Les pointeurs de fonctions

Ecrit par Lars Haendel
Traduit en francais par Francois Visconte (visconte.francois@wanadoo.fr)
Janvier 2002, Dortmund, Allemagne
http ://www.newty.de
version 2.042b

Soyéz conscient qu'il peut exister avoir une version plus récente de ce document ! Regardez sur http ://www.newty.de/fpt/zip/f_fpt.pdf pour obtenir la dernière version. Si vous voulez distribuer ce document, je vous suggère d'ajouter un lien vers l'addresse précédante pour eviter la distribution de version trop ancienes.

Vous pouvez aussi télécharger le code source des exemples sur http ://www.newty.de/fpt/zip/source.zip. Le code d'exemple est distrubué sous les termes de la licence GNU Général Public Licence.

Table des matières

1 Introduction aux pointeurs de fonction

Les pointeurs de fonction fournissent une technique de programmation extrêmement intéressante, efficace et élégante. On peut les utiliser pour remplacer les fonctions switch/if, pour réaliser nos propres "late-binding" ou pour implémenter les "Callback",fonctions de rappel. Malheureusement, sûrement a cause de leur complexité, ils sont souvent peu traités dans la plupart des livres et des documentations informatiques. Quand c'est le cas ils sont traités de manière assez brèves et superficielle. Il y a moins d'erreurs engendrées que par les pointeurs classiques parce que on n'alloue ou ne dessaoule jamais de la mémoire avec ces pointeurs. Il vaux mieux utiliser vos propres "late-binding" que d'utiliser les structures du C++ qui existent déjà, cela rend le code plus lisible et plus clair. Un aspect dans le cas des "late-binding" est l'aspect de routine (runtime) : si on appelle un fonction virtuelle, le programme doit déterminer quelle fonction doit être appeler. Cela ce fait grâce a une "V-Table" qui contient toutes les fonctions susceptibles d'être appelées. Ceci prend du temps a chaque fois que vous appelez une fonction, et, peut être peut on gagner du temps en appelant un pointeur de fonction plutôt qu'une fonction virtuelle. Peut être pas... [1]

1.1 Qu'est ce qu'un pointeur de fonction

Les pointeurs de fonction sont des pointeurs comme des pointeurs de variables, mais qui pointent sur l'adresse d'une fonction. On doit garder à l'esprit qu'un programme qui est exécuté prend un certain espace dans la mémoire principale. L'exécutable et les variables utilisés sont tous deux placés dans la mémoire principale. Ainsi une fonction dans le code du programme est comme, par exemple un champ de caractère, rien de plus qu'une adresse. La seule chose importante est comment l'on décrit, ou comment le compilateur décrit sur quelle zone mémoire pointe le pointeur.

1.2 Exemple introductif, ou comment remplacer la commande Switch

Quand vous voulez implémenter une fonction *DoIt()* à un certain point appelé *label* dans notre programme, on met juste l'appel à la fonction *DoIt()* au point *label* dans notre code source. Ensuite on compile notre code source et à chaque fois que le programme arrive au point *label*, notre fonction est appelée. Tout est bon. Mais que fait-on si l'on veut décider que c'est une routine ? Va-t-on utiliser quelque chose appelée une fonction de rappel, ou va-t-on sélectionner une fonction parmi une liste de fonctions possibles. Cependant, on peut résoudre ce problème en utilisant la fonction **switch**, ou l'on appelle la fonction quand on le veux et comme on veux l'appeler. Mais il existe une autre manière : utiliser les pointeurs de fonction.
Dans l'exemple suivant, on s'intéresse à la tache qui effectue l'une des quatre opérations arithmétiques basiques.[2]

```
//-----------------------------------------------------------------------
// 1.2 Exemple introductif : Comment remplacer la commande switch ?
// Tache: Réalise une des quatre opérations basiques spécifiées par
// les caractères
//  '+', '-' '*' ou '/'

// les quatre opérations arithmétiques
// une des ces fonctions est sélectionnée par une routine contenant
// l'une commande switch ou un pointeur de fonction
float Plus     (float a, float b) { return a+b; }
float Minus    (float a, float b) { return a-b; }
float Multiply (float a, float b) { return a*b; }
float Divide   (float a, float b) { return a/b; }

// solution utilisant la commande switch - <opCode> spécifie quel opération
// exécuter
void Switch(float a, float b, char opCode)
{
   float result;
```

