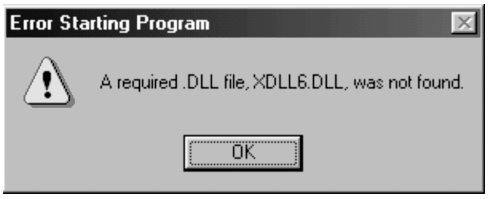

Рис. 4. Сообщение об ошибке для Windows 98

чик операционной системы немедленно прерывает процедуру создания процесса, выводит диалоговое окно для уведомления пользователя о возникшей проблеме и «прибивает» процесс. Сообщения об ошибке при этом для различных версий ОС выглядят немного по-разному (рис. 3 и рис. 4).

В случае неявной загрузки приложению требуются:

- h-файл (**header** — заголовочный) с прототипами функций, описаниями классов и типов, которые используются в приложении;
- lib-файл (**library** — библиотечный), в котором описывается список экспортируемых из DLL функций (переменных) и их смещения, необходимые для правильной настройки вызовов функций.

h-файл (*.h) требуется компилятору, чтобы определить суть (тип, список используемых параметров, их типы и проч.) имеющихся объектов.

lib-файл (*.lib) требуется компоновщику (линкеру) для правильной настройки виртуальных адресов в приложении. Эти адреса являются смещениями относительно некоторого базового адреса, по которому DLL будет загружена в адресное пространство.

В случае совпадения базового адреса и реального, по которому, собственно, DLL и будет размещена в памяти, дополнительная настройка адресов не требуется. Если же «не повезет» (адреса различаются), значит, при загрузке следует произвести модификацию адресов (этим занимается загрузчик) и лишь после этого продолжить выполнение приложения. Очевидно, что подобная модификация адресов отрицательно скажется на скорости загрузки и запуска приложения, использующего данную DLL.

В предыдущем разделе мы создали DLL, в которой реализован уникальный алгоритм сложения двух целых чисел. Теперь создадим приложение, использующее созданную библиотеку при помощи неявной загрузки.

Неявная загрузка позволяет на основе информации, прочитанной из *.lib-файла, связать идентификаторы клиентского приложения и DLL. Программист должен также предоставить компилятору заголовочный файл с определениями функций. Это позволит проверить корректность вызовов функций, а также сообщить, что данные функции должны экспортироваться.

Замечание:

Зачем необходим еще и h-файл, если имеется lib-файл? Дело в том, что lib-файл не содер-

жит информации (декорирование имен — не в счет) относительно аргументов функции: их количестве, типах. Также нельзя узнать и тип возвращаемого значения. А эта информация и содержится в h-файле. Кроме того, указания h-файла заставляют компилятор «понять», какие идентификаторы нужно связывать посредством lib-файла.

Для построения клиентского приложения необходимо проделать следующие шаги:

Замечание:

Здесь и далее мы будем использовать консольные приложения для демонстрации работы с DLL. Это позволит написать минимум кода, не усложняя и не запутывая его без лишней необходимости. Использование DLL в проектах других типов проводится аналогичным образом.

VC++ 6.0

1. Создать проект **«Win32 Console Application»**: в целях эксперимента создадим пустой проект.
2. Добавить в полученный проект файл **main.cpp** (**«File->New->C++ Source File»**).
3. Отредактировать полученный файл **main.cpp**.

VC++ 7.0

1. Создать новый проект **«Win32 Project»** — **XDllClient**.
2. Уточнить тип приложения — **«Console application»**. Для чистоты эксперимента вновь создадим пустое приложение, отметив галочкой **«Empty project»**.
3. На вкладке **«Overview»** можно просмотреть свойства создаваемого проекта.
4. Создаем и добавляем в проект файл **main.cpp**. Текст его приведен выше.

Компилируем... не получается! Компилятор выдает что-то вроде (см. стр. 48).

Дело в том, что подобные ошибки возникают в том случае, когда компоновщик (линкер) не может найти определение функции — именно это и происходит в нашем случае. Но как же мы можем выдать определение функции **getSum**, если у нас его нет и быть не может? Ведь оно спрятано в теле DLL, и извлечь его оттуда не представляется возможным (разве что только обратным дизассемблированием кода).

Что делать? В это время как раз и стоит вспомнить *.lib-файл. Именно он и поможет нам — ведь линкеру требуется определение функции, чтобы выяснить ее виртуаль-

Файл **main.cpp**

```
#include "../XDll6/XDll.h"
/* Этот файл (XDll.h) необходим для того, чтобы компилятор мог:
а) проверить корректность синтаксиса вызова функции (на основе ее определения);
б) узнать, что данные функции (в нашем случае - getSum) импортируются из DLL.
Как уже было сказано при построении DLL, это осуществляется при помощи различного определения символа
XDLL_API (в случае включения его в файлы проекта DLL он определяется как __declspec(dllexport); если же
мы включаем его в файл приложения, то он определяется как __declspec(dllimport) - это регулируется нали-
чием/отсутствием идентификатора XDLL_EXPORTS в настройках проекта DLL - см. "ProjectsSettings->C++->
General->Preprocessor definitions"; в клиентском приложении этот идентификатор не должен определять-
ся!) */

#include <iostream>

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

   std::cout << "getSum(10, 20): "<< res << std::endl;
}
```

```
--------------------------Configuration: XDllClient6 - Win32 Debug-------------------------
Compiling...
main.cpp
Linking...
main.obj : error LNK2001: unresolved external symbol
   "__declspec(dllimport) int __cdecl getSum(int,int)" (__imp_?getSum@@YAHHH@Z)
Debug/XDllClient6.exe : fatal error LNK1120: 1 unresolved externals
Error executing link.exe.
XDllClient6.exe - 2 error(s), 0 warning(s)
```

ный адрес. А *.lib-файл «знает» его — ведь он создавался на основе файла *.dll, в котором этот адрес «зашит» при компиляции.

Замечание:

Несмотря на то что мы имеем lib-файл, тем не менее, там находится не виртуальный адрес, а относительный виртуальный адрес (**RVA**). Полный виртуальный адрес становится известным только после проецирования DLL. Тем не менее информации об относительном виртуальном адресе линкеру вполне достаточно, чтобы достойно завершить сборку приложения. В этом случае «запоминаются» все точки входа, в которых указан **RVA**. При загрузке приложения и проецировании DLL происходит модификация данных точек — к адресу **RVA** прибавляется базовый адрес, по которому загружена DLL.

Для подключения *.lib-файла необходимо проделать следующие шаги.

Для VC++ 6.0:

Найти вкладку **«Link->General->Object/library modules»**.

Для VC++ 7.0:

Найти вкладку **«Linker->Input->Additional Dependencies»**.

Как видно, клиентское приложение использует уже достаточно системных DLL. Ну что ж, не будет лишним добавить здесь и наш *.lib-файл.

Попробуем провести компиляцию проекта теперь. Если к этому времени lib- и dll-файлы скопированы в директорию, которую среда **Visual Studio** «видит» (по умолчанию таковой директорией является главная директория проекта), то компиляция должна пройти успешно.

Замечание:

Альтернативный вариант подключения библиотеки импорта заключается в использовании директивы **#pragma**. В этом случае необходимо добавить примерно такой текст директивы в каком-нибудь cpp-файле:

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

Подробнее об этом можно узнать в разделе «Различные способы экспорта и импорта».

Это предоставляет дополнительные рычаги управления линкером. В данном случае мы просим линкер подключить к списку импортируемых библиотек еще одну — **xdll6.lib**.

В принципе, размещать подобную директиву можно где угодно (и не один раз). Тем не менее будьте внимательны, если работаете с файлами досрочной компиляции (**precompiled headers**) наподобие **stdafx.h**. Директива **#pragma** должна быть расположена после директивы включения (**#include**) такого файла — иначе она просто не будет выполнена.

Несмотря на простоту и доступность подобных дополнительных директив, злоупотреблять их применением все же не стоит. Позабыв однажды о том, что в тексте существует такая директива, можно настроить другие опции проекта таким образом, что они будут вступать в конфликт. Непонимание ситуации может продолжаться достаточно длительное время, потому что текст этих директив никоим образом не отражается в командном окне линкера (**«Link->General->Command Options»** в случае VC++ 6.0 и **«Linker->Command Line»** для VC++ 7.0).

Таким образом, после выполнения всех необходимых действий на экране появится примерно следующий результат:

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

Как видите, мы получили функцию, которая умеет складывать два числа.

VC# 7.0

1. Создайте простое консольное приложение типа **«Console Application»** — **XDllClient7**.

2. Модифицируйте код класса **Class1.cs**.

Как видите, благодаря механизму атрибутов можно легко создать «псевдометод», реализация которого находится в DLL. Вызов такого метода ничем не отличается от обычного (статического) метода класса. Кроме того, передавая в конструктор атрибута дополнительные параметры, можно определить также тип соглашения вызова (**DllImport.CallingConvention**), установить кодовый набор символов (**DllImport.CharSet**) и «переименовать» вызываемую функцию (**DllImport.EntryPoint**).

C++ Builder 6.0

Для демонстрации неявной загрузки DLL создадим простейшее приложение.

1. **«File->New->Other->Console Wizard»**.

2. В появившемся диалоге выбираем язык C++, оставляем выбранным флаг **«Console Application»**.

3. Нажимаем кнопку **«OK»**.

В созданном нами проекте теперь нужно объявить импортируемые из DLL функции. Как и раньше, мы создали DLL и экспортируем из нее функцию **getSum**:

```
int getSum(const int x, const int y);
```

Добавим определение функции в файл проекта (с точки зрения нашего тестового

Код класса **Class1.cs**

```
using System;
/* Platform Invocation Services (PInvoke) позволяют неуправляемому коду быть вызванным из управляемого. Сделано это с той целью, чтобы приложения .Net могли легко и просто пользоваться старым и проверенным кодом. */
using System.Runtime.InteropServices;
namespace XDllClient7
{
class MainClass
{
   static void Main(string[] args)
   {
      /* вызов функции ничем не отличается от обычного вызова статического метода класса */
      int n = getSum(10, 20);
      System.Console.WriteLine("getSum(10, 20): {0}", n);
   }
   /* вызов функции из DLL возможен благодаря наличию атрибута DllImport, который прикрепляется к описанию метода класса(такой метод в обязательном порядке должен иметь модификаторы static и extern!).
   */
   [DllImport("XDll6.dll")]
   static extern int getSum(int n1, int n2);
}
}
```

проекта эта функция будет импортируемой):

```
int __declspec(dllimport) getSum(const int x,
const int y);

int main(int argc, char* argv[])
{
   cout << getSum(10, 20) << endl;

   return 0;
```

Если мы попытаемся скомпилировать такой проект, то неизбежны сообщения об ошибках:

```
[Linker Error] Unresolved external 'getSum(int,
int)' referenced from MAINUNIT.OBJ
```

И действительно — мы ведь забыли указать линковщику соответствующий динамической библиотеке lib-файл! Без него компоновщик не может импортировать соответствующие функции из библиотеки DLL.

Для того чтобы компоновщик «увидел» DLL, нужно добавить соответствующий lib-файл в проект (**«Project->Add to Project...»**). Пробуем собрать проект еще раз — теперь все работает!

Замечание:

Очень удобно было бы иметь общий h-файл с описаниями функций для самой библиотеки DLL и использующего ее приложения. Для этого нужно в самом начале h-файла объявить **define**-конструкцию вида:

```
#if defined(_DLLEXPORT)
#define DLL_SPEC __declspec(dllexport)
#else
#define DLL_SPEC __declspec(dllimport)
#endif
```

После этого все функции в h-файле следует описывать следующим образом:

```
int DLL_SPEC getSum(const int, const int);
```

Обратите внимание на идентификатор **DLL_SPEC**.

Соответственно, для DLL-проекта следует определить идентификатор **_DLLEXPORT**:

```
#define _DLLEXPORT
```

Тогда все функции будут экспортируемыми.

В проекте, использующем DLL, такой директивы не будет, поэтому все функции будут объявлены импортируемыми.

Delphi 6.0

Создадим клиентское приложение и здесь. Для этого проделаем следующие шаги:

1. **«File->New->Other->Console Application»**.

2. Помощник создаст необходимые текстовые файлы для компиляции консольного проекта.

Придадим необходимый вид главному модулю:

```
program XDllClient;

{$APPTYPE CONSOLE}

// неявная загрузка
function getSum(const n1,n2: integer): integer;
external 'XDll.dll';

var
   n: integer;

begin
   // осуществляем вызов функции из DLL
   n := getSum(10, 20);

   WriteLn('n = ', n);

   WriteLn;
   WriteLn('Press any key...');
   ReadLn;
end.
```

Delphi для импортирования функций предоставляет ключевое слово **«external»**, после которого требуется указать название файла DLL, который содержит необходимый идентификатор. В этом случае при компиляции проекта автоматически будут установлены все связи, необходимые для вызова данной функции.

Замечание:

Будьте внимательны при импортировании функций. **Object Pascal** предполагает полное равнодушие к регистру используемых симво-

лов, но... только не в этом случае! При объявлении функции необходимо строго соблюдать регистр используемых символов (для правильного нахождения указанной функции в заголовке файла DLL). В программе уже можно спокойно использовать это имя, вновь не заботясь о регистре символов.

Еще одним удобным механизмом работы с DLL неявной загрузки является использование дополнительного модуля (**unit**), в интерфейсную часть которого выносятся все необходимые объявления.

Например, это можно сделать так — добавим в проект новый модуль **XDllFuncUnit**:

```
unit XDllFuncUnit;

interface
   // объявление функции в интерфейсе модуля
   function getSum(const n1,n2: integer): integer;

implementation
   // реализация находится в DLL
   function getSum(const n1,n2: integer): integer;
   external 'XDll.dll';

end.
```

Замечание:

Требование к регистру символов, упомянутое выше, в этом случае относится к именованию функции в интерфейсной части модуля. В секции реализации (**implementation**) такая функция может быть названа с игнорированием регистров символов.

