Динамически подключаемые библиотеки

Системное программирование

Концепция механизма отображения файлов в память

- Прежде чем перейти к описанию динамически подключаемых библиотек, рассмотрим механизм, который позволяет динамически выполнять это подключение. Этот механизм называется **отображением содержимого файла (file mapping) в виртуальную память** процесса.

- Операционная система создает файлы подкачки с виртуальными страницами, которые система отображает в адресные пространства процессов.

- В операционных системах Windows реализован механизм, который позволяет отображать в адресное пространство процесса не только содержимое файлов подкачки, но и содержимое обычных файлов. То есть в этом случае файл или его часть рассматривается как набор виртуальных страниц процесса, которые имеют последовательные логические адреса.

- Файл, отображенный в адресное пространство процесса, называется **представлением** или **видом файла (file view).**

- После отображения файла в адресное пространство процесса доступ к виду может осуществляться с помощью указателя, как к обычным данным в адресном пространстве процесса.

Концепция механизма отображения файлов в память

- Несколько процессов могут одновременно отображать один и тот же файл в свое адресное пространство. В этом случае операционная система обеспечивает согласованность содержимого файла для всех процессов, если доступ к этим данным осуществляется как к области виртуальной памяти процесса.
- Для доступа к файлу, который отображен в память, не используется функция writeFile.
- Согласованность данных, хранящихся в файле, отображенном в память несколькими процессами, называется **когерентностью данных**.
- Однако следует отметить, что когерентность данных для файла, отображенного в память, не поддерживается в том случае, если этот файл отображается в адресное пространство процессов, которые выполняются на других компьютерах в локальной сети.
- Основным назначением механизма отображения файлов в память является загрузка программы (файла с расширением exe) на выполнение, в адресном пространстве процесса, и динамическое подключение библиотек функций во время выполнения этой программы.
- Механизм отображения файлов в память позволяет осуществлять обмен данными между процессами, принимая во внимание то, что система обеспечивает когерентность данных в файле, отображаемом в память.

Концепция механизма отображения файлов в память

- Последовательность действий, которые необходимо выполнить для работы с отображаемым в память файлом:
  - открыть файл, который будет отображаться в память;
  - создать объект ядра, который выполняет отображение файла;
  - отобразить файл или его часть в адресное пространство процесса;
  - выполнить необходимую работу с видом файла;
  - отменить отображение файла;
  - закрыть объект ядра для отображения файла;
  - закрыть файл, который отображался в память.

Создание и открытие объекта, отображающего файл

- Если в память отображается существующий файл, то первым делом этот файл должен быть открыт для доступа, используя функцию **CreateFile**.

- Это делается для того, чтобы получить дескриптор файла, т. к. в дальнейшем этот дескриптор используется при создании объекта, отображающего файл в память процесса.

- Если отображаемый файл используется просто для обмена данными между процессами, то создавать для этого специальный файл на диске не обязательно. Для этого можно использовать файлы подкачки операционной системы.

Создание и открытие объекта, отображающего файл

- После того как файл открыт, создается объект, отображающий этот файл в память. Под объектом, отображающим файл в память, можно понимать объект ядра операционной системы, который выполняет отображение файла в адресное пространство процесса. Можно также представить, что этот объект позволяет рассматривать файл, отображаемый в память, как файл подкачки.

- Для создания этого объекта используется функция **CreateFileMapping**:

  Function CreateFileMapping(hFile: THandle; lpFileMappingAttributes: PSecurityAttributes; flProtect, dwMaximumSizeHigh, dwMaximumSizeLow: DWORD; lpName: PChar): THandle;

- В случае успешного завершения эта функция возвращает дескриптор объекта, отображающего файл в память, а в случае неудачи — 0.

Создание и открытие объекта, отображающего файл

- Параметр **hFile** должен содержать дескриптор открытого файла, для которого будет создаваться объект, отображающий этот файл в память процесса, в результате присвоения этому аргументу значения $**FFFFFFFF** или определенной в **Windows.pas** константы **MAXDWORD** можно связать создаваемый объект файлового отображения со страничным свап-файлом.

- Параметр **lpFileMappingAttributes** указывает на атрибуты защиты для объекта, отображающего файл в память, это указатель на запись типа **TSecurityAttributes**. Значение nil устанавливает атрибуты защиты по умолчанию. То есть в этом случае объект отображения не является наследуемым и принадлежит создавшему его пользователю.

Создание и открытие объекта, отображающего файл

- Параметр **f lProtect** - содержит флаги, которые задают режимы доступа к виду файла в памяти процесса. Этот параметр может принимать одно из следующих значений:
  - **page_readonly** — из вида файла можно только читать данные;
  - **page_readwrite** — разрешает чтение и запись данных в вид файла;
  - **page_writecopy** — разрешает чтение и запись данных в вид файла, но при записи создается новая копия вида файла.
- Значения этого параметра совпадают со значениями соответствующего параметра функции **virtualAlloc**, которая распределяет виртуальную память процессу. Режимы доступа к объекту, отображающему файл в память, должны соответствовать режимам доступа к файлу, для которого создается этот объект отображения.

- В параметре **flProtect** может быть установлена любая комбинация флагов, которые определяют атрибуты секций исполняемых файлов, заданных в переносимом формате (**portable executable files**).

Создание и открытие объекта, отображающего файл

- Параметры **dwMaximumSizeHigh** и **dwMaximumSizeLow** определяют соответственно значение старшей и младшей частей, которые в совокупности задают размер объекта, отображающего файл в память. Если эти значения установлены в 0, то объект, отображающий файл в память, имеет размер, равный размеру файла. Если размер этого объекта будет меньше размера файла, то система не сможет отобразить весь файл в память. Если же размер объекта, отображающего файл, больше чем размер файла, то размер файла увеличивается до размера объекта.

