Rétro-Ingénierie C++
Héritage et polymorphisme

par *Homeostasie (Nicolas.D)*

23/02/2015

(trashomeo (at) gmail (dot) com)

Table des matières

1 Introduction

Suite à un petit challenge de rétro-ingénierie sur un exécutable codé en C++, je me suis rendu compte que l'une des difficultés était de reconstruire la hiérarchie des classes et de retrouver dans certains cas, le réel code exécuté.

En effet, en fonction de l'implémentation de la classe parente (par exemple l'utilisation de méthodes virtuelles) et du type d'héritage (par exemple l'héritage en diamant), la représentation interne d'une classe fille pourra varier et fournir des informations sur la conception d'un programme.

Le but de ce document est d'acquérir une meilleure vision du code généré en fonction du type d'héritage.

Pour cela, j'ai simplement développé des exemples de code C++ puis j'ai examiné et documenté le code assembleur généré avec l'utilisation de IDA.

Le travail effectué est seulement valable pour une architecture x86. Les programmes ont été compilé avec Microsoft Visual Studio 8 et sans optimisation de compilation.

2 Généralités

2.1 Ordre d'appel des constructeurs

Le constructeur appelé en premier est celui de la classe la plus ancêtre. Le dernier étant toujours celui de la classe dérivée.

Dans le cas d'une dérivation directe, les constructeurs sont appelés les uns après les autres.

Dans le cas, d'une dérivation indirecte, les appels aux constructeurs sont imbriqués.

2.2 Ordre d'appel des destructeurs

Les destructeurs sont appelés dans l'ordre inverse des constructeurs. Ainsi, le destructeur appelé en premier est celui de la classe dérivée, le second celui de la classe immédiatement supérieure, et ainsi de suite jusqu'au destructeur de la classe la plus ancêtre.

2.3 Ligature statique et dynamique

2.3.1 Modèle de classes

```
class A
{
private:
	int m_iVal;

public:
	A::A(){m_iVal = 1;};
	int GetInt(){return m_iVal;}
};

class B:public A
{
private:
	int m_iVal2;
	int m_iVal3;

public:
	B::B(){m_iVal2 = 2; m_iVal3 = 2;};
	int GetInt(){return m_iVal2;}
};
```

Dans ce modèle, on a :

- la classe B qui hérite de la classe A,
- la fonction GetInt() de la classe A qui est redéfinie dans la classe B.

2.3.2 Code C++

```
int a = 0;

/* Instanciation dynamique */
A* poA = new A;
B* poB = new B;

a = poA->GetInt();
a += poB->GetInt();

/* Tentative de ligature dynamique pour poA pointe sur B */
poA = poB;
a += poA->GetInt();
```

On focalisera principalement sur l'assignation « poA=poB » où le but est que poA pointe sur l'instance de la classe B.

2.3.3 Code généré

```
...
call    A:A(void)
mov     ebx, eax
...
call    B:B(void)
mov     edi, eax
...
mov     esi, eax
mov     eax, ebx
call    A::GetInt(void)

mov     esi, eax
mov     eax, edi
call    B::GetInt(void)

add     esi, eax
mov     eax, edi
call    A::GetInt(void)

add     eax, esi
...
```

Malgré l'affectation « poA=poB », la fonction GetInt() de la classe A est appelée. Nous aurions pu nous attendre que cela soit celle de la classe B.

En fait, pendant la compilation, le compilateur lie le pointeur poA à la fonction A::GetInt(), et ceci, quelque soit le contenu de poA pendant l'exécution du programme. C'est ce que l'on appelle une ligature statique.

Le polymorphisme permet de contourner ce problème au travers l'utilisation de fonctions virtuelles afin que le compilateur génère une ligature dynamique. Ainsi le code à appeler sera déterminé pendant l'exécution du programme.

3 Héritage multiple

3.1 Modèle de classes

```
class A
{
  ...
  void SetInt(int iVal){m_iVal = iVal;}
  int GetInt(){return m_iVal;}
  ...
}

class B:public A
{
  ...
  void SetInt(int iVal){m_iVal = iVal;}
  int GetInt(){return m_iVal;}
  ...
}

class C
{
  ...
  void SetInt(int iVal){m_iVal = iVal+1;}
  int GetInt(){return this->m_iVal-1;}
  ...
}

class C1:public B, public C
{
  ...
  int GetInt(){     return this->m_iVal+3;}
  ...
}
```

Dans ce modèle, on a :

- la classe B qui hérite de la classe A,
- la classe C1 qui hérite de la classe B et C
- la fonction GetInt() est surchargée dans toutes les classes.

3.2 Instanciation d'un objet de la classe dérivée

3.2.1 Structures de la classe C1

```
00000000 CLASS_A         struc ; (sizeof=0x4)
00000000 m_iVal          dd ?
00000004 CLASS_A         ends

00000000 CLASS_B         struc ; (sizeof=0xC)
00000000 oA              CLASS_A ?
00000004 m_iVal1         dd ?
00000008 m_iVal2         dd ?
0000000C CLASS_B         ends

00000000 CLASS_C         struc ; (sizeof=0x4)
00000000 m_iVal          dd ?
00000004 CLASS_C         ends
```

```
00000000 CLASS_C1        struc ; (sizeof=0x14)
00000000 oB              CLASS_B ?
0000000C oC              CLASS_C ?
00000010 m_iVal          dd ?
00000014 CLASS_C1        ends
```

On observe simplement que :

- la classe C1 contient les objets de base des classes ancêtres (B et C),
- la classe B contient une instance de la classe ancêtre A.

4 Héritage indirecte d'une même classe parente

4.1 Modèle de classes

```
class A
{
  ...
  void SetInt(int iVal){m_iVal = iVal;}
  int GetInt(){return m_iVal;}
}

class B:public A
{
  ...
  void SetInt(int iVal){A::SetInt(m_iVal3+iVal);}
  int GetInt(){return (m_iVal3 = A::GetInt() + 1);}
}

class C:public A
{
  ...
  int GetInt(){return (m_iVal = GetInt() + 1);}
}

class C1:public B, public C
{
  ...
  int GetInt(){return m_iVal;}
}
```

Dans ce modèle, on a :

- la classe B qui hérite de la classe A,
- la classe C qui hérite de la classe A,
- la classe C1 qui hérite de la classe B et C
- la fonction GetInt() est redéfinie dans les classes B, C et C1,
- la fonction SetInt() de la classe A est redéfinie dans la classe B.

4.2 Instanciation d'un objet de la classe dérivée

4.2.1 Structures de la classe C1

```
00000000 CLASS_A         struc ; (sizeof=0x4)
00000000 m_iVal          dd ?
00000004 CLASS_A         ends

00000000 CLASS_B         struc ; (sizeof=0xC)
00000000 oA              CLASS_A ?
00000004 m_iVal2         dd ?
00000008 m_iVal3         dd ?
0000000C CLASS_B         ends

00000000 CLASS_C         struc ; (sizeof=0x8)
00000000 oA              CLASS_A ?
00000004 m_iVal          dd ?
00000008 CLASS_C         ends

00000000 CLASS_C1        struc ; (sizeof=0x18)
```

```
00000000 oB              CLASS_B ?
0000000C oC              CLASS_C ?
00000014 m_iVal          dd ?
00000018 CLASS_C1        ends
```

On observe que :

- la classe C1 contient les objets de base des classes ancêtres (B et C),
- les classes B et C contiennent une instance de la classe ancêtre A.
- la classe C1 hérite donc de la classe A par les classes B et C. Deux instances de la classe A sont donc accessibles à partir de C1.

