Delphi 7 Studio

Olivier Dahan

Paul Toth

ISBN : 2-212-11143-6

EYROLLES

19

La gestion des bibliothèques dynamiques (DLL)

Microsoft Windows fourmille de fichiers dotés de l'extension `DLL`, *dynamic-link library*, ou bibliothèque liée dynamiquement. La quasi-totalité de ce système d'exploitation est basée sur l'utilisation de ces fichiers. En fait, la moindre application Windows, dont les applications Delphi, utilise obligatoirement une ou plusieurs DLL, même de façon indirecte. Dans ce chapitre, nous allons voir l'utilité de ces fichiers, la façon de les utiliser avec Delphi et même comment il est possible de créer nos propres DLL.

L'utilisation des DLL

Rappels sur les DLL

Les DLL permettent à plusieurs applications de partager une même ressource ou une même portion de code. Le principe est assez similaire à celui des unités de Delphi ; plusieurs projets Delphi peuvent partager les mêmes unités, et plusieurs applications Windows peuvent partager les mêmes DLL. Pour ne pas semer le doute dans les esprits, précisons que la comparaison vaut ce qu'elle vaut : si Delphi regroupe les unités pour les compiler en un seul exécutable, il n'est pas possible de faire de même avec des DLL.

Toutefois, les DLL et les unités source de Delphi ont plus d'une similitude :

- La mise à jour d'une unité peut provoquer des erreurs de compilations d'un projet, et la mise à jour d'une DLL peut entraîner une erreur d'exécution d'un programme.
- La présence de deux unités homonymes (nommées identiquement) dans les répertoires de recherche de Delphi peut l'induire en erreur, la présence de deux DLL portant le même nom dans les répertoires de recherche de Windows peut provoquer la même erreur.

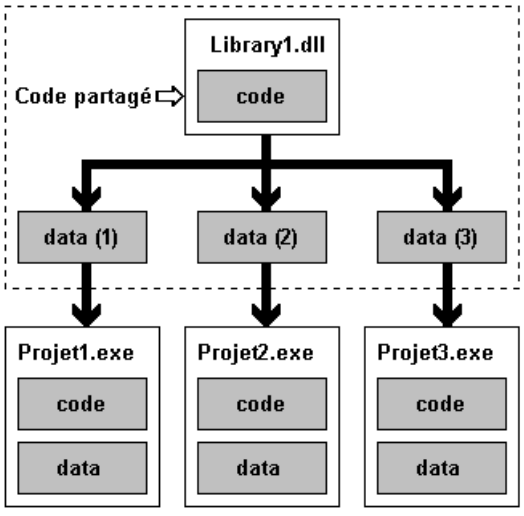
- Il n'est pas possible de donner le même nom à deux unités différentes au sein d'un même projet ; on ne peut pas davantage charger en mémoire deux DLL différentes portant le même nom.
- Les unités déclarent leurs fonctions dans une interface ; les DLL exportent leurs fonctions dans une table d'exportation.
- Une unité peut utiliser d'autres unités ; une DLL peut dépendre d'autres DLL.

Elles possèdent également quelques différences :

- Les unités peuvent déclarer des fonctions, des constantes, des variables… les DLL ne peuvent exporter que des fonctions.
- Une unité Delphi peut être compatible avec plusieurs systèmes d'exploitation (c'est le cas avec Kylix sous Linux) ; une DLL, *stricto sensu*, ne fonctionne que sous Microsoft Windows (ou éventuellement dans un environnement qui se fait passer pour lui comme Wine sous Linux). Si d'autres OS possèdent également des bibliothèques partagées (les `.so` sous Linux par exemple), leur format est propre à la plate-forme. On pourra cependant compiler le même projet sous Delphi pour produire un fichier `.dll`, et sous Kylix pour produire un fichier `.so`.
- Les unités sont obligatoirement écrites en Pascal ; les DLL peuvent être réalisées avec différents langages de programmation.

La figure 19-1 illustre le principe général des DLL. Les trois projets présentés possèdent chacun une zone mémoire contenant du code et une zone mémoire renfermant des données. Ils utilisent tous les trois une même DLL, `Library1.dll`, qui possède un code partagé par les trois applications. Le code DLL n'est présent qu'une seule fois en mémoire et s'accompagne d'une zone de données réservée pour chaque application.

Figure 19-1
Les instances d'une DLL



Le schéma ne reprend pas la notion de ressource partagée. Si la DLL contient des ressources (au sens Windows du terme), celles-ci pourront être partagées entre les différents projets exactement comme il en est de la zone de code.

Vous remarquerez que ce schéma ne fait pas explicitement référence à Delphi, et c'est tout à fait normal, car bien que Delphi permette de réaliser aussi bien des exécutables Windows que des DLL, il est tout à fait possible de combiner des programmes Windows et des DLL écrites par tout autre langage de programmation. C'est d'ailleurs là un des grands avantages des DLL ; la quasi-totalité des langages de programmation sous Windows est capable de faire appel à une DLL ; ce n'est pas toujours le cas pour des technologies plus récentes comme ActiveX ou DCOM.

Déclaration des fonctions

Lorsque vous réalisez une application Delphi, vous utilisez automatiquement un bon nombre de fonctions de l'API Windows. Si vous consultez l'unité `Windows.pas` et si vous y recherchez le mot-clé `external` (en minuscules pour le distinguer de `$EXTERNALSYM`), vous trouverez plus de 2000 lignes qui l'utilisent. Elles ne font pourtant référence qu'à une dizaine de DLL dont les principales sont : `kernel32`, `gdi32` et `user32`. C'est cela l'API Windows : une poignée de DLL qui définissent des milliers de fonctions permettant d'interagir avec le système d'exploitation.

Prenons l'exemple de l'une d'entre elles, celle qui permet tout simplement de créer un objet fenêtré : `CreateWindowEx` ; elle est utilisée par tous les dérivés de `TWinControl`.

