

Глава 18. Динамически компоуемые библиотеки.....	411
18.1. Что такое DLL?.....	412
18.2. Простой пример создания DLL.....	416
18.3. Замечания по использованию библиотек.	419
18.4. Хранения формы в динамических библиотеках.	420
18.5. Немодальные окна в динамических библиотеках.....	423
18.6. Явная загрузка библиотек.	426



Глава 18. Динамически компоуемые библиотеки.

Ты уже наверно много раз слышал заветное выражение «динамически компоуемые библиотеки. Пора познакомиться с ними поближе. В этой главе я постараюсь тебе дать всю необходимую информацию по ним, мы напишем несколько примеров, и ты всё прекрасно увидишь на практике.

В отличии от остальных глав, в этом вступительном слове я больше ничего говорить не буду, потому что если затронуть тему описания предназначения DLL файлов, то тема растянется в долгий разговор, поэтому я посвящу этому первую же часть этой главы.





18.1. Что такое DLL?

Программисты всех стран уже более 30 лет борются с проблемой многоразового использования однажды написанного кода. Так уж повелось, что 30-50% кода в простых офисных приложениях схожи между собой или решают одни и те же задачи. Ни один программист не захочет каждый раз снова писать один и тот же код. Как хорошо, когда можно использовать один раз написанный код многократно

Я сам не люблю в каждой новой программе писать одно и то же. Как хорошо, когда написал какой-то универсальный код, а потом только используешь его.

Решение №1.

Самым первым решением этой проблемы стал модульное программирование. Ты пишешь какой-то кусок кода, оформляешь его в виде модуля, а потом просто используешь его в своих программах. Все прекрасно и удобно, а главное, что все довольны. Теперь не надо каждый раз выдумывать велосипед, просто добавил к своей программе определенный модуль и без проблем используй код, когда-то написанный тобой или кем-то другим.

Казалось, что это было самое простое и самое эффективное решение. Но все было прекрасно, пока не появилась многозадачность. Вот тут программисты и простые пользователи заметили, что еще не все так эффективно и полно места, куда можно приложить свои руки для оптимизации выбранного решения.

Проблема №1.

Давай представим ситуацию, когда один добрый человек написал прекрасный модуль размером в 1 мегабайт. Другой добрый человек решил воспользоваться его возможностями и подключил к своей программе. Модуль и программа слились в одно целое. Вроде все нормально, но я же сказал, что программа и модуль слились в одно целое. Это значит, что размер результата увеличился на размер модуля, т.е. на 1 мегабайт. Не фига себе пельмень!!!

А теперь представь, что другой чел написал другую утилиту с использованием этого модуля.... Его программа тоже увеличилась на 1 мегабайт. Получается, что на винте пользователя хранится две программы, в которых по 1 мегабайту кода одинаковых. И кому это нужно?

Ну, конечно же, на счет модуля в 1 мегабайт я немного преувеличил. В те времена даже 100 кило модуль тяжело было найти. Но надо учитывать, что и винты тогда были не бесконечные. Тогда крутым винтом считался диск в 20 мегабайт. Это тебе не нынешние десятки гигабайт на одной пластине. Я сам застал такие машины только на первом курсе института, а это было почти 10 лет назад.

Проблема №2.

Пока существовали только однозадачные операционные системы, проблема с излишней растратой дискового пространства была единственной. Но как только задумались о многозадачности и в мыслях Билла Гейтса появились идеи создать Windows, так сразу возникла другая проблема.... Представь себе ситуацию, когда ты запускаешь обе

этих программы одновременно. При старте любой код грузится в оперативную память и только потом выполняется. Так что получается, что обе программы загрузят в память один и тот же код. Вот это уже абсолютно никому не нужно.

Это только в последнее время память подешевела в несколько раз, и теперь лишние сто кило погоды не сделают. А раньше она стоила достаточно дорого, и люди боролись за каждый байтик потом и кровью. Но если ты думаешь, что если поставить в свой компьютер 500 мегабайт оперативной памяти и проблема уйдёт сама собой, то ты крупно ошибаешься.

Хотя память и дешевая, программы от этого меньше не станут. Если посмотреть на запросы той же Windows 2000, то сразу понятно, что эти 500 мегабайт это только капля в море. Самая простая ОС Windows 2000 Professional отнимет от них около 128 мегабайт. Это что же там такое натворили, что Windows 2000 Server просит для нормальной работы минимум 256 мегабайт? А если учесть, что ещё недавно чипсеты не поддерживали памяти более 512 мегабайт (это сейчас можно от 2 до 3 гигабайт вставить), то о нормальной одновременной работе Windows 2000 Server + 3D Studio Max + MPEG4 можно забыть. Они всю память отберут, как термиты за пять сек.

Решение №2.

И вот тут было найдено вполне солидное решение: не стыковать модули с основной программой, а сохранять их в отдельный файл и пусть любая программа загружает его по мере надобности. Сказали, сделали. Так появились библиотеки DLL, что означает *Dynamic Link Library (DLL)*. Это библиотеки, которые подключаются к программе динамически. В них можно хранить исполняемый код в виде процедур или функций, ресурсы программы, графику или даже видео ролики.

Вот так. Теперь программа не увеличивалась на размер модуля при компиляции, а просто загружала код из DLL файла в память и использовала его. Если одна программа уже загрузила DLL, то следующая не будет уже делать этого. Она воспользуется уже загруженной версией. Таким образом, экономится не только диск, но и оперативная память, которой, как и денег, много не бывает.

Сейчас уже DLL - это не просто динамически подгружаемая библиотека. Ты наверно уже не раз слышал про компоненты *ActiveX*. Они так же могут быть выполнены в виде осх или dll файлов. Да оно и понятно, *ActiveX* используются сейчас достаточно много и занимают места в несколько раз больше чем самая большая DLL библиотека. Так что единственный и нормальный выход экономить место винта и памяти это засунуть *ActiveX* в динамически подгружаемую библиотеку. Хотя это уже не та DLL, но всё же работает по тем же принципам.

