esprit
Se former autrement

Programmation Procédurale 1

Equipe Algorithmique & Programmation

Année universitaire : 2021 - 2022

cdio
CONCEIVE DESIGN IMPLEMENT OPERATE

Enropean Accreditation of Engineering Programmes EUR-ACE®

Délivré par Cti Commission des Titres d'Ingénieur

CONFÉRENCE DES GRANDES ÉCOLES

Objectifs

A la fin de ce chapitre, l'étudiant sera capable de:

- Définir une fonction
- Comprendre le mécanisme d'appel d'une fonction
- Analyser un problème donné à l'aide des fonctions:
  - Identifier les fonctions associées aux différentes tâches demandées
  - Identifier les paramètres d'entrées/sorties

Objectifs

- Concevoir une solution au problème en organisant les fonctions identifiées dans la phase d'analyse

- Développer un programme structuré par les fonctions organisées selon la phase de conception

Les fonctions en C

Plan

- Définition et utilité
- Définition d’une fonction
- Appel d’une fonction
- Fonctions et tableaux
- Les paramètres d’une fonction
- Variables locales et variables globales.

Définition et utilité

Programmation modulaire en C

- Découper le programme en plusieurs parties : sous-programmes ou modules.
- Chaque module peut, si nécessaire, être décomposé à son tour en modules plus élémentaires ;
- Ce processus de décomposition pouvant être répété autant de fois que nécessaire : c'est le principe de la programmation structurée.
- Regrouper dans le programme principal les instructions décrivant les enchaînements des modules.
- La seule sorte du module existant en langage C est la fonction.

Définition d’une fonction

Une fonction est un bloc d’instructions structurées autour d’un nom générique appelé "le nom de la fonction".

Ce bloc d'instructions peut éventuellement avoir besoin de valeurs pour pouvoir travailler sur des données qu'on lui transmet :

on dit que la fonction peut avoir besoin de paramètres.

Intérêts des fonctions

- Améliorer la lisibilité du programme.
- Eviter des séquences d'instructions répétitives.
- Faciliter la maintenance.
- Faciliter la conception.

•Partager des outils communs qu'il suffit d'avoir écrits et mis au point une seule fois Faciliter la réutilisabilité.

Intérêts des fonctions

- Possibilité de découper le programme source en plusieurs parties stockées dans des fichiers sources séparés. Chaque fichier source, appelé par abus de langage "module", regroupe une ou plusieurs fonctions et peut être compilé séparément

 Faciliter le travail en équipe + Développement de grosses applications.

Caractéristiques des fonctions

- Dans un programme, on distingue :
  - des fonctions **externes** intégrées dans des bibliothèques externes : printf(), scanf(), …
  - des fonctions **internes** développées dans le programme : ce sont des fonctions personnalisées qui apportent une structuration au programme et qui peuvent être appelées dans plusieurs fonctions.
- Une fonction peut fournir un unique résultat (sa valeur de retour), comme ne pas en fournir du tout.

Caractéristiques des fonctions

- Une fonction peut retourner un résultat scalaire ou une structure
- La valeur d'une fonction peut dans certains cas ne pas être utilisée (comme les valeurs de retour de printf() et scanf())
- Une fonction peut modifier les valeurs de certains de ses arguments (voir chapitre Pointeurs)

Utilisation des fonctions

1. **Définition de la fonction**

```
#include <stdio.h>
void NomFonction()
{
 ……….
}
```

Fonction appelante

```
void main()
{
   int a, b, valmax;
   printf(“ Tapez deux entiers: “);
   scanf(“%d %d”, &a, &b);

   NomFonction ();
   ……..
}
```

Appel de la fonction

2. **Appel de la fonction par une autre fonction,** ici par la fonction principale main().

Définition d’une fonction

Définition d'une fonction

Syntaxe d'une fonction sans type de retour:

```
void  nom_Fonction  (type_1 paramètre1, type_2 paramètre2,….. )
{
      //  déclaration des variables locales;
      //  instructions;
         …
}
```

Définition d'une fonction

Syntaxe d'une fonction avec type de retour:

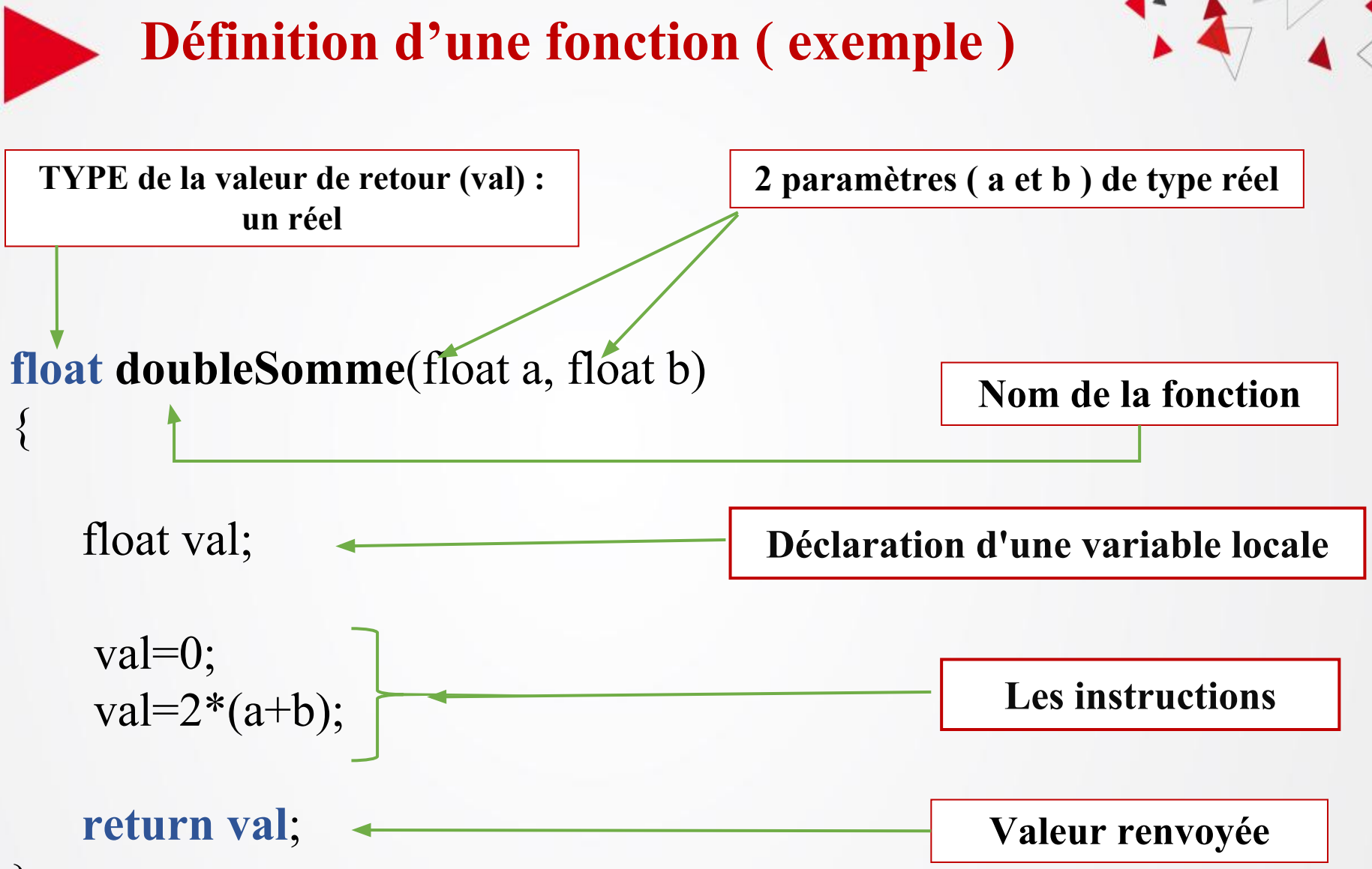
```
type de retour nom_Fonction  (type_1 paramètre1, type_2 paramètre2,…..
)
{
      //  déclaration des variables locales;
     //   instructions;
        ….
        return resultat;
 }
```

Définition d’une fonction

Dans la définition d'une fonction, nous indiquons:

1. Le nom de la fonction.
2. Le type et les noms des paramètres de la fonction.
3. Le type du résultat fourni par la fonction.
4. Les données locales à la fonction.
5. Les instructions à exécuter.

Définition d'une fonction (exemple)



Définition d'une fonction (remarques)

- Les noms des paramètres et de la fonction sont **des identificateurs** qui correspondent aux mêmes restrictions définies pour les identificateurs des variables.

- Si la fonction n'a pas de paramètres, déclarer la liste des paramètres comme **void** ou **ne rien mettre** entre les ( ).

Exemple:

```
void afficherBonjour ()
{
    printf ("Bonjour\n");
}
```

Il **est interdit de définir** des fonctions **à l'intérieur** d'une autre fonction.

Définition d’une fonction (type de retour)

Une fonction retourne une valeur avec l'instruction **return.**

return: permet de spécifier la fin de la fonction en cours et d’attribuer comme valeur de cette fonction la valeur d'une expression.

Plusieurs instructions return peuvent apparaître dans une fonction.

Le retour au programme appelant sera alors provoqué par le *premier return* rencontré lors de l'exécution.