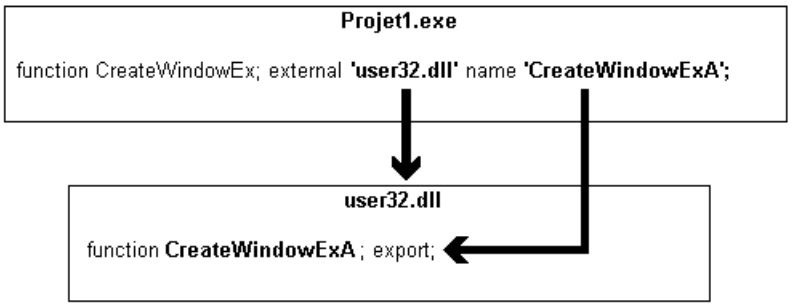
```
Interface
[...]
function CreateWindowEx(dwExStyle: DWORD; lpClassName: PChar;
  lpWindowName: PChar; dwStyle: DWORD; X, Y, nWidth, nHeight: Integer;
  hWndParent: HWND; hMenu: HMENU; hInstance: HINST; lpParam: Pointer): HWND; stdcall;
[...]
Implementation
[...]
const
 user32='user32.dll';
[...]
function CreateWindowEx; external user32 name 'CreateWindowExA';
```

La fonction est déclarée en deux temps ; l'interface se contente de donner ses paramètres et sa convention d'appel sur laquelle nous reviendrons plus tard. L'implémentation précise qu'elle est externe au projet, qu'elle se trouve dans la DLL `user32` et que son vrai nom est `CreateWindowExA` (voir figure 19-2). Le mot-clé `name` est facultatif ; il est utilisé ici car le nom de la fonction Delphi diffère du nom de la fonction externe.

La déclaration en deux temps n'est pas obligatoire ; la fonction peut être définie en une seule fois, aussi bien en interface qu'en implémentation selon la portée qu'on désire lui

donner. Il est cependant courant d'indiquer les références aux DLL dans la partie implémentation car cette information n'a pas réellement lieu d'apparaître dans la partie interface.

Figure 19-2
Les procédures externes



La convention d'appel indique comment les paramètres sont passés à la fonction. Si l'indication de convention est erronée, il y a toutes les chances que l'appel à la fonction provoque une erreur d'exécution. En effet, l'ordre de passage des paramètres est inversé entre les conventions `register` et `pascal`, et les conversions `cdecl`, `stdcall` et `safecall`. La convention `register` utilise jusque trois des registres du processeur pour passer des paramètres avant d'utiliser la pile comme tous les autres modes. En convention `cdecl`, il appartient au programme appelant de supprimer les paramètres de la pile alors que, dans tous les autres modes, c'est la fonction qui le fait. Enfin, la convention `safecall` implémente la gestion d'erreur et d'exception COM. Delphi utilise par défaut la convention `register`, alors que la plupart des DLL de Windows utilisent la convention `cdecl` ou `safecall` ; vous devrez donc très souvent spécifier la convention d'appel quand vous déclarerez une fonction externe.

La liaison statique

Lorsqu'on déclare des références externes, telles que celle que nous venons de voir, l'exécutable est directement lié à la DLL qu'il utilise. Le simple fait de déclarer la fonction implique que Windows s'occupe du chargement de la DLL en même temps que le programme, et que la fonction soit immédiatement disponible dans l'application comme n'importe quelle autre fonction locale.

> Bien que l'unité `Windows.pas` déclare plusieurs milliers de fonctions externes, seules les quelques fonctions qui seront réellement utilisées par l'application seront effectivement liées à l'exécutable final.

Nous pouvons mettre en valeur l'aspect statique de ces liens en déclarant dans un projet une procédure qui n'existe assurément pas :

```
procedure QuiNExistePas; external 'NulPart.dll';
```

La compilation ne pose aucun problème ; en revanche, l'exécution du projet ne sera pas possible pour peu qu'il invoque au moins une fois cette procédure. Invoquons-la par exemple dans l'événement de création d'une fiche :

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 QuiNExistePAs;
end;
```

Nous obtenons une erreur d'exécution retournée par Windows lui-même (voir figure 19-3).

Figure 19-3
Lien statique invalide

Même si nous déclarons que la fonction se trouve dans une DLL existante, `kernel32.dll` par exemple, il sera toujours impossible d'exécuter l'application (voir figure 19-4).

Figure 19-4
Lien statique manquant

La déclaration par nom oblige Windows à rechercher la fonction dans la liste d'exportation de la DLL afin d'obtenir son adresse. Il est possible de lui éviter cette tâche en indiquant directement l'index de la fonction dans la table d'exportation de la DLL par le mot-clé `index`. N'utilisez jamais cette technique à moins d'avoir une très bonne raison pour le faire ; elle n'offre aucun contrôle sur le lien. Il suffit d'ajouter une fonction à la DLL pour provoquer un décalage dans la table d'exportation de la DLL, ce qui rendrait invalides, ou pire erronés, tous les index utilisés.

La liaison dynamique

Il existe différents cas de figure dans lesquels la liaison statique ne peut être utilisée :

- Le programme ignore le nom de la DLL qu'il doit utiliser car celui-ci est paramétrable.
- Le programme ignore le nom de la fonction qu'il doit utiliser car celui-ci est paramétrable.

- Le programme doit pouvoir fonctionner sans la DLL, par exemple, sans la présence de TCP/IP.
- Le grand nombre de DLL utilisées par le projet nécessite une gestion plus fine de leur chargement en mémoire.
- Le programme s'adresse à des utilisateurs débutants ne pouvant se satisfaire des messages d'erreurs Windows.

Quel que soit le cas de figure, il suffit de recourir à trois fonctions pour résoudre le problème.

1. `LoadLibrary` qui permet de charger la bibliothèque en mémoire.
2. `GetProcAddress` qui permet de retrouver l'adresse d'une procédure de cette DLL.
3. `FreeLibrary` qui doit être invoquée pour libérer la DLL.

Du côté programme, quelques petits aménagements s'imposent ; les déclarations de fonctions deviennent des déclarations de variables :

```
var
 MaDLL:THandle; // handle de la DLL
 MaFonction: function:integer;

procedure TForm1.FormCreate(Sender :TObject) ;
begin
 MaDLL:=LoadLibrary('MaDLL.DLL');
 if MaDLL=0 then raise Exception.Create('Le fichier MaDLL.DLL est introuvable !');
 @MaFonction:=GetProcAddress(MaDLL,'MaFonction');
 if @MaFonction=nil then raise
  Exception.Create('MaDLL.DLL ne contient pas la fonction "MaFonction" !');
end;

procedure TForm1.FormDestroy(Sender :TObject);
begin
 if MaDLL<>0 then FreeLibrary(MaDLL);
end;
```

Dans cet exemple, la DLL est chargée en mémoire lors de la création de la fiche `Form1`, et elle est libérée lors de sa destruction.

La création de DLL

Du programme à la bibliothèque

Delphi permet de créer très simplement des DLL. Le principe est en fait très similaire à celui de la création d'application : à tel point qu'il suffit de modifier un seul mot-clé dans le fichier source du projet, le `DPR`, pour faire d'une application Delphi une DLL.