Un problème d'ambiguïté apparaît dans ce cas de figure et doit être résolu via l'opérateur de résolution de portée "::".

4.2.2 Appel respectif aux fonctions GetInt() et SetInt() à partir d'un objet C1

4.2.2.1 Code C++

```
iVal += poC1->A::GetInt();       (1)
iVal += poC1->B::GetInt();       (2)
iVal += poC1->C::GetInt();       (3)
poC1->B::SetInt(1);              (4)
poC1->C::SetInt(2);              (5)
```

4.2.2.2 Code généré

```
.text:00401315                mov     ecx, [ebp+poC1] ; this
.text:00401318                call    A::GetInt(void) ;(1)
.text:00401318
.text:0040131D                add     eax, [ebp+iVal]
.text:00401320                mov     [ebp+iVal], eax
.text:00401323                mov     ecx, [ebp+poC1] ; this
.text:00401326                call    B::GetInt(void) ;(2)
.text:00401326
.text:0040132B                add     eax, [ebp+iVal]
.text:0040132E                mov     [ebp+iVal], eax
.text:00401331                mov     ecx, [ebp+poC1]
.text:00401334                add     ecx, 0Ch        ; this
.text:00401337                call    C::GetInt(void) ;(3)
...
...
.text:0040135E                push    1               ; iVal
.text:00401360                mov     ecx, [ebp+poC1] ; this
.text:00401363                call    B::SetInt(int)  ;(4)
.text:00401363
.text:00401368                push    2               ; iVal
.text:0040136A                mov     ecx, [ebp+poC1]
.text:0040136D                add     ecx, 0Ch        ; this
.text:00401370                call    A::SetInt(int)  ;(5)
```

- Cas 1, 2 et 4: L'instance de la classe C1 possède comme premier champ un objet de classe B, incorporant lui-même un objet de la classe A en premier champ, "poC1" est donc directement passé en tant que pointeur "this",

- Cas 3 et 5: Il est nécessaire de faire pointer "this" sur l'objet de la classe C de l'instance poC1. Un offset de 0x0C (sizeof CLASS_B) est donc ajouté à "poC1".

5 Héritage en diamant

Lorsque au moins deux classes d'une hiérarchie possèdent une classe de base commune, il est possible de déclarer cette classe commune virtuelle.

De cette manière, on force le partage d'une unique instance plutôt que deux copies séparées.

On résout ainsi le problème d'ambiguïté précédemment décrit.

5.1 Modèle de classes

```
class A
{
  ...
  void SetInt(int iVal){m_iVal = iVal;}
  int GetInt(){return m_iVal;}
}

class B:virtual public A
{
  ...
  void SetInt(int iVal){A::SetInt(m_iVal3+iVal);}
  int GetInt(){return (m_iVal3 = A::GetInt() + 1);}
}

class C:virtual public A
{
  ...
  int GetInt(){return (m_iVal = GetInt() + 1);}
}

class C1:public B, public C
{
  ...
}
```

Dans ce modèle, on a :

- la classe B qui hérite virtuellement de la classe A,
- la classe C qui hérite virtuellement de la classe A,
- la classe C1 qui hérite de la classe B et C
- la fonction GetInt() est redéfinie dans les classes B et C,
- la fonction SetInt() de la classe A est redéfinie dans la classe B.

5.2 Instanciation d'un objet de la classe dérivée

5.2.1 Structures de la classe C1

```
00000000 CLASS_A          struc ; (sizeof=0x4)
00000000 m_iVal           dd ?
00000004 CLASS_A          ends

00000000 CLASS_B          struc ; (sizeof=0x10)
00000000 vbtable          dd ?  ; const C1::`vbtable'{for `C'} dd 0
00000000                        ;                              dd 0Ch
00000004 m_iVal2          dd ?
00000008 m_iVal3          dd ?
0000000C oA               CLASS_A ?
00000010 CLASS_B          ends

00000000 CLASS_C          struc ; (sizeof=0xC)
00000000 vbtable          dd ? ; const C::`vbtable' dd 0
00000000                       ;                    dd 8
00000004 m_iVal           dd ?
00000008 oA               CLASS_A ?
0000000C CLASS_C          ends

00000000 CLASS_C1         struc ; (sizeof=0x1C)
00000000 vbtable_for_B    dd ?  ; const C1::`vbtable'{for `B'} dd 0
00000000                        ;                              dd 18h
00000004 m_B_iVal2        dd ?
00000008 m_B_iVal3        dd ?
0000000C vbtable_for_C    dd ?  ; const C1::`vbtable'{for `C'} dd 0
0000000C                        ;                              dd 0Ch
00000010 m_C_iVal         dd ?
00000014 m_iVal           dd ?
00000018 oA               CLASS_A ?
0000001C CLASS_C1         ends
```

L'héritage en diamant sans polymorphisme fait apparaître deux nouveautés pour les classes déclarées virtuelles:

- L'ajout d'un nouveau membre nommé « vbtable » (Virtual Base Table)
- Une organisation différente des données où la classe de base commune est déclarée en fin de la classe fille au lieu d'être en début.

La « vbtable » possède deux champs sur 4 octets:

- Le premier correspondant à l'offset pour atteindre le sommet de la classe fille
- Le second correspondant à l'offset vers l'objet de la classe de base commune

On remarquera que le compilateur utilise la même « vbtable » (const C1::`vbtable'{for `C'}) pour la classe B et l'objet C de la classe C1. Probablement une optimisation du fait que les offsets utilisés sont identiques.

5.2.2 Appel respectif aux fonctions GetInt() et SetInt() à partir d'un objet C1

5.2.2.1 Code C++

```
iVal += poC1->A::GetInt();      (1)
iVal += poC1->B::GetInt();      (2)
iVal += poC1->C::GetInt();      (3)
poC1->B::SetInt(1);             (4)
poC1->C::SetInt(2);             (5)
```

5.2.2.2 Code généré

```
.text:0040139E                  mov     edx, [ebp+poC1]
.text:004013A1                  mov     eax, [edx+CLASS_C1.vbtable_for_B] ;
.text:004013A1                          ; const C1::`vbtable'{for `B'} dd 0
.text:004013A1                          ;                               dd 18h
.text:004013A3                  mov     ecx, [ebp+poC1]
.text:004013A6                  add     ecx, [eax+4]  avec [eax+4] = 18h
; this = poC1 + 0x18: Pointe sur l'objet A de C1
.text:004013A9                  call    A::GetInt(void)
.text:004013A9
.text:004013AE                  add     eax, [ebp+iVal]
.text:004013B1                  mov     [ebp+iVal], eax
.text:004013B4                  mov     ecx, [ebp+poC1]
; this = poC1: poC1 pointe directement sur l'objet B de C1
.text:004013B7                  call    B::GetInt(void)
.text:004013B7
.text:004013BC                  add     eax, [ebp+iVal]
.text:004013BF                  mov     [ebp+iVal], eax
.text:004013C2                  mov     ecx, [ebp+poC1]
.text:004013C5                  add     ecx, 0Ch
; this = poC1 + 0x0C: poC1 pointe sur l'objet C de C1
.text:004013C8                  call    C::GetInt(void)
...
...
.text:004013EC                  mov     [ebp+iVal2], eax
.text:004013EF                  push    1
.text:004013F1                  mov     ecx, [ebp+poC1]
; this = poC1: poC1 pointe directement sur l'objet B de C1
.text:004013F4                  call    B::SetInt(int)
.text:004013F4
.text:004013F9                  push    2
.text:004013FB                  mov     ecx, [ebp+poC1]
.text:004013FE                  mov     edx, [ecx+CLASS_C1.vbtable_for_B] ;
.text:004013FE                          ; const C1::`vbtable'{for `B'} dd 0
.text:004013FE                          ;                               dd 18h
.text:00401400                  mov     ecx, [ebp+poC1]
.text:00401403                  add     ecx, [edx+4]
; this = poC1 + 0x18: Pointe sur l'objet A de C1
.text:00401406                  call    A::SetInt(int)
```