Définition d'une fonction (type de retour)

Le type de retour d'une fonction peut être :

- Un type arithmétique (entier ou rationnel)
- Une structure
- Void

Si le type d'une fonction n'est pas déclaré explicitement, elle est automatiquement du type **int.**

Exemple:

```
int max ( )
{ ……
}
```

ou

```
max ( )
{ ……
}
```

Les paramètres d'une fonction sont optionnels.
Les parenthèses sont obligatoires.

Appel d’une fonction

Appel d'une fonction

L'appel des fonctions dépend du type de retour :

Cas 1: La fonction n'a pas un résultat à retourner :

Syntaxe:

nomDeFonction (param1, param2..);

Cas 2: La fonction a un résultat à retourner :

Syntaxe:

val = nomDeFonction(param1, param2..);

Où « **val** » est une variable de même type que le type de retour de la fonction.

Appel d'une fonction

Cas 1: La fonction n'a pas un résultat à retourner:

1. Lors de l'implémentation de la fonction:

   - La fonction a le type void dans sa déclaration
   - Aucune instruction return n'est nécessaire

```
void NomFonction()
{
  // instructions
}
```

1. À l'appel de la fonction:
   - L'appel de la fonction est une instruction

```
NomFonction
();
```

Cas 2: La fonction a un résultat à retourner:

1. Lors de l'implémentation de la fonction:

   - La fonction a un type défini à sa déclaration
   - La dernière instruction est un return d'une valeur de ce type

```
int NomFonction
()
{
   return Valeur;
}
```

1. À l'appel de la fonction:

   - La fonction est appelée dans une expression

```
A = NomFonction
();
```

Appel d'une fonction

- Il n'est pas obligatoire de stocker le résultat d'une fonction dans une variable.
- On peut appeler une fonction directement dans une autre fonction .

Exemple1:

```
float doubleSomme (float x, float  y)
{
      float val=0;
      val=2*somme(x,y);    //appel de la  fonction somme qui retourne la somme  de x et y
      return val;
}
```

Exemple2:

```
printf ("%f",doubleSomme(x,y));
```

Appel d'une fonction

```
#include <stdio.h>
void calculerMoy ( );      // Déclaration
void main( )
{
      int i;
      for (i = 1; i<=50; i++)
        calculerMoy ( );     // Appel
}

void calculerMoy ( )     // Implémentation
{
      float N1, N2, M;

      printf ("Note 1: ");
      scanf ("%f", &N1);
      printf ("Note 2: ");
      scanf ("%f", &N2);

      M=(N1*0.8)+(N2*0.2);
      printf ("Moyenne: %f", M);
}
```

```
#include <stdio.h>
void  calculerMoy ( );
float saisieNote ( );
void main()
{
    int i;
    for (i = 1; i<=50; i++)
        calculerMoy ( );
}
void calculerMoy ( )
{
    float N1, N2, M;
     N1 = saisieNote ();
    N2 = saisieNote ();
    M=(N1*0.8)+(N2*0.2);
    printf ("Moyenne: %f", M);
}

float saisieNote ( )
{
    float X;
    printf("Note:");
    scanf("%f", &X);
    return X;
}
```

Application

Exercices 1 et 2 de la série d'exercices

Fonctions et tableaux

Manipulation des tableaux par les fonctions

- Il est **interdit** de définir des fonctions qui ***retournent des tableaux*** !
- un tableau peut être passé en paramètre d'une fonction.

- *Passage d'un tableau en paramètre :*

Exemple: void lire_tableau(int tab[],int nb) ;

- Void f (int tab[10]) => la dimension 10 n'a aucune signification sur le compilateur.

Manipulation des tableaux par les fonctions

```
#include <stdio.h>
float MoyenneTableau (int v[], int Taille);
int main()
{
  int Valeurs[4] = {2, 4, 3, 8};
  float ValeurMoyenne;
  ValeurMoyenne = MoyenneTableau (Valeurs, 4);
  printf("Moyenne: %f\n", ValeurMoyenne);
}

float MoyenneTableau (int V[], int Taille)
{
  int i, Somme;
  Somme = 0;
  for (i = 0; i < Taille; i++)
    Somme = Somme + V[i];
  return (float)Somme / Taille;
}
```

Taille de v

Paramètres effectifs & paramètres formels

Paramètres effectifs et paramètres formels

1-Les paramètres formels:

- Se sont les paramètres qui figurent dans ***la définition*** de la fonction.
- Sont simplement des variables locales qui sont initialisées par les valeurs obtenues lors de l'appel.
- Sont utilisés lors de la définition de la fonction.

