


# Construire une procédure pointant sur une méthode

par Sébastien Doeraene ([sjrd.developpez.com](http://sjrd.developpez.com))

Date de publication : 18/10/2007

Dernière mise à jour : 28/10/2007

Beaucoup de procédures des API Windows, ou même de bibliothèques tierces, acceptent un paramètre de *call-back* dont le type est un pointeur de *procédure*. Cependant, on aimerait lui transmettre un pointeur de méthode, ne fut-ce que pour avoir des informations de contexte supplémentaires au sein du *call-back*.

Comme le tutoriel  **Combiner des procédures et des méthodes** vous l'explique, il n'est possible de transmettre un pointeur de méthode à la place d'un pointeur de procédure que si la procédure en question est prévue pour : qu'elle possède un paramètre "inutile" en première place. Or dans le cas que nous examinons maintenant, nous n'avons pas la main sur la définition de la procédure de *call-back*, puisqu'elle appartient soit à Windows, soit à une bibliothèque tierce dont nous n'avons peut-être même pas le code source.

Toute la problématique réside donc en ceci : obtenir un pointeur sur une *procédure* qui, lorsqu'on l'appelle avec des arguments donnés, appelle elle-même une *méthode* dont la signature est identique, mais qui évidemment demande un paramètre implicite **Self** supplémentaire.

Vous êtes invités à **donner vos commentaires sur le blog**.

Un grand merci à **Laurent Dardenne** pour ses nombreuses remarques toujours aussi constructives, ainsi qu'à **Pierre Rodriguez** pour sa correction orthographique attentive.



Merci également à **CapJack** pour sa remarque pertinente quant au besoin d'un exemple d'utilisation des **MakeProcOfXXXMethod**.

- I - Public visé et pré-requis
- II - La problématique
- III - L'idée de base
- IV - Que doit faire exactement CallbackProc ?
- V - Les conventions d'appel : le passage des paramètres
- VI - Comment utiliser les routines que nous allons développer ?
- VII - Convention d'appel stdcall
  - VII-A - Le code assembleur
  - VII-B - Code exécutable x86 correspondant
  - VII-C - Particularités des instructions JMP et CALL
  - VII-D - Générer dynamiquement le code
- VIII - Convention d'appel pascal
- IX - Convention d'appel register
  - IX-A - Les difficultés
  - IX-B - Cas de base : il ne faut pas stocker ECX
  - IX-C - Ranger ECX au sommet de la pile
  - IX-D - Ranger ECX plus haut dans la pile
  - IX-E - Une première fonction MakeProcOfRegisterMethod
  - IX-F - Quand MoveStackCount devient trop grand
  - IX-G - Mettre ensemble les deux techniques
- X - Convention d'appel cdecl
  - X-A - Les difficultés
  - X-B - Le code assembleur de base
  - X-C - Les routines de stockage et de récupération de l'adresse de retour
  - X-D - Que se passe-t-il en cas d'exception ?
  - X-E - Identifier les parties corrompues de la pile
  - X-F - Le code assembleur final
  - X-G - La fonction MakeProcOfCDeclMethod
  - X-H - La dernière touche contre les fuites mémoire
- XI - Conclusion
- XII - Liens
- XIII - Pour aller plus loin : se servir de ObjAuto
  - XIII-A - L'unité ObjAuto
  - XIII-B - Naviguer dans les RTTI étendues
  - XIII-C - Une unique routine avec CallingConvention
  - XIII-D - Compter le nombre de registres utilisés par la méthode
  - XIII-E - Déterminer le paramètre stocké par ECX dans la procédure
  - XIII-F - Le final
  - XIII-G - Code complet de GetAutoRegisterInfo

## I - Public visé et pré-requis


Public visé : expert

### Pré-requis

-  **Combiner des procédures et des méthodes**, de **Laurent Dardenne**
- Assembleur x86, dialecte BASM (Borland ASM) (voir éventuellement le tutoriel  **Utilisation de l'assembleur en ligne avec Delphi** de **Nono40**)

## II - La problématique

Beaucoup de procédures des API Windows, ou même de bibliothèques tierces, acceptent un paramètre de *call-back* dont le type est un pointeur de *procédure*. Cependant, on aimerait lui transmettre un pointeur de méthode, ne fut-ce que pour avoir des informations de contexte supplémentaires au sein du *call-back*.

Comme le tutoriel  **Combiner des procédures et des méthodes** vous l'explique, il n'est possible de transmettre un pointeur de méthode à la place d'un pointeur de procédure que si la procédure en question est prévue pour : qu'elle possède un paramètre "inutile" en première place. Or dans le cas que nous examinons maintenant, nous n'avons pas la main sur la définition de la procédure de *call-back*, puisqu'elle appartient soit à Windows, soit à une bibliothèque tierce dont nous n'avons peut-être même pas le code source.

Toute la problématique réside donc en ceci : obtenir un pointeur sur une *procédure* qui, lorsqu'on l'appelle avec des arguments donnés, appelle elle-même une *méthode* dont la signature est identique, mais qui évidemment demande un paramètre implicite **Self** supplémentaire.

### III - L'idée de base

Il faut donc que, d'une manière ou d'une autre, nous puissions produire une procédure qui appelle une méthode. Si nous savions à l'avance sur quel objet (global) nous devrions appeler la méthode, on pourrait écrire ceci :

```
procedure CallBackProc(Param1: Integer; const Param2: string);  
begin  
  GlobalObj.CallBackMethod(Param1, Param2);  
end;
```

Seulement voilà ! Nous ne pouvons pas connaître l'objet sur lequel appeler la méthode. Et à la rigueur, on pourrait même ne pas savoir la méthode à appeler !

Mais cette écriture est intéressante, car elle est bien ce que nous voulons. Si ce n'est que nous ne connaissons **GlobalObj** et **@CallBackMethod** qu'à l'exécution.

La solution ? Construire la routine **CallBackProc** pendant l'exécution du programme ! Et cela en allouant sur le tas une zone de données que l'on remplira avec du code machine x86, exécutable par le processeur.



## IV - Que doit faire exactement CallbackProc ?

Avant de se lancer dans la tâche ardue de construire une routine pendant l'exécution, cernons bien ce qu'a besoin de faire cette routine, que nous avons appelée **CallbackProc**.

En réalité, elle ne doit pas faire grand chose. Elle doit seulement *ajouter un paramètre (CallbackObj)* aux paramètres d'appel existant (sans toucher à ceux-ci), en première position ; puis *rediriger vers la méthode à appeler (CallbackMethod)*.

La redirection se fait avec un simple JMP FAR assembleur. L'ajout du paramètre est plus délicat, et nous devons pour pouvoir le faire bien comprendre comment sont passés les paramètres.

## V - Les conventions d'appel : le passage des paramètres

C'est un adage bien connu : l'informaticien est encore plus paresseux que le mathématicien. Je ne vais pas ici réinventer la roue, ni plagier. Je me contenterai donc de vous rediriger vers les sections  **Conventions d'appel** et  **Paramètres et résultat de la fonction** d'un tutoriel de **Nono40**.

Puisque **Self** est un type objet ou classe (donc pointeur) et est le premier paramètre, il est passé soit dans EAX (avec **register**), soit en premier paramètre sur la pile (avec les autres conventions).

## VI - Comment utiliser les routines que nous allons développer ?

Pour bien comprendre à quoi peuvent servir les routines magiques que nous allons développer pas à pas dans la suite de ce tutoriel, nous allons d'abord montrer comment les utiliser.

Nous supposons l'existence d'une routine **MakeProcOfStdCallMethod**, définie comme suit (il s'agit d'une des routines qui vont être écrites) :

```
function MakeProcOfStdCallMethod(const Method: TMethod): Pointer;
```

Cette méthode prend en paramètre une méthode dont la convention d'appel est **stdcall**, et renvoie une procédure qui redirige sur celle-ci. Le pointeur renvoyé doit être libéré avec **FreeMem**.

Voici encore une petite routine pratique, qui permet de construire un **record TMethod** comme **Point** construit un **record TPoint**.


```
function MakeMethod(Code: Pointer; Data: Pointer = nil): TMethod;  
begin  
  Result.Code := Code;  
  Result.Data := Data;  
end;
```

Pour l'exemple, nous allons énumérer les handle des fenêtres du processus courant. Je choisis cet exemple car il s'agit d'une question courante, qui a d'ailleurs réponse dans la FAQ Delphi : [FAQ Comment récupérer les handles des fenêtres d'un processus ?](#)

La FAQ montre comment exploiter le paramètre **LParam** du call-back pour avoir des informations de contexte. Je vais montrer ici comment faire sans - ce qui serait obligatoire si le call-back ne proposait pas ce paramètre "à contenu indéterminé".

