

Les objets légers

Le concept d'objet « léger »

D'abord, il faut savoir qu'en plus de la lourdeur de la gestion de l'interface IDispatch, tout module de classe utilise au moins 96 octets de mémoire (sans compter le code) à la base ce qui fait beaucoup quand on pense que (hormis le code des méthodes) que la structure minimum d'un objet qui ne supporte que IUnknown est :

'le contenu de la vtable

```
Private Type VTable  
    Table(0 To 2) As Long  
End Type
```

```
Private Type Objet
```

'le pointeur vers la vtable

pVTable as long

'un compteur d'instance interne

cRefCount as long

```
End type
```

Ce qui nous fait une taille de $4 + 4 = 8$ octets. On ne compte pas la vtable car elle est commune à toutes les instances de l'objet.

Ensuite, il faut savoir que, bien que VB n'est pas type correspondant à IUnknown, il est tout à fait capable de gérer cette interface sans IDispatch. On peut donc déclarer des interfaces à base de IUnknown dans une typelib et les implémenter dans VB.

Un objet léger à n méthodes sera donc un objet qui, une fois instancié, paraîtra comme un objet normale. La seule différence est qu'il ne fera que $8 +$ la taille des données octets pour une instance.

L'implémentation d'un objet léger

Eh bien là ca n'est pas si simple. Vous allez me dire « C'est simple, une typelib et un Implements et le tour est joué ». Eh bien non ! Un objet léger ca se gère dans un module BAS et avec une typelib.

Vous déclarer votre interface en langage ODL, vous compilez avec mktypelib.exe (pas avec MIDL.exe) et cela vous donne votre typelib. Vous devez définir tous vos interfaces (même ceux déjà existants, comme IEnumVARIANT). Voir le tutoriel sur les bibliothèques de types.

Dans votre projet, vous devez ajouter une référence à la bibliothèque de types créée. Dans un module BAS, vous définissez une fonction, disons InitObjet du type de l'interface avec les paramètres pour les données initiales de l'objet (qui seraient passées au constructeur en C++). Dans cette fonction, il faut :

- Initialiser la vtable si ce n'est pas déjà fait
- Initialiser la structure de l'objet pour la vtable et le nombre de références à 1
- Copier l'adresse de la structure dans la variable InitObjet avec CopyMemory

La gestion des objets léger dans le tas

La structure de l'objet sera alloué dans le tas, c'est à dire dans une zone de mémoire allouée dynamiquement par le système d'exploitation. Il faudra donc contrôler la fonction Release pour libérer la mémoire au bon moment.

Il nous faudra les déclaration suivantes :

```
Private Type VTable
    Methods(0 To 2) As Long
End Type

Private m_pVTable As Long
Private m_VTable As VTable

Private Type typObjetHeap
    pVTable As Long
    cCount As Long

    'données attachées
    '-----
    ' ...
End Type
```

Toutes les définitions sont privées car la gestion de la structure se fait en interne. On expose uniquement l'interface.

Notons d'abord la première structure Vtable qui contient seulement un tableau. On pourrait se dire que l'on pourrait simplifier en déclarant simplement un tableau mais les tableaux sont libérer avant les structure à la fin de l'exécution. Il pourrait alors arriver que la vtable soit libérée avant l'objet, ce qui conduirait à un crash. La variable globale m_VTable sera donc détruite assurément après l'objet.

La variable globale m_pVTable sert à connaître l'adresse de la vtable et ainsi savoir si elle a été initialisée.

Notons, enfin, la structure typObjetHeap qui est la structure d'un objet en mémoire. Elle comprend pVTable, le pointeur vers la vtable et cCount, le compteur d'instances. Dans l'exemple je n'ai pas repris les données de l'objet.

