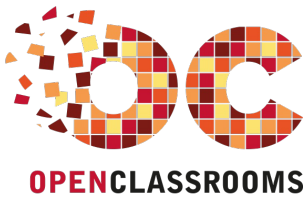
[C++] Les pointeurs sur fonctions

Par Matthieu Schaller (Nanoc)



www.openclassrooms.com

Dernière mise à jour le 27/06/2011

Sommaire

[C++] Les pointeurs sur fonctions

Par  Matthieu Schaller (Nanoc)

Mise à jour : 27/06/2011
Difficulté : Facile

Si vous vous souvenez de votre début en C, un des premiers passages délicats a sûrement été celui de l'utilisation des pointeurs. Car la notion n'est pas toujours très intuitive. En C++, les pointeurs ont perdu un peu de leur utilité suite à l'introduction des références. Cependant, il est un domaine où les pointeurs sont irremplaçables, ce sont les pointeurs sur des fonctions.

Dans ce chapitre, je vais vous apprendre à créer et à utiliser des pointeurs sur des fonctions. Dans une deuxième partie, je vous montrerai quelles difficultés apparaissent quand on utilise des classes.

Dans le C++ moderne, les pointeurs de fonctions sont de moins en moins utilisés au profit des *foncteurs* (que vous pouvez découvrir dans le tutoriel officiel) mais comme de nombreuses bibliothèques venues du C utilisent des pointeurs de fonctions, il est parfois nécessaire de savoir les utiliser.
Sommaire du tutoriel :

Choisir une fonction

Pour vous présenter concrètement l'utilité des pointeurs sur fonctions, je vais passer par un exemple.

Le minimum d'une fonction

Imaginons que vous ayez envie d'écrire une fonction (informatique) permettant de calculer le minimum d'une fonction (mathématique) $f(x)$ donnée sur un intervalle $[a;b]$. C'est une fonctionnalité proposée par les calculatrices scientifiques un peu avancées, il est donc tout à fait légitime d'essayer d'implémenter ce genre de choses. Si l'on ne connaît pas les pointeurs sur une fonction, on serait tenté d'écrire quelque chose comme ceci:

Code : C++ - Minimum d'une fonction

```
#include <cmath>
using namespace std;

//Liste des fonctions "calculables"

//double f(double x) { return x*x;}
double f(double x) { return 1/x;}
//double f(double x) { return sqrt(x);}
//double f(double x) { return exp(x);}
//...


double minimum(double a,double b)
{
   double min(100000);
```

```
    for(double x=a; x<b; x+= 0.01)  //On parcourt l'intervalle
      min = min< f(x)? min : f(x);  //Et si la nouvelle valeur est
plus petite que l'ancienne
                                    //On la garde
    return min;
}

//..
```

L'algorithme de calcul du minimum n'est pas bien malin, mais ce n'est pas ce qui nous préoccupe ici.

Chaque fois que l'on voudrait calculer le minimum d'une autre fonction, il faudrait dé-commenter la ligne concernée et recompiler le programme. Ceci n'est évidemment pas satisfaisant, il serait beaucoup mieux de pouvoir passer un argument à la fonction permettant de savoir de quelle fonction $f(x)$, elle doit chercher le minimum.

Utiliser un type énuméré ?

Il faudrait donc transmettre un élément supplémentaire à la fonction `minimum`. La meilleure chose à transmettre serait donc une sorte d'indice unique et la fonction pourra ainsi savoir quelle fonction mathématique utiliser. Une solution intelligente consisterait à utiliser un type énuméré et à placer un switch dans la fonction `minimum`, par exemple:

Code : C++ - Type énuméré

```
//Type énuméré représentant les fonctions calculables
enum Fonctions{CARRE,INVERSE,RACINE,EXPONENTIELLE};

//Liste des fonctions "calculables"

double carre(double x) { return x*x;}
double inverse(double x) { return 1/x;}
double racine(double x) { return sqrt(x);}
double exponentielle(double x) { return exp(x);}

double minimum(double a,double b,Fonctions fonction_choisie)
{
double min(100000);

    switch(fonction_choisie)
    {
    case CARRE:
          for(double x=a; x<b; x+= 0.01)  //On parcourt l'intervalle
             min = min< carre(x)? min : carre(x);
          break;
    case INVERSE:
         for(double x=a; x<b; x+= 0.01)  //On parcourt l'intervalle
             min = min< inverse(x)? min : inverse(x);
         break;
    //...
    };

    return min;
}

//..
```

Ce code est mieux que le précédent, il n'y a plus besoin de recompiler à chaque fois que l'on veut changer la fonction à évaluer. Cependant, ce n'est toujours pas très pratique. Si l'on veut ajouter une nouvelle fonction, il faut modifier l'`enum` et surtout, il faut

modifier le code de la fonction `minimum`, ce qui évidemment une très mauvaise chose. On ne devrait pas avoir à modifier la fonction `minimum` à chaque fois qu'on ajoute une nouvelle fonction mathématique.

