Sous-algorithmes

1. **Motivation, définitions, et exemples**

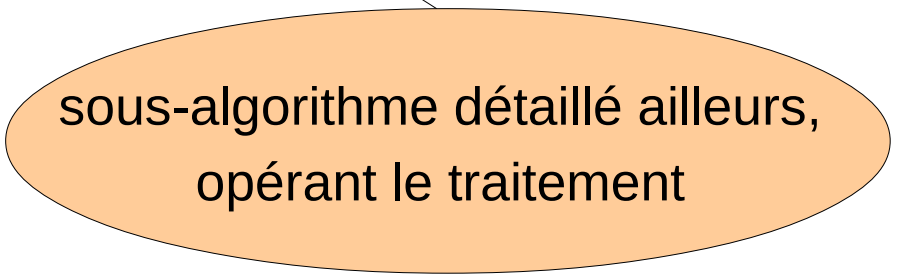
2. **Retour aux tableaux :procédures et fonctions de manipulation de tableaux**

Motivation

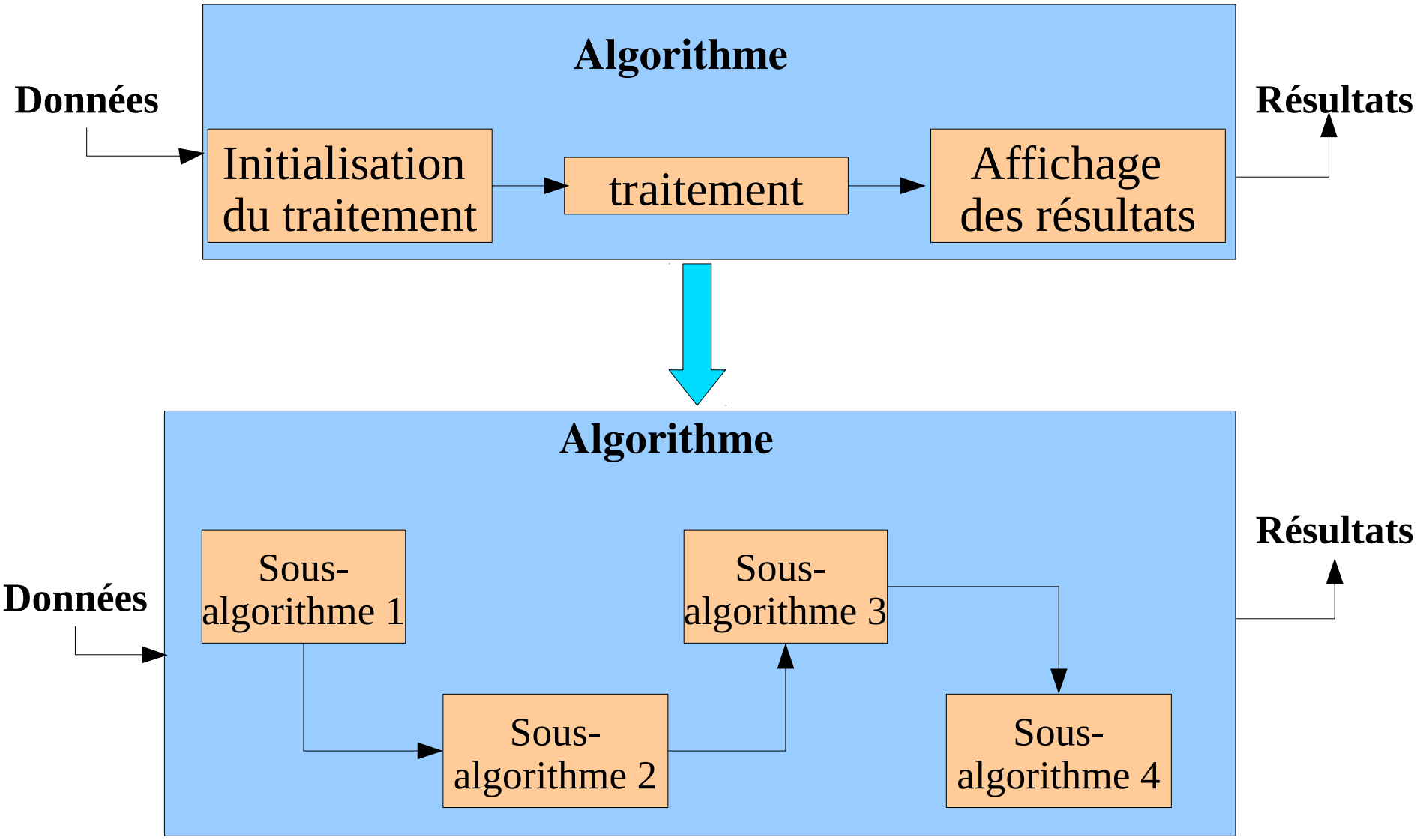
```
Algorithme Puissance
variables valeur, puissance, cpt, nbPuiss : entier
Début
    Écrire("Donnez une valeur positive non nulle : ")
    Lire(valeur)
    Écrire("Le nombre de puissances successives :")
    Lire(nbPuiss)
    //{calcul des nbPuissances puissances successives de valeur }
    puissance ← 1
    pour cpt ← 1 à nbPuiss faire
        Puissance ← puissance * valeur
          Écrire("La", cpt, "ième puissance de", valeur, "est", puissance)
    finpour
fin
```

En simplifiant l'écriture

```
Algorithme Puissance
variables valeur, nbPuiss : entier
Début
    Écrire("Donnez une valeur positive non nulle : ")
    Lire(valeur)
    Écrire("Le nombre de puissances successives :")
    Lire(nbPuiss)
    calculPuissance(valeur,nbPuiss)
fin
```



Nouveau schéma d'un algorithme



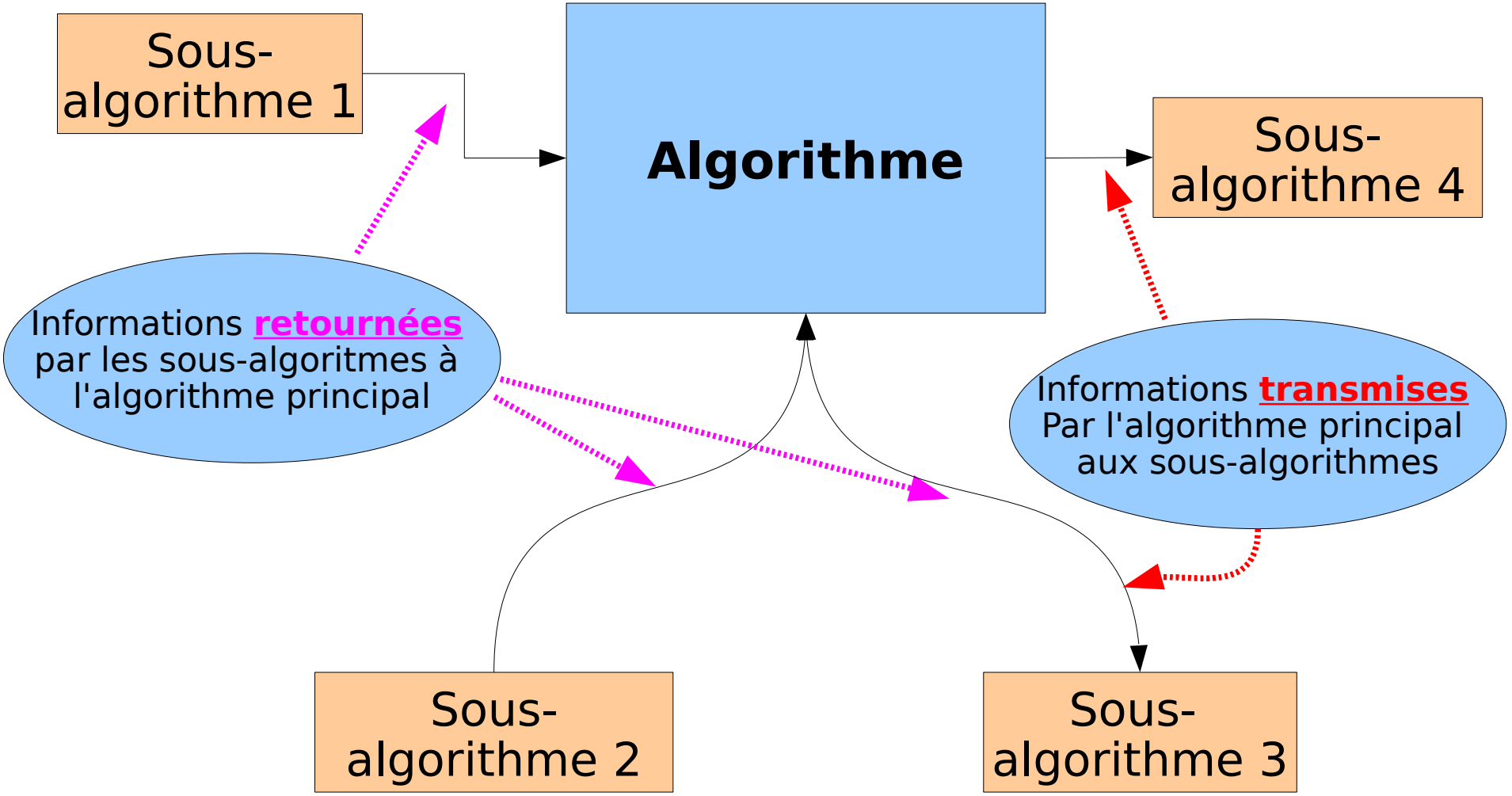
Notion de sous-algorithmes

- Un algorithme **appelle** un sous-algorithme :
  - L'algorithme appelant *passe momentanément le contrôle* de l'exécution du traitement au sous-algorithme.