- Последний параметр **lpName** используется для задания имени объекта, отображающего файл в память. Это имя используется для доступа к одному и тому же объекту в разных процессах. Если процесс пытается получить доступ к уже созданному объекту, отображающему файл, то флаги доступа, установленные в параметре **flProtect**, должны соответствовать флагам доступа, уже установленным в существующем объекте, отображающем файл.

- После завершения работы с объектом, отображающим файл в память, его дескриптор нужно закрыть, используя функцию **CloseHandle**.

Отображение файла в память

- После того как создан объект, отображающий файл в память, файл или его часть должна быть отображена в память процесса. Другими словами, должен быть создан вид файла или его части в адресном пространстве процесса.

- Для отображения файла или его части в адресное пространство процесса используется функция **MapviewOfFile**:

```
Function MapViewOfFile(hFileMappingObject: THandle; dwDesiredAccess:
    DWORD; dwFileOffsetHigh, dwFileOffsetLow, dwNumberOfBytesToMap:
    DWORD): Pointer;
```

- В случае успешного завершения функция возвращает указатель на вид файла в адресном пространстве процесса, а случае неудачи — **nil**.

Отображение файла в память

- Параметр **hFileMappingObject** должен содержать дескриптор объекта, отображающего файл в память, который был предварительно создан функцией **CreateFileMapping**.

- Параметр dwDesiredAccess задает режим доступа к виду файла и может принимать одно из следующих значений:
  - **file_map_write** — чтение и запись в вид файла;
  - **file_map_read** — только чтение из вида файла;
  - **file_map_all_access** — чтение и запись в вид файла;
  - **file_map_copy** — при записи в вид файла создается его копия, а исходный файл не изменяется.

- Установленное в этом параметре значение должно соответствовать режиму доступа, который установлен для объекта, отображающего файл в память.

Отображение файла в память

- Параметры ***dwFileOffsetHigh*** и ***dwFileOffsetLow*** задают смещение от начала файла, т.е. первый байт файла, начиная с которого файл отображается в память. Это смещение задается в байтах и должно быть кратно гранулярности распределения виртуальной памяти в системе (**allocation granularity**), которая может быть определена при помощи вызова функции **Getsysteminfo**. Единственным параметром этой функции является указатель на структуру **SYSTEM_INFO**, в поле **wAllocationGranularity** которой функция **Getsysteminfo** помещает гранулярность (в байтах). Это значение зависит от архитектуры компьютера и в большинстве случаев равно 64 Кбайт.

- Параметр **dwNumberofBytesToMap** задает количество байт, которые будут отображаться в память из файла. Если значение этого параметра равно нулю, то в память будет отображен весь файл.

Отображение файла в память

- Если необходимо отобразить файл в адресное пространство процесса, начиная с некоторого заданного виртуального адреса, то для этой цели нужно использовать функцию **MapViewofFileEx**:

```
function MapViewOfFileEx (hFileMappingObject: THandle; dwDesiredAccess: DWORD;
    dwFileOffsetHigh: DWORD; dwFileOffsetLow: DWORD; dwNumberOfBytesToMap: DWORD;
    lpBaseAddress: pointer): pointer;
```

- hFileMappingObject - дескриптор объекта, отображающего файл
- dwDesiredAccess - режим доступа
- dwFileOffsetHigh - старшее двойное слово смещения
- dwFileOffsetLow - младшее двойное слово смещения
- SIZE_T dwNumberOfBytesToMap - количество отображаемых байтов
- lpBaseAddress - начальный адрес отображения файла

- Последний параметр этой функции указывает на начальный виртуальный адрес вида отображаемого файла. Этот параметр может быть установлен в nil. В этом случае система сама выберет начальный адрес загрузки.

Отображение файла в память

- После окончания работы с видом файла в памяти нужно отменить отображение файла в адресное пространство процесса. Отмена отображения файла освобождает виртуальные адреса процесса.

- Если отображение файла в адресное пространство процесса не отменено, то система продолжает держать отображаемый файл открытым до тех пор, пока существует его вид, независимо от того, закрыт этот файл функцией **closeHandle** или нет.

- Для отмены отображения файла в память используется функция **UnMapViewOfFile**:

```
Function UnMapViewOfFile(lpBaseAddress: Pointer): Boolean;
```

- *Результат функции* - в случае успешного выполнения функции получим **TRUE**, иначе **FALSE**.

- Единственным параметром этой функции является начальный адрес вида файла - l***pBaseAddress*** - в этот аргумент должно быть помещено значение, возвращаемое функцией **MapViewOfFile** или **MapViewOfFileEx**.

Закрытие объекта файлового отображения

- Для уничтожения объекта файлового отображения и освобождения памяти используется функция **CloseHandle**.

Function CloseHandle(hFileMapObj:THandle):Boolean;

- ***hFileMapObj*** – дескриптор объекта файлового отображения, полученный в результате выполнения функции **CreateFileMapping**.
- *Результат функции* - в случае успешного выполнения функции получим **TRUE**, иначе **FALSE**. Для правильного завершения работы с объектом файлового отображения вначале следует применить функцию **UnMapViewOfFile**, а затем **CloseHandle**.

- Прототипы функций (кроме CloseHandle) и необходимые константы определены в файле **Windows.pas**. Этот файл находится в созданной при установке **Delphi** директории **BorlandDelphi_XSourceRtlWin**

Пример

- Поместите на форму TEdit, 2 экземпляра TBitBtn, TTimer и 3 экземпляра TLabel.
- В строку ввода будем вводить свои данные в виде текста. Нажав на кнопку OK, передадим данные в объект файлового отображения. Считывать данные из этого объекта будем на событие OnTimer.

- Описатель объекта файлового отображения поместим в глобальную переменную hFileMapObj типа THandle, указатель на начальный адрес данных объекта поместим в глобальную переменную lpBaseAddress типа PChar *(хотя в результате выполнения функции MapViewOfFile возвращается переменная типа Pointer)*.