[1] Les compilateurs modernes sont très bien ! avec mon compilateur Borland, le temps que j'ai pu gagner en appelant une fonction virtuelle qui multiplie deux nombres à virgule flottantes était à peut près de 2 %

[2] Ceci est seulement un exemple et la tache est tellement simple que je suppose que personne n'utilisera de pointeur de fonction pour faire cela.

```
   // exécute l'opération
   switch(opCode)
   {
      case '+' : result = Plus      (a, b); break;
      case '-' : result = Minus     (a, b); break;
      case '*' : result = Multiply  (a, b); break;
      case '/' : result = Divide    (a, b); break;
   }

   cout << "switch: 2+5=" << result << endl;          // Affiche le résultat
}

// solution avec les pointeurs de fonction -<pt2Func> est un pointeur
// de fonction qui pointe sur  une fonction qui prend deux nombres à
// virgule flottante en argument et qui retourne un nombre à virgule
// flottante.
// Le pointeur de fonction ``spécifie'' quelle opération exécuter.
void Switch_With_Function_Pointer(float a, float b, float (*pt2Func)(float, float))
{
   float result = pt2Func(a, b);

   cout << "switch replaced by function pointer: 2-5=";  // Affiche le resultat
   cout << result << endl;
}

// exécute le code d'exemple
void Replace_A_Switch()
{
   cout << endl << "Executing function 'Replace_A_Switch'" << endl;

   Switch(2, 5, /* '+' spécifie que la fonction 'Plus' vas être executé */ '+');
   Switch_With_Function_Pointer(2, 5, /* pointeur vers la fonction 'Minus' */ &Minus);
}
```

Note importante : Un pointeur de fonction pointe sur une fonction possedant une signature spécifique ! Ainsi, toute les fonctions que l'on voudra utiliser avec le même pointeur devront avoir les mêmes types de paramètres et les mêmes types de valeurs de retour.

2 La syntaxe des pointeurs de fonctions en C et C++

En ce qui concerne la syntaxe, il y a deux sortes de pointeurs de fonctions : d'un côté, il y a les pointeurs classiques qui pointent vers des fonction C ordinaires ou des fonctions membres C++ statique, d'un autre côté, il y a les pointeurs vers les les fonctions membres C++ non statiques. La différence basique est que tout les pointeurs qui pointent sur des membres non statiques, ont besoin d'un **argument caché** : Le pointeur **this**, qui pointe ver l'instance d'une classe. Gardons toujours cela à l'esprit : ces deux types de pointeurs de fonction sont incompatibles les un avec les autres.

2.1 Définir un pointeur de fonction

Si l'on considère qu'un pointeur de fonction n'est rien d'autre qu'une variable, il doit être défini comme une variable. Dans l'exemple qui suit on définit deux pointeurs de fonction appelés *pt2Function* et *pt2Menber*. Ils pointent sur une fonction qui a comme paramètres un nombre à virgule flottante (float) et deux caractères (char), et retourne un entier (int). Dans l'exemple en C++ on considère que la fonction vers laquelle pointe le pointeur est une fonction membre de TMyClass.

```
// 2.1 définition d'un pointeur de fonction
int (*pt2Function)        (float, char, char);           // C
int (TMyClass::*pt2Member)(float, char, char);           // C++
```

2.2 Conventions d'appel

Normalement nous n'avons pas a nous préoccuper des conventions d'appel de fonctions : Le compilateur attribue *__cdecl* par défaut si vous ne spécifiez pas une autre convention. Cependant, si vous voulez en savoir plus, lisez ce qui suit ... La convention d'appel de fonction décrit au compilateur plusieurs choses comme : comment passer les arguments, ou comment générer le nom d'une fonction. Quelques autres exemples de convention d'appel sont : *__stdcall*, *__pascal*et *__frastcall*. Les conventions d'appel font partie des signatures de fonction : **Ainsi, des fonctions avec des pointeurs de fonction différents sont incompatibles entre elles.** Pour les compilateurs Borland et MicroSoft on spécifie une convention d'appel spécifique entre le type de retour et le nom du pointeur de fonction. Pour le compilateur GNU GCC, on utilise le mot clef __attribute__ : On écrit la définition de fonction suivi du mot clef __attribute__, et ensuite de la convention d'appel entre deux parenthèses.[3]

```
// 2.2 définition des convention d'appel
void __cdecl DoIt(float a, char b, char c);  // Borland and Microsoft
void         DoIt(float a, char b, char c)  __attribute__((cdecl)); // GNU GCC
```

