

Cours 6

Les fonctions

Vincent Lepetit

`vincent.lepetit@epfl.ch`

Intérêt des fonctions

Un programme doit souvent réaliser plusieurs fois une même tâche.

Les **fonctions** permettent d'éviter d'avoir à écrire plusieurs fois le code pour cette tâche: Elles regroupent les instructions nécessaires à la tâche, et peuvent être utilisées à plusieurs endroits du programme.

De plus, les fonctions peuvent être **paramétrées**: elles peuvent s'adapter à des tâches semblables mais pas complètement identiques.

Par exemple:

- une même fonction pourra être utilisée pour afficher des tableaux différents;
- les fonctions mathématiques sont des fonctions faisant partie de la bibliothèque standard de fonctions, ce qui évite d'avoir à les coder soi-même.

Dans l'exemple:

```
float c = cos(3.14);
```

3.14 est un paramètre pour la fonction `cos`.

Attention, le nom de fonction est trompeur, et ne correspond pas tout à fait à la notion de fonction mathématique:

- une fonction C ou C++ peut ne pas retourner de valeur;
- elle peut également (sous certaines conditions) modifier les valeurs des variables passées en paramètres.

Premier exemple

Nous allons commencer par des fonctions à un paramètre, proches du sens mathématique du terme.

Nous verrons ensuite des situations plus générales: fonction à plusieurs paramètres, fonction sans paramètre, fonction sans résultat...

Commençons par une *fonction qui calcule le cube* d'une valeur flottante.

Cette fonction

- possède un unique **paramètre** (on dit aussi **argument**);
- fournit une valeur en *résultat*.

Il faut distinguer:

- La **définition** de la fonction: c'est-à-dire la définition du nom de la fonction (ici nous choisirons `cube`), du type de son paramètre (ici `float`), du type de la valeur qu'elle calcule (ici `float` également), les instructions qui la constituent;
- Son **utilisation** (on dit aussi *l'appel* à la fonction).

Définition d'une fonction

Voici comment définir la fonction `cube`:

```
float cube(float x)
```

← En-tête de la fonction

```
{  
    float y;  
  
    y = x * x * x;  
  
    return y;  
}
```

← Corps de la fonction

En-tête d'une fonction

cube: nom de la fonction



float cube (float x)



float: définit le type du résultat de la fonction.

(float x): type et nom du paramètre.

Le nom `x` désigne la valeur du paramètre qui sera reçu par la fonction `cube`.

Ce nom est choisi librement comme n'importe quel nom de variable.

Corps d'une fonction

```
float cube(float x)
{
    float y;

    y = x * x * x;

    return y;
}
```

Déclaration d'une variable `y`: la fonction utilise une variable `y` de type `float`.
Cette déclaration est comme celles que nous avons vues jusqu'ici.

`y` est appelée une **variable locale**;
`y` ne peut être utilisée que dans la fonction `cube`.

Affectation classique.

`return y`; précise la valeur qui est fournie (on dit aussi renvoyée ou retournée) par la fonction.

Mise en œuvre

Voici un exemple de programme qui définit et utilise la fonction `cube`:

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

```
float cube(float x)
{
    float y;

    y = x * x * x;

    return y;
}
```

La fonction doit être définie avant son utilisation.

```
int main(int argc, char **argv)
{
    float c;

    c = cube(2);
    cout << "c vaut " << c << endl;

    c = 2 * cube(3);
    cout << "c vaut " << c << endl;

    cout << "Le cube de 2.2 est " << cube(2.2) << endl;

    return 0;
}
```

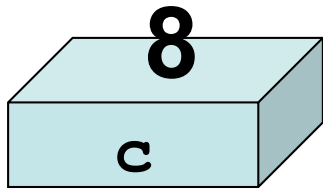
Le programme peut utiliser la fonction autant de fois qu'on veut.