- Cas 1: L'appel implique de passer dans "this" un pointeur vers l'objet A à partir de la classe C1. L'objet A est retrouvé à partir du deuxième champ de la vbtable qui correspond à un offset pour atteindre l'objet A à partir de l'adresse de la classe C1.
- Cas 2,4: Étant donné que l'instance de la classe C1 possède comme premier champ un objet de classe B, "poC1" est directement passé en tant que "this". Cependant, il est important d'avoir conscience que la « vbtable » utilisée ne sera pas celle par défaut de la classe B (const C1::`vbtable'{for `C'}) mais celle de la classe C1 (const C1::`vbtable'{for `B'}). Un exemple illustrant cet aspect sera présenté plus bas.
- Cas 3: Il est nécessaire de faire pointer "this" sur l'objet de la classe C de l'instance poC1. Comme précédemment, la « vbtable » utilisée n'est pas celle par défaut de la classe C (const C::`vbtable') mais celle de la classe C1 (const C1::`vbtable'{for `C'}).
- Cas 5: La fonction SetInt() de la classe C n'étant pas redéfinie, celle de la classe parente A est directement appelé. Le comportement est ensuite le même que le cas 1.

5.2.3 Appel à la fonction GetInt() à partir d'une instance C1 – Autre cas

Cette fois-ci, nous allons visualiser le code généré lors d'un appel à la fonction GetInt() à partir d'une instance C1, appelant elle-même la fonction GetInt() de la classe A.

Ainsi, au travers cet exemple, nous allons illustrer l'intérêt primordial des « vbtables » pour chaque objet dérivant d'une classe virtuelle.

Cela nous permettra de comprendre comment l'héritage en diamant résout le problème d'ambiguïté en ne recopiant qu'un seul objet de la classe de base commune et pourquoi il existe des « vbtables » surchargées pour chaque objet dérivé.

5.2.3.1 Code C++

```
int A::GetInt()
{
  return m_iVal;
}

int B::GetInt()
{
  return (m_iVal3 = A::GetInt() + 1);
}

int _tmain(int argc, _TCHAR* argv[])
{
  ...
  iVal = poC1->B::GetInt();
  ...
}
```

5.2.3.2 Code généré

```
.text:00401100 public: int __thiscall B::GetInt(void) proc near
.text:00401100
.text:00401100 this            = dword ptr -4
.text:00401100
.text:00401100                 push    ebp
.text:00401101                 mov     ebp, esp
.text:00401103                 push    ecx
.text:00401104                 mov     [ebp+this], ecx
.text:00401107                 mov     eax, [ebp+this]
.text:0040110A                 mov     ecx, [eax+CLASS_B.vbtable] ;
.text:0040110A                         ; const C1::`vbtable'{for `C'} dd 0
.text:0040110A                         ;                              dd 0Ch
.text:0040110A                         ; réécrit par :
.text:0040110A                         ; const C1::`vbtable'{for `B'} dd 0
.text:0040110A                         ;                              dd 18h
.text:0040110C                 mov     edx, [ebp+this]
.text:0040110F                 add     edx, [ecx+4]
.text:00401112                 mov     ecx, edx        ; this
.text:00401114                 call    A::GetInt(void)
...
...
.text:00401270 ; int __cdecl main(int argc, const char **argv, const char **envp)
...
…
; this = poC1: poC1 pointe directement sur l'objet B de C1
.text:004013B4                 mov     ecx, [ebp+poC1]
.text:004013B7                 call    B::GetInt(void)
...
```

D'après les précédentes structures, nous savons que:

- La classe B a pour vbtable "const C1::`vbtable'{for `C'}". L'objet A se situe à l'offset 0x0C de l'adresse de début de la classe B,
- La classe C1 a pour vbtable "const C1::`vbtable'{for `B'}", L'objet A se situe à l'offset 0x18 de l'adresse de début de la classe C1

Dans ce cas présent, si nous utilisions la « vbtable » par défaut de la classe B avec l'instance poC1 , on obtiendrait un offset vers l'objet A à "poC1+0x0C".

Or à cette adresse, nous pointerions sur le champ « vbtable_for_C » de la classe C1 et ce champ serait considéré comme le pointeur "this" de la classe A. Ceci serait désastreux !

Pour remédier à cela, le compilateur génère une « vbtable » propre à la classe C1 pour atteindre l'objet A et surcharge celle par défaut de la classe B. Ainsi, il est possible d'appeler une méthode de la classe A à partir d'un objet de la classe C1 dérivant de A par l'intermédiaire de la classe B, tout en évitant le problème d'ambiguïté.

6 Polymorphisme et héritage multiple

Pour palier au problème lié à la ligature statique, le C++ offre la possibilité de faire du polymorphisme en utilisant le mot clé "virtual" devant les fonctions ciblées.

Afin de s'y retrouver, le compilateur va générer une table de fonctions virtuelles associées à chaque objet implémentant des méthodes virtuelles ou dérivant d'une classe proposant des méthodes virtuelles.

6.1 Modèle de classes

```
class A
{
  ...
  virtual void SetInt(int iVal){m_iVal = iVal;}
  virtual int GetInt(){return m_iVal;}
}

class B:public A
{
  ...
  void SetInt(int iVal){m_iVal = iVal;}
  int GetInt(){return m_iVal;}
}

class C
{
  ...
}

class C1:public B, public C
{
  ...
}
```

Dans ce modèle, on a :

- la classe A qui définie deux fonctions virtuelles GetInt() et SetInt(),
- la classe B qui hérite de la classe A,
- la classe C1 qui hérite des classes B et C,
- les fonctions GetInt() et SetInt() sont redéfinies dans la classe B.

6.2 Instanciation d'un objet de la classe dérivée

6.2.1 Table de fonctions virtuelles générée

```
.rdata:00402184 const A::vftable dd offset A::SetInt(int)
.rdata:00402188                  dd offset A::GetInt(void)

.rdata:00402190 const B::vftable dd offset B::SetInt(int)
.rdata:00402194                  dd offset B::GetInt(void)

.rdata:0040219C const C1::vftable dd offset B::SetInt(int)
.rdata:004021A0                   dd offset B::GetInt(void)
```

Nous pouvons constater les points suivants:

- La classe A possède sa propre « vftable » (Virtual Function Table).
- La classe B redéfinie les fonctions virtuelles de la classe A. De ce fait, la « vftable » de la classe B contient les adresses vers ses fonctions redéfinies.
- La classe C, ne dérivant d'aucune classe et ne possédant pas de fonctions virtuelles, n'a donc pas de « vftable ».
- La classe C1 n'implémente pas de fonctions virtuelles. Mais étant donné qu'elle dérive d'une classe implémentant des fonctions virtuelles, elle possèdent une « vftable ».

A noter que si la classe C1 redéfinie une méthode virtuelle alors la « vftable » contiendrait l'adresse de la fonction redéfinie.