2.3 Assigner une adresse à un pointeur de fonction

C'est assez facile d'assigner l'adresse d'une fonction à un pointeur de fonction. On prend le nom d'une fonction membre adéquate et connue. Utiliser l'opérateur d'adressage & devant le nom de fonction est optionnel.
Note : On aura peut être à préciser le nom complet de la fonction dans le cas d'une fonction membre, le nom de la classe auquel elle appartient et l'opérateur (: :). On doit également s'assurer que l'on a autorisé l'accès au droit de la fonction duquel le pointeur dépend.

```
// 2.3 assigne une adresse au pointeur de fonction

// C
int DoIt  (float a, char b, char c){ printf("DoIt\n");   return a+b+c; }
int DoMore(float a, char b, char c){ printf("DoMore\n"); return a-b+c; }

pt2Function = DoMore;               // assigne-ment
pt2Function = &DoIt;                // utilisation alternative de l'opérateur d'adressage


// C++
class TMyClass
{
public:
    int DoIt  (float a, char b, char c){ cout << "TMyClass::DoIt"   << endl; return a+b+c; };
    int DoMore(float a, char b, char c){ cout << "TMyClass::DoMore" << endl; return a-b+c; };

   /* en plus de TMyClass */
};
pt2Member = TMyClass::DoIt;         // assigne-ment
pt2Member = &TMyClass::DoMore;      // utilisation alternative de l'opérateur d'adressage
```

2.4 Comparer des pointeurs de fonction

On peut utiliser l'opérateur de comparaison (==) de la même façon. Dans l'exemple qui suit ceci est vérifié, *pt2Function* et et *pt2Member* contiennent l'adresse de la fonction DoIt et TMyClass : :DoMore. Un texte est affiché dans le cas d'une égalité.

[3]Si quelqu'un en sait plus à ce sujet, faites le moi savoir. Et si vous voulez savoir comment les appels de fonctions fonctionnent en détail vous devriez jeter un oeil sur le chapitre *Sous Programmes* dans le livre de Paul Carter : "PC Assembly Tutorial".

```
// 2.4 comparaison de pointeurs de fonction

// C
if(pt2Function == &DoIt)
   printf("pointer points to DoIt\n");

// C++
if(pt2Member == &TMyClass::DoMore)
   cout << "pointer points to TMyClass::DoMore" << endl;
```

2.5 Appeler une fonction en utilisant un pointeur de fonction

En C nous avons deux alternatives pour appeler une fonction en utilisant un pointeur de fonction : Nous pouvons simplement utiliser le nom du pointeur de fonction au lieu du nom de fonction, ou l'on peut le déférencer explicitement. En C++, ceci est légèrement plus compliqué puisque l'on doit avoir une instance de classe à appeler pour appeler une de ces fonction membres non statique. Si l'appel est effectué avec une autre fonction membre on peut utiliser le pointeur **this**.

```
// 2.5 calling a function using a function pointer
int result1 = pt2Function    (12, 'a', 'b');          // C maniere courte
int result2 = (*pt2Function) (12, 'a', 'b');          // C

TMyClass instance;
int result3 = (instance.*pt2Member)(12, 'a', 'b');    // C++
int result4 = (*this.*pt2Member)(12, 'a', 'b');       // C++ Si le pointeur "this"
                                                      // peut etre utilisé
```