- Un sous-algorithme est conçu pour faire un traitement bien **défini**, bien **délimité**, si possible *indépendamment* du contexte particulier de l'algorithme appelant.

- Remarque : un sous-algorithme peut en appeler un autre.

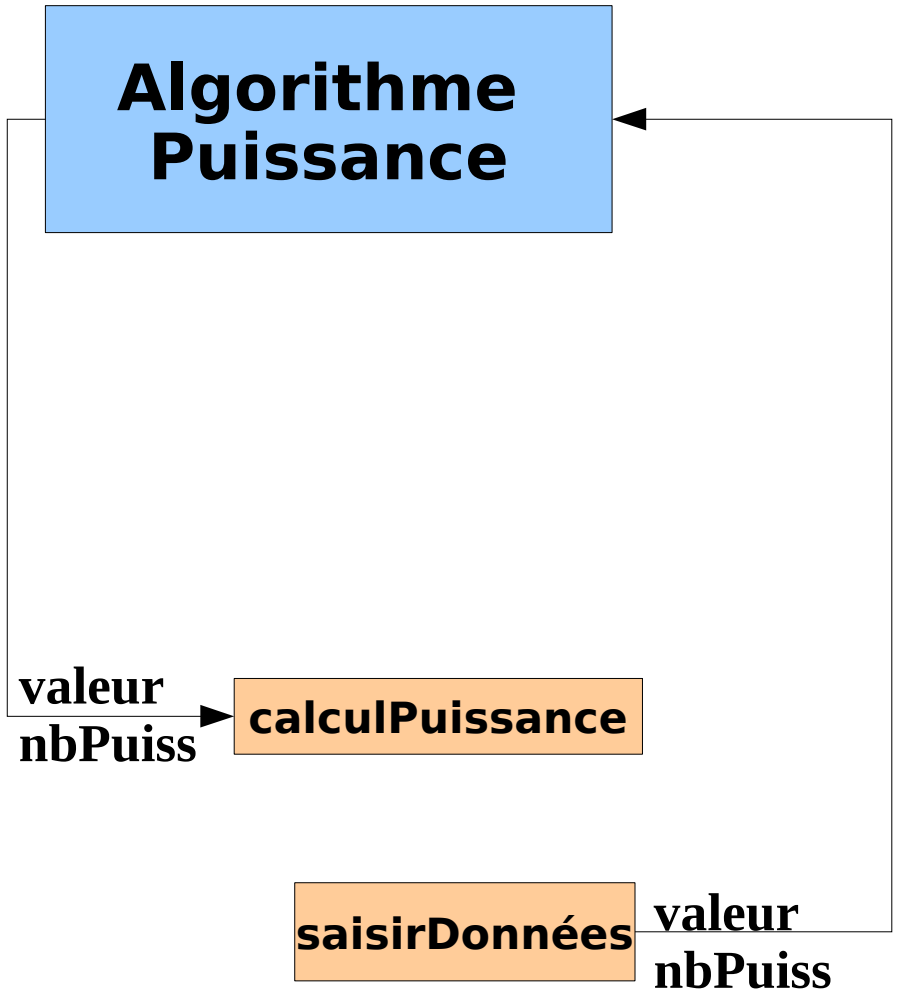
Comment les informations circulent ?



Exemple

```
Algorithme Puissance
variables valeur,
          nbPuiss : entier
Début
     Écrire(" ... ")
     Lire(valeur)
     Écrire(" ... ")
     Lire(nbPuiss)
     calculPuissance(valeur,nbPuiss)
fin
```

```
Algorithme Puissance
variables valeur,
          nbPuiss : entier
Début
     SaisirDonnees(valeur,nbPuiss)
     calculPuissance(valeur,nbPuiss)
fin
```



Communications d'informations

algorithme

sous-algorithme

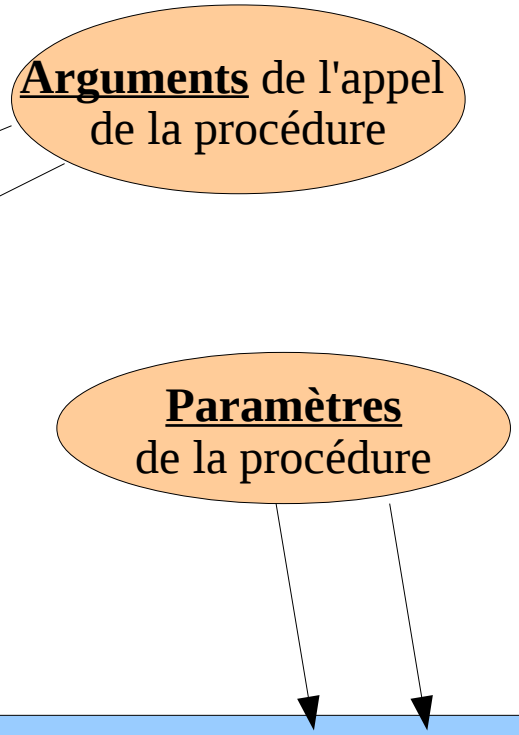
Paramètres en « donnée »

Paramètres en « résultat »

Paramètres en « donnée » et « résultat »

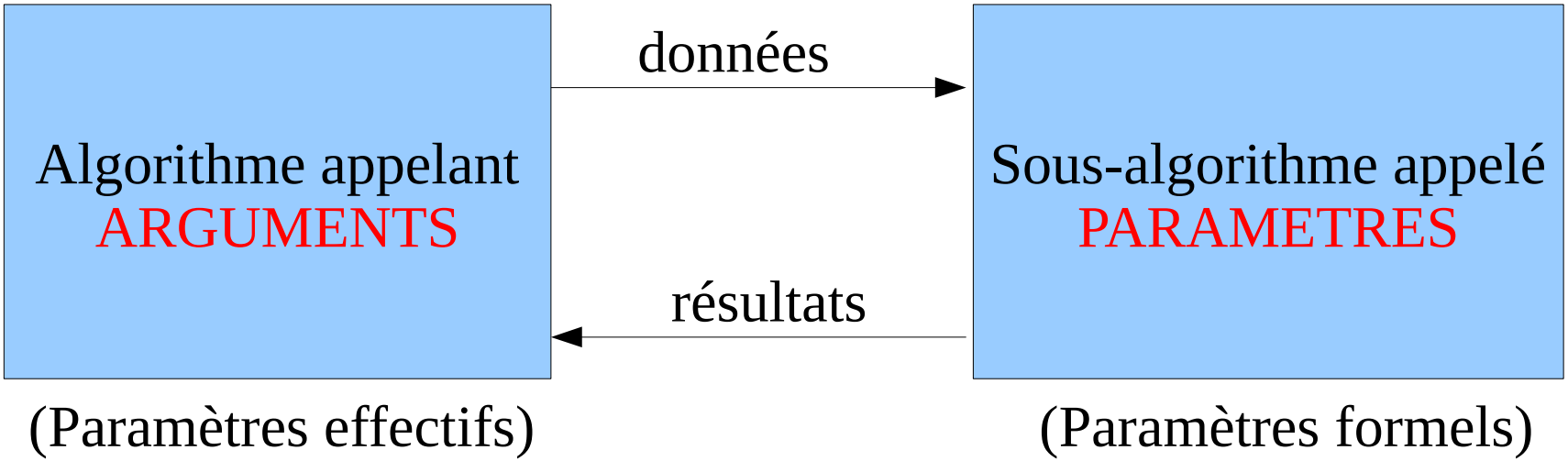
Paramètres et Arguments

```
Algorithme Puissance
variables valeur,nbPuiss : entier
Début
   Écrire(" … ")
   Lire(valeur)
   Écrire(" … ")
   Lire(nbPuiss)
   calculPuissance(valeur,nbPuiss)
fin
```




```
Procédure calculPuissance(val,nb)
variables val, nb : entier
Début
   ...
fin
```

Paramètres et arguments (suite)