Pas-à-pas

```
float cube(float x)
{
    float y;

    y = x * x * x;

    return y;
}
```



```
int main(int argc, char
{
    // ...

    → c = 2 * cube(3);

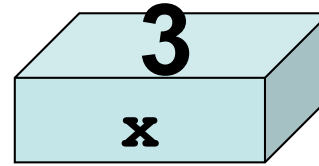
    // ...
}
```

1. L'expression

2 * cube(3)

est évaluée.

→ La fonction `cube` est appelée.



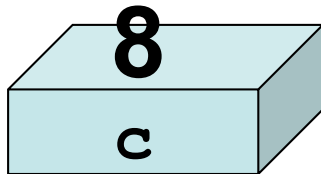
2. Le paramètre `x` est créé.
La valeur 3 est copiée dans `x`.

→

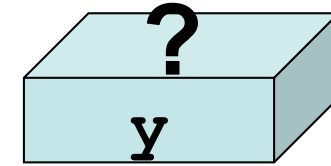
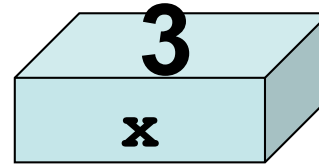
```
float cube(float x)
{
    float y;

    y = x * x * x;

    return y;
}
```

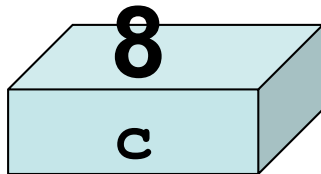


```
int main(int argc, char ** argv)
{
    // ...
    → c = 2 * cube(3);
    // ...
}
```



```
float cube(float x)
{
  float y;
  y = x * x * x;
  return y;
}
```

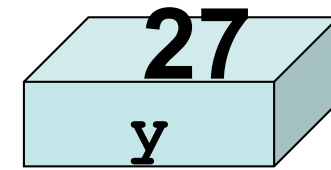
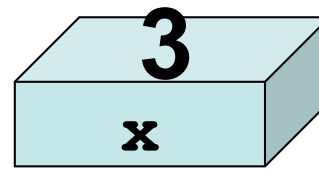
3. La variable locale *y* est créée.



```
int main(int argc, char ** argv)
{
  // ...

  c = 2 * cube(3);

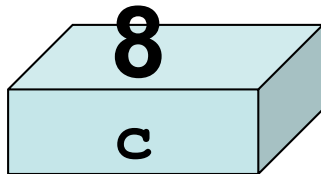
  // ...
}
```



```
float cube(float x)
{
    float y;
    y = x * x * x;
    return y;
}
```



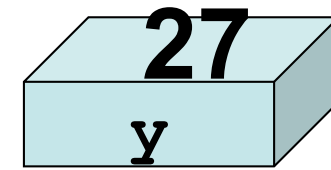
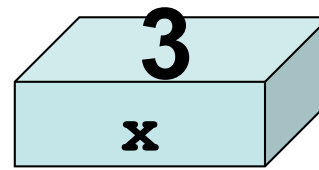
4. La valeur de $x * x * x$ est affectée à y .



```
int main(int argc, char ** argv)
{
    // ...

    c = 2 * cube(3);

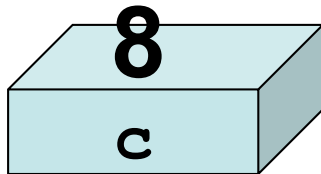
    // ...
}
```



```
float cube(float x)
{
    float y;
    y = x * x * x;
    return y;
}
```



5. La fonction retourne la valeur de y (27).



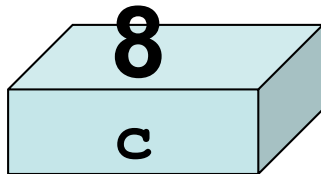
```
int main(int argc, char ** argv)
{
    // ...
    c = 2 * cube(3);
    // ...
}
```

```
float cube(float x)
{
    float y;

    y = x * x * x;

    return y;
}
```