2.6 Comment passer un pointeur de fonction en argument ?

On peut passer un pointeur de fonction en argument d'une fonction appelé. On a besoin de cette méthode, par exemple, pour passer un pointeur a une fonction de callback. L'exemple suivant montre comment passer un pointeur de fonction qui prend un nombre à virgule flottante et deux caractère en tant que paramètres.

```
//-----------------------------------------------------------------------------------------
// 2.6 Comment passer un pointeur de fonction en argument

// <pt2Func> est un pointeur de fonction qui retourne un entier et
// qui prend un nombres a virgule flottante et deux caractères en argument
void PassPtr(int (*pt2Func)(float, char, char))
{
   float result = pt2Func(12, 'a', 'b');     // appel utilisant les pointeurs de fonctions
   cout << result << endl;
}

// execute le code d'exemple - 'DolIt est une fonction qui
// convient comme définit plus haut
void Pass_A_Function_Pointer()
{
   cout << endl << "Executing 'Pass_A_Function_Pointer'" << endl;
   PassPtr(&DoIt);
}
```

2.7 Comment retourner un pointeur de fonction ?

C'est un peut plus compliqué mais un pointeur de fonction peut être la valeur de retour d'une fonction. Dans l'exemple qui suit il y a deux solutions pour passer un pointeur de fonction en valeur de retour d'une fonction qui prend un nombre à virgule flottante et deux caractères en paramètres. Si l'on veut retourner un pointeur qui pointe vers une fonction membre il faut simplement changer la définition/déclaration de tout les pointeurs de fonction.

```
//-----------------------------------------------------------------------------------------
// 2.8 Comment utiliser des tableaux de pointeurs de fonction

// C
// définition de types : 'pt2Function' peut être utilisé comme type
typedef int (*pt2Function)(float, char, char);

// illuste comment utiliser un tableau de pointeurs de fonction
void Array_Of_Function_Pointers()
{
   printf("Executing 'Array_Of_Function_Pointers'\n");

   // <funcArr> est un tableau de 10 pointeurs de fonction qui
   // retourne un entier et prend un nombre a virgule flottante
   // et deux caractères en arguments
   pt2Function funcArr[10];

   // assigne l'adresse de la fonction  - 'DoIt' et 'DoMore' sont
   // des fonction qui conviennent comme définit plus haut (2.1-4)
   funcArr[0] = &DoIt;
   funcArr[1] = &DoMore;
   /* encore des assignation */

   // appelle d'une fonction en utilisant un index pour adresser
   // le pointeur de fonction
   printf("%d\n", funcArr[1](12, 'a', 'b'));
   printf("%d\n", funcArr[0](12, 'a', 'b'));
}


// C++
// définition de type: 'pt2Member' peut être utilisé en tant que type
typedef int (TMyClass::*pt2Member)(float, char, char);

// illustre comment travailler avec un tableau de pointeurs ver
// des fonctions membres
void Array_Of_Member_Function_Pointers()
{
   cout << endl << "Executing 'Array_Of_Member_Function_Pointers'" << endl;

   // <funcArr> est un tableau de dix pointeurs vers des fonctions
   // membres qui retournent un entier et prennent comme argument
   // un nombre à virgule flottante et deux caractères
   pt2Member funcArr[10];

   // assigne l'adresse de la fonction  - 'DoIt' et 'DoMore' sont des
   // fonctions membre qui conviennent de la classe TMyClass comme
   // définit précédemment (2.1-4)
   funcArr[0] = &TMyClass::DoIt;
   funcArr[1] = &TMyClass::DoMore;
   // encore des assignation

   // appelle d'une fonction en utilisant un index pour adresser le
   // pointeur vers la fonction membre
   // note: une instance de TMyClass est requise pour appeler la fonction membre
   TMyClass instance;
```

```
    cout << (instance.*funcArr[1])(12, 'a', 'b') << endl;
    cout << (instance.*funcArr[0])(12, 'a', 'b') << endl;
}
```

3 Comment implémenter les fonctions de rappel en C et C++

3.1 Introduction au concept de fonction de rappel

Les pointeurs de fonction permettent de mettre en oeuvre les fonction de rappel. Si vous ne savez pas vraiment comment utiliser les pointeurs de fonctions jetez un oeil sur "Introduction aux pointeurs de fonctions". Je vais essayer d'introduire le concept de fonction de rappel utilisant un algorithme de tri bien connue, l'algorithme de tri rapide. Cette fonction tri les positions d'un champ de n'importe quel type ; ce champ est passé a la fonction de trie en utilisant un pointeur de type *void* la taille d'une position et le nombre total de positions dans le champ doivent également être passé à la fonction de tri.**La question est la suivante : Comment la fonction de tri peut elle effectuer le tri sans aucune information concernant le type d'une position ?** La réponse est simple : la fonction reçoit le pointeur d'une fonction de comparaison qui prend comme argument un pointeur *void* vers deux positions du champ, évalue leur rang puis retourne le résultat re-typé en tant qu'un entier. Ainsi, a chaque fois que l'algorithme de tri a besoin d'une decision à propos du rang de deux positions, le programme appelle juste la fonction de comparaison via le pointeur de fonction : ceci est un callback.