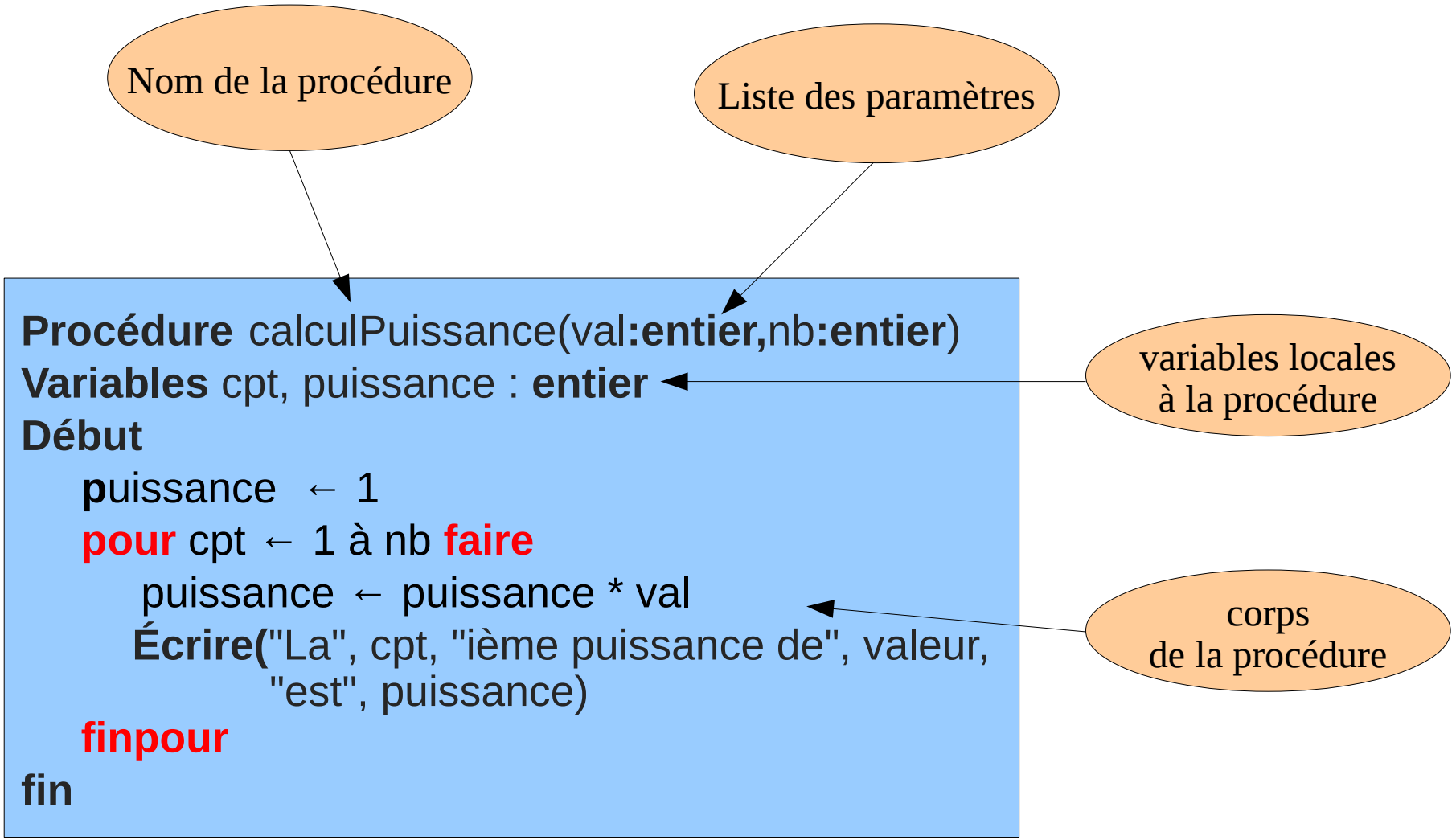


Paramètres et arguments

- Les paramètres (formels) d'un sous-algorithme sont simplement des **variables locales** qui sont initialisées par les valeurs obtenues lors de l'appel (paramètres effectifs ou **arguments**) :
  - Au moment de l'appel du sous-algorithme , les valeurs des arguments sont affectés aux paramètres formels de statut (D) ou (D/R) ;

- Le nombre de **paramètres** doit correspondre au nombre **d'arguments**.
- Le **type** du **k-ième** argument doit correspondre au type du **k-ième** paramètre.
- Un paramètre défini en "**Donnée**" doit correspondre à un argument qui possède une valeur dans l'algorithme appelant au moment de l'appel.
- Un paramètre défini en "**Résultat**" doit recevoir une valeur dans le sous-algorithme.

Procédure : syntaxe



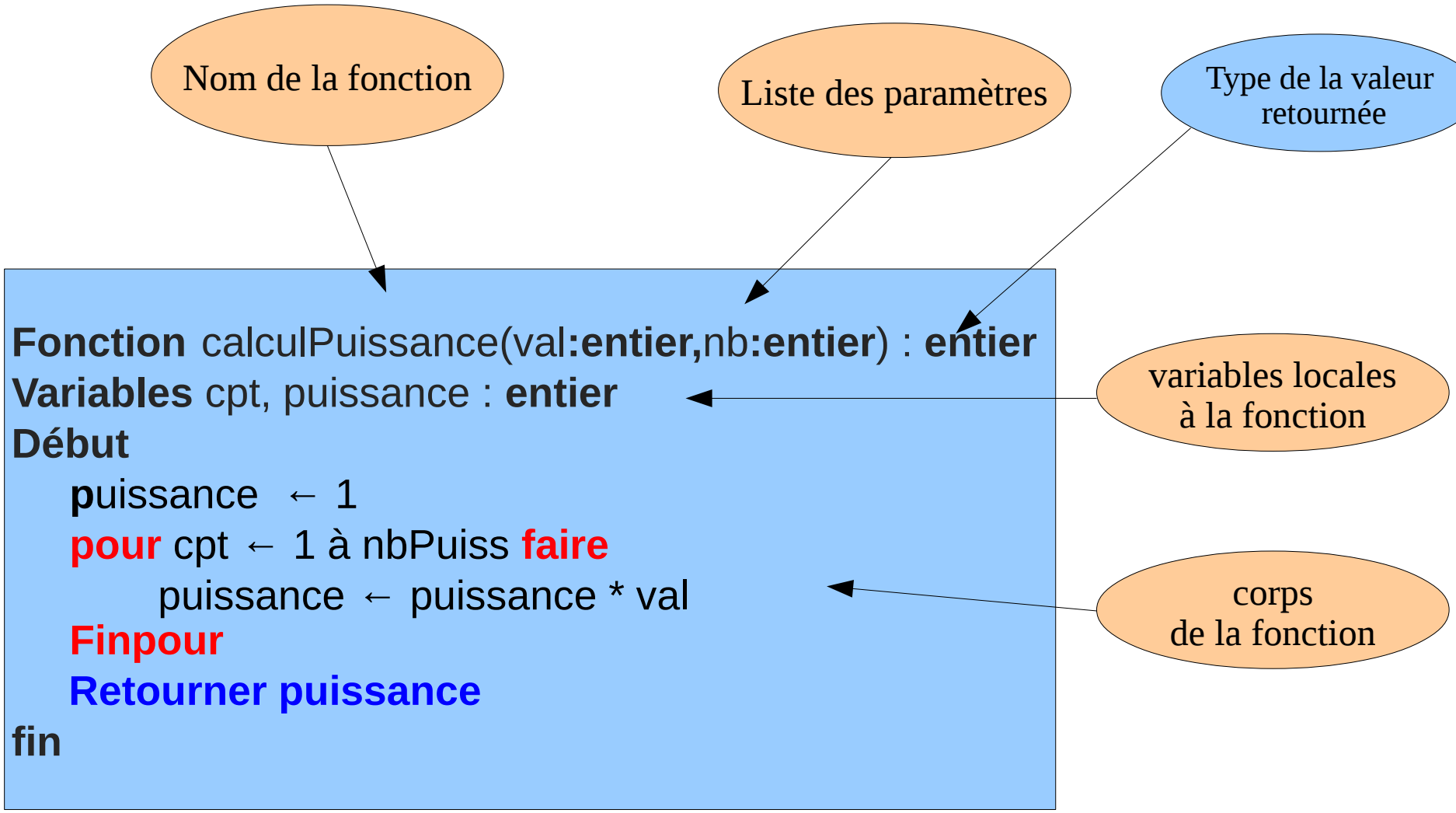
Fonctions

- Une fonction est un sous-algorithme qui retourne une valeur.

```
Algorithme Puissance
variables valeur, nbPuiss, puissance : entier
Début
    Écrire("Donnez une valeur positive non nulle : ")
    Lire(valeur)
    Écrire("Le nombre de puissances successives :")
    Lire(nbPuiss)
    puissance ← calculPuissance(valeur,nbPuiss)
    Écrire("La puissance de", valeur, "est", puissance)
fin
```

sous-algorithme détaillé ailleurs, opérant le traitement, **et retournant une valeur**

