7. Fonctions

1 Définition et utilité des fonctions

Les fonctions sont des parties de programme totalement ou partiellement indépendantes du reste, auxquelles on peut fournir des données, et qui accomplissent une tâche en renvoyant éventuellement un résultat.

L'indépendance des fonctions fait leur intérêt : on peut les utiliser, dans une certaine mesure, comme des boîtes noires, sans avoir à se soucier de leur intégration dans le programme qui les utilise. Ceci clarifie grandement la construction et le développement des programmes.

Dans ce chapitre on considérera uniquement des fonctions écrites dans le même fichier [1] que le programme principal. Pour une fonction d'usage universel, par exemple la multiplication de matrices, ce n'est pas pratique puisqu'il faut recopier cette fonction dans le fichier de chaque programme qui y fait appel. Dans un chapitre ultérieur on verra comment mettre des fonctions dans des fichiers autonomes tout en pouvant les utiliser à partir d'autres fichiers. Ceci permet de constituer des « bibliothèques » de fonctions écrites une fois pour toutes.

2 Exemples

Exemple 1

Supposons qu'on ait à calculer la quantité :

$$y = \frac{1}{1+x^2}$$

pour diverses valeurs de x : x_1, x_2, ... Au lieu d'écrire :

```
y1 = 1./(1.+x1*x1)
y2 = 1./(1.+x2*x2)
...
```

il est plus élégant et plus lisible de définir une fonction, au sens de la programmation, nommée par exemple `y`, qui calcule y pour toute valeur de x fournie en argument. Ceci s'écrit :

```
#include<iostream>
using namespace std;
int main() {
  double x1, x2;
  double y(double);    // <- prototype
  x1 = 1.; x2 = 4.;
  cout << y(x1) << endl;
  cout << y(x2) << endl;
  return 0;
}
double y(double x) {   // <- en-tete
  return 1./(1.+x*x);
}
```

On voit que dans ce cas il existe un autre bloc que celui de `main`. C'est dans ce bloc supplémentaire qu'est définie la fonction. Le programme est donc divisé en deux parties :

— le `main`

— une fonction comprise dans le bloc suivant l'instruction `double y(double x)` (nommée « en-tête de la fonction »)

On remarque qu'il faut déclarer le type de la fonction et de ses arguments deux fois :

— une fois à l'intérieur de `main` par `double y(double)` (cette déclaration se nomme un « prototype »)

— une fois dans l'en-tête de la fonction par `double y(double x)`

On voit que dans un prototype il n'est pas nécessaire d'inclure les noms des arguments (ici `x`), seulement leurs types (ici `double`). Par contre, ce n'est pas proscrit non plus de les inclure et cela peut rendre un prototype plus lisible.

Remarque

1. On peut dire aussi, ce qui est équivalent, dans le même module.

Dans cet exemple :

`x` est la variable de la fonction, on l'appelle « argument muet »
`x1`, `x2`, ... sont des valeurs particulières de `x` pour lesquelles on veut calculer la fonction, on les appelle « arguments effectifs » .

Remarque importante

L'ordre dans lequel on écrit le `main` et la fonction `y` n'est pas imposé. Si on écrit la fonction en premier il n'est plus nécessaire de la déclarer dans le `main`. Cette possibilité est intéressante si on veut éviter des déclarations redondantes.

Exemple 2

Calcul de la distance de deux points du plan avec une fonction. La différence avec l'exemple précédent est qu'il y a plusieurs arguments et qu'on a défini en premier la fonction.

```
#include<iostream>
#include<math.h>
using namespace std;
double dist(double xa, double ya, double xb, double yb) { // <- en-tete de la fonct. dist
  return sqrt((xb-xa)*(xb-xa) + (yb-ya)*(yb-ya));
}
int main() {
  double dist(double, double, double, double); // <- prototype facultatif ici
                                               // parceque la définition de
                                               // dist précède la fonction main
                                               // qui l'appelle
  double x1, y1, x2, y2;
  x1 = 1.; y1 = 2.; x2 = 3.; y2 = 4.;
  cout << dist(x1,y1,x2,y2) << endl;
  x1 = 0.5; y1 = -1.; x2 = 3.7; y2 = -0.3;
  cout << dist(x1,y1,x2,y2) << endl;
  return 0;
}
```

Exemple 3

Fonction sinus cardinal $\sin x/x$, en prévoyant le cas $x = 0$:

```
#include<iostream>
#include <math.h>
using namespace std;
double sinc(double x) {
  if (x == 0.)
    return 1;
  else
    return sin(x)/x;
}
int main() {
  double u;
  u = 1.5;
  cout << sinc(u) << endl;
  return 0;
}
```

Une fonction peut ne pas retourner de résultat ou ne pas avoir d'argument.