La meilleure chose serait de pouvoir donner directement à `minimum`, la fonction que l'on souhaite évaluer. Et si je vous en parle, c'est que c'est faisable (vous en doutiez ?) et même de manière élégante.

Les pointeurs sur fonctions

Comme une fonction n'est pas un objet ou une variable, il n'est pas possible de passer une fonction directement en argument à notre fonction `minimum`. Par contre, comme toutes les choses présentes dans un programme C++, les fonctions ont une adresse. Il est donc possible de déclarer un pointeur vers cette adresse et de passer ce pointeur à la fonction `minimum`.

Déclarer un pointeur sur fonction

Bon puisque vous êtes encore là, allons-y, déclarons un pointeur sur une fonction. La syntaxe est la suivante:

```
type_de_retour (*monPointeur) (type_argument_1,type_argument_2,...)
```

Décortiquons les 3 parties de la déclaration:

La première partie (en rouge) indique le type de retour de la fonction pointée, cet élément fait donc partie du "type" d'une fonction. Le "type" d'une fonction n'est donc pas uniquement constitué de la signature de la fonction.

La deuxième partie (en vert) est le nom que vous souhaitez donner à votre pointeur. Les parenthèses sont nécessaires afin que le compilateur sache que l'étoile est liée au nom et pas au type de retour.

La troisième partie (en bleu) consiste en une liste des types des arguments que la fonction pointée doit recevoir.

Maintenant que nous somme munis de ces quelques notions, déclarons quelques pointeurs:

Code : C++ - Quelques pointeurs sur fonctions

```
int (*pointeur_1)(int);
// Déclaration d'un pointeur nommé "pointeur_1" qui pourra pointer
// sur des fonctions recevant un int et renvoyant un int.

int (*pointeur_2)(int,double);
// Déclaration d'un pointeur nommé "pointeur_2" qui pourra pointer
// sur des fonctions recevant un int et un double et renvoyant un
int.

void (*pointeur_3)(double);
// Déclaration d'un pointeur nommé "pointeur_3" qui pourra pointer
// sur des fonctions recevant un double et ne renvoyant rien.

void (*pointeur_4)();
// Déclaration d'un pointeur nommé "pointeur_4" qui pourra pointer
// sur des fonctions ne recevant rien et ne renvoyant rien non
plus.
```

`int (*ptr)(int,double)` n'est pas équivalent à `int (*ptr)(double,int)`, l'ordre des arguments joue un rôle. De même `void (*ptr)(double)` n'est pas équivalent à `void (*ptr)(int)`, même si les transformations de `int` en `double` sont automatiques.

Un pointeur sur fonction est un type comme un autre. Vous pouvez donc tout à fait créer un tableau de pointeurs sur fonctions, un std::vector<> de pointeurs sur fonctions ou même un pointeur sur pointeur de fonction.

Bon c'est bien joli tout ça, mais notre pointeur pour le moment ne pointe sur rien (ou en tout cas pas sur une fonction que vous

avez créée), voyons donc comment l'affecter.

Affecter un pointeur sur fonction

Comme on parle de pointeurs sur fonctions, vous pourriez tout à fait vous dire que c'est très facile de l'affecter, il suffit de récupérer l'adresse mémoire d'une fonction et de la mettre dans le pointeur. Et vous auriez tout à fait raison ! Il est tout à fait possible d'utiliser l'opérateur **&** pour récupérer l'adresse d'une fonction. Par exemple :

Code : C++ - Affectation d'un pointeur

```
#include <string>
using namespace std;

int fonction(double a,string phrase)  //Une jolie fonction
{
    //blablabla
}

int main()
{
   int (*monPointeur)(double,string);   //On déclare un pointeur sur
fonction

   monPointeur = &fonction;   //Et on le fait pointer sur "fonction"

   //...
}
```

Ce code est tout à fait correct, cependant les créateurs du C++ ont voulu simplifier ceci et ont décidé qu'il n'était pas nécessaire d'utiliser l'opérateur **&**. Le code suivant revient donc au même:

Code : C++ - Affectation d'un pointeur par la bonne méthode

```
#include <string>
using namespace std;

int fonction(double a,string phrase)  //Une jolie fonction
{
    //blablabla
}

int main()
{
   int (*monPointeur)(double,string);   //On déclare un pointeur sur
fonction

   monPointeur = fonction;   //Et on le fait pointer sur "fonction"
   // Notez l'absence du '&' !!

   //...
}
```

Et c'est cette dernière méthode qui est utilisée par tous les programmeurs. Il n'est pas faux d'utiliser le **&**, mais personne ne le fait. Puisque la notation est assez explicite sans, il n'est pas nécessaire pour le compilateur et pour les programmeurs d'ajouter le &.