У динамических библиотек есть единственный недостаток - на ее загрузку тратится лишнее время. Если бы код, находящийся в DLL был бы скомпонован с программой, то он грузился бы намного быстрее. Зато если библиотека уже загружена другой программой, то она появляется намного быстрее. Не веришь? Отложи сейчас книгу и возьми в руки секундомер. Теперь запусти Word или Excel. Засеки сколько времени будет проходить загрузка. Теперь закрой эту программу и запусти ее снова. Она появится на экране практически моментально. Это потому что после выхода из программы, DLL файл не выгружается из памяти. Это происходит только тогда, когда операционной системе не хватает памяти и ни одна из программ не использует в данный момент эту библиотеку.

А теперь представь себе, что такое Word!!! Представил? Это и текстовый редактор, и проверка орфографии, и построитель диаграмм, редактор формул и куча еще всякой всячины. Представь себе, что было бы, если все это засунуть в один файл? Нет, ты это не можешь представить. Это был бы один запускной файл размером в 30-50 мегабайт.

А теперь вспомни, что я тебе сегодня говорил: перед запуском, программа загружается в память. Представляешь теперь, сколько бы грузился Word? А сколько

памяти он съёл бы? А тебе ведь и половина его возможностей абсолютно не нужна. И зачем же их грузить в память?

А при использовании динамических библиотек в запуском файле находится только самое основное, а дополнительные возможности подгружаются по мере надобности из DLL-файлов. Например, когда стартует Word, то загружается только модуль текстового редактора. Когда ты выбрал редактор формул или объект WordArt, то Word подгружает из dll файла код выбранного объекта и выполняет его. Таким образом, суммарная скорость загрузки уменьшается, причем очень даже значительно.

Ещё одно большое преимущество динамических библиотек – при их использовании код программы разбивается на несколько файлов (зависит от количества DLL файлов). Допустим, что в одной из функций находящейся в DLL оказался код с ошибкой. В этом случае не надо обновлять всю программу, а достаточно передать всем пользователям только этот DLL файл, и программа получит необходимые обновления.

У динамических библиотек сплошные преимущества и только два недостатка:

1. Код из DLL файла выполняется в том же участке памяти, что и основная программа. Поэтому программа и DLL используют один и тот же стек данных, что иногда накладывает свои ограничения. Например, DLL не может хранить глобальных переменных. Воспринимай динамические библиотеки просто как набор процедур и функций, которые могут хранить только локальные переменные.

2. Изначально динамические библиотеки были процедурными. Хотя сейчас умельцы умудряются использовать их для хранения объектов, но это очень неудобно. Но, несмотря на это, ActiveX (изначально объектные) могут храниться в файлах с расширением dll.

Но всё же динамические библиотеки получили широкое распространение и программисты используют их на каждом углу, когда надо и когда не надо. Никогда нельзя быть уверенным, что какой-то код уже больше никогда не понадобится. Всегда нужно рассчитывать на будущее.

Я надеюсь, что я тебя убедил в великих возможностях динамических библиотек. Это действительно так. Конечно же, ActiveX более продвинуты, но они требуют неудобной регистрации в системе (в реестре) и намного сложнее в программировании, а библиотеки пишутся достаточно просто и их достаточно только скопировать на другой компьютер, чтобы программа смогла её найти.

Из чего же сделан Windows?

Все наверно помнят такую песенку: "Из чего же, из чего же, из чего же, сделаны эти мальчишки?". Глупейшая песня, и я со слезами на глазах вспоминаю, как я в лагере (я имею ввиду пионерский, а не концлагерь) распевал ее вместе с остальными пионерами. Ох, и веселые были времена. Жаль, что сейчас так не развлечешься. О чем это я? Ах да... Я хотел рассказать тебе, из чего состоит Windows.

Большинство думает, что Windows - это все что находится в папке c:\Windows, а ее ядро - это win.com. В какой-то степени это так, но не совсем. Ядро ОС Windows - это простой DLL файл, а если быть конкретнее, то это Kernel32.dll. При старте Windows эта библиотека загружается в память в единственном экземпляре, и любая программа может обращаться к содержащемуся в ней коду и использовать его в своих целях. В этой библиотеке расположены API функции, предназначенные для распределения памяти и многое другое. Мы эти функции не вызываем напрямую, потому что Delphi прячет этот сложный процесс от нас, но иногда тебе может понадобиться воспользоваться ими. Так что помни, если ты выделяешь память, то в этот момент используется Kernel32.dll.

Точно так же, за вывод графики в Windows отвечает GDI32.DLL, которая так же загружается при старте в единственном экземпляре. Все функции для работы с графикой находятся в этой библиотеке. Есть и ещё одна библиотека, User32.dll, которая отвечает за

создание окон и обработку сообщений. Все эти три библиотеки составляют ядро ОС Windows.

В Windows очень много недостатков, но динамические библиотеки это достаточно гениальное решение многократно используемого кода.

Графические движки.

Любой игрок обязан знать про существование OpenGL. Что это такое? Какой-то пакет программ? Какой-то SDK для создания графики? Ничего подобного, это всего лишь две динамические библиотеки `opengl.dll` (`opengl32.dll`) и `glu.dll` (`glu32.dll`).

Что такое DirectX? Это графическая библиотека, которая состоит из `DirectDraw`, `DirectInput`, `DirectMusic`, `DirectPlay` и так далее. Все это не что иное, как простые динамически подгружаемые библиотеки. `DirectDraw` это `Ddraw.dll`, `DirectInput` это `Dinput.dll`, `DirectMusic` это `Dmusic.dll` и так далее. Хотя DirectX это не простые библиотеки – это библиотеки созданные на основе технологии COM (та же технология, что и ActiveX).

Любые игровые движки выполнены в виде динамически загружаемых библиотек, потому что их использование очень простое и удобное для любого программиста.

Давай подведём итог тому, что уже было сказано. Динамические библиотеки практически ничем не отличаются от EXE файлов. Это такой же скомпилированный код, только он не может запускаться самостоятельно, потому что в библиотеке нет точки входа (точки, с которой начинает своё выполнение любая программа) В dll файлах хранятся только процедуры и функции, которые можно вызывать из других программ.