```
Library MaDLL;
Uses
 Dialogs;
begin
 ShowMessage('DLL Chargée') ;
end.
```

En remplaçant le mot-clé `program` par le mot-clé `library`, on indique à Delphi la nature de l'application. La section `begin end` principale est invoquée lors du chargement de la DLL ; vous pouvez y placer un code d'initialisation.

> Les variables globales déclarées dans une DLL ne sont pas partagées entre les différentes instances de la DLL. Bien au contraire, chaque DLL possède son propre jeu de variables totalement indépendant. Delphi n'offre pas d'outil particulier pour gérer des données communes, et Borland recommande l'utilisation des fichiers projetés en mémoire.

Nous l'avons évoqué plus haut, à l'image des unités la DLL doit déclarer les symboles qu'elle publie ; on parle dans ce cas d'exportation. Pour déclarer les fonctions publiques de la DLL, il suffit de les énumérer dans une section `exports`. Nous retrouvons dans cette clause les options que nous avons évoquées pour le mot-clé `external`, à savoir la possibilité – non recommandée – de spécifier un index de fonction, et celle de renommer la fonction lors de son exportation. La convention d'appel, quant à elle, doit être spécifiée dans la déclaration de la fonction car cette information est nécessaire pour sa compilation.

```
exports
 Procedure1,
 Procedure2 name 'ProcedureDeux',
 Procedure3 index 12;
```

Il est courant de placer cette clause juste avant le code d'initialisation de la DLL, mais elle peut également être placée en début de source ou dans la partie interface ou implémentation d'une unité.

Le point d'entrée multithread

Si vous devez gérer le cas particulier des appels multithreads, vous trouverez dans la documentation Windows une référence à la fonction `DLLEntryPoint`. Celle-ci n'est pas directement disponible sous Delphi, mais l'unité `System` déclare la variable `DLLProc que` vous pouvez utiliser pour indiquer l'adresse d'une fonction qui sera invoquée en lieu et place de la `DLLEntryPoint`. Voici le code d'initialisation qu'il faut utiliser pour reproduire le fonctionnement traditionnel des DLL :

```
library Library1;

Uses
 Windows;

procedure LibraryProc(Reason:integer);
 begin
 end;

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

Delphi ne reprend qu'un seul des trois paramètres de la fonction `DLLEntryPoint`. Le paramètre `hInstance` est disponible dans une variable globale du même nom, mais le paramètre `lpvReserved` n'est en revanche pas disponible.

Exemple de DLL

À titre d'exemple, nous allons créer un programme qui fera des opérations mathématiques sur deux nombres. Les fonctions utilisées seront bien évidemment implémentées dans une DLL.

```
function ConvertEuro(Montant:double):double;
function ConvertFranc(Montant:double):double;
```

Les fonctions `ConvertEuro` et `ConvertFranc` acceptent un paramètre `Montant` qui est converti selon le cas de franc à euro, ou inversement.

```
library Convert;

Const
  DevEuro=6.55957;

function ConvertEuro(Montant:double):double; stdcall;
begin
 Result:=Montant*DevEuro;
end;

function ConvertFranc(Montant:double):double; stdcall;
begin
 Result:=Montant/DevEuro;
end;

exports
 ConvertEuro,
 ConvertFranc;

begin
{partie initialisation de la dll}
end.
```

Ce source produit une DLL nommée `Convert.dll` qui exporte les deux fonctions qui nous intéressent. Elles utilisent toutes les deux la convention d'appel `stdcall` qui assure à la DLL une compatibilité avec le plus grand nombre de langages de programmation.

Sous Windows 32 bits, il n'est plus nécessaire d'indiquer le mot-clé `export` sur une fonction qui doit être exportée. Delphi accepte cependant ce paramètre pour des raisons de compatibilité ascendante.

Et voici un exemple d'utilisation de cette DLL :

1. Ouvrez un nouveau projet.
2. Placez-y une boîte d'édition `Edit1`.

3. Placez-y un bouton que vous nommez `Euro`.
4. Placez-y un bouton que vous nommez `Francs`.
5. Saisissez le code ci-après en prenant soin d'utiliser les événements `OnClick` des boutons :

```
implementation

{$R *.dfm}

function ConvertEuro(Montant:double):double; stdcall; external 'Convert.dll';
function ConvertFranc(Montant:double):double; stdcall; external 'Convert.dll';

procedure TForm1.EuroClick(Sender: TObject);
begin
 Edit1.Text:=FloatToStr(ConvertEuro(StrToFloat(Edit1.Text)));
end;

procedure TForm1.FrancsClick(Sender: TObject);
begin
 Edit1.Text:=FloatToStr(ConvertFranc(StrToFloat(Edit1.Text)));
end;
Et le même exemple en liaison dynamique cette fois :
implementation

{$R *.dfm}

var
 Convert     :THandle;
 ConvertEuro :function (Montant:double):double; stdcall;
 ConvertFranc:function (Montant:double):double; stdcall;

procedure TForm1.FormCreate(Sender: TObject);
begin
 Convert:=LoadLibrary('Convert.dll');
 if Convert=0 then raise Exception.Create('La DLL de conversion est introuvable');
 @ConvertEuro:=GetProcAddress(Convert,'ConvertEuro');
 if @ConvertEuro=nil then
  raise Exception.Create('La fonction ConvertEuro est introuvable');
 @ConvertFranc:=GetProcAddress(Convert,'ConvertFranc');
 if @ConvertFranc=nil then
  raise Exception.Create('La fonction ConvertFranc est introuvable');
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
 if Convert<>0 then FreeLibrary(Convert);
end;

procedure TForm1.EuroClick(Sender: TObject);
```

```
begin
 if @ConvertEuro<>nil then
  Edit1.Text:=FloatToStr(ConvertEuro(StrToFloat(Edit1.Text)));
end;

procedure TForm1.FrancsClick(Sender: TObject);
begin
 if @ConvertFranc<>nil then
  Edit1.Text:=FloatToStr(ConvertFranc(StrToFloat(Edit1.Text)));
end;
```

Réalisation de plug-ins

Un des usages les plus intéressants de la DLL est son intégration dans une application sous forme de *plug-in*. En l'occurrence, en respectant une certaine norme, la DLL viendra ajouter des options dans un menu de l'application hôte, comme Microsoft Word par exemple, ou ajoutera des fonctions telles que l'affichage d'un nouveau format graphique sous le navigateur de Netscape.