- s'il n'y a pas de résultat retourné : dans l'en-tête on remplace le type de la fonction par le mot clé `void`
- s'il n'y a pas d'argument : dans les parenthèses de l'en-tête on remplace les types et les noms des arguments par rien du tout (parenthèses vides).

Deux exemples de telles fonctions :

Exemple 4

Fonction imprimant les n premières puissances de 2 (de 2^0 à 2^{n-1}) et un exemple de son appel avec n=10 :

```
#include<iostream>
using namespace std;
void p(int n) {
  int i, x = 1;
  for (i = 0; i <= n-1; i++) {
    cout << x << endl;
    x = x*2;
  }
}
int main() {
  p(10);
  return 0;
}
```

Exemple 5

Fonction qui, à chaque appel, écrit une ligne d'étoiles :

```
#include<iostream>
using namespace std;
void ligne_etoiles() {
  cout << "****************" << endl;
}
int main() {
  ligne_etoiles();
  return 0;
}
```

Exemple 6

On a vu au chapitre **Tests et boucles** un programme calculant, par la méthode des rectangles, l'intégrale :

$$\int_a^b \frac{1}{1+x^2}\,dx$$

Réorganisons le en écrivant séparément la fonction à intégrer, le calcul de l'intégrale et le programme principal :

```
#include<iostream>
using namespace std;
double f(double x) {
  return 1./(1.+x*x);
}
double integ(double a,double b,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;
}
int main() {
  double x1, x2;
  int np;
  x1 = -0.57; x2 = 1.17; np = 1000;
  cout << integ(x1,x2,np) << endl;
  return 0;
}
```

Sous cette forme le programme est plus lisible, il est plus facile de changer la fonction à intégrer sans risquer d'erreur, il suffit pour cela de modifier l'expression de `f` dans sa définition.

Remarque

On voit sur ces exemples qu'une fonction peut à la fois effectuer une tâche et renvoyer une valeur.

3 Passage des valeurs en argument

Il est important de comprendre comment les valeurs des arguments sont transmises à une fonction. Une fonction est un élément de programme qui s'exécute à chaque appel. Si, par exemple, on a une fonction d'en-tête :

```
int f(int x)
  ...
```

et qu'on l'appelle par :

```
  ...
  y=3;
  ... f(y);
```

ou :

```
  ...
  ... f(3);
```

cela signifie que la fonction s'exécute comme si on avait ajouté à son début l'instruction `x=3` et c'est tout. On appelle ce type de transmission des valeurs des arguments « passage par valeur » . Supposons que la fonction contienne une instruction qui modifie la valeur de `x` fournie par l'appel (`3` dans ce cas) comme suit :

```
int f(int x)
  ...
  x = 4
  ...
```

Lorsque l'exécution de la fonction est terminée et que l'on retourne à la fonction appelante (ici `main`), la valeur de la variable effective `y` du programme appelant est inchangée, elle est restée égale à `3`. La transmission des valeurs se fait uniquement dans le sens fonction appelante vers la fonction appelée. Ce fonctionnement est différent de celui d'un langage comme le Fortran où la transmission peut avoir lieu dans les deux sens et est dite « par adresse » . [2]

4 Variables locales, variables globales

Les variables déclarées dans une fonction (variables déclarées à l'intérieur et arguments muets) sont dites « locales » à la fonction (ceci s'applique au `main` qui peut être considéré comme une fonction particulière sans argument). Les variables de l'une des fonctions ne sont pas accessibles à l'autre et réciproquement. Les variables de deux fonctions différentes n'ont rien à voir entre elles : même si elles portent le même nom elles désignent des emplacements de mémoire différents, leurs contenus sont complètement indépendants. C'est d'ailleurs ce qui fait l'intérêt principal des fonctions : on peut en écrire une sans se soucier si les noms de variables utilisés sont déjà employés ailleurs dans le programme, ce qui assure automatiquement son indépendance.
Cependant il est souvent nécessaire de disposer de variables communes à deux ou plusieurs fonctions. Si, par exemple, un ensemble de paramètres est utilisé par plusieurs fonctions il est compliqué de devoir soit les définir dans une des fonctions et les passer en argument aux autres soit les définir dans chacune des fonctions. La solution est donnée par l'emploi de variables « globales » . On peut déclarer des variables à l'extérieur de toute fonction : elles sont alors communes à toutes les fonctions suivant cette déclaration. Une modification de leur contenu est connue simultanément de toutes les fonctions qui les ont en commun. Considérons un programme contenant une fonction nommée `f` :

```
int main() {
  double x,z;
  ...
}
... f(...) {          /* symbolise l'en tete de la fonction f */
  double y,z;
  ...
}
```

