8. Adresses, pointeurs

Remarque préliminaire

Dans ce chapitre, comme dans le reste du cours, on suppose que :

le type `int` occupe 4 octets, `double` 8 et `char` 1.

Il en est ainsi sur les PC utilisés en TD mais cela ne fait pas partie de la norme et peut varier d'un ordinateur à l'autre. Tout ce qui est exposé ici se transpose immédiatement aux autres cas.

1 Introduction

Dans ce cours les pointeurs sont présentés en vue de :

créer des variables indicées analogues à celles que l'on utilise en mathématiques (x_i, y_{ij}, z_{ijk}, etc.)
étendre les possibilités des fonctions (étudiées au chapitre **Fonctions**).

Le début de l'exposé sur les pointeurs est un peu formel, leur utilité n'apparaît qu'à partir de la section Application des pointeurs : transmission à une fonction par adresse.

2 Adresses

Dans la mémoire de l'ordinateur les octets sont numérotés, comme les chambres d'un hôtel. Quand on déclare une variable par son type et son nom on lui réserve une suite de n octets consécutifs : 4 pour un entier de type `int`, 8 pour un réel de type `double`, 1 pour un caractère de type `char`. Chaque variable a une « adresse » qui est le numéro du premier des octets qui lui sont réservés. La figure suivante schématise cette organisation pour les trois types précédents.

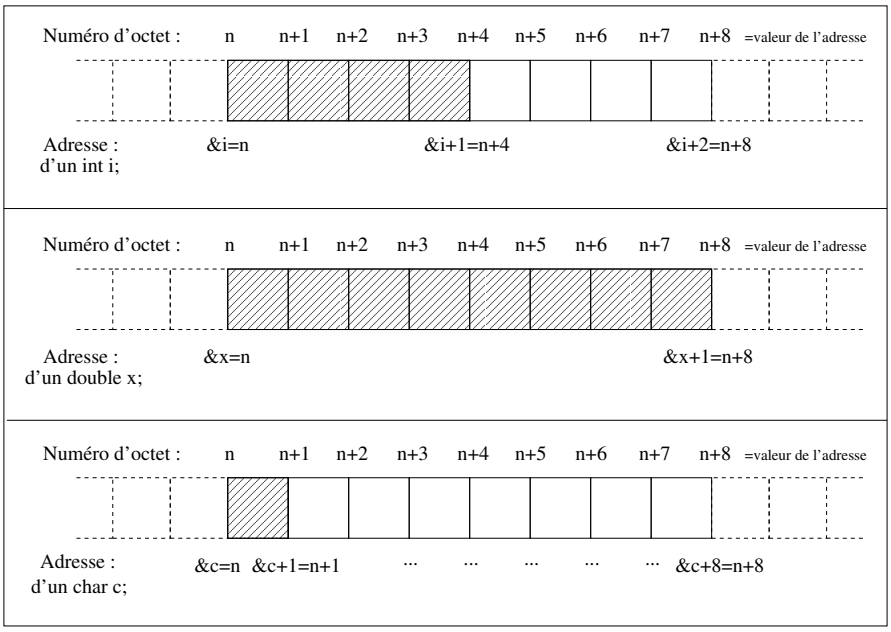

FIGURE 1 – Chaque carré symbolise un octet

L'adresse de la variable `i` est désignée par `&i`. Il est possible de connaître cette adresse explicitement par une instruction telle que `cout << (unsigned int)&i` [1], qui donne par exemple le résultat :

3219020668

Il faut bien distinguer entre l'adresse symbolique `&i` et sa valeur 3219020668. En effet si on écrit :

1. Si on écrivait simplement `cout << &i;` on obtiendrait cette adresse en notation hexadécimale.

```
#include<iostream>
using namespace std;
int main() {
  int i; double x;
  cout << (unsigned int)&i << " " << (unsigned int)(&i+1) << endl;
  cout << (unsigned int)&x << " " << (unsigned int)(&x+1) << endl;
  return 0;
}
```