Au lieu d'avoir un call-back qui est une routine, nous faisons une méthode de la fenêtre principale :

```
type  
  TFormMain = class(TForm)  
    ...  
  private  
    function EnumWndCallBack(Handle: HWND; LParam: LPARAM): Boolean; stdcall;  
    ...  
  end;  
  
function TFormMain.EnumWndCallBack(Handle: HWND; LParam: LPARAM): Boolean;  
var  
  ProcessID: DWord;  
begin  
  GetWindowThreadProcessId(Handle, ProcessID);  
  if ProcessID = GetCurrentProcessID then  
    ListBoxHandleList.Items.Add(IntToStr(Handle));  
  Result := True;  
end;
```

 *Quel est l'avantage d'une méthode par rapport à une routine ? L'avantage, c'est que le paramètre **Self** contient des informations de contexte. Ici il s'agit de l'instance de la fiche qui est concernée par le call-back. C'est beaucoup plus propre que d'utiliser une variable globale ; et si on travaillait avec des threads, ce serait même impossible à mettre en place autrement.*

Nous voulons donc appeler **EnumWindows** comme suit :

```
procedure TFormMain.ButtonEnumWindowsClick(Sender: TObject);
begin
  ListBoxHandleList.Clear;
  EnumWindows(@TFormMain.EnumWndCallBack, 0);
end;
```

Cependant, cela n'est évidemment pas possible, car **EnumWndCallBack** est une méthode, pas une routine. Nous allons donc utiliser **MakeProcOfStdCallMethod** pour avoir une routine :

```
procedure TFormMain.ButtonEnumWindowsClick(Sender: TObject);
type
  TEnumWndProc = function(Handle: HWND; LParam: Integer): Boolean stdcall;
var
  EnumWndProc: TEnumWndProc;
begin
  ListBoxHandleList.Clear;
  EnumWndProc := MakeProcOfStdCallMethod(MakeMethod(
    @TFormMain.EnumWndCallBack, Self));
  try
    EnumWindows(@EnumWndProc, 0);
  finally
    FreeMem(@EnumWndProc);
  end;
end;
```

Et voilà qui est fait. Simple non ? Dans le reste du tutoriel, nous allons montrer comment écrire **MakeProcOfStdCallMethod** et compagne (pour les autres conventions d'appel).

## VII - Convention d'appel stdcall

### VII-A - Le code assembleur

La convention **stdcall** n'est pas standard pour rien : elle est la plus simple, *surtout* pour ce que nous voulons faire ici. Nous allons donc commencer par celle-ci pour introduire tous les trucs et astuces que nous mettrons en oeuvre.

En effet, tous les paramètres, sans exception, sont passés sur la pile, et en plus en ordre inverse. Il nous suffit donc d'ajouter notre **CallbackObj** sur la pile : puisqu'ajouté en dernier sur la pile, il est le premier dans l'ordre des paramètres. Nous n'avons pas besoin de nous soucier des autres paramètres éventuels, car ils n'interféreront jamais.

D'autre part, c'est la méthode appelée qui supprime de la pile les paramètres (contrairement à **cdecl**). Or celle-ci est bien au courant qu'un paramètre supplémentaire existe, donc nous n'avons rien à faire (nous verrons plus loin que justement, avec **cdecl**, cela pose de nombreux soucis).



La seule astuce est qu'il faut insérer le paramètre **CallbackObj** devant l'adresse de retour, qui est ajoutée automatiquement par CALL et exploitée par RET. Il faut donc sauvegarder la valeur présente actuellement dans la pile et la remettre après avoir stocké **CallbackObj**. Puisque EAX est volatil, nous avons parfaitement le droit de l'utiliser pour stocker temporairement cette adresse de retour.

Le code assembleur de **CallbackProc** doit donc être celui-ci :

```
POP     EAX
PUSH   CallbackObj
PUSH   EAX
JMP    CallbackMethod
```

### VII-B - Code exécutable x86 correspondant

Tout ça c'est très bien, mais on ne connaît toujours pas **CallbackObj** ni **@CallbackMethod**. On ne les connaîtra qu'à l'exécution. Donc il faut générer ce code assembleur à l'exécution. Assembleur ? Que dis-je, non ! Il faut générer le code machine équivalent à ce code assembleur.

Un tutoriel très intéressant pour pouvoir générer du code x86 est celui sur le  **décodage du jeu d'instructions x86/x64**, écrit par **Neitsa**. Ce tutoriel recommande par ailleurs le débogueur  **OllyDbg**, qui a l'énorme avantage de vous permettre de coder des instructions assembleur "sur le vif" et de voir leur encodage x86.

Grâce à cet outil (ou un autre), on obtient facilement le code x86 correspondant à nos 4 instructions assembleur :


```
58          POP     EAX
68 xx xx xx xx  PUSH   CallbackObj
50          PUSH   EAX
E9 yy yy yy yy  JMP    CallbackMethod
```

### VII-C - Particularités des instructions JMP et CALL

La valeur "yy yy yy yy" dans le code précédent ne représentent pas directement l'adresse de la méthode **CallbackMethod**. En réalité, elle indique le *déplacement* à appliquer au registre d'instruction EIP. Et ceci après que l'instruction a été exécutée, donc lorsque EIP pointe sur l'instruction *suivante*.

Pour connaître l'argument d'une instruction JMP (ou CALL, qui partage son fonctionnement sur ce point), on utilise donc la formule que calcule la fonction **JmpArgument** suivante :

```
function JmpArgument(JmpAddress, JmpDest: Pointer): Integer; inline;
begin
  Result := Integer(JmpDest) - Integer(JmpAddress) - 5;
end;
```

 Cette fonction est un cas typique d'utilisation de la directive **inline**. Celle-ci suggère au compilateur d'optimiser chaque appel à cette fonction en remplaçant cet appel par le contenu de la fonction. Cela rend le code plus rapide, mais a le désavantage de l'agrandir en taille. De plus, ce n'est pas toujours possible. Consultez l'aide de Delphi pour plus d'informations sur que l'on appelle l'inlining.

**JmpArgument** est bien disposée pour l'inlining car elle opère une formule mathématique. Une unique instruction **Result := Expression**.


Les connaisseurs de C/C++ reconnaîtront l'adaptation des macros au Pascal.

Comme ce ne sera pas la seule instruction JMP/CALL que nous créerons, épargnons-nous encore un peu de travail avec ces quelques facilités :

```
type
  TJumpInstruction = packed record
    OpCode: Byte;      // OpCode
    Argument: Integer; // Destination
  end;

procedure MakeJump(var Instruction; Dest: Pointer);
begin
  TJumpInstruction(Instruction).OpCode := $E9; // OpCode de JMP
  TJumpInstruction(Instruction).Argument := JmpArgument(@Instruction, Dest);
end;

procedure MakeCall(var Instruction; Dest: Pointer);
begin
  TJumpInstruction(Instruction).OpCode := $E8; // OpCode de CALL
  TJumpInstruction(Instruction).Argument := JmpArgument(@Instruction, Dest);
end;
```

 J'ai fait ici le choix d'un paramètre *Instruction* non typé, et de le transtyper en **TJumpInstruction** à l'intérieur de la procédure. Ceci permet éventuellement d'appeler ces routines avec en paramètre du contenu de n'importe quel type. Ce ne sera pas utile ici mais j'ai eu l'occasion de m'en servir personnellement par ailleurs.

Accessoirement, c'est l'occasion de faire découvrir l'usage de ce type de paramètres.

Au point où nous en sommes (génération d'OpCodes x86), on peut se permettre un peu d'assembleur en ligne pour ces deux routines. Cela n'est utile que pour les versions de Delphi qui ne supportent pas la directive **inline**. En

effet, avec l'*inlining* de **JmpArgument**, le code produit par Delphi n'a jamais qu'un MOV de plus que ce que nous présentons ici. En revanche, sans le support de **inline**, le code est inutilement beaucoup plus long.

```

procedure MakeJump(var Instruction; Dest: Pointer);
asm
    { -> EAX Pointer to a TJumpInstruction record }
    { -> EDX Pointer to destination           }

    MOV     BYTE PTR [EAX], $E9
    SUB     EDX, EAX
    SUB     EDX, 5
    MOV     [EAX+1], EDX
end;

procedure MakeCall(var Instruction; Dest: Pointer);
asm
    { -> EAX Pointer to a TJumpInstruction record }
    { -> EDX Pointer to destination           }

    MOV     BYTE PTR [EAX], $E8
    SUB     EDX, EAX
    SUB     EDX, 5
    MOV     [EAX+1], EDX
end;
    
```

## VII-D - Générer dynamiquement le code

Nous avons donc tout en main pour créer la première des quatre routines **MakeProcOfXXXMethod**. Elle est vraiment simple, à présent :


```

function MakeProcOfStdCallMethod(const Method: TMethod): Pointer;
type
    PStdCallRedirector = ^TStdCallRedirector;
    TStdCallRedirector = packed record
        PopEAX: Byte;
        PushObj: Byte;
        ObjAddress: Pointer;
        PushEAX: Byte;
        Jump: TJumpInstruction;
    end;
begin
    GetMem(Result, SizeOf(TStdCallRedirector));
    with PStdCallRedirector(Result)^ do
        begin
            PopEAX := $58;
            PushObj := $68;
            ObjAddress := Method.Data;
            PushEAX := $50;
            MakeJump(Jump, Method.Code);
        end;
    end;
end;
    
```



*Vous ne trouvez pas **CallbackObj** et **CallbackMethod** ? Ils se trouvent dans les deux champs de **TMethod**, respectivement **Data** et **Code**.*

Pour utiliser cette fonction, il suffit de lui transmettre une méthode transtypée en **TMethod**, et l'on récupère un pointeur que l'on peut retrans typer en le type procédure du *call-back*.