`x` est inconnue de la fonction, `y` est inconnue de `main`. Les variables `z` de `main` et de la fonction n'ont rien à voir entre elles (elles pourraient très bien ne pas être du même type, par exemple).
Si maintenant on déclare comme suit :

2. On verra que la transmission peut aussi se faire par adresse en C, au chapitre **Pointeurs**.

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

Les statuts de `x` et `y` sont inchangés mais `z` devient unique et commune au `main` et à la fonction.
Si on veut qu'une variable `v` soit commune uniquement à deux fonctions `f1` et `f2`, il faut écrire :

```
int main() {
  ...
}
double v;
... f1(...) {
  ...
}
... f2(...) {
  ...
}
```

car `v` n'est commune qu'aux fonctions qui suivent sa déclaration. La définition des fonctions `f1` et `f2` se trouvant dans ce cas après `main`, il est nécessaire de déclarer `f1` et `f2` par leur prototype dans `main`.

Remarque

En cas d'homonymie, dans une fonction, entre une variable locale et une variable globale c'est la variable locale qui est prise en considération, on dit que la variable globale est « masquée » .

5 Durée de vie des variables

Les variables locales des fonctions n'existent que le temps de l'exécution de la fonction. Au retour à la fonction appelante les emplacements mémoire de ces variables sont libérés pour être utilisés à autre chose. Si on a donné une certaine valeur à une variable locale d'une fonction lors d'un premier appel, cette valeur ne sera pas retrouvée à l'appel suivant. Pour la fonction `main` la question ne se pose pas puisqu'elle n'est exécutée qu'une fois.
Si l'on a besoin que la valeur de la variable soit conservée d'un appel à l'autre il faut la déclarer avec l'attribut `static` :

```
static int i;
```

par exemple. Une telle variable est dite « statique » . Cela ne veut pas dire que `i` est constante mais qu'elle a, au début d'un appel, la valeur qu'elle avait à la fin de l'appel précédent. Un exemple sera vu à la section **Initialisation des variables**.
Une variable locale non statique est dite « automatique » .

6 Initialisation des variables

6.1 Variables locales du programme principal

Les variables déclarées dans le `main` ne sont pas initialisées à priori, leur valeur peut être n'importe quoi tant que l'utilisateur ne leur en a pas attribué explicitement une, ce qui peut se faire par :

```
int i = 1;
```

ou, ce qui est équivalent, par :

```
int i;
i = 1;
```

sauf s'il s'agit d'une constante qui doit nécessairement être initialisée et ne peut l'être que par :

```
const int i = 1;
```

6.2 Variables locales des fonctions

6.2.1 Cas des variables automatiques

Elles ne sont pas initialisées à priori. Elles peuvent l'être par :

```
int i;
i = 1;
```

ou

```
int i = 1;
```

et dans les deux cas l'initialisation est faite à chaque appel de la fonction.

6.2.2 Cas des variables statiques

Elles sont initialisées à priori à 0. On peut les initialiser à une autre valeur par :

```
static int i = 1;
```

et dans les deux cas l'initialisation n'est faite qu'au premier appel, ce qui est évidemment nécessaire.

6.3 Variables globales

Elles sont à priori initialisées à 0 par le compilateur, on peut bien sûr les initialiser à une autre valeur par :

```
int j = 1;
```

6.4 Valeurs autorisées pour l'initialisation

6.4.1 Variables statiques

Elles ne peuvent être initialisées que par des valeurs connues à la compilation, c'est à dire par des constantes explicites ou déclarées par `const`, ou encore des expressions constituées de telles constantes, comme dans l'exemple suivant :

```
const int m = 3, n = 5;
static int k = m*n + 2*m + 4*n;
...
```

6.4.2 Variables automatiques

Elles peuvent être initialisées par une expression contenant des variables et des fonctions quelconques ayant été définies précédemment :

```
int m = 3, n = 5, k = m*n + 2*m + 4*n;
double pi = acos(-1.), dpi = 2*pi;
```

car ces dernières déclarations ne sont qu'une abréviation de :

```
int m, n, k;
double pi, dpi;
m = 3; n = 5; k = m*n + 2*m + 4*n;
pi = acos(-1.); dpi = 2*pi;
```