6. On sort de la fonction: x et y sont supprimés...



```
int main(int argc, char ** argv)
{
    // ...

    c = 2 * cube(3);

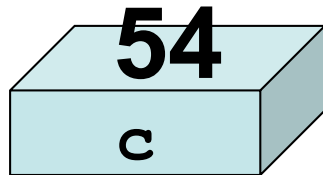
    // ...
}
```

...et `cube(3)` est remplacée par la valeur retournée par la fonction (27).

```
float cube(float x)
{
    float y;

    y = x * x * x;

    return y;
}
```



```
int main(int argc, char ** argv)
{
    // ...
    → c = 2 * cube(3);
    // ...
}
```

7. La valeur $2 * 27$ est affectée à `c`.

Une fois qu'on a compris que la fonction `cube` calculait effectivement le cube de la valeur passée en paramètre, on n'a plus à exécuter pas-à-pas la fonction pour comprendre ce que fait l'instruction

```
c = 2 * cube(3) ;
```

on sait que l'expression à droite du signe `=` vaut 2×3^3

Vocabulaire

"Appeler la fonction *cube*":

```
c = 2 * cube(3);
```

= utiliser la fonction `cube`

"3 est passé en paramètre":

```
c = 2 * cube(3);
```

→ Cela signifie que la valeur 3 est copiée dans le paramètre `x`.

"la fonction retourne la valeur de *y*", "la fonction retourne le cube de *x*":

```
    return y;
```

```
}
```

...

```
c = 2 * cube(3);
```

→ Cela signifie que `cube(3)` sera remplacée par la valeur de `y`.

`cos(0)` retourne le cosinus de 0.

`srand` est également une fonction, qui ne retourne pas de valeur.

Portée des variables

On peut utiliser des noms quelconques pour les paramètres et les variables locales, sans se préoccuper des autres noms qui apparaissent ailleurs.

Par exemple, `main` et `cube` peuvent utiliser toutes les deux une variable appelée `c`.

Ce sont deux variables différentes.

On dit que la *portée* des variables locales (ici `c`) et des paramètres (ici `x`) est limitée à la fonction où ils sont définis.

```
float cube(float x)
{
    float c;
    c = x * x * x;
    return c;
}
```

`c` fait ici référence à la variable locale de la fonction `cube`, déclarée là

```
int main(int argc, char ** argv)
{
    float c;
    c = cube(2);
}
```

`c` fait ici référence à la variable locale de la fonction `main`, déclarée là

Portée des variables

```
float cube(float x)
{
    float c;
    c = x * x * x;
    return c;
}

int main(int argc, char ** argv)
{
    float x;
    ...
}
```

x fait ici référence au paramètre de la fonction cube, défini là

L'instruction `return`

L'instruction `return` peut retourner n'importe quelle expression, pas seulement une simple variable. Par exemple, la fonction `cube` aurait pu être définie de la façon suivante:

```
float cube(float x)
{
    return x * x * x;
}
```

L'instruction `return` fait deux choses:

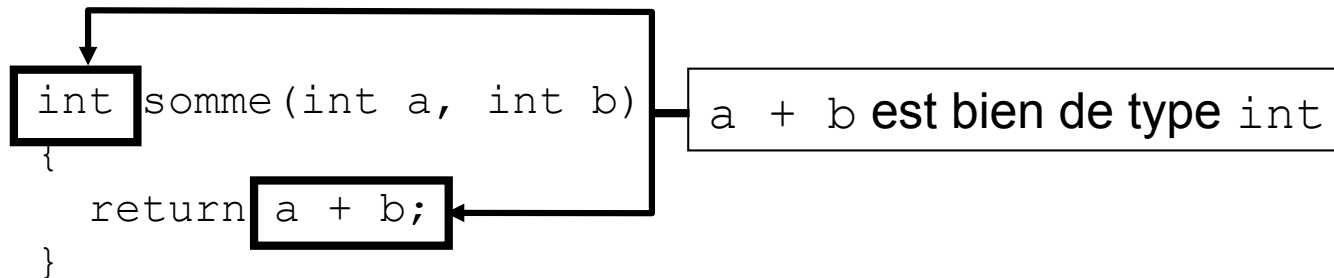
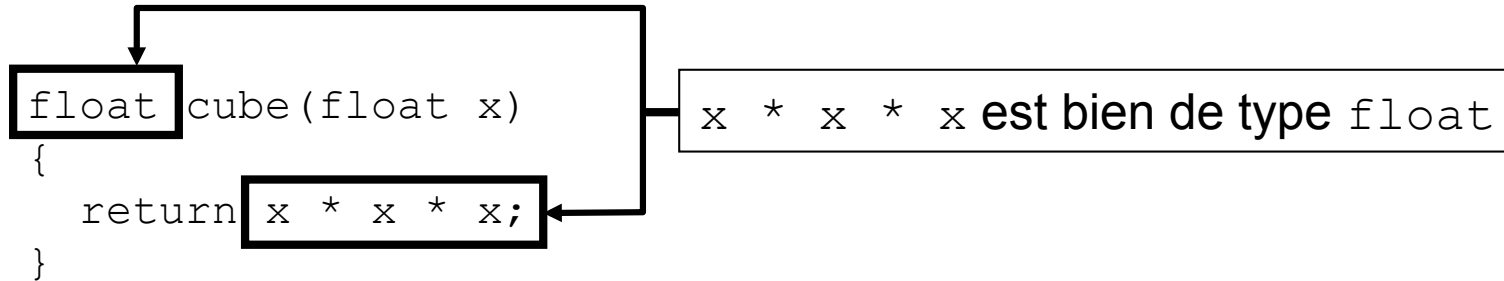
- elle précise la valeur qui sera fournie par la fonction en résultat;
- `return` met *fin à l'exécution* des instructions de la fonction.

Quand une fonction ne fournit aucun résultat (`void f(int x)`, voir plus loin):

- soit on ne place aucun `return` dans sa définition;
- soit on utilise l'instruction `return` sans la faire suivre d'une expression: `return;`

L'instruction `return`

La valeur après l'instruction `return` est forcément du type placé avant le nom de la fonction:



L'instruction `return`

Il est possible de placer plusieurs instructions `return` dans une même fonction.

Par exemple, une fonction déterminant le maximum de deux valeurs peut s'écrire avec une instruction `return`:

```
float max2(float a, float b)
{
    float m;
    if (a > b)
        m = a;
    else
        m = b;

    return m;
}
```

ou deux:

```
float max2(float a, float b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

LES FONCTIONS PRECEDENTES `cube` et `max2` UTILISAIENT UNE VARIABLE LOCALE POUR SIMPLIFIER L'INTRODUCTION DES FONCTIONS.

DES MAINTENANT, ECRIVEZ DE TELLES FONCTIONS SUR CE MODELE:

```
float cube(float x)
{
    return x * x * x;
}
```

```
float max2(float a, float b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

La fonction `main()`

`main` est aussi une fonction, avec un nom imposé.

Par convention, tout programme C doit avoir une fonction `main`, qui est appelée automatiquement quand on exécute le programme.

Cette fonction doit retourner une valeur de type `int`. La valeur 0 indique par convention que le programme s'est bien déroulé.

Les variables que l'on déclare dans la fonction `main` sont donc elles aussi des variables locales.

Dans l'exemple:

```
int main(int argc, char ** argv)
{
    float c;
    ...
}
```

la variable `c` est une *variable locale de la fonction `main`*.

Fonctions à plusieurs paramètres

La fonction `cube` n'avait qu'un paramètre. Voici la définition d'une fonction, nommée `max`, qui fournit en résultat la plus grande des trois valeurs entières reçues en paramètres:

```
int max3(int a, int b, int c)
{
    int m = a;

    if (m < b)
        m = b;

    if (m < c)
        m = c;

    return m;
}
```

Cette fois, l'en-tête définit 3 paramètres `a`, `b` et `c`.

