Delphi-Source.de - Tutorials

DLL-Tutorial von Martin Strohal

Einleitung

Dieses DLLs-Tutorial soll Delphi-Einsteigern erklären, wie man DLLs erstellt und wie man sie anschließend in eigene Programme einbindet. Doch zu allererst: Was sind DLLs und wozu braucht man sie überhaupt? "DLL" ist die Abkürzung für Dynamic Link Library, also eine Programmbibliothek, die erst zur Laufzeit in eine Anwendung eingebunden wird. Eine DLL enthält Prozeduren und Funktionen. Sie kann aus mehreren Units bestehen, und es ist sogar möglich, Formulare ("Fenster") zu verwenden. Und wozu braucht man das nun? Eine DLL hat folgende Vorteile:

- Prozeduren, die von mehreren Anwendungen genutzt werden, können in DLLs ausgelagert werden. Die einzelnen Anwendungen werden dadurch kleiner, und der gemeinsam genutzte Code muss nur noch an einer Stelle gepflegt werden.
- Ein Programmteil, der sich des öfteren ändert, kann in eine DLL ausgelagert werden, damit bei Updates nur noch eine kleine DLL weitergegeben muss.
- Prozeduren und Funktionen in einer DLL können auch von anderen Programmiersprachen aus genutzt werden. So kann man aus einer C++-Anwendung auf eine Delphi-DLL zugreifen und umgekehrt.

Jetzt wissen Sie, welche wichtigen Programmierkenntnisse Sie verpassen, wenn Sie nicht weiterlesen.

Das Grundgerüst

Nun geht's los: Das Erstellen einer DLL mit Delphi. (Die hier angegebene Beschreibung bezieht sich auf Delphi 5. Für ältere Versionen sind möglicherweise Änderungen nötig.) Wählen Sie in der Delphi-Entwicklungsumgebung den Menüpunkt "Neu..." im Menü "Datei" aus. Es öffnet sich ein Dialogfenster, in dem auch ein Punkt "DLL" vorhanden ist. Dieser muss mit der Maus markiert und anschließend über "OK" bestätigt werden. Und schon haben wir das Grundgerüst einer DLL vor uns. Auf die darin enthaltene Warnung wird später noch eingegangen.

```
library Project1;

{ Wichtiger Hinweis zur DLL-Speicherverwaltung: ShareMem muss sich
  in der ersten Unit der unit-Klausel der Bibliothek und des Projekts
  befinden (Projekt-Quelltext anzeigen), falls die DLL Prozeduren oder
  Funktionen exportiert, die Strings als Parameter oder
  Funktionsergebnisse weitergibt. Dies trifft auf alle Strings zu, die
  von oder zur DLL weitergegeben werden -- auch diejenigen, die sich
  in Records oder Klassen befinden. ShareMem ist die Schnittstellen-
  Unit zu BORLNDMM.DLL, der gemeinsamen Speicherverwaltung, die
  zusammen mit der DLL weitergegeben werden muss. Um die Verwendung
  von BORLNDMM.DLL zu vermeiden, sollten String-Informationen als
  PChar oder ShortString übergeben werden. }

uses
```

```
  SysUtils,
  Classes;

{$R *.RES}

begin

end.
```

Nun geht es daran, Funktionen und Prozeduren in die DLL einzubauen. Um das Tutorial ganz einfach zu halten, werden wir nur eine einzige Funktion implementieren, deren einzige Aufgabe darin besteht, zwei Zahlen zusammenzuzählen. Dazu mehr in Schritt 2.