 *N'oubliez pas de libérer le pointeur ainsi obtenu au moyen de **FreeMem** lorsque vous n'en avez plus besoin.*

## VIII - Convention d'appel pascal

La différence entre les conventions d'appel **stdcall** et **pascal** est subtile mais bien présente. En **stdcall**, les paramètres sont empilés dans l'ordre inverse de leur déclaration. Ce qui tombe bien, puisqu'ainsi nous avons pu ajouter très facilement le *premier* paramètre (**Self**) en *dernier*, après ce que l'appelant a déjà fait.

En **pascal** cependant, les paramètres sont empilés dans l'ordre réel de leur déclaration. Ce qui veut dire que la position du premier paramètre dans la pile est dépendante des autres paramètres.

Avant de se lancer dans un déplacement de mémoire de la pile - ce qui semble nécessaire puisqu'il faut insérer le paramètre **Self** en premier - il est *extrêmement* intéressant de découvrir ce fait : qu'en réalité, le paramètre **Self**, en convention **pascal**, est "déclaré" *en dernier* ! Et donc se retrouve à être empilé en dernier aussi. Exactement comme en **stdcall** !

Vous l'aurez compris, nous n'allons pas traîner plus longtemps sur **pascal**, car la routine **MakeProcOfPascalMethod** est identique à **MakeProcOfStdCallMethod**.


```
function MakeProcOfPascalMethod(const Method: TMethod): Pointer;  
begin  
    Result := MakeProcOfStdCallMethod(Method);  
end;
```

## IX - Convention d'appel register


### IX-A - Les difficultés

Cette fois-ci, plus question de s'en sortir aussi facilement. Et pourtant la convention **register** est la convention par défaut en Delphi. Il est donc crucial de pouvoir la traiter correctement.

Pour rappel, cette convention d'appel - qui est la convention **fastcall** de Pascal - utilise dans l'ordre les registres EAX, EDX et ECX pour transmettre les trois premiers paramètres qui peuvent se transmettre dans ces registres. S'il reste des paramètres, ceux-ci sont transmis sur la pile, dans l'ordre de leur déclaration (comme **pascal**).


 *Les chaînes courtes, variants, tableaux et **record** longs, bien que plus grands que 4 octets, sont transmis par adresse, et entrent donc bien dans les registres. Les types flottants, quels qu'ils soient, sont toujours transmis sur la pile, y compris le type **Single** qui est stocké sur 4 octets. Les types pointeur sur méthode sont stockés sur 8 octets et toujours transmis sur la pile, tandis que les types pointeur sur procédure peuvent passer par un registre, puisqu'ils sont stockés sur 4 octets.*

*Tous les paramètres **var** et **out**, de quel type qu'ils soient, sont toujours transmis par adresse, et peuvent donc être passés par registre. Les paramètres non typés partagent ce comportement.*

 *La valeur de retour d'une fonction fait parfois partie des paramètres également ! Lorsque le type de cette valeur de retour fait partie des types qui se transmettent toujours par adresse (chaînes courtes, variants, tableaux et **record** longs), c'est l'appelant qui alloue l'espace où stocker la valeur de retour. L'adresse de cet espace constitue alors un paramètre supplémentaire dans l'appel de la fonction (en dernière position).*

Le paramètre **Self**, qu'il soit de type objet ou meta-classe, est toujours un pointeur stocké sur 4 octets, et peut donc être passé par registre. Puisqu'il est le premier, il est *toujours* passé dans le registre EAX.

Ce qu'il faut, en revanche, c'est enregistrer l'ancienne valeur de EAX dans EDX, l'ancienne valeur de EDX dans ECX, et l'ancienne valeur de ECX sur la pile, *et au bon endroit*.

 *MAIS, car il y a un mais, il ne faut surtout pas ajouter quelque chose sur la pile si la méthode n'a pas de paramètre passé sur la pile. Sinon, la pile sera tout à fait corrompue après le retour de la méthode appelée.*

Comment savoir s'il faut ou non insérer ECX dans la pile, et où l'insérer ? Pour ne pas devoir tout avaler d'un coup, nous allons progressivement prendre en charge des cas de plus en plus complexes.

### IX-B - Cas de base : il ne faut pas stocker ECX

Pour commencer, nous allons faire la supposition honteuse qu'il ne faut jamais stocker ECX dans la pile. Quand cela arrive-t-il ? Quand il y a maximum 2 paramètres *de la procédure* qui sont passés par registre. Que ce soit parce qu'il n'y en a que 2, ou parce que les autres doivent obligatoirement passer sur la pile (flottants et pointeurs de méthode).

Dans ces cas, qu'a besoin de faire **CallbackProc** ? Simplement de ranger EDX dans ECX, EAX dans EDX, et finalement stocker dans EAX l'adresse de l'objet. Si EDX ou EAX n'étaient déjà pas utilisés avant, il n'y a aucune conséquence : ces trois registres sont volatils.

```
MOV    ECX, EDX
MOV    EDX, EAX
MOV    EAX, CallbackObj
JMP    CallbackMethod
```

La pile n'est alors pas altérée du tout, et tout est très facile.

## IX-C - Ranger ECX au sommet de la pile

Dans un certain nombre de cas, ECX doit être rangé au sommet de la pile, de la même façon qu'on y stockait l'adresse de l'objet pour la convention **stdcall**. Cette situation arrive lorsque le paramètre Delphi que porte ECX est le *dernier paramètre déclaré*. En effet, ce paramètre doit alors être empilé juste avant l'instruction CALL.

Dans ces cas-là, on utilise une technique similaire à celle utilisée avec **stdcall** : déplacer l'adresse de retour un élément plus loin dans la pile, et stocker ECX à l'endroit laissé vide.

Cependant, avec **stdcall**, nous pouvions utiliser le registre EAX comme endroit temporaire où récupérer l'adresse de retour. Ici, nous n'avons pas le droit de perdre le contenu de EAX, puisqu'il s'agit d'un paramètre. Nous nous servons donc d'une instruction bien pratique : XCHG. Cette instruction échange les informations contenues dans ses deux opérandes. Ainsi, XCHG ECX,[ESP] place la valeur de ECX dans [ESP] et celle de [ESP] dans ECX.

```
XCHG   ECX, [ESP]
PUSH   ECX
MOV    ECX, EDX
MOV    EDX, EAX
MOV    EAX, CallbackObj
JMP    CallbackMethod
```

Vous remarquerez que les quatre dernières instructions ici sont les mêmes que les quatre instructions du cas de base. Nous en profiterons plus tard.

## IX-D - Ranger ECX plus haut dans la pile

Lorsqu'ECX n'est pas le dernier paramètre déclaré, tous les paramètres qui le suivent sont empilés après lui. Il faut donc insérer ECX devant ceux-ci. Si l'on connaît le nombre de "cases" de pile à déplacer ainsi (la pile croît et décroît toujours par double mot sous Win32), on peut répéter plusieurs fois l'instructions XCHG avec un offset sur ESP.

Si par exemple il y a 2 cases à déplacer (deux entiers par exemple), on aura ceci :

```
XCHG   ECX, [ESP+8]
XCHG   ECX, [ESP+4]
XCHG   ECX, [ESP]
PUSH   ECX
MOV    ECX, EDX
MOV    EDX, EAX
MOV    EAX, CallbackObj
JMP    CallbackMethod
```

Encore une fois, cette suite d'instructions contient entièrement la suite de la section précédente : plus on complexifie, plus la "préparation" est longue.

La seule limitation qui reste est que le nombre de cases à déplacer ainsi n'atteigne pas 64 (256/4). Sinon on devra utiliser les versions longues de XCHG, et de toutes façons on aurait trop d'instructions XCHG ! Une pour chaque case !

## IX-E - Une première fonction MakeProcOfRegisterMethod

Nous allons déjà construire une première version de **MakeProcOfRegisterMethod** qui se base sur ce que nous avons déjà trouvé.

Elle a malheureusement besoin, outre le paramètre **Method**, de deux paramètres **UsedRegCount** et **MoveStackSize**, indiquant respectivement le nombre de registres utilisés dans la *procédure* (0 à 3) et le nombre de cases de pile à déplacer. Notez que **MoveStackSize** doit être 0 si **UsedRegCount** < 3.

