

Fonctions

Dans un programme : certaines opérations (ou séquences d'opérations) peuvent se répéter plusieurs fois : affichage de tableau, saisie, ou même calculs.

Exemple : si on veut illustrer le programme d'insertion classée dans un tableau : on peut afficher le tableau d'origine, le tableau avec la case libérée, et le tableau avec la valeur insérée.

Le cas se présente très souvent dans les programmes.

Plutôt que d'écrire plusieurs fois les mêmes instructions ou opérations : les écrire une seule fois, mais pouvoir les utiliser plusieurs fois.

Notion de **sous-programmes** : parties de programme que l'on peut utiliser quand on en a besoin en les **appelant**. Ce sont des **fonctions**.

Fonctions

Une fonction est un regroupement d'instructions qui effectue une certaine tâche.

Une fonction a une syntaxe et un mode de fonctionnement particuliers.

Elle a ses propres variables (ce sont des variables dites 'locales'); qui n'existent que pour la fonction.

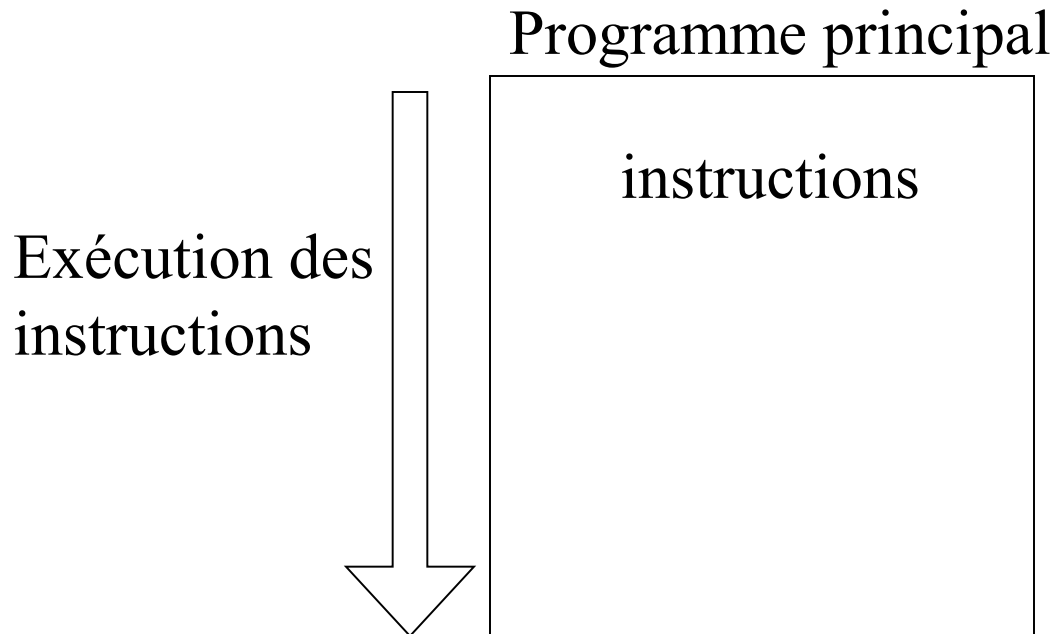
En fait, un programme est composé de :

un programme dit principal, là où débutent les instructions;

un ou des sous-programmes ou fonctions, qui seront appelées (mises en œuvre) par le programme principal ou par d'autres fonctions.

Représentation par blocs

On peut symboliser le déroulement d'un programme (sans sous-programme) par un bloc :



C'est ce qui se passe à l'exécution du programme

Représentation par blocs

Visualisation de la séquence : peu importe qu'il y ait des boucles ou des tests : on indique que l'on va du début à la fin.

Intervention d'un sous-programme : il est écrit une fois, et on peut aussi le symboliser par un bloc.