Funktion hinzufügen

Die Funktion zum Addieren zweier Zahlen sieht folgendermaßen aus:

```
function addiere(zahl1, zahl2: integer): integer;
begin
  result:=zahl1+zahl2;
end;
```

Wer hierbei jetzt große Überraschungen erlebt hat, sollte sich besser zuerst mit Pascal-Grundlagen befassen, bevor er hier weiterliest. Nun muss diese Funktion nur noch in das DLL-Grundgerüst eingefügt werden. Zuvor ist aber noch anzumerken, dass es bei DLLs eine Besonderheit gibt: Wenn es möglich sein soll, dass Funktionen oder Prozeduren einer DLL auch von anderen Programmiersprachen genutzt werden können, muss das reservierte Wort **stdcall** hinter den Funktions- bzw. Prozedurkopf geschrieben werden. In unserm Fall also:

```
function addiere(zahl1, zahl2: integer): integer; stdcall;
```

Bei stdcall handelt es sich um eine sog. "Aufrufkonvention". Weitere sind register, cdecl und safecall. Sie unterscheiden sich darin, wie Parameter übergeben werden und ob dabei CPU-Register verwendet werden. Für Aufrufe der Windows-API wird stdcall verwendet; register ist das effizienteste. Auf jeden Fall muss darauf geachtet werden, dass sowohl in der DLL auch beim Aufruf aus einer Anwendung die gleiche Aufrufkonvention verwendet wird. Und ganz wichtig auch in dem Zusammenhang: Bei DLLs ist die Schreibweise der Namen von Funktionen und Prozeduren, die von außen aufrufbar sein sollen - im Gegensatz zum "normalen" Pascal - von Bedeutung. Also auf Groß- und Kleinschreibung achten!

Der exports-Abschnitt

Nun fügen wir unsere kleine Funktion in das von Delphi vorgegebene DLL-Grundgerüst ein und erhalten dadurch folgenden Code (die Kommentarzeilen wurden entfernt):

```
library Project1;

uses
  SysUtils,
  Classes;

{$R *.RES}
```

```
function addiere(zahl1, zahl2: integer): integer; stdcall;
begin
  result:=zahl1+zahl2;
end;

begin

end.
```

In einer normalen Unit müsste der Funktionsname jetzt noch in einem Interface-Teil eingefügt werden. Bei DLLs funktioniert das etwas anders. Funktionen bzw. Prozeduren, die von außen - also aus einer anderen Anwendung - ansprechbar sein sollen, müssen in einem exports-Abschnitt bekanntgegeben werden, allerdings nur mit ihrem Namen, nicht mit evtl. Parametern. Dieser Abschnitt befindet sich in der Regel am Ende der DLL. Wenn wir weitere Funktionen benötigen würden, um unser Ergebnis zu berechnen, die aber nicht von außen direkt aufgerufen werden sollten, müssten diese auch nicht im exports-Abschnitt stehen. Komplett sieht unsere DLL also folgendermaßen aus:

```
library Project1;

uses
  SysUtils,
  Classes;

{$R *.RES}

function addiere(zahl1, zahl2: integer): integer; stdcall;
begin
  result:=zahl1+zahl2;
end;

exports
  addiere;

begin

end.
```

Jetzt speichern wir die Datei unter dem Namen rechnen.dpr. Beim Kompilieren (Strg+F9) entsteht dann unsere DLL, rechnen.dll.

Möglichkeiten der Einbindung

Jetzt wollen wir unsere neue DLL natürlich auch aus einer Delphi-Anwendung heraus nutzen. Dazu müssen wir sie in das Programm einbinden. Dafür gibt es zwei Möglichkeiten: die Einbindung zum Startzeitpunkt der aufrufenden Anwendung und die spätere Einbindung bei Bedarf. Die **Einbindung zum Startzeitpunkt (Load-Time Dynamic Linking)** ist einfach zu handhaben, hat jedoch den Nachteil, dass gleich zu Programmstart alle in das Programm auf diese Art eingebundenen DLLs mitgeladen werden, auch wenn sie vielleicht nicht benötigt werden. Dies braucht Speicherplatz und verlängert den Startvorgang der Anwendung. Ist eine DLL nicht vorhanden, tritt gleich beim Start ein Fehler auf. **Zur Laufzeit engebundene DLLs (Run-Time Dynamic Linking)** werden dagegen nur dann in den Arbeitsspeicher