Afin d'en bien comprendre le mécanisme, nous allons créer à la fois une application hôte et un de ses plug-ins.

Notre application hôte considérera qu'une DLL est un plug-in si elle exporte la fonction `InitPlugin`. Cette fonction permettra au plug-in de venir s'enregistrer dans la barre de menu de l'application.

La classe TPlugin

Pour faciliter le traitement des plug-ins, nous allons créer une classe `TPlugin` qui se chargera de tout. Afin de partager la structure `TPluginOptions` entre l'application hôte et les plug-ins, nous placerons le code dans une unité `Plugins.pas` dédiée.

```
unit Plugins;

interface

uses
 Windows,Classes,SysUtils;

type
 TPluginOptions=record
  Caption :PChar;
  OnClick :procedure; stdcall;
  ShortCut:TShortCut;
  Bitmap  :THandle;
 end;

 TInitPlugin=procedure(var Options:TPluginOptions); stdcall;

 TPlugin=class
 private
```

```
  fHandle :THandle;
  fInit   :TInitPlugin;
  fOptions:TPluginOptions;
 public
  Constructor Create(FileName:string);
  Destructor Destroy; override;
  Procedure DoClick;
  Property Caption:PChar read fOptions.Caption;
  Property ShortCut:TShortCut read fOptions.ShortCut;
  Property Bitmap:THandle read fOptions.Bitmap;
 end;
```

La fonction d'initialisation des plug-ins, `TInitPlugin`, renseigne une structure qui centralise toutes les informations dont nous avons besoin pour l'exploiter. Cela permet de n'exporter qu'une seule fonction dans la DLL au lieu de déclarer une fonction par paramètre.

Nous prendrons soin d'exclure de cette structure toute information liée à la VCL et toute référence aux variables dynamiques comme les strings, afin de supporter des plug-ins écrits dans d'autres langages de programmation. Dans le même souci d'ouverture, toutes les fonctions publiées utilisent la convention d'appel `stdcall`.

Le `Constructor` du `TPlugin` charge la DLL passée en paramètre et recherche la fonction d'initialisation. En cas d'erreur, une exception est générée et le `Destructor` est automatiquement invoqué par Delphi.

```
implementation

constructor TPlugin.Create(FileName:string);
 begin
 // Chargement de la DLL
  fHandle:=LoadLibrary(PChar(FileName));
  if fHandle=0 then raise Exception.Create('Impossible de charger la DLL');
 // Recherche de la fonction d'init
  fInit:=GetProcAddress(fHandle,'InitPlugin');
  if @fInit=nil then raise Exception.Create('Ce n''est pas un plugin');;
 // Lecture des options
  fInit(fOptions);
 end;

destructor TPlugin.Destroy;
 begin
 // Libération de la DLL
  if fHandle<>0 then FreeLibrary(fHandle);
  inherited;
 end;
```

Les propriétés de la classe renvoyant directement les valeurs de la structure `fOptions`, il ne reste plus qu'à traiter le clic.

```
procedure TPlugin.DoClick;
 begin
  if @fOptions.OnClick<>nil then fOptions.OnClick;
 end;
```

L'application hôte

Nous nous contenterons d'une application simpliste qui charge les plug-ins présents dans son répertoire :

1. Créez une nouvelle application.
2. Placez un menu sur la fiche pour recevoir les plug-ins.
3. Placez une boîte liste qui contiendra la liste des DLL trouvées.
4. Ajoutez une liste privée `PlugList`.
5. Ajoutez une méthode `PluginClick` pour gérer le clic sur le menu des plug-ins.

```
type
  TForm1 = class(TForm)
    MainMenu1: TMainMenu;
    PluginsMenu: TMenuItem;
    ListBox1: TListBox;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { Déclarations privées }
    PlugList:TList;
    procedure PluginClick(Sender:TObject);
  public
    { Déclarations publiques }
  end;
```

La liste des plug-ins sera créée en même temps que la fiche sur son événement `OnCreate` ; elle sera détruite sur `OnDestroy`.

```
uses
 Plugins;

procedure TForm1.FormCreate(Sender: TObject);
var
 Path     :string;
 SearchRec:TSearchRec;
 PlugIn   :TPlugIn;
 Menu     :TMenuItem;
begin
// La liste des plugins chargés
 PlugList:=TList.Create;
// Le répertoire contenant les plugins
 Path:=ExtractFilePath(Application.ExeName);
// Recherche toutes les DLL
 if FindFirst(Path+'*.DLL',faAnyFile,SearchRec)=0 then begin
  repeat
   ListBox1.Items.Add(SearchRec.Name);
   try
   // On essaye d'en faire un Plugin
    PlugIn:=TPlugIn.Create(SearchRec.Name);
    Menu:=NewItem(PlugIn.Caption,PlugIn.ShortCut,False, True, PlugInClick, 0, '');
```

```
    // faire le lien entre le TMenuItem et le TPlugIn
     Menu.Tag:=PlugList.Add(Plugin);
     Menu.Bitmap.Handle:=PlugIn.Bitmap;
     PlugInsMenu.Add(Menu);
    except
     on e:Exception do ListBox1.Items.Add(e.Message);
    end;
   until FindNext(SearchRec)<>0;
 end;
 FindClose(SearchRec);
end;

procedure TForm1.FormDestroy(Sender: TObject);
var
 i:integer;
begin
// libération de tous les plugins
 for i:=0 to PlugList.Count-1 do TPlugin(PlugList[i]).Free;
 PlugList.Free;
end;
```

Pour chaque fichier DLL qui se trouve dans le répertoire de l'application, nous créons une instance de la classe `TPlugin`. Nous avons vu qu'en cas d'erreur, celle-ci générait une exception qui détruit automatiquement la classe `plugin` ; il nous suffit donc d'englober la création du menu `plugin` dans une section `try except`.