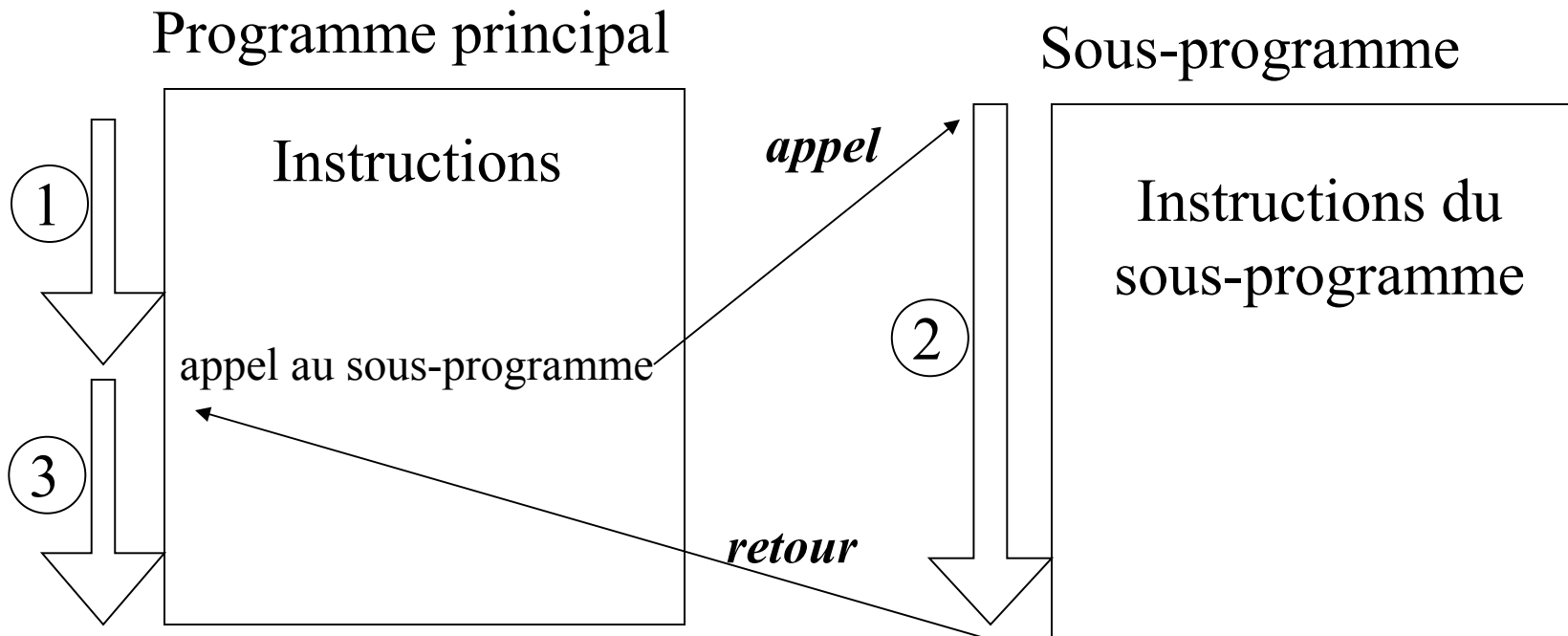
Quand est-il exécuté ?

Lorsque le programme principal fait **appel** à ce sous-programme ou fonction.

Ce sont alors les instructions du sous-programme qui sont exécutées, puis le programme principal reprend quand le sous-programme est terminé

Représentation par blocs

Illustration :



Représentation par blocs

De la même manière : on peut appeler plusieurs fois une même fonction à partir d'un programme, un programme peut appeler plusieurs fonctions différentes, une fonction peut elle-même appeler une fonction.

séparation des variables :

le programme principal a ses variables qui lui sont propres : les fonctions ne les connaissent pas;

les fonctions ont leur propre variables, que le programme (et les autres fonctions) ne connaissent pas.

Communication ?

Interfaces des fonctions

Comment écrire une fonction pour communiquer avec d'autres parties du programme ?

On pourra lui fournir des valeurs **en entrée** pour qu'elle puisse fonctionner.

Elle sera capable de fournir une valeur **en sortie** avant de se terminer : ce sera son résultat.

Pour que le programme la reconnaisse, on lui fournit aussi un nom (comme pour une variable) : même règles d'identification.

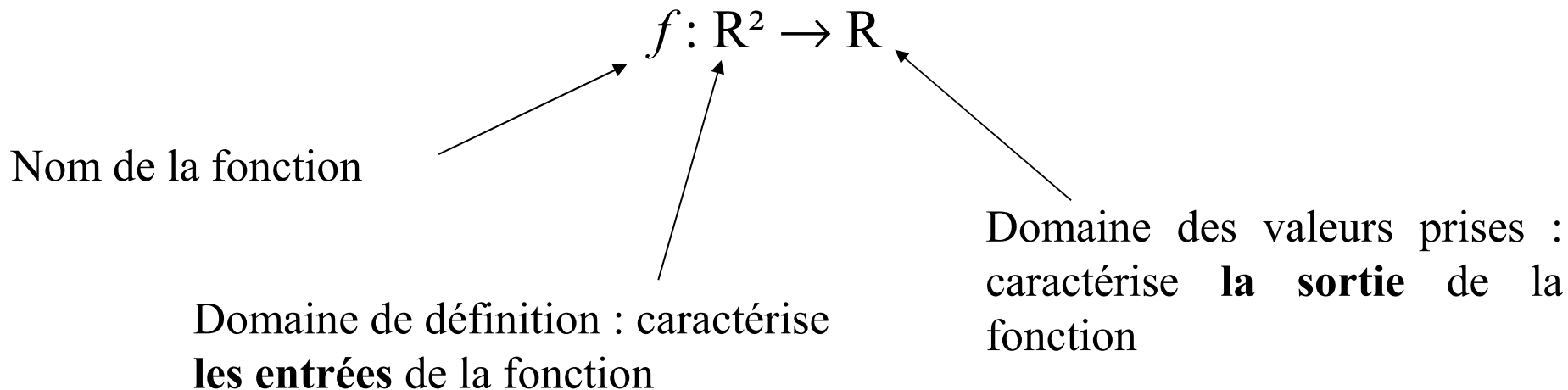
Analogie avec les fonctions mathématiques

Analogie : fonctions mathématiques

En mathématique, on définit une fonction en indiquant dans un premier temps :

- son domaine de définition,
- le domaine des valeurs prises,
- son nom.