Nous utilisons la remarque que les préparations s'ajoutent au début pour construire plus facilement la procédure. Combinée avec des jeux de calculs de tailles, on peut faire quelque chose de pas trop lent.

```
function MakeShortProcOfRegisterMethod(const Method: TMethod;
    UsedRegCount: Byte; MoveStackSize: Word): Pointer;

const
    MoveStackItem: LongWord = $00244C87; // 874C24 xx XCHG ECX,[ESP+xx]
    MoveRegisters: array[0..7] of Byte = (
        $87, $0C, $24, // XCHG ECX,[ESP]
        $51, // PUSH ECX
        $8B, $CA, // MOV ECX,EDX
        $8B, $D0 // MOV EDX,EAX
    );

type
    PRegisterRedirector = ^TRegisterRedirector;
    TRegisterRedirector = packed record
        MovEAXObj: Byte;
        ObjAddress: Pointer;
        Jump: TJumpInstruction;
    end;

var
    MoveStackSize, MoveRegSize: Integer;
    InstrPtr: Pointer;
    I: Cardinal;
begin
    Assert((MoveStackSize = 0) or (UsedRegCount >= 3));

    if UsedRegCount >= 3 then
        UsedRegCount := 4;
        MoveRegSize := 2*UsedRegCount;
        MoveStackSize := 4*MoveStackSize;

    GetMem(Result, MoveStackSize + MoveRegSize + SizeOf(TRegisterRedirector));
    InstrPtr := Result;

    for I := MoveStackSize downto 1 do
    begin
        // I shl 26 => I*4 in the most significant byte (kind of $ I*4 00 00 00)
        PLongWord(InstrPtr)^ := (I shl 26) or MoveStackItem;
        Inc(Integer(InstrPtr), 4);
    end;
end;
```

```

Move(MoveRegisters[SizeOf(MoveRegisters) - MoveRegSize],
    InstrPtr^, MoveRegSize);
Inc(Integer(InstrPtr), MoveRegSize);

with PRegisterRedirector(InstrPtr)^ do
begin
    MoveEAXObj := $B8;
    ObjAddress := Method.Data;
    MakeJump(Jump, Method.Code);
end;
end;
    
```

## IX-F - Quand MoveStackCount devient trop grand

Comme il a déjà été fait remarquer, la taille du code croît avec **MoveStackCount**. Et en plus, trop d'instructions XCHG diminuent les performances.

Lorsque **MoveStackCount** dépasse un certain seuil, nous allons employer une autre technique, qui s'appuie sur un REP MOVSD pour déplacer en masse les éléments de la pile.

Puisque ESI et EDI ne sont pas volatils, et qu'on ne doit pas perdre l'information enregistrée dans ECX non plus, on doit stocker temporairement le contenu de ces trois registres sur la pile - ce qui implique de prendre en compte qu'ils y sont aussi !

```

; First move the stack

PUSH    ESI
PUSH    EDI
PUSH    ECX

MOV     ESI,ESP
PUSH    ECX
MOV     EDI,ESP
MOV     ECX,MoveStackCount+4 ; 4 are ECX, EDI, ESI and ReturnAddress

REP     MOVSD

POP     ECX
POP     EDI
POP     ESI

; Now move registers and jump to the method

MOV     [ESP + 4*(MoveStackCount+1)],ECX ; 1 is ReturnAddress
MOV     ECX,EDX
MOV     EDX,EAX
MOV     EAX,CallBackObj
JMP     CallBackMethod
    
```

Ce code est en fait beaucoup plus facile à générer que le premier : il est de taille constante et seules quelques parties sont dépendantes des paramètres. On simplifie énormément en définissant une constante tableau de **Byte** qui contient le code immuable. Cela donne la routine suivante :

```

function MakeLongProcOfRegisterMethod(const Method: TMethod;
    MoveStackCount: Word): Pointer;

type
    
```

```

PRegisterRedirector = ^TRegisterRedirector;
TRegisterRedirector = packed record
    Reserved1: array[0..8] of Byte;
    MoveStackCount4: LongWord;
    Reserved2: array[13..20] of Byte;
    FourMoveStackCount1: LongWord;
    Reserved3: array[25..29] of Byte;
    ObjAddress: Pointer;
    Jump: TJumpInstruction;
end;

const
Code: array[0..SizeOf(TRegisterRedirector)-1] of Byte = (
    $56,           // PUSH   ESI
    $57,           // PUSH   EDI
    $51,           // PUSH   ECX

    $8B, $F4,      // MOV    ESI,ESP
    $51,           // PUSH   ECX
    $8B, $FC,      // MOV    EDI,ESP
    $B9, $FF, $FF, $FF, $FF, // MOV    ECX,MoveStackCount+4

    $F3, $A5,      // REP    MOVSD

    $59,           // POP    ECX
    $5F,           // POP    EDI
    $5E,           // POP    ESI

    $89, $8C, $24, $EE, $EE, $EE, $EE, // MOV    [ESP + 4*(MoveStackCount+1)],ECX
    $8B, $CA,      // MOV    ECX,EDX
    $8B, $D0,      // MOV    EDX,EAX
    $B8, $DD, $DD, $DD, $DD, // MOV    EAX,0
    $E9, $CC, $CC, $CC, $CC // JMP    MethodAddress
);

begin
    GetMem(Result, SizeOf(Code));
    Move(Code[0], Result^, SizeOf(Code));

    with PRegisterRedirector(Result)^ do
    begin
        MoveStackCount4 := MoveStackCount+4;
        FourMoveStackCount1 := 4 * (MoveStackCount+1);
        ObjAddress := Method.Data;
        MakeJump(Jump, Method.Code);
    end;
end;

```



Cette version n'accepte pas de paramètre **UsedRegCount** : on a en effet décidé de n'utiliser celle-ci que lorsque **MoveStackCount** dépasse un certain seuil. Et nous avons remarqué que **MoveStackCount** ne peut être différent de 0 que si **UsedRegCount** vaut 3.

## IX-G - Mettre ensemble les deux techniques

Il ne reste plus qu'à mettre ensemble les deux techniques. Et donc à déterminer un seuil de **MoveStackCount**. En se basant exclusivement sur la longueur du code généré - ce qui est une approximation "valable" du temps d'exécution -, on trouve que la première technique est utilisable jusqu'à **MoveStackCount** = 8. Au-dessus de cela, on utilise la seconde version.

Nous avons donc trois procédures finales, dont la dernière est la seule à devoir être utilisée de l'extérieur.

```

function MakeShortProcOfRegisterMethod(const Method: TMethod;
    UsedRegCount: Byte; MoveStackCount: Word): Pointer;

const
    MoveStackItem: LongWord = $00244C87; // 874C24 xx  XCHG ECX,[ESP+xx]
    MoveRegisters: array[0..7] of Byte = (
        $87, $0C, $24, // XCHG  ECX,[ESP]
        $51,           // PUSH  ECX
        $8B, $CA,     // MOV   ECX,EDX
        $8B, $D0      // MOV   EDX,EAX
    );

type
    PRegisterRedirector = ^TRegisterRedirector;
    TRegisterRedirector = packed record
        MovEAXObj: Byte;
        ObjAddress: Pointer;
        Jump: TJumpInstruction;
    end;

var
    MoveStackSize, MoveRegSize: Integer;
    InstrPtr: Pointer;
    I: Cardinal;
begin
    if UsedRegCount >= 3 then
        UsedRegCount := 4;
        MoveRegSize := 2*UsedRegCount;
        MoveStackSize := 4*MoveStackCount;

        GetMem(Result, MoveStackSize + MoveRegSize + SizeOf(TRegisterRedirector));
        InstrPtr := Result;

        for I := MoveStackCount downto 1 do
            begin
                // I shl 26 => I*4 in the most significant byte (kind of $ I*4 00 00 00)
                PLongWord(InstrPtr)^ := (I shl 26) or MoveStackItem;
                Inc(Integer(InstrPtr), 4);
            end;

            Move(MoveRegisters[SizeOf(MoveRegisters) - MoveRegSize],
                InstrPtr^, MoveRegSize);
            Inc(Integer(InstrPtr), MoveRegSize);

            with PRegisterRedirector(InstrPtr)^ do
                begin
                    MovEAXObj := $B8;
                    ObjAddress := Method.Data;
                    MakeJump(Jump, Method.Code);
                end;
            end;

function MakeLongProcOfRegisterMethod(const Method: TMethod;
    MoveStackCount: Word): Pointer;

type
    PRegisterRedirector = ^TRegisterRedirector;
    TRegisterRedirector = packed record
        Reserved1: array[0..8] of Byte;
        MoveStackCount4: LongWord;
        Reserved2: array[13..20] of Byte;
        FourMoveStackCount1: LongWord;
        Reserved3: array[25..29] of Byte;
        ObjAddress: Pointer;
        Jump: TJumpInstruction;
    end;
    
```

```

const
Code: array[0..SizeOf(TRegisterRedirector)-1] of Byte = (
    $56, // PUSH ESI
    $57, // PUSH EDI
    $51, // PUSH ECX

    $8B, $F4, // MOV ESI,ESP
    $51, // PUSH ECX
    $8B, $FC, // MOV EDI,ESP
    $B9, $FF, $FF, $FF, $FF, // MOV ECX,MoveStackCount+4

    $F3, $A5, // REP MOVSD

    $59, // POP ECX
    $5F, // POP EDI
    $5E, // POP ESI

    $89, $8C, $24, $EE, $EE, $EE, $EE,
        // MOV [ESP + 4*(MoveStackCount+1)],ECX
    $8B, $CA, // MOV ECX,EDX
    $8B, $D0, // MOV EDX,EAX
    $B8, $DD, $DD, $DD, $DD, // MOV EAX,0
    $E9, $CC, $CC, $CC, $CC // JMP MethodAddress
);

begin
GetMem(Result, SizeOf(Code));
Move(Code[0], Result^, SizeOf(Code));

with PRegisterRedirector(Result)^ do
begin
MoveStackCount4 := MoveStackCount+4;
FourMoveStackCount1 := 4 * (MoveStackCount+1);
ObjAddress := Method.Data;
MakeJump(Jump, Method.Code);
end;
end;

function MakeProcOfRegisterMethod(const Method: TMethod;
UsedRegCount: Byte; MoveStackCount: Word = 0): Pointer;
begin
Assert((MoveStackCount = 0) or (UsedRegCount >= 3));

if MoveStackCount <= 8 then
Result := MakeShortProcOfRegisterMethod(
Method, UsedRegCount, MoveStackCount)
else
Result := MakeLongProcOfRegisterMethod(Method, MoveStackCount);
end;