Un exemple de plug-in

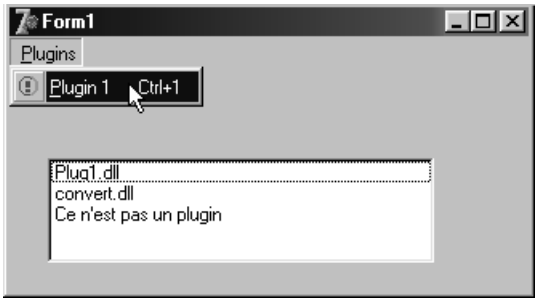
Nous avons défini une application hôte de telle façon que des DLL plug-in puissent ajouter des options dans son menu. Comme la fonction que le plug-in doit exporter a également été définie, il ne nous reste plus qu'à suivre ces recommandations pour écrire un exemple de plug-in :

```
library Plug1;

uses
  Windows,Menus,Classes,Dialogs,PlugIns;

{$r bitmap.res}

procedure PluginClick; stdcall;
 begin
  ShowMessage('OnClick !');
 end;

procedure InitPlugin(Var Options:TPluginOptions); stdcall;
 begin
  Options.Caption :='Plugin 1';
  Options.OnClick :=@PluginClick;
  Options.ShortCut:=TextToShortCut('Ctrl+1');
  Options.Bitmap  :=LoadBitmap(hInstance,'PLUGIN');
 end;
```

```
exports
 InitPlugin;

begin
end.
```

Ce plug-in très simple se contente de renseigner la structure `Options` avec notamment l'adresse de la fonction `PluginClick` qui affiche le message `OnClick` ! quand on clique sur le plug-in (voir figure 19-5).

Figure 19-5
Le menu plug-in



Utilisation des interfaces

L'exemple que nous venons de voir permet de mettre assez simplement en place des fonctions plug-in ; mais il est assez frustrant, avec un outil évolué comme Delphi, de se limiter à l'utilisation de fonctions et de `record`. Nous allons voir maintenant qu'il est tout à fait possible de manipuler de façon quasi transparente une instance objet placée dans une DLL.

> Les explications données ici le sont dans un esprit d'ouverture à d'autres langages de programmation ; le chapitre 22, « La gestion des packages », présente, avec les paquets, une solution spécifique à Delphi qui est beaucoup plus simple à mettre en œuvre.

Présentation des interfaces

Le modèle objet de Delphi est différent de celui de C++. De ce fait, il n'est pas plus possible d'importer des DLL objet C++ sous Delphi que le contraire.

> L'API QT sur laquelle est basée la CLX est conçue sur le modèle de DLL objet C++. Afin de pouvoir l'exploiter sous Delphi, Borland a eu recours à un *wrapper* de classes. La DLL `QTINTF.DLL` offre en effet une série de fonctions standards qui manipulent *via* un handle des instances de classes C++.

Un standard, indépendant du langage, permet d'interconnecter des structures objets différentes ; il s'agit des interfaces. Celles-ci remplissent la même fonction que la structure `TPluginOptions` que nous avons utilisée précédemment, mais elles sont normalisées et objet.

Nous allons reprendre l'exemple des plug-ins en nous appuyant cette fois sur une interface `IPlugin`.

Déclaration d'une interface

La déclaration de l'interface `IPlugin` est tout à fait similaire à celle de la structure `TPluginOptions` ; mais les interfaces étant exclusivement constituées de fonctions, toutes les références aux champs de la structure sont remplacées par des fonctions.

```
Type
 IPlugin=interface
  procedure DoClick; stdcall;
  function Caption:pchar; stdcall;
  function ShortCut:TShortCut; stdcall;
  function Bitmap:THandle; stdcall;
 end;
```

Tout comme la partie interface d'une unité, les interfaces ne contiennent aucun code ; elles se contentent d'indiquer la liste des méthodes qui les constituent.

L'interface est utilisée lors de la déclaration d'une classe objet ; en indiquant que celui-ci supporte l'interface, le compilateur prend soin de vérifier que toutes les méthodes de l'interface sont bien effectivement implémentées dans la classe. C'est là encore un processus assez similaire à ce que l'on trouve dans les unités ; toutes les fonctions déclarées dans la partie interface doivent être implémentées dans la partie implémentation.

Objet et interface

La déclaration de la classe `TIPlugin` qui sera utilisée dans la DLL est un exemple d'implémentation de l'interface `IPlugin`.

```
Type
 TIPlugin=class(TInterfacedObject,IPlugin)
 private
  fCaption :string;
  fShortCut:TShortCut;
  fBitmap  :THandle;
 public
  Constructor Create(ACaption:string; AShortCut:TShortCut; ABitmap:THandle);
// interface IPlugin
  procedure DoClick; stdcall;
  function Caption:pchar; stdcall;
  function ShortCut:TShortCut; stdcall;
  function Bitmap:THandle; stdcall;
 end;
```

Le mot-clé `class` précise que la classe `TIPlugin` hérite de la classe `TInterfacedObject` et qu'elle supporte l'interface `IPlugin`. On retrouve en effet toutes les méthodes de l'interface dans cette classe.

> Toutes les interfaces héritent par défaut de l'interface `IInterface` qui déclare les trois fonctions de base des interfaces : `QueryInterface`, `AddRef` et `Release`. Delphi propose la classe `TInterfacedObject` qui implémente ces trois fonctions ; c'est donc cet ancêtre qu'il faut utiliser si vous ne souhaitez pas gérer vous-même cet aspect des interfaces.

En faisant abstraction de la notion d'interface, l'implémentation de cet objet ne pose pas de problèmes particuliers, comme le montre le source ci-après.

```
Constructor TIPlugin.Create(ACaption:string; AShortCut:TShortCut; ABitmap:THandle);
 begin
  inherited Create;
  fCaption:=ACaption;
  fShortCut:=AShortCut;
  fBitmap:=ABitmap;
 end;

procedure TIPlugin.DoClick;
 begin
  ShowMessage('Clicked on '+fCaption);
 end;

function TIPlugin.Caption:pchar;
 begin
  Result:=PChar(fCaption);
 end;

function TIPlugin.ShortCut:TShortCut;
 begin
  Result:=fShortCut;
 end;

function TIPlugin.Bitmap:THandle;
 begin
  Result:=fBitmap;
 end;
```

C'est là que se trouve tout l'intérêt des interfaces ; notre classe objet interfacée n'est pas différente des classes traditionnelles ; il a simplement fallu implémenter les méthodes de l'interface en plus de celles dont nous avions réellement besoin.