Il est tout à fait possible d'affecter un pointeur directement à l'initialisation. Le code serait alors:
`void (*pointeur)(int) = fonction;`

Utiliser la fonction pointée

A nouveau, comme l'on parle de pointeurs, vous pouvez vous douter que l'on va utiliser l'opérateur *. Et puisque vous êtes des lecteurs attentifs, vous pourriez penser qu'il existe à nouveau une convention permettant d'omettre le *. Et je ne pourrais que vous donner raison. Voyons cela sur un exemple:

Code : C++ - Utiliser une fonction pointée

```
int maximum(int a,int b)  //Retourne le plus grand de deux entiers
{
    return a>b ? a : b;
}

int main()
{
   int (*ptr) (int,int); //Un joli pointeur

   ptr = maximum;  //que l'on affecte à la fonction "maximum"

   int resultat = (*ptr)(1,2); //On calcule le maximum de 1 et 2
via la fonction pointée
   //Notez l'utilisation obligatoire des ()

   int resultat_2 = ptr(3,4); //Et on fait la même chose pour 3 et
4
   //Notez l'absence de *
}
```

Les lignes 12 et 15 sont tout à fait équivalentes au niveau du programme généré. Mais comme précédemment, personne n'utilise la version avec *. On peut donc déduire la règle suivante:

On utilise un pointeur sur fonction de la même manière qu'on utilise la fonction pointée.

Récrire le code de l'exemple initial

Maintenant que nous maitrisons un nouvel outil, nous pouvons récrire le code du premier exemple de la manière suivante:

Code : C++ - La bonne manière de faire

```
#include <iostream>
#include <cmath>
using namespace std;

//Un petit typedef pour simplifier la notation
typedef double(*Fonction)(double);

//Liste des fonctions "calculables"
double carre(double x) { return x*x;}
double inverse(double x) { return 1/x;}
double racine(double x) { return sqrt(x);}
double exponentielle(double x) { return exp(x);}

double minimum(double a,double b,Fonction f)  //On passe le pointeur
en argument
{
   //Et on reprend le code du tout premier exemple
   double min(100000);

   for(double x=a; x<b ; x+= 0.01)
     min = min< f(x)? min : f(x);
   //Mais cette fois c'est la fonction pointée qui est utilisée
```

```
    return min;
}

int main()
{
   cout << "De quelle fonction voulez-vous chercher le minimum ?" <<
endl;
   cout << "1 -- x^2" << endl;
   cout << "2 -- 1/x" << endl;
   cout << "3 -- racine de x" << endl;
   cout << "4 -- exponentielle de x" << endl;
   cout << "5 -- sinus de x" << endl;

   int reponse;
   cin >> reponse;

   Fonction monPointeur; //On declare un pointeur sur fonction

   switch(reponse){  //Et on déplace le pointeur sur la fonction
choisie
      case 1:  monPointeur = carre; break;
      case 2:  monPointeur = inverse;  break;
      case 3:  monPointeur = racine;  break;
      case 4:  monPointeur = exponentielle;  break;
      case 5:  monPointeur = sin; break; //On peut même utiliser les
fonctions de cmath !
   }

   //Finalement on affiche le résultat de l'appel de la fonction
via le pointeur
   cout << "Le minimum de la fonction entre 3 et 4 est: " <<
minimum(3,4,monPointeur) <<endl;

   return 0;
}
```

C'est certainement la meilleure manière de réaliser ce que l'on voulait faire.

La fonction main n'est pas du tout sécurisée au niveau des entrées et du switch, mais ce n'est pas le but ici.

Le cas particulier des fonctions membres

Bon maintenant que nous avons vu la base, il faut quand même que je vous parle d'une erreur classique dans ce domaine. Imaginons que vous ayez écrit une classe et que vous aimeriez déclarer un pointeur sur une de ses fonctions membres.