Voici, maintenant, le code pour initialiser un objet (notez que cette construction apporte plus que New et donne un équivalent des constructeurs du C++) :

```
Public Function FuncPtr(ByVal addr As Long) As Long
    FuncPtr = addr
End Function

Public Function InitObjetHeap() As <nom_interface>
    'pointeur vers la structure de l'objet
    Dim ptrObjet As Long
    'contenu de l'objet
    Dim Objet As typObjetHeap

    'si la vtable n'est pas initialisée
```

```

If m_pVTable = 0 Then
    'on la remplit
    With m_VTable
        'IUnknown
        .Methods(0) = FuncPtr(AddressOf QueryInterface)
        .Methods(1) = FuncPtr(AddressOf AddRef)
        .Methods(2) = FuncPtr(AddressOf Release)

        'initialisation des autres méthodes de l'interface à implémenter dans l'ordre de
        définition dans la typelib
    End With
    'et on en garde l'adresse
    m_pVTable = VarPtr(m_VTable)
End If

'on construit l'objet
With Objet
    'le pointeur vers la vtable
    .pVTable = m_pVTable
    'le compteur de référence : on crée un objet donc il est à un
    .cCount = 1

    'initialisation des autres données de l'objet
    '...
End With

'on alloue de l'espace mémoire pour l'objet
ptrObjet = CoTaskMemAlloc(LenB(Objet))
'si succès
If ptrObjet Then
    'on remplit l'objet
    CopyMemory ByVal ptrObjet, Objet, LenB(Objet)
End If

'on assigne la référence à la variable de retour de la fonction
CopyMemory ByVal VarPtr(InitObjetHeap), ptrObjet, 4&

ZeroMemory Objet, LenB(Objet)
End Function

```

Notons les points suivants :

- il nous faut une variable temporaire pour stocker la structure de l'objet avant de la transférer dans la zone mémoire allouée à cet effet.
- nous ne pouvons pas affecter l'adresse d'une fonction, obtenue avec AddressOf, directement à une variable. Il faut donc une fonction qui renvoie le paramètre que l'on lui passe. Si la vtable n'est pas initialisée, on remplit le tableau correspondant.
- nous initialisons la structure de l'objet : on pointe la vtable et on met à 1 le nombre d'instance puisque l'objet renvoyé est une instance.
- Nous allouons une zone mémoire dans le tas, puis nous copions l'objet dans cette zone.

- Comme une variable de type objet (ou interface) est un pointeur, nous copions l'adresse de la structure dans le tas dans la variable de retour de la fonction.
- Le ZeroMemory est nécessaire pour que les éventuelles instances d'objets présentes dans la structure de l'objet ou des tableaux ne soient pas supprimées du fait de la destruction de la structure temporaire.

Utilisation

On écrira le code suivant :

```
Dim objet As <interface>
```

```
Set objet = InitObjetHeap
```

```
'l'utilisation
```

```
Set objet = Nothing
```

La gestion des objets légers dans la pile

La structure de l'objet sera allouée dans la pile, par le biais d'une variable locale. Il faudra donc veiller à ne pas utiliser l'objet en dehors de la portée de la fonction dans laquelle est déclarée la variable locale. Sinon, on risque de libérer la vtable avant l'appel à Release ce qui produit un crash de VB ou de l'exé. A NOTER, il faut toujours déclarer la variable du type de l'interface de l'objet AVANT la structure afin de ne pas libérer la structure avant l'objet. IL EST AUSSI NECESSAIRE DE FAIRE **Set variable_objet = Nothing** A LA FIN DE LA FONCTION.

```
Public Type typObjetStack
    'le pointeur vers la vtable
    pVTable As Long
```

```
    'données attachées
    '-----
    '...
```

```
End Type
```