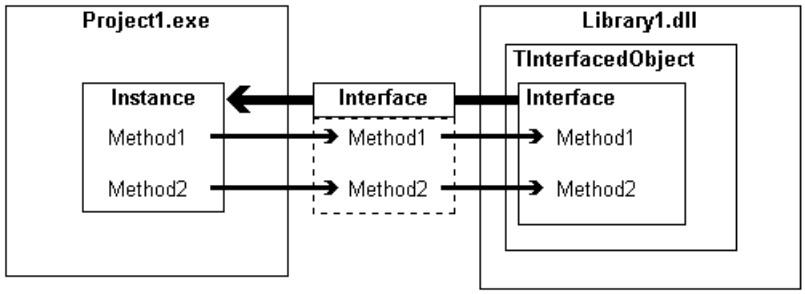
Maintenant que nous avons une classe objet supportant l'interface, nous allons pouvoir exporter la fonction qui permettra de créer des instances de cette classe en dehors de la DLL.

```
function CreatePlugin:IPlugin; stdcall;
 begin
  Result:=TIPlugin.Create(
   'IPlugin 1',
```

```
    TextToShortCut('Ctrl+1'),
    LoadBitmap(hInstance,'PLUGIN')
   );
 end;

exports
 CreatePlugin;

begin
end.
```

Remarquez que la fonction crée une instance de `TIPlugin` mais qu'elle retrouve une instance de `IPlugin`. En interne, nous travaillons avec notre classe objet ; de l'extérieur, elle est vue comme une interface (voir figure 19-6).

Figure 19-6
Instance et interface



Interfaces et instances

Nous avons vu comment la DLL plug-in était créée, et nous savons qu'elle exporte une fonction retournant une instance de `IPlugin`. Voyons à quoi ressemble maintenant la classe `TPlugin` qui s'occupe de charger la DLL et d'en extraire le plug-in.

```
type
 TCreatePlugin=function:IPlugin; stdcall;

 TPlugin=class
 private
  fHandle :THandle;
  fCreate :TCreatePlugin;
  fPlugin :IPlugin;
 public
  Constructor Create(FileName:string);
  Destructor Destroy; override;
  property Instance:IPlugin read fPlugin;
 end;
```

Nous faisons le choix de publier directement l'instance du plug-in ; il ne reste donc qu'à gérer la création et la suppression du plug-in.

```
Constructor TPlugin.Create(FileName:string);
 begin
 // Chargement de la DLL
  fHandle:=LoadLibrary(PChar(FileName));
  if fHandle=0 then raise Exception.Create('Impossible de charger la DLL');
 // Recherche de la fonction d'init
  fCreate:=GetProcAddress(fHandle,'CreatePlugin');
  if @fCreate=nil then raise Exception.Create('Ce n''est pas un i-plugin');
 // Création du plugin
  fPlugin:=fCreate;
 end;

Destructor TPlugin.Destroy;
 begin
 // Libérer l'interface
  fPlugin:=nil;
 // Libération de la DLL
  if fHandle<>0 then FreeLibrary(fHandle);
  inherited;
 end;
```

On procède tout simplement à la libération de l'interface en affectant le pointeur `nul` à la variable `fPlugin`. Cette approche est totalement différente de ce qui se passe avec les classes traditionnelles qui dissocient complètement la variable de l'instance.

Vous pouvez vérifier que l'objet est bel et bien libéré en surchargeant le destructeur de la classe `TIPlugin`.

```
Destructor TIPlugin.Destroy;
 begin
  ShowMessage('bye !');
  inherited;
 end;
```

Cette particularité des interfaces est liée aux méthodes de la classe `IInterface`. À chaque fois que l'on fait référence à une interface, la méthode `AddRef` est invoquée ; quand la variable est forcée à `nul`, c'est la méthode `Release` qui est invoquée. L'interface sait donc exactement quand elle n'est plus référencée nulle part ; et, dans ce cas, `TInterfaceObject` se détruit automatiquement.

> Vous trouverez parfois des sources Delphi qui déclarent des classes comprenant uniquement des méthodes virtuelles et abstraites – avec les mots-clés `virtual` et `abstract`. Il s'agit de l'ancienne méthode de gestion des interfaces ; avec le mot-clé `Interface`, il n'est plus nécessaire de préciser l'aspect virtuel des méthodes.

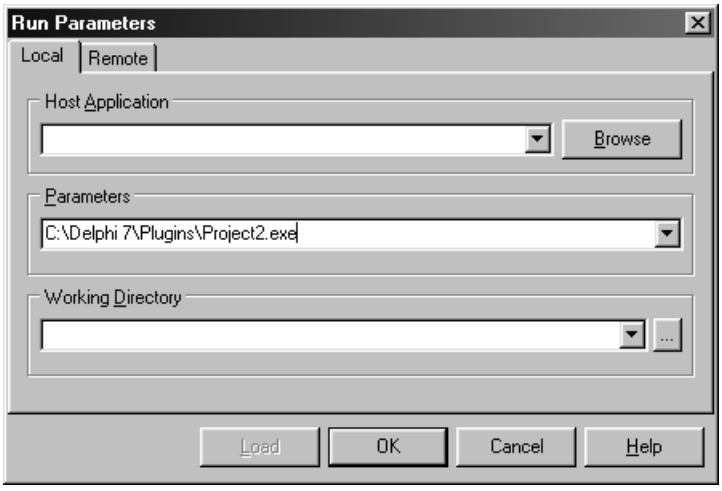
Trucs et astuces

Déboguer une DLL

Delphi est doté d'un puissant débogueur qui permet de suivre pas à pas l'exécution des programmes que l'on réalise. Mais malgré tout son savoir-faire, il ne peut pas deviner comment fonctionnent les DLL que vous réalisez. En effet, par définition, les fonctions d'une DLL ne s'exécutent pas selon un ordre établi, mais uniquement à la demande d'un programme hôte.

Eh bien ! justement… vous pouvez déboguer vos DLL pour peu que vous fournissiez à Delphi un programme hôte qui se chargera d'orchestrer les appels aux fonctions de la DLL. Le menu Exécution|Paramètres… vous permet d'indiquer le nom de l'application que Delphi doit invoquer pour faire appel à la DLL en cours (voir figure 19-7).

Figure 19-7
Déboguer une DLL



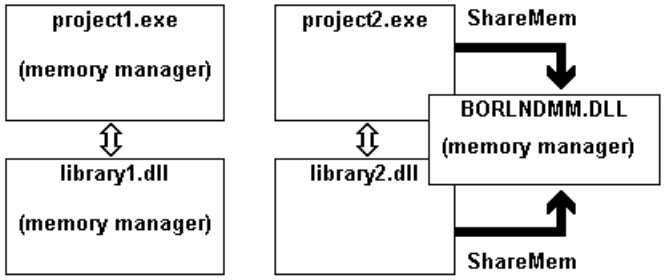
Grâce à cet artifice, vous pouvez placer vos points d'arrêts où bon vous semble dans le code de la DLL, et lancer l'application ; Delphi suspendra l'exécution du programme dès qu'il rencontrera un point d'arrêt.