2-Les paramètres effectifs:

- Se sont les paramètres qui figures dans ***l'appel*** de la fonction et qui sont manipulés par celle-ci.
- Le type des paramètres effectifs doit être le même que celui des paramètres formels correspondants.
- L'appel d'une fonction est réalisé en invoquant le nom de la fonction suivi de la liste des paramètres effectifs

Paramètres effectifs et paramètres formels

```
#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;
}
```

Paramètres formels

Paramètres effectifs

Paramètres effectifs et paramètres formels

Déclaration

Fonction principale

Appel

Définition

Paramètres **formels**

Paramètres **effectifs**

```
#include <stdio. h>

int max( int v1, int v2);
void main()
{
  int a, b, valmax;
  printf(" Tapez deux entiers: ");
  scanf("%d %d", &a, &b);

  valmax = max( a, b);
  printf("Maximum des deux valeurs: %d\n",
  valmax);
}
int max( int v1, int v2)
{
  int maximum;
  if (v1 > v2) maximum = v1;
  else maximum = v2;
  return maximum;
}
```

Modes de passage des paramètres

Modes de passage des paramètres

Il existe deux modes de passage des paramètres:

1. le mode par copie de valeur
2. le mode par copie d'adresse.

Passage par copie de valeur

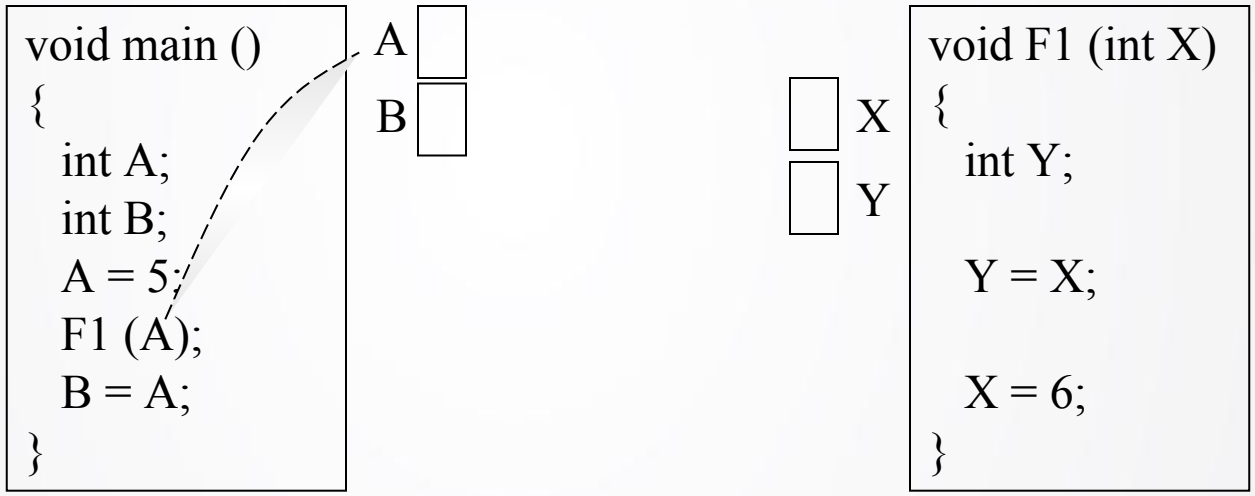
- Au moment de l'appel, la valeur du **paramètre effectif** est *copiée* dans la variable locale désignée par les **paramètres formels** correspondants.
- La fonction travaille sur **des copies** des paramètres **et ne peut pas** les modifier.

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 crée dans l'espace mémoire.
- Cette variable est accessible de manière interne par la fonction à partir de l'argument formel correspondant.