Notons que cette fois la structure est publique puisqu'il faudra déclarer une variable locale de ce type pour stocker l'objet (en plus de la référence d'objet).

```
Public Function InitObjetStack(ByRef lpStruct As typObjetStack, ByVal cMaxSize As Long)
    As <nom_interface>
    Dim ptrObjet As Long
```

```
    'si la vtable n'est pas initialisée
    If m_pVTable = 0 Then
```

```
        'on la remplit
```

```
        With m_VTable
```

```
            'Unknown
```

```
            .Methods(0) = FuncPtr(AddressOf QueryInterface)
```

```
            .Methods(1) = FuncPtr(AddressOf AddRefRelease)
```

```
            .Methods(2) = FuncPtr(AddressOf AddRefRelease)
```

```

        'Le reste des méthodes de l'interface
    End With
    'et on en garde l'adresse
    m_pVTable = VarPtr(m_VTable)
End If

'on construit l'objet
With lpStruct
    'le pointeur vers la vtable
    .pVTable = m_pVTable

    'on remplit le reste des membres de l'objet
End With

ptrObjet = VarPtr(lpStruct)
'on assigne la référence à la variable de retour de la fonction
CopyMemory ByVal VarPtr(InitObjetStack), ptrObjet, 4&
End Function

```

Notons les points suivants :

- La structure est publique puisqu'il faut passer une variable de ce type en paramètre pour instancier l'objet
- On n'alloue plus d'espace mémoire dans le tas : on se sert de l'espace mémoire de la variable locale passée en paramètre
- On copie l'adresse de la variable dans la référence d'objet

Gérer les méthodes de l'interface de base IUnknown

La méthode *QueryInterface* de l'interface *IUnknown* est codée comme suit SI la fonction constructeur renvoie un type « interface de l'objet » :

```

'cette fonction sert à demander à l'objet s'il sait gérer l'interface iid (c'est un GUID)
'normalement VB n'appelle jamais QueryInterface
'puisque l'on assigne à une variable du type de l'interface et que l'on ne supporte (à part
IUnknown) qu'une seule interface
Private Function QueryInterface( _
    ByVal This As typObjet, _
    ByVal iid As Long, _
    ByRef ppvObject As Long _
) As Long
    'on se contente de refuser l'interface
    ppvObject = 0
    QueryInterface = E_NOINTERFACE
End Function

```

Voyons donc *QueryInterface* dans ce cas... Dans l'utilisation normale des objets légers, cette méthode ne devrait jamais être appelée. En effet, on ne peut pas réellement affecter le retour de la fonction constructeur (*InitObjet*) à une variable *Object* (puisque les interfaces que nous pouvons implémentons ne supportent pas *IDispatch*). Nous nous contenterons donc de ne pas retourner de référence d'objet et renvoyer *E_NOINTERFACE* pour signaler à VB que l'on ne veut pas de *QueryInterface*.

Si la fonction constructeur renvoie un *IUnknown* on codera *QueryInterface* comme suit :

```
Private Function QueryInterface( _  
    ByRef This As typObjet, _  
    ByVal iid As Long, _  
    ByRef ppvObject As Long _  
) As Long  
If This.cCount > 1 Then  
    ppvObject = 0  
    QueryInterface = E_NOINTERFACE  
Else  
    This.cCount = This.cCount + 1  
    ppvObject = VarPtr(This)  
    QueryInterface = 0  
End If  
End Function
```

Dans ce cas, il faut autoriser un seul *QueryInterface* pour l'affectation à la variable qui va référencer l'objet. Pour cela, on regarde s'il y a moins de deux instances d'objet. Si oui, on incrémente le nombre de référence et on renvoie l'adresse de *This* (la référence) dans *ppvObject*. Puis il faut refuser tout autre cast pour éviter les erreurs de cast que l'objet ne supporte pas. Dans ce cas, on met 0 dans *ppvObject* et on renvoie *E_NOINTERFACE* pour dire que l'on ne veut pas de ce cast.

Voyons maintenant, *AddRef* et *Release* :

'cette fonction incrémente un compteur de référence (nombre d'instance) de l'objet

```
Private Function AddRef(ByRef This As typObjetHeap) As Long  
This.cCount = This.cCount + 1  
AddRef = This.cCount  
End Function
```

'cette fonction décrémente un compteur de référence (nombre d'instance) de l'objet
'quand le compteur atteint 0, sa structure est libérée

```
Private Function Release(ByRef This As typObjetHeap) As Long
```

```
This.cCount = This.cCount - 1
```

```
Release = This.cCount
```