Exemple :

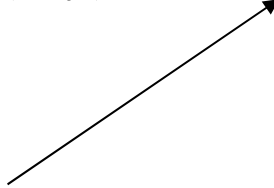


Analogie : fonctions mathématiques

Attention à ne pas pousser l'analogie trop loin : un ordinateur ne sait pas ce qu'est une fonction mathématique, il sait juste calculer, à partir de valeurs numériques, d'autres valeurs numériques avec une formule qu'on lui fournit !

En ce qui concerne ce que contient la fonction : comment utiliser les valeurs d'entrée pour arriver à calculer la sortie → des instructions.

Ex :

$$f: \mathbb{R}^2 \rightarrow \mathbb{R}$$
$$(x,y) \rightarrow z = x^2 + x.y + y^2$$


Manière de transformer les entrées en sorties : traduit en instructions dans la fonction

Utilisation et écriture

Pour utiliser une fonction dans un programme : attention aux interfaces !

Le programme n'a pas besoin de tout connaître d'une fonction : il doit savoir, dans un premier temps, comment l'**appeler** correctement :

Quelles valeurs lui donner **en entrée** ?

Quelle valeur récupérer **en sortie** ?

Quel est le **nom** de la fonction ?

Pour matérialiser : écrivons la fonction informatique correspondant au cas mathématique présenté :

Utilisation et écriture

Quelles valeurs lui donner **en entrée** ? : la fonction a deux entrées : deux nombres réels. Leur nom importe peu, c'est leur type qui est important. Informatiquement, ce seront deux valeurs de type *float* ou *double*.

Quelle valeur récupérer **en sortie** ? De même que pour les valeurs d'entrée, il faut préciser le type de valeur de sortie : ici, *float* ou *double*.

Quel est le **nom** de la fonction ? On utilise le nom que l'on veut : ici, par exemple : *fonc_calc*.

Ces 3 informations, sorte de **mode d'emploi**, forment le **prototype** d'une fonction.

Utilisation et écriture

Le **prototype** d'une fonction est l'ensemble des informations permettant de savoir **comment utiliser (appeler)** une fonction.

Un prototype ne décrit pas comment la fonction transforme les entrées en sorties, et n'est pas un appel à la fonction.

Analogie avec un outil, par exemple une perceuse : le mode d'emploi n'indique pas comment est constituée la perceuse, et ce n'est pas le mode d'emploi qui fait un trou dans un mur, mais bien la perceuse !

Syntaxe en C : un prototype s'écrit de la manière suivante

```
type_sortie    nom(type entrée 1, type entrée 2,...,type entrée N);
```

Utilisation et écriture

Suite de l'exemple mathématique :

voici le prototype de la fonction de calcul

```
float fonc_calc(float, float);
```

on le lit de la manière suivante : la fonction dont le **nom** est *fonc_calc* a besoin de deux valeurs de type *float* en **entrée**, et elle donne une valeur de type *float* en **sortie**.

Au niveau du vocabulaire : un **prototype** est aussi appelé **déclaration** de fonction.

La **sortie** s'appelle aussi **valeur de retour** de la fonction.

Écriture d'une fonction

Après le mode d'emploi, il faut maintenant écrire les instructions qui composent la fonction : comment, à partir des entrées, obtenir la sortie ou valeur de retour.

Il s'agit de la **définition** de la fonction. On doit rappeler les valeurs d'entrée, le nom et le type de la valeur de retour. La syntaxe d'une définition de fonction est la suivante :

```
type_de_retour nom(type_entrée_1 nom_entrée1, type_entrée_2 nom_entrée2,...)  
{  
    déclarations des variables utilisées par la fonction;  
  
    instructions de la fonction;  
}
```

Deux parties : entête (première ligne) et corps (le reste)

Écriture d'une fonction

Attention ! Différences entête / prototype :

- pas de point-virgule après la première ligne (entête)
- on donne cette fois-ci le nom des entrées, puisqu'elles vont servir à calculer la sortie : on leur donne le nom que l'on veut.

Exemple :

```
float fonc_calc(float val_1, float val_2)
{
    variables de la fonction;
    instructions de la fonction;
}
```

Écriture d'une fonction

Exemple : entête + corps de la fonction

```
float fonc_calc(float val_1,float val_2)
{
    /* on utilise une variable resul */
    /* pour stocker le resultat */

    float resul;

    /* on fait le calcul */

    resul = (val_1*val_1)+(val_1*val2)+(val_2*val-2);

    /* il faut maintenant préciser que resul */
    /* est la valeur que doit renvoyer la fonction */

    instruction_de_retour;
}
```


Écriture d'une fonction

Instruction de retour :

la valeur est calculée dans la fonction, il faut indiquer qu'il s'agit de la valeur de retour :

instruction **return**

indique que la fonction est terminée et indique la valeur de retour.

Syntaxe : `return valeur_de_retour;`

dans l'exemple : `return resul;`

effet : la fonction se termine, et renvoie la valeur de `resul`.

Possibilité de vérifier : on indique le type de la valeur de retour dans le prototype et l'entête : la valeur retournée doit être du même type.

Écriture d'une fonction

Sur l'exemple : la fonction `fonc_calc` doit renvoyer une valeur de type *float*. Elle renvoie `resul`, qui est de type *float*.