Fonction : syntaxe



Mise en ouvre d'une fonction

- L’utilisation d’une fonction nécessite trois étapes :
  - la déclaration de la fonction
    (interface, en-tête ou encore **prototype**)
  - la définition du corps de la fonction
  - l’appel de la fonction

PROTOTYPE

- SYNTAXE:

  fonction identificateur (paramètres_formels) : type

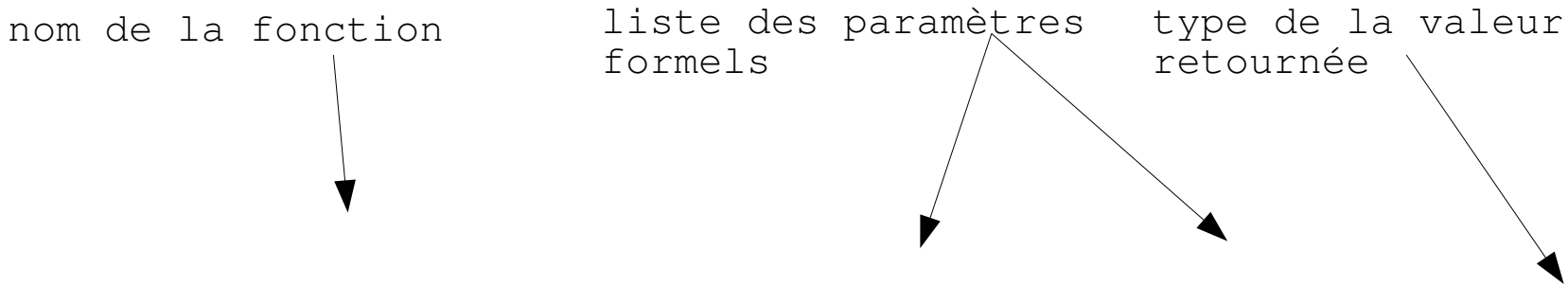
- Exemple :

  fonction puissance(p : entier, n : entier) : réel ;

  ou encore

  fonction puissance(entier, entier) : réel ;

Exemple



- Fonction calculPuissance (x:entier, n:entier) : entier

```
    var i, puis : entier ;
    Début
    i = 1 , puis = 1
    pour i = 1 à n faire
         puis  =  puis * x;
    retourner puis ;
    Fin
```

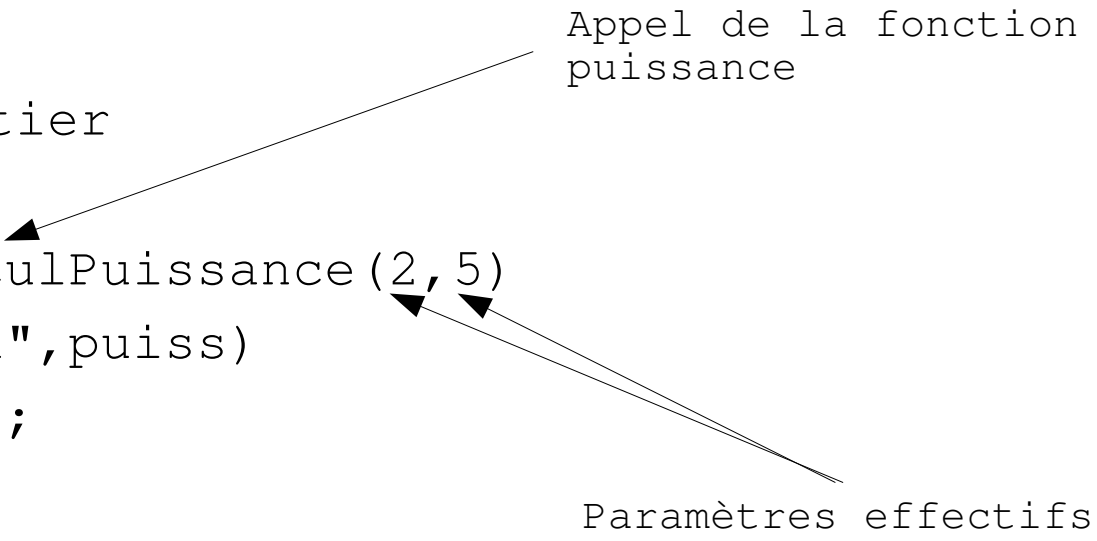
déclarations des variables locales

Appel d'une fonction

- L'appel d'une fonction est réalisé en invoquant le nom de la fonction suivi de la liste des paramètres effectifs.

- Exemple :

```
algo calPuiss
   var puiss : entier
   Début
      puiss = calculPuissance(2,5)
      écrire("%d\n",puiss)
      retourner 0 ;
   Fin
```



Procédure ou fonction ?

```
Fonction calculPuissance (x:entier, n:entier) : entier
variables i, puis : entier ;
Début
        i ←1 , puis ← 1
        pour i ← 1 à n faire
                puis  ← puis * x
                retourner puis
        finpour

Fin
```

```
Procédure calculPuissance (x:entier, n:entier,
puis:entier)
variables i, puis : entier
Début
        i ←1 , puis ← 1
        pour i ← 1 à n faire
                puis  ← puis * x
        finpour

Fin
```

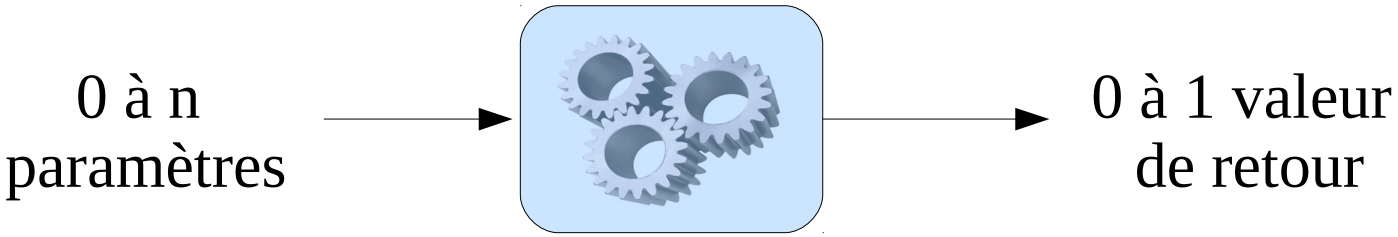
A l'appel (dans l'algorithme principal) :

- **version fonction** : puissance ← **calculPuissance**(valeur,nbPuiss)
- **version procédure** : **calculPuissance**(valeur,nbPuiss,puissance)

Les fonctions en C

Définition de fonctions (1)

- Principe d'une fonction:



- Syntaxe:

```
type_retour  identifiant (parametres_formels)
    {
       déclarations ;
       instructions ;
       return expression;
    }
```

} En-tête de la fonction

} Corps de la fonction

Définition de fonctions (2)

- Exemples:

```
int somme (int a , int b){
    int res = a + b;
    return res;
}

int sommeBis(int a, int b){
    return (a + b);
}

float multiplie (float f, int a){
    return (f * a) ;
}

void afficheCoucou(){
printf("Coucou \n");
}
```

Définition de fonctions (3)

- Type de retour:

  type de la valeur retournée par la fonction

  - `int, float, char, void`

  Une fonction dont le type de retour est "void" est une fonction qui ne retourne rien. Ces fonctions sont également appelées "procédures" dans d'autres langages

- Identifiant : nom de la fonction (même règles pour les noms de variables)

- Paramètres formels :
  - Liste de déclarations des paramètres associés aux arguments transmis à la fonction lors de son appel.
  - Leur durée de vie est égale à la durée d’exécution de la fonction.

Définition de fonctions (4)

- Corps de la fonction: suite d'instructions
  - Le corps de la fonction définit la suite des instructions nécessaires à la réalisation du sous-programme.
  - Le corps de la fonction est un bloc d'instructions.
  - Les déclarations de variables sont réalisées en tête du bloc de la fonction.
  - Pour toute exécution de fonction dont le type de retour n'est pas "void", chaque branche d'exécution doit se terminer par une instruction de retour de valeur :

```
return expression ;
```

Définition de fonctions (5)

- L'instruction retourner : **`return expression ;`**

  Où expression correspond au résultat retourné par la fonction.

- Permet à la fonction de retourner la valeur expression à l'instruction appelante ;

- Provoque une sortie immédiate de la fonction

- Remarques :
  - une fonction retourne au plus une valeur
  - Le type de l'expression est le même que le type de retour de la fonction.