6.2.2 Structures de la classe C1

```
00000000 CLASS_A         struc ; (sizeof=0x4)
00000000 m_iVal          dd ?
00000004 CLASS_A         ends

00000000 CLASS_VIRTUAL_A struc ; (sizeof=0x8)
00000000 vtable          dd ?  ; offset const A::`vftable'
00000004 m_iVal          dd ?
00000008 CLASS_VIRTUAL_A ends

00000000 CLASS_B         struc ; (sizeof=0xC)
00000000 oA              dd ?
00000004 m_iVal2         dd ?
00000008 m_iVal3         dd ?
0000000C CLASS_B         ends

00000000 CLASS_VIRTUAL_B struc ; (sizeof=0x10)
00000000 vtable          dd ?  ; offset const B::`vftable'
00000004 oA              CLASS_A ?
00000008 m_iVal2         dd ?
0000000C m_iVal3         dd ?
00000010 CLASS_VIRTUAL_B ends

00000000 CLASS_VIRTUAL_C1 struc ; (sizeof=0x18)
00000000 vtable          dd ?   ; offset const C1::`vftable'{for `B'}
00000004 oA              CLASS_A ?
00000008 oB              CLASS_B ?
00000010 oC              CLASS_C ?
00000014 m_iVal          dd ?
00000018 CLASS_VIRTUAL_C1 ends
```

La « vftable » de C1 sera mise à jour lors de l'appel à chaque constructeur des classes ancêtres.

/!\ La classe C1 contient les objets de base des classes ancêtres sans leurs « vftables ». J'ai donc dû créer deux structures pour représenter un objet A et B.

7 Polymorphisme et héritage indirecte d'une même classe parente

Le but principal est de pouvoir utiliser la ligature dynamique.

7.1 Modèle de classes

```
class A
{
  ...
  virtual void SetInt(int iVal){m_iVal = iVal;}
  virtual int GetInt(){return m_iVal;}
}

class B:public A
{
  ...
  virtual int GetIntB(){return m_iVal2;}

  void SetInt(int iVal){m_iVal = iVal;}
  int GetInt(){return m_iVal;}
}

class C:public A
{
  void SetIntC1(int iVal){m_iVal = iVal;}
  int GetIntC1(){return m_iVal;}
}

class C1:public B, public C
{
  ...
  int GetInt() {
    return m_iVal++;
  }

  int GetIntB() {
    m_iVal = B::GetIntB();
    return m_iVal++;
  }
}
```

Dans ce modèle, on a :

- la classe A qui définie deux fonctions virtuelles GetInt() et SetInt(),
- la classe B qui hérite de la classe A et qui définie une fonction virtuelle GetIntB(),
- la classe C qui hérite de la classe A,
- la classe C1 qui hérite de la classe B et C. Deux instances de la classe A sont donc accessible à partir de C1.
- la fonction GetInt() est redéfinie dans les classes B et C1,
- la fonction GetIntB() de la classe B est redéfinie dans la classe C1.

7.2 Instanciation d'un objet de la classe dérivée

7.2.1 Table de fonctions virtuelles générée

```
.rdata:00402124 const A::`vftable' dd offset A::SetInt(int)
.rdata:00402128                     dd offset A::GetInt(void)

.rdata:00402130 const B::vftable dd offset B::SetInt(int)
.rdata:00402134                  dd offset B::GetInt(void)
.rdata:00402138                  dd offset B::GetIntB(void)

.rdata:00402140 const C::vftable dd offset A::SetInt(int)
.rdata:00402144                  dd offset A::GetInt(void)

.rdata:0040214C const C1::vftable{for B} dd offset B::SetInt(int)
.rdata:00402150                          dd offset C1::GetInt(void)
.rdata:00402154                          dd offset C1::GetIntB(void)

.rdata:0040215C const C1::vftable{for C} dd offset A::SetInt(int)
.rdata:00402160                          dd offset [thunk]:C1::GetInt adjustor{16}(void)
```

On constate que la classe C1 possède deux « vftables » du fait que:

- La classe B redéfinie les fonctions GetInt() et SetInt() de la classe A
- La classe C1 redéfinie la fonction GetIntB() de la classe B
- La classe C ne redéfinie pas les fonctions de GetInt() et SetInt() de la classe A
- La classe C1 redéfinie la fonction GetInt() de la classe A

Par exemple, en fonction de si on utilise l'objet B ou C de la classe C1, on appelle réciproquement B::SetInt(int) ou A::SetInt(int).

Le signification de "[thunk]:C1::GetInt`adjustor{16}' (void)" au niveau de la vftable "const C1::`vftable'{for `C'}" a pour but d'indiquer:

- L'utilisation de la fonction C1::GetInt(void)
- De soustraire au "this" courant, passé en paramètre, la valeur 16 afin de considérer "this" comme un pointeur sur l'objet CLASS_VIRTUAL_B correspondant finalement à l'adresse de base de l'instance de la classe C1(c.f. Chapitre suivant 7.2.2).

On obtient par exemple le code suivant :

```
.text:004012E0 [thunk]:public: virtual int __thiscall C1::GetInt adjustor{16} (void)
proc near
.text:004012E0                 sub     ecx, 10h
.text:004012E3                 jmp     C1::GetInt(void)
```

7.2.2 Structures de la classe C1

```
00000000 CLASS_A struc ; (sizeof=0x4)
00000000 m_iVal          dd ?
00000004 CLASS_A ends

00000000 CLASS_VIRTUAL_A struc ; (sizeof=0x8)
00000000 vtable          dd ?  ; offset const A::vftable
00000004 m_iVal          dd ?
00000008 CLASS_VIRTUAL_A ends

00000000 CLASS_B         struc ; (sizeof=0xC)
00000000 oA              dd ?
00000004 m_iVal2         dd ?
00000008 m_iVal3         dd ?
0000000C CLASS_B         ends

00000000 CLASS_VIRTUAL_B struc ; (sizeof=0x10)
00000000 vtable          dd ?  ; offset const B::vftable
00000004 oA              CLASS_A ?
00000008 m_iVal2         dd ?
0000000C m_iVal3         dd ?
00000010 CLASS_VIRTUAL_B ends

00000000 CLASS_VIRTUAL_C struc ; (sizeof=0xC)
00000000 vtable          dd ?  ; const C::vftable
00000004 oA              CLASS_A ?
00000008 m_iVal          dd ?
0000000C CLASS_VIRTUAL_C ends

00000000 CLASS_VIRTUAL_C1 struc ; (sizeof=0x20)
00000000 oVirtualB       CLASS_VIRTUAL_B ?       ; const C1::vftable{for B}
00000010 oVirtualC       CLASS_VIRTUAL_C ?       ; const C1::vftable{for C}
0000001C m_iVal          dd ?
00000020 CLASS_VIRTUAL_C1 ends
```

On constate que:

- la classe C1 contient une instance des classes virtuelles B et C mais avec leurs « vftables » par défaut modifiées. Ce comportement s'apparente en partie à celui des « vbtables » précédemment décrit,
- les données de l'objet de la classe A sont dupliquées puisque les classes B et C possèdent une instance de la classe A.