```

## X - Convention d'appel cdecl

### X-A - Les difficultés

La convention d'appel **cdecl** ressemble *très* fort à la convention **stdcall**. La seule différence est que en **stdcall**, la procédure appelée fait un RET N où N est la taille des paramètres en pile : c'est donc elle qui libère les paramètres en pile. En **cdecl**, l'appelé fait toujours un RET simple, et c'est l'appelant qui effectue un ADD ESP,N après le CALL : c'est donc l'appelant qui libère les paramètres de la pile.

Cette différence peut sembler bénigne, et pourtant c'est par sa faute que la convention **cdecl** est la plus difficile à traiter. En effet, le lecteur attentif se souviendra que lorsque nous avons introduit **stdcall**, nous avons dit que justement, comme l'appelé - qui libère les paramètres - sait bien qu'elle a un paramètre **Self**, les paramètres sont libérés correctement sans que nous ayons besoin de faire quoi que ce soit.

Ici, ce n'est plus le cas, la routine qui libère les paramètres, c'est l'appelant, et lui ne sait pas qu'il y a un paramètre supplémentaire. Il en libère donc un de moins qu'il n'en faut : si par exemple la *procédure* a 3 paramètres, alors la *méthode* en a 4 ; le code d'appel créé par Delphi passe 3 paramètres, et en libère donc 3, or il devrait en libérer 4, car entre temps on en a ajouté un. C'est le crash assuré dans les instructions qui suivent, car alors la pile est corrompue.

Nous devons donc faire un CALL au lieu d'un JMP, de manière à pouvoir libérer notre paramètre, avant de retourner à l'appelant original. Seulement voilà, si on fait un CALL, on ajoute notre propre adresse de retour dans la pile, et donc tous les paramètres sont décalés pour l'appelé !

Il faut donc écrire notre adresse de retour à *la place* de l'adresse de retour originale. Mais dans ce cas, *nous* (notre procédure construite à l'exécution) ne savons plus où retourner après, puisqu'on a perdu cette information. Il faut donc la stocker ailleurs, sans changer l'adresse des paramètres par rapport au bas de la pile.

La première idée qui vient à l'esprit est de réutiliser la méthode de déplacement de mémoire que nous avons utilisée avec **register**. Mais les routines **cdecl** ont cette faculté extraordinaire de pouvoir accepter une liste variable de paramètres (les ... du C/C++, transposés en Delphi avec le mot-clef **varargs**). On ne peut donc pas toujours savoir la taille des paramètres à déplacer.

La deuxième idée est de se fixer une limite raisonnable pour la taille des paramètres, par exemple  $64 \times 4 = 256$  octets, et de déplacer tous ces 256 octets pour aller placer notre information à cet endroit. Cependant, cela déplacerait des données sur lesquelles pourraient éventuellement pointer des adresses passées en paramètre. En fait, c'est extrêmement fréquent. Ce n'est donc pas une solution acceptable.

La troisième solution est donc de stocker cette adresse de retour dans une variable globale. Comme des appels de ce type sont susceptibles d'être imbriqués, il faut en faire une liste (chaînée, sans doute). Et comme chaque thread a sa propre liste d'appels, cette liste doit être stockée dans une variable **threadvar**.

### X-B - Le code assembleur de base

Pour ne pas surcharger le code des procédures à créer à l'exécution, nous allons nous servir de deux routines utilitaires chargées de stocker et de récupérer la valeur de retour depuis la liste. Pour l'instant, voici leurs prototypes, elles seront détaillées dans la section suivante.

```
procedure StoreCDeclReturnAddress(ReturnAddress: Pointer); stdcall;  
function GetCDeclReturnAddress: Pointer;
```

💡 Pourquoi avoir utilisé la convention d'appel **stdcall** pour **StoreCDeclReturnAddress** ? Cela se justifie parce que l'appel de cette routine sera la première instruction du code assembleur, et qu'à ce moment l'adresse de retour que nous voulons stocker se trouve précisément au bas de la pile, là où l'attend la convention d'appel **stdcall**. On évite un **POP EAX**, autrement dit.

Avec ces deux routines supposées définies, le code assembleur des routines à créer ressemble à ceci :

```
CALL    StoreCDeclReturnAddress
PUSH   CallbackObj
CALL   CallbackMethod
CALL   GetCDeclReturnAddress
JMP    EAX
```

Simple ? Oui, mais on a oublié un détail : que se passe-t-il si **CallbackMethod** est une *fonction*, qui renvoie donc une valeur, et ce potentiellement dans EAX et EDX (ce dernier pour les **Int64**) ? Les informations sont alors perdues par l'appel à **PopCDeclReturnAddress**. Il faut donc les sauvegarder quelque part. Ici, la pile semble être le meilleur choix.

```
CALL    StoreCDeclReturnAddress
PUSH   CallbackObj
CALL   CallbackMethod
PUSH   EAX
POP    EDX
CALL   GetCDeclReturnAddress
POP    EDX
XCHG  EAX,[ESP]
RET
```

Le **XCHG EAX,[ESP]** remplit de double rôle de poper EAX et de pusher le résultat de **GetCDeclReturnAddress**, lui-même réutilisé immédiatement après par le **RET**. Une autre solution aurait été de sauvegarder ce résultat dans ECX, poper EAX, puis faire un **JMP ECX** au lieu du **RET**. Mais cette alternative est plus lente.

## X-C - Les routines de stockage et de récupération de l'adresse de retour

Ces routines n'ont rien d'exceptionnel. C'est du Delphi habituel, avec la gestion d'une pile chaînée. Je ne vois pas ici le besoin d'expliquer outre mesure leur fonctionnement.

```
type
    PCDeclCallInfo = ^TDeclCallInfo;
    TDeclCallInfo = packed record
        Previous: PCDeclCallInfo; // Pointeur vers le contexte précédent
        ReturnAddress: Pointer; // Adresse de retour de l'appel
    end;

threadvar
    // Liste des infos sur les appels cdecl (spécifique à chaque thread)
    CDeclCallInfoList: PCDeclCallInfo;

procedure StoreCDeclReturnAddress(ReturnAddress: Pointer); stdcall;
var
    CurInfo: PCDeclCallInfo;
begin
    New(CurInfo);
```

```
CurInfo.Previous := CDeclCallInfoList;  
CurInfo.ReturnAddress := ReturnAddress;  
CDeclCallInfoList := CurInfo;  
end;  
  
function GetCDeclReturnAddress: Pointer;  
var  
  CurInfo: PCDeclCallInfo;  
begin  
  CurInfo := CDeclCallInfoList;  
  Assert(CurInfo <> nil);  
  CDeclCallInfoList := CurInfo.Previous;  
  Result := CurInfo.ReturnAddress;  
  Dispose(CurInfo);  
end;
```

Fini ? Non. Il reste un problème, et non des moindres...

## X-D - Que se passe-t-il en cas d'exception ?

Delphi est un langage de haut niveau dont une des plus grandes forces, après l'orienté objet et les RTTI, est le mécanisme des exceptions.

Seulement, dans notre cas, ce mécanisme est bien ennuyeux. Une simple exception déclenchée au sein de la méthode appelée va corrompre cette pile d'adresses de retour.

Je vous entends déjà crier : il faut mettre un **try..finally** autour de l'appel à la méthode ! Je me suis fait la même réflexion, et j'ai tenté de l'implémenter, en assembleur bien sûr.

Des tutoriels existent sur Internet qui expliquent comment fonctionnent les **try..finally** en assembleur, notamment  [A Crash Course on the Depths of Win32 : Structured Exception Handling](#) (générique) ou  [Exception Handling Dans VB6](#) (orienté VB mais en français). Cependant, avant de vous jeter dessus, sachez que la solution du **try..finally** n'est pas acceptable ici.