on obtient :

```
3219020668 3219020672
3219020656 3219020664
```

ce qui montre que la valeur de l'adresse `&i+1` est égale à la valeur de l'adresse de `i` augmentée du nombre d'octets occupés par un `int` (c'est à dire 4) et que, de même, la valeur de l'adresse `&x+1` est égale à la valeur de l'adresse de `x` augmentée du nombre d'octets occupés par un `double` (c'est à dire 8).
Il faut remarquer également qu'on ne sait pas si ce qui se trouve à l'adresse `&i+1` est un `int` ou un `double` ou autre chose. Cette remarque s'applique aussi à `&x+1` sauf que, dans ce cas particulier, on peut affirmer qu'il ne s'agit pas d'un `double` puisqu'il n'y a pas assez de place pour un `double` à l'adresse 3219020664, l'octet à l'adresse 3219020668 étant déjà occupé par l'entier `i`. Les octets d'adresses 3219020664 à 3219020667 peuvent avoir été affectés à un `int`, des `char` ou autre chose, ou bien encore être libres, c'est à dire ne pas avoir été réservés.
De façon plus générale, `k` étant une variable entière, `&i+k` est l'adresse de `i` augmentée de `k` fois la taille d'un `int` et `&x+k` est l'adresse de `x` augmentée de `k` fois la taille d'un `double`. Mais, encore une fois, on ne sait pas à quoi sont affectés les octets suivant l'adresse `&i+k` ni ceux suivant l'adresse `&x+k` (sauf, évidemment, dans le cas particulier `k=0`).

On va voir dans ce qui suit qu'en général, on utilise les adresses par l'intermédiaire de variables d'un genre nouveau nommées « pointeurs » .

3 Pointeurs

3.1 Définition

Un pointeur est une variable qui contient non pas un nombre ou un caractère comme les variables étudiées jusqu'ici que nous pouvons qualifier d' « ordinaires » , mais l'adresse d'une variable « ordinaire » . Pour utiliser un pointeur il faut le déclarer en précisant le type de variable dont il pourra contenir l'adresse. Par exemple les instructions suivantes :

```
int* a;
double* b;
char* c;
```

déclarent trois pointeurs :

un nommé `a` susceptible de contenir une adresse de variable de type `int`
un nommé `b` susceptible de contenir une adresse de variable de type `double`
un nommé `c` susceptible de contenir une adresse de variable de type `char`.

Remarques

1. Un pointeur donné ne peut donc pas contenir l'adresse de n'importe quelle variable.
2. C'est le signe `*` qui indique que c'est une variable pointeur que l'on déclare.
3. Les déclarations `int* a;` et `int *a;` sont équivalentes pour le compilateur. La notation `int* a;` est plus claire : on déclare une variable qui s'appelle `a` et qui est du type « pointeur sur un `int` » (noté comme `int*`) et non pas une variable qui s'appelle `*a` et qui est du type `int`. Cependant, un peu illogique, quand on déclare plusieurs pointeurs avec une seule commande il faut obligatoirement répéter le signe `*` :

   ```
   int *a, *b;
   ```

Si on a déclaré les variables ordinaires suivantes :

```
int i;
double x;
char l;
```

on peut alors écrire des égalités telles que :

```
a = &i;
b = &x;
c = &l;
```

et `a` contient alors l'adresse de la variable entière `i`, `b` contient celle de la variable réelle `x`, `c` contient celle de la variable caractère `l`.

Remarque

L'attribution des adresses échappe complètement à l'utilisateur. Il est impossible d'écrire par exemple :

```
&x = ...
```

Une adresse ne peut donc pas figurer à gauche d'un signe `=`, on dit que ce n'est pas une « lvalue (left value) » , alors qu'un pointeur, lui, est une lvalue.

3.2 Opérations sur les pointeurs

3.2.1 L'opérateur *

L'opérateur `*` est un opérateur[2] qui s'applique à un pointeur : `*a` désigne la variable ordinaire dont l'adresse est celle contenue dans le pointeur `a`. Donc si `a` contient l'adresse de `i`[3], la variable `i` peut être désignée de deux façons différentes :

par `i` (façon habituelle)
par `*a` (par l'intermédiaire du pointeur).

Les instructions suivantes :

```
double i, j;
double* a;
...
a = &i;
j = *a;
```

attribuent à `j` la même valeur que si on avait simplement écrit :

```
double i, j;
...
j = i;
```

De même :

```
*a = j;
```

est équivalent à :

```
i = j;
```

Remarques

1. L'opérateur `*` s'applique aussi à une adresse : `*(&i)` est la même variable que `i`.
2. Suivant le contexte, l'opérateur `*` a deux significations en quelque sorte inverses l'une de l'autre :
   — la déclaration `int* i;` ou `int *i;` signifie que `i` est un pointeur (il faut voir le signe `*` ici comme faisant partie du nom du type `int*` et non pas comme un opérateur agissant sur `i`) ;
   — `i` étant un pointeur `*i` est une variable ordinaire.