L'instruction retourner (suite)

- Fonction sans valeur de retour :

```
void affiche_flot(float x){
    printf ("%f \n", x);
    return; }
```

- Fonction sans arguments :

```
float pi(void){
    return 3.14159;
}
```

- Il peut y avoir plusieurs instructions retourner, mais c'est la première instruction rencontrée qui provoque l'arrêt de la fonction.

```
double abs_sum(double x, double y){
    double s = x + y ;
    if (s > 0) return s ;
    else return (-s) ;
}
```

Définition de fonctions (6)

- Que se passe-t-il si l'instruction « return » d'une fonction de type de retour non void est oubliée ?

```
int somme (int a , int b){
    int res = a + b;
}
```

Attention à la compilation

attention : « return » manquant dans une fonction devant retourner une valeur

Prototype d'une fonction

Syntaxe :

```
type_retour identifiant (paramètres_formels);
```

- Le prototype d'une fonction décrit comment utiliser cette fonction.
- Le prototype est une déclaration. Il déclare la fonction.
- Le prototype n'est pas suivi du corps de la fonction.

Intérêt :

- Un prototype indique comment appeler une fonction
- Placé avant le point d’appel, il permet au compilateur de vérifier la cohérence de l’appel.

Exemples de prototypes :

```
int factorielle(int n);
int somme(int a, int b) ;
```

Exemple

```
#include<stdio.h>
int puissance(int x, int n);


int puissance(int x, int n)
{
int i, p=1;
for (i = 1; i <= n; i++)
p = p * x;
return p;
}
```

prototype de la fonction puissance

corps de la fonction puissance avec deux paramètres formels : x et n

Déclaration des variables locales

Exemple complet (en C)

```
#include<stdio.h>
int calculPuissance(int x, int n);
int main(void)
{
int puiss;
puiss = calculPuissance(2,5);
printf("%d\n", puiss);
return 0;
}
int calculPuissance(int x, int n)
{
int i, p=1;
for (i = 1; i <= n; i++)
p = p * x;
return p;
}
```

prototype de la fonction puissance

Appel de la fonction puissance

Définition de la fonction puissance avec deux paramètres formels : x et n

Appels de fonctions (1)

- La définition d'une fonction n'exécute pas le programme qui la compose.

- Pour exécuter les instructions, il faut appeler la fonction :

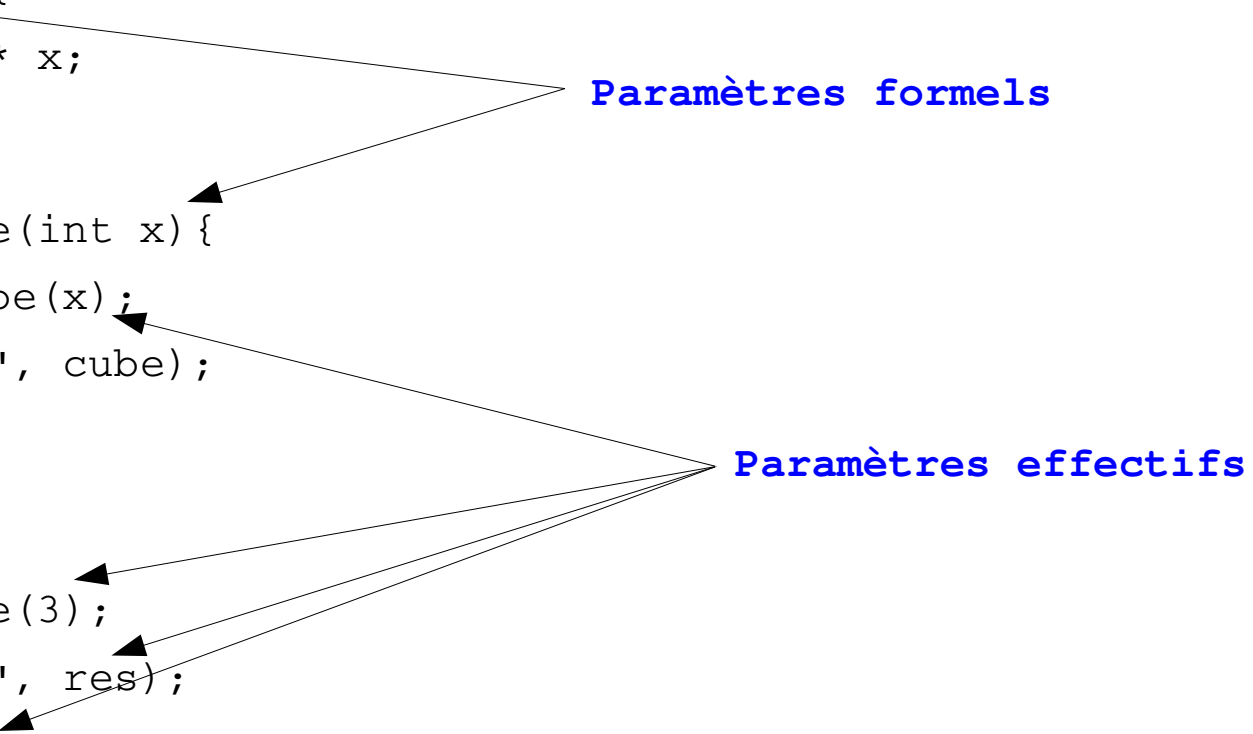
  identifiant (paramètres_effectifs);

- Paramètres formels, paramètres effectifs :
  - Les paramètres formels sont utilisés lors de la définition de la fonction.
  - Lors de l'appel, des valeurs doivent être données à ces paramètres → paramètres effectifs.
  - Un paramètre effectif est soit une variable (sa valeur), soit une valeur d'expression ou de retour d'un appel de fonction.

Appels de fonctions (2)

Exemple :

```
#include <stdio.h>
int cube(int x){
  return x * x * x;
  }

void afficheCube(int x){
  int cube = cube(x);
  printf("%d \n", cube);
  }

int main(){
  int res = cube(3);
  printf("%d \n", res);
  afficheCube(4);
  return 0;
}
```



Appels de fonctions (3)

Exemple :

```
#include <stdio.h>
int cube(int x){
  return x * x * x;
  }

int main(){
  int res = cube(3);
  printf("%d \n", res);
  afficheCube(4);
  return 0;
}

void afficheCube(int x){
     int cube = cube(x);
     printf("%d \n", cube);
}
```

Attention à la compilation :

```
In function ‘main’:
test.c:11: attention :
implicit declaration
of function ‘afficheCube’
```

Appels de fonctions (4)

Exemple :

```
#include <stdio.h>
int cube(int x){
  return x * x * x;
  }

void afficheCube(int x){
  int cube = cube(x);
  printf("%d \n", cube);
  }

AffichеCube(4);

int main(){
  int res = cube(3);
  printf("%d \n", res);
  return 0;
}
```

Erreur de compilation :

```
test.c:11: erreur: expected declaration specifiers
or `...' before numeric constant
test.c:11: attention : data definition has no type or
storage class
test.c:11: attention : type defaults to `int' in
declaration of `afficheCube'
test.c:11: erreur: conflicting types for `afficheCube'
test11.c:6: erreur: previous definition of `afficheCube'
was he
```

Appels de fonctions (5)

Exemple :

```
#include <stdio.h>
int cube(int x){
  return x * x * x;
  }

void afficheCube(int x){
  int cube = cube(x);
  printf("%d \n", cube);
  }


int main(){
  int res = cube(3,5);
  printf("%d \n", res);
   AfficheCube(4);
  return 0;
}
```

Erreur de compilation :

```
test.c: In function 'main':
test.c:12: erreur: too many
arguments to function 'cube'
```

Appels de fonctions (6)

Exemple :

```
#include <stdio.h>
int cube(int x){
  return a * a * a;
  }

void afficheCube(int x){
  int cube = cube(x);
  printf("%d \n", cube);
  }


int main(){
  int res = cube(3);
  printf("%d \n", res);
   AfficheCube(4);
  return 0;
}
```

Erreur de compilation :

```
test.c: In function 'cube':
test.c:3: erreur: 'a' undeclared
(first use in this function)
```

Portée des variables (1)

- Portée des variables :
  - La portée d'une variable désigne la partie du programme dans laquelle on peut l'utiliser.
- On peut se demander quelle est **la portée** d'une variable ?
  - une variable déclarée dans une fonction peut-elle être utilisée dans une autre partie du programme ?
  - quelle est sa "durée de vie" ?
- On distingue deux types de variables :
  - Les variables **globales**
  - Les variables **locales**

Les variables globales

- **Définition** : Une variable globale est une variable déclarée en dehors de toute fonction (même le main) :
  - Elles sont disponibles à toutes les fonctions du programme
  - En général, elles sont déclarées immédiatement derrière les instructions **`#include`** au début du programme.

- **Attention** :
  - Les variables déclarées au début de la fonction principale main() ne sont pas des variables globales, mais **elles sont locales à main()** !
  - Il faut faire attention à ne pas cacher involontairement des variables globales par des variables locales du même nom.

- L'utilisation de variables globales devient inévitable, si
  - plusieurs fonctions d'un programme ont besoin du même ensemble de variables. Ce serait alors trop encombrant de passer toutes les variables comme paramètres d'une fonction à l'autre.