Le problème est que Windows est très paranoïaque dès qu'il a affaire à des exceptions. Il teste si tout est bien comme il a prévu que ce soit. En particulier, il teste si le SEH enregistré est bien dans la pile. Pas de chance, nous n'avons pas le droit d'enregistrer notre SEH sur la pile, pour les mêmes raisons que nous ne pouvons pas y stocker l'adresse de retour. Si l'on tente de l'enregistrer dans le tas, le programme plante littéralement en cas d'exception : Windows croit avoir affaire à un cas de pile corrompue...

Rien à faire, on ne peut donc pas utiliser les **try..finally**. Il faut trouver une autre astuce.

## X-E - Identifier les parties corrompues de la pile

Puisqu'on ne peut établir de gestionnaire **try..finally**, on ne peut garantir l'intégrité de la pile de contextes. Afin de pouvoir déterminer jusqu'à quel point la pile est corrompue, nous allons introduire une nouvelle information de contexte dans **TCDeclCallInfo**.

L'idée est la suivante. Lorsqu'on appelle **StoreCDeclReturnAddress**, celle-ci identifie les parties corrompues de celle-ci, les supprime, et ensuite seulement enregistre son information. **GetCDeclReturnAddress** applique le même algorithme, sauf qu'elle vérifie en plus que l'information identifiée valide qu'elle trouve correspond à l'information donnée.

Mais quelle est l'information supplémentaire ? Ma solution est la valeur du registre ESP, le pointeur de pile. En effet, cette valeur diminue strictement au fur à mesure que l'imbrication des appels est profonde. Elle est insensible à la récursion, et est mise à jour par le système de gestion des exceptions de Windows.

Les parties invalides de la pile, ou plutôt *la* partie invalide, est l'ensemble des enregistrements qui portent une valeur enregistrée de ESP inférieure à la valeur courante de ce registre. Le fait que **StoreCDeclReturnAddress** supprime les parties corrompues autant que que **GetCDeclReturnAddress** garantit que la pile des contextes a des éléments dont les valeurs enregistrées de ESP sont strictement croissantes (à partir du haut de la pile).

Nous allons d'abord modifier les routines **StoreCDeclReturnAddress** et **GetCDeclReturnAddress**, en leur adjoignant quelques autres routines utilitaires. Celles-ci prennent toutes deux un paramètre supplémentaire indiquant la valeur courante du registre ESP : **StackPointer**.

```
type
  PCDeclCallInfo = ^TDeclCallInfo;
  TDeclCallInfo = packed record
    Previous: PCDeclCallInfo; /// Pointeur vers le contexte précédent
    StackPointer: Pointer; /// Valeur de ESP au moment de l'appel
    ReturnAddress: Pointer; /// Adresse de retour de l'appel
  end;

threadvar
  /// Liste des infos sur les appels cdecl (spécifique à chaque thread)
  CDeclCallInfoList : PCDeclCallInfo;

function GetLastValidCDeclCallInfo(StackPointer: Pointer;
  AllowSame: Boolean): PCDeclCallInfo;
var
  Previous: PCDeclCallInfo;
begin
  Result := CDeclCallInfoList;
  while (Result <> nil) and
    (Cardinal(Result.StackPointer) <= Cardinal(StackPointer)) do
  begin
    if AllowSame and (Result.StackPointer = StackPointer) then
      Break;
    Previous := Result.Previous;
    Dispose(Result);
    Result := Previous;
  end;
end;

procedure StoreCDeclReturnAddress(
  StackPointer, ReturnAddress: Pointer); stdcall;
var
  LastInfo, CurInfo: PCDeclCallInfo;
begin
  LastInfo := GetLastValidCDeclCallInfo(StackPointer, False);

  New(CurInfo);
  CurInfo.Previous := LastInfo;
  CurInfo.StackPointer := StackPointer;
  CurInfo.ReturnAddress := ReturnAddress;
  CDeclCallInfoList := CurInfo;
end;

function GetCDeclReturnAddress(StackPointer: Pointer): Pointer; register;
var
  LastInfo: PCDeclCallInfo;
begin
  LastInfo := GetLastValidCDeclCallInfo(StackPointer, True);

  if (LastInfo = nil) or (LastInfo.StackPointer <> StackPointer) then
```

```
begin
  // This is a very, very bad case: everything has gone wrong! Stop dead!
  CDeclCallInfoList := LastInfo;
  Assert(False);
  Halt(1);
end;

CDeclCallInfoList := LastInfo.Previous;
Result := LastInfo.ReturnAddress;
Dispose(LastInfo);
end;
```

## X-F - Le code assembleur final

Nous avons ajouté un paramètre aux deux routines de stockage et de récupération. Il faut donc légèrement (pas mal en fait) revoir le code assembleur. Il faut passer la bonne valeur de ESP à chacune de ces deux routines, et bien faire en sorte que la *même* valeur soit envoyée aux deux routines (pour un même appel de notre routine créée, cela s'entend).

```
PUSH    ESP
CALL    StoreCDeclReturnAddress
PUSH    CallbackObj
CALL    CallbackMethod
MOV     [ESP], EAX
MOV     EAX, ESP
PUSH    EDX
CALL    GetCDeclReturnAddress
POP     EDX
XCHG   EAX, [ESP]
RET
```

Mis à part le jeu de transfert de valeurs pour conserver EAX et EDX, tout en envoyant la bonne valeur de ESP, il n'y a rien de bien méchant dans ce code, comparé à ce que nous avons déjà pu voir avec **register**.

## X-G - La fonction MakeProcOfCDeclMethod

On commence à être habitué : identifier les OpCodes de chaque instruction, et les placer dans une zone de mémoire allouée sur le tas. Puisque cette fois-ci il y a plus de code immuable, il était intéressant d'utiliser un tableau de **Byte** constant pour initialiser ces parties.

```
function MakeProcOfCDeclMethod(const Method: TMethod): Pointer;

type
  PCDeclRedirector = ^TCDeclRedirector;
  TCDeclRedirector = packed record
    PushESP: Byte;
    CallStoreAddress: TJumpInstruction;
    PushObj: Byte;
    ObjAddress: Pointer;
    CallMethod: TJumpInstruction;
    MovESPEAX: array[0..2] of Byte;
    MoveAXESP: array[0..1] of Byte;
    PushEDX: Byte;
    CallGetAddress: TJumpInstruction;
    PopEDX: Byte;
    Xchg: array[0..2] of Byte;
```

```

Ret: Byte;
end;

const
Code: array[0..SizeOf(TCDeclRedirector)-1] of Byte = (
    $54, // PUSH ESP
    $E8, $FF, $FF, $FF, $FF, // CALL StoreCDeclReturnAddress
    $68, $EE, $EE, $EE, $EE, // PUSH CallbackObj
    $E8, $DD, $DD, $DD, $DD, // CALL CallbackMethod
    $89, $04, $24, // MOV [ESP],EAX
    $8B, $C4, // MOV EAX,ESP
    $52, // PUSH EDX
    $E8, $CC, $CC, $CC, $CC, // CALL GetCDeclReturnAddress
    $5A, // POP EDX
    $87, $04, $24, // XCHG EAX,[ESP]
    $C3 // RET
);

begin
    GetMem(Result, SizeOf(Code));
    Move(Code[0], Result^, SizeOf(Code));

    with PCDeclRedirector(Result)^ do
    begin
        MakeCall(CallStoreAddress, @StoreCDeclReturnAddress);
        ObjAddress := Method.Data;
        MakeCall(CallMethod, Method.Code);
        MakeCall(CallGetAddress, @GetCDeclReturnAddress);
    end;
end;

```

## X-H - La dernière touche contre les fuites mémoire

Nous avons omis de préciser jusqu'ici que cette solution est sujette à des fuites mémoire en cas d'exception. Si une nouvelle routine de ce style est rappelée après, pas de problème : la suppression des données corrompues libèrera les données. Mais si rien ne vient plus jusqu'à la fin du thread, on s'expose à des fuites.

Pour remédier un minimum à cela, ajoutons encore une petite routine toute simple :

```

procedure ClearCDeclCallInfo;
begin
    GetLastValidCDeclCallInfo(Pointer($FFFFFFFF), False);
    CDeclCallInfoList := nil;
end;

```

L'appel de cette routine en fin de thread va libérer toute la pile des contextes. Pour épargner la peine de l'appeler aux programmes mono-thread, on ajoute un appel à cette routine dans le code de finalisation de l'unité.

```

initialization
finalization
    ClearCDeclCallInfo;
end.

```

À présent, il faut que les conditions suivantes soient *toutes* remplies pour avoir des fuites mémoire :

- Une exception est levée et non gérée dans une méthode ainsi appelée ;
- ET cela se passe dans un thread ;
- ET aucune autre méthode n'est appelée de la même façon d'ici la fin du thread ;
- ET le thread n'appelle pas ClearCDecllInfo d'ici la fin de son exécution.