3.2.2 Addition et soustraction d'entiers à des pointeurs, soustraction de pointeurs

À un pointeur on peut ajouter ou soustraire un entier et soustraire un autre pointeur du même type. Supposons que :

```
...
int k;
double x, y;
double *a, *b;
k = 5;
a = &x; b = &y;
...
```

2. Sans rapport avec la multiplication.
3. On dit alors que « `a` pointe sur `i` » .

donc que `a` pointe sur `x` et `b` sur `y`. Alors :

`a+k` est une adresse dont la valeur est égale à celle contenue dans le pointeur `a`, augmentée de `k` fois la taille d'un `double`, c'est donc ici `&x+k` [4]
`a-b` n'est pas une adresse, c'est simplement l'entier égal à la différence des valeurs de `&x` et `&y`.

Comme pour une adresse, une augmentation de `1` du pointeur augmente sa valeur d'un nombre d'octets qui dépend arithmétiquement du type du pointeur :

1 pour un `char`
4 pour un `int`
8 pour un `double`

Cette signification particulière des signes + et − pour les adresses et les pointeurs explique en partie pourquoi, en général, les pointeurs ont un type.

Remarque

`a+k` est une adresse et non un pointeur, on ne peut écrire par exemple `a+k = &x`.

Remarque importante

En conséquence de ce qui précède, `*(a+k)` est une variable ordinaire qui donne accès à ce qui est écrit à l'adresse de `x` augmentée de `k` fois la taille d'un `double`. Mais rien ne permet de savoir quel usage l'ordinateur fait de l'emplacement de mémoire situé à cette adresse [5]. Une instruction du type :

```
cout << *(a+k) << endl;
```

affichera ce qui est à l'adresse `&x+k` interprété comme un `double` mais ce n'en est pas un en général et, même si c'en est un, on ne sait pas à quoi il correspond.
Plus grave, une instruction du type :

```
*(a+k) = 1.;
```

par exemple, écrit à l'adresse `&x+k` alors que cet emplacement n'a pas été réservé et qu'il est affecté à un autre usage par l'ordinateur. Dans ce cas il y a violation de mémoire, ce qui peut entraîner des conséquences catastrophiques pour l'exécution du programme.
On ne se soucie pas de ce problème pour l'instant et on considère les manipulations sur les pointeurs de façon purement formelle. L'intérêt pratique de ces manipulations apparaîtra au chapitre **Tableaux dynamiques**.

3.2.3 L'opérateur « crochets » : `[]`

Soit `x` un pointeur sur un type quelconque et `k` une variable de type `int`. Par définition de l'opérateur crochets, `x[k]` est simplement une autre notation, strictement équivalente, pour la variable `*(x+k)` [6]. En particulier, `x[0]` et `*x` sont deux notations pour la même variable.

Remarques

1. Comme l'opérateur `*`, l'opérateur `[]` s'applique aussi aux adresses.
2. Pour le lecteur connaissant les tableaux standards du C, qui ne sont pas étudiés ni utilisés dans ce cours : malgré la similitude de la notation, le `x[k]` défini ici n'est pas un élément de tableau standard du C.

4 Application des pointeurs : transmission à une fonction par adresse

Considérons par exemple la fonction suivante et son appel :

```
int f(int y) {
  ...
}
int main() {
  int x;
  ...
```

4. `k` pourrait être négatif.
5. Sauf, évidemment dans le cas particulier `k=0` où il s'agit de l'emplacement de la variable ordinaire `x`.
6. Il se trouve qu'on peut aussi utiliser la notation `k[x]` au lieu de `x[k]` pour désigner `*(x+k)`. Le compilateur comprend les deux notations. Nous n'utiliserons pas cette possibilité.

```
  ... f(x) ...
  ...
}
```

On a vu au chapitre **Fonctions** que l'appel `f(x)` transmet à `y` la valeur de `x` et que la fonction s'exécute simplement comme si on avait écrit `y = x` à son début. Il y a une façon plus générale de transmettre des arguments à la fonction, à l'aide des adresses et pointeurs. Tout se passe alors comme si on transmettait non seulement des valeurs mais des variables. Jusqu'ici l'argument muet de `f`, `y`, est une variable ordinaire. Il est possible de la remplacer par un pointeur. On a alors :

```
int f(int* y) {
  ...
}
int main() {
  int x;
  ...
  ... f(&x) ...
  ...
}
```