Тогда главное приложение претерпит небольшие изменения:

```
program XDllClient;

uses
   XDllFuncUnit in 'XDllFuncUnit.pas';

begin
   // осуществляем вызов функции из DLL
   n := getSum(10, 20);
   ...
end.
```

В этом случае можно совершенно не заботиться о том, где именно размещена функция **getSum**. Любые изменения коснутся лишь модуля, в котором объявлена эта функция (в данном случае **XDllFuncUnit**). Особенно удобно использовать такой принцип, когда из DLL импортируются сразу несколько функций. Именно такой подход используется в самой среде **Delphi**. Убедиться в этом можно, если внимательно изучить, например, файл **Windows.pas**:

```
...
function LoadLibrary; external kernel32 name
'LoadLibraryA';
function LoadLibraryA; external kernel32 name
'LoadLibraryA';
function LoadLibraryW; external kernel32 name
'LoadLibraryW';
...
```

Синтаксис **external** предполагает возможность определения альтернативного имени для импортированной функции в клиентском приложении. Приведем классический пример. Как известно, системная библиотека **user32.dll** (и не только она) экспортирует два идентификатора для каждой функции, которая работает с символами, — ANSI-версию (суффикс «A») и ее UNICODE-аналог (обычно с добавлением суффикса «W»).

Так вот, в случае когда мы хотим использовать одну из двух указанных функций (на примере **MessageBoxA** и **MessageBoxW**), нам необходимо написать что-то вроде (используем UNICODE-аналог):

```
function MessageBox(HWnd: Integer; Text,Caption:
PChar; Flags: Integer):
   Integer; stdcall; external 'user32.dll' name
   'MessageBoxW';
```

Если мы хотим использовать **MessageBoxA**, достаточно подправить лишь одну строчку в нашей программе, не изменяя ничего другого (ключевое слово **stdcall** определяет так называемые **«calling conventions»** — соглашения вызова функции. Подробнее об этом рассказано в разделе «Декорирование имен»):

```
function MessageBox(HWnd: Integer; Text, Caption:
PChar; Flags: Integer):
   Integer; stdcall; external 'user32.dll' name
   'MessageBoxA';
```

В определенных случаях такая возможность бывает очень даже полезной, а использование директив условной компиляции (**{$DEFINE}-{$IFDEF}-{$ELSE}-{$ENDIF}**) позволяет унифицировать этот процесс.

И, конечно же, в результате исполнения данного приложения получается уже порядком поднадоевший результат **«n = 30»**.

Visual Basic

VB полностью поддерживает конвенцию вызова функций DLL (**Microsoft StdCall**), поэтому эта система может использовать динамические библиотеки на все сто процентов. Существует всего два способа вызвать нужную вам функцию:

- прототайпинг (описание заголовка функции);
- подключение библиотеки как класса **ActiveX DLL**.

В первом случае необходимо просто описать заголовок функции (полезно для этих целей создать отдельный модуль, в котором и будут описаны все вызовы, которых может быть больше, чем самого текста программы). Именно благодаря такому объявлению функции VB «поймет», каким образом передаются и принимаются параметры.

Описание заголовка вызываемой функции состоит из нескольких частей. Приведем и рассмотрим пример для нашей любимой функции **getSum**:

```
Option Explicit

Public Declare Function getSum Lib "XDll6.dll" ( _
   ByVal x As Integer, ByVal y As Integer) As Integer
```

Замечание:

Обратите внимание на '_' в конце первой строки декларации — в VB это необходимо для всех многострочных операторов!

Прежде всего, здесь используется зарезервированное слово **DECLARE** (что означает «объявить»), которое и подсказывает компилятору о необходимости подключения внешней библиотеки. За этим словом идут стандартные слова **Function** | **Sub** и лексема **LIB**. Строка, которая следует за **LIB**, должна содержать имя DLL (при регистрации и создании ссылки) или непосредственно путь к DLL.

С точки зрения написанного кода вызов такой функции ничем не отличается от обычного вызова:

```
Public Sub TestFunction()
   Print getSum(10, 20)
End Sub
```

И вновь мы получим все тот же результат — 30.

Как видите, ничего сложного здесь нет. Теперь давайте рассмотрим пример объявления функции **WinApi**:

```
Public Declare Function FW Lib "user32" Alias
"FindWindowA" ( _
   ByVal lpClassName As String, ByVal lpWindowName
   As String) As Long
```

В этом примере вызывается функция **WinApi**, поэтому можно использовать не полное имя DLL, а лишь ее «название» в операционной системе.

Обратите внимание на то, что имя функции используется исключительно внутри конкретной программы и никак не связано с именем реальной функции. Настоящую связь имеет «псевдоним», следующий за словом **ALIAS**. Поэтому важно проследить, чтобы заголовок внутренней функции походил на заголовок реальной функции (идентификаторы параметров функции не должны совпадать с реальными идентификаторами, главное — совпадение их типов). Хотелось бы также заметить, что при указании доступа **Public** все вызовы должны быть описаны в отдельном модуле.

Использовать такую функцию можно как обычно. Вот пример:

```
hwndCurrent = FW(vbNullString, sAppName)
```

Для того чтобы согласовать входные-выходные параметры, полезно использовать утилиту **API Text Viewer** (поставляется с **Visual Studio**, например). Там уже есть готовые заголовки функций, и при желании можно всегда быстро найти то, что нужно. Еще можно смело рекомендовать очень полезную утилиту — **Api-guide** (зайдите на http://www.mentalis.org). Не забывайте, что иногда (особенно в ОС с ядром **WinNT**) имя функции в конкретной системной DLL может при вызове заменяться ее порядковым номером. Хотя такой способ и применяется (в первую очередь, из-за небольшого ускорения процесса вызова), он нецелесообразен, так как порядковый номер функции может и измениться.

Существуют и «подводные камни», основным из которых является то, что VB использует свои типы строк (названные **BasicStrings** — **BSTR**), которые могут конфликтовать с входными параметрами, являющихся обычно стандартными **LPSTR**. Для уверенной работы вызова используйте переданные по значению строки, длина которых не больше 255. В некоторых случаях рекомендуется явно указывать длину строки при ее создании:

```
Dim s_in as String * 255
```

Всегда старайтесь передавать строку в функцию по значению (**ByVal**). Хотя строки всегда передаются по ссылке, «передача по значению» отличается вот чем: VB автоматически преобразует строку в указатель с завершающим **null**-символом; если же лексема **ByVal** отсутствует, строка представляется как обычная **BSTR** (в DLL, написанных на VB, эти строки работают нормально). Иногда строки, отформатированные программой, содержат внутри себя «лишние» **null**-символы. В таком случае произойдет простое усечение строки до первого «символа конца», что не вызовет исключения.

В любом случае обращайтесь к документации по DLL (хотя таковая часто отсутствует), если возникает необходимость использовать строки.

Второй проблемой может стать передача массивов в API-функции. Обычно библиотеки, написанные для совместимости со всеми системами разработки, используют **OLE Automation**. В таком случае ошибок может и не быть. Проблема же заключается вот в чем: VB использует **SAFEARRAY**, поэтому и функция DLL должна поддерживать такой стандарт. В любом случае выходом может стать вот такой «трюк»: достаточно передать лишь первый элемент массива (необходимо иметь массив из чисел). Всегда смотрите внимательно на все эти константы, которые, несомненно, «прилагаются» к функции API — к примеру, не запоминать же каждый раз шестнадцатеричное число. При передаче параметров в функцию используйте именно эти константы.

Случается, что в одной библиотеке могут присутствовать несколько версий одной и той же функции. Возможно, что одна из них будет предназначаться для 16-разрядной ОС, другая может служить в 32-разрядной среде. Также при работе со строковыми функциями присутствует несколько функций для разных кодировок (**ANSI/Unicode**).

Некоторые советы при вызове функций из DLL в VB:

- Всегда убеждайтесь в работоспособности функции перед ее вызовом.
- Следите за совпадением типов параметров и возвращаемого значения.
- Будьте начеку при передаче данных по ссылке, так как функция может испортить данные в памяти.
- Можно указывать «зону видимости» объявлений, используя ключевые слова **Public** или **Private** перед **Declare**.

В случае использования **ActiveX DLL** не забывайте:

• перед использованием ее необходимо зарегистрировать с помощью утилиты **regsvr32** (регистратор **ActiveX**).

Выводы

Неявное связывание — это очень удобный способ использования DLL, при котором необходимо сделать очень мало дополнительных действий, связанных с ее подключением и использованием.

При этом подчеркнем тот факт, что утилита **dumpbin** с ключом **/exports** выдала нам странные результаты (в случае использования VC++) вместо нормального имени функции, напоминающие что-то вроде **?getSum@@YAHHH@Z** (связано это с так называемым декорированием имен). Тем не менее мы смогли легко и просто использовать эту функцию без лишних хлопот, благодаря поддержке неявной загрузки со стороны компилятора — так как среда и язык разработки для DLL и нашего клиентского приложения совпадали, то вызов в коде функции **getSum** привел к генерации правильного адреса функции **getSum@@YAHHH@Z**. Также неявное связывание очень часто бывает простым способом экспортировать из DLL классы. Но об этом уже следует почитать в разделе «Экспортирование переменных и классов из DLL».

Из минусов данного подхода можно отметить однозначную загрузку DLL на момент старта приложения. Мы не можем никаким образом повлиять на этот процесс. А если DLL нам нужна лишь на некоторое время работы? Зачем тратить место в ОП, занимая его попусту? Ведь DLL неявной загрузки выгружается из памяти лишь при завершении работы всего приложения. Кроме того, если наше приложение использует много DLL, то процесс их инициализации может отнять слишком много времени.

Явная загрузка

Этот способ связан с явным использованием основных функций **Windows API** из предложенного набора. В случае использования явной загрузки программист берет на себя львиную долю забот при работе с DLL.

Ниже перечислен наиболее часто используемый набор предоставляемого **WinApi** для работы с DLL явной загрузки:

• **DisableThreadLibraryCalls** — функция, «запрещающая» получать DLL уведомления **DLL_THREAD_ATTACH** и **DLL_THREAD_DETACH** (см. раздел «Зачем нужна функция DllMain?»). Это бывает полезно в многопоточных приложениях, когда постоянно создаются и уничтожаются рабочие потоки, а DLL не требует получения подобных уведомлений. В целях оптимизации исполняемого кода обычно вызывается в ответ на сообщение **DLL_PROCESS_ATTACH**.

• **FreeLibrary** — функция, используемая для явной выгрузки DLL из ОП. Используется для DLL, которая была перед этим загружена при помощи вызова **LoadLibrary**.

• **FreeLibraryAndExitThread** — функция, позволяющая потоку, созданному в коде DLL, быть безопасно уничтоженным (с последующей выгрузкой DLL).

• **GetModuleFileName[Ex]** — позволяют получить полный путь к конкретному модулю с идентификатором **HMODULE**.

• **GetModuleHandle[Ex]** — позволяют получить идентификатор **HMODULE** по имени модуля. Функция возвращает корректное значение **HMODULE** только для тех модулей, которые были спроецированы на адресное пространство вызывающего процесса.

• **GetProcAddress** — функция, позволяющая получить виртуальный адрес экспортируемой из DLL функции (или переменной) для ее последующего вызова.

• **LoadLibrary[Ex]** — позволяют спроецировать DLL на адресное пространство вызывающего процесса.

Основная нагрузка в этом случае ложится на функции **LoadLibrary**, **LoadLibraryEx**, **FreeLibrary** и **GetProcAddress**. Рассмотрим назначение основных функций.

Функции LoadLibrary и LoadLibraryEx

Прототипы данных функций:

```
HMODULE LoadLibrary(
   LPCTSTR lpFileName.// имя файла
);

HMODULE LoadLibraryEx(
   LPCTSTR lpFileName,// имя файла
   HANDLE hFile,      // зарезервировано, должно
                      // быть NULL
   DWORD dwFlags      // дополнительные параметры
);
```

Данная функция предназначена для возможности проецирования указанной DLL (в качестве параметра принимается название файла DLL с расширением и, возможно, относительный/абсолютный путь к нему) на адресное пространство вызывающего процесса. В качестве имени файла также может использоваться имя не только dll-файла, но и, скажем, exe-файла. Это может быть полезным в том случае, если необходимо использовать ресурсы исполняемого файла — например, при помощи функций **FindResource/LoadResource**.

В случае успешности загрузки количество клиентов данной DLL увеличивается на единицу, и пользователю возвращается значение **HMODULE** загруженного модуля. Данный дескриптор может быть в дальнейшем использован в качестве параметра при использовании функций **GetProcAddress** и **FreeLibrary**.

Возврат значения **NULL** свидетельствует о невозможности загрузки. Информацию об ошибке можно получить, используя функцию **GetLastError**. Дополнительные подробности можно посмотреть в стандартной справочной службе **MSDN** компании **Microsoft**.

Отметим несколько моментов:

- Небезопасно использовать **LoadLibrary** в функции **DllMain** (как вы помните, это функция инициализации DLL — см. раздел «Зачем нужна функция DllMain?»). Дело в том, что в момент инициализации данной DLL другие динамические библиотеки могут быть еще не спроецированы на адресное пространство процесса — и это может привести к взаимной блокировке. Будьте внимательны!
- Получаемые дескрипторы не являются глобальными или наследуемыми. Каждый процесс должен самостоятельно вызывать **LoadLibrary**.
- При указании имени DLL без указания пути используется определенный алгоритм поиска.
- Название имени файла DLL должно являться ANSI-строкой. Как уже упоминалось ранее, в секции экспорта DLL все имена экспортируемых функций сохраняются в виде ANSI-строки. Убедиться в этом можно, если внимательно приглядеться к типу второго параметра функции **GetProcAddress** (см. пример ниже).
- В случае отсутствия расширения в имени DLL подразумевается значение «.dll». Чтобы указать, что имя файла не имеет расширения, используйте ограничивающий символ '**.**', завершающий имя DLL.
- Очень полезным (в целях оптимизации или каких-либо других) может быть использование флагов в функции **LoadLibraryEx**. Например, значение флага **DONT_RESOLVE_DLL_REFERENCES** заставляет ОС не вызывать функцию **DllMain** с различного рода уведомлениями. Флаг **LOAD_LIBRARY_AS_DATAFILE** позволяет загрузить DLL как обычный файл, не подготавливая к последующему выполнению кода в нем. Это бывает полезно в случае распаковки ресурсов из DLL. И, наконец, флаг **LOAD_WITH_ALTERED_SEARCH_PATH** заставляет использовать альтернативный путь поиска при загрузке DLL.