Prenons donc un code classique (modifié) venant du forum:

Code : C++ - L'erreur classique

```
class A{
    public:
       int fonction(int a);
    //...
};

int A::fonction(int a){return a*a;}

int main()
{
   int(*ptr)(int) = fonction;     //aïe

   int(*ptr)(int) = A::fonction;  //re-aïe mais déjà mieux

   int (A::*ptr)(int) = &A::fonction;  //Cette fois c'est bon !
```

```
}
```

Il y a deux erreurs possibles. La première consiste à se tromper dans le nom de la fonction. Elle ne se nomme pas fonction mais A::fonction, car elle fait partie de la classe A.

La deuxième erreur est beaucoup plus subtile. En effet une fonction membre reçoit en réalité implicitement un argument supplémentaire, une sorte de pointeur this sur l'objet qui appelle la fonction ! Il faut donc d'une certaine manière en tenir compte lors de la déclaration du pointeur, c'est pour cela qu'il faut mettre A:: devant le nom du pointeur lors de sa déclaration. La dernière ligne du code est donc correcte. Remarquez qu'il est nécessaire dans ce cas d'utiliser l'opérateur **&**.

S'il s'agit d'une fonction membre static, cette deuxième règle ne s'applique pas. En effet une fonction statique ne reçoit pas de pointeur sur un objet en argument "caché" puisqu'elle peut être appelée sans objet.

Cependant, un autre problème survient rapidement. Comment utiliser le pointeur puisqu'il nécessite un objet ?

Cela se fait de la manière suivante:

Code : C++ - Utilisation d'un pointeur sur fonction membre

```
class A{
    public:
       int fonction(int a);
    //...
};

int A::fonction(int a){return a*a;}

int main()
{
   int (A::*ptr)(int) = &A::fonction;  //On déclare un pointeur sur
la fonction membre

   A instance;  //On crée une instance de la classe A

   int resultat = (instance.*ptr)(2);
   //On calcule le résultat de la fonction pointée par "ptr"
appliquée à
   //l'objet "instance" avec comme argument "2"

   cout<< resultat  << endl;
   //Et on affiche.

   return 0;
}
```

Pfffou... 
En effet, les pointeurs sur fonctions membres ne sont pas très digestes à utiliser.
Vous remarquerez quand même l'utilisation obligatoire de l' opérateur* et la présence indispensable des parenthèses.

Simplifier les choses

Pour se simplifier la vie, il existe une solution que l'on rencontre parfois. Elle consiste à utiliser une macro. Normalement, les macros sont à éviter en C++ parce qu'on peut presque toujours s'en sortir sans. Ici, le but n'est pas d'utiliser une macro comme une fonction, mais plutôt comme moyen de simplifier drastiquement la notation.

Une des premières choses à faire est d'utiliser un **typedef** .

Code : C++ - Un typedef simplificateur

```
class A{
    public:
        int fonction(int x)
        {
            return x*x;
        }
    //...
};

typedef int (A::*APointeurFonction)(int)
```

En faisant ça, on peut utiliser ApointeurFonction à la place de la déclaration habituelle présente dans l'exemple précédent, ce qui donne:

Code : C++

```
int main()
{
   APointeurFonction ptr = &A::fonction;  //On utilise le typedef
ce qui simplifie la notation

   A instance;  //On crée une instance de la classe A

   int resultat = (instance.*ptr)(2);

   cout << resultat << endl;

   return 0;
}
```

L'appel à la fonction reste malgré tout fastidieux. C'est là qu'intervient la macro.

Code : C++ - La macro

```
#define appelleFonctionMembre(objet,pointeur) ((objet).*(pointeur))
```

L'appel à la fonction membre sera alors:

Code : C++

```
int main()
{
   APointeurFonction ptr = &A::fonction;

   A instance;

   int resultat = appelleFonctionMembre(instance,ptr)(2);
   //Avec la macro, c'est plus simple !

   cout << resultat << endl;

   return 0;
}
```

L'utilisation de la macro simplifie donc grandement le travail puisqu'on a eu besoin de réfléchir qu'une seule fois à la position des * et des parenthèses.

Je me répète, mais c'est important. Les macros sont généralement à éviter en C++. Ici, c'est un des rares cas où leur emploi se justifie.

Voilà, c'est tout ce qu'il y avait à dire sur ce petit sujet. Mais je ne pense pas que vous aurez besoin souvent des pointeurs de fonctions membres. 
Comme déjà dit dans l'introduction, il est plus simple d'utiliser des *foncteurs*, la "version C++" du pointeur de fonction qui permet de faire les choses plus facilement.

Merci à Chlab_lak pour avoir proposé l'utilisation de la macro.

Ce tutoriel a été corrigé par les zCorrecteurs.