Cela ne signifie pas que la fonction marche bien, c'est une vérification.

Au niveau du vocabulaire : les **entrées**, lors de la **définition** (ou écriture) d'une fonction sont appelés **paramètres** de la fonction.

On peut les utiliser dans la fonction comme s'ils s'agissaient de variables déclarées dans la fonction.

Les paramètres d'une fonction n'existent pas en dehors de la fonction.

Écriture d'une fonction

Prenons un deuxième exemple pour résumer tout cela :

on veut faire une fonction calculant la moyenne de trois nombres entiers positifs.

difficultés avec les fonctions : trouver les paramètres et la valeur de retour !

En entrée de la fonction : 3 nombres entiers (type `int` de manière arbitraire);

En sortie : la moyenne : de type réel (il y aura une division) : type `float`;

le nom de la fonction : `moyenne` (arbitraire là aussi).

Écriture d'une fonction

D'où le prototype :

```
float moyenne(char, char, char);
```

et la définition :

```
float moyenne(char a, char b, char c)
{
    float resultat;
    resultat=(float) (a+b+c)/3.0;
    return(resultat);
}
```

ou encore :

```
float moyenne(char a, char b, char c)
{
    return((float) (a+b+c)/3.0);
}
```

Utilisation du type void

Certaines fonctions n'ont pas de sortie (pas de valeur de retour) : exemple des fonctions d'affichage par exemple. On utilise, dans la définition de la fonction, **printf**, mais la fonction ne fait pas de calcul, ne renvoie rien.

On utilise alors le type void (rien) pour la valeur de retour. Il faudra tout de même utiliser **return;** pour terminer la fonction.

De même, certaines fonctions n'ont pas d'entrées (affichage d'un message qui ne change pas par exemple) : on utilise alors le type void, on peut même omettre de préciser le type des entrées.

Utilisation du type void

Fonction sans valeur de retour : affichage d'un entier avec conditions, cet entier est par exemple le nombre de bonnes réponses à un quizz.

En entrée : un entier; en sortie : rien; nom : affEntier.

Prototype :

```
void affEntier(int);
```

définition :

```
void affEntier(int nbRep)
{
    if (nbRep==0)
        { printf("pas de bonne réponse\n"); }
    else if (nbRep==1)
        { printf("%d bonne réponse\n",nbRep); }
    else
        { printf("%d bonnes réponses\n",nbRep); }

    return; /* le return n'est pas obligatoire */
}
```

Utilisation du type void

Fonction sans entrée : saisie et renvoi d'un valeur à virgule.

En entrée : rien; en sortie : une valeur double; nom : saisiVal.

Prototype :

```
double saisiVal(void);          ou          double saisiVal();
```

définition :

```
double saisiVal(void)
{
    double valeurASaisir;

    printf("entrez la valeur à saisir :");
    scanf("%lf", &valeurASaisir);

    return valeurASaisir;
}
```

Appel de la fonction

Résumé avant la suite :

prototype de fonction : mode d'emploi; comment utiliser une fonction
: ne décrit pas ce qu'elle fait, ne l'utilise (l'appelle) pas

définition de fonction : décrit ce qu'elle fait (comment elle traite les entrées, ce qu'elle fournit en sortie) : ne donne pas son mode d'emploi, ne l'utilise (l'appelle) pas.

Reste à pouvoir l'utiliser : appel depuis un programme principal ou depuis une autre fonction. Le programme ou fonction qui appelle une fonction donnée est appelé **programme appelant**.

Appel de la fonction

Le programme appelant une fonction doit juste savoir comment utiliser la fonction, et non pas comment elle fonctionne : doit connaître le prototype de la fonction à appeler.

Le **prototype** de la fonction à appeler doit être situé **avant** le programme appelant.

La **définition** par contre, peut être située avant ou après, c'est le compilateur qui fera le lien.

L'appel d'une fonction est une demande pour utiliser la fonction: on doit donner des valeurs aux entrées pour que la fonction les traite, et on récupère la valeur de retour de la fonction.

Appel de la fonction

Un appel de fonction est traité comme une instruction dans le programme appelant.

Syntaxe d'un appel de fonction :

soit une fonction nommée dont on connaît le prototype :

`type_retour nom_fonction(type entrée1, type entrée_2, ...);`

`variable = nom_fonction(valeur entrée_1, valeur entrée_2, ...);`



2 effets : récupération de la valeur de retour; appel à la fonction.

Appel de la fonction

Les valeurs des entrées sont les valeurs avec lesquelles va effectivement travailler la fonction. Ce sont les **arguments** (ne pas confondre avec les **paramètres**).

La variable dans laquelle on récupère la valeur de retour doit être du même type que la valeur de retour elle-même.

On peut juste faire un appel sans récupérer la valeur de retour dans deux cas :

- il n'y a pas de valeur de retour (la fonction est de type void);
- on ne veut pas l'utiliser.

il suffit d'écrire la partie **appel** de la fonction.

Positionnement dans un programme

Où positionner les prototypes, appels et définitions dans un programme ?