Les variables locales

- **Définition** : Une variable locale est une variable déclarée à l'intérieur d'une fonction :
  - Par défaut, elles sont visibles uniquement à l'intérieur de la fonction dans laquelle elles sont déclarées ;
  - A la sortie de la fonction, les variables locales sont détruites et leur valeur perdues.

```
#include <stdio.h>
void fonction(){
char note = 'l';
printf("La note est %c\n",note);
note=’s’;
return;
}

main(){
fonction();
fonction();
}
```

Portées des variables (2)

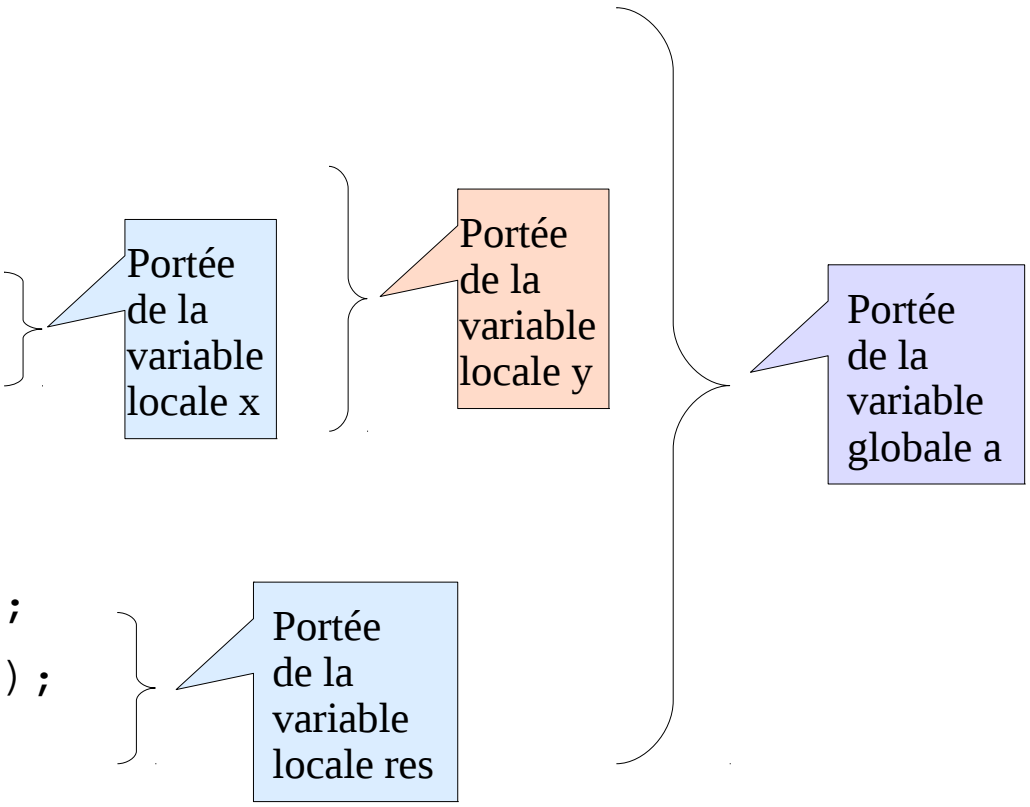
- Structure d'un programme :

```
#include <stdio.h>
#define ...
/* Déclaration des variables globales */
/* Définition des fonctions */
int main(){
/* Définition des éventuelles variables locales de main */
...
...
return 0;
}
```

Portées des variables (3)

- Exemple :

```
#include <stdio.h>
int a = 2;

int somme(int y){
    int x = 3;
    return a + x + y;
}

int main(){
    int res = somme(7);
    printf("%d \n",res);
    return 0;
}
```

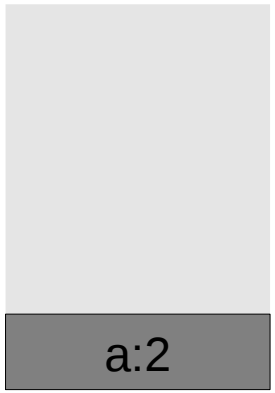


Portées des variables (4)

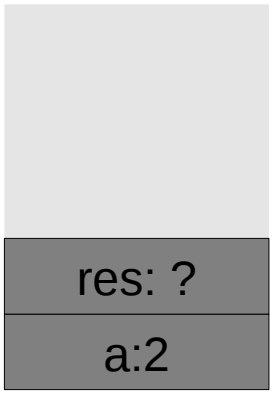
Que se passe-t-il en mémoire ?

```
#include <stdio.h>
int a = 2;

int somme(int y){
    int x = 3;
    return a + x + y;
}

int main(){
    int res = somme(7);
    printf("%d \n",res);
    return 0;
}
```

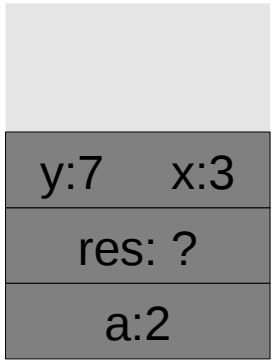
1 - Avant l'exécution de main :



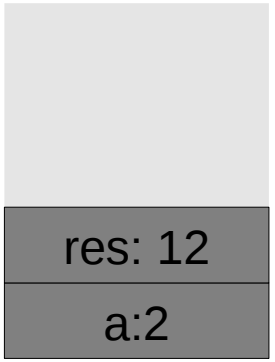
2 – Empilement de la variable locale de main :



3 – Empilement du Paramètre d'appel et de la variable locale de somme:



4 – Après appel de somme:



Portées des variables (5)

- Variables locales et globales de même nom :

  Lorsqu'une variable locale et une variable globale portent le même nom, la portée de la variable locale masque la portée de la variable globale.

```
#include <stdio.h>
int a = 2;

int somme(int y){
    int x = 3;
    int a = 5;
    return a + x + y;
}

int main(){
    int res = somme(7);
    printf("%d \n",res);
    return 0;
}
```

Portées des variables (6)

```
#include <stdio.h>
int note1 = 1;
int note2 = 1;
void f1();

main() {
    int note3 = 3;
    printf("note1 est %d\n",note1);
    printf("note2 est %d\n",note2);
    printf("note3 est %d\n",note3);
    f1();
    printf("note1 est %d\n",note1);
    printf("note2 est %d\n",note2);
    printf("note3 est %d\n",note3);
}
```

```
void f1(){
    int note3, note2;
    note1 = 2;
    note2 = 2;
    note3 = 2;
    printf("Dans f note1 est
            %d\n",note1);
    printf("Dans f note2 est
            %d\n",note2);
    printf("Dans f note3 est
                %d\n",note3);
    return;
}
```

Portées des variables (6)

```
note1 est 1
note2 est 1
note3 est 3
Dans f note1 est 2
Dans f note2 est 2
Dans f note3 est 2
note1 est 2
note2 est 1
note3 est 3
```

Portées des fonctions (1)

- Une fonction f peut être appelée dans n'importe quelle fonction définie après la définition de f. Une fonction ne peut pas être appelée en dehors de toute fonction.

```
#include <stdio.h>
int f1(int y){
      int a = 5;
      return a + x ;
}
int f2(int x){
      return x*x;
}
int f3(int x){
      return x* f2(x);
}
int main(){
      int res = f1(7);
      printf("%d \n %d \n", f2(res), f3(res))
      return 0;
 }
```

On ne peut PAS appeler f2 et f3

On peut appeler f1
On ne peut PAS appeler f3

On peut appeler f1 et f2

On peut appeler f1, f2 et f3

Portées des fonctions (2)

- Une fonction f2 peut être définie localement à une autre fonction f1.
- La définition de f2 doit être incluse dans f1. f2 n'est utilisable que dans f1.

```
#include <stdio.h>
int f1(int x){
      int f2(int x){
              return x*x;
      }
      return x*f2(x);
}
int main(){
      int res = f1(7);
      printf("%d \n %d \n",res)
      return 0;
 }
```

Portée de f2

A éviter Absolument !

On peut appeler f1
On ne peut PAS appeler f2

Passage de paramètres

- L'appel d'une fonction avec des arguments (paramètres effectifs) provoque l’exécution complète de la fonction.

- On s’intéresse ici au passage des valeurs des paramètres, c'est à dire à la transmission de la valeur d'un des paramètres à la fonction.