geladen, wenn sie auch wirklich gebraucht werden. Man kann also noch bis kurz vor den Aufruf den Namen der DLL festlegen und prüfen, ob diese auf dem System auch vorhanden ist. Diese Einbindungsart hat den Nachteil, dass sie komplizierter zu programmieren ist, da sich der Entwickler selber um den Ladevorgang und das anschließende Entfernen der DLL aus dem Arbeitsspeicher kümmern muss. Im Folgenden soll auf beide Möglichkeiten eingegangen werden.

Einbinden zum Ladezeitpunkt

Beginnen wir also mit der Einbindung zum Ladezeitpunkt. Dazu legen wir in unserem Projekt am besten eine neue Unit an (wiederum über das Menü Datei/Neu...) und deklarieren unsere neue Funktion, als ob wir sie ganz normal direkt in die Unit eintippen würden:

```
unit Unit1;

interface
  function addiere(zahl1, zahl2: integer): integer; stdcall;

implementation

function addiere(zahl1, zahl2: integer): integer; stdcall;
external 'rechnen.dll';

end.
```

Im Implementationsteil schreiben wir statt der eigentlichen Funktion aber nur noch "external" gefolgt von dem Namen der DLL. Die DLL sollte sich dazu im gleichen Verzeichnis wie die Anwendung befinden. Sollen mehrere Anwendungen aus verschiedenen Verzeichnissen auf eine DLL zugreifen können, ist sie am besten im System-Verzeichnis (z.B. C:\Windows\System) aufgehoben. Verwendet wird die Funktion anschließend ganz normal. Überall, wo obige Unit "Unit1" über "uses" eingebunden ist, kann geschrieben werden:

```
  x:=addiere(12, 3);
```

Einbinden zur Laufzeit (1)

Damit kommen wir zur komplizierteren Art der Einbindung, der Einbindung zur Laufzeit. Wir gehen hier wieder von einer leeren Unit "Unit1" aus. Zuerst müssen wir einen Typ für die aufzurufende Funktion erstellen:

```
type TSummenFunktion = function(zahl1, zahl2: integer): integer;
```

Hinter dem Gleichheitszeichen folgt das Wort function oder procedure und anschließend in Klammern die Parameter dazu. Der eigentliche Funktions-/Prozedurname wird weggelassen. Nun kommen wir zu der Stelle, von der aus die Funktion in der DLL benötigt wird. Nehmen wir an, das geschieht wie im Beispiel der Einbindung bei Programmstart in einer extra "Schnittstellen-Unit", in der sich nur die Schnittstellen zu Funktionen und Prozeduren in DLLs befinden. Zur Unterscheidung mit der Funktion in der DLL, nennen wir die Schnittstellen-Funktion "addieren". Folgende Variablen müssen definiert werden:

```
function addieren(zahl1, zahl2: integer): integer;
var SummenFunktion: TSummenFunktion;
    Handle: THandle;
```

Schon an der Verwendung eines Handles merkt man, dass man hier wieder auf Windows-API-Ebene arbeitet

und nicht die Vereinfachungen von Delphi genießen kann. Laden wir also unsere DLL. Hier muss nun der DLL-Dateiname angegeben werden; zuvor könnte geprüft werden, ob diese Datei auch existiert.

```
  Handle:=LoadLibrary('rechnen.dll');
```

Im Folgenden wird geprüft, ob das Handle unleich Null ist, das Laden also erfolgreich war, und anschließend die Adresse der von uns benötigten Funktion "addiere" ermittelt.

```
  if Handle <> 0 then begin
    @SummenFunktion := GetProcAddress(Handle, 'addiere');
    if @SummenFunktion <> nil then begin
```

