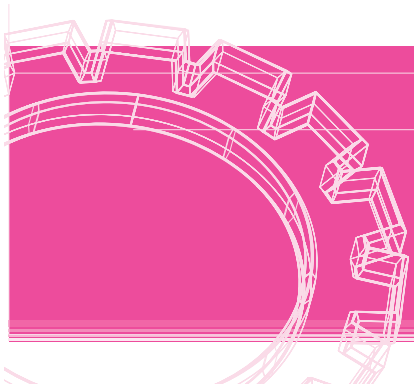
Chapitre 3
Les fonctions



Rappels

Généralités

Une fonction est un bloc d'instructions éventuellement paramétré par un ou plusieurs arguments et pouvant fournir un résultat nommé souvent « valeur de retour ». On distingue la définition d'une fonction de son utilisation, cette dernière nécessitant une déclaration.

La définition d'une fonction se présente comme dans cet exemple :

```
float fexple (float x, int b, int c)    // en-tête de la fonction
{  // corps de la fonction
}
```

L'en-tête précise le nom de la fonction (`fexple`) ainsi que le type et le nom (muet) de ses différents arguments (`x`, `b` et `c`). Le corps est un bloc d'instructions qui définit le rôle de la fonction.

Au sein d'une autre fonction (y compris `main`), on utilise cette fonction de cette façon :

```
float fepxle (float, int, int) ;  // déclaration de fexple ("prototype")
   .....
fexple (z, n, p) ; //appel fexple avec les arguments effectifs z, n et p
```

La déclaration d'une fonction peut être omise lorsqu'elle est connue du compilateur, c'est-à-dire que sa définition a déjà été rencontrée dans le même fichier source.

Mode de transmission des arguments

Par défaut, les arguments sont transmis par valeur. Dans ce cas, les arguments effectifs peuvent se présenter sous la forme d'une expression quelconque.

En faisant suivre du symbole & le type d'un argument dans l'en-tête d'une fonction (et dans sa déclaration), on réalise une transmission par référence. Cela signifie que les éventuelles modifications effectuées au sein de la fonction porteront sur l'argument effectif de l'appel et non plus sur une copie. On notera qu'alors l'argument effectif doit obligatoirement être une `lvalue` du même type que l'argument muet correspondant. Toutefois, si l'argument muet est, de surcroît, déclaré avec l'attribut `const`, la fonction reçoit quand même une copie de l'argument effectif correspondant, lequel peut alors être une constante ou une expression d'un type susceptible d'être converti dans le type attendu.

Ces possibilités de transmission par référence s'appliquent également à une valeur de retour (dans ce cas, la notion de constance n'a plus de signification).

Note La notion de référence est théoriquement indépendante de celle de transmission d'argument ; en pratique, elle est rarement utilisée en dehors de ce contexte.

L'instruction return

L'instruction `return` sert à la fois à fournir une valeur de retour et à mettre fin à l'exécution de la fonction. Elle peut mentionner une expression ; elle peut apparaître à plusieurs reprises dans une même fonction ; si aucune instruction `return` n'est mentionnée, le retour est mis en place à la fin de la fonction.

Lorsqu'une fonction ne fournit aucun résultat, son en-tête et sa déclaration comportent le mot `void` à la place du type de la valeur de retour, comme dans :

```
void fSansValRetour (...)
```

Lorsqu'une fonction ne reçoit aucun argument, l'en-tête et la déclaration comportent une liste vide, comme dans :

```
int fSansArguments ()
```

Les arguments par défaut

Dans la déclaration d'une fonction, il est possible de prévoir pour un ou plusieurs arguments (obligatoirement les derniers de la liste) des valeurs par défaut ; elles sont indiquées par le signe =, à la suite du type de l'argument, comme dans cet exemple :

```
float fct (char, int = 10, float = 0.0) ;
```

Ces valeurs par défaut seront alors utilisées lorsqu'on appellera ladite fonction avec un nombre d'arguments inférieur à celui prévu. Par exemple, avec la précédente déclaration, l'appel `fct ('a')` sera équivalent à `fct ('a', 10, 0.0)` ; de même, l'appel `fct ('x', 12)` sera équivalent à `fct ('x', 12, 0.0)`. En revanche, l'appel `fct ()` sera illégal.