Voici la structure type d'un programme utilisant des fonctions :

inclusions de fichiers d'entête (comme `stdio.h`)

définition des constantes par `#define`

prototypes de fonctions

programme principal (contient les *appels de fonctions*)

définition des fonctions

Positionnement dans un programme

Exemple avec un programme utilisant un calcul de moyenne de 3 valeurs :

```
#include <stdio.h>
float moyenne(char, char, char); ← prototype

void main()
{
    char x1, x2, x3;
    float moy;

    /* saisie des valeurs de x1, x2, x3 */

    moy = moyenne(x1, x2, x3); ← appel

    printf("la moyenne est : %f\n", moy);
}
```

Positionnement dans un programme

```
float moyenne(char a, char b, char c)
{
    float res = (float) (a+b+c)/3.0;
    return res;
}
```

définition

Le programme principal peut appeler la fonction moyenne, car le prototype (mode d'emploi) est situé avant.

Remarque : les noms des paramètres sont indépendants des noms des arguments.

Autres exemples de fonctions

En fait, on utilise (sans le savoir), des fonctions depuis le début !
Lesquelles ?

- *printf*, *scanf*, *sqrt* sont des fonctions !
- *main* est une fonction : rôle particulier : c'est là que le programme va commencer à s'exécuter, sinon aucune différence !

En fait, le prototype de *sqrt* est, par exemple :

```
double sqrt(double);
```

il se trouve dans le fichier **math.h**, c'est pour cela qu'il est inclus dans les programmes utilisant *sqrt* ou les autres fonctions mathématiques !

main n'a pas de prototype défini, on peut en fait lui donner le type de retour que l'on veut et les paramètres que l'on veut !

Paramètres et arguments

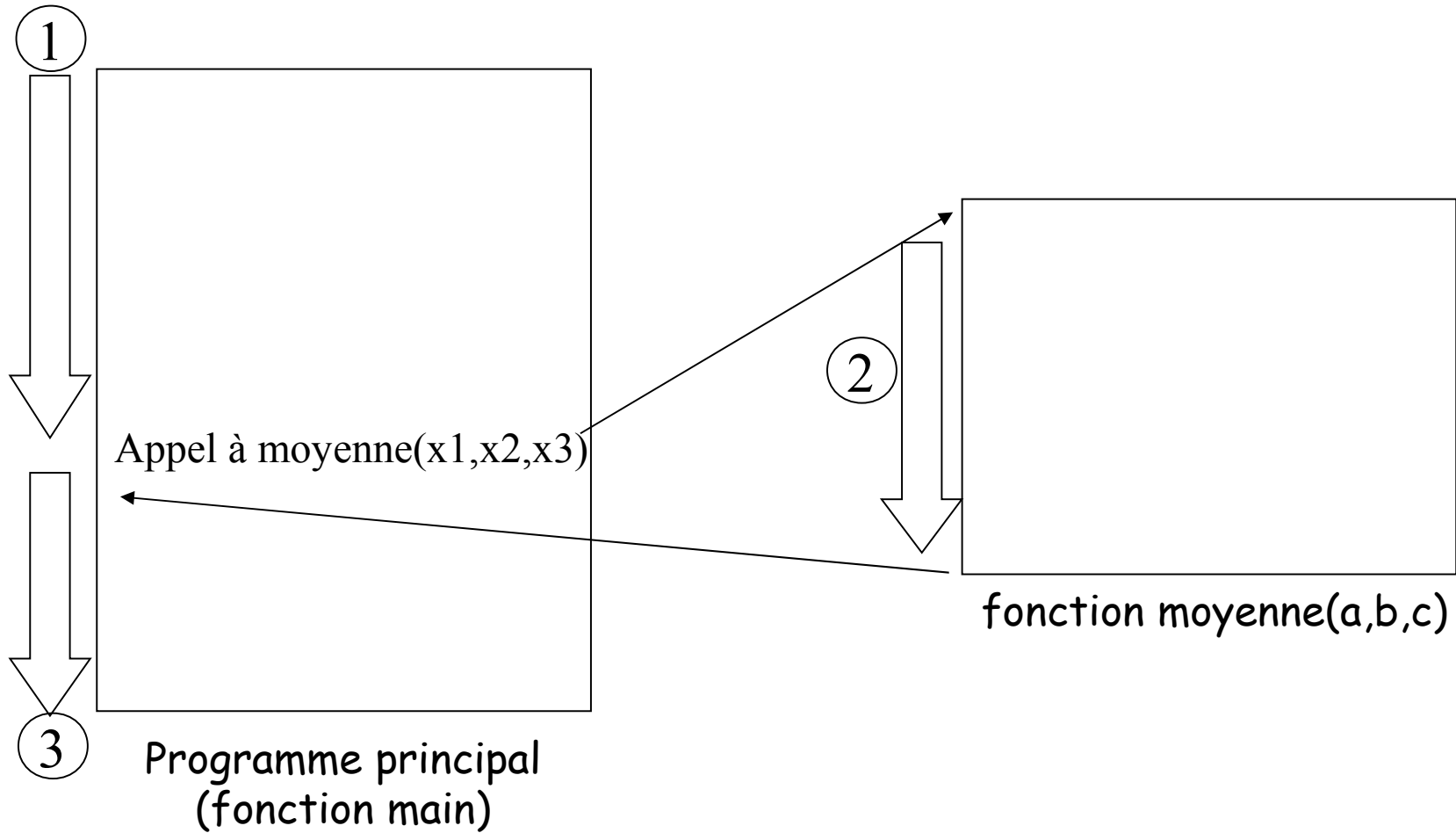
Attention à la différence : ordre d'écriture / ordre d'exécution : dans l'exemple : la définition de la fonction moyenne se trouve après le programme principal.