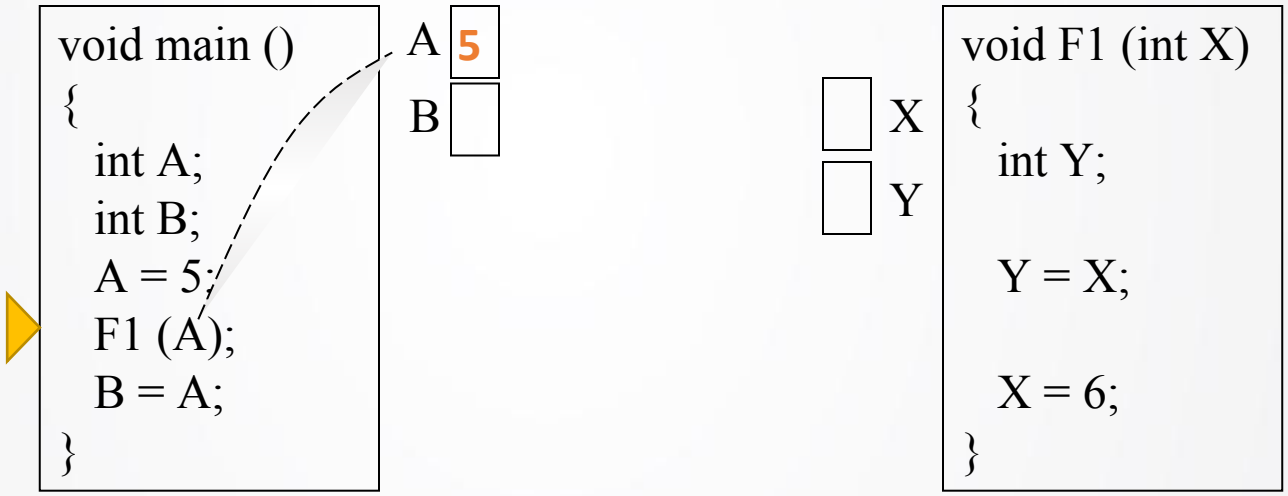
Passage par copie de valeur

Copie des valeurs des paramètres effectifs dans les paramètres formels.



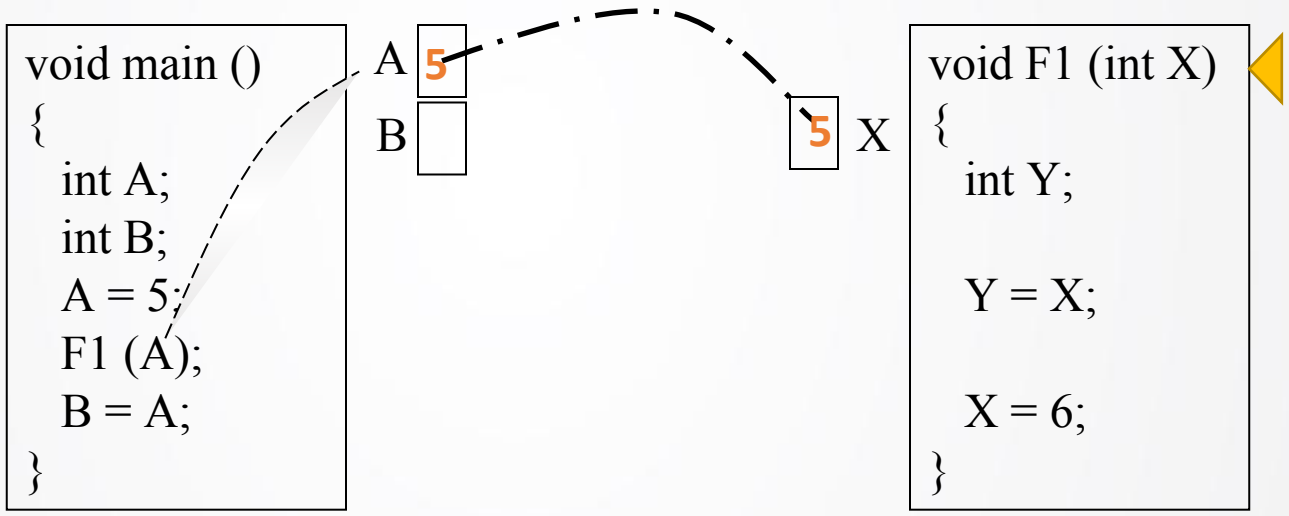
Passage par copie de valeur

Copie des valeurs des paramètres effectifs dans les paramètres formels.



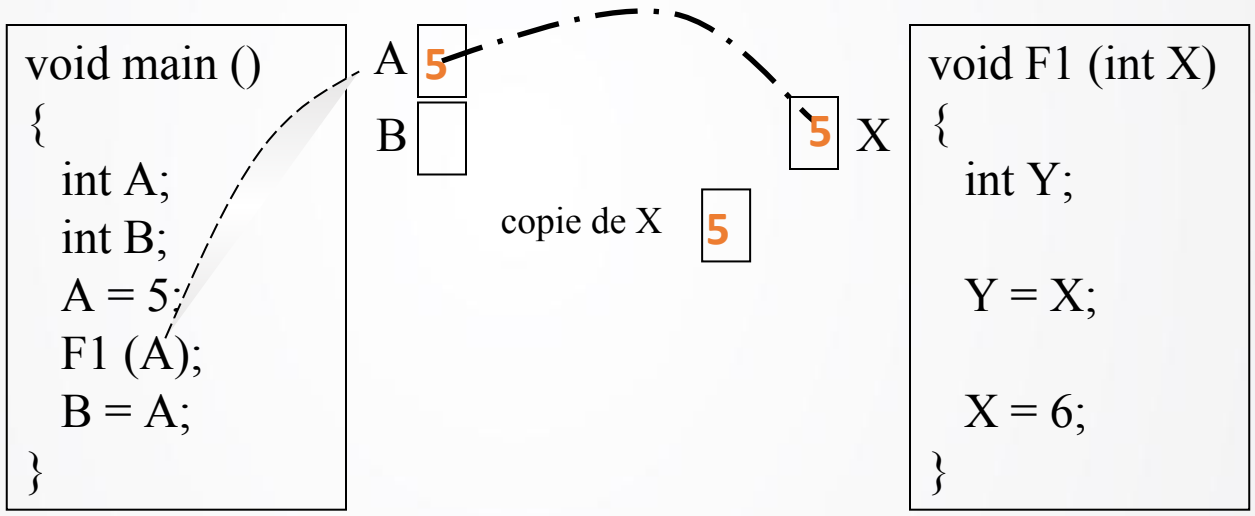
Passage par copie de valeur

Copie des valeurs des paramètres effectifs dans les paramètres formels.