3.2 Comment implémenter une fonction de rappel en C ?

Ceci prend juste la déclaration qsort qui lit elle-même comme ceci.[4]

```
void qsort(void* field, size_t nElements, size_t sizeOfAnElement,
                 int(_USERENTRY *cmpFunc)(const void*, const void*));
```

field pointe vers le premier élément du champ qui doit être trié,*nElements* est le nombre d'éléments dans le champ, *sizeOfAnElement* est la taille d'un seul élément en octets et ***cmpFunc est le pointeur vers la fonction de comparaison***. Cette fonction de comparaison prend comme arguments deux pointeurs de type *void* et retourne un entier. La syntaxe utilisée pour utiliser pour définir une fonction qui prend un pointeur de fonction comme paramètre semble un peut étrange. Si ceci vous pose des difficultés regardez : "Comment définir un pointeur de fonction". Une fonction de rappel a été construit, exactement comme un appel de fonction pourrait être effectué : on utilise juste le nom du pointeur de fonction à la place du nom de la fonction. Ceci est montré ici.
Note : Tous les arguments d'appel à part le pointeur de fonction ont été omis pour s'intéresser qu'aux choses qui ont de l'intérêt.

```
void qsort( ... , int(_USERENTRY *cmpFunc)(const void*, const void*))
{
   /* algorythme de trie  - note: item1 et item2 sont des pointeurs void */

   int bigger=cmpFunc(item1, item2);  // reallize le rappelle de fonction

   /* utilisation du résultat */
}
```

3.3 Code d'exemple de l'utilisation de *qsort*

Dans l'exemple suivant un champ de nombres à virgule flottante est trié.

```
//-----------------------------------------------------------------------------
// 3.3 Comment réaliser une fonction de rappel en C en utilisant la fonction qsort
```

[4]Cet exemple a ete testé avec le compilateur Borland C++ 5.02 (BC5.02)

```
#include <stdlib.h>        // utilise pour :  qsort
#include <time.h>          //                 randomize
#include <stdio.h>         //                 printf

// fonction de comparaison pour l'algorithme de tri
// deux champ sont pris par le pointeur void, convertis puis comparés
int CmpFunc(const void* _a, const void* _b)
{
   // you've got to explicitly cast to the correct type
   const float* a = (const float*) _a;
   const float* b = (const float*) _b;

   if(*a > *b) return 1;             // le premier champ est plus grand que le second
   else
      if(*a == *b) return  0;        // egualité -> return 0
      else         return -1;        // le second champ est plus grand que le
                                     // premier -> return -1
}

// example for the use of qsort()
void QSortExample()
{
   float field[100];

   ::randomize();                    // initialise le generateur de nombre aleatoires
   for(int c=0;c<100;c++)            // choisis au hazard tous les elements du tableau
      field[c]=random(99);

   // trie en utilisant qsort()
   qsort((void*) field, /*nombre de champ */ 100, /*taille d'un champ*/ sizeof(field[0]),
         /*fonction de comparaison*/ CmpFunc);

   // affiche le premier des 10 element du tableau tri
   printf("The first ten elements of the sorted field are ...\n");
   for(int c=0;c<10;c++)
      printf("element #%d contains %.0f\n", c+1, field[c]);
   printf("\n");
}
```

3.4 Comment implementer les fonction de rappel a une fonction membre statique ?

Cela ce fait de la même manière qu'avec les fonctions C. Les fonctions membres C++ statique n'ont pas besoin d'un objet pour être invoqué et ont donc la même signature qu'une fonction C, avec les mêmes conventions d'appel, les même arguments d'appel et mêmes types de retour.