7.2.3 Appels respectifs aux fonctions GetInt() à partir d'un objet C1

7.2.3.1 Code C++

```
iVal += poC1->A::GetInt();      (1)
iVal += poC1->B::GetInt();      (2)
iVal += poC1->C::GetInt();      (3)
```

7.2.3.2 Code généré

```
.text:004011D5                  ; Cas 3
.text:004011D5                  mov     eax, [esi+CLASS_VIRTUAL_C1.oVirtualB]
                                                        ; const C1::vftable{for B}
.text:004011D7                  mov     edx, [eax+4]
.text:004011DA                  mov     ecx, esi
.text:004011DC                  call    edx             ; C1::GetInt(void)

.text:004011DE                  ; Cas 1
.text:004011DE                  lea     ecx, [esi+CLASS_VIRTUAL_C1.oVirtualC]
.text:004011E1                  mov     edi, eax
.text:004011E3                  call    A::GetInt(void)

.text:004011E8                  ; Cas 2
.text:004011E8                  mov     ecx, esi        ; this
.text:004011EA                  mov     ebx, eax
.text:004011EC                  call    B::GetInt(void)
```

- Cas 1: On récupère l'adresse de l'objet virtuel C (dérivant de la classe A) de la classe C1. Le premier membre est donc un objet A et comme la classe C ne redéfinie aucunes fonctions de la classe A, la fonction "A::GetInt(void)" est directement appelée.
- Cas 2: L'instance de la classe C1 possède comme premier champ un objet de classe B, le pointeur sur la classe C1 est directement passé en tant que pointeur "this".
- Cas 3: On récupère un pointeur sur la « vftable » de la classe C1. Puis on appelle la fonction située à l'offset 4 correspondant à "C1::GetInt(void)".

/!\ On remarquera que l'ordre d'appel des méthodes est différent de celui écrit dans le code C++. Cela est probablement dû à de l'optimisation lié au fait que l'addition est une opération commutative.

8 Polymorphisme et héritage multiple avec utilisation d'une classe de base abstraite

8.1 Modèle de classes

```
class A
{

private:
  int m_iVal;

public:
  virtual void SetInt(int iVal) = 0;
  virtual int GetInt() = 0;
  ...
}

class B:public A
{
private:
  int m_iVal2;
  int m_iVal3;

public:
  void SetInt(int iVal){m_iVal = iVal+1;}
  int GetInt(){return m_iVal-1;}
  ...
}

class C:public A
{
private:
  int m_iVal;

public:
  void SetInt(int iVal){this->m_iVal = iVal+2;}
  int GetInt(){return this->m_iVal-2;}
  ...
}

class C1:public B, public C
{
private:
  int m_iVal;

public:
  int GetInt(){ return this->m_iVal+3;}
  ...
}
```

Dans ce modèle, on a :

- la classe A qui définie deux fonctions virtuelles pures GetInt() et SetInt(),
- la classe B qui hérite de la classe A et qui redéfinie les fonctions GetInt() et SetInt(),
- la classe C qui hérite de la classe A et qui redéfinie les fonctions GetInt() et SetInt(),
- la classe C1 qui hérite de la classe B et C. Deux instances de la classe A sont donc accessible à partir de C1.
- la classe C1 redéfinie la fonction GetInt().

8.2 Instanciation d'un objet de la classe dérivée

8.2.1 Table de fonctions virtuelles générée

```
.rdata:00402124 const A::vftable dd offset __purecall
.rdata:00402128                  dd offset __purecall

.rdata:00402130 const B::vftable dd offset B::SetInt(int)
.rdata:00402134                  dd offset B::GetInt(void)
.rdata:00402138                  dd offset B::GetIntB(void)

.rdata:00402140 const C::vftable dd offset C::SetInt(int)
.rdata:00402144                  dd offset C::GetInt(void)

.rdata:0040214C const C1::vftable{for B} dd offset B::SetInt(int)
.rdata:00402150                          dd offset C1::GetInt(void)
.rdata:00402154                          dd offset C1::GetIntB(void)

.rdata:0040215C const C1::vftable{for C} dd offset C::SetInt(int)
.rdata:00402160                          dd offset C1::GetInt(void)
```

Nous pouvons constater que la classe A possède une « vftable » composée de deux pointeurs vers la fonction « __purecall ».

En fait, quand on définit une fonction virtuelle pure, le compilateur place l'adresse d'une fonction de la bibliothèque « C-runtime » nommée « __purecall ».

Pour retrouver l'implémentation de ces fonctions virtuelles pures au sein des classes filles, il suffit d'accéder à l'offset de la « vftable » correspondant à celui de la classe de base.

Dans le cas présenté, les classes B et C dérivent de la classe A directement et la classe C1 dérive indirectement de la classe A.

Pour chacune de ces classes dérivées, on retrouvera l'implémentation des fonctions virtuelles pures SetInt() et GetInt() de la classe A respectivement aux offsets 0 et 4 de leurs « vftables » respectives.

En définissant une nouvelle méthode « AnotherFunction » au sein de la classe A entre les deux fonctions virtuelles pures, nous aurions obtenu la « vftable » de la classe A comme suit:

```
const A::vftable dd offset __purecall
                 dd offset A::AnotherFunction(int)
                 dd offset __purecall
```

8.2.2 Structures de la classe C1

```
00000000 CLASS_A         struc ; (sizeof=0x8)
00000000 vftable         dd ?  ; const A::`vftable' dd offset __purecall
00000000                       ;                    dd offset __purecall
00000004 m_iVal          dd ?
00000008 CLASS_A         ends
```

```
00000000 CLASS_B         struc ; (sizeof=0x10)
00000000 oA              CLASS_A ?
                                 ; const B::`vftable' dd offset B::SetInt(int)
00000000                         ;                     dd offset B::GetInt(void)
00000000                         ;                     dd offset B::GetIntB(void)
00000008 m_iVal2         dd ?
0000000C m_iVal3         dd ?
00000010 CLASS_B         ends

00000000 CLASS_C         struc ; (sizeof=0xC)
00000000 oA              CLASS_A ?
                                 ; const C::`vftable' dd offset C::SetInt(int)
00000000                         ;                    dd offset C::GetInt(void)
00000008 m_iVal          dd ?
0000000C CLASS_C         ends

00000000 CLASS_C1        struc ; (sizeof=0x20)
00000000 oB              CLASS_B ?
00000000              ; const C1::`vftable'{for `B'} dd offset B::SetInt(int)
00000000              ;                              dd offset C1::GetInt(void)
00000000              ;                              dd offset C1::GetIntB(void)
00000010 oC              CLASS_C ?
                      ; const C1::`vftable'{for `C'} dd offset C::SetInt(int)
00000010              ;                              dd offset C1::GetInt(void)
0000001C m_iVal          dd ?
00000020 CLASS_C1        ends
```

Les structures ne laissent rien apparaître de nouveau par rapport à ce que l'on a déjà vu précédemment.

Le problème d'ambiguïté pour accéder à l'objet A est présent puisqu'on peut l'atteindre une instance de A par l'objet B ou par l'objet A de la classe C1.

8.2.3 Appels respectifs aux fonctions GetInt() à partir d'un objet C1

8.2.3.1 Code C++

```
iVal += poC1->B::GetInt();       (1)
iVal += poC1->C::GetInt();       (2)
iVal += poC1->GetInt();          (3)
```

8.2.3.2 Code généré

```
.text:0040135D                   ; cas 1
.text:0040135D                   mov     ecx, [ebp+poC1] ; this
.text:00401360                   call    B::GetInt(void)
.text:00401360
.text:0040136B                   ; cas 2
.text:0040136B                   mov     ecx, [ebp+poC1]
.text:0040136E                   add     ecx, 10h        ; this
.text:00401371                   call    C::GetInt(void)
.text:00401371
.text:0040137C                   ; cas 3
.text:0040137C                   mov     eax, [ebp+poC1]
.text:0040137F                   mov     edx, [eax+CLASS_C1.oB.oA.vftable]
                                                 ; const C1::`vftable'{for `B'}
.text:00401381                   mov     ecx, [ebp+poC1]
.text:00401384                   mov     eax, [edx+4]
.text:00401387                   call    eax
```

- Cas 1: L'instance de la classe C1 possède comme premier champ un objet de classe B, le pointeur sur la classe C1 est directement passé en tant que pointeur "this".
- Cas 2: On récupère l'adresse de l'objet C de la classe C1 à l'offset 0x10 (sizeof(B)) pour passer cette valeur en tant que pointeur "this".
- Cas 3: On récupère un pointeur sur la « vftable » de la classe C1. Puis on appelle la fonction située à l'offset 4 correspondant à "C1::GetInt(void)".