Hier geht es nun auch erst weiter, wenn die Funktion "addiere" in der DLL gefunden worden ist. Nun kommt der eigentliche Aufruf der Funktion:

```
      result:=SummenFunktion(zahl1, zahl2);
    end;
```

Und zu guter Letzt muss die DLL wieder aus dem Arbeitsspeicher freigegeben werden:

```
    FreeLibrary(Handle);
```

Einbinden zur Laufzeit (2)

Nun hat sich also folgende Unit ergeben:

```
unit Unit1;

interface

uses windows;

type
  TSummenFunktion = function(zahl1, zahl2: integer): integer;
  function addieren(zahl1, zahl2: integer): integer;

implementation

{$R *.DFM}

function addieren(zahl1, zahl2: integer): integer;
var SummenFunktion: TSummenFunktion;
    Handle: THandle;
begin
  Handle:=LoadLibrary('rechnen.dll');
  if Handle <> 0 then begin
    @SummenFunktion := GetProcAddress(Handle, 'addiere');
    if @SummenFunktion <> nil then begin
      result:=SummenFunktion(zahl1, zahl2);
    end;
    FreeLibrary(Handle);
  end;
end;
```

```
end.
```

Auf den ersten Blick ersichtlich, dass die Einbindung zur Laufzeit etwas aufwändiger ist als die zum Programmstart. Allerdings hat sie - wie erwähnt - auch ihre Vorteile. Jetzt fehlt nur noch eins: Was bedeutet der große Hinweis, der im automatisch erzeugten DLL-Grundgerüst stand?

Strings als Parameter oder Rückgabewert

Zur Erinnerung, so sah er aus:

```
{ Wichtiger Hinweis zur DLL-Speicherverwaltung: ShareMem muss sich
  in der ersten Unit der unit-Klausel der Bibliothek und des Projekts
  befinden (Projekt-Quelltext anzeigen), falls die DLL Prozeduren oder
  Funktionen exportiert, die Strings als Parameter oder
  Funktionsergebnisse weitergibt. Dies trifft auf alle Strings zu, die
  von oder zur DLL weitergegeben werden -- auch diejenigen, die sich
  in Records oder Klassen befinden. ShareMem ist die Schnittstellen-
  Unit zu BORLNDMM.DLL, der gemeinsamen Speicherverwaltung, die
  zusammen mit der DLL weitergegeben werden muss. Um die Verwendung
  von BORLNDMM.DLL zu vermeiden, sollten String-Informationen als
  PChar oder ShortString übergeben werden. }
```

Im Großen und Ganzen eigentlich selbsterklärend: Es ist zu empfehlen, keine Stringwerte zwischen Anwendung und DLL zu übergeben. Denn um das machen zu können, muss die Unit ShareMem eingebunden und die BORLNDMM.DLL mit der Anwendung weitergegeben werden. So etwas ist immer problematisch, da sich die DLL von Delphi-Version zu Delphi-Version ändern kann (früher hieß sie meines Wissens DELPHIMM.DLL). Es ist also um Einiges einfacher, nur ShortStrings bzw. PChars zu übergeben. Wird die BORLNDMM.DLL verwendt, so muss die Unit ShareMem sowohl in der DLL als auch in der einbindendn Unit des aufrufenden Programms an erster Stelle der uses-Klausel genannt werden. Sollen AnsiStrings oder dynamische Arrays benutzt werden, dann ist die DLL nur mit Delphi nutzbar. Da die Speicherverwaltungs-DLL sich bei verschiedenen Delphi-Versionen unterscheidet, kann eine solche DLL auch nur mit einer Anwendung zusammen verwendet werden, die mit der gleichen Compilerversion compiliert wurde. Auf stdcall kann dann übrigens verzichtet werden. Nach diesem Tutorial sollte es nun für Sie kein Problem sein, eigene DLLs zu erstellen und diese in eigenen Anwendungen zu verwenden.