3.5 Comment implémenter une fonction de rappel a une fonction menbre C++ non statique ?

L'approche du "Wrapper"

Les pointeurs vers des membres non statiques sont différents des pointeurs de fonctions C classiques, ils ont besoin du pointeur *this* de la classe objet pour être passé. Ainsi les pointeurs de fonctions ordinaires ou de fonctions membres non statiques ont une signature différente et incompatible ! Si l'on veut juste créer une fonction de rappel vers un membre d'une classe spécifique il suffit de changer le code utilisé pour une fonction ordinaire pour l'adapter a un pointeur de fonction membre. Mais que fait on si l'on veut créer une fonction de rappel vers un membre d'une classe non statique arbitraire ? C'est un petit peu compliqué. On doit écrire un membre **statique** en tant que "Wrapper". Un fonction membre statique a la même signature qu'une fonction

C classique ! On doit donc "caster" les pointeurs vers l'objet sur celui que l'on veut pour invoquer la fonction membre comme **void*** et le passer à la fonction "Wrapper" en tant que argument additionnell ou via une variable globale. [5] Bien sur on doit aussi passer les arguments d'appel pour la fonction membre. Le "Wrapper" "cast" le pointeur *void* sur un pointeur vers une instance de la classe correspondante et appelle la fonction membre. Vous trouverez, ci dessous deux exemples.

Exemple A : Le pointeur de l'instance de classe est passé en tant qu'argument additionnel La fonction **DoItA** fait quelque chose avec les objets de la classe TClassA, ce qui implique une fonction de rappel. Pour cela un pointeur vers l'objet de la class TClassA et un pointeur vers la fonction "Wrapper" statique : **TClassA : :Wrapper_To_Call_Display** sont passés a la fonction DoItA. Ce "Wrapper" est la fonction de rappel. On peut écrire arbitrairement d'autres classes comme TClassA et les utiliser avec DoItA tant que ces classes fournissent les fonctions nécessaires.

Note : Cette solution peut être très utile si l'on crée une fonction de rappel nous même. Cela se révèle mieux que la seconde solution qui utilise une variable globale.

```
//-----------------------------------------------------------------------
// 3.5 Example A: Fonction de rappel vers une fonction membre
//                en utilisant un argument additionnel
// Tas: la fonction 'DoItA' fait quelque chose qui implique
//      l'utilisation d'une fonction de rappel ver la fonction
//      membre 'Display'. Pour réaliser cela, la fonction ''wrapper''
//      'Wrapper_To_Call_Display' est utilisé

#include <iostream.h>   // pour l'utilisation de cout

class TClassA
{
public:

   void Display(const char* text) { cout << text << endl; };
   static void Wrapper_To_Call_Display(void* pt2Object, char* text);

   /* suite de TClassA */
};

// fonction ''wrapper'' statique pour pouver rappeler la fonction menbre Display()
void TClassA::Wrapper_To_Call_Display(void* pt2Object, char* string)
{
    // Cast explicitement sur un pointeur TClassA
    TClassA* mySelf = (TClassA*) pt2Object;

    // call member
    mySelf->Display(string);
}

// la fonction fait quelquechose qui implique l'utilisation d'un callback
// note: bien sur, cette fonction peut également être une fonction membre
void DoItA(void* pt2Object, void (*pt2Function)(void* pt2Object, char* text))
{
   /* fait quelque chose */
   pt2Function(pt2Object, "hi, i'm calling back using an argument ;-)");  // fonction de rappel
}

// execute le code d'exemple
void Callback_Using_Argument()
{
   // 1. instancie l'objet de TClassA
```

[5] Si l'on utilise une variable globale il est très important de s'assurer qu'il pointera toujours vers le bon objet !

```
   TClassA objA;

   // 2. appelle 'DoItA' pour <objA>
   DoItA((void*) &objA, TClassA::Wrapper_To_Call_Display);
}
```

Exemple B : Pointeur vers l'instance de classe utilise une variable globale La fonction **DoItB** fait quelque chose avec les objets de la classe TClassB ce qui implique une fonction de rappel. Un pointeur vers la fonction statique **TClassB : :Wrapper_To_Call_Display** est passé à DoItB. Ce "Wrapper" est la fonction de rappel. Ce "Warpper" utilise la **variable globale** *void *pt2Object* et le "cast" explicitement dans une instance de TClassB. Il est très important de toujours initialiser la variable globale pour qu'elle pointe sur la bonne instance de classe. On peut arbitrairement créer d'autres classes comme TClassB et les utiliser avec DoItB tant que ces classes fournissent les fonctions correctes.