Функция GetProcAddress

```
FARPROC GetProcAddress(
  HMODULE hModule, //HMODULE спроецированной
                   //DLL
  LPCSTR lpProcName//название функции в формате
                   //те ANSI или наименование
                   //переменной.
```

```
                    //Также вместо названия
                    //функции может быть указан
                    //ее порядковый номер
);
```

Позволяет получить по имени функции необходимый виртуальный адрес для работы с ней. В случае невозможности получения адреса функция возвращает значение **NULL**.

Таким образом, функция **GetProcAddress** — это второй необходимый шаг, который совершается для вызова функции из DLL в случае использования механизма явной загрузки. Далее мы рассмотрим пример кода, чтобы окончательно понять, как это происходит. Несмотря на то что наиболее часто функция **GetProcAddress** используется для получения виртуального адреса какой-либо функции, тем не менее, она же используется и для получения адреса какой-либо переменной, экспортируемой из DLL. В разделе «Экспорт классов и переменных из DLL» мы рассмотрим, как можно осуществить данный процесс на практике.

Функция FreeLibrary

```
BOOL FreeLibrary(
   HMODULE hModule // HMODULE спроецированной
                   // ранее DLL при помощи вы-
                   // зова LoadLibrary
);
```

FreeLibrary вызывается на заключительном этапе работы с DLL. При этом происходит уменьшение счетчика клиентов данной DLL. В случае равенства его нулю DLL немедленно выгружается из памяти.

Таким образом, процесс работы с DLL в случае явной загрузки состоит из трех этапов:

1. Загрузка DLL посредством вызова функции **LoadLibrary**. В результате обращения вызывающий процесс получает доступ к описателю загружаемой DLL (**HMODULE**), что позволяет обращаться к этой DLL в дальнейшем (до момента ее выгрузки — в случае последующей загрузки ей может быть присвоен совершенно другой дескриптор).

2. Вызовы функции **GetProcAddress** для получения указателей на требуемые объекты. Особо отметим тот факт, что одна и та же функция используется как при работе с функциями, так и с переменными. Получение виртуального адреса функции может происходить как при помощи ANSI-строки, так и при помощи ее порядкового номера (но делать этого не рекомендуется — см. замечания ниже).

3. Вызов функции **FreeLibrary** после завершения всех требуемых действий с объектами данной библиотеки. В результате этого освобождается место в ОП, проводятся действия по деинициализации DLL. Разумеется, если вы забудете вызвать данную функцию при завершении приложения, за вас это сделает система. Но лучше всегда все делать самому! Этим самым вы в дальнейшем сможете избежать лишних неприятностей. Также полезно проверять значение, возвращенное функцией **FreeLibrary** — функция вернет **FALSE**, если закрытие описателя **HMODULE** невозможно. Наиболее вероятной причиной может являться то, что дескриптор уже был по ошибке закрыт где-либо в другом месте программы.

Итак, для закрепления материала данного раздела модифицируем наше клиентское приложение.

VC++ 6.0/7.0

Теперь файл **main.cpp** должен иметь примерно следующий вид (см. стр. 57).

Замечание:

Не забудьте, что количество вызовов функции **FreeLibrary** должно точно соответствовать количеству вызовов **LoadLibrary**!

Важно не забыть убрать из списка lib-файлов файл **xdll6.lib**. Нет, конечно, ничего плохого не случится, но явная загрузка предполагает, что подобного файла у нас нет (или он нам по каким-либо причинам

```
/* Теперь этот файл (XDll.h) нам не требуется - достаточно знать сигнатуру функции для последующего ее
правильного определения при помощи typedef. Ведь если мы неправильно определим указатель на функцию, то
последующий вызов по этому указателю приведет к краху приложения в связи с нарушением доступа. Это ведь
достаточно веская причина, чтобы не ошибиться? */
//#include "../XDll6/XDll.h"

#include <iostream>
#include <CRTDBG.H>
#include <WINDOWS.H>

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);

   std::cout << "pGetSum(10, 20): " << res << std::endl;

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

   return 0;
}
```

просто-напросто не подходит). Убрали? Теперь запускаем и...

Наша программа замечательно «рухнула» на втором проверочном условии. Почему? Вроде бы все сделали правильно. А вот теперь пора бы и вспомнить вновь о том, что именно выдавала нам утилита **dumpbin** относительно наименования нашей функции **getSum**:

```
Ordinal hint  RVA         name
   1      0   00001005    ?getSum@@YAHHH@Z
```

dumpbin утверждала, что она имеет название в виде **«?getSum@@YAHHH@Z»**. Пока именно его нам и придется использовать в функции **LoadLibrary** — ведь на самом деле сейчас из DLL не экспортируется никакая другая функция, кроме этой (как превратить это имя во что-то более удобочитаемое, мы узнаем чуть позже).

Итак, немного перепишем часть кода:

```
...
/* пытаемся получить адрес функции getSum */
PGetSum pGetSum = (PGetSum)GetProcAddress(hModule,
"?getSum@@YAHHH@Z");
/* проверяем успешность получения адреса */
_ASSERT(pGetSum != NULL);
...
```

Ну что ж, не будем останавливаться на достигнутом. Ранее утверждалось, что функцию можно вызвать и по ее порядковому номеру (хотя, повторимся, делать это все же не рекомендуется). Дело в том, что в случае связывания по имени функция **GetProcAddress** всегда гарантированно вернет **NULL**, если запрошенная функция вообще отсутствует в используемой DLL. В случае же использования связывания по порядковому номеру такой гарантии нет — функция **GetProcAddress** может вернуть ненулевое, однако неверное значение адреса, использование которого немедленно приведет к краху приложения.

Еще раз перепишем эту же часть кода:

```
...
/* пытаемся получить адрес функции getSum */
PGetSum pGetSum = (PGetSum)GetProcAddress(hModule,
MAKEINTRESOURCE(1));
/* проверяем успешность получения адреса */
_ASSERT(pGetSum != NULL);
...
```

Несмотря на то что и этот вариант вполне работоспособен, еще раз повторим, что делать этого ни в коем случае не рекомендуется — изменение порядкового номера (например, в случае добавления в DLL новых экспортируемых идентификаторов) приведет к немедленному краху приложения.

В заключение этого подраздела зададимся двумя вопросами:

а) Что будет, если все-таки не вызывать **FreeLibrary**?

б) Что будет, если вызвать **LoadLibrary** два и более раз?

Ответы:

а) Ничего плохого не случится. Система при завершении приложения сама освободит все используемые ресурсы (в том числе выгрузит из памяти все неиспользуемые DLL — конечно, в том случае, если счетчик ссылок на DLL со стороны внешних пользователей достигнет своего нулевого значения);

б) В этом случае счетчик ссылок увеличится на такое же число. Для завершения (выгрузки) DLL потребуется аналогичное число раз вызвать **FreeLibrary**.

Ну что ж, на этом этапе уже понятны те плюсы и минусы, которые дает данный способ работы с DLL:

(+) явное управление процессом жизни DLL позволяет достичь определенной гибкости в вопросах, связанных со временем жизни DLL в ОП;

(–) перекладывание большей части работы по управлению DLL на программиста;

(+) возможность кратковременной загрузки и последующей выгрузки DLL в/из ОП.

Да, и последнее. В случае когда DLL не может быть найдена, функция **LoadLibrary** вернет значение **NULL** (также вы получите значение **NULL**, если вызываемая функция **DllMain** вернет значение **FALSE** в ответ на уведомление **DLL_PROCESS_ATTACH**). Но, тем не менее, работа приложения при этом может быть продолжена. Неявная загрузка приведет к появлению сообщения об ошибке при старте приложения, после чего приложение будет немедленно завершено.

C++Builder 6.0

Для явной загрузки библиотеки следует сначала объявить тип указателя на загружаемые функции. Мы будем загружать функцию **getSum** из хорошо знакомой нам подопытной библиотеки **XDll6**:

```
...
/* определяем тип "указатель на функцию" */
typedef int __cdecl (*dll_func)(int, int);

dll_func pGetSum = NULL;
...
```

Обратите внимание: спецификатор **__cdecl** обозначает, что библиотечные функции

поддерживают стандарт вызовов языка C (**calling convensions**). Другие альтернативы — это директивы **__pascal**, **__fastcall** и **__stdcall**. Следует обращать особое внимание на соответствие соглашения о вызовах библиотеки и использующего ее приложения. Неправильные соглашения о вызовах — одна из наиболее частых ошибок при использовании библиотек DLL (см. раздел «Типичные проблемы при работе с DLL»).

Теперь нам нужно загрузить библиотеку и получить указатели на соответствующие функции:

```
...
/* проецируем DLL на адресное пространство вызывающего процесса */
HMODULE lib = LoadLibrary("XDll6.dll");
if (lib == NULL)
{
   std::cout << "Can't load XDll6.dll library!"
   << std::endl;
   return -1;
}

/* пытаемся найти в таблице экспорта необходимую нам функцию */
pGetSum = (dll_func)GetProcAddress(lib, "getSum");
...
```

Теперь эту функцию можно вызывать, не забыв проверить, успешно ли получен указатель (в случае ошибки функция **GetProcAddress** возвращает значение **NULL**):

```
...
if (pGetSum == NULL)
{
   std::cout << "Can't find getSum function in
   dll!" << std::endl;
   return -2;
}

/* используем функцию так, словно мы сами ее написали */
const int n = pGetSum(10, 20);
std::cout << "getSum(10,20): " << n << std::endl;
...
```

Ну и разумеется, когда библиотека больше не нужна, ее следует выгрузить:

```
...
FreeLibrary(lib);
...
```

Если библиотеку не выгрузить, система сделает это сама при закрытии приложения. Тем не менее хорошим тоном программирования считается явная выгрузка.

Delphi 6.0

В этом случае приложение **Delphi** будет очень сильно напоминать только что рассмотренные выше программы (написанную на VC++ и на **C++ Builder**). Дело в том, что явная загрузка предполагает использование **WinAPI**, которое (что не удивительно!) является одним для всех. Отсюда и подобная «схожесть» приложений:

```
program XDllClient;

{$APPTYPE CONSOLE}

// явная загрузка
uses
   Windows;

var
   hModule: THandle;
   //объявляем переменную типа "указатель на функцию"
   pGetSum: function(const n1, n2: integer):
integer;
   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);

   WriteLn;
   WriteLn('Press any key...');
   ReadLn;

   FreeLibrary(hModule);
   end.
```

Для использования функций **Windows API** необходимо подключить модуль **Windows**. Это обеспечит импортирование в наше приложение функций **LoadLibrary**, **GetProcAddress** и **FreeLibrary**. Затем определить тип «указатель на функцию», отобразить DLL на адресное пространство нашего процесса при помощи **LoadLibrary**, получить необходимый адрес из DLL посредством **GetProcAddress**. После выполнения требуемых действий необходимо выгрузить библиотеку из памяти при помощи вызова **FreeLibrary**.

Стоит обратить внимание на один интересный нюанс. Использование адреса функции в условии **assert** при помощи записи **«pGetSum nil»** недопустимо, так как компилятор может подумать, что это обычный вызов функции (особенно если у нее нет параметров). Запись **«@pGetSum nil»** убедит компилятор: вы знаете, что делаете.

Кстати, как вы думаете: какой результат получится на экране при запуске приложения?

Выводы

После прочтения этого подраздела может сложиться мнение, что лучше неявной загрузки все равно ничего быть не может. Действительно, как можно было убедиться на примерах, явная загрузка предоставляет конечному программисту большую гибкость в управлении временем жизни как самой DLL, так и функций, ее составляющих: в любой момент после явного проецирования DLL на адресное пространство вызывающего процесса можно получить указатель на конкретную функцию; после того, как необходимость в использовании функций DLL отпадает, можно выгрузить динамическую библиотеку из памяти. Но за эту гибкость приходится расплачиваться достаточно большим количеством кода, который необходимо написать для этого.

Отложенная загрузка в определенных случаях сможет оказать существенную помощь программисту.

Отложенная загрузка

Этот вариант загрузки появился значительно позже своих «собратьев», описанных выше. Например, среда **Visual Studio** поддерживает данную функциональность, начиная с шестой версии. Если внимательно прочесть описания двух предыдущих способов работы с DLL, то будет очевидно, что каждый из них обладает определенными достоинствами и недостатками.

Явная загрузка:

(+) явное управление процессом жизни DLL;

(–) перекладывание большей части работы по управлению DLL на программиста.

Неявная загрузка:

(+) все заботы берут на себя компилятор и линкер;

(–) ресурсы занимаются все время, а не на момент использования;

(–) при работе с ПО в «неизвестной» обстановке такая загрузка не всегда бывает полезной.

Приведем пример, чтобы пояснить последний пункт. Упоминавшаяся ранее среда системы **MatLab** обладает гибким интерфейсом с программами, написанными на C++ (причина этого должна быть ясна — достаточно взглянуть на таблицу эффективности исполнения кода в разделе «Что такое DLL», и все вопросы отпадут сами собой). При этом интерфейс основан на использовании DLL, экспортирующей определенную функцию (**mexFunction**), на вход которой поступают специальные параметры, полностью описывающие свой тип, размеры и количество. Это позволяет построить DLL таким образом, что требуемая функциональность будет использоваться как в программе, написанной на чистом C++ (компилируемый вариант), так и в программе, написанной на m-языке (интерпретируемый язык).

При этом m-язык среды **MatLab** использует **mexFunction** для доступа к функцио-

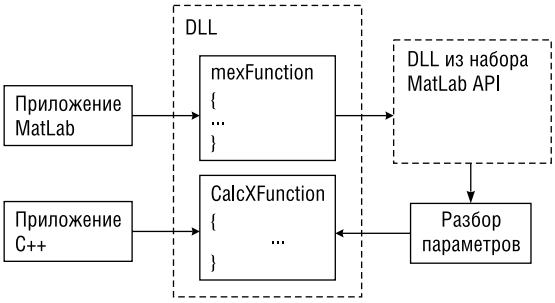

Рис. 5. Взаимодействие с DLL приложений MatLab и C++

нальности некоторой функции **CalcXFunc**, а C++ использует данную функцию напрямую (без посредников) — рисунок 5.

Но тут возникает вполне очевидная проблема. Дело в том, что для описания и последующего получения параметров в **mexFunction** используются специальные функции из набора **MatLab API**, которые, в свою очередь, заключены также в DLL. В программе на C++, естественно, эти функции не требуются (их вообще может и не быть на компьютере пользователя).