Conversion des arguments

Lorsqu'un argument est transmis par valeur, il est éventuellement converti dans le type mentionné dans la déclaration (la conversion peut être dégradante). Ces possibilités de conversion disparaissent en cas de transmission par référence : l'argument effectif doit alors être une *lvalue* du type prévu ; ctte dernière ne peut posséder l'attribut `const` si celui-ci n'est pas prévu dans l'argument muet ; en revanche, si l'agument muet mentionne l'attribut `const`, l'argument effectif pourra être non seulement une constante, mais également une expression d'un type quelconque dont la valeur sera alors convertie dans une variable temporaire dont l'adresse sera fournie à la fonction.

Variables globales et locales

Une variable déclarée en dehors de toute fonction (y compris du `main`) est dite globale. La portée d'une variable globale est limitée à la partie du programme source suivant sa déclaration. Les variable globales ont une classe d'allocation statique, ce qui signifie que leurs emplacements en mémoire restent fixes pendant l'exécution du programme.

Une variable déclarée dans une fonction est dite locale. La portée d'une variable locale est limitée à la fonction dans laquelle elle est définie. Les variables locales ont une classe d'allocation automatique, ce qui signifie que leurs emplacements sont alloués à l'entrée dans la fonction, et libérés à la sortie. Il est possible de déclarer des variables locales à un bloc.

On peut demander qu'une variable locale soit de classe d'allocation statique, en la déclarant à l'aide du mot-clé `static`, comme dans :

```
static int i ;
```

Les variables de classe statique sont, par défaut, initialisées à zéro. On peut les initialiser explicitement à l'aide d'expressions constantes d'un type compatible par affectation avec celui de la variable. Celles de classe automatique ne sont pas initialisées par défaut. Elles doivent être initialisées à l'aide d'une expression quelconque, d'un type compatible par affectation avec celui de la variable.

Surdéfinition de fonctions

En C++, il est possible, au sein d'un même programme, que plusieurs fonctions possèdent le même nom. Dans ce cas, lorsque le compilateur rencontre l'appel d'une telle fonction, il effectue le choix de la « bonne » fonction en tenant compte de la nature des arguments effectifs.

D'une manière générale, si les règles utilisées par le compilateur pour sa recherche sont assez intuitives, leur énoncé précis est assez complexe, et nous ne le rappellerons pas ici. On trouvera de nombreux exemples de surdéfinition et un récapitulatif complet des règles dans nos ouvrages consacrés au C++ (publiés également aux éditions Eyrolles). Signalons simplement que la recherche d'une fonction surdéfinie peut faire intervenir toutes les conversions usuelles (promotions numériques et conversions standards, ces dernières pouvant être dégradantes), ainsi que les conversions définies par l'utilisateur en cas d'argument de type classe, à condition qu'aucune ambiguïté n'apparaisse.

Les fonctions en ligne

Une fonction en ligne (on dit aussi « développée ») est une fonction dont les instructions sont incorporées par le compilateur (dans le module objet correspondant) à chaque appel. Cela évite la perte de temps nécessaire à un appel usuel (changement de contexte, copie des valeurs des arguments sur la « pile »...) ; en revanche, les instructions en question sont générées plusieurs fois.

Une fonction en ligne est nécessairement définie en même temps qu'elle est déclarée (elle ne peut plus être compilée séparément) et son en-tête est précédée du mot-clé `inline`, comme dans :

```
inline fct ( ...) {  ..... }
```

Exercice 24

Énoncé

Quelle modification faut-il apporter au programme suivant pour qu'il devienne correct :

```
#include <iostream>
using namespace std ;
main()
{  int n, p=5 ;
   n = fct (p) ;
   cout << "p = " << p << " n = " << n ;
}
int fct (int r)
{  return 2*r ;
}
```

Solution

La fonction `fct` est utilisée dans la fonction `main`, sans être encore « connue » du compilateur, ce qui provoque une erreur de compilation. Pour y rémédier, on dispose de deux possibilités :

- déclarer la fonction avant son utilisation dans `main`, de préférence au début :

```
#include <iostream>
using namespace std ;
main()
{  int fct (int) ;   // déclaration de fct ; on pourrait écrire int fct
(int x)
   int n, p=5 ;
   n = fct (p) ;
   cout << "p = " << p << " n = " << n ;
}
int fct (int r)
{  return 2*r ;
}
```

- placer la définition de la fonction avant celle de la fonction `main` :

```
#include <iostream>
using namespace std ;
int fct (int r)
{  return 2*r ;
}
main()
{  int n, p=5 ;
   n = fct (p) ;
   cout << "p = " << p << " n = " << n ;
}
```

Il est conseillé d'utiliser la première démarche qui a le mérite de permettre de modifier les emplacements des définitions de fonctions dans un fichier source ou, même, de le scinder en plusieurs parties (compilation séparée).