Les chaînes de caractères

Depuis la version 2, Delphi utilise un mécanisme d'allocation dynamique pour les chaînes de caractères de type `string`. L'ancien type de chaînes statiques hérité de Turbo Pascal est renommé `shortstring`. Il faut donc prendre des précautions si l'on désire utiliser ce type en paramètre d'une fonction exportée. En effet, il est impératif que l'exécutable et la DLL gèrent chacun de leur côté leurs allocations mémoire ; si le programme peut lire une zone mémoire allouée par la DLL, il lui est interdit de la désallouer.

Pour résoudre ce problème et permettre d'utiliser les `strings` en toute tranquillité, Borland a mis au point la DLL `BORLNDMM.DLL` (que l'on retrouve sous le nom `DELPHIMM.DLL` dans les précédentes versions) qui permet à un programme et une DLL Delphi d'utiliser le même gestionnaire de mémoire (voir figure 19-8).

Figure 19-8
Gestionnaire de mémoire partagé



Pour activer ce mécanisme, il suffit d'ajouter l'unité `ShareMem` en tête de liste de la clause `Uses` du programme et de la DLL ; et cela ne peut fonctionner que si les deux l'utilisent.

Vous l'aurez compris, cette solution rend la DLL incompatible avec d'autres langages de programmation ; si vous ne pouvez accepter cette contrainte, vous devrez avoir recours aux traditionnels `PChar`.

> De façon générale, toute zone mémoire que le programme ou la DLL sont susceptibles d'allouer ou modifier doit faire l'objet d'une attention toute particulière. Soit la zone mémoire est gérée entièrement d'un côté, l'autre se contentant de la lire ou d'y écrire, soit il faut utiliser `ShareMem` des deux côtés pour partager le même gestionnaire de mémoire qui pourra gérer les allocations de façon totalement transparente. Les tableaux dynamiques et les appels à `GetMem` et `FreeMem` sont concernés au même titre que les `strings`.

Les allocations dynamiques

Si vous désirez faire en sorte que vos DLL soient compatibles avec les autres langages de programmation, vous devez proscrire l'utilisation de `ShareMem`. Voici quelques méthodes que vous pouvez employer pour échanger des données dynamiques entre un programme et une DLL.

L'allocation par le programme

L'API Windows utilise souvent une technique assez simple pour renvoyer une information de taille variable. Prenons pour exemple une fonction qui retourne un texte dans un `PChar`.

```
function GetText(P:PChar; var Size:integer):boolean;
 begin
 // déterminer si la copie peut avoir lieu
  Result:=Size>Length(Text);
 // Renvoyer la taille nécessaire
  Size:=Length(Text)+1;
 // si possible, renvoyer le texte
  if Result then StrPCopy(P,Text);
 end;
```

La fonction attend un pointeur sur un buffer alloué par l'application et un paramètre indiquant sa taille. Elle retourne la taille réellement nécessaire ainsi que le texte si le buffer est de taille suffisante. Voici un exemple d'appel à cette fonction.

```
function GetString(var s:string):boolean;
 var
  i:integer;
 begin
  i:=50;
  SetLength(s,i);
  if GetText(@s[1],i)=False then begin
   SetLength(s,i);
   if not GetText(@s[1],i) then i:=1;
  end;
  SetLength(s,i-1);
 end;
```

Voici une variante qui renvoie toujours la taille nécessaire, et le texte si le pointeur est non nul.

```
function GetText(p:pchar):integer;
 begin
  Result:=Length(Text)+1;
  if p<>nil then StrCopy(p,Text);
 end;
```

Et un exemple d'appel :

```
function GetString:string;
 var
  i:integer;
 begin
  i:=GetText(nil);
  SetLength(s,i+1);
  GetText(@s[1]);
 end;
```

L'allocation par la DLL

Les fonctions précédentes obligent le programme à invoquer deux fois la fonction ; une première fois pour obtenir la taille de la chaîne, et une seconde fois pour obtenir la chaîne

elle-même. On peut également laisser la DLL gérer l'allocation mémoire, mais dans ce cas il faut prévoir que ce sera également à elle de gérer la désallocation.

```
function GetText:pchar;
 begin
  GetMem(Result,Length(Text)+1);
  StrPCopy(Result,Text);
 end;

function FreeText(p:pchar);
 begin
  FreeMem(p);
 end;
```

Exemple d'utilisation :

```
function GetString:string;
 var
  p:pchar;
 begin
  p:=GetText;
  Result:=p;
  FreeText(p);
 end;
```

Les fonctions callback

Une dernière méthode permet de gérer dynamiquement des listes d'éléments : il s'agit de l'utilisation des fonctions *callback*, c'est-à-dire des fonctions appelées en retour. Le principe en est simple et efficace : il suffit de passer à la fonction l'adresse d'une procédure à invoquer pour chaque information de la liste. Voici par exemple l'implémentation d'une fonction qui retourne les chaînes d'une liste :

```
type
 TLineProc=procedure(UserData:pointer; Line:PChar); stdcall;

procedure GetLines(LineProc:TLineProc ; UserData:pointer);
 var
  i:integer;
 begin
  for i :=0 to Lines.Count-1 do LineProc(UserData, Lines[i]);
 end;
```

Le paramètre `UserData` est générique ; il permet simplement à l'application qui utilise la fonction d'indiquer le paramètre de son choix à retourner à chaque appel de la fonction. On comprend tout l'intérêt de ce paramètre dans les deux exemples d'implémentation suivants. Réalisons tout d'abord une DLL avec notre fonction exploitant un callback :

1. Créez un projet par l'expert DLL dans le menu Fichier|Nouveau|Autre.
2. Déclarez et exportez la fonction `GetLines` présentée ci-après.
3. Enregistrez le projet sous le nom `MaDLL.DPR` et compilez-le.

```
Library madll;

type
 TLineProc=procedure(userData:pointer; Line:PChar); stdcall;

procedure GetLines(LineProc:TLineProc; UserData:pointer); stdcall;
 begin
  LineProc(UserData,'ligne 1');
  LineProc(UserData,'ligne 2');
  LineProc(UserData,'ligne 3');
 end;

exports
 GetLines;

begin
end.
```

Maintenant, dans un nouveau projet, faisons appel à cette fonction :

1. Créez un nouveau projet.
2. Placez deux boîtes de liste sur la fiche : `ListBox1` et `ListBox2`.
3. Déclarez la fonction `GetLines` en généralisant le paramètre `LineProc` pour éviter les soucis de transtypage.
4. Ajoutez le reste du code qui initialise, dans l'événement `OnCreate` de la fiche, le contenu des deux listes.

```
// utilisation du paramètre UserData comme pointeur de classe
procedure LineProc1(Lines:TStrings; Line:PChar); stdcall;
 begin
  Lines.Add(Line);
 end;

// utilisation du paramètre UserData comme pointeur d'instance (SELF)
procedure TForm1.LineProc2(Line:PChar);
 begin
  ListBox2.Items.Add(Line);
 end;

// appel aux deux fonctions callback
procedure TForm1.FormCreate(Sender: TObject);
begin
 GetLines(@LineProc1,ListBox1.Items);
 GetLines(@TForm1.LineProc2,Self);
end;
```

La procédure `LineProc1` est globale ; elle exploite alors le paramètre `UserData` en tant que liste de chaînes, et en effet l'appel se fait avec le paramètre `ListBox1.Items`.