Пример

- **unit** Main;

  interface

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

  type
  TForm1 = class(TForm)
  edVariable: TEdit;
  lbEnterValues: TLabel;
  bbOK: TBitBtn;
  bbExit: TBitBtn;
  lbShowValue: TLabel;
  lbVariable: TLabel;
  Timer1: TTimer;

- **procedure** FormCreate(Sender: TObject);
  procedure bbExitClick(Sender: TObject);
  procedure FormClose(Sender: TObject; var Action: TCloseAction);
  procedure bbOKClick(Sender: TObject);
  procedure Timer1Timer(Sender: TObject);
  private
  { Private declarations }
  public
  { Public declarations }
  end;

  var
  Form1: TForm1;
  //глобальные переменные
  hFileMapObj:THandle;*//дескриптор FaleMapping*
  lpBaseAddress:PChar;*//"указатель" на начальный адрес данных*

Пример

```
implementation

    {$R *.DFM}

    procedure TForm1.FormCreate(Sender: TObject);
    begin
    //создадим FileMapping с именем MySharedValue
    //и передадим его дескриптор в глобальную переменную hFileMapObj
    hFileMapObj:=CreateFileMapping(MAXDWORD,Nil,PAGE_READWRITE,0,4,'MySharedValue');
    If (hFileMapObj=0) Then
    //ошибка
ShowMessage('Не могу создать FileMapping!')
    Else
    //подключим FileMapping к адресному пространству
    //и получим начальный адрес данных
    lpBaseAddress:=MapViewOfFile(hFileMapObj,FILE_MAP_WRITE,0,0,0);
    If lpBaseAddress=Nil Then
    //ошибка
    ShowMessage('Не могу подключить FileMapping!');
    end;
```

Пример

- **procedure** TForm1.bbExitClick(Sender: TObject);
  begin
  //закроем форму
  Close;
  end;

  procedure TForm1.FormClose(Sender: TObject; **var** Action: TCloseAction);
  begin
  //кое-что надо сделать при закрытии формы:
  //отключим FileMapping от адресного пространства
  UnMapViewOfFile(lpBaseAddress);
  //освободим объект FileMapping
  CloseHandle(hFileMapObj);
  //теперь форму можно закрыть
  Action:=caFree;
  end;

Пример

- **procedure** TForm1.bbOKClick(Sender: TObject);
  begin
  //поместим в адресное пространство свои данные
  //переменная типа PChar имеет в конце завершающий #0, значит при считывании данных
  //система сама сможет определить, где находится конец нужных данных
  StrPCopy(lpBaseAddress,edVariable.Text);
  end;

  procedure TForm1.Timer1Timer(Sender: TObject);
  begin
  //считаем нужные данные, обратившись по начальному адресу
  //данных адресного пространства FileMapping
  lbVariable.Caption:=PChar(lpBaseAddress);
  end;

  end.

Обмен данными между процессами через отображаемый в память файл

- Так как один и тот же файл может быть отображен в память несколькими процессами, и система поддерживает когерентность таких отображений, то механизм отображения файлов в память может использоваться для обмена данными между процессами.

- В операционных системах Windows все остальные механизмы обмена данными между процессами базируются на отображении файлов в память. Поэтому можно сказать, что отображение файлов в память обеспечивает наилучшую производительность по сравнению со всеми остальными способами обмена данными между процессами.

- Так как безразлично, какой файл используется для обмена данными между процессами, то в этом случае лучше использовать файл подкачки страниц.

Обмен данными между процессами через отображаемый в память файл

- В программах из листингов 1 и 2 показывается, как передать данные другому процессу через файл подкачки страниц.
- Первая программа создает объект, отображающий файл подкачки, а затем записывает в вид файла подкачки массив целых чисел.
- После этого она запускает вторую программу, которая будет читать созданный массив, через отображаемый в память файл, используя тот же объект, отображающий файл в память.
- Отметим, что для обращения к одному и тому же объекту, отображающему файл, используется имя этого объекта.

Программа 1

```
program Project2;

{$APPTYPE CONSOLE}

uses
  SysUtils,windows;
const MappingName='MappingName';
  n = 10; // размерность массива
type a=array[0..n-1]of integer;
var
  hMapping:THandle; // дескриптор объекта, отображающего файл
  ptr:^a; // для указателя на массив
  si:^STARTUPINFO;
  piApp:^PROCESS_INFORMATION;
  i:integer;
  lpszAppName:PChar;
```

Программа 1

- begin
- { TODO -oUser -cConsole Main : Insert code here }
- writeln('This is a parent process.');
- // открываем объект отображения файла в память
- hMapping:=CreateFileMapping(
- INVALID_HANDLE_VALUE, // файл подкачки страниц
- nil, // атрибуты защиты по умолчанию
- PAGE_READWRITE, // режим доступа: чтение и запись
- 0, // старшее слово = О
- n*sizeof(integer), // младшее слово = длине массива
- MappingName); // имя объекта отображения

Программа 1

```
if hMapping=0
 then
  writeln('Create file mapping failed. ', GetLastError())
 Else
  begin
  // создаем вид файла
  ptr:=MapViewOfFile(
  hMapping, // дескриптор объекта отображения
  FILE_MAP_WRITE, // режим доступа к виду
  0,0, // отображаем файл с начала
  0); // отображаем весь файл
  // инициализируем массив и выводим его на консоль
  writeln('Array: ');
  for i:=0 to n-1 do
   begin
    ptr^[i]:=i;
    write(ptr^[i],' ');
   end;
  writeln;
```

Программа 1

```
//создаем процесс, который будет читать данные из отображаемого
  //в память файла
  lpszAppName:='C:\ConsoleProcess.exe';
  new(si);
  ZeroMemory(si,sizeof(STARTUPINFO));
  si^.cb:=sizeof(STARTUPINFO);
// создаем новый консольный процесс
  if CreateProcess(lpszAppName,nil,nil,nil,FALSE,CREATE_NEW_CONSOLE,nil,nil,si^,piApp^)=false
   then writeln('Create process failed.',GetLastError())
   else
    begin
    // ждем завершения созданного прцесса
    WaitForSingleObject(piApp.hProcess, INFINITE);
    // закрываем дескрипторы этого процесса в текущем процессе
    CloseHandle(piApp.hThread);
    CloseHandle(piApp.hProcess);
```