9 Polymorphisme et héritage en diamant

Le but est de pouvoir:

- Éviter la duplication de données lors d'un héritage indirecte d'une classe commune
- Utiliser la ligature dynamique

9.1 Modèle de classes

```
class A
{
  ...
  virtual void SetInt(int iVal){m_iVal = iVal;}
  virtual int GetInt(){return m_iVal;}
}

class B:virtual public A
{
  ...
  int GetIntB(){return m_iVal2;}
  void SetInt(int iVal){A::SetInt(m_iVal3+iVal);}
  int GetInt(){return (A::GetInt() + 1);}
}

class C:virtual public A
{
  ...
  int GetInt(){return (A::GetInt() + 1);}
}

class C1:public B, public C
{
  ...
  int GetIntB(){return (B::GetIntB() + 1);}
  int GetInt(){return this->m_iVal;}
}
```

Dans ce modèle, on a :

- la classe A qui définie deux fonctions virtuelles GetInt() et SetInt(),
- la classe B qui hérite de la classe A avec l'attribut «virtual»,
- la classe B qui redéfinit les fonctions GetInt() et SetInt() de la classe A,
- la classe C qui hérite de la classe A avec l'attribut « virtual »,
- la classe C qui redéfinit la fonction GetInt() de la classe A,
- la classe C1 qui hérite des classes B et C,
- la classe C1 qui redéfinit la fonction GetInt() et la fonction GetIntB() de la classe B (non déclarée avec l'attribut « virtual » pour cette dernière).

9.2 Instanciation d'un objet de la classe dérivée

9.2.1 Table des fonctions virtuelles et table des objets de base

```
.rdata:00403124 const A::vftable dd offset A::SetInt(int)
.rdata:00403128                  dd offset A::GetInt(void)

.rdata:00403130 const B::vftable dd offset [thunk]:B::SetInt vtordisp{-4,0}(int)
.rdata:00403134                  dd offset [thunk]:B::GetInt vtordisp{-4,0}void)

.rdata:00403138 const C1::vbtable{for C} dd 0
.rdata:0040313C                          dd 10h

.rdata:00403144 const C::vftable dd offset A::SetInt(int))
.rdata:00403148                  dd offset [thunk]:C::GetInt vtordisp{-4,0}void)

.rdata:0040314C const C::vbtable dd 0
.rdata:00403150                  dd 0Ch

.rdata:00403158 const C1::vftable dd offset [thunk]:B::SetInt vtordisp{-4,12} (int)
.rdata:0040315C                    dd offset [thunk]:C1::GetInt vtordisp{-4,0}(void)

.rdata:00403160 const C1::vbtable{for B} dd 0
.rdata:00403164                          dd 1Ch
```

9.2.2 Structures de la classe C1

```
00000000 CLASS_A          struc ; (sizeof=0x8)
00000000 vftable          dd ?  ; const A::vftable dd offset A::SetInt(int)
00000000                        ;                  dd offset B::GetIntB(void)
00000004 m_iVal           dd ?
00000008 CLASS_A          ends

00000000 CLASS_B          struc ; (sizeof=0x18)
00000000 vbtable          dd ?  ; const C1::vbtable{for C} dd 0
00000000                        ;                          dd 10h
00000004 m_iVal2          dd ?
00000008 m_iVal3          dd ?
0000000C dwOffset_0       dd ?
00000010 oA               CLASS_A ?
                    ; const B::vftable dd offset [thunk]:B::SetInt vtordisp{-4,0}(int)
00000010            ;                  dd offset [thunk]:B::GetInt vtordisp{-4,0}(void)
00000018 CLASS_B          ends

00000000 CLASS_C          struc ; (sizeof=0x14)
00000000 vbtable          dd ?  ; const C::vbtable dd 0
00000000                        ;                  dd 0Ch
00000004 m_iVal           dd ?
00000008 dwOffset_0       dd ?
0000000C oA               CLASS_A ?
                  ; const C::`vftable' dd offset A::SetInt(int)
0000000C          ;                    dd offset [thunk]:C::GetInt vtordisp{-4,0}(void)
00000014 CLASS_C          ends

00000000 CLASS_C1         struc ; (sizeof=0x24)
00000000 vbtable_for_B    dd ?  ; const C1::vbtable{for B} dd 0
00000000                        ;                          dd 1Ch
00000004 m_B_iVal2        dd ?
00000008 m_B_iVal3        dd ?
0000000C vbtable_for_C    dd ?  ; const C1::vbtable{for C} dd 0
```

```
0000000C                        ;                           dd 10h
00000010 m_C_iVal       dd ?
00000014 m_iVal         dd ?
00000018 dwOffset       dd ?    ; 1) Constructeur de B à partir de C1:
00000018                        ; dwOffset = 0x0C
00000018                        ; 2) Constructeur de C à partir de C1:
00000018                        ; dwOffset = 0x04
00000018                        ; 3) Constructeur de C1 à partir de C1:
00000018                        ; dwOffset = 0x00
0000001C oA             CLASS_A ?
          ; 1) Constructeur de B à partir de C1:
0000001C  ; const B::vftable dd offset [thunk]:B::SetInt vtordisp{-4,0}(int)
0000001C  ;                  dd offset [thunk]:B::GetInt vtordisp{-4,0}(void)
0000001C  ; 2) Constructeur de C à partir de C1:
0000001C  ; const C::vftable dd offset A::SetInt(int)
0000001C  ;                  dd offset [thunk]:C::GetInt vtordisp{-4,0}(void)
0000001C  ; 3) Constructeur de C1 à partir de C1:
0000001C  ; const C1::vftable dd offset [thunk]:B::SetInt vtordisp{-4,12(int)
0000001C  ;                   dd offset [thunk]:C1::GetInt vtordisp{-4,0}(void)
00000024 CLASS_C1       ends
```

On constate:

- L'utilisation de « vbtables » et de « vftables » comme attendu,
- L'apparition d'un nouveau champ que j'ai nommé « dwOffset » dont je n'ai pas compris le rôle.

/!\ La classe A est la seule à posséder une « vftable ». Les autres classes possèdent seulement des « vbtables ».

En fait, les classes qui dérivent de la classe A de manière directe ou indirecte accéderont à leur « vftable » en deux étapes:

- Utilisation de leur « vbtable » pour atteindre l'objet A,
- Utilisation de la « vftable » de l'objet A spécifiquement générée pour chaque classe dérivée lors de l'appel au constructeur.

Nous obtenons donc trois vftables possibles pour l'objet A puisque les classes B, C et C1 dérivent de A:

- const B::`vftable' qui fournit l'implémentation des fonctions redéfinies SetInt() et GetInt() de la classe B,
- const C::`vftable' qui fournit l'implémentation de la fonction redéfinie GetInt() de la classe A et utilise la fonction de base SetInt() de la classe A,
- const C1::`vftable' qui fournit l'implémentation de la fonction redéfinie GetInt() de la classe B et utilise la fonction de base SetInt() de la classe B.

Certains pourrait se demander pourquoi la méthode GetIntB() redéfinie dans la classe C1, initialement définie dans la classe B, n'apparaît pas dans la « vftable » de C1.