La fonction `max3` utilise une variable locale, nommée `m`, qui servira à déterminer le maximum des valeurs passées en paramètre et contenues par `a`, `b` et `c`.

Fonctions à plusieurs paramètres

Exemple de programme utilisant la fonction `max`:

```
int max3(int a, int b, int c)
{
    int m = a;

    if (m < b)
        m = b;

    if (m < c)
        m = c;

    return m;
}

int main(int argc, char ** argv)
{
    int m, n, p, q;

    cout << max3(3, 5, 2) << endl;

    n = 3; p = 5; q = 7;
    cout << max3(n, p, q) << endl;
    ...
}
```

Qu'affichent ces programmes ?

A:

```
float f(float m, float p)
{
    float y = m / p;
    return y * y;
}
```

...

```
cout << f(16, 2) << endl;
float m = f(2, 1), p = f(8, 2), y = f(15, 3);
cout << m << " " << p << " " << y << endl;
y = f(p, m);
cout << y << endl;
```

B:

```
float a(float m, float p)
{
    return m + p;
}
```

```
float b(float m, float p)
{
    return m * p;
}
```

```
float c(float m)
{
    return b(m, m);
}
```

...

```
cout << a(5, 2) << " " << b(5, 2) << endl;
cout << a( b(3, 2), a(5, 4) ) << endl;
cout << c(5) << endl;
```

Exercices

Écrivez l'en-tête puis le corps des fonctions suivantes. Pour trouver l'en-tête, il faut déterminer:

- les paramètres de la fonction, et leur type;
- le type de la fonction.

- Fonction de nom `f` qui a 1 paramètre `x`, et qui renvoie la valeur de $\sin(x) + \exp(x)$.
- Fonction de nom `degre2` qui a 4 paramètres `a`, `b`, `c` et `x`, de type `float`, et qui retourne la valeur de l'expression ax^2+bx+c
- Fonction `fact` qui renvoie la factorielle d'un nombre entier `n` passé en paramètre, c'est-à-dire le produit des entiers de 1 à `n`.
- Fonction nommée `coeff_binomial` qui calcule le coefficient binomial de 2 nombres `n` et `m` passés en paramètre, à l'aide de la formule :

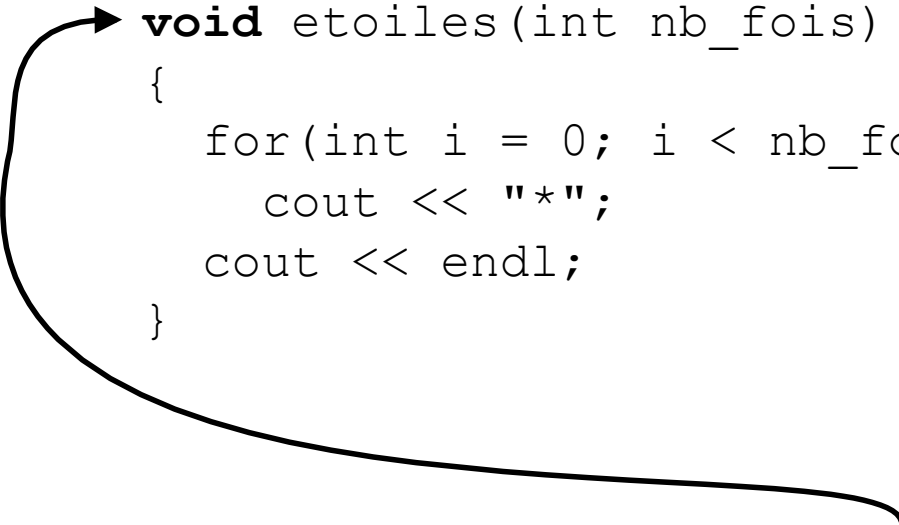
$$C_n^m = \frac{n!}{(n-m)!m!}$$

$n!$, $m!$ et $(n-m)!$ sont respectivement les factorielles de n , m et $(n-m)$. `coeff_binomial` devra appeler la fonction `fact` de la question précédente pour calculer ces valeurs.

void: Fonctions sans résultat

Une fonction C ou C++ peut ne pas retourner de résultat. Voici un exemple:

```
void etoiles(int nb_fois)
{
    for(int i = 0; i < nb_fois; i++)
        cout << "*";
    cout << endl;
}
```



L'en-tête commence par le mot `void` qui indique que la fonction ne retourne pas de résultat.