7 Appel d'une fonction par une fonction

Toute fonction peut être appelée par n'importe quelle autre fonction, à condition que la fonction appelée soit déclarée par son prototype dans la fonction appelante ou qu'elle soit placée avant la fonction appelante dans le fichier contenant les instructions du programme. L'argument d'une fonction peut très bien être la valeur retournée par elle même ou une autre fonction donc on peut écrire :

`f(f(f(...)))` ou `f(g(h(...)))` sans limitation [3]

3. Fonction composée, à ne pas confondre avec la notion de fonction récursive.

8 Fonctions à nombre variable d'arguments

Une fonction peut avoir un nombre variable d'arguments. Considérons la fonction suivante, nommée `prod`, à laquelle on fournit en arguments d'entrée :

- un entier par l'intermédiaire de la variable `n`
- `n` nombres

et dont la valeur est le produit des `n` nombres.

```
#include<iostream>
#include<stdarg.h>
using namespace std;
double prod(int n, ...) {
  int i; double p;
  va_list ap;
  va_start(ap,n);
  for(p = 1, i = 0; i < n; i++)
    p *= va_arg(ap,double);
  va_end(ap);
  return p;
}
/*************************************************************************/
int main() {
  cout << prod(2, 1.5, 3.) << endl;
  cout << prod(3, 1.2, 4., 10.) << endl;
  return 0;
}
```

Les pointillés dans l'en-tête de la fonction signifient que c'est une fonction à nombre variable d'arguments. [4] Une telle fonction doit avoir au moins un argument fixe, ici c'est `n`, et le ou les arguments fixes doivent figurer en premier.
Le type `va_list` et les fonctions pré-définies `va_start`, `va_arg`, `va_end` font partie de la bibliothèque `stdarg.h`, il faut donc la directive `#include<stdarg.h>`.
Le second argument de `va_start` (ici `n`) doit être le dernier des arguments muets fixes de la fonction à nombre variable d'arguments. Ce n'est pas nécessairement une variable contenant le nombre d'arguments variables effectifs dans l'appel.
Il n'est pas demandé aux étudiants de maîtriser les fonctions de ce type, mais la connaissance de leur existence et de leur utilisation peut être utile en pratique. Elles ne seront employées dans ce cours que pour initialiser les tableaux dynamiques, étudiés dans la suite.

9 Fonctions récursives

Ce sont des fonctions qui s'utilisent elles-même dans leur définition, ce qui est autorisé par le C. On peut par exemple écrire une fonction calculant $n!$ de la façon suivante :

```
int fac(int n) {
  if (n == 0)
    return 1;
  else
    return n*fac(n-1);
}
```

La récursivité conduit dans certains cas à une programmation élégante. Par contre elle peut conduire à un fort encombrement de la mémoire et à des calculs lents. Elle est mentionnée ici uniquement pour son importance « culturelle » et ne sera pas utilisée dans ce cours.

4. Contrairement à tous les autres cas dans ce poly, où les pointillés indiquent simplement des types, des arguments ou des lignes qu'on n'a pas explicités, ici il faut vraiment écrire des pointillés dans le programme.

10 Limites de l'utilisation des fonctions telle qu'elle vient d'être exposée

Un inconvénient majeur, dans le cadre étudié jusqu'à présent, est que les fonctions ne peuvent retourner que la valeur d'une seule variable. On ne peut pas écrire par exemple :

```
...
return (a,b,c);
```

On ne peut pas retourner par la valeur de la fonction les deux racines d'une équation du second degré ou les composantes d'un vecteur.
Il serait également très intéressant de pouvoir fournir une fonction en argument d'une fonction. Supposons par exemple que l'on écrive une fonction C, nommée `som`, par exemple, qui calcule numériquement l'intégrale :

$$\int_a^b f(x)\,dx$$

f étant une fonction fixée, par exemple $\dfrac{1}{1+x^2}$. La fonction `som` a pour argument `a` et `b`, on l'appelle donc par `som(a,b)`. Mais elle ne peut calculer l'intégrale que de la fonction $\dfrac{1}{1+x^2}$ puis celle-ci est incluse dans sa définition. Il serait beaucoup plus général de pouvoir fournir aussi en argument la fonction à intégrer par un appel du genre `som(a,b,f)`, `f` étant la fonction à intégrer.
Pour parvenir à ces buts il faut utiliser les pointeurs.