Note : Cette solution est utile quand l'on a une interface de fonction de rappel qui existe déjà et qui ne peut pas être changé. Autrement, ce n'est pas une très bonne solution puisque l'utilisation d'une variable globale est très dangereux et peut causer des erreurs.

```
//-----------------------------------------------------------------------------------------
// 3.5 Example B: Fonction de rappel vers une fonction membre en utilisant une variable
// globale
// Task: La fonction 'DoItB' fait quelque chose qui implique une fonction de rappel vers
//       la fonction membre 'D'isolat'. Pour ce faire la fonction membre
//       'Wrapper_To_Call_Display est utilisée

#include <iostream.h>   // utilise pour:   cout

void* pt2Object;        // variable globale qui pointe sur un objet arbitraire

class TClassB
{
public:

   void Display(const char* text) { cout << text << endl; };
   static void Wrapper_To_Call_Display(char* text);

   /* suite de  TClassB */
};

// fonction ``wrapper'' statique pour pouvoir rappeler la fonction membre Display()
void TClassB::Wrapper_To_Call_Display(char* string)
{
    // Cast explicitement les variables globales <pt2Object> sur un pointer vers TClassB
    // Attention: <pt2Object> DOIT pointer sur un objet approprie
    TClassB* mySelf = (TClassB*) pt2Object;

    // appel de la fonction membre
    mySelf->Display(string);
}

// La fonction realise une opération qui implique l'utilisation d'une fonction de rappel
// note: bien sur la fonction peut aussi être une fonction membre
void DoItB(void (*pt2Function)(char* text))
{
   /* fait quelque chose*/
   pt2Function("hi, i'm calling back using a global ;-)");   // make callback
}
```

```
// execute le code d'exemple
void Callback_Using_Global()
{
   // 1. instancie l'objet de TClassB
   TClassB objB;

   // 2. assigne la variable globale qui est utilisée par la fonction wapper statique
   // important: ne jamais oublier de faire ceci!!!
   pt2Object = (void*) &objB;

   // 3. appel 'DoItB' pour <objB>
   DoItB(TClassB::Wrapper_To_Call_Display);
}
```

4 Utilisation des "Functors" pout encapsuller les pointeurs de fonctions C et C++

4.1 Qu'est ce qu'un "Functor" ?

Les functors sont des *fonctions a etat.* En C++ l'on peut en réaliser en tant que classe avec plus d'un membre privé pour stocker l'état et un opérateur surchargé () pour exécuter la fonction. Les "functors" peuvent encapsuler des pointeurs de fonction C et C++ en utilisant le principe des classes et du polymorphisme. On peut les construire grâce à une liste de pointeurs vers les fonctions membres de classes arbitraires et toutes les appeler à travers la même interface sans se préoccuper de leurs classes respectives ou de la nécessité d'un pointeur vers une instance. Toutes les fonctions ont juste à avoir le même typage de valeurs de retour et le même typage des arguments d'appel. Quelquefois les "Functors" sont aussi connus sous le nom de "Closures". On peut aussi utiliser les "functors" pour implémenter les fonctions de rappel.