'Si l'on est dans le cas des objets sur pile, on n'a pas besoin de ce qui suit.

```
If This.cCount = 0 Then
```

```
    'on libère éventuellement les ressources allouées pour l'objet
```

```
    '...
```

```
    'et celle de l'objet
```

```
    CoTaskMemFree ByVal VarPtr(This)
```

```
End If
```

```
End Function
```

AddRef et *Release* sont complémentaires. VB appelle *AddRef* pour dire qu'il ajoute une référence de l'objet (un pointeur). VB appelle *Release* à chaque fois qu'une variable du type de l'interface sort de sa portée ou qu'elle reçoit *Nothing*. Pour la première fonction, on incrémente un compteur (et on renvoie le compteur, valeur qui n'est pas utilisée). Pour la seconde, on décrémente le compteur (et on renvoie aussi le compteur).

Dans le cas des objets alloués dans le tas, lorsque le compteur atteint 0, on doit libérer la structure de l'objet que l'on a allouée dans le tas. Ceci est impératif pour ne pas avoir de fuites de mémoire.

Dans le cas des objets alloués dans la pile, on n'a rien à faire puisque la mémoire est libérée automatiquement à la fin de la procédure qui contient la variable locale.

Gérer les autres méthodes personnelles

1) Le pointeur This

Toute méthode d'un objet reçoit implicitement un premier paramètre This qui est un pointeur vers la structure de l'objet pour lequel la méthode vient d'être appelée. Il est du type de la structure de l'objet, dans notre cas, il s'agit de typObjetHeap.

2) La valeur de retour

Il y a deux cas à prendre en compte :

- Gestion d'erreur minimale
- Pas de gestion d'erreur

a) Gestion minimale d'erreur

Le premier niveau de gestion d'erreur sous VB se fait par le type HRESULT que vous ne pouvez pas utiliser vous même. C'est l'équivalent d'un Long. Dans les module de classe, la valeur de retour des fonctions est toujours un HRESULT. Mais alors, me dirais vous, comment VB retourne-t-il une valeur à l'appelant.

Eh bien, il utilise un paramètre qui se trouve à la fin de la liste des paramètres. Ce paramètre est de type pointeur et possède les attributs [out,retval] dans le fichier ODL. Dans VB, il sera déclaré comme un ByRef As <type>.

Et Alors dans le HRESULT, on met quoi ? Le code d'erreur biensûr ! Il faut savoir que VB et l'objet Err transpose les codes d'erreur dans la plage &H0001 à &HFFFF et qu'il transpose plusieurs code d'erreur en un seul de l'objet Err. Il se trouve que toutes les erreurs de VB se retrouve dans les valeur de HRESULT dans la plage &H800A0001 à &H800AFFFF. On peut alors renvoyer le code d'erreur VB (par exemple 7 « Mémoire insuffisante ») que l'on Or-era avec &H800A0000. Biensûr, s'il n'y a pas d'erreur, on renvoie 0.

C'est le principe de base de la gestion d'erreur dans VB.

Dans le langage ODL, on aura des méthodes définies comme suit :
HRESULT <nom méthode>(<paramètres>);

b) Pas de gestion d'erreur

Bien que VB s'attende à un HRESULT, on peut passer directement la valeur de retour dans la valeur de retour si la taille est inférieure ou égale à 4 octets. Sinon, il A VOIR...

Si l'on retourne une valeur dans le retour de fonction, on n'aura pas la possibilité de renvoyer un code d'erreur.

Dans le langage ODL, on aura des méthodes définies comme suit :
<type non pointeur> <nom méthode>(<paramètres>);

Si l'on n'a ni besoin de valeur de retour, ni de code d'erreur, on remplacera <type non pointeur> par void.

3) Les différents type de paramètres

Tous les types de bases : Byte, Integer, Long, Single, Double, Boolean, Currency, Date, Enum sont des types qui peuvent être passé directement par valeur. Pour ces types, si le paramètre est défini :

- [in] <type> , c'est un ByVal As <type>
- [in,out] <type>*, c'est un ByRef As <type>

<type> est à choisir parmi un nom d'enum, unsigned char, short, long, float, double, boolean, currency, DATE.