DLL нельзя загрузить неявно — при старте приложения система постарается загрузить DLL из таблицы импорта, что (в случае отсутствия установленной версии **MatLab**) опять приведет к ошибке. Придется использовать явную загрузку, что, к сожалению, повлечет дополнительный труд со стороны программиста.

Выход есть! Использование механизма отложенной загрузки позволяет избежать подобных проблем. В этом случае необходимые в **mexFunction** функции определяются как связанные с DLL отложенной загрузки. Тогда программа, использующая интерфейс **mexFunction**, будет вынуждена иметь требуемые DLL. При первом вызове необходимой функциональности данные DLL будут загружены в память. Программа на C++, не имеющая представления о **mexFunction**, сможет спокойно работать с данной DLL, не требуя дополнительных функций **MatLab API**.

Другой пример. Если программа предназначена для работы в различных версиях ОС, то, скажем, часть функций может появиться лишь в поздних версиях ОС и не присутствовать в ней на данный момент. Но ведь пока мы явно не обратимся к конкретной функции, DLL нам не нужна и мы можем спокойно продолжать работу. В момент обращения к несуществующей функции пользователю можно выдать соответствующее предупреждение.

Что же для этого нужно сделать? Ответим — немного! Хотя кое-что действительно придется сделать в клиентском приложении, которое хочет загрузить DLL способом отложенной загрузки.

VC++ 6.0/7.0

Для VC++ 6.0:

1. Как обычно, разместить в **«Link->General->Object/Library Modules»** lib-файл для используемой DLL.
2. Добавить туда же **delayimp.lib**.
3. В командную строку линкера (**«Link->General->Project Options»**) добавить команду /DELAYLOAD:xdll6.dll.

Замечание:

Как оказалось, некоторые дистрибутивы Visual C++ 6.0 поставляются с некорректным файлом **delayimp.lib**. В результате чего воспользоваться отложенной загрузкой не получится — при компиляции в окне **Build** появляется примерно следующая информация:

```
...
Linking...
delayimp.lib(delayhlp.obj) : error : Internal
error during Pass2
   ExceptionCode             = C0000005
   ExceptionFlags            = 00000000
   ExceptionAddress          = 1030C94B
   NumberParameters          = 00000002
   ExceptionInformation[ 0]  = 00000000
   ExceptionInformation[ 1]  = 00000002
CONTEXT:
   Eax     = 00000000  Esp    = 0012F0B4
   Ebx     = 00377DD0  Ebp    = 0012F0B8
```

```
   Ecx     = 00E700C0  Esi    = 10301934
   Edx     = 00001000  Edi    = 00001003
   Eip     = 1030C94B  EFlags = 00010206
   SegCs   = 0000001B  SegDs  = 00000023
   SegSs   = 00000023  SegEs  = 00000023
   SegFs   = 00000038  SegGs  = 00000000
   Dr0     = 0012F0B4  Dr3    = 00377DD0
   Dr1     = 0012F0B8  Dr6    = 00E700C0
   Dr2     = 00000000  Dr7    = 00000000
Error executing link.exe.
   ...
```

Чтобы воспользоваться отложенной загрузкой, необходимо заменить файл **DelayImp.lib**. Если сравнить корректный файл **DelayImp.lib** с тем, который поставляется с дистрибутивом VC++ (например, при помощи утилиты **fc.exe** с ключом **/b**), то легко обнаружить, что у них различаются двенадцать байтов. Таким образом, именно это (некорректный формат lib-файла) и приводит к аварийному завершению линкера.

Еще один вариант — это воспользоваться перекомпиляцией и построить заново проект, который будет непосредственно содержать требуемый код, поддерживающий отложенную загрузку. Для этого необходимо найти файлы **DelayImp.h** и **DelayHlp.cpp** в каталоге, содержащем установленную версию дистрибутива VC++ (обычно это **«VC98/Include»**), добавить в файл **DelayHlp.cpp** следующие строки:

```
...
extern "C"
PUnloadInfo __puiHead = 0;

PfnDliHook __pfnDliNotifyHook = NULL;
                      // добавим эту строку
PfnDliHook __pfnDliFailureHook = NULL;
                      // и эту тоже

struct ULI : public UnloadInfo {
...
```

Добавить эти файлы в проект (при этом не следует включать файл **DelayImp.lib** в настройки проекта), после этого завершить компиляцию проекта, в котором должна быть использована DLL отложенной загрузки. Все должно замечательно работать (но придется делать это для каждого проекта!). Да, и самое главное — в VC++ 7.0 подобной ошибки быть не должно.

Все, переходим на версию 7.0!

Для VC++ 7.0:

1. Поместить в **«Linker->Input->Additional Dependencies»** соответствующий lib-файл (например: **xdll.lib**).
2. В тот же раздел поместить **delayimp.lib**.
3. В **«Linker->Input->Delay Loaded DLL's»** прописать требуемые DLL, которым необходима отложенная загрузка.

Вот, в принципе, и все. Это позволит линкеру при компоновке exe-файла определить, что для DLL требуется отложенная загрузка, и удалить ее из таблицы импорта. При этом создастся новый раздел отложенной загрузки (**.didata**) — в этом можно убедиться, просмотрев секцию импорта при помощи утилиты **dumpbin** с ключом **/imports**:

```
...
Section contains the following delay load
imports:

XDll16.dll
    00000001 Characteristics
    0049E1F0 Address of HMODULE
    004A2070 Import Address Table
    004A2040 Import Name Table
    004A21B8 Bound Import Name Table
    00000000 Unload Import Name Table
           0 time date stamp

   0043846D0 ?getSum@@YAHHH@Z

Summary
    6000 .data
    1000 .didat
    1000 .idata
    C000 .rdata
   67000 .text
   30000 .textbss
...
```

Ну что ж... Мы поэкспериментировали с DLL отложенной загрузки.

Итак, вот общий алгоритм работы DLL с отложенной загрузкой (все эти действия совершаются абсолютно прозрачно для программиста):

1. Вызов вспомогательной функции (**helper**-функции — функции-переходника) вместо той, которая указана на самом деле. Адрес этой функции находится в специальной таблице, похожей на структуру IAT (и называется она **pseudoIAT**). При компоновке кода линкер заполняет **pseudoIAT** адресами функций-переходников.

2. Проверить — загружена ли необходимая DLL? Если библиотека еще не загружена, то загрузить ее, вызвав **LoadLibrary**.

3. Вызвать **GetProcAddress** с именем необходимой функции.

4. Вызвать функцию по полученному адресу, запомнив его для последующих вызовов. Осуществляется это в функции **__delayLoadHelper** (см. файл **DelayHlp.cpp**). Эта же функция занимается тем, что на место вызова функции-переходника в **pseudoIAT** подставляет адрес правильной функции, поиск которой был осуществлен в предыдущем пункте. Таким образом, все последующие обращения происходят уже напрямую.

5. При выгрузке DLL отложенной загрузки **pseudoIAT** вновь инициализируется первоначальными значениями, так что повторный вызов функции приведет к выполнению алгоритма с п. 1.

Таким образом, следует четко уяснить, что поддержка отложенной загрузки осуществляется исключительно средствами компоновщика и внешним дополнительным кодом, который для **Microsoft Visual C++** поставляется в виде исходных файлов.

Также можно использовать **callback**-функции для отслеживания состояний загрузки и выгрузки DLL отложенной загрузки:

- при запуске **helper**-функции;
- перед вызовом **LoadLibrary**;
- перед вызовом **GetProcAddress**;
- при ошибке вызова **LoadLibrary**;
- при ошибке вызова **GetProcAddress**;
- после завершения обработки **helper**-функцией.

Приведем небольшой пример, как производится подобная обработка в этом случае. Для начала необходимо определить вспомогательную функцию, которая будет получать соответствующие уведомления. Например, так (см. стр. 65).

Предоставленный каркас функции обработки может быть использован для реагирования на соответствующие уведомления. Для их получения необходимо присвоить соответствующим обработчикам необходимые адреса:

```
PfnDliHook __pfnDliNotifyHook2 = delayHook;
PfnDliHook __pfnDliFailureHook2 = delayHook;

void main()
{
    ...
}
```

Как видите, существует два различных обработчика: для уведомлений о ходе обработки (**__pfnDliNotifyHook2**) и уведомлений об ошибках (**__pfnDliFailureHook2**). Мы определили одну и ту же функцию для обоих случаев, что в определенных ситуациях бывает более удобным.

Для VC++ 6.0 эти функции употребляются в той же нотации без индекса «2».

Предположим, мы хотим вернуть вместо функции сложения двух целых чисел (**getSum**) некоторую другую функцию. Перепишем немного фрагмент кода основной функции **main**.

```
// функция вычитания двух чисел
int getDiff(const int n1, const int n2)
{
    return n1 - n2;
}

FARPROC WINAPI delayHook(unsigned int dliNotify,
PDelayLoadInfo pdli)
{
```

```
    switch (dliNotify) {
        ...
        case dliNotePreGetProcAddress:
            // вернем наш ответ Чемберлену!
            return (FARPROC)getDiff;
        ...
    }
}

// объявляем обработчик нотификационных
// уведомлений
PfnDliHook __pfnDliNotifyHook2;

void main()
{
    // указываем свой обработчик
    __pfnDliNotifyHook2 = delayHook;
    // используем функцию так, словно мы сами
    // ее написали
    const int res = getSum(10, 20); // (1)

    std::cout << "getSum(10, 20): " << res <<
    std::endl;

    // сообщаем, что DLL нам больше не нужна
    const BOOL b = __FUnloadDelayLoadedDLL2("XDll.
    dll");
    _ASSERT(b);

    // а вот здесь обработчик нам уже не требуется
    __pfnDliNotifyHook2 = NULL;
    // используем функцию так, словно мы сами ее
    // написали
    const int res1 = getSum(20, 30); // (2)

    std::cout<<"getSum(20,30):"<<res<<std::endl;
}
```

Как видите, ничего сложного. Мы определили свою собственную функцию **getDiff**, алгоритм которой не намного сложнее **getSum**. В ответ на уведомление **dliNotePreGetProcAddress** возвращаем адрес функции **getDiff**. Таким образом, в результате вызова (1) получаем на экране:

```
getSum(10, 20): -10
```

— что может очень сильно удивить обычного пользователя.

А вот вызов (2) приведет к появлению вполне ожидаемого результата:

```
getSum(20, 30): 50
```

Происходит это из-за того, что в первом случае мы явно указали требуемый обработчик уведомления **dliNotePreGetProcAddress**, в котором реализована логика подмены адреса функции. Во втором случае мы отказались от обработчика, а следовательно, **helper**-функция провела свою работу по очевидному маршруту.

Отложенная загрузка предполагает ряд ограничений. Перечислим некоторые из них:

1. Импортирование данных при помощи отложенной загрузки не может быть осуществлено. Придется использовать для этого средства явной загрузки. При попытке компиляции появится сообщение об ошибке:

```
...
Linking...
LINK : fatal error LNK1194: cannot delay-load XDll6.
dll due to import of data symbol "__declspec
(dllimport) int g_N" (__imp_?g_N@@3HA);
relink without /DELAYLOAD:XDll6.dll
Error executing link.exe.

XDllClient6.exe - 1 error(s), 0 warning(s)
...
```

Замечание:

Подобное сообщение об ошибке появляется лишь в случае использования lib-файла, построенного по DLL, сгенерированной **Visual C++**. Если же DLL была получена другими средствами (например, **Borland**), то сообщение об ошибке возникать не будет, но и значение экспортируемой переменной получить не удастся.

2. Нельзя загрузить **kernel32.dll** при помощи отложенной загрузки. Это связано с тем, что в ней находится код, связанный с осуществлением отложенной загрузки.

3. На данный момент не существует возможности провести отложенную загрузку только для какой-то одной конкретной функции из DLL.

4. Связывание перенаправленных функций не поддерживается для DLL отложенной загрузки.

```
FARPROC WINAPI delayHook(unsigned int dliNotify, PDelayLoadInfo pdli)
{
   switch (dliNotify) {
      case dliStartProcessing:
           // можно вернуть указатель на функцию, которая будет вызвана helper-функцией
           // вместо той, которая должна быть вызвана при обработке по умолчанию; если это
           // не требуется, верните NULL
           break;
      case dliNotePreLoadLibrary:
           // можно вернуть некоторое свое значение HMODULE вместо того, чтобы helper-функция
           // вызывала LoadLibrary (как это происходит по умолчанию); если это не требуется,
           // верните NULL
           break;
      case dliNotePreGetProcAddress:
           // можно вернуть указатель на функцию, которая будет вызвана helper-функцией
           // вместо той, которая должна быть вызвана при обработке по умолчанию; если это
           // не требуется, верните NULL
           break;
      case dliFailLoadLib:
           // ошибка вызова LoadLibrary
               // чтобы не принимать во внимание эту ошибку, верните NULL - в этом случае
               // будет возбуждено исключение (ERROR_MOD_NOT_FOUND) с последующим выходом;
               // также можно загрузить некоторую альтернативную DLL и вернуть ассоциированное
               // с ней значение HMODULE; тогда helper-функция попробует разрешить требуемую
               // ссылку в этой DLL
           break;
      case dliFailGetProc:
           // ошибка вызова GetProcAddress
               // чтобы не принимать во внимание ошибку, верните NULL - в этом случае будет
               // возбуждено исключение (ERROR_PROC_NOT_FOUND) с последующим выходом; вы также
               // можете вернуть указатель на альтернативную функцию
           break;
      case dliNoteEndProcessing:
           // это уведомление присылается после завершения обработки
               // здесь уже невозможно каким-либо образом повлиять на поведение helper-функции
               // за исключением longjmp()/throw()/RaiseException. Возвращаемое значение
               // игнорируется.
           break;
      default:
           return NULL;
   }

   return NULL;
}
```

5. Статические (глобальные) указатели на импортированные функции должны быть дополнительно инициализированы после отложенной загрузки DLL. Это связано с тем, что изначально подобные указатели ссылаются на функцию-заглушку.

В заключение данного раздела скажем пару слов о выгрузке DLL отложенной загрузки.