4.2 Comment implémenter les "Functors" ?

Tout d'abord, nous avons besoin d'une classe de base **TFunctor** qui fournit une fonction virtuelle nommée *Call* ou bien un opérateur virtuel surcharge () avec lequel l'on peut appeler la fonction menbre. L'on peut choisir indifféremment l'opérateur surchargé ou une fonction comme *Call*. De la classe de base on dérive une classe **Template TSpecificFunctor** qui est initialisé avec un pointeur vers un objet et un pointeur vers une fonction membre de son constructeur.**La classe dérive passe outre la fonction *Call* et/ou l'opérateur () de la classe de base :** dans la version passée outre, on appelle la fonction membre en utilisant le pointeur stocké qui pointe vers l'objet et vers la fonction membre. Si vous n'êtes pas sur de savoir comment utiliser les pointeurs de fonction jetez un oeil sur la partie "Introduction aux pointeurs de fonctions".

```
//-----------------------------------------------------------------------------------------
// 4.2 Comment implémenter les ‘‘Functors’’

// extraie la classe de base
class TFunctor
{
public:

   // deux fonctions possibles pour appeler la fonction membre.
   // Virtuelle car les classes dérivées devront utiliser un
   // pointeur sur un objet et un pointeur sur une fonction membre
   // pour réaliser l'appel de la fonction
   virtual void operator()(const char* string)=0;  // appelle utilisant une operateur
   virtual void Call(const char* string)=0;        // appelle utilisant la fonction
};

// classe template dérive
```

```
template <class TClass> class TSpecificFunctor : public TFunctor
{
private:
   void (TClass::*fpt)(const char*);   // pointeur vers une fonction membre
   TClass* pt2Object;                  // pointeur vers un objet

public:

   // constructeur - prend un pointeur vers un objet et un pointeur ver une
   // fonction membre et les stokes dans deux variables privés
   TSpecificFunctor(TClass* _pt2Object, void(TClass::*_fpt)(const char*))
      { pt2Object = _pt2Object;  fpt=_fpt; };

   // surcharge l'opérateur "()"
   virtual void operator()(const char* string)
    { (*pt2Object.*fpt)(string);};             // execute la fonction menbre

   // surcharge la  fonction "Call"
   virtual void Call(const char* string)
     { (*pt2Object.*fpt)(string);};            // execute la fonction menbre
};
```

4.3 Exemple d'utilisation des "Functors"

Dans l'exemple suivant nous avons deux classes tout à fait banales qui fournissent une fonction appelé *Display* qui ne retourne aucune valeur (*void*) et à besoin, comme argument d'une chaîne de caractères (*const char **). On crée un tableau de deux pointeurs vers **TFunctor** et on initialise les champs du tableau avec deux pointeurs vers **TSpecificFunctor** qui en-capsule le pointeur vers l'objet et le pointeur vers, respectivement, un membre de *TClassA* et un menbre de *TClassB*. Ensuite on utilise le tableau "Functor" pour appeler les fonctions membres respectives. **Aucun pointeur vers un objet n'est nécessaire pour appeler la fonction, et l'on ne se soucie pas de la classe**

```
//-----------------------------------------------------------------------------------------
// 4.3 Exemple d'utilisation des  ‘‘Functors’’

// classe A
class TClassA{
public:

   TClassA(){};
   void Display(const char* text) { cout << text << endl; };

   /* suite de TClassA */
};

// classe B
class TClassB{
public:

   TClassB(){};
   void Display(const char* text) { cout << text << endl; };

   /* suite de TClassB */
};

// programme principal
int main(int argc, char* argv[])
{
   // 1. objets instanciés de TClassA et TClassB
```

```
    TClassA objA;
    TClassB objB;


    // 2. instancie l'objet TSpecificFunctor ...
    //    a ) functor qui en-capsule les pointeur vers l'objet et vers
    //        le menbre de TClassA
    TSpecificFunctor<TClassA> specFuncA(&objA, TClassA::Display);

    //    b) functor qui encapsule les pointeurs vers l'objet et vers
    //       le membre de TClassB
    TSpecificFunctor<TClassB> specFuncB(&objB, &TClassB::Display);

    // 3. crée un tableau avec les pointeurs vers TFunctor,
    // la classe de base et ....
    TFunctor** vTable = new TFunctor*[2];

    // ... assigne l'adresse du functor a l'adresse du tableau de
    // pointeurs de fonction
    vTable[0] = &specFuncA;
    vTable[1] = &specFuncB;

    // 4. utilisa le tableau pour appeler les fonctions menbres sans utiliser un objet
    vTable[0]->Call("TClassA::Display called!");        // via la fonciton "Call"
    (*vTable[1])   ("TClassB::Display called!");        // via l'opérateur "()"

    // 5. délivre
    delete[] vTable;

    // Appuyer sur enter pour finir
    cout << endl << "Hit Enter to terminate!" << endl;
    cin.get();

    return 0;
}
```