Программа 1

```
// отменяем отображение файла в память
      if UnmapViewOfFile(ptr)=false
       then
        writeln('Unmap view of file failed.',GetLastError())
       else
        begin
        // закрываем объект отображения файла в память
        if CloseHandle(hMapping)=false
         then writeln('Close file failed.',GetLastError())
         else
          // ждем команду на завершение процесса
          begin
           writeln('Input any char to exit: ');
           readln;
          end;
        end;
       end;
      end;
end.
```

Программа 2

```
program Project3;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  Windows;

const n=10; // размерность массива
type a=array[0..n-1] of integer;
var  MappingName:PChar;
  hMapping:THandle;// дескриптор объекта, отображающего файл
  ptr:^a; // для указателя на массив
  i:integer;
```

Программа 2

- begin
- MappingName:='MappingName';
- writeln('This is a child process.');
- // открываем объект отображения файла в память
- hMapping:=CreateFileMapping(
- INVALID_HANDLE_VALUE, // файл подкачки страниц
- nil, // атрибуты защиты по умолчанию
- PAGE_READWRITE, // режим доступа: чтение и запись
- 0, // старшее слово = О
- n*sizeof(integer), // младшее слово = длине массива
- MappingName); // имя объекта отображения

Программа 2

```
if hMapping=0
 then
   writeln('Create file mapping failed. ',GetLastError())
 else
   begin
    // создаем вид файла
    ptr:=MapViewOfFile(
    hMapping, // дескриптор объекта отображения
    FILE_MAP_WRITE, // режим доступа к виду
    0,0, // отображаем файл с начала
    0); // отображаем весь файл
    // выводим массив из вида на консоль
    writeln('Array: ');
    for i:=0 to n-1 do writeln(ptr^[i],' ');
    // отменяем отображение файла в память
```

Программа 2

```
if UnmapViewOfFile(ptr)=false
      then writeln('Unmap view of file failed.',GetLastError())
      else
        begin
          // закрываем объект отображения файла в память
          if CloseHandle(hMapping)=false
            then writeln('Close file failed.',GetLastError())
            else
              begin
                // ждем команду на завершение процесса
                writeln('Input any char to exit: ');
                readln
              end
        end
    end
end.
```

Сброс вида в файл

- Для записи на диск содержимого вида файла используется функция **FlushviewOfFile**, которая имеет следующий прототип:

```
BOOL FlushviewOfFile(
LPCVOID lpBaseAddress, // базовый адрес вида
SIZE_T dwNumberOfButesToFlush // количество записываемых
    байтов
);
```

- В случае успешного завершения функция возвращает ненулевое значение, а в случае неудачи — false.

Сброс вида в файл

- Параметр lpBaseAddress этой функции должен содержать начальный адрес, который принадлежит диапазону адресов одного из видов и, начиная с которого, содержимое вида записывается на диск, а параметр dwNumberOfButesToFlush — количество записываемых байт.
- Если значение второго параметра равно нулю, то на диск записывается весь вид файла, начиная с базового адреса и до конца этого вида.
- Следует отметить, что функция FlushviewOfFile гарантирует, что данные были записаны на диск локального компьютера, но не гарантирует запись этих данных в вид на удаленном компьютере в локальной сети.

Сброс вида в файл

- В листингах 4 и 5, приведены две программы, которые иллюстрируют использование функции **FlushviewOfFile**.
- Первая программа создает файл, затем отображает его в память и изменяет его содержимое.
- После этого запускается вторая программа, которая читает содержимое отображенного в память файла.
- Чтобы вторая программа прочитала измененный файл, вид перед ее запуском сбрасывается на диск при помощи вызова функции FlushviewOfFile.

Сброс вида в файл

- Листинг 3. Сброс вида в файл на диске
- #include <windows.h>
- #include <fstream.h>
- int main ()
- int a[10] = {0, 1, 2, 3, 4, 5, б, 7, 8, 9};
- char f ile_name[] = "C:\\Demo.bin";
- char mapping_name [ ] = "MappingName";
- HANDLE hFile, hMapping; // дескрипторы файла и объекта отображения
- int *ptr; // для указателя на массив
- // открьшаем файл для вывода
- ofstream out(file_name, ios::out | ios::binary);
- if (lout)
- {
- cerr « "File constructor failed." « endl;
- return 0;
- }

Сброс вида в файл

- // выводим исходный массив в файл и на консоль
- cout « "Initial array: ";
- for (int i = 0; i < 10; ++i)
- {
- out.write((char*)&a[i], sizeof(int));
- cout « a[i] « ' ';
- }
- cout « endl;
- // закрываем выходной файл
- out.close();
- //
- // открываем файл для отображения в память
- hFile = CreateFile(file_name, GENERIC_READ | GENERIC_WRITE,
- FILE_SHARE_READ | FILE_SHARE_WRITE,
- NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

Сброс вида в файл

- if (hFile == INVALID_HANDLE_VALUE)
- {
- cerr « "Create file failed." « endl;
- return GetLastError();
- }
- // открываем объект, отображающий файл в память
- hMapping = CreateFileMapping(
- hFile, // дескриптор открытого файла
- NULL, // атрибуты защиты по умолчанию
- PAGE_READWRITE, // режим доступа
- 0, 0, // размер объекта отображения равен размеру файла
- mapping_name); // имя объекта отображения
- if (IhMapping)
- {
- cerr « "Create file mapping failed." « endl;
- return GetLastError();
- }