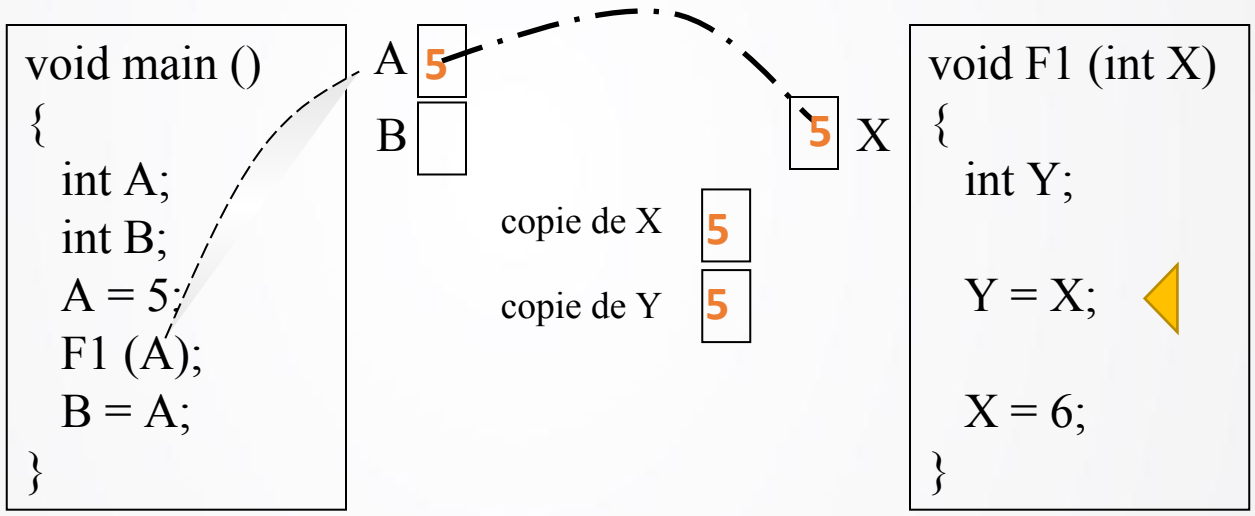
Passage par copie de valeur

Copie des valeurs des paramètres effectifs dans les paramètres formels.



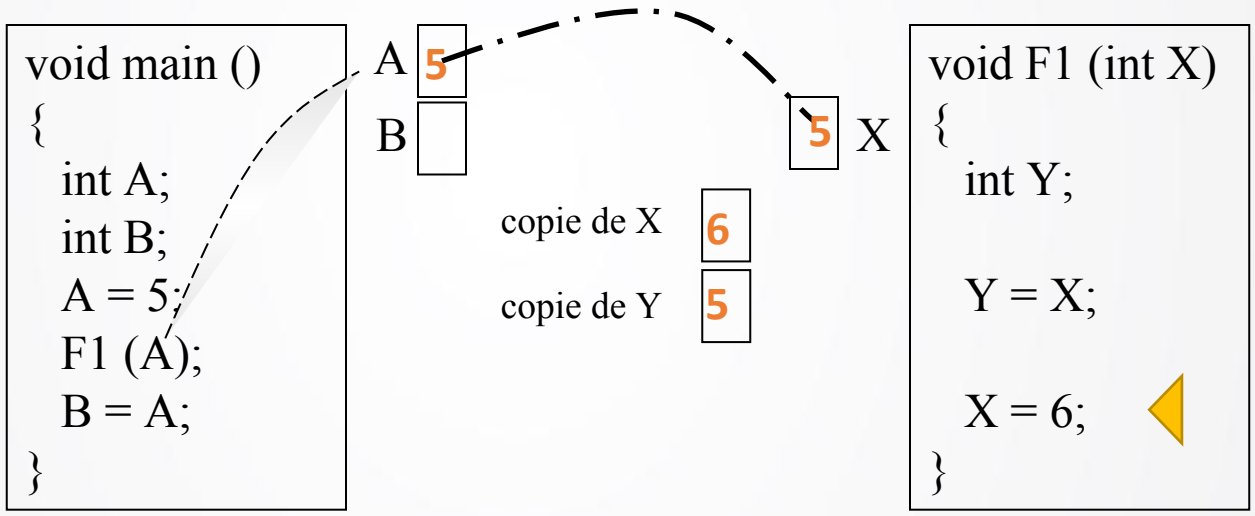
Passage par copie de valeur

Copie des valeurs des paramètres effectifs dans les paramètres formels.