Même si une telle fonction ne calcule pas de valeur, elle peut néanmoins réaliser des actions, comme ici un affichage.

return dans des fonctions définies avec void

Quand la fonction est définie avec `void` on utilise soit `return` seul, soit rien:

```
void affiche_racine(float a)
{
    if (a < 0)
        return;
    cout << sqrt(a);
}
```

← Grâce à ce `return`, on quitte la fonction avant de calculer `sqrt(a)` si `a` est négatif.

Appel des fonctions sans résultat

On n'appelle pas les fonctions de type `void` de la même façon que les autres:

```
x = etoiles(3); // incorrect !!!
```

```
cout << etoiles(3) << endl; // incorrect !!!
```

Il suffit d'appeler la fonction ainsi:

```
etoiles(3);
```

Fonctions sans résultat: exemple

```
void etoiles(int nb_fois)
{
    for(int i = 0; i < nb_fois; i++)
        cout << "*";
    cout << endl;
}

int main(int argc, char ** argv)
{
    for(int i = 0; i < 5; i++)
        etoiles(i);

    return 0;
}
```

```
*
**
***
****
```

Les paramètres formels et les paramètres effectifs

- Les paramètres figurant dans l'en-tête d'une fonction se nomment des ***paramètres formels***:

```
int max(int a, int b, int c)
```

- Les paramètres fournis lors de l'appel de la fonction se nomment des ***paramètres effectifs***:

```
m = max(10, 3 * n, p + q);
```

Passage par valeur

En langage C, les paramètres sont transmis *par valeur*.

Considérons le programme suivant:

```
void mis_a_zero(int a)
{
    cout << "avant: a = " << a << endl;
    a = 0;
    cout << "apres: a = " << a << endl;
}

int main(int argc, char ** argv)
{
    int n = 10;

    cout << "avant appel: n = " << n << endl;

    mis_a_zero(n);

    cout << "apres appel: n = " << n << endl;
}
```

Ce programme affiche:

```
avant appel: n = 10
avant: a = 10
apres: a = 0
apres appel: n = 10
```

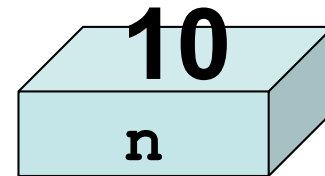
On a bien changé la valeur de `a`,
mais pas de `n` !!!

Pas-à-pas

```
void mis_a_zero(int a)
{
    cout << "avant: a = " << a << endl;
    a = 0;
    cout << "apres: a = " << a << endl;
}

int main(int argc, char ** argv)
{
    int n = 10;

    cout << "avant appel: n = " << n << endl;
    → mis_a_zero(n);
    cout << "apres appel: n = " << n << endl;
}
```



Pas-à-pas

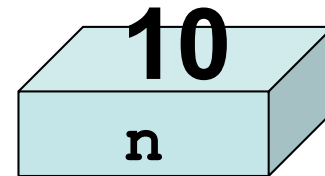
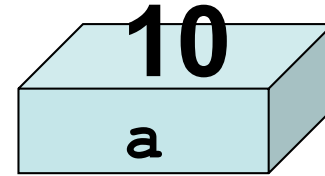
```
→ void mis_a_zero(int a)
{
    cout << "avant: a = " << a << endl;
    a = 0;
    cout << "apres: a = " << a << endl;
}

int main(int argc, char ** argv)
{
    int n = 10;

    cout << "avant appel: n = " << n << endl;

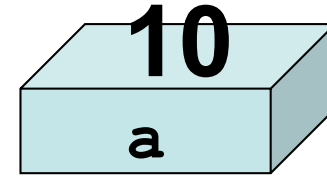
    → mis_a_zero(n);

    cout << "apres appel: n = " << n << endl;
}
```