Сброс вида в файл

- // создаем вид файла
- ptr = (int*)MapViewOfFile(
- hMapping, // дескриптор объекта отображения
- FILE_MAP_WRITE, // режим доступа к виду
- 0,0, // отображаем файл с начала
- 0); // отображаем весь файл
- // изменяем значения элементов массива
- for (i = 0; i < 10; ++i)
- ptr[i] += 10;
- // сбрасываем весь вид на диск
- if (IFlushViewOfFile(ptr, 0))
- {
- cerr « "Flush view of file failed." « endl;
- return GetLastError();
- }

Сброс вида в файл

- // создаем процесс, который будет читать данные из отображаемого
- //в память файла
- char IpszAppName[] = "C:\\ConsoleProcess.exe";
- STARTUPINFO si;
- PROCESS_INFORMATION piApp;
- ZeroMemory(&si, sizeof(STARTUPINFO));
- si.cb = sizeof(STARTUPINFO);
- // создаем новый консольный процесс
- if (!CreateProcess(IpszAppName, NULL, NULL, NULL, FALSE,
- CREATE_NEW_CONSOLE, NULL, NULL, &si, &piApp))
- {
- cerr « "Create process failed." « endl;
- return GetLastError();
- // ждем завершения созданного прцесса
- WaitForSingleObj ect(piApp.hProcess, INFINITE);
- // закрываем дескрипторы этого процесса в текущем процессе
- CloseHandle(piApp.hThread);
- CloseHandle(piApp.hProcess);

Сброс вида в файл

- // отменяем отображение файла в память
- if (!UnmapViewOfFile(ptr))
- {cerr « "Unmap view of file failed." « endl;
- return GetLastError(); }
- // закрываем объкт отображения файла в память
- if (!CloseHandle(nMapping))
- {cerr « "Close file failed." « endl;
- return GetLastError(); }
- // закрываем файл
- if (!CloseHandle(hFile))
- {
- cerr « "Close file failed." « endl;
- return GetLastError();
- }
- // ждем команду на завершение процесса
- char c;
- cout « "Input any char to exit: ";
- cin » c;
- return 0;

Сброс вида в файл

- Листинг 5. Чтение данных из файла, который отображен а память другим процессом

- #include <windows.h>
- #include <fstream.h>
- int main()
- char f ile_name [ ] = " C: \ \Demo.bin" ;
- int a[10];
- ifstream in(f ile__name, ios::in | ios::binary); // открываем файл для ввода
- if (!in)
- {cerr « "File constructor failed." « endl;
- return 0; }

Сброс вида в файл

- cout « "Final array: "; // вводим финальный массив из файла и выводим на консоль
- for (int i = 0; i < 10; ++i)
- { in.read((char*)&a[i], sizeof(int));
- cout « a[i] « ' '; }
- cout « endl;
- in.close(); // закрываем входной файл
- char c; // ждем команду на завершение процесса
- cout « "Input any char to exit: ";
- cin » c;
- return 0;

Концепция динамически подключаемых библиотек

- **Динамически подключаемая библиотека (DLL, Dynamic Link Library)** представляет собой программный модуль, который может быть загружен в виртуальную память процесса как статически, во время создания исполняемого модуля процесса, так и динамически, во время исполнения процесса операционной системой.

- Программный модуль, оформленный в виде DLL, хранится на диске в виде файла, который имеет расширение dll, и может содержать как функции, так и данные.

- Для загрузки DLL в память используется механизм отображения файлов в память.

Концепция динамически подключаемых библиотек

- Динамически подключаемые библиотеки предназначены, главным образом, для разработки функционально-замкнутых библиотек функций, которые могут использоваться разными приложениями.

- Это позволяет снизить затраты на разработку программного обеспечения, т. к. один и тот же программный код может использоваться разными разработчиками.

- Динамически подключаемые библиотеки позволяют уменьшить объем используемой физической памяти при одновременной работе нескольких приложений, которые используют одну и ту же библиотеку. Это достигается благодаря механизму проецирования DLL в виртуальную память процессов, т. к. в этом случае все приложения разделяют один и тот же экземпляр исполняемого кода DLL, загруженный в физическую память.

Концепция динамически подключаемых библиотек

- Исполняемые файлы и файлы динамических библиотек, т. е. файлы с расширениями exe и dll разбиты на разделы, каждый из которых содержит данные только определенного типа.

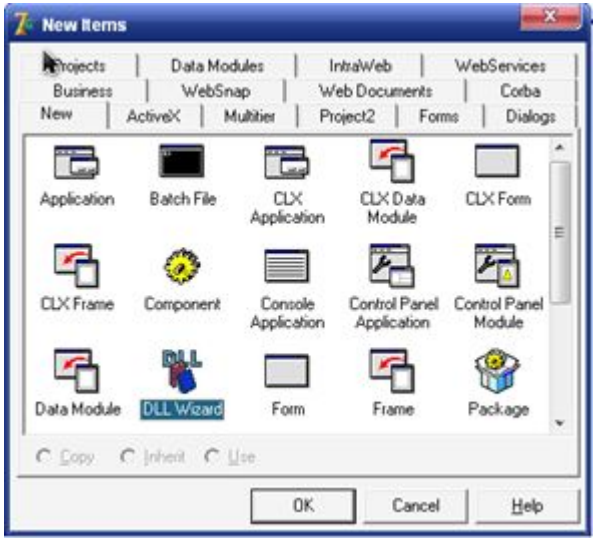
- Один из этих разделов содержит только исполняемый код приложения или динамически подключаемой библиотеки. Вот этот раздел и хранится в физической памяти в одном экземпляре и отображается в адресное пространство всех процессов.

- Те же разделы, которые содержат данные, хранятся для каждого процесса в отдельном экземпляре.

- Поэтому получается, что процессы совместно разделяют исполняемый код из динамически подключаемой библиотеки, но каждый процесс имеет свой набор переменных из этой библиотеки.

Создание Dll

- В меню **File** выберите пункт **New -> Other**.
- В диалоговом окне на вкладке **New** выберите **DLL Wizard**.
- Автоматически будет создан модуль – пустой шаблон будущей **DLL**.