- **Deux modes** :
  - Passage par valeur ;
  - Passage par variable ;

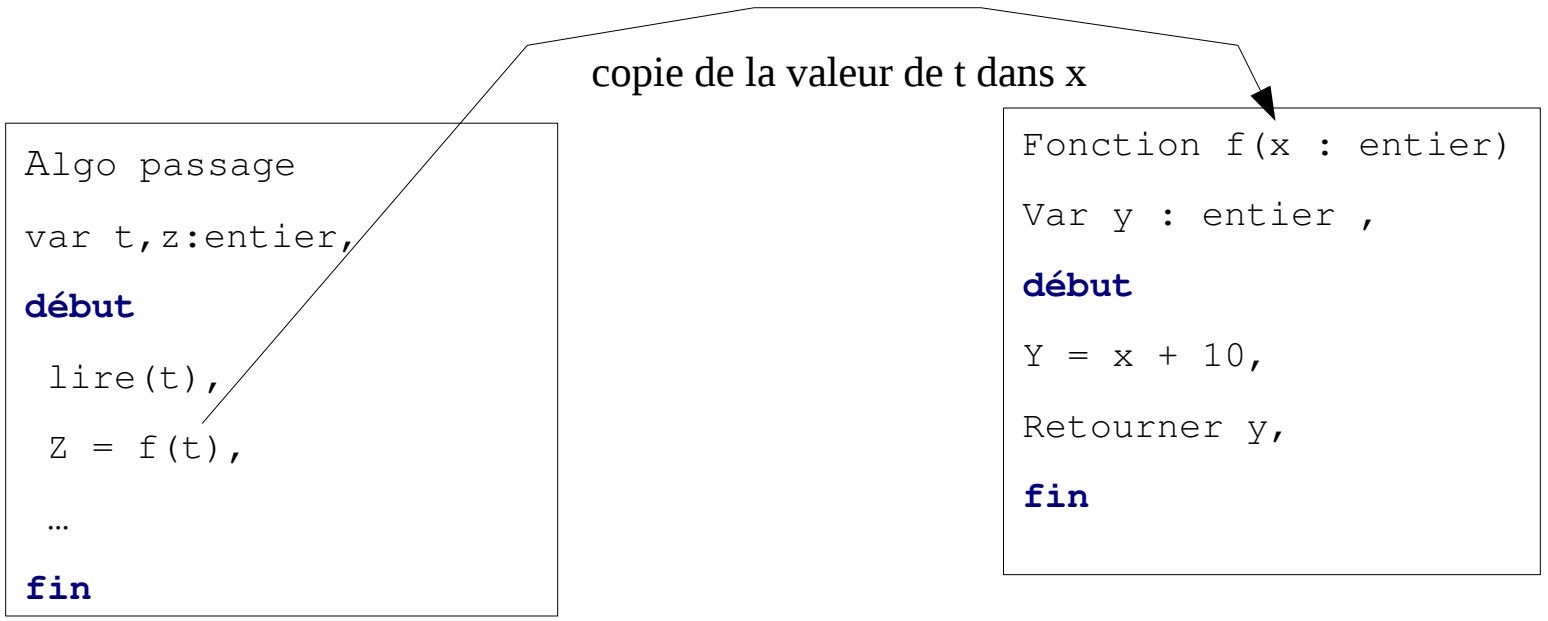
Passage par valeur (1)

- **Définition** : on appelle passage par valeur quand seule la valeur d'un paramètre effectif est connue de la fonction
  - Les valeurs transmises sont recopiées dans la fonction appelée, on travaille sur cette copie.

- **Principe** :
  - la fonction appelante fait une copie de la valeur passée en paramètre,
  - passe cette copie à la fonction appelée à l'intérieur d'une variable créée dans l'espace mémoire
  - cette variable est accessible de manière interne par la fonction à partir de l'argument formel correspondant.

→ **Le C ne permet de faire que des passages par valeur**

Passage par valeur (2)



- Il y a recopie de la valeur de l'argument dans une variable locale à la fonction :
  - Toutes les modifications effectuées sur le paramètre formel n'affectent que cette valeur locale et **ne sont pas visibles dans l'algorithme appelant**.

- La fonction ne travaille que sur **une copie** qui va être supprimée à la fin de la fonction.

Passage par valeur (3)

- **Que se passe-t-il ?**

```
#include <stdio.h>
void mystere(int y){
    y = 3;
}

int main(){
    int y = 7;
    printf("%d \n", y);
    mystere(y);
    printf("%d \n",y) ;
    return 0;
}
```

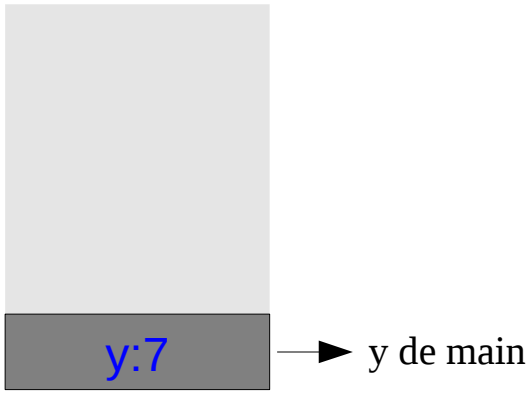
Quelles sont les valeurs affichées ?

Passage par valeur (4)

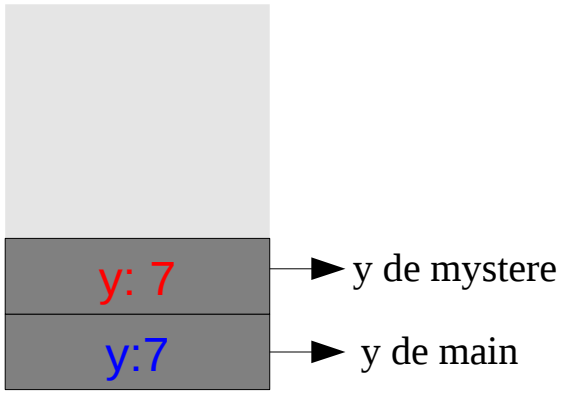
- **Que se passe-t-il ?**

```
#include <stdio.h>
void mystere(int y){
      y = 3;
}
int main(){
      int y = 7;
      printf("%d \n", y);
      mystere(y);
      printf("%d \n",y) ;
      return 0;
}
```

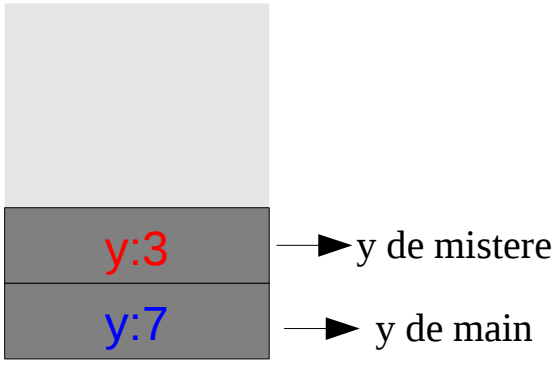
1 - Avant mystere:



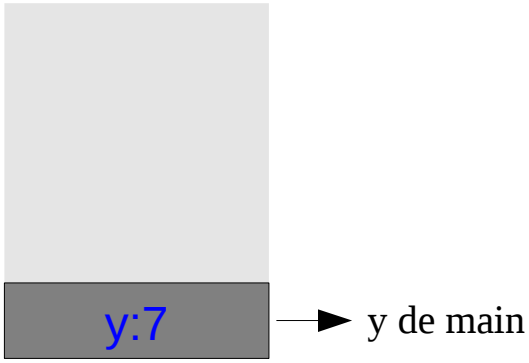
2 – Entrée dans mystere:



3 – avant de sortir de mystere:



4 – Après mystere :



Passage par valeur (5)

```
procédure échange(a:entier, b:entier)
var tmp : entier ;
début
Afficher("a = %d, b = %d",a, b) ;
tmp = a ;
a = b ;
b = tmp ;
Afficher("a = %d, b = %d",a, b) ;
fin
```

```
Algorithme principal
var a, b : entier ;
début
a = 6, b = 1 ;
afficher("avant : a = %d, b = %d",a, b) ;
échange(a,b) ;
afficher("après : a = %d, b = %d",a, b) ;
fin
```

Passage par variable (1)