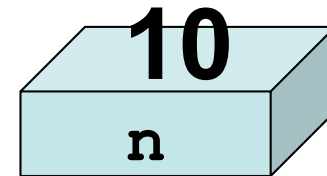
Pas-à-pas

```
void mis_a_zero(int a)
{
    cout << "avant: a = " << a << endl;
    → a = 0;
    cout << "apres: a = " << a << endl;
}
```



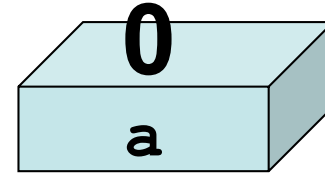
```
int main(int argc, char ** argv)
{
    int n = 10;

    cout << "avant appel: n = " << n << endl;
    → mis_a_zero(n);
    cout << "apres appel: n = " << n << endl;
}
```



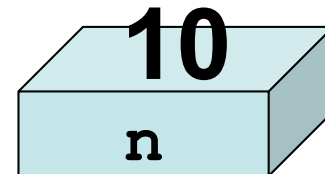
Pas-à-pas

```
void mis_a_zero(int a)
{
    cout << "avant: a = " << a << endl;
    → a = 0;
    cout << "apres: a = " << a << endl;
}
```



```
int main(int argc, char ** argv)
{
    int n = 10;

    cout << "avant appel: n = " << n << endl;
    → mis_a_zero(n);
    cout << "apres appel: n = " << n << endl;
}
```



Pas-à-pas

```
void mis_a_zero(int a)
{
    cout << "avant: a = " << a << endl;
    a = 0;
    cout << "apres: a = " << a << endl;
}
```

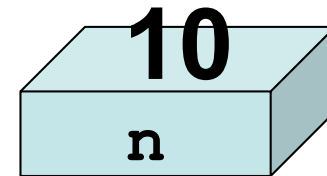
```
int main(int argc, char ** argv)
{
    int n = 10;
```

```
    cout << "avant appel: n = " << n << endl;
```

```
    mis_a_zero(n);
```

```
    cout << "apres appel: n = " << n << endl;
```

```
}
```



Même si la fonction a changé les valeurs du paramètre formel `a`, le changement n'est pas répercuté sur le paramètre effectif `n`.

Un autre usage de `void`: Fonctions sans paramètres

Une fonction peut ne pas nécessiter de paramètre.

Son en-tête doit alors comporter le mot `void` à la place de la liste de paramètres, par exemple:

```
void bonjour(void)  
{  
    cout << "Bonjour" << endl;  
}
```

L'appel d'une fonction sans paramètre doit néanmoins comporter des *parenthèses vides*:

```
bonjour ();
```

Les variables globales

En langage C, il est (malheureusement) possible de définir des variables globales. Il s'agit de variables ou de tableaux qui sont accessibles par toutes les fonctions du programme.

Voici un exemple de programme faisant intervenir une variable globale:

```
int nb_fois; // nb_fois est ici une variable globale
```

```
void bonjour(void)
{
    for(int i = 0; i < nb_fois; i++)
        cout << "bonjour" << endl;
}

int main(int argc, char ** argv)
{
    nb_fois = 2; bonjour();
    nb_fois = 3; bonjour();
}
```

Ce programme affiche:

```
bonjour
bonjour
bonjour
bonjour
bonjour
```

Attention à l'utilisation des variables globales

Qu'affiche ce programme ?

```
#include <stdio.h>

int i; // i est ici une variable globale

void fonction1(void)
{
    for(i = 0; i < 3; i++)
        cout << "fonction1: " << i << endl;
}

int main(int argc, char ** argv)
{
    for(i = 0; i < 2; i++)
        fonction1();
}
```