Шаблон DLL

- library MyDll;
  uses
  <используемые модули>
  <объявления и описания функций>
  exports
  <экспортируемые функции>
  begin
  <инициализационная часть>
  end.

- Заготовка отличается от заготовки для создания кода *.exe-файла тем, что используется служебное слово Library вместо Program. Кроме того, отсутствуют обращение к методам объекта TApplication (хотя экземпляр этого объекта в действительности создается в DLL!), а также модуль реализации главной формы.

Синтаксис DLL

```
library Example_dll;
uses SysUtils, Classes;
var { Объявляем переменные } k1: integer; k2: integer;
{ Объявляем функцию }
function SummaTotal(factor: integer): integer; register;
begin
factor:= factor * k1 + factor;
factor:= factor * k2 + factor;
result:= factor; end;
{ Экспортируем функцию для дальнейшего использования программой }
exports SummaTotal;
{ Инициализация переменных }
begin k1:= 2; k2:= 5;
end.
```

- Для того, чтобы построить **DLL**, выберите **Project -> Build *Имя_проекта***.

Видимость функций и процедур

- Функции и процедуры могут быть локальными и экспортируемыми из **DLL**.

- **Локальные** функции и процедуры могут быть использованы внутри **DLL**. Они видны только внутри библиотеки и ни одна программа не может их вызывать извне.

- **Экспортируемые** функции и процедуры могут быть использованы за пределами **DLL**. Другие программы могут вызывать такие функции и процедуры.

- Исходный код ранее использует экспортируемую функцию. Имя функции следует за зарезервированным словом **Exports**.

Загрузка DLL

- В Delphi есть два вида загрузки **DLL**:
  - Статическая загрузка
  - Динамическая загрузка

- **Статическая загрузка.** При запуске приложения загружается автоматически. Она остается в памяти на протяжении выполнения программы. Очень просто использовать. Просто добавьте слово **external** после объявления функции или процедуры.

```
function SummaTotal(factor: integer): integer; register;
external ‘Example.dll’;
```

- Если **DLL** не будет найден, программа будет продолжать работать.

Загрузка DLL

- В Delphi есть два вида загрузки **DLL**:
  - Статическая загрузка
  - Динамическая загрузка

- **Статическая загрузка.** При запуске приложения загружается автоматически. Она остается в памяти на протяжении выполнения программы. Очень просто использовать. Просто добавьте слово **external** после объявления функции или процедуры.
- function SummaTotal(factor: integer): integer; register; external 'Example.dll'; Если **DLL** не будет найден, программа будет продолжать работать.
- **Динамическая загрузка. DLL** загружается в память по мере необходимости. Ее реализация более сложная, потому что Вы сами должны загружать и выгружать ее из памяти. Память используется более экономно, поэтому приложение работает быстрее. Программист сам должен следить, чтобы все работало правильно. Для этого нужно:
- Объявить тип описываемой функции или процедуры.
- Загрузить библиотеку в память.
- Получить адрес функции или процедуры в памяти.
- Вызвать функцию или процедуру.
- Выгрузить библиотеку из памяти.

Загрузка DLL

- **Динамическая загрузка. DLL** загружается в память по мере необходимости. Ее реализация более сложная, потому что Вы сами должны загружать и выгружать ее из памяти.

- Память используется более экономно, поэтому приложение работает быстрее. Программист сам должен следить, чтобы все работало правильно.

- Для этого нужно:
  - Объявить тип описываемой функции или процедуры.
  - Загрузить библиотеку в память.
  - Получить адрес функции или процедуры в памяти.
  - Вызвать функцию или процедуру.
  - Выгрузить библиотеку из памяти.

Загрузка DLL

- **Объявление типа, описывающего функцию**
- type TSumaTotal = function(factor: integer): integer;

- **Загрузка библиотеки**
- dll_instance:= LoadLibrary('Example_dll.dll');

- **Получаем указатель на функцию**
- @SummaTotal:= GetProcAddress(dll_instance, 'SummaTotal');

- **Вызов функции**
- Label1.Caption:= IntToStr(SummaTotal(5));

- **Выгрузка библиотеки из памяти**
- FreeLibrary(dll_instance); Динамический вызов **DLL** требует больше работы, но легче управлять ресурсами в памяти.

Загрузка DLL

- Если Вы должны использовать **DLL** в пределах программы, тогда предпочтительнее статическая загрузка.

- Не забывайте использовать блок **try…except** и **try…finally**, чтобы избежать ошибок во время выполнения программы.

Экспорт строк

- Созданная **DLL** с использованием Delphi, может быть использована и в программах, написанных на других языках программирования.
- Типы, которые существуют в Delphi могут отсутствовать в других языках. Желательно использовать собственных типы данных из Linux или Windows.
- Можно использовать *строки* и *динамические массивы* в **DLL**, написанной в Delphi, но для этого нужно подключить модуль **ShareMem** в раздел **uses** в **DLL** и программе, которая будет ее использовать. Кроме того, это объявление должно быть первым в разделе **uses** каждого файла проекта.
- Типов **string** C, C++ и других языках не существует, поэтому желательно использовать вместо них **Pchar**.

Пример DLL

```
library Example_dll;
uses SysUtils, Classes;
Var
 { Объявляем переменные } k1: integer; k2: integer;
{ Объявляем функцию }
function SummaTotal(factor: integer): integer; register;
begin
factor:= factor * k1 + factor;
factor:= factor * k2 + factor;
 result:= factor;
end;
{ Экспортируем функцию для дальнейшего использования программой }
exports SummaTotal;
 { Инициализация переменных }
 begin
 k1:= 2; k2:= 5;
end.
```

Пример вызова функции из DLL

- unit Unit1;
- interface
- uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;
- type TForm1 = class(TForm)
- Button1: TButton;
- procedure Button1Click(Sender: TObject);
- private
- { Private declarations }
- public
- { Public declarations }
- end;
- type TSummaTotal = function(factor: integer): integer;
- var Form1: TForm1;
- implementation
- {$R *.dfm}
- procedure TForm1.Button1Click(Sender: TObject);
- var dll_instance: Thandle;
- SummaTotal: TSummaTotal;
- begin
- dll_instance:= LoadLibrary('Example_dll.dll');
- @SummaTotal:= GetProcAddress(dll_instance, 'SummaTotal');
- Label1.Caption:= IntToStr(SummaTotal(5));
- FreeLibrary(dll_instance);
- end;
- end.