Чаще всего динамические библиотеки имеют расширение `dll`, но можно установить и любое другое расширение или вообще убрать его. Библиотеки имеют свои разновидности, например, драйверы – это тоже динамические библиотеки и им принято давать расширение `drv`. Возможно так же использование расширения `sys` для системных файлов. Так что операционная система не накладывает ограничений на расширения динамических библиотек, главное, чтобы тебе было понятно и удобно работать.

Когда одно приложение загружает библиотеку, то она загружается в глобальную память, а потом только проецируется в адресное пространство программы. Это значит, что программа будет видеть функции библиотеки как родные, хотя они расположены совершенно в отдельном адресном пространстве.

Операционная система Windows гарантирует, что в любой момент будет загружена в память только одна версия `dll` файла. Если две программы обращаются к одной и той же библиотеке, то в памяти будет находиться только одна копия `dll` файла.

Говоря о библиотеках (`dll`) я всё время говорил о динамических библиотеках. Но существуют и статические варианты библиотек. Чем они отличаются? В принципе, библиотека одна и та же, поэтому термин статической DLL я считаю неправильным (хотя иногда встречаю в литературе). Но всё же я вынес это название в заголовок, чтобы показать тебе на ошибку.

Библиотеки `dll` всегда динамические и создаются они с целью динамической загрузки находящихся в них ресурсов. Но не смотря на это, многие компиляторы позволяют присоединять код `dll` статически. В этом случае при компиляции программы код или данные, находящиеся в `dll`, становятся неотъемлемой частью исполняемого файла. В этом случае программа и библиотека становятся как единое целое. Это очень удобно, когда библиотека небольшая или тебе необходимо, чтобы программа состояла только из одного запускового файла. Здесь динамическая загрузка не подходит, и надеяться на существование библиотеки на машине клиента нельзя. Такая ситуация может возникнуть, когда ты хочешь показать клиенту демонстрационный файл твоей программы и удобно иметь только один запусковой файл, а не множество библиотек.

18.2. Простой пример создания DLL.

Так как DLL – это отдельный файл, то и создаётся он в Delphi как отдельный проект. Для создания новой динамической библиотеки нужно выбрать из меню *File* пункт *New* и затем *Other...* В окне создания нового проекта (рисунок 18.2.1) нужно выбрать на закладке *New* пункт *DLL Wizard*. Выбери этот пункт и нажми *OK*.

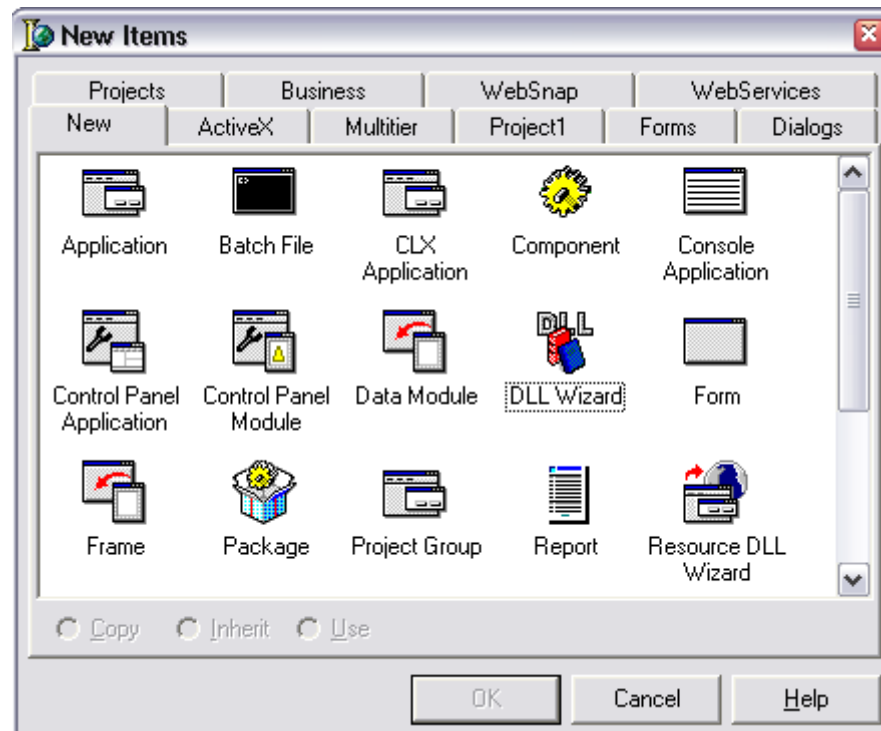


Рисунок 18.2.1 Окно создания нового проекта.

Несмотря на то, что мы выбрали *DLL Wizard* (слово *Wizard* говорит о том, что должен запускаться мастер), будет просто создан пустой проект с одним только модулем. Модуль будет содержать следующий текст (я убрал только комментарии):

```
library Project2;
```

```
uses
  SysUtils,
  Classes;
```

```
{$R *.res}
```

```
begin
end.
```

Если открыть менеджер проектов (Меню *View->Project Manager*), то в окне вообще не будет видно ни одного модуля. Это потому что код, который мы видели выше относится к самой библиотеки. Выбери из меню *File* пункт *Save All*, и тебе предложат сохранить только один проект и никаких модулей. Я сохранил проект под именем *FirsDLLProject*. Потом открыл файл проекта *FirsDLLProject.dpr* с помощью блокнота и увидел тот же самый код.

Теперь давай добавим в нашу библиотеку одну функцию с именем *Summ*. У этой функции будет два параметра в виде целых чисел, и возвращать она будет сумму этих чисел:

```
library FirsDLLProject;

uses
  SysUtils,
  Classes;

function Summ(X,Y:Integer):Integer; StdCall;
begin
  Result:=X+Y;
end;

exports Summ;

{$R *.res}

begin
end.
```

Обрати внимание, что функция у нас объявлена не так как всегда. В конце строки объявления, после типа возвращаемого значения стоит ключевое слово **StdCall**. Оно говорит о том, что для вызова процедуры нужно использовать стандартный тип вызова.