Passage par copie de valeur

Copie des valeurs des paramètres effectifs dans les paramètres formels.



Passage par copie de valeur

```
int max(int v1,int v2);
int main()
{
   int a,b,m;
   printf("introduire a et b\n");
   scanf("%d%d",&a,&b);
   m=max(a,b);
   printf("\ n le maximum est %d",m);
}
int max(int v1,int v2)
{
   if(v1>v2)
      return v1;
   else
      return v2;
}
```

Le valeur de a est copié dans v1
La valeur de b est copié dans v2

L'appel de la fonction prend comme valeur la valeur renvoyée par l'instruction return

Passage par copie de valeur

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;
}
```

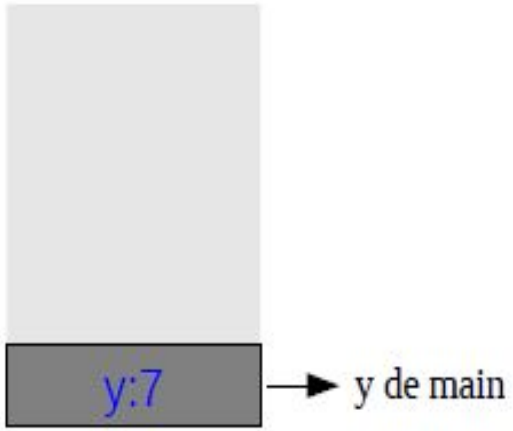
Quelles sont les valeurs affichées ?

Passage par copie de valeur

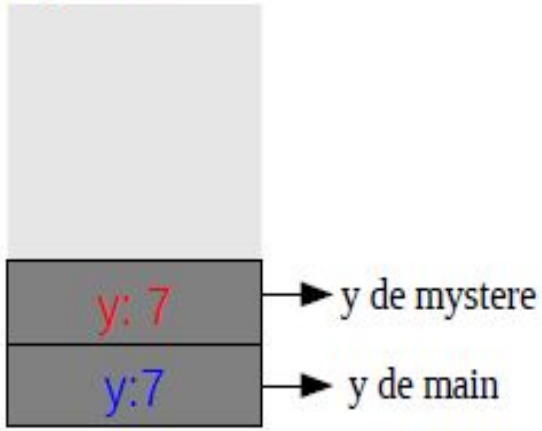
- **Que se passe-t-il ?**

```
#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;
}
```

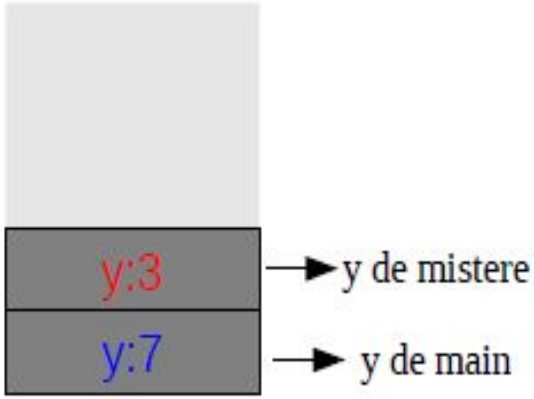
1 - Avant mystere:



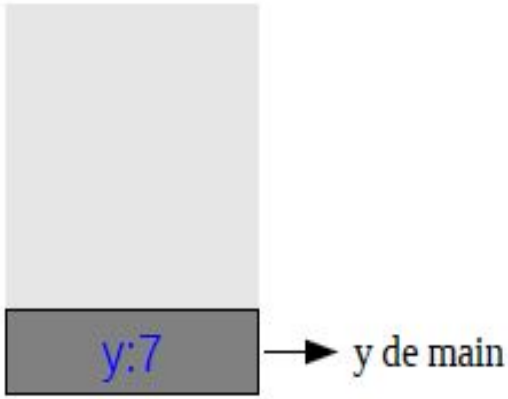
2 – Entrée dans mystere:



3 – avant de sortir de mystere:



4 – Après mystere :



Passage par copie de valeur

- Toutes les modifications effectuées sur le paramètre formel n'affectent que cette valeur locale et **ne sont pas visibles dans la fonction appelante**.
- La fonction ne travaille que sur **une copie** qui va être supprimée à la fin de la fonction.

Passage par adresse

•on appelle passage par adresse 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 adresse

- Toute modification sur un paramètre transmis par adresse entraine la modification directe de la variable correspondante.
- Pour le passage par adresse on utilise les pointeurs.

Variables locales & variables globales

Variables locales

- Ce sont les variables déclarées dans un bloc d'instructions.
- 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 valeurs seront perdues.

Exemple