После использования функциональности DLL не всегда имеет смысл держать ее дальше в ОП. Для того чтобы осуществить явную выгрузку DLL из памяти, следует проделать следующее:

Для VC++ 6.0:

1. Проставить в опциях линкера (**«Link->General->Project Options»**) ключ **/DELAY:UNLOAD**.

2. Вызывать функцию **__FUnloadDelayLoadedDLL** для выгрузки DLL из ОП.

Для VC++ 7.0:

1. Проставить **«Linker->Advanced->Delay Loaded DLL»** в значение **Support Unload**.

2. Вызывать функцию **__FUnloadDelayLoadedDLL2** для выгрузки DLL из ОП.

Замечание:

Если не выполнить действия п. 1, то функции **__FUnloadDelayLoadedDLL/__FUnloadDelayLoadedDLL2** будут возвращать значение **FALSE** (что свидетельствует о невозможности выгрузки DLL).

Продемонстрируем сказанное на примере. Вновь модифицируем файл **main.cpp**:

```
#include "../XDll6/XDll.h"
#include <windows.h>

/* в этом файле (delayimp.h) содержится объявление функции __FUnloadDelayLoadedDLL2; не забудьте ПЕРЕД ним также включить windows.h для нахождения требуемых констант */
#include <delayimp.h>
#include <iostream>
#include <crtdbg.h>

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

    std::cout << "getSum(10,20): " << res << std::endl;

    /* сообщаем, что DLL нам больше не нужна */
    const BOOL b = __FUnloadDelayLoadedDLL2("XDll6.dll");
    _ASSERT(b);

    const int res1 = getSum(20, 30);
}
```

Здесь есть один подводный камень — дело в том, что параметр функции **__FUnloadDelayLoadedDLL2** должен в точности совпадать с тем именем, которое указано в параметре **/DELAYLOAD** (с точностью до указанного расширения). Если ошибиться в регистре хотя бы одного символа, то функция вернет значение **FALSE**.

Как же проще всего отследить момент загрузки и выгрузки DLL? VC++ и здесь нам поможет. Версия 6.0 содержит подменю **«Debug->Modules»** (которое появляется на момент отладки приложения). В нем содержится список спроецированных на адресное пространство отлаживаемого процесса модулей.

Правда, существует большое неудобство при работе с ним — так как это диалоговое окно модального типа, то придется постоянно (после вызова функции) искать пункт данного меню, чтобы просмотреть вновь этот список.

VC++ 7.0 и в этом пошел нам навстречу. Заходим в **«Debug->Windows->Modules»**. На экране появляется плавающее окно, которое обновляется автоматически. Исследуем наш код — пошаговая отладка явно показывает, что **XDll6.dll** будет загружена при вызове функции **getSum**, затем выгружена при вызове функции **__FUnloadDelayLoadedDLL2** и вновь загружена при втором вызове **getSum**. Отложенная загрузка работает!

C++Builder 6.0

Использование отложенной загрузки динамических библиотек в среде **C++Builder** мало отличается от неявной загрузки. Мы аналогично добавляем в проект h-файл с описаниями и lib-файл, соответствующий библиотеке DLL. Далее во вкладке **«Project->Options->Advanced Linker->Delay load»** добавляем **XDll6.dll**. Готово! (Аналогичный эффект будет иметь добавление опции компилятора **/dXDll.dll**.)

Теперь об отличиях: нельзя использовать отложенную загрузку для библиотек, имеющих секцию импорта (т.е. использующих другие библиотеки), а также **Kernel32.dll** и **RTLDLL.dll** (так как функции поддержки отложенной загрузки как раз и находятся в последней).

Кроме того, когда библиотека больше не нужна, ее следует явно выгрузить. Для этого нужно воспользоваться функцией **__FUnloadDelayLoadedDLL**, передав ей в качестве параметра имя DLL (в точности до регистра символов, включая расширение). Следует осторожно пользоваться этой функцией в многопоточных приложениях: если один поток все еще использует функции DLL, а другой уже выгрузил ее, это может привести к непредсказуемым последствиям.

Delphi 6.0

Замечание:

Среда Delphi на данный момент не предоставляет требуемой функциональности для обеспечения механизма отложенной загрузки DLL.

Выводы

Ну что ж, теперь рассмотрим, каковы преимущества и недостатки у этого способа работы с DLL:

(+) дополнительная гибкость в управлении жизненным циклом DLL;

(+) минимум дополнительной работы с точки зрения программиста;

(–) небольшие дополнительные издержки, связанные с первоначальным вызовом функций. Дело в том, что для вызова функции нужно знать ее точный виртуальный адрес. Узнать его мы можем исключительно после проецирования DLL на адресное пространство вызывающего процесса. А это происходит лишь при вызове какой-либо функции из этой DLL, но мы ведь не знаем ее виртуальный адрес. И так можно продолжать до бесконечности. Разработчики этой технологии осуществили ее реализацию следующим образом: в месте вызова функции из DLL отложенной загрузки генерируется код вызова не самой функции, а некоторой вспомогательной (**helper**) функции. Именно ее код и отвечает за проверку загрузки DLL и, в случае отсутствия библиотеки в ОП, последующего ее проецирования. После этого вычисляется виртуальный адрес функции и запоминается. Последующие вызовы данной функции уже осуществляются напрямую — через полученный ранее виртуальный адрес;

(+) простота и удобство эксплуатации.

Таким образом, отложенная загрузка — это механизм, который позволяет отложить процесс инициализации DLL до момента первого вызова использования. В случае когда обычных средств для работы с DLL бывает недостаточно, это именно тот способ, который сможет вам помочь в самых сложных ситуациях.

Типичные проблемы при работе с DLL

Какие подводные камни могут подстерегать ничего не подозревающего разработчика на пути использования DLL? Наиболее часто возникающие вопросы по этой теме рассмотрены далее.

Замечание:

Эта информация носит познавательный характер и не претендует на полноту!

Проблема 1. Почему моя DLL не загружается?

Такой вопрос нередко задают начинающие программисты. Должны признаться, что мы также неоднократно ломали голову над этой проблемой. Почему функция **Lo-**

adLibrary вместо правильного значения **HINSTANCE** для загружаемой DLL возвращает **NULL**, хотя файл DLL гарантированно находится в текущем каталоге программы и доступен для загрузки?

Ничего таинственного в этом нет. Вспомним еще раз, как система загружает DLL: загрузчик операционной системы отыскивает файл DLL и подключает (проецирует) его к адресному пространству загружающего процесса. Затем загрузчик просматривает секцию импорта загружаемой DLL, извлекает из нее имена библиотек, неявно загружаемых (**implicit linking**) данной DLL, и пытается подключить их к адресному пространству процесса, затем просматривает секции импорта этих DLL и так далее. Процесс продолжается рекурсивно до тех пор, пока весь необходимый код и данные не окажутся успешно спроецированы на адресное пространство процесса. Если в какой-то момент загрузчик не сможет найти хотя бы одну из необходимых DLL, то немедленно прервет загрузку и отключит от адресного пространства процесса все загруженные модули, а функция **LoadLibrary** вернет значение **NULL**, сигнализируя о неуспешной загрузке.

Можно потерять не один час, пытаясь разобраться, почему не загружается DLL при запуске на одной машине, хотя на другой машине эта же DLL может загружаться успешно. Как правило, проблема заключается в следующем.

Приложение загружает DLL-библиотеку, которая неявно (**implicit linking**) загружает другую DLL, а та, в свою очередь, неявно загружает одну из библиотек системного назначения (например, **Msvcp60.dll**). Вот этато библиотека и отсутствует на одной машине и присутствует на другой. Для решения проблемы необходимо включить требуемую DLL в дистрибутив программы.

Чтобы избежать этой неприятности, используйте утилиту **Dependency Walker** (**depends.exe**) из **Platform SDK** для отслеживания взаимозависимостей между загружаемыми модулями либо утилиты аналогичного назначения других фирм. В дистрибутив программы необходимо включить все необходимые приложению DLL.

Проблема 2. Несовместимость интерфейсов используемых функций

Представим себе другую ситуацию: мы имеем DLL, которую хотим использовать в своей программе. Эта DLL великолепно отлажена, задокументирована, написана в такой же среде, в которой мы собираемся писать программу, имеются снабженные комментариями на русском языке заголовочные файлы, которые без проблем включаются в текст программы. Еще лучше — есть исходники библиотеки. А еще лучше — мы сами эту библиотеку и написали...

На самом деле чаще всего бывает как раз наоборот: документация к библиотеке неполная или вообще отсутствует, заголовочных файлов нет, исходников нет и т. д. Основной проблемой загрузки библиотеки в таком случае становится определение и корректная декларация параметров функций (что будет в случае их несоответствия? Об этом можно узнать в разделе «Разработка и использование DLL в различных средах»). Проблема состоит еще и в том, что типы данных в различных языках и даже в различных реализациях одного языка отличаются. В этой ситуации выходом может быть использование типов **WinAPI**-библиотек, таких как **BYTE**, **WORD**, **DWORD**, **PVOID** и т. д. Такие типы, как правило, корректно поддерживаются во всех средах, поддерживающих библиотеку функций **WinAPI**.

Проблема 3. Декорирование имен

Не менее актуальна и проблема декорирования имен, когда исходное имя функции при экспортировании искажается с целью сохранения дополнительной информации об этом имени (причем правила декорирования меняются от компилятора к компилятору). Разработчик не всегда об этом помнит, а потому пытается использовать то имя, которое было определено в исходном коде DLL. Этот процесс в действии рассматривался в предыдущем разделе — при использова-

нии функции **getSum** в VC++ нам пришлось писать что-то вроде **?getSum@@YAHHH@Z**. В противном случае линкер не может связать необходимые имена, о чем сообщает в виде ошибки «Неразрешимая внешняя ссылка». Как избавиться от декорирования, более подробно изложено в разделе «Декорирование имен».

Проблема 4. Конфликт версий

Другой распространенной проблемой при загрузке DLL является конфликт версий. Как определить, что требуемая DLL имеет нужную версию и что в ней содержится требуемая функция? Одним из выходов в такой ситуации является использование ресурса **VERSIONINFO** для определения версии библиотеки. Подробнее об этом можно почитать в разделе «DLL Hell».

Проблема 5. Конфликты базовых адресов

Среди технических проблем загрузки DLL следует отметить конфликты базовых адресов библиотек. Дело в том, что у каждой динамически компонуемой библиотеки есть предпочтительный базовый адрес. Это адрес памяти, по которому DLL будет загружена в кратчайший срок. Для просмотра предпочтительных базовых адресов программы используется утилита **DUMPBIN.EXE** из пакета **Visual Studio** с ключом **/HEADERS** или другие утилиты аналогичного назначения (например, **TDUMP.EXE** компании **Borland**).

Если несколько библиотек претендуют на один и тот же адрес, то только одна из них загружается по заданному базовому адресу, все остальные же будут загружены по другим базовым адресам, при этом время загрузки сильно увеличивается — немногие об этом знают и, тем более, задумываются!

Замечание:

По умолчанию большинство компоновщиков устанавливают базовый адрес загрузки DLL в **0x10000000**. Сегодня как минимум половина библиотек DLL в мире пытается загрузиться по этому адресу.

Для того чтобы сократить время загрузки, следует изменить предпочтительные адреса библиотек таким образом, чтобы они не перекрывали друг друга. Для этого подойдет утилита **REBASE.EXE** из пакета **Visual Studio**.

Еще раз повторим, что если не позаботиться о разнесении предпочтительных базовых адресов используемых библиотек, то эту работу проделает загрузчик. Правда, сделать он это сможет, если при компиляции библиотеки сохранить раздел переадресации (**reallocation table**). Если же такой раздел не будет сохранен (в целях уменьшения размера DLL), то загрузка динамической библиотеки приведет к ошибке.

Кстати, Джеффри Рихтер разработал полезную функцию в приложении **ProcessInfo.exe**, которая позволяет отслеживать, какие DLL были загружены по их предпочтительным адресам, а какие нет!

Ниже приведена часть листинга модулей, которые находятся в адресном пространстве процесса **devenv.exe** (см. стр. 70).

Поле **BaseAddr** показывает адрес, по которому модуль был спроецирован на адресное пространство. Поле **ImageAddr** является пустым, если модуль отображен на его предпочтительный базовый адрес. Иначе в скобках указывается тот адрес, который был указан при создании модуля. Как видим, у **Microsoft Visual Studio** с этим проблем нет!

Замечание:

Microsoft вообще очень внимательно отслеживает подобные проблемы и избегает конфликтов базовых адресов.