Управление загрузкой DLL

- Если Вы хотите подготовить **DLL** при инициализации, у Вас может быть функция с именем ***DllMain***, которая будет делать инициализацию. Windows вызывает функцию ***DllMain*** из **DLL** в четырех случаях:
  - Когда процесс присоединяет DLL
  - Когда поток присоединяет DLL
  - Когда процесс отсоединяет DLL
  - Когда нить отсоединяет DLL

- Каждая DLL имеет точку входа. Эта точка входа реализована как функция обратного вызова. Она вызывается системой, когда событие происходит. Функция имеет имя по умолчанию ***DllMain***. DLL, которая не должна быть загружена по определению, может просто возвращать **False** в функции ***DllMain()*** как только она обнаруживает, кто вызывает ее.

Управление загрузкой DLL

- В Delphi, ***DLLProc*** используется для определения процедуры, которая вызывается каждый раз, когда вызывается точка входа библиотеки DLL. Процедура назначенная на *DLLProc* получает один параметр целое число (Reason).

- Функция API ***GetModuleFileName()*** возвращает название вызывающего модуля, если Вы передаете 0 как его первый аргумент. Этот параметр - дескриптор модуля, имя которого Вы хотите знать.

- Когда параметр *Reason* - **DLL_PROCESS_ATTACH**, устанавливая ***ExitCode*** в ненулевое значение, заставляет точку входа возвращать **False**.

Управление загрузкой DLL

- Код, который позволяет только MyCallingApp.exe загружать библиотеку (Примечание: у DLL в этом примере нет НИКАКОГО кода кроме процедуры DLLMain):

- library OnlyMyDLL;
- uses SysUtils, Windows; procedure DllMain(reason: integer) ;
- var buf : array[0..MAX_PATH] of char; loader : string;
- begin
- case reason of
- DLL_PROCESS_ATTACH:
- begin GetModuleFileName(0, buf, SizeOf(buf)) ;
- loader := buf;

Управление загрузкой DLL

- if Pos('MyCallingApp.exe', loader) > 0 then ExitCode := -1
- end;
- DLL_PROCESS_DETACH begin // DLL выгружается
- ... end;
- end;
- end;
- (*DllMain*)
- begin DllProc := @DllMain;
- DllProc(DLL_PROCESS_ATTACH) ;
- // другой код для этой DLL
- ... end.

Локализация приложений Delphi, используя StringTable

- В то время, как ресурсные файлы позволяют хранить больше, чем код программы в EXE файле, включением ресурсов **StringTable** в приложение, разработчик Delphi может легко проектировать многоязыковые приложение.

- **Ресурсы StringTable**
- Символьные строки, содержащиеся в **StringTable** не занимают память, пока они не будут определенно загружены приложением.

- При использовании ресурсов **StringTable** при создании многоязычных приложений, добавления нового языка является хорошим тоном, так как **StringTable** могут быть легко отредактированы. Более того, если **StringTable** хранится в ресурсном **DLL**, при этом не нужно будет повторно компилировать приложение.

- Как и любой другой тип ресурсов, ресурсы **StringTable** компилируются в .RES файл, который прилагается к EXE файлу приложения во время компиляции.

Создание StringTable ресурсов

- Для создания ресурса **StringTable** приложения для двух языков:
  - Создайте текстовый **.RC** файл, который содержит строковые ресурсы в директории проекта.
  - Назовите файл *StringTableLanguage.rc*. Имя может быть любое, главное, чтобы расширение файла было **.RC**

- StringTableLanguage.rc
- STRINGTABLE { 1000, "English" 1001, "Display selected" 1002, "Yes" 1003, "No" 1004, "Maybe" 2000, "Русский" 2001, "Выбор отображения" 2002, "Да" 2003, "Нет" 2004, "Возможно" }

  - Компилируйте этот **RC** файл в **RES** файл при помощи компилятора ресурсов **BRCC32**.

- Обратите внимание: Файл *StringTableLanguage.rc* может содержать любое дополнительное количество ресурсов другого типа (иконки, изображения, данные и т.д.)

Создание StringTable ресурсов

- Таблица строк начинается с ключевого слова **StringTable**. Строки заключены в фигурные скобки. Каждой строке присвоен числовой идентификатор. Символьные строки заключены в кавычки. Если Вы хотите использовать нестандартный символ, вставьте обратный слеш и далее номер символа, который Вы хотите вставить. Единственное ограничение: когда Вам нужно будет вставить обратный слеш, Вам необходимо будет вставить двойной обратный слеш.

- Например: 1, "A two\012line string" 2, "c:\\Borland\\Delphi" **Использование таблиц строк**

- Чтобы загрузить определенную строку из **StringTable** нужно использовать функцию **LoadString**. Один из параметров в вызове **LoadString** - индекс строки в таблице строк.

Связывание в приложении

- Как и любой **.RES** файл, Вы можете связать файл ресурсов с Вашим приложением просто добавив следующую инструкцию в код Вашего приложения (после **implementation**).

- {$R StringTableLanguage.RES}

- Как только файл ресурсов будет связан с Вашей программой, Вы сможете загружать ресурс из любого модуля, даже если Вы определили директиву ***$R*** в другом модуле.