L'appel se fait maintenant en passant l'adresse de `x` puisque `y` est un pointeur. La fonction s'exécute comme si on avait écrit `y = &x` à son début. Par ce procédé `x` et `*y` désignent le même emplacement de mémoire et sont donc deux noms différents pour une même variable, cette variable s'appelant `x` dans le programme principal et `*y` dans la fonction et ceci tout le temps de l'exécution de la fonction. Toute modification de `*y` dans la fonction est automatiquement répercutée sur `x`. On voit qu'il y a là une autre possibilité de retourner des valeurs au programme appelant, par l'intermédiaire des arguments et non plus seulement par le `return` de la fonction. Ce type de transmission est nommé « transmission par adresse » .

Exemple 1

C'est l'exemple déjà vu au chapitre **Fonctions** :

```
#include<iostream>
using namespace std;
int f(int y) {
  y = y+1;
  return y;
}
int main() {
  int x;
  x = 1;
  cout << x << endl;
  cout << f(x) << endl;
  cout << x << endl;
  return 0;
}
```

Résultat :

```
1
2
1
```

C'est la transmission par valeur, dans le `main`, `x` n'est pas modifié par l'appel de la fonction.

Exemple 2

On remplace l'argument de la fonction `f` par un pointeur :

```
#include<iostream>
using namespace std;
int f(int* y) {
  *y = *y+1;
  return *y;
}
```

```
int main() {
  int x;
  x = 1;
  cout << x << endl;
  cout << f(&x) << endl;
  cout << x << endl;
  return 0;
}
```

Résultat :

```
1
2
2
```

C'est la transmission par adresse. Durant l'exécution de la fonction `f`, le `x` du `main` est la même variable que le `*y` de la fonction donc il est aussi augmenté de 1 après l'exécution de la fonction.

Remarque marginale mais troublante

Si on écrit le programme précédent en faisant simplement afficher les trois quantités par le même `cout`, en ne changeant rien d'autre :

```
#include<iostream>
using namespace std;
int f(int *y) {
  *y = *y+1;
  return *y;
}
int main() {
  int x;
  x = 1;
  cout << x << f(&x) << x << endl;
  return 0;
}
```

on obtient :

```
2 2 1
```

parce que le calcul des quantités à afficher n'est pas fait dans l'ordre de gauche à droite comme on s'y attendrait.

Exemple 3

On met deux pointeurs au lieu d'un en argument de la fonction `f` :

```
#include<iostream>
using namespace std;
int f(int* y, int* z) {
  *y = *y+1;
  *z = *z+1;
  return *y;
}
int main() {
  int x;
  x = 1;
  cout << x << endl;
  cout << f(&x,&x) << endl;
  cout << x << endl;
  return 0;
}
```

Résultat :

```
1
3
3
```

Puisqu'on appelle la fonction avec deux fois l'argument effectif `&x` le `x` du `main()` et les `*y` et `*z` de la fonction ne sont qu'une seule et même variable dont la valeur est deux fois augmentée de 1, donc le résultat doit bien être 3.

Exemple d'utilisation du retour de valeur par argument :

On veut résoudre l'équation du second degré :

$$ax^2 + bx + c = 0$$

pour n'importe quelles valeurs réelles des trois paramètres a, b, c. On écrit une fonction à laquelle on fournit ces trois paramètres en argument et ayant la valeur :

0 s'il n'y a pas de racine réelle
1 s'il y a une seule racine réelle (simple ou double)
2 s'il y a deux racines réelles distinctes
3 dans le cas $a = b = c = 0$ (tout x est solution)

La fonction retourne de plus par ses arguments les valeurs des racines quand il y en a.

```
#include<iostream>
#include<math.h>
using namespace std;
int r2(double a, double b, double c, double* xp, double* xs) {
  double delta, rd;
  if (a == 0.)
    if (b == 0.)
      if (c == 0.)
        return 3;
      else
        return 0;
    else {
      *xp = -c/b;
      *xs = *xp;
      return 1;
    }
  else {
    delta = b*b - 4.*a*c;
    if (delta < 0.)
      return 0;
    else if (delta == 0.) {
      *xp = -b/2./a;
      *xs = *xp;
      return 1;
    }
    else {
      rd = sqrt(delta);
      *xp = (-b-rd)/2./a;
      *xs = (-b+rd)/2./a;
      return 2;
    }
  }
}
int main() {
  double x1, x2;
  int ind;
  ind = r2(1., -9., 14., &x1, &x2);
  if (ind == 0) cout << "Pas de racines" << endl;
  if (ind == 1) cout << "Une racine : " << x1 << endl;
  if (ind == 2) cout << "Deux racines : " << x1 << " " << x2 << endl;
  if (ind == 3) cout << "Infinité de racines (pas d'équation)" << endl;
```

```
  return 0;
}
```