Tout simplement parce que GetIntB() n'est pas déclarée comme virtuelle (absence du mot clef 'virtual' dans sa déclaration).

Nous sommes par conséquent dans le cas d'un héritage sans polymorphisme pour cette méthode.

Si nous avions déclaré B::GetIntB() en tant que méthode virtuelle, nous aurions obtenu les deux « vftables » suivantes associées à la classe C1:

```
.rdata:00403160 const C1::vftable{for B}dd offset C1::GetIntB(void)
.rdata:00403168 const C1::vftable{for A}dd offset [thunk]:B::SetInt vtordisp{-4,12}(int)
.rdata:0040316C                          dd offset [thunk]:C1::GetInt vtordisp{-4,0}(void)
```

Ce cas de figure sera expliqué par la suite.

9.2.3 Appels respectifs aux fonctions GetInt() à partir d'un objet C1

9.2.3.1 Code C++

```
iVal += poC1->A::GetInt();       (1)
iVal += poC1->B::GetInt();       (2)
iVal += poC1->C::GetInt();       (3)
iVal += poC1->GetInt();          (4)
iVal2 += poC1->B::GetIntB();     (5)
iVal2 += poC1->GetIntB();        (6)
```

9.2.3.2 Code généré

```
.text:004014DE                   ; cas 1
.text:004014DE                   mov     edx, [ebp+poC1]
.text:004014E1                   mov     eax, [edx+CLASS_C1.vbtable_for_B] ;
.text:004014E1                                   ; const C1::vbtable{for B} dd 0
.text:004014E1                                   ;                          dd 1Ch
.text:004014E3                   mov     ecx, [ebp+poC1]
                                 ; ecx = this = poC1+0x1C = poC1->oA
.text:004014E6                   add     ecx, [eax+4]
.text:004014E9                   call    A::GetInt(void)
.text:004014E9
.text:004014EE                   ; cas 2
.text:004014EE                   add     eax, [ebp+iVal]
.text:004014F1                   mov     [ebp+iVal], eax
.text:004014F4                   mov     ecx, [ebp+poC1]
.text:004014F7                   add     ecx, 10h         ; this - poC1+0x10
.text:004014FA                   call    B::GetInt(void)
.text:004014FA
.text:004014FF                   ; cas 3
.text:004014FF                   add     eax, [ebp+iVal]
.text:00401502                   mov     [ebp+iVal], eax
.text:00401505                   mov     ecx, [ebp+poC1]
.text:00401508                   ; this - poC1 + 0x18
.text:00401508                   add     ecx, 18h
.text:0040150B                   call    C::GetInt(void)
.text:00401533
.text:00401533                   ; cas 4
.text:00401533                   mov     [ebp+iVal], eax
.text:00401536                   mov     ecx, [ebp+poC1]
.text:00401539                   mov     edx, [ecx+CLASS_C1.vbtable_for_B] ;
.text:00401539                                   ; const C1::vbtable{for B} dd 0
```

```
.text:00401539                                ;                               dd 1Ch
.text:0040153B                  mov     eax, [edx+4] ; eax = vbtable[1] = 0x1C
.text:0040153E                  mov     ecx, [ebp+poC1]
.text:00401541                  mov     edx, [ecx+CLASS_C1.vbtable_for_B] ;
.text:00401541                                ; const C1::vbtable{for B} dd 0
.text:00401541                                ;                               dd 1Ch
.text:00401543                  mov     ecx, [ebp+poC1]
.text:00401546                  add     ecx, [edx+4]; ecx = this = poC1+0x1C = poC1->oA
.text:00401549                  mov     edx, [ebp+poC1]
.text:0040154C
.text:0040154C                  ; eax = *(poC1+0x1C) = poC1->oA.vftable
.text:0040154C                  mov     eax, [edx+eax]
.text:0040154F                  mov     edx, [eax+4]
        ;[thunk]:public: virtual int __thiscall C1::GetInt vtordisp{-4, 0}(void)
.text:00401552                  call    edx
.text:00401552
.text:0040155A                  ; cas 5
.text:0040155A                  mov     ecx, [ebp+poC1] ; this
.text:0040155D                  call    B::GetIntB(void)
.text:0040155D
.text:00401568                  ; cas 6
.text:00401568                  mov     ecx, [ebp+poC1] ; this
.text:0040156B                  call    C1::GetIntB(void)
```

- Cas 1: L'appel implique de passer dans "this" un pointeur vers l'objet A à partir de la classe C1. L'objet A est retrouvé à partir du deuxième champ de la « vbtable » qui corresponds à un offset pour atteindre l'objet A à partir de l'adresse de la classe C1,

- Cas 2: Un décalage de 0x10 est passé à ecx; Ce décalage corresponds à l'offset par défaut de la « vbtable » appliquée à la classe B pour atteindre l'objet A. Ce décalage est finalement annulé une fois dans la fonction B::GetInt(void). Quel intérêt?!

- Cas 3: Principe identique au cas 2,

- Cas 4: Principe du cas 1 pour récupérer un pointeur vers l'objet A de la classe C1. Puis on accède à la « vftable » de la classe C1 (const C1::`vftable') à partir de l'objet A .

- Cas 5: L'instance de la classe C1 possède comme premier champ un objet de classe B, le pointeur sur la classe C1 est directement passé en tant que "this",

- Cas 6: La fonction B::GetIntB(), redéfinie dans C1 et n'étant pas déclarée virtuelle, le compilateur lie statiquement l'appel à la fonction C1::GetIntB(void).

/!\ Concernant le cas 4, étant donné que nous appelons une méthode de la classe C1, on s'attend à positionner le registre « ecx » (correspondant au pointeur « this ») avec comme valeur le pointeur poC1. Or, le registre est configuré avec un pointeur sur l'instance de la classe A.

En examinant le code généré pour la fonction GetInt() de la classe C1, on remarquera que

finalement l'accès au membre de cette classe s'effectue de manière relative à l'objet A de la classe C1.

Ci-dessous le code de la fonction GetInt() redéfinie :

```
; int __thiscall C1__GetInt(B *this)
public: virtual int __thiscall C1::GetInt(void) proc near

this= dword ptr -4

push    ebp
mov     ebp, esp
push    ecx
mov     [ebp+this], ecx
mov     eax, [ebp+this]
mov     eax, [eax-8]
mov     esp, ebp
pop     ebp
retn
public: virtual int __thiscall C1::GetInt(void) endp
```

L'accès à l'adresse pointée par « this-8 » permet d'accéder au champ m_iVal de la classe C1, soit l'offset 0x14.

10 Polymorphisme et héritage en diamant - Encore plus loin

Le but est de pouvoir:

- Éviter la duplication de données lors d'un héritage indirecte d'une classe commune
- Utiliser la ligature dynamique

10.1 Modèle de classes

```
class A
{
  ...
  virtual void SetInt(int iVal){m_iVal = iVal;}
  virtual int GetInt(){return m_iVal;}
}

class B:virtual public A
{
  ...
  virtual void SetIntB(int iVal){m_iVal3 = iVal;}
  virtual int GetIntB(){return m_iVal2;}

  void SetInt(int iVal){A::SetInt(m_iVal3+iVal);}
  int GetInt(){return (A::GetInt() + 1);}
}

class C:virtual public A
{
  ...
  int GetInt(){return (A::GetInt() + 1);}
}

class C1:public B, public C
{
  ...
  int GetIntB(){return (B::GetIntB() + 1);}
  int GetInt(){return this->m_iVal;}
}
```

Dans ce modèle, on a :

- la classe A qui définie deux fonctions virtuelles GetInt() et SetInt(),
- la classe B qui hérite de la classe A avec l'attribut « public »,
- la classe B qui redéfinit les fonctions GetInt() et SetInt() de la classe A,
- la classe C qui hérite de la classe A avec l'attribut « public »,
- la classe C qui redéfinit la fonction GetInt() de la classe A,
- la classe C1 qui hérite des classes B et C,
- la classe C1 qui redéfinit la fonction GetInt() et la fonction GetIntB() de la classe B (cette fois-ci déclarée avec l'attribut « virtual »).