LoadString

- Вот пример, как использовать функцию **LoadString** для загрузки строки из **StringTable**:

- function GetString(const Index: integer) : string;
- var buffer : array[0..255] of char;
- ls : integer;
- begin
- Result := '';
- ls := LoadString(hInstance, Index, buffer, sizeof(buffer));
- if ls <> 0 then Result := buffer;
- end;

Включение шрифта из файла ресурсов

- Чтобы подключить свой файл ресурсов, Вы должны включить директиву компилятора *{$R MyFont.RES}* в раздел **implementation**.

- Для извлечения шрифта из ресурса необходимо создать объект типа **TResourceStream** и добавить шрифт процедурой **AddFontResource**, а также Вы должны использовать сообщение **WM_FONTCHANGE**.

Включение шрифта из файла ресурсов

- В ресурсном файле нужно создать раздел *MYFONT*, который будет содержать файл шрифта.
- {$R MyFont.RES}
- { ... }
- procedure TForm1.FormCreate(Sender: TObject);
- var MyResStream: TResourceStream;
- begin
- MyResStream:=TResourceStream.Create(hInstance, 'MYFONT', RT_RCDATA); MyResStream.SavetoFile('Gen4.ttf'); AddFontResource(PChar('Gen4.ttf')); SendMessage(HWND_BROADCAST,WM_FONTCHANGE,0,0); Label1.Font.Charset:=SYMBOL_CHARSET;
- Label1.Font.Size:=24; Label1.Font.Name:='Gen4';
- end;

Как извлечь RTF текст из ресурсов и записать его на диск

- Иногда требуется сохранить текст в **RTF** формате в ресурсах (Dll, EXE и т.д.), а затем извлечь его.
- Вот основные шаги:
  - Создайте файл ресурсов
  - Подключите его к Вашему проекту
  - Загрузите файл из файла ресурсов в **TResourceStream**
  - Создайте **TFileStream** с именем файла, который Вы хотите записать на диск
  - Используйте **CopyFrom, чтобы получить данные из TResourceStream в TFileStream**
  - **Освободите оба потока**

- **Файл очень просто запишется на диск без вызова какой-либо процедуры записи.**

Как извлечь RTF текст из ресурсов и записать его на диск

- **Вот пример извлечения *test.rtf* из ресурса *TEST.RES* и сохранения его на диск как *test2.rtf* в папке приложения:**

```
procedure TfrmMain.Button1Click(Sender: TObject);
var ResStream: TResourceStream MyFileStream: TFileStream;
begin
try MyFileStream := TFileStream.Create(
    ExtractFilePath(Application.ExeName) + 'test2.rtf ', fmCreate or
    fmShareExclusive );
ResStream := TResourceStream.CreateFromID(HInstance, 1, RT_RCDATA);
    MyFileStream.CopyFrom(ResStream, 0);
finally MyFileStream.Free;
ResStream.Free;
end;
end;
```

DLL с ресурсами типа строки

- Для создания строковых ресурсов создаем текстовый файл типа **MyString.rc**, в котором пишем:
- STRINGTABLE { 00001, "String1" 00002, "String2" }

- Затем компилируем его при помощи **BRCC32 MyString.rc**, получается файл ресурсов **MyString.res**.

- Далее делаем **DLL**:

```
library MyString;
{$R MyString.res}
begin end.
```

- Компилируем при помощи Delphi и получаем **DLL** *MyString.dll*.

DLL с ресурсами типа строки

- Теперь ее можно использовать в своих программах:

```
var h: THandle;
S: array [0..255] of Char;
begin
h := LoadLibrary('MYSTRING.DLL');
if h <= 0
then ShowMessage('Не могу загрузить Dll')
else
begin
SetLength(S, 512);
LoadString(h, 1, @S, 255);
FreeLibrary(h);
end;
end;
```

Добавление форм в DLL

- **Порядок действий**
- **Создание DLL**
- **Написание кода в DLL**
- **Создание приложения**
- **Написание кода для Приложения**

Добавление форм в DLL

- **Создание DLL**
- Сначала в этом проекте нужно создать **DLL**, которая будет содержать форму. Чтобы сделать это, следуйте за этими шагами:
- Начните новый проект, используя *File -> New DLL*, у Вас будет шаблон **DLL**, в котором Вы должны создать форму, для этого выполните *File -> New Form*
- Затем нужно добавить форму в проект. Но сначала сохраним форму под каким-либо именем (например *DllForm*) и добавим ее в проект, используя *File -> Add To Project...*, в диалоговом окне выберите сохраненную форму (*DllForm.pas*)
-

Добавление форм в DLL

- **Создание приложения**
- Для создания приложения нужно выполнить следующие шаги:
- Начните новый проект, используя *File -> New Application*, затем добавьте две кнопки на главную форму: одну для показа обычной формы в **DLL**, а другую, чтобы показать ее модально.
- Измените надписи на кнопках, чтобы они отображали суть, что они будут делать и измените размеры в случае необходимости.
- Теперь мы имеем простое приложение, которое не делает ничего. Нам нужно добавить код для кнопок, чтобы они выполняли требуемые функции.
-

Добавление форм в DLL

- **Написание кода для Приложения**
- Для начала нужно написать объявления процедуры и функции, которыми мы будем вызывать **DLL**. Это делается перед разделом **implementation**:
- procedure ShowForm;stdcall; external 'Project1dll.dll' name 'ShowDllForm';
- function ShowFormModal:integer;stdcall; external 'Project1dll.dll' name 'ShowDllFormModal';

- Снова используем **stdcall**, поскольку так была объявлена процедура в **DLL**.

Добавление форм в DLL

- Затем мы должны вызвать эти процедуры (функции), это делается в событии *OnClick* кнопок, которые мы создали ранее:
- procedure TForm1.Button1Click(Sender: TObject);
- begin
- ShowFormModal;
- end;

- procedure TForm1.Button2Click(Sender: TObject);
- begin
- ShowForm; e
- nd;

- Все, что мы здесь делаем - вызываем процедуры в **DLL** в зависимости от того, какая кнопка нажата.