Exercice 25

Énoncé

Écrire :

- une fonction, nommée `f1`, se contentant d'afficher « `bonjour` » (elle ne possédera aucun argument, ni valeur de retour) ;
- une fonction, nommée `f2`, qui affiche « `bonjour` » un nombre de fois égal à la valeur reçue en argument (`int`) et qui ne renvoie aucune valeur ;
- une fonction, nommée `f3`, qui fait la même chose que `f2`, mais qui, de plus, renvoie la valeur (`int`) `0`.

Écrire un petit programme appelant successivement chacune de ces 3 fonctions, après les avoir convenablement déclarées (on ne fera aucune hypothèse sur les emplacements relatifs des différentes fonctions composant le fichier source).

Solution

L'énoncé ne précisant rien, nous utiliserons une transmission d'arguments par valeur. Comme on n'impose pas d'ordre aux définitions des différentes fonctions dans le fichier source, on déclarera systématiquement toutes les fonctions utilisées.

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

 void f1 (void)
 {  cout << "bonjour\n" ;
 }

 void f2 (int n)
 {  int i ;
    for (i=0 ; i<n ; i++)
        cout << "bonjour\n" ;
 }

 int f3 (int n)
 {  int i ;
    for (i=0 ; i<n ; i++)
        cout << "bonjour\n" ;
    return 0 ;
 }
```

```
main()
{  void f1 (void) ;
   void f2 (int) ;
   int f3 (int) ;
   f1 () ;
   f2 (3) ;
   f3 (3) ;
}
```

Exercice 26

Énoncé

Quels résultats fournira ce programme :

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

int n=10, q=2 ;
main()
{
   int fct (int) ;
   void f (void) ;
   int n=0, p=5 ;
   n = fct(p) ;
   cout << "A : dans main, n = " << n << " p = " << p
        << " q = " << q << "\n" ;
   f() ;
}

int fct (int p)
{
   int q ;
   q = 2 * p + n ;
   cout << "B : dans fct,  n = " << n << " p = " << p
        << " q = " << q << "\n" ;
   return q ;
}

void f (void)
{
   int p = q * n ;
   cout << "C : dans f,    n = " << n << " p = " << p
        << " q = " << q << "\n" ;
}
```

Solution

```
B : dans fct,  n = 10, p = 5, q = 20
A : dans main, n = 20, p = 5, q = 2
C : dans f,    n = 10, p = 20, q = 2
```

Exercice 27

Énoncé

Écrire une fonction qui reçoit en arguments 2 nombres flottants et un caractère, et qui fournit un résultat correspondant à l'une des 4 opérations appliquées à ses deux premiers arguments, en fonction de la valeur du dernier, à savoir : addition pour le caractère +, soustraction pour -, multiplication pour * et division pour / (tout autre caractère que l'un des 4 cités sera interprété comme une addition). On ne tiendra pas compte des risques de division par zéro.

Écrire un petit programme (`main`) utilisant cette fonction pour effectuer les 4 opérations sur les 2 nombres fournis en donnée.

Solution

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

float oper (float v1, float v2, char op)
{   float res ;
    switch (op)
    { case '+' : res = v1 + v2 ;
                 break ;
      case '-' : res = v1 - v2 ;
                 break ;
      case '*' : res = v1 * v2 ;
                 break ;
      case '/' : res = v1 / v2 ;
                 break ;
      default  : res = v1 + v2 ;
    }
    return res ;
}

main()
{   float oper (float, float, char) ;      // déclaration de oper
    float x, y ;

    cout << "donnez deux nombres réels : " ;
    cin >> x >> y ;

    cout << "leur somme est :      " << oper (x, y, '+') << "\n" ;
    cout << "leur différence est : " << oper (x, y, '-') << "\n" ;
    cout << "leur produit est :    " << oper (x, y, '*') << "\n" ;
    cout << "leur quotient est :   " << oper (x, y, '/') << "\n" ;
}
```

Exercice 28

Énoncé

Transformer le programme (fonction + `main`) écrit dans l'exercice précédent de manière que la fonction ne dispose plus que de 2 arguments, le caractère indiquant la nature de l'opération à effectuer étant précisé, cette fois, à l'aide d'une variable globale.

Solution