Проблема 6. Пути поиска DLL

Как правило, это оборачивается проблемой при связывании DLL (в случае неявной загрузки) — в этом случае на экране появляется сообщение об ошибке, информирующее о невозможности найти требуемую DLL (в случае использования явной загрузки **LoadLibrary** возвращает значение **NULL**). Подробнее об этом можно почитать в разделе «Алгоритм загрузки DLL». Также это

```
Filename: devenv.exe
   PID=000005C0, ParentPID=0000074C, PriorityClass=8, Threads=27, Heaps=39

Modules Information:
   Usage  BaseAddr(ImagAddr) Size    Module
   Fixed  00400000          196608  ... .NET\Common7\IDE\devenv.exe
       2  50940000          86016   ... .NET\Common7\IDE\1033\msenvmui.dll
       9  50000000          3399680 ... .NET\Common7\IDE\msenv.dll
      15  50880000          241664  ... .NET\Common7\IDE\1033\msenvui.dll
       1  54EC0000          253952  ... .NET\Common7\IDE\VS SCC\VisualStudioTeamCore.dll
       1  54F10000          24576   ... .NET\Common7\IDE\VS SCC\1033\VisualStudioTeamCoreui.dll
       2  504C0000          360448  ... .NET\Common7\IDE\vsmacros.dll
       5  50930000          40960   ... .NET\Common7\IDE\1033\vsmacrosui.dll
       1  50520000          40960   ... .NET\Common7\IDE\msenvmnu.dll
       2  50440000          233472  ... .NET\Common7\IDE\vsbrowse.dll
       4  509D0000          36864   ... .NET\Common7\IDE\1033\vsbrowseui.dll
       1  516C0000          663552  ... .NET\Common7\Packages\Debugger\vsdebug.dll
       4  517D0000          102400  ... .NET\Common7\Packages\Debugger\1033\VSDebugUI.dll
       1  53420000          499712  ... .NET\Vc7\vcpackages\VCProject.dll
       1  538A0000          151552  ... .NET\Vc7\VCPackages\1033\VCProjectUI.dll
       2  53670000          1052672 ... .NET\Vc7\vcpackages\VCProjectEngine.dll
       1  50410000          118784  ... \Common Files\Microsoft Shared\MSEnv\msenvp.dll
       1  534A0000          1736704 ... .NET\Vc7\vcpackages\vcpkg.dll
       1  538E0000          323584  ... .NET\Vc7\vcpackages\1033\vcpkgui.dll
      25  79170000          135168  ... \WINDOWS\System32\mscoree.dll
       1  50800000          118784  ... .NET\Common7\IDE\compluslm.dll
       1  53CF0000          368640  ... .NET\Vc7\vcpackages\VCProjectConversion.dll
       4  10300000          233472  ... .NET\Common7\IDE\mspdb70.dll
       1  5CC00000          1900544 ... .NET\Vc7\vcpackages\FEACP.DLL
       3  51690000          155648  ... .NET\Common7\Packages\Debugger\sdm2.dll
       1  51770000          53248   ... .NET\Common7\Packages\Debugger\encmgr.dll
       1  54000000          65536   ... .NET\Common7\Packages\Debugger\ecbuild.dll
       1  53EB0000          610304  ... .NET\Common7\Packages\Debugger\NatDbgDE.dll
       1  53E70000          98304   ... .NET\Common7\IDE\msenc70.dll
       1  53F90000          233472  ... .NET\Common7\Packages\Debugger\NatDbgEE.dll
       1  10400000          290816  ... .NET\Common7\IDE\msdis130.dll
       1  53E90000          20480   ... .NET\Common7\Packages\Debugger\NatDbgTLLoc.dll
       1  53F60000          163840  ... .NET\Common7\Packages\Debugger\NatDbgDM.dll
```

касается момента отладки, когда среда «не видит» DLL — в этом случае может быть установлена неправильная директория по умолчанию либо DLL расположена не в том каталоге, который требуется.

Проблема 7. Инициализация DLL

DLL может содержать специальную функцию **DllMain**, которая в общем случае вызывается при отображении динамической библиотеки на адресное пространство. Если в процессе исполнения **DllMain** вернет значение **FALSE**, то попытка загрузки библиотеки закончится неудачей (в случае явной загрузки вызов функции **LoadLibrary** вернет значение **NULL**). Информация о назначении и использовании функции **DllMain** находится в разделе «Зачем нужна функция DllMain?»

Проблема 8. Связывание адресов

Если провести предварительное связывание адресов (в целях оптимизации загрузки DLL), то установка любого **service pack** (пакета обновлений) приведет к необходимости повторного связывания адресов, о которой можно легко забыть.

Что такое связывание? Это процесс, позволяющий «заранее» вычислить виртуальные адреса всех импортируемых в приложение функций (такая операция проводится только один раз — например, при установке приложения). Вычисленный адрес сохраняется в специальной таблице импортируемых адресов (**IAT — Import Address Table**) вашего модуля. Поэтому системному загрузчику не требуется вычислять эти адреса каждый раз при старте приложения, что позволяет существенно ускорить загрузку такого приложения (особенно в том случае, когда оно использует много DLL). Связывание можно выполнить вручную выполнением утилиты bind.exe, но мы настоятельно не рекомендуем это делать, поскольку стабильность работы системы и/или прикладных программ при установке любого пакета обновления ОС будет катастрофически нарушена и весь процесс связывания для каждого приложения придется производить заново.

Проблема 9. Использование отложенной загрузки в VC++ 6.0

Проблема с DLL отложенной загрузки в VC++ 6.0 была подробно разобрана в предыдущем подразделе (см. «Как приложение загружает DLL»).

Проблема 10. Ошибка выгрузки DLL отложенной загрузки

В случае использования отложенной загрузки возможна неправильная установка ключей при компиляции exe-приложения. Например, если «забыть» указать ключ **/DELAY:UNLOAD**, то любые попытки выгрузить DLL из ОП при помощи **__FUnloadDelayLoadedDLL/__FUnloadDelayLoadedDLL2** закончатся неудачей (и возвратом значения **FALSE**).

Кроме этого, параметр, передаваемый в функцию выгрузки DLL отложенной загрузки из памяти, должен в точности (вплоть до указанного расширения) совпадать с тем, что был указан в параметре ключа **/DELAYLOAD**.

Проблема 11. Несоответствие моделей управления памятью

Более тонкие ошибки происходят в связи с несоответствием в моделях управления памятью. Как правило, такие ошибки связаны с тем, что динамически выделенная память размещается в разных хипах (heaps — локальные кучи) — например, ее выделение происходит в контексте DLL, а освобождение — в контексте приложения. К чему это может привести? В общем случае — к полному краху вашего приложения (а если особенно «повезет», то изредка появляющимися ошибками).

Почему такие ошибки возникают? Дело в том, что различные языки программирования используют различные стандарты и принципы размещения и освобождения памяти в куче — например, известно, что **Pascal** предполагает резервирование четырех байт, предшествующих возвращенному адресу участка кучи, в которых записывается размер выделенного фрагмента. Если память выделена и освобождена кодом, сгенерированным одним компилятором, то, скорее всего, все пройдет неплохо (хотя и здесь есть свои сложности, связанные с наличием в памяти различных версий RTL (**runtime library**), которые также могут использовать различные варианты выделения памяти). А если нет?

Как написать DLL с нормальными экспортами (как у системных DLL Windows)?

VC 6.0 / VC 7.0

В действительности проблема написания DLL с нормальными экспортами напря-

мую связана с понятием декорирования имен.

Декорирование (*или искажение, mangling*) *имен* — это специфическое явление, присущее компиляторам языка C++, которое необходимо учитывать при разработке DLL на этом языке. Оно заключается в том, что компилятор C++ к имени функции всегда добавляет сокращенный список формальных параметров и тип возвращаемого значения. Например, имя функции, прототип которой выглядит как

```
int MyFunc(int,int,char*);
```

будет преобразовано компилятором в нечто вроде

```
?MyFunc@@YAHHHPAD@Z
```

Замечание:

Так декорирует имя функции компилятор **Microsoft Visual C++**. У разных компиляторов различные схемы искажения имен, поэтому результаты компиляторов могут отличаться.

Но зачем это нужно? Не проще ли было бы обойтись без декорирования имен? Оказывается — нет. Дело в том, что в языке C++ широко распространена перегрузка функций — объявление функции с тем же именем, но с другим набором формальных параметров. Именно благодаря декорированию имен компилятор может правильно связать вызовы функций в программе. Так, например, имена функций

```
int MyFunc(int,int,char*);
int MyFunc(int);
```

будут преобразованы компилятором в:

```
?MyFunc@@YAHHHPAD@Z
?MyFunc@@YAKH@Z
```

Таким образом, с точки зрения компилятора и компоновщика (линкера) это различные имена.

Аналогичным образом искажаются имена функций — членов классов. Например, имя функции

```
CString::operator=(CString const &)
```

преобразуется компилятором в

```
??4CString@@QAEABV0@ABV0@@Z
```

Декорирование имен усложняет экспорт функций из DLL, написанной на языке C++. Например, в использующем DLL приложении попытка получить адрес функции **MyFunc** закончится неудачей — вызов **GetProcAddress(«MyFunc»)** вернет **NULL**, потому что компоновщик честно поместит в секцию экспорта DLL декорированное имя **?MyFunc@@YAHHHPAD@Z** вместо ожидаемого **MyFunc**.

Чтобы избежать этого, необходимо объявлять все экспортируемые функции с модификатором **extern «C»** — тогда компилятор не будет искажать имя функции.

Для экспорта из DLL нескольких функций без декорирования их имен можно поместить объявления в блок, помеченный этим модификатором:

```
extern "C"
{
   int Func1();
   const char* Func2(const char*);
   void Func3(int, int);
   }
```

Имена функций **Func1**, **Func2**, **Func3** будут экспортированы без искажений. К сожалению, этот прием нельзя применять для экспорта из DLL перегруженных функций, равно как и для экспорта функций классов.

Подробнее о способах декорирования имен можно почитать в разделе «Декорирование имен».

Borland C++ Builder 6.0

Сначала рассмотрим вкратце, как вообще создается DLL в среде **C++ Builder**. Для соз-

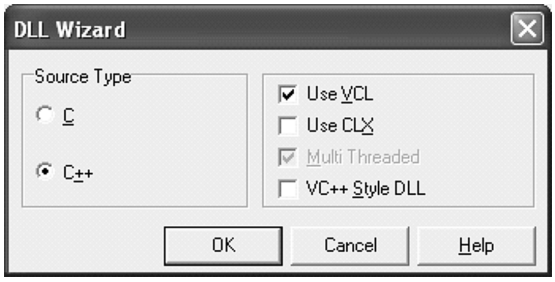

Рис. 6. DLL Wizard среды Borland C++ Builder

дания нового проекта DLL нужно выбрать команду **«File->New->Other…»**, затем в окне из списка выбрать **«DLL Wizard»**. После этих манипуляций появляется окно настройки некоторых свойств будущей DLL.

«Source type» определяет, какой язык будет использоваться для написания DLL: «чистый» C или C++. **«Use VCL»** и **«Use CLX»** указывает линковщику, нужно ли подключать к DLL библиотеки VCL и CLX. «**Multi Threaded»** указывает, будет ли генерируемая библиотека создавать дополнительные потоки (**threads**). И, наконец, **«VC++ Style DLL»** указывает компилятору, какая функция будет точкой входа в DLL. Если снять эту галочку, точкой входа в DLL будет функция

```
int WINAPI DllEntryPoint (HINSTANCE hinst, unsigned
long reason, void* lpReserved)
```

Если эту галочку установить, входной функцией будет

```
BOOL WINAPI DllMain (HINSTANCE hinstDLL, DWORD
fwdreason, LPVOID lpvReserved)
```

Такой вариант, очевидно, введен для совместимости кода с 16-битными приложениями.

Однако если построить (**«Build»**) такую библиотеку, то функция **Factorial** примет несколько иной вид (примерно такой: **@Factorial$ql**), и импортировать ее из другого компилятора будет затруднительно. Сейчас мы ставим перед собой задачу избавиться от декорирования и получить DLL с «нормальными» именами.

Первое, что нужно сделать, это в свойствах проекта (**«Projects->Options…»**) на закладке **«Advanced Compiler»** в разделе **«Output»** убрать галочку с **«Generate underscores»**. Это «отучит» компилятор добавлять символ подчеркивания перед именем экспортируемой функции.

Далее, нужно изменить описание функции следующим образом:

```
extern "C" __declspec(dllexport) long Factorial
(long);
```

Оператор **extern «C»** указывает компилятору, что для этой функции декорирование применять не нужно.

Кроме того, можно указать компилятору не применять декорирование имен для группы функций:

```
extern "C"
   {
   __declspec(dllexport) void Func1();
   __declspec(dllexport) void Func2();
   __declspec(dllexport) void Func3();
};
```

или даже ко всему header-файлу с функциями:

```
extern "C"
{
   #include "MyFunctions.h"
};
```

Замечание:

Компилятор **Borland C++ Builder**, даже несмотря на модификатор **extern «C»**, по умолчанию добавляет к имени функции знак '_' (**underscore**, подчеркивание), поэтому имя **MyFunc** в этой среде будет экспортировано как **_MyFunc**. Чтобы избежать этого, необходимо в настройках компилятора снять соответствующую опцию (**«generate underscore»**). Только тогда имя **MyFunc** будет экспортировано правильно.

После всех этих манипуляций экспортируемые функции в DLL будут иметь «нормальный» («системный» © **Shunix**) вид.

Delphi 6

Из всего того, что было написано про работу с DLL в среде **Delphi** в предыдущих главах, становится ясно, что эта среда не поддерживает декорирование имен.

Все экспортируемые имена появляются в заголовке файла DLL без каких-либо искажений.

Как уже отмечалось, при экспортировании имен в **Delphi** мы используем специальное ключевое слово **exports**. Именно оно указывает компилятору, под каким именем тот или иной идентификатор будет записан в бинарный файл (*.dll).

В нашем примере для экспортирования **getSum** мы использовали запись следующего вида:

```
exports
   getSum;
```

Используя утилиту **dumpbin** с параметром **/exports**, можно видеть, что в DLL экспортированное имя находится безо всяких искажений:

```
Ordinal hint  RVA         name
   1      1   00009CA4    getSum
```

Аналогичный результат получим и при использовании утилиты **tdump** с ключом **–ea**:

```
Exports from XDll.dll
   1 exported name(s), 1 export addresse(s).
   Ordinal base is 1.
   Not sorted
      RVA        Ord.  Hint   Name
      -------    ----  ----   ----
      00009CA4     1   0001   getSum
```

Замечание:

Среда **Delphi** поддерживает декорирование (**mangling**) только для объектных файлов *.obj (см. **«Project->Options->Linker->Linker Output»**). В случае генерации obj-файлов в формате C++ (**«Generate C++ object files»**) становится возможным подключение и использование таких файлов, например, в среде **Builder C++**.

Чтобы просмотреть, в каком виде находятся имена в obj-файле, достаточно использовать утилиту **tdump** с параметром **–d**:

```
Dynamic link export (EXPDEF)
   Exported by: name
   Exported Name: getSum
   Internal Name: __stdcall getSum(const int,
   const int)
```

В данном случае опция декорирования имен (манглинг) C++ включена.

Visual Basic

Средствами чистого **Visual Basic** создание DLL общего назначения (с экспортируемыми функциями как у системных DLL **Windows**) невозможно[2].

Visual Basic поддерживает создание только **ActiveX DLL**, т. е. DLL-библиотек, содержащих COM-объекты. Если просмотреть таблицу экспорта такой DLL, то окажется, что любая **ActiveX DLL** экспортирует только четыре функции:

- **DllRegisterServer**
- **DllUnregisterServer**
- **DllCanUnloadNow**
- **DllGetClassObject**

Поскольку сущность и применение технологии COM выходят за рамки данной статьи, мы отсылаем читателя к соответствующей литературе, например к книге Дэйла Роджерсона «Основы COM».