5 Pointeur de pointeur

5.1 Définition

Comme une variable ordinaire, un pointeur a une adresse et on peut définir des variables qui contiennent l'adresse d'un pointeur : ce sont des pointeurs de pointeur. Ils sont, par exemple, déclarés de la façon suivante :

```
double** p
```

`p` est un pointeur de pointeur pouvant contenir l'adresse d'un pointeur sur un `double`. Supposons que :

```
double a;
double* b;
double** c;
b = &a;
c = &b;
```

alors :

`a` et `*b` sont deux notations pour la même variable ordinaire (déjà vu)
`b` et `*c` sont deux notations pour le même pointeur
`**c`[7] et `a` sont deux notations pour la même variable ordinaire

On peut ainsi écrire par exemple :

```
#include<iostream>
using namespace std;
int main() {
  double a, *b, **c;
  a = 7.; b = &a; c = &b;
  cout << a << " " << *b << " " << **c << endl;
  return 0;
}
```

qui donne :

```
7 7 7
```

5.2 L'opérateur [] appliqué à un pointeur de pointeur

Définissons :

```
double x, *p, **pp;
p = &x; pp = &p;
```

7. Qui représente `*(*c)`

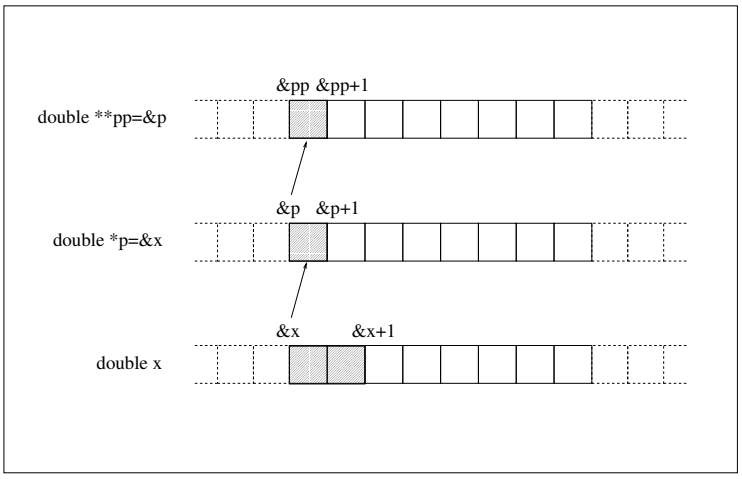

FIGURE 2 – Chaque carré symbolise quatre octets. On suppose ici un système d'exploitation à 32 bits, où les pointeurs ont une taille de 4 octets. Dans un système d'exploitation à 64 bits les pointeurs ont une taille de 8 octets.

On a :

`pp` est un pointeur de pointeur sur un `double`
`pp[i]` et `*(pp+i)` représentent la même variable (qui est un pointeur sur un `double`)
`pp[i][j]` [8] et `*(*(pp+i)+j)` représentent la même variable (qui est une variable `double` ordinaire).

Il faut rappeler que `pp[i]`, `*(pp+i)`, `pp[i][j]` et `*(*(pp+i)+j)`, déclarées comme elles le sont, sont des variables « illégales » , sauf si `i=0` et `j=0`, mais que nous avons choisi de ne pas nous préoccuper de cette question pour l'instant. Dans le cas particulier `i=0` et `j=0`, `x`, `*p`, `p[0]`, `**pp`, `*pp[0]` et `pp[0][0]` représentent la même variable « légale » (qui est une variable `double` ordinaire).