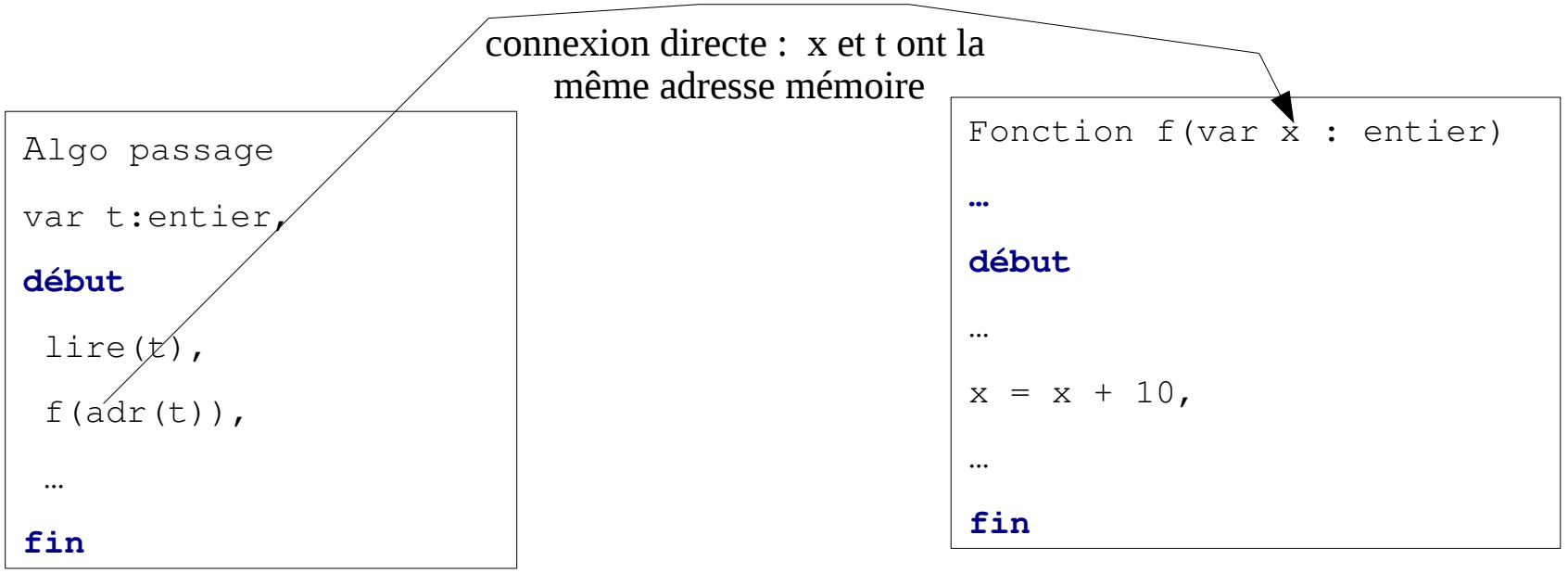
- **Définition** : on appelle passage par variable quand le paramètre effectif qui est transmis à la fonction lors de l'appel est **l'adresse de la variable**.

➔ La fonction ne travaille plus sur une copie de l'objet mais sur l'objet lui-même, puisque elle en connait l'adresse.

➔ Le paramètre effectif est alors l'adresse de la variable.

- **Principe** :
  - la fonction appelée range l'adresse transmise dans un paramètre approprié (de type adresse) qui est une variable locale à la fonction appelée.
  - la fonction appelée a maintenant accès, via ce paramètre à la variable de la fonction appelante.

Passage par variable (2)



- Toute modification sur un paramètre transmis par adresse (précédé par le mot **var**) entraine la modification directe de l'argument correspondant.

Passage par variable (3)

```
Procédure sum1(x:entier,y:entier, z:entier)
début
z = x+y ;
fin

Procédure sum2(x:entier,y:entier,var z:entier)
début
z = x+y ;
fin
```

```
Algorithme principal
var a, b, c : entier ;
début
a = 10, b = 20, c = 0 ;
sum1(a,b,c) ;
afficher("c = %d",c) ;
sum2(a,b,adr(c)) ;
afficher("c = %d",c) ;
fin
```

- Lors de l'appel de la procédure `sum2`, le troisième paramètre correspond, cette fois ci, à l'**adresse de la variable** `c`.
- Le paramètre formel `z` est maintenant **une variable de type adresse** destinée à recevoir une adresse.

Sous-algorithmes

1. Motivation, définitions,et exemples

2. Retour aux tableaux :procédures et fonctions de manipulation de tableaux

Retour sur les tableaux (1)

- Il est interdit de définir des fonctions qui retournent des tableaux !
- En revanche, un tableau peut être passé en paramètre d'une fonction.
- Déclaration : il y a deux manières :
  - int tab[] ;
- Passage d'un tableau en paramètre :
  - void lire_tableau(int tab[],int nb) ;
- La présence d'un argument de type tableau dans le prototype de la fonction n'entraîne aucune réservation mémoire :
  - `Void f (int tab[10])` => la dimension 10 n'a aucune signification sur le compilateur.

Tableaux en arguments des fonctions (1)

- Choisir entre **int tab[]** ou **const int tab[]** :

```
Void affiche_tab(int tab[],int nb ){
  int i ;
  for (i=0 ; i<nb ; i++) printf("%d",tab[i]);
 return ;
}
```

- Si `affiche_tab` ne modifie pas la valeur de tab, on peut utiliser le quantifieur **const :**

- **`Void affiche_tab(const int tab[], int nb )`** :
  - tab est constant .

Retour sur les tableaux (2)

```
#include <stdio.h>
#define Max 100
int main() {
     int T[TAILLE] ;

{remplissage du tableau}

{vérification de la saisie}

{affichage du contenu du tableau}

{somme des éléments du tableau}

{recherche d 'un élément dans le tableau}

Return 0 ;
}
```

Saisie d'un tableau 1D (1)

```
Void saisieTab1D (int tab[ ], int nbElt) {
   int i ;
   for (i =0 ; i < nbElt ; i++){
     printf("Donnez la valeur n° %d",i)
     Scanf(''%d'',&tab[i]) ;
    }
    return ;
}
```

Saisie d'un tableau 1D (2)

```
#include<stdio.h>
#define max 100

Void saisieTab1D(int tab[],int nbElt);

int main(void)

{

  int n; int t[max] ;

  printf("entrez n: \n") ;

  scanf("%d", &n) ;

  saisieTab1D(t,n) ;

  for(i=0; i<nb; i++)
    printf("t[%d]:%d",i,t[i]) ;

  return 0;
}
```

Somme des éléments d'un tableau 1D (1)

```
int SommeTab1D(int tab [], int nbElt)
{
   int i = 0, somme = 0 ;
   while (i < nbElt) {
      somme ← somme + tab[i ]
            i++
   }

    return somme
}
```

Somme des éléments d'un tableau 1D (2)

```
#include<stdio.h>
#define max 100

Void saisieTab1D(int tab[],int nbElt);
int SommeTab1D(int tab[],int nbElt) ;

int main(void)

{
  int n, somme; int t[max] ;

  printf("entrez n: \n") ;

  scanf("%d", &n) ;

  saisieTab1D(t,n) ;

  for(i=0; i<nb; i++)
    printf("t[%d]:%d",i,t[i]) ;

  Somme = SommeTab1D(t,n);

  return 0;
}
```

Affichage des éléments d'un tableau 1D (1)

```
Void affiche_tab(int tab[],int nbElt) ){

  int i ;

  for (i=0 ; i<nbElt ; i++)
      printf("tab[%d]:%d",i,tab[i]) ;

 return ;

}
```

Somme des éléments d'un tableau 1D (3)

```
#include<stdio.h>
#define max 100

Void saisieTab1D(int tab[],int nbElt);
int SommeTab1D(int tab[],int nbElt) ;

int main(void)

{
  int n, somme; int t[max] ;

  printf("entrez n: \n") ;

  scanf("%d", &n) ;

  saisieTab1D(t,n) ;

  affiche_tab(t,n) ;

  Somme = SommeTab1D(t,n);

  return 0;
}
```

Tableaux 2D

- Un tableau à deux dimensions est, par définition, un tableau de tableaux ;

  `int tab[10][15] //` un tableau d'entiers à deux dimensions

- `tab[i]`, pour `i` $\in$`[0..9]`, correspond à l'adresse du premier élément de la ligne d'indice `i` :

  `tab[i]` $\Leftrightarrow$ `&tab[i][0]`

Tableaux 2D en arguments des fonctions

- Pour qu'une fonction puisse accéder à un tableau 2D reçu en argument, il est nécessaire que la seconde dimension soit connue et fixée dans l'en-tête de la fonction, sous forme d'expression constante.
  - `Void affiche_tab(int tab[][15],int nb)`

- La déclaration suivante est incorrecte :
  - Void affiche_tab(int tab[ ][ ], int nb)

Saisie d'un tableau 2D (1)

```
// Exemple de saisie d'une matrice carrée

void saisieTab2D (int dim, int tab[][dim]) {
   int i,j;

   for (i=0 ; i < dim ; i++)
      {
         for (j=0 ; j < dim ; j++)
            {
               printf("M[%d][%d] =",i+1,j+1) ;
               Scanf(''%d'',&tab[i][j]) ;
            }
      }
}
```

Affichage des éléments d'un tableau 2D

```
Void afficheTab2D(int dim, int tab[][dim]){
  int i,j;
  for (i=0 ; i<dim ; i++)
    {
     for (j=0 ; j<dim ; j++)
        {
            printf("M[%d][%d] = %d",i+1,j+1,tab[i][j]);
        }
    }
}
```

Saisie d'un tableau 2D (2)

```
#include<stdio.h>

Void saisieTab2D(int dim,int tab[][dim]);

int main(void)

{

  int n;

  printf("entrez la taille de la matrice: \n") ;

  scanf("%d", &n) ;

  int matrice[n][n] ;

  saisieTab2D(n,matrice) ;

  afficheTab2D(n,matrice) ;

  return 0;
}
```