[2] Дэн Эпплман в своей монографии «The Visual Basic Programmer's Guide to the Win32 API» пишет: «Visual Basic не позволяет экспортировать функции, которые могут быть напрямую вызваны из других приложений. DLL, созданные VB, используют OLE-интерфейс. В тех случаях, когда вам действительно необходимо создать DLL, экспортирующую функции, вам нужно либо использовать дополнительные программные средства, дополняющие стандартные возможности VB (такие как, например, Desaware's SpyWorks), либо более традиционные средства разработки и языки, предназначенные для этого».

Тем не менее с помощью дополнительных утилит сторонних фирм все-таки оказывается возможным создать DLL-библиотеку общего назначения. В качестве примера отправим читателя на http://www.vbadvance.com. Тем не менее необходимо иметь в виду, что для использования такой DLL общего назначения, написанной на языке **Visual Basic**, обязательно потребуется исполняющая система VB — библиотека **msvbvm50.dll**, **msvbvm60.dll** и т. д. — объемом ни много ни мало 1,3 Мбайт!

Поскольку, по мнению авторов этой работы, упомянутые утилиты сторонних фирм не являются в чистом виде средой **Visual Basic**, на этом мы и закончим обсуждение создания DLL общего назначения средствами **Visual Basic**.

Как видно из таблиц экспорта, декорирование имен функций (манглинг) средой **Visual Basic** не применяется.

Как получить таблицу экспортируемых имен?

Все экспортируемые имена DLL компоновщик (линкер) помещает в специальную таблицу в секции экспорта PE-файла. Каждый элемент в этой таблице содержит имя экспортируемой функции или переменной, а также относительный адрес этой функции или переменной (**RVA — relative virtual address**) внутри DLL-файла. Все списки имен сортируются по алфавиту.

Воспользовавшись утилитой **dumpbin.exe** с ключом **-exports** из состава **Microsoft Visual Studio**, можно увидеть содержимое секции экспорта DLL. Вот лишь небольшой фрагмент такого раздела для системной библиотеки **kernel32.dll**:

```
C:\WINDOWS\SYSTEM32DUMPBIN -exports kernel32.dll

Microsoft (R) COFF Binary File Dumper Version 6.00.8447
Copyright (C) Microsoft Corp 1992–1998. All rights reserved.

Dump of file C:\WINDOWS\SYSTEM32\kernel32.dll

File Type: DLL

Section contains the following exports for KERNEL32.dll

          0 characteristics
   3B7DDFD8 time date stamp Sat Aug 18 07:24:08 2001
       0.00 version
          1 ordinal base
        928 number of functions
        928 number of names

ordinal hint RVA      name

      1    0 00012ADA ActivateActCtx
      2    1 000082C2 AddAtomA
      3    2 0000D39F AddAtomW
      4    3 00065B2D AddConsoleAliasA
      5    4 00065AF6 AddConsoleAliasW
      6    5 00052A10 AddLocalAlternateComputerNameA
      7    6 000528FB AddLocalAlternateComputerNameW
      8    7 00060FFA AddRefActCtx
```

Остальной вывод для экономии места опущен.

Содержимое секции импорта выводится в четырех колонках:

- **ordinal** — это номер (ординал) функции — см. раздел «Вызов по имени и по ординалу»;
- **hint** — так называемая «подсказка», ускоряющая поиск имени в таблице экспорта;
- **RVA** — относительный адрес функции в файле (**relative virtual address**);
- **name** — имя экспортируемой функции.

Из приведенной информации видно, что библиотека **kernel32.dll** экспортирует 928 имен функций — немало.

Замечание:

В случае предпочтения продуктов компании **Borland** можно использовать утилиту **tdump** аналогичного назначения.

Декорирование имен

По ходу уже не раз встречалось понятие «декорирование имен». Настало время узнать об этом немного больше. Кроме того, здесь же будут приведены способы, позволяющие избавиться от этого замечательного процесса (потому что использование де-

корированных имен не всегда бывает удобным и необходимым).

Итак, что такое декорирование имен?

Декорирование (**decorate** — украшать, награждать знаками отличия) — процесс преобразования имен с целью сохранения информации об этом имени.

Microsoft сообщает, что это способ сохранения дополнительной информации о типе — например, в случае определения функции или члена класса, можно некоторым образом «восстановить» (при помощи обратного процесса) по этому «странному» виду ее текстовое описание.

Однако ни одна версия декорирования имен не имеет стандарта в своей основе (как, например, стандарт ISO/IEC 14482 определяет Стандарт языка C++). Поэтому декорированные имена от **Visual Studio 2.0** могут быть непонятными для **Visual Studio 4.0**. Алгоритм декорирования может изменяться от версии к версии (о чем любезно предупреждает документация конкретной среды разработки).

Кстати, декорирование имен — это также и один из способов обеспечить дополнительную уникальность экспортируемых имен.

Допустим, мы захотим экспортировать перегруженную функцию **getSum** с двумя и тремя параметрами. Посмотрим, что из этого получится:

```
Ordinal hint  RVA         name
    1      0   00001253    ?getSum@@YAHHH@Z
    2      1   00001334    ?getSum@@YAHHHH@Z
```

Как видим, две перегруженные функции различаются по именам. Первая из них — «старая» **getSum** с двумя параметрами, а вторая — это та же **getSum**, но уже с тремя параметрами. Таким образом, мы всегда легко сможем использовать обе экспортированные функции из DLL.

Как расшифровать то или иное декорированное имя? Здесь поможет еще одна утилита из состава **Visual Studio** под названием **undname.exe**. Она также является приложением консольного типа. На вход поставляются декорированные имена, а она пытается их привести в нормальный вид. Параметр **–f** (используется только для версии из набора VC++ 6.0) заставляет производить полное де-декорирование.

Рассмотрим результаты ее работы на основе полученной выше информации.

```
C:\...osoft Visual Studio\Common\Tools>undname.exe ?getSum@@YAHHH@Z

Microsoft(R) Windows NT(R) Operating System

UNDNAME Version 5.00.1768.1Copyright (C) Microsoft Corp. 1981-1998

>> ?getSum@@YAHHH@Z == getSum
```

А теперь запустим ее с ключом **–f** сразу для двух идентификаторов:

```
C:\...osoft Visual C:\...osoft Visual Studio\Common\Tools>undname.exe  -f  ?getSum@@YAHHH@Z ?getSum@@YAHHHH@Z

Microsoft(R) Windows NT(R) Operating System

UNDNAME Version 5.00.1768.1Copyright (C) Microsoft Corp. 1981-1998

>>?getSum@@YAHHHH@Z==int __cdecl getSum(int,int)

>>?getSum@@YAHHHH@Z==int __cdecl getSum(int,int,int)
```

Таким образом, на основе декорированного имени можно получить:

- точный тип возвращаемого значения;
- точные типы и количество входных параметров;
- правила вызова функции.

Разумеется, не всегда бывает удобным (особенно в случае использования явной загрузки DLL) использовать имена, аналогичные приведенным выше. Поэтому процессом декорирования можно управлять, но при этом вся сохраненная информация будет утеряна. Перегрузка функций также будет недоступна (что, в принципе, не так плохо, потому что другие языки могут не поддерживать такую перегрузку).

Разумно предположить, что если используемые версии и типы компиляторов при

сборке exe- и dll-файлов совпадают, то они (по умолчанию) будут понимать друг друга. И это действительно так. Таким образом, можно быть уверенным: подготовив DLL в Visual C++, а затем использовав DLL в проекте этой же среды разработки, что все пройдет гладко.

Поэтому в этом случае о декорировании имен можно даже не вспоминать... если, конечно, пишется не системная DLL.

Замечание:

По сообщениям из **MSDN** алгоритм декорирования, который применялся в компиляторе VC++ 2.0, был изменен в версии VC++ 4.2. Таким образом, lib-файлы, созданные каждым из компиляторов, будут непонятны для другого.

Посмотрим таблицу экспорта одной из таких системных DLL (**KERNEL.DLL**).

Это лишь небольшой список из находящихся в этой DLL девятисот сорока двух (!) экспортируемых функций. Как видно, имена построены «человеческим» способом, чтобы такая DLL могла быть использована любыми средствами разработки без особых трудностей, не заботясь об их «происхождении».

KERNEL.DLL

```
...Microsoft Visual Studio .NET\Vc7\bin\dumpbin.exe" /exports kernel32.dll
Microsoft (R) COFF/PE Dumper Version 7.00.9466
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file kernel32.dll

File Type: DLL

   Section contains the following exports for KERNEL32.dll

   00000000 characteristics
   3D6DE616 time date stamp Thu Aug 29 13:15:02 2002
       0.00 version
          1 ordinal base
        942 number of functions
        942 number of names

   ordinal  hint RVA         name
      1       0   000137E8   ActivateActCtx
      2       1   000093FE   AddAtomA
      3       2   0000D496   AddAtomW
      4       3   000607C5   AddConsoleAliasA
      5       4   0006078E   AddConsoleAliasW
      6       5   0004E0A1   AddLocalAlternateComputerNameA
      7       6   0004DF8C   AddLocalAlternateComputerNameW
      8       7   00035098   AddRefActCtx
      9       8              AddVectoredExceptionHandler
                             (forwarded to NTDLL.RtlAddVectoredExceptionHandler)
      10      9   00036909   AllocConsole
      11    A     000520CE   AllocateUserPhysicalPages
      12    B     0000DF51   AreFileApisANSI
      13    C     0000261A   AssignProcessToJobObject
      14    D     00060CCE   AttachConsole
...
```

Замечание:

Заботиться о происхождении все же приходится. Дело в том, что такие функции должны использовать стандартные правила вызова функции. Существует несколько различных стратегий:

- стратегия управления стеком:
  - стек очищает вызываемая функция;
  - стек очищает вызывающая функция;
- стратегия передачи параметров:
  - параметры передаются слева направо;
  - параметры передаются справа налево.

Стратегия вызова может задаваться явно при помощи нестандартных ключевых слов. Например, ключевое слово **__fastcall** заставляет передавать параметры через регистры процессора, а остаток помещать в стек справа налево. Вызываемая функция обязана очистить стек перед возвратом управления. В свою очередь, **__stdcall** подразумевает размещение всех параметров в стеке справа налево; вызываемая процедура также обязана очистить стек перед возвратом управления.

Несоответствие в правилах вызова функции немедленно приведет к краху приложения.

Для того чтобы DLL могла бы быть использована другими приложениями, необходимо позаботиться:

а) об отмене декорирования имен;

б) о соответствии правил вызова (calling conventions).

Функции WinAPI используют соглашение **__stdcall**.

По умолчанию среды **VC++** и **Borland C++ Builder** использует соглашения вызова **__cdecl**. Это позволяет, в частности, экспортировать функции с переменным числом параметров (подобно семейству функций **xprintf**).

Среда **Borland Delphi** по умолчанию использует соглашение **register**. В этом случае параметры передаются слева направо, вызываемая функция очищает стековую область; по возможности, параметры передаются через регистры процессора.

Изменить параметры соглашения можно:

1. Используя ключевые слова явного указания параметров соглашения (**__stdcall**, **__cdecl** и проч.):

VC++:

```
int __stdcall getSum(const int n1, const int n2);
```

Delphi:

```
function getSum(const n1, n2: integer): integer;
cdecl; external 'XDll6.dll';
```

2. Используя опции компилятора **/Gd**, **/Gr**, **/Gz**:

- **/Gd** — опция по умолчанию; используется **__cdecl** для всех функций, за исключением членов-функций классов C++ и функций, помеченных **__stdcall** или **__fastcall**.
- **/Gr** — используется **__fastcall** для всех функций, за исключением членов-функций классов C++ и функций, помеченных **__cdecl** или **__stdcall**. Для всех **__fastcall**-функций должны быть объявлены прототипы.
- **/Gz** — используется **__stdcall** для всех C-функций, за исключением функций с переменным числом параметров и функций, помеченных **__cdecl** или **__fastcall**. Для всех **__stdcall**-функций должны быть объявлены прототипы.

Замечание:

Для автоматической настройки ключей используйте **«Project Options->C/C++->Code Generation->Calling Conventions»**.

Из всего сказанного выше следует, что явные спецификаторы вызова функции отменяют действие опции компилятора.

VC++ 6.0:

Добавьте в **«Project Settings->C/C++-> Project Options»** нужное вам значение (например, **/Gz**).

VC++ 7.0:

Установите значение поля **«C/C++-> Advanced->Calling Conventions»**.

Отмена процесса «украшения» имен может быть произведена одним из описанных ниже способов:

1. Использование **extern «C» __declspec (dllexport)**.
2. Использование def-файла.
3. Использование директивы **#pragma**.
4. Использование настроек проекта.

Замечание:

Как вы поняли, на основе «украшенного» имени можно определить всю информацию относительно любого экспортируемого идентификатора. Почему нельзя использовать эту информацию без дополнительного использования h-файла? Причины просты:

а) нет стандартизации процесса декорирования — каждый декорирует, как хочет;

б) не все компиляторы поддерживают декорирование в силу пункта а).

Давайте вновь рассмотрим применение этих способов. По мере изложения материала мы будем приводить ссылки на соответствующие разделы, где работа с тем или иным вариантом изложена более подробно.

1. Использование **extern «C» __declspec (dllexport)**.

По ходу чтения статьи неоднократно упоминалось использование директивы **__declspec(dllexport)** для экспортирования имен. Оказывается, можно сделать немного больше, чтобы защитить имена от искажения. Для этого используется ключевое слово **extern «C»** совместно с использованием **__declspec**. При этом предотвращается искажение имен — так, как это делается в случае написания программы на языке C (не C++!). Но, соответственно, использовать эту директиву можно только в программах на C++. Поэтому обобщение для конструкции **__declspec (dllexport)** можно представить в таком виде:

```
#ifdef XDLL6_EXPORTS
  #ifdef __cplusplus
   #define XDLL_API extern "C" __declspec
   (dllexport)
  #else
   #define XDLL_API __declspec(dllexport)
  #endif // __cplusplus
#else
  #define XDLL_API __declspec(dllimport)
#endif // XDLL6_EXPORTS
```

Это позволяет использовать одну и ту же конструкцию не только в файлах проекта DLL и клиентских приложениях, но и в программах, написанных как на C++, так и на C. Ведь символ препроцессора **__cplusplus** определен только в проектах, написанных на языке C++, но не на языке C.