Exemple

```
#include<iostream>
using namespace std;
int main() {
  double x, *p, **pp;
  p = &x; pp = &p;
  x = 5.;
  cout << x << " " << *p << " " << p[0] << " " << **pp << " " << *pp[0]
       << " " << pp[0][0] << endl;
  return 0;
}
```

donne :

```
5 5 5 5 5 5
```

6 Pointeur sur une fonction

6.1 Déclaration d'un pointeur sur une fonction

Soit, par exemple, la fonction `f` de prototype :

```
double f(double);
```

`f` a une adresse et cette adresse peut être attribuée à un pointeur `p` déclaré avec le type correspondant au prototype de `f`, c'est à dire ici :

```
double (*p)(double);
```

8. Qui est équivalent à `(pp[i])[j]`

`p` ne peut pointer que sur des fonctions de type `double` ayant un seul argument de type `double`. Pour une fonction de prototype :

```
double g(double,int);
```

il faut un pointeur déclaré par :

```
double (*q)(double,int);
```

Ceci permet d'utiliser successivement un même nom pour des fonctions différentes, ou si l'on préfère, de définir une « fonction variable » . Par exemple :

```
double f(double x) {
  ...
}
double g(double y) {
  ...
}
int main() {
  double a;
  ...
  double (*p)(double);
  ...
  p = f; /* à partir d'ici (*p)(a), ou plus simplement p(a), est équivalent à f(a) */
  ...
  p = g; /* à partir d'ici (*p)(a), ou plus simplement p(a), est équivalent à g(a) */
  ...
}
```

On remarque sur cet exemple que l'adresse d'une fonction est donnée par `f` et non `&f`.

6.2 Fonction en argument d'une fonction

Reprenons la fonction `integ` qui calcule l'intégrale d'une fonction par la méthode des rectangles, écrite au chapitre **Fonctions**. Elle a un défaut important : elle ne peut, durant une même exécution du programme, intégrer qu'une seule fonction, celle définie par ailleurs avec le nom `f`, puisque ce nom figure explicitement dans `integ`. Si l'on veut intégrer une autre fonction il faut modifier le programme et recompiler. Il serait beaucoup plus général que la fonction `f` puisse varier d'un appel à l'autre de `integ`. Ceci peut être obtenu en mettant `f` en argument à l'aide d'un pointeur. Il suffit d'ajouter dans `integ` un argument muet `double (*f)(double)` qui est un pointeur sur une fonction. Dans `integ`, la fonction à intégrer sera alors désignée par `f` ou `(*f)` [9].
Ce qui donne :

```
double integ(double a, double b, double (*f)(double), int n) {
  double h, s;
  int i;
  h = (b-a)/n;
  s = 0.;
  for (i = 1; i <= n; i++)
    s = s + f(a+(i-0.5)*h));
  return h*s;
}
```

L'utilisation de `integ` peut alors se faire selon l'exemple suivant :

```
#include<iostream>
#include<math.h>
using namespace std;
double f1(double x) {
  return 1./(1.+x*x);
}
double f2(double x) {
  return x*(x-1.);
```

9. Les deux sont possibles.

```
}
double integ(double a, double b, double (*f)(double), int n) {
  double h, s; int i;
  h = (b-a)/n; s = 0.;
  for(i = 1; i <= n; i++)
    s += f(a+(i-0.5)*h);
  return h*s;
}
int main() {
  double a1, b1, a2, b2;
  int np1, np2;
  a1 = 1.; b1 = 2.; np1 = 1000;
  a2 = 0.; b2 = 1.; np2 = 1000;
  cout << integ(a1,b1,f1,np1) << endl;;
  cout << integ(a2,b2,f2,np2) << endl;
  return 0;
}
```

Remarques

1. Dans les arguments muets de l'en-tête d'une fonction il est possible d'enlever l'étoile et les parenthèses dans la déclaration d'un pointeur sur une fonction : `double f(double)` au lieu de `double (*f)(double)` dans cet exemple-ci. Mais cela n'est pas possible si le pointeur sur une fonction est déclaré à l'intérieur d'une fonction. Il n'est donc pas possible de remplacer `double (*p)(double);` par `double p(double);` dans l'exemple de la section précédente. C'est logique : cette dernière notation est la façon de déclarer le prototype d'une fonction.
2. Attention : pour utiliser une fonction `g` dans une fonction `f` il n'est nullement nécessaire de transmettre `g` en argument de `f`. Ce n'est que dans le cas où une fonction `g` utilisée par `f` doit pouvoir changer d'un appel à l'autre de `f` qu'il faut mettre `g` en argument de `f`.
3. Ne pas confondre `f(g)` et `f(g(x))`. Dans le premier cas c'est la fonction `g` qui est transmise en argument à la fonction `f`, tandis que dans le second, c'est la valeur de `g` au point `x` qui est transmise à `f`.
4. Dans le chapitre **Chaînes de caractères** (hors programme) il est décrit comment on peut, de plus, fournir la fonction à intégrer comme la valeur d'une variable lue par un `cin`.