```
void fonction( )
{
 char note = 'l';              // note est une variable locale
 printf("La note est %c\n",note);
 note='s';
}
main()
```

Variables globales

- 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.
- 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.

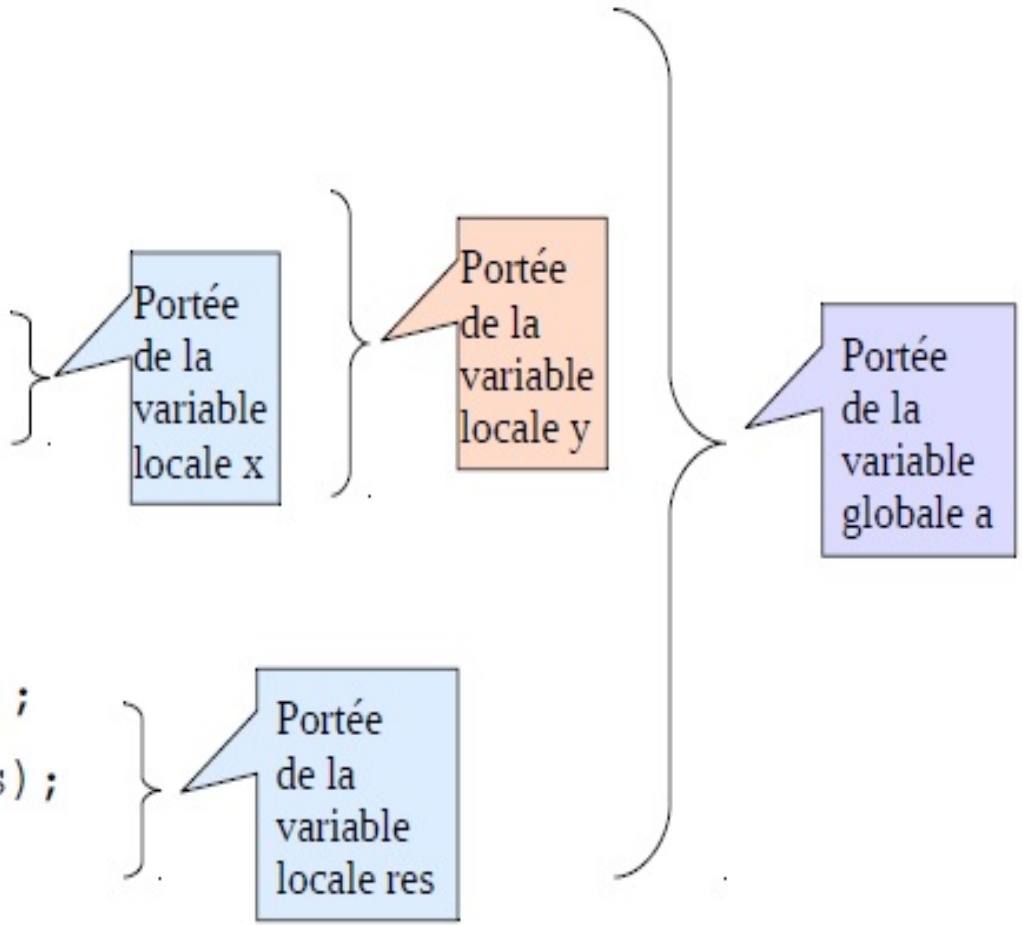
Structure d'un programme

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;
}
```

Portée d'une variable

```
#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;
}
```



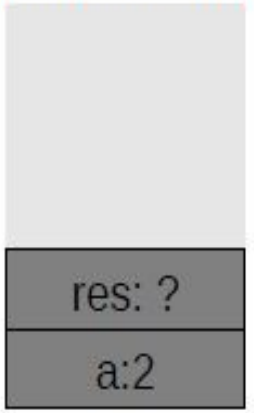
Portée d'une variable

Que se passe-t-il en mémoire ?

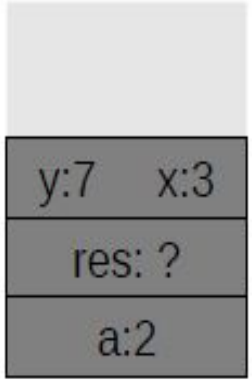
```
#include <stdio.h>
int a = 2;

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 - Chargement de la variable locale de main :



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



4 – Après appel de somme:

Exercice 3 de la série d'exercices

Références bibliographiques

1. Kernighan, B.-W. et Ritchie, D. (1988). *The C Programming Language* (2e éd.). Prentice Hall.

2. Delannoy, C. (2009). *Programmer en Langage C* (5e éd.). France, Paris: Eyrolles.