Я тебе уже говорил, что все параметры, передаваемые в процедуры и в функции, передаются через стек. Если не указать ключевое слово **StdCall**, то параметры будут передаваться способом, заложенным фирмой *Borland*. Этот способ работает быстрее, но он не совместим со стандартными правилами. Если ты уверен, что к процедуре будут обращаться только программы скомпилированные компиляторами фирмы *Borland*, то можешь не ставить это ключевое слово. Но если библиотека будет выложена на всеобщее использование или к ней будут обращаться программы сторонних разработчиков, то желательно ставить **StdCall**, иначе у программистов на языках Visual C++ или других языках будут проблемы. Я сделал для себя правилом всегда ставить **StdCall**, потому что способ *Borland* даёт незначительный выигрыш.

В остальном, функция ничем не отличается от тех, что мы уже писали.

После описания функции идёт новое ключевое слово **exports**. После этого ключевого слова должно идти описания процедур, которые должны быть доступны внешним программам. Если нашу функцию *Summ* не описать в разделе **exports**, то мы её не сможем вызвать из внешней программы.

Теперь откомпилируй проект (нажми Ctrl+F9 или выбери из меню *Project* пункт *Compile FirsDLLProject*), чтобы создать нашу динамическую библиотеку. Можешь не пытаться запускать проект, потому что это библиотека, и она не может выполняться самостоятельно. Так что, единственное, что ты можешь увидеть – ошибку.

Теперь напишем программу, которая будет использовать написанную функцию из динамической библиотеки. Создай новый проект простого приложения (*File->New->Application*). На форму брось только одну кнопку и по её нажатию напиши следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
```



```
r:Integer;  
begin  
r:=Summ(10,34);  
Application.MessageBox(PChar(IntToStr(r)), 'Результат функции Summ');  
end;
```

В первой строчке я вызываю функцию *Summ* с двумя числовыми параметрами. Результат записывается в переменной *r*. Вторая строка всего лишь выводит окно с результатом.

Если ты попытаешься сейчас откомпилировать проект, то у тебя ничего не выйдет. Компилятор Delphi скажет, что он не знает такой функции *Summ*. Мы должны показать Delphi, что это за функция и где её искать.

Для начала покажем компилятору, что это за функция. Для этого в разделе *type*, после описания объекта *TForm1* (нашей главной формы) нужно написать следующую строку:

```
function Summ(X,Y:Integer):Integer;StdCall;
```


В принципе, это такое же объявление функции, которое описано в библиотеке, только здесь нет **begin** и **end** и самого кода процедуры. По этой строке Delphi узнаёт, что где-то существует такая функция *Summ*, у неё есть два параметра, и она должна вызываться стандартным вызовом.

Теперь нужно сказать компилятору, где же искать эту загадочную функцию. Для этого после слова **implementation** напиши следующий код:

```
function Summ; external 'FirsDLLProject.dll' name 'Summ';
```

Здесь написано, что есть такая функция *Summ*. После точки с запятой стоит ключевое слово **external**, которое говорит о том, что функция внешняя, не принадлежит программе. После этого слова указывается имя динамической библиотеки, где нужно искать функцию. Далее идёт ключевое слово **name**, которое означает, что функцию надо искать по имени. После этого ключевого слова указывается точное имя функции в библиотеке.

Вот теперь проект готов и его можно компилировать, запускать и проверять результат.

 На компакт диске, в директории \Примеры\Глава 18\FirstDLL ты можешь увидеть пример этой программы.

В нашем примере использовался вызов по имени функции. Когда программе нужно выполнить функцию *Summ*, то она просматривает все функции динамической библиотеки и ищет функцию с указанным именем. Это очень неэффективно и перед первым вызовом будет ощущаться большая задержка. Чтобы хоть немного ускорить процесс вызова таких функций можно использовать индексы. Каждой функции в библиотеке может быть назначен индекс, и при вызове можно указывать его.

Давай подкорректируем наш пример на индексы. Открой проект динамической библиотеки *FirsDLLProject.dpr*. Найди ключевое слово **export** и напиши там такой код:

```
exports Summ index 10;
```

После имени функции стоит ключевое слово **index** и числовой индекс функции. Я люблю нумеровать свои функции, начиная с 10. Этой я дал десятый индекс (ты можешь попробовать другое число. Индексы и имена должны быть уникальными!!! Вот несколько примеров:

```
exports
  Func1 index 10 name 'Fun',
  Func2 Insert,
  Func3 index 11,
  Func4 index 11, //Ошибка, такой индекс уже существует
  Func5 name 'Don';
```


В объявлении последней процедуры я явно использовал ключевое имя **name**, чтобы указать экспортной функции новое имя. Теперь внутри библиотеки эта функция реализована как *Func5*, но внешние приложения должны обращаться к ней по имени *Don*.
Объявлять можно и так:

```
exports Func1 index 10 name 'Fun',
exports Func2 Insert,
exports Func3 index 11,
```

Перекомпилируй проект, чтобы изменения вошли в силу (нажми Ctrl+F9).
Теперь возвращаемся в проект, где мы используем функцию. В разделе **implementation** подправляем описание нашей функции:

```
function Summ; external 'FirsDLLProject.dll' index 10;
```

Теперь вместо ключевого слова **name** стоит слово **index** и тот же номер.
Запусти проект и убедись, что он работает корректно.

 На компакт диске, в директории \Примеры\Глава 18\IndexName ты можешь увидеть пример этой программы.

18.3. Замечания по использованию библиотек.

Когда я сказал, что DLL файлы нельзя запускать, то я и соврал, и нет. В принципе, библиотеки действительно нельзя запускать, но Delphi может сделать это. Открой нашу библиотеку и выбери из меню *Run* пункт *Parameters*. Перед тобой откроется окно, как на рисунке 18.3.1.

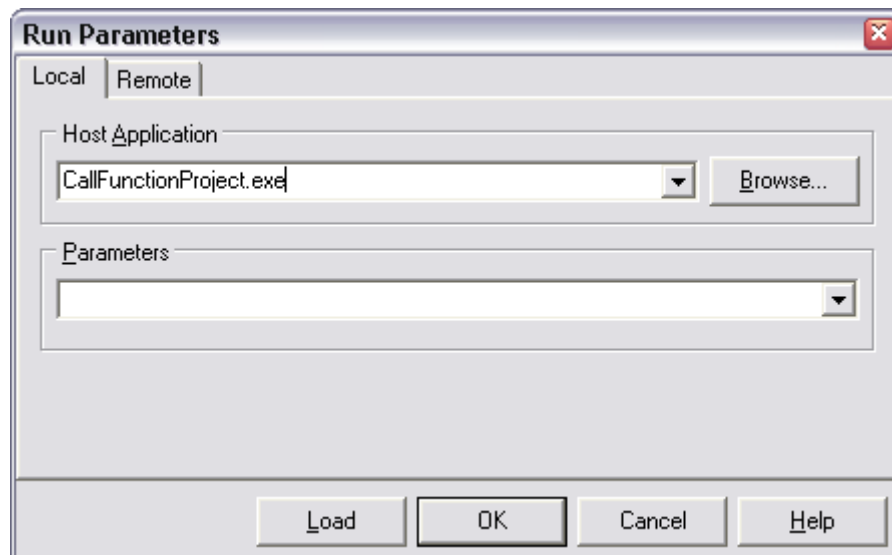


Рисунок 18.3.1 Окно параметров запуска программы.

В строке *Host Application* нужно указать имя приложения, которое умеет загружать библиотеку. Теперь попробуй запустить проект (клавиша F9). Запустить указанная программа.

Зачем нужен этот способ? Если ты попытался запустить программу, и она показала ошибку в коде, где вызывается функция из динамической библиотеки, то можно попытаться запустить библиотеку таким образом. Если снова произойдет ошибка, то Delphi покажет строку с ошибкой.

Небного позже я покажу тебе, как можно отлаживать программы, выполнять их по шагам. Тогда ты сможешь узнать, что таким образом можно отлаживать и динамические библиотеки. А именно, они могут выполняться в пошаговом режиме (построчно) и ты будешь контролировать весь процесс выполнения.

Пока что я о библиотеках говорил достаточно много хорошего, но не сказал самого главного. Функции и процедуры из динамической библиотеки не могут прямо влиять на ход основной программы. Это значит, что мы не можем получить доступ к окнам основной программы, изменить какие-то переменные или ещё чего-нибудь.

Функции библиотеки – как бы изолированы от всего остального, хотя и выполняются в одном адресном пространстве с основной программой. Они могут использовать только переданные параметры, а результат работы возвращать в качестве результата работы функции.

Имена библиотек пиши полностью, вместе с расширением. Без расширения DLL может быть не найдена в Windows NT/2000/XP, хотя в Windows 98 всё будет работать нормально.

Обязательно соблюдай индексы и параметры процедуры, иначе могут возникнуть ошибки. Лучше лишний раз проверить, чем потом долго искать опечатку.

18.4. Хранения формы в динамических библиотеках.

Теперь я хочу показать, как можно хранить в динамических библиотеках целые окна. Очень удобно, когда редко используемые окна, находятся в динамической библиотеке. В этом случае основной файл очень сильно разгружается от лишнего кода.

Ещё одно преимущество такого кода – библиотека может использовать для вывода информации на экран своё окно. Как я уже сказал, из dll файла нельзя получить доступ к переменным и данным основной программы. Это значит, что из DLL файла нельзя ничего вывести в окна основной программы. Но библиотека может создать собственное окно и использовать для вывода необходимых данных именно его.

Итак, создавай новую DLL библиотеку и сохрани её под именем *ProjectDLL*. Теперь добавим к нашей библиотеки одну экспортную процедуру *ShowAbout*:

```
library ProjectDLL;  
  
uses  
    SysUtils, Classes;  
  
{$R *.RES}  
  
exports ShowAbout index 10;  
  
begin  
end.
```

Я добавил только одну строку *exports ShowAbout index 10*; . У нас будет только одна процедура *ShowAbout* с индексом 10. Эта процедура будет показывать окно «О программе».

Теперь щёлкаем File->New Form , чтобы создать новую форму. Нарисуй на ней что-нибудь, можно даже то, что сделал я (рисунок 18.4.1).

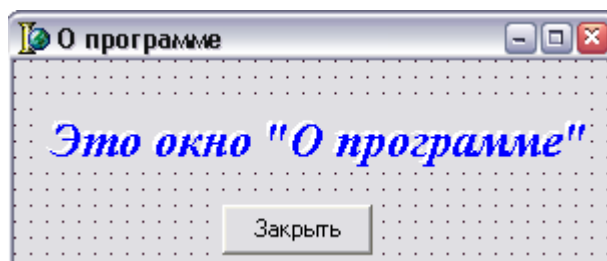


Рисунок 18.4.1. Окно «О программе».

Переходи в текст модуля. В разделе *var*, после объявления формы опиши процедуру *ShowAbout*:

```
var  
    Form1: TForm1;  
    procedure ShowAbout(Handle: THandle);export;stdcall;
```

Опять присутствует ключ **export** и добавлен ещё **stdcall**, указывающий на обязательность использования стандартного вызова процедуры.

Теперь напишем саму функцию после ключевого слова **implementation** и ключа *{\$R *.DFM}*:

```
procedure ShowAbout(Handle: THandle);
```

```

begin
//Установить указатель на приложение
Application.Handle := Handle;
//Создать форму
Form1:= TForm1.Create(Application);
//Отобразить
Form1.ShowModal;
//Очистить
Form1.Free;
end;

```

Эта процедура получает в качестве параметра указатель на главное приложение. В первой строке я устанавливаю этот указатель в свойство *Handle* объекта *Application*. Этот объект хранит настройки всего приложения, и этим присваиванием мы связали оба приложения.

Во второй строке кода я создаю окно *TForm1.Create(Application)*, в результате чего мне будет возвращён указатель на это окно. Результат я сохраняю в переменной *Form1*. Эта переменная объявлена в разделе **var** проекта.

Следующей строкой я отображаю модально созданное нами окно. Как только оно закроется, будет выполнена последняя строка кода этой процедуры, а именно, окно будет уничтожено из памяти и процедура закончит своё выполнение.

В процедурах DLL библиотек будь более внимателен к высвобождению памяти. По моей практике могу сказать, что ошибки в библиотеках переносятся программами более критично, потому что тут основная программа практически бессильна.

Откомпилируй библиотеку (*Ctrl+F9*) и DLL-файл готов. Можно закрывать этот проект (*File->Close All*) и создавать новое приложение, из которого мы будем вызывать созданную в библиотеке процедуру (*File->New Application*).

В новом проекте переходим в текст формы и объявляем функцию *ShowAbout*:

```

unit Unit2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

procedure ShowAbout(Handle: THandle)stdcall;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  procedure ShowAbout;external 'ProjectDLL.dll' index 10;

implementation

```

Обрати внимание, что первое описание процедуры я написал не в разделе **type**, а до него:


```
procedure ShowAbout(Handle: THandle)stdcall;
```

Это не является ошибкой, и ты можешь выбрать любой из этих способов. Я чаще всего объявляю внешние процедуры до раздела **var**, чтобы их потом легче было найти.

Теперь ставим на форму кнопочку и пишем по её событию *OnClick* следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  ShowAbout(Handle);  
end;
```

Запусти пример и убедись в том, что пример работает корректно. Как видишь окно не так уж трудно засунуть в библиотеку, и в нём могут быть свои события и свои процедуры и функции. К тому же окно может создавать дочерние окна по отношению к себе и той же динамической библиотеки.

 На компакт диске, в директории \Примеры\Глава 18\Form ты можешь увидеть пример этой программы.

18.5. Немодальные окна в динамических библиотеках.

В предыдущем примере я поместил в библиотеку модальное окно. А что, если тебе понадобится показать немодальное окно? Ведь мы показываем окно и по его закрытию должны освободить память. А как узнать, что окно закрыто? Некоторые ленятся и просто не освобождают память, выделенную под окно. Но это не правильно и просто глупо, потому что показать немодальное окно не намного сложнее.

Давай откроем предыдущий пример и подкорректируем его. Для начала нужно добавить одну экспортную процедуру *FreeAbout* с индексом 11. Теперь у нас будет экспортироваться две процедуры:

```
exports ShowAbout index 10;  
exports FreeAbout index 11;
```

Теперь переходим в модуль *Unit1*, где у нас находится форма динамической библиотеки. Процедуру *ShowAbout* я превращаю в функцию, которая будет возвращать значение типа *LongInt*. В качестве возвращаемого значения будет идентификатор окна, по которому мы потом будем его закрывать.

Ещё нужно добавить процедуру *FreeAbout* с одним параметром типа *LongInt*.

```
function ShowAbout(Handle: THandle):LongInt;export;stdcall;  
procedure FreeAbout(FormRef: LongInt);export;stdcall;
```

Как я уже говорил, динамические библиотеки не могут хранить переменных. Именно поэтому после создания окна мы должны вернуть идентификатор основной программе, чтобы она хранила эту переменную. Когда нужно будет закрыть окно, мы передадим этот идентификатор библиотеке, и она освободит память, выделенную под окно.

Теперь посмотрим на реализацию функции *ShowAbout*:

```
function ShowAbout(Handle: THandle):LongInt;
begin
  Application.Handle := Handle;
  Form1:= TForm1.Create(Application);
  Form1.Show;
  Result:=LongInt(Form1);
end;
```

Здесь всё осталось также, за исключением последней строчки. Если раньше мы освобождали память, то сейчас возвращаем окно *Form1* приведённую к типу *Integer*. Если бы мы тут вызвали метод *Free*, то окно сразу же после появления закрылось бы.

Теперь посмотрим на процедуру *FreeAbout*:

```
procedure FreeAbout(FormRef: LongInt);
begin
  if FormRef>0 then
    TForm1(FormRef).Free;
end;
```

В этой процедуре мы сначала проверяем, если переменная *FormRef* (идентификатор окна) больше нуля, то окно можно уничтожить, иначе оно могло быть уже уничтожено. Во второй строке я вызываю метод *Free* нашего окна. Так как переменная *FormRef* – это числовая переменная и у неё нет методов, то мы должны перевести её обратно к объекту - *TForm1(FormRef)*.

Теперь подкорректируем проект, который использует DLL файл. Для начала подправь объявления процедур библиотеки. Перед разделом **type** напиши следующее:

```
function ShowAbout(Handle: THandle):LongInt;stdcall;
procedure FreeAbout(FormRef: LongInt);export;stdcall
```

В разделе **var** пишем следующее:

```
function ShowAbout;external 'ProjectDLL.dll' index 10;
procedure FreeAbout;external 'ProjectDLL.dll' index 11;
```

Всё это уже должно быть знакомо и не должно вызывать вопросов. Теперь в разделе **private** объекта главной формы добавляем переменную *f* типа *LongInt*.

Подготовка закончено. Осталось только вызвать эти процедуры. Добавь на форму ещё одну кнопку. По нажатию первой, мы будем показывать окно:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  f:=ShowAbout(Handle);  
end;
```

Здесь я вызываю функцию показа окна и сохраняю результат в переменной. По нажатию второй кнопки вызываем процедуру освобождения памяти:

```
procedure TForm1.Button2Click(Sender: TObject);  
begin  
  FreeAbout(f);  
end;
```


Запусти пример и убедись, что всё работает корректно. Самое сложное здесь – определить, когда пользователь самостоятельно закрыл окно (например, кнопкой «Закрыть» в нашем окне «О программе»), чтобы мы вызвали процедуру *FreeAbout*. В качестве решения такой проблемы могу посоветовать следующее. При старте приложения присваивать переменной *f* значение 0. В этом случае, мы будем застрахованы от попадания в неё случайного числа. Перед созданием окна вызывать *FreeAbout*. В этом случае, сначала будет происходить проверка переменной *f* на ноль. Если переменная больше 0, то окно уже создавалось, но память не освободилась. Вот обновлённый код нажатия кнопки показа диалогового окна:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  if f>0 then  
    FreeAbout(f);  
  f:=ShowAbout(Handle);  
end;
```

Здесь идёт проверка, если *f* больше нуля, то надо освободить память от старого окна, а потом пытаться создавать новое.