```
#include <iostream>
using namespace std ;
char op ;      // variable globale pour la nature de l'opération
               //  attention : doit être déclarée avant d'être utilisée

float oper (float v1, float v2)
{   float res ;
    switch (op)
    { case '+' : res = v1 + v2 ;
                 break ;
      case '-' : res = v1 - v2 ;
                 break ;
      case '*' : res = v1 * v2 ;
                 break ;
      case '/' : res = v1 / v2 ;
                 break ;
      default  : res = v1 + v2 ;
    }
    return res ;
}

main()
{   float oper (float, float) ;      /* prototype de oper */
    float x, y ;

    cout << "donnez deux nombres réels : " ;
    cin >> x >> y ;

    op = '+' ;
    cout << "leur somme est :      " << oper (x, y) << "\n" ;
    op = '-' ;
    cout << "leur différence est : " << oper (x, y) << "\n" ;
    op = '*' ;
    cout << "leur produit est :    " << oper (x, y) << "\n" ;
    op = '/' ;
    cout << "leur quotient est :   " << oper (x, y) << "\n" ;
}
```

Remarque

Il s'agissait ici d'un exercice d'« école » destiné à forcer l'utilisation d'une variable globale. Dans la pratique, on évitera autant que possible ce genre de programmation qui favorise trop les risques d'« effets de bord ».

Exercice 29

Énoncé

Écrire une fonction, sans argument ni valeur de retour, qui se contente d'afficher, à chaque appel, le nombre total de fois où elle a été appelée sous la forme :

appel numéro 3

Solution

La meilleure solution consiste à prévoir, au sein de la fonction en question, une variable de classe statique. Elle sera initialisée une seule fois à zéro (ou à toute autre valeur éventuellement explicitée) au début de l'exécution du programme. Ici, nous avons, de plus, prévu un petit programme d'essai.

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

void fcompte (void)
{
   static int i ;     // il est inutile, mais pas interdit, d'écrire i=0
   i++ ;
   cout << "appel numéro " << i << "\n"  ;
}

     /* petit programme d'essai de fcompte */
main()
{  void fcompte (void) ;
   int i ;
   for (i=0 ; i<3 ; i++) fcompte () ;
}
```

Là encore, la démarche consistant à utiliser comme compteur d'appels une variable globale (qui devrait alors être connue du programme utilisateur) est à proscrire.

Exercice 30

Énoncé

Écrire 2 fonctions à un argument entier et une valeur de retour entière permettant de préciser si l'argument reçu est multiple de 2 (pour la première fonction) ou multiple de 3 (pour la seconde fonction).

Utiliser ces deux fonctions dans un petit programme qui lit un nombre entier et qui précise s'il est pair, multiple de 3 et/ou multiple de 6, comme dans cet exemple (il y a deux exécutions) :

```
donnez un entier : 9
il est multiple de 3

   ---------------
donnez un entier : 12
il est pair
il est multiple de 3
il est divisible par 6
```

Solution

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

int mul2 (int n)
{  if (n%2) return 0 ;
       else return 1 ;
}

int mul3 (int n)
{  if (n%3) return 0 ;
       else return 1 ;
}

main()
{  int mul2 (int) ;
   int mul3 (int) ;
   int n ;
   cout << "donnez un entier : " ;
   cin >> n ;
   if (mul2(n))            cout << "il est pair\n" ;
   if (mul3(n))            cout << "il est multiple de 3\n" ;
   if (mul2(n) && mul3(n)) cout << "il est divisible par 6\n" ;
}
```

Exercice 31

Énoncé

Écrire une fonction permettant d'ajouter une valeur fournie en argument à une variable fournie également en argument. Par exemple, l'appel (n et p étant entiers) :