10.2 Instanciation d'un objet de la classe dérivée

10.2.1 Table des fonctions virtuelles et table des objets de base

```
.rdata:00403124 const A::vftable dd offset A::SetInt(int)
.rdata:00403128                  dd offset A::GetInt(void)

.rdata:00403130 const B::vftable{for B} dd offset B::SetIntB(int)
.rdata:00403134                         dd offset B::GetIntB(void)

.rdata:0040313C const B::vftable{for A} dd offset [thunk]:B::SetInt vtordisp{-4,0}(int)
.rdata:00403140                         dd offset [thunk]:B::GetInt vtordisp{-4,0}(void)

.rdata:00403144 const B::vbtable dd -4
.rdata:00403148                  dd 10h

.rdata:00403150 const C::vftable dd offset A::SetInt(int)
.rdata:00403154                  dd offset [thunk]:C::GetInt vtordisp{-4,0}(void)

.rdata:00403158 const C::vbtable dd 0
.rdata:0040315C                  dd 0Ch

.rdata:00403164 const C1::vftable{for B} dd offset B::SetIntB(int)
.rdata:00403168                          dd offset C1::GetIntB(void)

.rdata:00403170 const C1::vftable{for A}dd offset [thunk]:B::SetInt vtordisp{-4,12}(int)
.rdata:00403174                         dd offset [thunk]:C1::GetInt vtordisp{-4,0}(void)

.rdata:00403178 const C1::vbtable{for B} dd -4
.rdata:0040317C                          dd 1Ch

.rdata:00403180 const C1::vbtable{for C} dd 0
.rdata:00403184                          dd 10h
```

Par rapport au cas précédent, nous remarquons l'ajout d'une nouvelle table de fonctions virtuelles pour la classe B et la classe C1.

10.2.2 Structures de la classe C1

```
00000000 CLASS_A         struc ; (sizeof=0x8)
00000000 vftable         dd ?  ; const A::vftable dd offset A::SetInt(int)
00000000                       ;                  dd offset A::GetInt(void)
00000004 m_iVal          dd ?
00000008 CLASS_A         ends

00000000 CLASS_B         struc ; (sizeof=0x1C)
00000000 vftable         dd ?  ; const B::vftable{for B} dd offset B::SetIntB(int)
00000000                       ;                         dd offset B::GetIntB(void)
00000004 vbtable         dd ?  ; const B::`vbtable' dd -4
00000004                       ;                    dd 10h
00000008 m_iVal2         dd ?
0000000C m_iVal3         dd ?
00000010 m_dwOffset      dd ?
00000014 oA              CLASS_A ?
          ; const B::`vftable'{for `A'} dd offset [thunk]:B::SetInt vtordisp{-4,0}(int)
00000014                       dd offset [thunk]:B::GetInt vtordisp{-4,0}(void)
0000001C CLASS_B         ends

00000000 CLASS_C         struc ; (sizeof=0x14)
00000000 vbtable         dd ?  ; const C::vbtable dd 0
00000000                       ;                  dd 0Ch
00000004 m_iVal          dd ?
```

```
00000008 dwOffset        dd ?
0000000C oA              CLASS_A ?
          ; const C::vftable dd offset A::SetInt(int)
0000000C  ;                 dd offset [thunk]:C::GetInt vtordisp{-4,0}(void)
00000014 CLASS_C         ends

00000000 CLASS_C1        struc ; (sizeof=0x28)
00000000 vftable_for_B   dd ?  ; const C1::vftable{for B} dd offset B::SetIntB(int)
00000000                       ;                          dd offset C1::GetIntB(void)
00000004 vbtable_for_B   dd ?  ; const C1::vbtable{for B} dd -4
00000004                       ;                          dd 1Ch
00000008 m_B_iVal2       dd ?
0000000C m_B_iVal3       dd ?
00000010 vbtable_for_C   dd ?  ; const C1::vbtable{for C} dd 0
00000010                       ;                          dd 10h
00000014 m_C_iVal        dd ?
00000018 m_iVal          dd ?
0000001C dwOffset        dd ?
00000020 oA              CLASS_A ?
          ; 1) Constructeur de B à partir de C1:
00000020  ; const B::vftable{for A} dd offset [thunk]:B::SetInt vtordisp{-4,0}(int)
00000020  ;                         dd offset [thunk]:B::GetInt vtordisp{-4,0}(void)
00000020  ;
00000020  ; 2) Constructeur de C à partir de C1:
00000020  ; const C::vftable dd offset A::SetInt(int)
00000020  ;                  dd offset [thunk]:C::GetInt vtordisp{-4,0}(void)
00000020  ;
00000020  ; 3) Constructeur de C1 à partir de C1:
00000020  ; const C1::vftable{for A} dd offset [thunk]:B::SetInt vtordisp{-4,12}(int)
00000020  ;                          dd offset [thunk]:C1::GetInt vtordisp{-4,0}(void)
00000028 CLASS_C1        ends
```

L'ajout de méthodes virtuelles dans la classe B provoque la création d'un nouveau champ dans la classe B (vftable) et C1 (vftable_for_B) en plus de la « vbtable ». Le comportement est ensuite identique au cas précédent.

10.2.3 Appels respectifs aux fonctions GetIntB() à partir d'un objet C1

10.2.3.1 Code C++

```
iVal2 += poC1->B::GetIntB();(1)
iVal2 += poC1->GetIntB();   (2)
```

10.2.3.2 Code généré

```
.text:004015C6                 mov     ecx, [ebp+poC1] ; this
.text:004015C9                 call    B::GetIntB(void)
.text:004015C9
.text:004015D4                 mov     ecx, [ebp+poC1]
.text:004015D7                 mov     edx, [ecx+CLASS_C1.vftable_for_B] ;
.text:004015D7               ; const C1::vftable{for B} dd offset B::SetIntB(int)
.text:004015D7               ;                          dd offset C1::GetIntB(void)
.text:004015D9                 mov     ecx, [ebp+poC1]
.text:004015DC                 mov     eax, [edx+4]
.text:004015DF                 call    eax  ; C1::GetIntB(void)
```

- Cas 1: L'instance de la classe C1 possède comme premier champ un objet de classe B, le pointeur sur la classe C1 est directement passé en tant que "this".
- Cas 2: On accède à la vftable pour la classe B à partir de la classe C1.

11 Conclusion

J'ai tâché de présenter les différents cas d'héritage C++ qu'il est possible de rencontrer et d'en expliquer le code généré en x86. Il reste certainement encore des cas exotiques.

Évidemment, un compilateur autre que celui utilisé dans ce document (Microsoft Visual Studio 8) générera un code différent. Il pourrait donc être intéressant de répéter l'exercice avec un autre compilateur tel que « g++ », voire même sur une architecture différente telle que du x64.

En tout cas, j'espère que cette approche vous a plu, vous a apporter de nouveaux éléments et vous sera peut-être utile...

Pour d'éventuelles questions, remarques, ou précisions sur le contenu de ce document, vous pouvez me contacter à l'adresse «trashomeo (at) gmail (dot) com».