J'estime cela assez acceptable. Si ce n'est pas votre cas, je serai ravi de découvrir votre solution au problème :-).







## XI - Conclusion

Ce tutoriel vous a emmené dans une petite partie des profondeurs de Delphi, pour vous faire découvrir des astuces de programmation très utiles. Si vous pensez que c'est moche de programmer ainsi, je ne vous en blâmerai pas. Mais sachez que c'est avec un système similaire que sont implémentées les fonctionnalités les plus puissantes du langage Delphi, comme le partage COM d'interfaces ou l'encapsulation des **IDispatch** dans des **Variant**.


Si vos envies de frissons ne sont pas encore apaisées, je vous invite à lire encore l'annexe Pour aller plus loin : se servir de ObjAuto.

J'espère que ce tutoriel vous aura plu et aidé. Ou mieux, qu'il vous aura donné l'envie de découvrir les secrets les plus obscurs du langage Delphi.

## XII - Liens

-  **Combiner des procédures et des méthodes**
-  **Utilisation de l'assembleur en ligne avec Delphi**
-  **Décodage du jeu d'instructions x86/x64**
-  **OlllyDbg**
-  **A Crash Course on the Depths of Win32 : Structured Exception Handling**
-  **Exception Handling Dans VB6**


## XIII - Pour aller plus loin : se servir de ObjAuto

 Cette section suppose l'utilisation d'une version 2005 ou supérieure de Delphi.

Je ne vous cacherai pas que je suis resté déçu de mon implémentation de **MakeProcOfRegisterMethod**. En effet, celle-ci ne peut fonctionner si l'on ne renseigne pas correctement le nombre de registres utilisés, ainsi que la taille de la pile à déplacer. D'autre part, si la signature des méthodes et routines mises en jeu change, il faut également modifier l'appel à **MakeProcOfRegisterMethod**.

J'ai donc fouillé un peu plus profondément dans les unités système de Delphi, et j'ai trouvé une unité fort intéressante : l'unité **ObjAuto.pas**. Et avec elle la directive **{METHODINFO ON/OFF}**

Pour cette section, un pré-requis supplémentaire est de connaître les RTTI de base des méthodes publiées, activées avec la directive **{TYPEINFO ON}** ou **{M+}**. La directive **{METHODINFO ON}** ajoute, en plus de ces RTTI de base, des informations détaillées sur les méthodes.

 Les cas dans lesquels il est possible de profiter de ce que nous allons voir ici sont très rares. Et je n'ai moi-même pas encore pu imaginer de telle situation. Je vous entraîne ici seulement dans un de mes plaisirs : explorer le langage Delphi plus loin que personne d'autre ;-)

*Donc soit vous vous faites plaisir en lisant cette annexe, soit ne la lisez pas : vous en seriez déçu.*

### XIII-A - L'unité ObjAuto

L'unité **ObjAuto** définit trois structures qui permettent de lire ces RTTI.

```

type
  TCallingConvention = (ccRegister, ccCdecl, ccPascal, ccStdCall, ccSafeCall);

  TParamFlags = set of (pfVar, pfConst, pfArray, pfAddress, pfReference, pfOut,
    pfResult);

  PMethodInfoHeader = ^TMethodInfoHeader;
  TMethodInfoHeader = packed record
    Len: Word;
    Addr: Pointer;
    Name: ShortString;
  end;

  PReturnInfo = ^TReturnInfo;
  TReturnInfo = packed record
    Version: Byte; // Must be 1
    CallingConvention: TCallingConvention;
    ReturnType: PTypeInfo;
    ParamSize: Word;
  end;

  PParamInfo = ^TParamInfo;
  TParamInfo = packed record
    Flags: TParamFlags;
    ParamType: PTypeInfo;
    Access: Word;
    Name: ShortString;
    
```

```
end;
```

**!** Tout comme au sein des RTTI des types de données, les **ShortString** sont ici ce que j'ai coutume d'appeler des **packed ShortString**. Elles n'ont que la taille strictement utile, donc leur longueur + 1 en octets.

Pour "commencer", on récupère un *header* de type **PMethodInfoHeader** au moyen de la routine **GetMethodInfo** déclarée dans cette même unité.

```
function GetMethodInfo(Instance: TObject;
  const MethodName: ShortString): PMethodInfoHeader;
```

Si les RTTI étendues (METHODINFO) ont été activées, la structure **TMethodInfoHeader** est suivie en mémoire d'une structure **TReturnInfo**, elle-même suivie d'autant de **TParamInfo** que la méthode a de paramètres.

L'unité **ObjAuto** propose encore quelques routines qui ne nous intéressent pas ici.

## XIII-B - Naviguer dans les RTTI étendues

Pour pouvoir "passer au-dessus" des **packed ShortString**, nous utiliserons la routine suivante :

```
function SkipPackedShortString(Value: PShortstring): Pointer; inline;
begin
  Result := Pointer(Integer(Value) + PByte(Value)^ + 1);
end;
```

Pour les versions ne supportant pas l'*inlining*, voici une alternative en assembleur, inspirée de **TypeInfo.GetTypeData** :

```
function SkipPackedShortString(Value: PShortstring): Pointer;
asm
  { ->   EAX Pointer to a packed ShortString           }
  { <-   EAX Pointer to data following this packed ShortString }
  XOR    EDX, EDX
  MOV    DL, [EAX]
  LEA   EAX, [EAX].Byte[EDX+1]
end;
```

Avec cette routine, il est plus facile de naviguer dans les RTTI avancées.

Puisqu'il n'existe pas de champ indiquant le nombre de paramètres, il faut jouer sur le champ **Len** de **TMethodInfoHeader** : parcourir les paramètres jusqu'à dépasser la taille des RTTI avancées.

Voici un code basique parcourant ces données.

```
var
  MethodInfo: PMethodInfoHeader;
  InfoEnd: Pointer;
  ReturnInfo: PReturnInfo;
```

```
ParamInfo: PParamInfo;
begin
  MethodInfo := GetMethodInfo(SomeObject, 'SomeMethod');
  InfoEnd := Pointer(Integer(MethodInfo) + MethodInfo.Len);
  ReturnInfo := SkipPackedShortString(@MethodInfo.Name);
  ParamInfo := PParamInfo(Integer(ReturnInfo) + SizeOf(TReturnInfo));

  while Cardinal(ParamInfo) < Cardinal(InfoEnd) do
  begin
    DoSomethingWithParamInfo;

    ParamInfo := SkipPackedShortString(@ParamInfo.Name);
  end;
end;
```

On peut l'appliquer à un objet **SomeObject** de type **TSomeClass**, ressemblant à ceci :

```
type
{$METHODINFO ON}
TSomeClass = class
public
  function SomeMethod(Param: Integer): Boolean;
end;
{$METHODINFO OFF}
```

### XIII-C - Une unique routine avec CallingConvention

Une information des plus intéressantes est le champ **CallingConvention** de **TReturnInfo**. Il va en effet nous permettre de centraliser les quatre routines **MakeProcOfXXXMethod** en une seule.


Le code de cette routine **MakeProcOfAutoMethod** est somme toute basique. La seule difficulté réside dans les méthodes **register**, pour lesquelles on délèguera à une autre routine **GetAutoRegisterInfo** la tâche ingrate de déterminer **UsedRegCount** et **MoveStackCount**. Cette routine sera développée dans les sections suivantes.

```
function MakeProcOfAutoMethod(Self: Pointer;
  MethodInfo: PMethodInfoHeader): Pointer;
var
  Method: TMethod;
  ReturnInfo: PReturnInfo;
  UsedRegCount: Byte;
  MoveStackCount: Word;
begin
  Method := MakeMethod(MethodInfo.Addr, Self);

  ReturnInfo := SkipPackedShortString(@MethodInfo.Name);
  Assert(Cardinal(ReturnInfo) < Cardinal(MethodInfo) + MethodInfo.Len);
  Assert(ReturnInfo.Version = 1);

  case ReturnInfo.CallingConvention of
    ccRegister:
      begin
        GetAutoRegisterInfo(MethodInfo, UsedRegCount, MoveStackCount);
        Result := MakeProcOfRegisterMethod(Method, UsedRegCount, MoveStackCount);
      end;
    ccCdecl: Result := MakeProcOfCDeclMethod(Method);
    ccPascal: Result := MakeProcOfPascalMethod(Method);
    ccStdCall, ccSafeCall: Result := MakeProcOfStdCallMethod(Method);
    else Result := nil; // should never get here
```

```
end;
end;
```

 Vous aurez remarqué l'assertion portant sur l'adresse de **ReturnInfo**. Le rôle de cette assertion est de vérifier que la méthode en question a été compilée avec **{METHODINFO ON}**. Si ce n'était pas le cas, les RTTI seraient cantonnées à l'adresse et au nom de la fonction. Ces informations sont exploitées par les méthodes **MethodAddress** et **MethodName** de **TObject**.