Ne veut pas dire que cette fonction va être effectuée après le programme principal (n'a pas de sens, à la fin du programme principal, il n'y a plus rien à faire !).

La fonction est exécutée quand elle est appelée (dans le programme principal), schéma déjà vu mais repris ici, avec l'exemple précédent.

On utilise le schéma bloc.

Paramètres et arguments



Paramètres et arguments

Que se passe-t-il exactement lors de l'appel à la fonction pour les arguments ?

Ils se trouvent dans le programme principal, inconnus de la fonction. Essai d'utilisation de x_1 , x_2 , x_3 dans la fonction : erreur.

Et pour les paramètres ? Connus dans la fonction, inconnus du programme principal. Essai d'utilisation de a , b , c dans le programme principal : erreur.

Rappel pour clarifier :

la fonction décrit comment à partir de 3 paramètres a , b et c , on fait pour calculer leurs moyennes.

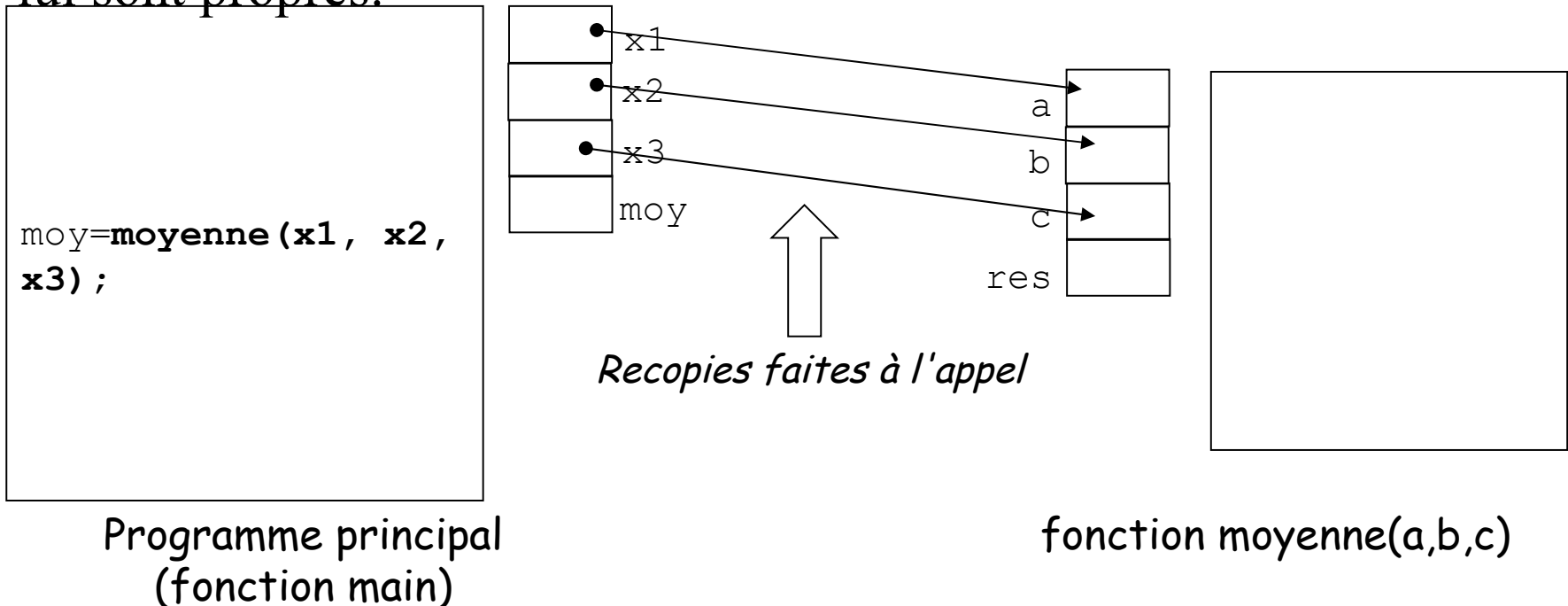
Le programme principal veut récupérer la moyenne de x_1 , x_2 , x_3 .

Paramètres et arguments

Il y a une recopie des **arguments** vers les **paramètres** :

dans le programme, l'appel `moyenne(x1, x2, x3)` signifie : exécuter la fonction `moyenne` avec $a = x1$, $b = x2$ et $c = x3$.

Symbolisation : on fait figurer à côté de chaque bloc les variables qui lui sont propres.