По событию *OnClose* для главной формы тоже не помешает вызвать процедуру освобождения памяти. Если программа закрывается, то окно из библиотеки уж точно уже не понадобится, значит, переменную *f* можно проверять на 0 и если там большее значение, то освободить память.

На компакт диске находится пример, в котором уже реализовано всё сказанное и ты можешь увидеть этот код своими глазами и проверить его в действии.

 На компакт диске, в директории \Примеры\Глава 18\NoModal ты можешь увидеть пример этой программы.

18.6. Явная загрузка библиотек.

Предыдущие примеры хороши тем, что они просты, но у них есть недостаток – динамическая библиотека загружается автоматически при старте программы. В этом есть два недостатка. Во-первых, загрузка программы немного теряет в скорости (не сильно, но всё же), а функции из библиотеки могут вообще не понадобиться за всё время выполнения программы. Во-вторых, тебе может понадобиться поставлять программу в укороченном варианте без некоторых функций (без каких-либо dll файлов), но это не получится, потому что программа на этапе загрузке будет выдавать ошибку о том, что библиотека не найдена.

От всего этого можно избавиться, если использовать явную загрузку библиотеки в определённый момент. Для этого надо немного попотеть.

Давай в предыдущем нашем примере будем использовать явную загрузку библиотеки для вызова функции *ShowAbout*. В принципе, если уже делать явную загрузку, то для всех функций и процедур библиотеки, потому что если ты оставишь хотя бы одну неявной, то библиотека всё равно будет грузиться на этапе старта программы. Но для примера я взял только одну функцию, а процедуру попробуй перевести на явную загрузку сам. Тем более, что для этого не надо делать много изменений.

Итак, загружай приложение, написанное в прошлой части, которое использует динамическую библиотеку. В основном модуле убирай объявления функции *ShowAbout*, чтобы ничего не осталось. Теперь в разделе **type** пиши объявление нового типа:

ShowA=function (Handle: THandle):LongInt;stdcall;

Здесь я объявляю новый тип *ShowA*, который равен функции с параметрами функции *ShowAbout* из динамической библиотеки. Параметры должны быть точными, как при объявлении, иначе могут возникнуть проблемы.

Всё, этого достаточно. Теперь нужно переходить к обработчику события *OnClick* для первой кнопки, где мы показывали окно. Подкорректируй уже имеющийся код до следующего:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  dllHandle:THandle;
  sa:ShowA;
begin
  if f>0 then
    FreeAbout(f);

  dllHandle:=LoadLibrary('ProjectDLL.dll');

  if dllHandle=0 then
    exit;//Библиотека не загрузилась

  @sa:=GetProcAddress(dllHandle, 'ShowAbout');

  if @sa=nil then
    exit;//Функция не найдена

  f:=sa(Handle);
  FreeLibrary(dllHandle);
end;
```

Здесь у меня объявлено две локальные переменные:
dllHandle – здесь будет храниться указатель на загруженную библиотеку.
sa – имеет тип *ShowA*, т.е. тип функции из библиотеки.

В начале кода я выполняю уже знакомую проверку переменной *f*. Если она больше нуля, то окно уже показывалось и нужно освободить память старого окна, прежде чем создавать новое.

Дальше, я вызываю функцию *LoadLibrary*. Эта функция загружает указанную в качестве параметра динамическую библиотеку в память. Результатом выполнения функции является указатель на загруженную библиотеку. Этот указатель я сохраняю в переменной *dllHandle*. После этого нужно проверить, если указатель *dllHandle* равен нулю, то библиотека не загрузилась.


Теперь нам надо получить адрес функции *ShowAbout* в загруженной памяти, чтобы мы могли выполнить процедуру. Для этого я вызываю функцию *GetProcAddress*. Процедуре нужно передать два параметра:

1. Указатель на загруженную библиотеку.
2. Имя искомой процедуры.

Результатом будет адрес искомой функции, и я его сохраняю по адресу переменной *@sa*. Теперь *sa* указывает на адрес, по которому загружена библиотека *ShowAbout*. Единственное, что надо проверить – корректность адреса. Если он равен **nil**, то процедура не найдена (возможно, это старая версия библиотеки или неправильно указано имя).

Если всё нормально, то я вызываю функцию через переменную *f:=sa(Handle)*, почти так же, как это делалось раньше. Результат выполнения функции сохраняется в переменной *f*.

Последняя строка кода выгружает динамическую библиотеку из памяти - *FreeLibrary*. Точнее сказать, на этом этапе реальной выгрузки не происходит. Функция только сообщает системе о том, что больше библиотека программе не нужна. Если эту библиотеку использует другая программа (я же говорил, что одну библиотеку может использовать одновременно несколько программ), то она останется в памяти, пока та не сообщит о ненужности в загруженном DLL файле.

 На компакт диске, в директории \Примеры\Глава 18\CallFunc ты можешь увидеть пример этой программы.

18.7. Точка входа.

Ты наверно заметил, что в исходнике библиотеки есть *begin* и *end* не относящиеся к ни одной из процедур или функций. Код, описанный здесь, выполняется самым первым при загрузке библиотеки в память. Но зачем это нужно? Здесь можно было бы инициализировать какие-то переменные, но библиотека не может хранить их.

В библиотеках есть одна глобальная переменная, которая существует всегда и её имя *DLLProc*. Это не просто переменная, а указатель на процедуру. По умолчанию он равен **nil**, но если сюда записать адрес реальной процедуры, то эта процедура может вызываться на определённые события, происходящие в библиотеке. В процедуре можно вылавливать следующие события:

- DLL_PROCESS_ATTACH* – это событие генерируется при загрузке библиотеки.
- DLL_PROCESS_DETACH* – это событие генерируется при выгрузке библиотеки.
- DLL_THREAD_ATTACH* – при создании нового потока.

DLL_THREAD_DETACH – при отключении нового потока.

Честно скажу, что всё это тебе может и не пригодиться, но я всё же дам маленький пример, чтобы ты увидел, как это работает.

Открываем нашу библиотеку, написанную в прошлой части. Теперь добавляем в неё следующий код:

```
library ProjectDLL;

uses
  SysUtils,
  Classes,
  Windows,
  dialogs,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

exports ShowAbout index 10;
exports FreeAbout index 11;

procedure DLLEntryPoint(dwReason:DWord);
begin
  case dwReason of
    DLL_PROCESS_ATTACH:ShowMessage('Attach to process');
    DLL_PROCESS_DETACH:ShowMessage('Detach to process');
    DLL_THREAD_ATTACH:ShowMessage('Thread attach to process');
    DLL_THREAD_DETACH:ShowMessage('Thread detach to process');
  end;
end;

begin
  DLLProc:=@DLLEntryPoint;
  DLLEntryPoint(DLL_PROCESS_ATTACH);
end.
```

В разделе **uses** появилось объявление двух новых модулей *windows* и *dialogs*, без них наш код не скомпилируется. Чуть дальше появилась процедура *DLLEntryPoint* с одним параметром, в котором будет передаваться событие, которое произошло. Внутри процедуры я проверяю оператором **case** тип пришедшего сообщения и в зависимости от этого вывожу сообщение.

Между **begin** и **end** библиотеки я назначаю переменной *DLLProc* нашу процедуру. После этого я вызываю её и в качестве параметра указываю событие *DLL_PROCESS_ATTACH*.

 На компакт диске, в директории \Примеры\Глава 18\Entry ты можешь увидеть пример этой программы.

18.8. Вызов из библиотек процедур основной программы.

Теперь я хочу тебе показать ещё один трюк с использованием DLL библиотек. Мы уже познали многое и пора увидеть, как можно из библиотеки DLL вызывать процедуры, описанные в основной программе. Это очень сильный

способ сообщать основной программе, о каких либо событиях происходящих в библиотеке.

Создай новый проект динамической библиотеки. В основном модуле напишем следующий код:

```
library FuncProject;

uses
  SysUtils,
  Classes;

type
  TCompProc= procedure(Str:PChar);StdCall;

procedure CompS(Str:PChar; Proc:TCompProc);StdCall;
begin
  if @Proc<>nil then
    TCompProc(Proc)(Str);
end;

exports CompS index 10;

{$R *.res}

begin
end.
```

Первое, что здесь бросается в глаза – объявление в разделе **type** нового типа – *TCompProc*. Новый тип объявлен как процедура с одним параметром в виде переменной типа *PChar* и имеющей стандартный вызов. Объявление этого типа необходимо, чтобы объяснить динамической библиотеке, какого вида будет процедура в основной программе, которую надо будет вызывать.

Напоминаю, что тип *PChar* – это указатель на строку оканчивающуюся нулём (шестнадцатеричный #0). Сама переменная типа *PChar*, это только указатель на начало строки, а конец строки определяется по наличию значения #0. Но о конце строки нам никогда не придётся заботиться, нас больше будет волновать выделенная память, потому что под строку типа *PChar* нужно резервировать память.

Но всё это небольшое отступление и напоминание уже пройденного материала, так что продолжим рассматривать наш модуль. После объявления нового типа процедуры идёт процедура *CompS*, которая будет экспортироваться из модуля. Эту процедуру мы будем вызывать из основной программы, а из неё уже будем обращаться к процедуре основной программы.

В первой строке процедуры я проверяю значение переданного параметра *Proc*. В этом параметре мы должны получать адрес процедуры, которую надо вызвать. Если параметр не равен нулю, то можно вызывать процедуру. Для вызова я пишу следующий код: *TCompProc(Proc)(Str)*. Этим кодом я вызываю процедуру и передаю ей в качестве параметра переменную *Str*, которую мы сами же получили в качестве входного параметра.

В принципе, с процедурой всё. Остальное тебе уже должно быть знакомо. Переходим к написанию основного модуля.

Создай новое приложение. В разделе **type** сразу же объяви следующее:

type

```
TCompProc= procedure(Str:PChar);StdCall;  
procedure CompS(Str:PChar; Proc:TCompProc);export;StdCall;
```

В первой строке я объявляю тот же процедурный тип, что и в динамической библиотеке. Во второй строке объявляется процедура, которую мы экспортируем из библиотеки. Объявление должно быть именно в таком порядке. Если ты попытаешься объявить сначала процедуру из библиотеки, то при компиляции Delphi выдаст ошибку, потому что в качестве второго параметра в процедуре стоит тип *TCompProc* и сначала его нужно описать, а потом использовать.

Теперь напишем процедуру *CallFromDLL*. Эта процедура будет вызываться из динамической библиотеки. Она будет выглядеть так:

```
procedure CallFromDLL(Str:PChar);StdCall;  
begin  
  ShowMessage('DLL вызвала эту процедуру. Параметр равен: '+Str);  
end;
```

Наша процедура должна соответствовать объявленному типу *TCompProc*, а именно, в типе описано, что это процедура, что она имеет один параметр типа *PChar* и вызывается стандартно. Процедура должна соответствовать всему этому описанию, иначе произойдёт ошибка.

Внутри процедуры я вызываю только одну функцию *ShowMessage*, которая показывает на экран окно сообщения. В качестве единственного параметра нужно указать текст сообщения.

Теперь пометим на форму кнопку и по её нажатию напишем следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  CompS('Привет', @CallFromDLL);  
end;
```

Здесь я просто вызываю процедуру *CompS*, хранящуюся в динамической библиотеке. Попробуй запустить приложение и проверить результат работы программы.

 На компакт диске, в директории \Примеры\Глава 18\Call ты можешь увидеть пример этой программы.