La seconde assertion sert à vérifier la "version" des RTTI étendues. À ce jour, on en est toujours à la version 1. Mais si Borland a prévu ce champ, c'est parce qu'ils ont en tête de possibles changements futurs.

### XIII-D - Compter le nombre de registres utilisés par la méthode

N'allons pas trop vite, et commençons par compter le nombre de registres utilisés. Et encore ! Nous allons compter le nombre de registres utilisés pour l'appel de la *méthode*. Il s'avère que ce comptage est vraiment facile.

En effet, la structure **TParamInfo** comporte le champ **Access**, qui indique la façon dont est transmise (et récupérée) un paramètre. Ce champ a deux types de valeurs possibles. Soit il vaut **paEAX**, **paEDX** ou **paECX**, indiquant le registre utilisé pour le transmettre. Soit il vaut une valeur supérieure et divisible par 4, indiquant sa position dans la pile.

La position dans la pile est "optimisée" pour la *récupération* des paramètres. Cela se traduit par le fait qu'elle vaut 8 octets "de trop". Ces 8 octets sont ceux pris par les empilements respectifs de l'adresse de retour et du registre EBP.

Pour compter les registres, il suffit donc de compter les paramètres dont la valeur **Access** est strictement inférieure à **paStack**.

```
procedure GetAutoRegisterInfo(MethodInfo: PMethodInfoHeader;
    out UsedRegCount: Byte; out MoveStackCount: Word);
var
    InfoEnd: Pointer;
    ReturnInfo: PReturnInfo;
    ParamInfo: PParamInfo;
begin
    InfoEnd := Pointer(Integer(MethodInfo) + MethodInfo.Len);
    ReturnInfo := SkipPackedShortString(@MethodInfo.Name);
    ParamInfo := PParamInfo(Integer(ReturnInfo) + SizeOf(TReturnInfo));

    // Compute UsedRegCount (for the method, not for the procedure!)
    UsedRegCount := 0;
    while (UsedRegCount < 3) and (Cardinal(ParamInfo) < Cardinal(InfoEnd)) do
    begin
        if ParamInfo.Access < paStack then
            Inc(UsedRegCount);
        ParamInfo := SkipPackedShortString(@ParamInfo.Name);
    end;

    ...
end;
```

### XIII-E - Déterminer le paramètre stocké par ECX dans la procédure

L'étape suivante est de déterminer le paramètre qui est stocké dans ECX dans l'appel de *procédure*. Rappelez-vous les difficultés de la convention **register**, c'était ce paramètre qui jouait un rôle déterminant.

Un réflexe est de dire que ce paramètre est celui qui suit immédiatement la paramètre stocké dans ECX pour la méthode. Malheureusement, ce n'est pas tout à fait vrai. Il se peut en effet qu'il existe des paramètres qui soient toujours passés par la pile, et qui donc ne peuvent être passés dans ECX. Il faut trouver un moyen de les ignorer.


Rappelons les règles qui déterminent si un paramètre doit être passé par la pile. Il doit l'être s'il est de type flottant, méthode ou **Int64**, et s'il n'est ni **var** ni **out**. Il se trouve que les champs **Flags** et **ParamType** de **TParamInfo** permettent d'obtenir ces informations.

**Flags** est un ensemble de valeurs dont 3 nous intéressent ici : **pfVar**, **pfOut** et **pfResult**. Si l'une de ces trois valeurs est incluse dans cet ensemble, alors le paramètre est passé par référence, et quelque soit son type, il peut être passé dans un registre. **pfVar** et **pfOut** ont un sens évident. **pfResult** lui, est présent lorsque **ParamInfo** pointe sur le pseudo-paramètre qui contient l'adresse où enregistrer la valeur de retour. Il n'est présent que lorsque cette valeur est effectivement retournée *via* un emplacement mémoire alloué par l'appelant.

**ParamType** pointe sur les RTTI du type du paramètre. On peut tester si la valeur **Kind** fait partie de l'ensemble **[tkFloat, tkMethod, tkInt64]** et, le cas échéant, exiger un passage sur la pile.

Cela nous conduit à la petite routine ci-dessous :

```
function NeedStack(ParamInfo: PParamInfo): Boolean;
const
  RefFlags = [pfVar, pfOut, pfResult];
  StackKinds = [tkFloat, tkMethod, tkInt64];
begin
  Result := (ParamInfo.Flags * RefFlags = []) and
    (ParamInfo.ParamType^.Kind in StackKinds);
end;
```

 Pour rappel, l'opérateur *\** employé avec des opérands d'un type ensemble construit l'intersection entre les deux opérands.

Nous pouvons donc maintenant facilement identifier le paramètre qui sera stocké dans ECX dans la procédure. En sortie du code déjà donné, la variable **ParamInfo** pointe sur le paramètre suivant directement celui stocké dans ECX dans la méthode. Il est donc aussi le premier candidat à être le paramètre passé dans ECX dans la procédure.

Il suffit de boucler jusqu'à arriver à la fin des paramètres ou à trouver un paramètre qui accepte d'être passé par registre.

```
...
// Skip parameters that need to be passed by stack
while (Cardinal(ParamInfo) < Cardinal(InfoEnd)) and NeedStack(ParamInfo) do
  ParamInfo := SkipPackedShortString(@ParamInfo.Name);
...
```

## XIII-F - Le final

La fin est toute proche. Il reste à différencier deux cas de figure.

Le premier est si l'on a atteint la fin des paramètres. Dans ce cas, il n'y a aucun paramètre stocké dans ECX dans la procédure. Il faut alors décrémenter **UsedRegCount**, puisque dans la procédure on utilise un registre de moins (le **Self**) et on en n'a pas de plus. Accessoirement, **MoveStackCount** vaut 0.

```
...
// If there are no more parameters, the procedure will have 1 used reg less
if Cardinal(ParamInfo) >= Cardinal(InfoEnd) then
begin
  Dec(UsedRegCount);
  MoveStackCount := 0;
end else
...

```

Dans le cas contraire, **UsedRegCount** vaut forcément 3, et doit le rester. Il faut alors déterminer **MoveStackCount**. Pour cela, nous nous servons pour la seconde fois de l'information stockée dans le champ **Access** de **TParamInfo**.

Cette fois il s'agit dans tous les cas de l'offset par rapport au bas de la pile, en récupération. Il se trouve qu'il s'agit exactement du nombre d'octets dont il faut déplacer la pile (+ 8 pour l'adresse de retour et EBP). En outre, il faut diviser cette donnée par 4 puisque **MoveStackCount** se compte en cases, et non en octets.

```
...
// Otherwise, UsedRegCount = 3 and
// MoveStackCount can be found in ParamInfo.Access, because ParamInfo
// points to the parameter stored in ECX in the procedure
begin
  MoveStackCount := ParamInfo.Access div 4;
  Dec(MoveStackCount, 2); // Access numbers include RET and EBP
end;
end;
```

## XIII-G - Code complet de GetAutoRegisterInfo

```
procedure GetAutoRegisterInfo(MethodInfo: PMethodInfoHeader;
  out UsedRegCount: Byte; out MoveStackCount: Word);

function NeedStack(ParamInfo: PParamInfo): Boolean;
const
  RefFlags = [pfVar, pfOut, pfResult];
  StackKinds = [tkFloat, tkMethod, tkInt64];
begin
  Result := (ParamInfo.Flags * RefFlags = []) and
    (ParamInfo.ParamType^.Kind in StackKinds);
end;

var
  InfoEnd: Pointer;
  ReturnInfo: PReturnInfo;
  ParamInfo: PParamInfo;
begin
  InfoEnd := Pointer(Integer(MethodInfo) + MethodInfo.Len);
  ReturnInfo := SkipPackedShortString(@MethodInfo.Name);
  ParamInfo := PParamInfo(Integer(ReturnInfo) + SizeOf(TReturnInfo));

  // Compute UsedRegCount (for the method, not for the procedure!)
  UsedRegCount := 0;
  while (UsedRegCount < 3) and (Cardinal(ParamInfo) < Cardinal(InfoEnd)) do
  begin

```

```
    if ParamInfo.Access < paStack then
        Inc(UsedRegCount);
        ParamInfo := SkipPackedShortString(@ParamInfo.Name);
    end;

    // Skip parameters that need to be passed by stack
    while (Cardinal(ParamInfo) < Cardinal(InfoEnd)) and NeedStack(ParamInfo) do
        ParamInfo := SkipPackedShortString(@ParamInfo.Name);

    // If there are no more parameters, the procedure will have 1 used reg less
    if Cardinal(ParamInfo) >= Cardinal(InfoEnd) then
        begin
            Dec(UsedRegCount);
            MoveStackCount := 0;
        end else

        // Otherwise, UsedRegCount = 3 and
        // MoveStackCount can be found in ParamInfo.Access, because ParamInfo
        // points to the parameter stored in ECX in the procedure
        begin
            MoveStackCount := ParamInfo.Access div 4;
            Dec(MoveStackCount, 2); // Access numbers include RET and EBP
        end;
    end;
```