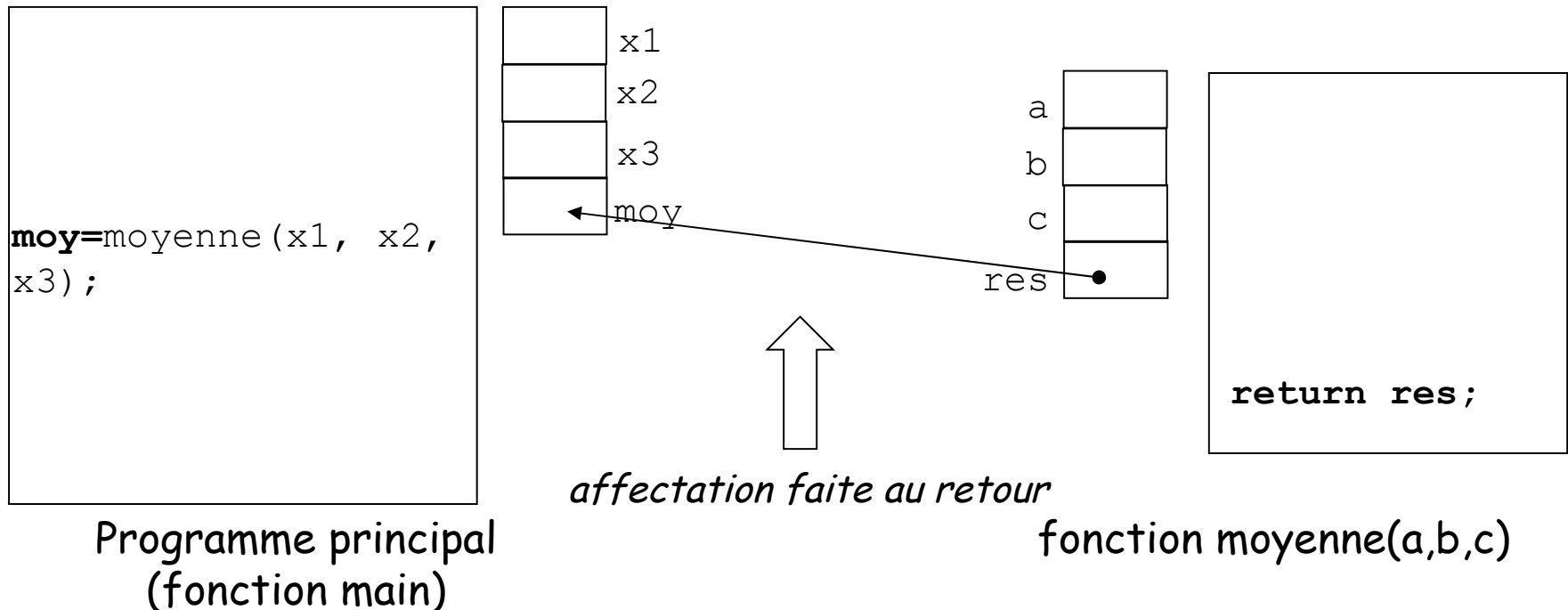
Paramètres et arguments

Paramètres : on peut aussi utiliser des constantes numériques comme arguments (pas comme paramètres). Exemple :

`moy=moyenne(2,5,12)`; dans ce cas ce sont les constantes qui sont copiées dans les paramètres pour que la fonction puisse se dérouler

Paramètres et arguments

Retour : si l'on veut récupérer la valeur de retour, il y a une affectation de la valeur de retour de la fonction (donnée par l'instruction `return` à l'intérieur de celle-ci).



Implications

Ainsi, un appel à une fonction ne modifie pas la valeur de ses arguments, puisque la fonction travaille avec ses paramètres qui sont indépendants des arguments.

Il y a un **passage de paramètres par valeurs** : on recopie la valeur de l'argument dans le paramètre de la fonction.

Illustration avec la représentation par blocs, soit le programme et la fonction suivante :

```
#include <stdio.h>
int modifi(int);

void main()
{
    int a, b;
    a=7;
    b=6;
    printf("a=%d, b=%d\n", a, b);
}
```

Implications

```
/* suite du programme */  
  
    b=modifi(a);  
    printf("a=%d, b=%d\n",a,b);  
}  
  
int modifi(int param)  
{  
    param = param+1;  
    return param;  
}
```

que va afficher ce programme ? Faire l'illustration avec les schémas-blocs.

Où sont les prototypes ? Les définitions de fonctions ? Les appels de fonction ?

Implications

Tendance à nommer les paramètres et les arguments de la même manière (on croit que le lien se fait avec le nom), mais le lien est fait par la position de l'entrée dans l'écriture de l'appel.

Au contraire, utilisez des noms différents pour paramètres et arguments : renforce le découpage, bonne habitude et bonne approche !

N'oubliez pas : une fonction peut appeler une autre fonction (appel de *printf* dans la fonction *moyenne* par exemple). Important plus tard pour la récursivité.

Travail avec les tableaux

Cas des variables de type simple traité; fonctions travaillent aussi avec des tableaux : différence au niveau des paramètres et arguments.

Illustration : fonction réalisant l'affichage d'un tableau contenant des valeurs de type double.

Toujours procéder dans l'ordre : prototype - définition - appels.

Nom : affTab;

entrées : le nombre d'éléments à afficher (la taille utile du tableau), l'ensemble des valeurs contenues dans le tableau : on ne donne pas les éléments individuellement, mais le tableau lui-même.