```
ajouter (2*p+1, n) ;
```

ajoutera la valeur de l'expression 2*p+1 à la variable n.
Écrire un petit programme de test de la fonction.

Solution

Étant donné que la fonction doit être en mesure de modifier la valeur de son second argument, il est nécessaire que ce dernier soit transmis par référence.

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

void ajoute (int exp, int & var)
{ var += exp ;
  return ;
}
main()
{ void ajoute (int, int &) ;
  int n = 12 ;
  int p = 3 ;
  cout << "Avant, n = " << n << "\n" ;
  ajoute (2*p+1, n) ;
  cout << "Après, n = " << n << "\n" ;
}
```

Exercice 32

Énoncé

Soient les déclarations suivantes :

```
int fct (int) ;          // fonction I
int fct (float) ;        // fonction II
void fct (int, float) ;  // fonction III
void fct (float, int) ;  // fonction IV
int n, p ;
float x, y ;
char c ;
double z ;
```

Les appels suivants sont-ils corrects et, si oui, quelles seront les fonctions effectivement appelées et les conversions éventuellement mises en place ?

a. `fct (n) ;`

b. `fct (x) ;`

c. `fct (n, x) ;`

d. `fct (x, n) ;`

e. `fct (c) ;`

f. `fct (n, p) ;`

g. `fct (n, c) ;`

h. `fct (n, z) ;`

i. `fct (z, z) ;`

Solution

Les cas **a**, **b**, **c** et **d** ne posent aucun problème. Il y a respectivement appel des fonctions I, II, III et IV, sans qu'aucune conversion d'argument ne soit nécessaire.

e. Appel de la fonction I, après conversion de la valeur de `c` en `int`.

f. Appel incorrect, compte tenu de son ambiguïté ; deux possibilités existent en effet : conserver `n`, convertir `p` en `float` et appeler la fonction III ou, au contraire, convertir `n` en `float`, conserver `p` et appeler la fonction IV.

g. Appel de la fonction III, après conversion de `c` en `float`.

h. Appel de la fonction III, après conversion (dégradante) de `z` en `float`.

i. Appel incorrect, compte tenu de son ambiguïté ; deux possibilités existent en effet : convertir le premier argument en `float` et le second en `int` et appeler la fonction III ou, au contraire, convertir le premier argument en `int` et le second en `float` et appeler la fonction IV.

Exercice 33

Énoncé

a. Transformer le programme suivant pour que la fonction `fct` devienne une fonction en ligne.

```
#include <iostream>
using namespace std ;
main()
{  int fct (char, int) ;            // déclaration (prototype) de fct
   int n = 150, p ;
   char c = 's' ;
   p = fct ( c , n) ;
   cout << "fct (\'" << c << "\', " << n << ") vaut : " << p ;
}
int fct (char c, int n)             // définition de fct
{  int res ;
   if (c == 'a')      res = n + c ;
   else if (c == 's') res = n - c ;
   else               res = n * c ;
   return res ;
}
```

b. Comment faudrait-il procéder si l'on souhaitait que la fonction `fct` soit compilée séparément ?

Solution

a. Nous devons donc d'abord déclarer (et définir en même temps) la fonction `fct` comme une fonction en ligne. Le programme `main` s'écrit de la même manière, si ce n'est que la déclaration de `fct` n'y est plus nécessaire puisqu'elle apparaît auparavant (il reste permis de la déclarer, à condition de ne pas utiliser le qualificatif `inline`).

```
#include <iostream>
using namespace std ;
inline int fct (char c, int n)
{   int res ;
    if (c == 'a')      res = n + c ;
    else if (c == 's') res = n - c ;
    else               res = n * c ;
    return res ;
}
main ()
{   int n = 150, p ;
    char c = 's' ;
    p = fct (c, n) ;
    cout << "fct (\'" << c << "\', " << n << ") vaut : " << p ;
}
```

b. Il s'agit en fait d'une question piège. En effet, la fonction `fct` étant en ligne, elle ne peut plus être compilée séparément. Il est cependant possible de la conserver dans un fichier d'extension `h` et d'incorporer simplement ce fichier par `#include` pour compiler le `main`. Cette démarche se rencontrera d'ailleurs fréquemment dans le cas de classes comportant des fonctions en ligne. Alors, dans un fichier d'extension `h`, on trouvera la déclaration de la classe en question, à l'intérieur de laquelle apparaîtront les « déclarations-définitions » des fonctions en ligne.