Il n'y pas d'autres combinaisons. A noter que pour ces types, on peut définir une valeur par défaut, qui rend le paramètre optionel, avec l'attribut defaultvalue(<valeur>).

Les types String (BSTR, LPSTR, LPWSTR), Object (IDispatch*) et autres types objets (toutes les interfaces) sont, par définition, des types pointeurs. Pour ces types, on définit :

- [in] BSTR, [in] LPSTR, [in] LPWSTR, c'est un ByVal As String
- [in,out] BSTR*, [in,out] LPSTR*, [in,out] LPWSTR*, c'est un ByRef As String
- [in] IDispatch*, c'est un ByVal As Object
- [in,out] IDispatch**, c'est un ByRef As Object

A noter que ByRef As Object (ou As <interface>) n'est nécessaire que lorsque l'on compte modifier la référence d'objet, ou en renvoyer une. Pour un simple passage de paramètre à des fins d'utilisation, on optera toujours pour un ByVal

Les types tableaux sont eux aussi des pointeurs :

- [in] SAFEARRAY(<type>)* et [in,out] SAFEARRAY(<type>)* sont équivalent puisque l'on ne peut pas passer de tableau par valeur.

<type> est n'importe quel type, simple, structure ou tableau.

h) Les paramètres [out,retval]

Le dernier paramètre de la liste peut être attribué avec [out,retval] s'il est du type ByRef c'est à dire pointeur (long*, boolean*, ..., BSTR*, IDispatch**, <interface>**, ...). Il ne peut y en avoir qu'un seul.

i) Les ParamArray : l'attribut vararg

Pour déclarer que la liste des paramètres n'est pas connue à partir d'un certain paramètre, on utilise ParamArray suivi d'un nom de paramètre de type tableau de Variant. En langage ODL, il faut ajouter vararg dans les attributs de la méthode (et non dans les attributs du paramètre). Il s'en suit une définition de méthode comme suit :

```
[vararg] <type> <nom>(<liste de paramètres connus>,SAFEARRAY(VARIANT)* <nom arg>);
```

j) Les propriétés : les attributs propput, propputref et propget

Pour déclarer une propriété en lecture-écriture, il faut deux fonctions : une pour lire et une pour écrire. Nous distinguerons deux cas :

- Les propriétés normales
- Les propriétés objets

1. Les propriétés normales : Property Get / Property Let (propget / propput)

Une propriété de ce type aura deux entrées dans la vtable. En langage ODL, elle auront le prototype suivant :

- Pour la lecture, Property Get :
 - [propget] HRESULT <nom propriété>([out,retval] <type>* <nom>) ;
 - Fonction get_<nom propriété>(ByRef This, ByRef <nom> As <type>) As Long
- Pour l'écriture, Property Let :
 - [propput] HRESULT <nom propriété>([in] <type> <nom>);
 - Fonction put_<nom propriété>(ByRef This, ByVal <nom> As <type>) As Long

2. Les propriétés objets : Property Get / Property Set (propget / propputref)

Une propriété de ce type aura deux entrées dans la vtable. En langage ODL, elle auront le prototype suivant :

- Pour la lecture, Property Get :
 - [propget] HRESULT <nom propriété>([out,retval] <interface>** <nom>) ;
 - Fonction get_<nom propriété>(ByRef This, ByRef <nom> As <interface>) As Long
- Pour l'écriture, Property Let :
 - [propputref] HRESULT <nom propriété>([in] <interface>* <nom>);
 - Fonction put_<nom propriété>(ByRef This, ByVal <nom> As <interface>) As Long

On peut aussi avoir des propriétés en lecture-seule ou écriture-seule en supprimant une des deux fonctions

Conclusion

Bon, eh bien, vous allez sûrement encore me dire : « A part optimiser, qu'est ce que l'on peut faire de tout ça ». Eh bien, utiliser des pointeurs de fonctions...mais ça c'est le tutoriel suivant...Et puis l'implémentation de For Each...