Sortie : aucune, c'est un affichage.

Travail avec les tableaux

D'où le prototype :

```
void affTab(float [], int);
```

attention à la syntaxe du type pour le prototype !
'tableau de *type*' s'écrit *type* [].

Définition :

```
void affTab(float tablo_f[], int nbElem)
{
    int compt;
    for (compt=0; compt < nbElem; compt++)
    {
        printf("%f ", tablo_f[compt]);
    }
    return; /* pas obligatoire : affTab est de type void */
}
```

Travail avec les tableaux

Dernière étape : appel d'une fonction avec un tableau en argument.
Attention, lors de l'appel, l'argument n'est pas le **type**, mais la **valeur** avec laquelle la fonction doit opérer.

Exemple de programme utilisant la fonction `affTab` avec une erreur classique (évitable quand on sait exactement ce que l'on fait).

```
#include <stdio.h>

void affTab(float [], int);

void main()
{
    float tablo[5]={0.0,2.5E+2,16000.2};
    int    taille = 3;

    affTab(tablo[], taille);
}

/* définition de la fonction Afftab ici */
```

tablo[] n'a aucun sens : comment l'interpréter ?

Penser à la compatibilité des types : le paramètre est de type : tableau de float.

Or tablo est un tableau de float, tablo[] ne signifie rien !

```
#include <stdio.h>

void affTab(float [], int);

void main()
{
    float tablo[5]={0.0,2.5E+2,16000.2};
    /* cette ligne se lit : tablo est un tableau */
    /* avec au maximum 5 valeurs de type float ! */

    int    taille = 3;

    affTab(tablo, taille); /* appel correct */
}

/* définition de la fonction Afftab ici */
```

Le compilateur ne comprend pas l'écriture tablo[] par ailleurs !

Tableau en sortie

Traitements sur les tableaux dans des fonctions : possible (souhaitable même !).

Subtilité : faire attention !

But : modifier **les éléments** du tableau.

Dans une fonction, lorsque l'on modifie les éléments d'un tableau passé en paramètre, les modifications sont répercutées sur l'argument.

Cela semble contraire à ce que l'on a déjà vu : un argument dans le programme appelant n'est pas modifié par une fonction appelée...

Tableau en sortie

Différence existante entre :

- le tableau lui-même : regroupement de valeurs de même type,
- les éléments individuels du tableau.

Lorsque le tableau est passé en argument, il n'est pas modifié. Mais les valeurs individuelles des éléments peuvent être modifiées !

Pas besoin d'avoir une sortie lorsqu'on traite les éléments d'un tableau dans une fonction !

Tableau en sortie

Exemple : fonction `inversTab` qui échange les deux premiers éléments d'un tableau (exemple : tableau de float pour continuer le programme précédent).

Prototype et définition :

```
void inversTab(float [], int);
```

```
void inversTab(float paramTab[], int nbElem)
{
    float valTemp;

    if (nbElem >1)
    {
        valTemp=paramTab[0];
        paramTab[0] = paramTab[1];
        paramTab[1] = valTemp;
    }
    return;
}
```


Tableau en sortie

Essai de cette fonction dans le programme précédent :

```
void main()
{
    float tabEx[4]={1.,2.,65.,1.E+12};
    int taille = 4;

    affTab(tabEx,taille);
    inverstTab(tabEx,taille);
    affTab(tabEx,taille);
}
```

résultat :

```
1.00 2.00 65.00 1.0E+12
2.00 1.00 65.00 1.0E+12
```

Pointeurs et sortie de fonction

Jusqu'ici : le type de retour des fonctions : type simple.

On peut utiliser les pointeurs sans problème (c'est un type comme un autre). Une fonction peut renvoyer un pointeur sans aucun problème !

Exemple illustratif, avec emploi de l'allocation dynamique.

On veut écrire une fonction à qui l'on transmet un nombre entier p , et qui retourne un pointeur sur une zone mémoire de p caractères. Cette fonction vérifie que p est >0 .

(c'est le rôle de **malloc()** en fait, avec une amélioration).

Pointeurs et sortie de fonction

En entrée de la fonction : un entier : int

en sortie de la fonction : un pointeur sur entier : int *

nom : alloueEntiers

d'où le prototype : int *alloueEntiers(int);

et la définition de la fonction :

```
int *alloueEntiers(int nbCases)
{
    int *ptr_retour;

    ptr_retour=NULL;
    if (nbCases >0)
    {
        ptr_retour = (int *)malloc(nbCases*sizeof(int));
    }
    return ptr_retour;
}
```

Pointeurs et sortie de fonction

Un exemple d'appel de cette fonction : on peut l'affecter directement à un pointeur sur entier.

```
void main()
{
    int *unPointeur;
    int nbElements;

    printf("combien de cases à allouer ?:");
    fflush(stdin);
    scanf("%d", &nbElements);

    unPointeur = alloueEntiers(nbElements);

    if (unPointeur==NULL)
    {
        printf("L'allocation a echoue\n");
    }
}
```

Pointeurs et sortie de fonction

Pas de limitations, on peut très bien avoir un prototype de fonction ressemblant à ceci :

```
float ****maFonction(paramètres);
```