La méthode `LineProc2` (qu'il faut déclarer également dans le type `TForm1`) ne déclare pas le paramètre `UserData`, qui est utilisé par le paramètre implicite `Self`. Notez bien que, pour recourir à cette astuce, il est impératif que le paramètre soit déclaré en premier dans la liste de paramètres de la fonction `GetLines`. La déclaration de la convention d'appel est également primordiale ; n'oubliez pas qu'en Pascal l'ordre des paramètres est inversé par

rapport à l'appel `stdcall`. Un programme qui essaierait d'exécuter le code d'une fonction callback en partant de l'adresse de la classe `TForm1` essuierait un « plantage » assuré.

```
type
  TForm1 = class(TForm)
    ListBox1: TListBox;
    ListBox2: TListBox;
    procedure FormCreate(Sender: TObject);
  private
    { Déclarations privées }
    procedure LineProc2(Line:PChar); stdcall;
  public
    { Déclarations publiques }
  end;
```

Les DLL de ressources

Les DLL sont parfois utilisées pour regrouper en un seul fichier différentes ressources. Cela permet de modifier très aisément ces éléments sans devoir toucher au source du programme. Il est également possible de prévoir une DLL par langue supportée par l'application. Nous allons voir combien il est simple de créer et d'utiliser de telles DLL.

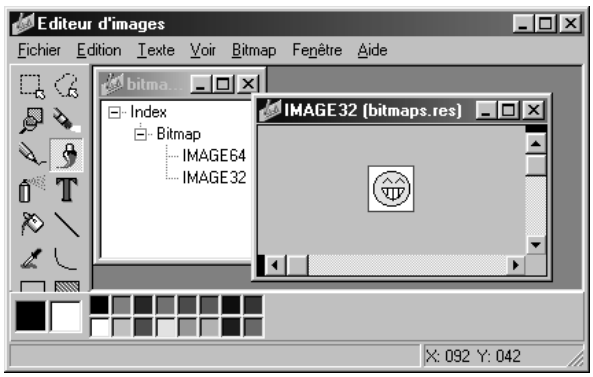
Pour créer un exemple de DLL, il suffit de peu de choses ; le source de `IMAGES.DLL` semble presque incomplet :

```
library images;

{$R bitmaps.res}

begin
end.
```

Comme vous pouvez le constater, il n'exporte aucune fonction et ne possède aucun code d'initialisation ; en fait, son seul intérêt est d'être lié à la ressource `bitmaps.res`.

L'éditeur d'images livré avec Delphi permet de créer des fichiers ressources ; lancez-le par le menu Outils et créez un fichier `bitmaps.res` dans lequel vous placerez au moins un bitmap nommé `IMAGE32` (voir figure 19-9).

Figure 19-9
L'éditeur d'images



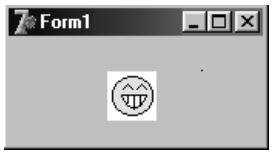
Compilez le projet pour obtenir la DLL, puis suivez les étapes suivantes pour tester son utilisation :

1. Créez un nouveau projet par le menu Fichier|Nouveau|Application.
2. Enregistrez-le dans le même répertoire que le projet `Images.dpr` ci-avant.
3. Placez un composant `TImage` sur la fiche.
4. Saisissez le code ci-après correspondant à l'événement `OnCreate` de la fiche.

```
procedure TForm1.FormCreate(Sender: TObject);
var
 dll:THandle;
begin
 dll:=LoadLibrary('IMAGES.DLL');
 if dll=0 then
  ShowMessage('DLL non trouvée')
 else begin
  Image1.Picture.Bitmap.LoadFromResourceName(dll,'IMAGE32');
  FreeLibrary(dll);
 end;
end;
```

Lors de l'exécution du programme, le composant `TImage` doit afficher le bitmap défini dans l'éditeur d'images (voir figure 19-10). Si vous obtenez le message `DLL non trouvée`, c'est que vous avez probablement placé le programme et la DLL dans des répertoires différents.

Figure 19-10
Utilisation d'une ressource externe



Rappel

Quand Windows doit charger une DLL, il la recherche, dans l'ordre, dans les répertoires suivants :

1. le répertoire de l'application,
2. le répertoire courant,
3. le répertoire `SYSTEM` (ou `SYSTEM32` selon la version),
4. le répertoire `WINDOWS` (ou `WINNT` selon la version),
5. les répertoires présents dans la variable `PATH`.

Conclusion

Windows ne serait pas ce qu'il est sans la présence des DLL. Depuis les toutes premières versions de ce système d'exploitation, elles ont permis la mise en commun du code des applications. Très largement supportées par les différents langages de programmation, elles restent une solution simple et efficace pour étendre les possibilités d'un produit *via* des fonctions plug-in. Mais sachez les utiliser à bon escient. La multiplication des DLL dans un produit permet peut-être de réduire la taille des mises à jour en ciblant les fichiers à modifier, mais elle fait de la gestion des mises à jour du produit un vrai casse-tête. Il est parfois préférable d'avoir un exécutable un peu plus gros plutôt que d'entrer dans la problématique des conflits de versions que peuvent occasionner les DLL.

En partie VI de cet ouvrage, le chapitre 22, intitulé « La gestion des packages », présente en détail l'utilisation des paquets d'exécution qui sont, dans l'environnement Delphi, une solution bien plus efficace au principe de découpage du programme. Le chapitre suivant fournit quant à lui des explications sur le fonctionnement des ActiveX qui sont une autre forme de plug-ins. Prenez donc le temps d'étudier toutes les possibilités que vous offre Delphi pour déterminer la technique la mieux adaptée à votre situation.