1. *i* est initialisée à 0, et `fonction1` est appelée.
2. La boucle dans `fonction1` modifie la variable globale *i*. Quand `fonction1` est terminée, *i* contient 3.
3. Au retour dans la boucle de `main`, *i* contient toujours 3. Donc on sort de la boucle `for(i = 0; i < 2; i++)`

Donc, le programme affiche:

```
fonction1: 0
fonction1: 1
fonction1: 2
et c'est tout...
```

N'utilisez pas de variables globales !

Tableaux transmis en paramètre

Le langage C permet de transmettre un tableau en paramètre d'une fonction.

Dans ce cas, ***la fonction peut modifier les valeurs du tableau.*** Nous verrons pourquoi dans la suite du cours.

Par exemple, cette fonction initialise à 0 tous les éléments du tableau passé en paramètre:

```
void mise_a_zero(int v[5])
{
    for(int i = 0; i < 5; i++)
        v[i] = 0;
}
```

Cette fonction peut alors être utilisée de la façon suivante:

```
int main(int argc, char ** argv)
{
    int t1[5];

    mise_a_zero(t1);

    // ...
}
```

Tableaux transmis en paramètre

La fonction `mise_a_zero` précédente avait comme en-tête:

```
void mise_a_zero(int v[5])
```

Ici, la définition du tableau comme paramètre (`int v[5]`) est donc similaire à la déclaration d'un tableau.

En fait, plusieurs notations sont admises. On aurait pu écrire:

```
void mise_a_zero(int * v)
```

ou

```
void mise_a_zero(int v[])
```

Même si on peut donner une taille de tableau dans la définition du paramètre (5 dans l'exemple), le compilateur n'en tient pas compte.

Tableaux transmis en paramètre

Dans la suite, nous utiliserons la notation avec une étoile:

```
void mise_a_zero(int * v)
```

Fonction retournant un booléen

On veut écrire la fonction d'en-tête:

```
bool tous_positifs(int * T, int nb_elements)
```

qui retourne `true` si tous les éléments de `T` sont positifs, `false` si au moins un des éléments est inférieur à 0.

Solution:

```
bool tous_positifs(int * T, int nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
        if (T[i] < 0)
            return false;

    return true;
}
```

Attention

On ne peut pas faire:

```
bool tous_positifs(int * T, in nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
        if (T[i] < 0)
            return false;
        else
            return true; // !!
}
```

→ Si le premier élément est positif, la fonction renvoie `true`, alors que la fonction n'a pas encore testé les éléments suivants.

Appel d'une fonction retournant un booléen

On peut appeler la fonction ainsi:

A éviter en général

```
if (tous_positifs(tab, 5) == false)
    cout << "Au moins un des elements est negatif."
    << endl;
```

ECRIVEZ:

```
if (!tous_positifs(tab, 5))
    cout << "Au moins un des elements est negatif."
    << endl;
```

Erreurs classiques

Il n'y a pas de point-virgule à la fin de l'en-tête de la fonction:

```
float cube(float x)
```

Si le type de la fonction est `void`, il peut ne pas y avoir de `return` dans le corps de la fonction.

Si le type est différent de `void`, il y a forcément (au moins) une instruction `return`.

Les instructions après `return` ne sont pas exécutées.

C:

```
float g(bool c, float m, float p)
{
    if (c)
        return m;
    else
        return p;
}

...
cout << g(2 < 3, 10, 100) << endl;
if (g(3 % 2 == 0, 1, -1) < 0)
    cout << "a" << endl;
else
    cout << "b" << endl;
```

D:

```
bool h(int deb, int fin, int div)
{
    for(int i = deb; i <= fin; i++)
        if (i % div == 0)
            return true;

    return false;
}

...
if ( h(7, 13, 5) )
    cout << "oui" << endl;
else
    cout << "non" << endl;
```