Замечание:

Эта техника не работает, если вы используете спецификаторы вызова функции (**calling conventions**) в явном виде.

Замечание:

В случае использования языка C искажения имен не происходит в любом случае. Так что это можно также считать еще одним способом избавления от декорирования имен.

Замечание:

Даже несмотря на это, компилятор **Borland** (**bcc32.exe**) все равно искажает имена — добавляет '_' (подчеркивание). Для того чтобы все-таки получить «правильное» имя, необходимо явным образом — в настройках среды или в опциях командной строки — указать компилятору, что добавлять символ подчеркивания не следует.

2. Использование def-файла.

Файл определений (def-файл) используется для дополнительного описания характеристик приложения. Он может состоять из различного набора секций. В случае работы с DLL нас особенно сильно заинтересует секция **EXPORTS**. Именно в ней описываются экспортируемые объекты DLL

(в качестве таких объектов могут выступать функции и переменные). Кроме того, вместе с указанием имени идентификатора могут быть использованы необязательные поля (**@ordinal**, **PRIVATE**, **DATA**) для указания дополнительных характеристик объекта. В случае обнаружения файла определений (процесс использования подробно описан в разделе «Различные способы экспорта») линкер пытается определить относительный виртуальный адрес функции по ее идентификатору для последующей записи требуемой информации в lib-файл. В случае обнаружения функции в раздел экспорта DLL помещается то имя, которое указано в def-файле. А именно это нам и нужно, чтобы окончательно избавиться от декорирования имен.

Замечание:

В общем случае def-файл позволяет даже «переименовать» конкретную функцию. В этом случае в разделе **EXPORTS** должна появиться запись примерно следующего содержания:

```
EXPORTS
   NewFunc = getSum
```

3. Использование директивы **#pragma**.

Директива **#pragma** предполагает явное управление настройками проекта. Такой подход также рассматривался в разделе «Различные способы экспорта». Как там было сказано, он обладает определенным недостатком, наличие которого сводит практически на нет его применение. Дело в том, что для избавления от декорирования необходимо знать точное декорированное имя объекта:

```
#pragma comment(linker, "/export:getSum=
?getSum@@YAHHH@Z")
```

В этом случае предполагается, что функции **?getSum@@YAHHH@Z** должна быть экспортирована также и под именем **getSum**. В связи с тем что имена функций слева и справа в достаточной степени «похожи», линкер будет экспортировать только одно имя (то, что указано в левой части).

4. Использование настроек проекта.

Настройка линкера **/export** позволяет явно указать (в качестве параметра) те функции, которые необходимо экспортировать. При этом указанное имя добавляется в раздел экспорта без каких-либо искажений. Использование такой техники изложено в разделе «Различные способы экспорта».

Замечание:

Как следует из всего написанного выше, проблемы декорирования — это проблемы совместимости с точки зрения использования. С точки зрения исполнения кода возникают уже совершенно другие проблемы (связанные, например, с несоответствием соглашений вызова, отличиями в моделях управления памятью и проч.).

Экспорт/импорт по имени и по порядковому номеру

Еще раз взглянем на прототип функции **GetProcAddr**.

Как видно, в описании функции **GetProcAddr**, в случае использования метода явной загрузки, указатель на функцию можно получить, задав в качестве второго параметра этой функции не ее имя, а порядковый номер.

В случае получения адреса функции по ее названию (как это происходит в боль-

GetProcAddr

```
FARPROC GetProcAddress(
   HMODULE hModule,   // HMODULE спроецированной DLL
   LPCSTR lpProcName  // название функции в формате ANSI или наименование переменной.
                      // Также вместо названия функции может быть указан ее порядковый номер
);
```

шинстве случаев) программист использует определенное имя (которое находится в таблице экспорта данной библиотеки) для получения адреса вызываемой функции.

При этом в программе используется код, подобный показанному ниже:

```
...
  //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);
...
```

В этом случае по заданному значению **hModule** происходит обращение к таблице экспорта DLL, которая предварительно была спроецирована на адресное пространство вызывающего процесса. В этой таблице находятся символьные идентификаторы всех экспортируемых объектов, каждому из которых соответствует определенное значение RVA (**relative virtual address**).

Замечание:

Одна и та же функция может иметь несколько синонимов для ее вызова. В этом случае в поле RVA будут одинаковые значения.

Методом построчного сравнения находится необходимый идентификатор, по нему получается соответствующее значение RVA. Это значение суммируется с базовым адресом, по которому DLL была спроецирована ранее — виртуальный адрес требуемой функции получен, теперь его можно использовать в программе для обращения к запрошенной функции.

Как видим, если функций в библиотеке достаточно много, процесс поиска необходимого идентификатора может занять относительно продолжительное время. Впрочем, как правило, приложение всего лишь несколько раз за свой жизненный цикл обращается к функции **GetProcAddress** —

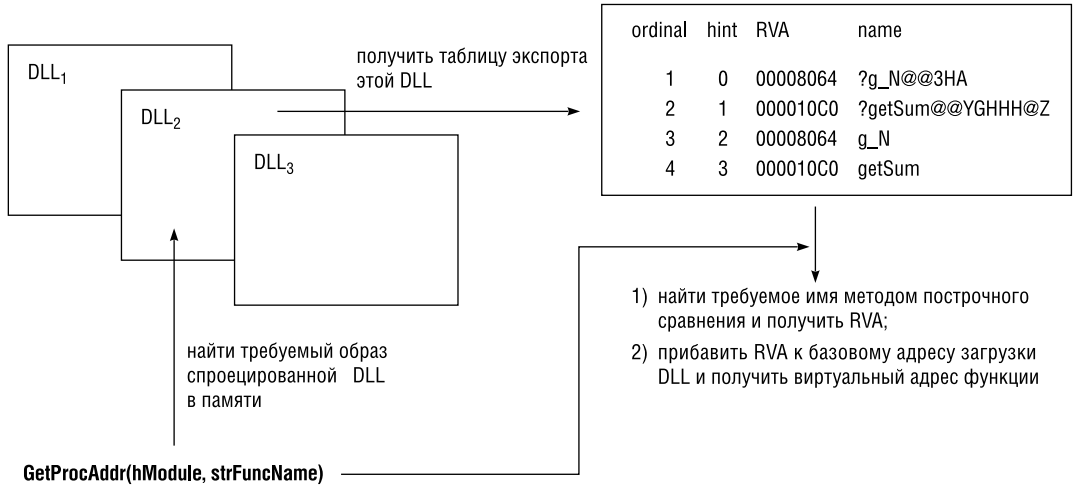

Рис. 7. Алгоритм получения виртуального адреса функции

один раз полученный виртуальный адрес может использоваться многократно.

Замечание:

Система оптимизирует процесс поиска необходимого имени, используя специальное поле **«hint»**, сохраняемое в таблице экспорта.

Чтобы сократить время поиска, можно воспользоваться так называемым порядковым номером функции в таблице экспорта. Этот порядковый номер можно наблюдать в выводе утилиты **dumpbin** в столбце **«ordinal»**:

```
Ordinal hint  RVA         name
   1      0   00008064    ?g_N@@3HA
   2      1   000010C0    ?getSum@@YGHHH@Z
   3      2   00008064    g_N
   4      3   000010C0    getSum
```

Этот номер служит для прямого обращения к необходимому адресу RVA — следовательно, никакой перебор и построчное сравнение не требуются. В этом случае мы используем примерно такой код:

```
...
  //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,
MAKEINTRESOURCE(4));
  //проверяем успешность получения адреса
_ASSERT(pGetSum != NULL);

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

Макрос **MAKEINTRESOURCE** используется для преобразования порядкового номера в строковый формат и передачи его в функцию **GetProcAddress**.

Вроде бы все замечательно — получили указатель на функцию с минимумом затрат и без особых проблем.

Опасность использования данного метода заключается в том, что этот номер может меняться от версии к версии. Скажем, мы решили расширить функциональность нашей библиотеки, добавив новую функцию **getDiff**.

Файл XDll.h

```
...
XDLL_API int getDiff(const int n1, const int n2);
...
```

Файл XDll.cpp

```
...
////////////////////////////////////////////
// getDiff function
int getDiff(const int n1, const int n2)
{
   const int n = n1 - n2;
   g_N = n;

   return n;
}
...
```

Замечание:

В отличие от идентификатора экспортируемой переменной **g_N** и экспортируемой функции **getSum**, для функции **getDiff** мы определяем только одно (в данном случае — декорированное) имя. Как добавить альтернативные имена для экспортируемых идентификаторов (без декорирования), подробно рассказано в разделе «Декорирование имен».

И что мы видим?

```
Ordinal hint  RVA         name
   1      0   000A9020    ?g_N@@3HA
   2      1   00033D42    ?getDiff@@YAHHH@Z
   3      2   0003350E    ?getSum@@YAHHH@Z
   4      3   000A9020    g_N
   5      4   0003350E    getSum
```

А видим мы следующее. В таблице экспорта появился новый идентификатор, который занял второй порядковый номер. В этом случае обращение по порядковому номеру 4 вновь выдаст нам некоторый адрес, но это будет уже адрес другого экспортируемого идентификатора (в данном случае — **g_N**). И об этом мы никак не узнаем, пока наша программа замечательно не «рухнет»

Замечание:

Обрушение программы произойдет в случае несовпадения интерфейсов вызываемых функций. Если же функции имеют совпадающие прототипы, значит, программа всего лишь будет выдавать неправильные результаты.

Таким образом, при использовании записи вида **MAKEINTRESOURCE(4)** вместо указателя на функцию **getSum** мы получим указатель на экспортированную переменную **g_N**. Что случится с программой при попытке вызова функции по этому указателю — догадаться несложно.

Если же использовать имя функции для получения ее виртуального адреса, то никаких проблем с расширением функциональности библиотеки не возникнет — мы всегда точно получим адрес именно той функции, адрес которой мы ждем получить. Если же запрошенной функции не окажется в таблице экспорта библиотеки, функция **GetProcAddress** вернет нам значение **NULL**, что будет сигналом об отсутствии этой функции в списке экспортируемых идентификаторов.

Подытоживая сказанное выше, еще раз отметим, чем эти способы различаются, в чем их преимущества и недостатки.

Случай использования имени функции:

- небольшое снижение быстродействия в связи с поиском и сравнением заданного имени функции в таблице IAT (**import address table**);
- нет проблем использования функции в случае расширения функциональности библиотеки — каждому имени соответствует указатель на функцию, связанный с этим именем.

Случай использования порядкового номера:

- быстрый поиск в таблице IAT — порядковый номер однозначно определяет смещение, по которому находится требуемый адрес вызываемой функции;
- проблемы поиска функции в случае расширения функциональности библиотеки — при изменении версии библиотеки у пользователя не может быть твердой уверенности, что получен адрес именно той функции, вызов которой запланирован в программе.

Если функциональность библиотеки изменяться в дальнейшем не будет, то можно смело использовать способ получения адреса функции по ее порядковому номеру. В случае же если вероятность изменения библиотеки в дальнейшем не нулевая, уверенности в правильности работы программ, использующих эту DLL посредством получения необходимых адресов функции через порядковые номера, не может быть никакой!

Как подключить к своему проекту чужую DLL?

Подключить к своему проекту чужую DLL можно двумя способами — неявной загрузкой (**implicit linking**) — для этого потребуется просто подключить к проекту соответствующий *.lib-файл или же явной загрузкой — вызовом функций **LoadLibrary** и **GetProcAddress**, с последующим вызовом функции **FreeLibrary**.

При использовании неявной загрузки DLL необходимо только подключить к проекту требуемый lib-файл и объявить необходимые функции (или переменные) как импортируемые.

Как известно, lib-файл, поставляемый вместе с файлом динамической библиоте-

ки, содержит всю необходимую линкеру информацию для автоматического (неявного) связывания имен вызываемых функций с соответствующими RVA (**relative virtual address**). Например, для уже изученной нами функции **getSum** из библиотеки **XDll6** необходимо написать:

```
extern "C" __declspec(dllimport) int getSum
(int, int);
```

Дальнейшее использование объявленной таким образом функции (в приведенном примере — функции **getSum**) не отличается от обычного. Всю необходимую работу по связыванию вызовов с кодом используемой DLL компоновщик (линкер) выполнит самостоятельно.

Строго говоря, даже объявление импортируемой функции с модификатором **__declspec(dllimport)** не обязательно — линкер все равно правильно выполнит всю необходимую работу по связыванию; однако компилятор сгенерирует более эффективный код, если ему заранее будет известно, что искомая функция импортируется из DLL.

Замечание:

При использовании неявной загрузки обратите внимание, что форматы lib-файлов, генерируемых различными компиляторами (в частности, компиляторами **Borland** и **Microsoft**), различаются. Поэтому возможен такой поворот событий, когда имеющийся lib-файл компилятор «не понимает». Для разрешения этой проблемы необходимо будет создать def-файл и сгенерировать новый lib-файл, «понятный» компилятору.

При использовании явной загрузки DLL необходимо точно знать имя искомой функции, соглашение вызова и набор передаваемых ей параметров — как правило, эта информация доступна из заголовочных файлов или документации. Весь процесс распадается на три шага:

1. Вызов функции **LoadLibrary** и загрузка DLL.
2. Получение адреса экспортируемой функции (или переменной) вызовом функции **GetProcAddress**.

Выполнение кода...

3. Вызов функции **FreeLibrary** и выгрузка DLL.

Хотя функция, используемая для получения адреса, и называется **GetProcAddress**, она может возвращать не только адрес экспортируемой функции, но и адрес любого экспортируемого объекта (например, переменной). Разумеется, в этом случае также необходимо использовать явное приведение типа.

По завершении использования библиотеки необходимо закрыть дескриптор модуля и выгрузить библиотеку вызовом функции **FreeLibrary**. Не забывайте также проверять результат вызова **FreeLibrary** — функция вернет **FALSE**, если вызов завершился неудачно (например, если дескриптор по ошибке был уже закрыт где-то в другом месте).

И последнее замечание. Разумеется, если используемая среда поддерживает отложенную загрузку (и необходимо использовать именно этот способ работы с DLL), то также можно использовать отложенную загрузку и при работе с чужой библиотекой.

Список литературы

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

Редактор: В следующих номерах журнала мы продолжим публикацию материала об особенностях реализации и использования динамических библиотек.