UNIVERSITÉ PARIS 8 VINCENNES-SAINT-DENIS

LIASD LABORATOIRE INFORMATIQUE AVANCÉE DE SAINT-DENIS

Cours 4

Programmation impérative

Emna Chebbi, Revekka Kyriakoglou

Plan du cours

Motivation

 Plus il y a de lignes dans notre programme, plus il est important d'utiliser des fonctions !

 Si nous ajoutons simplement toutes les instructions dans la main :

- le programme sera illisible.
- il comprendrait beaucoup de variables.
- nous ne pourrons pas faire de "debugging".
- ...

Déclaration, définition, utilisation

- **Déclaration :**

```
int maFonction(int);
```

- **Définition :**

```
int maFonction(int x)
{
    int a;
    ...
    return(a+x);
}
```

- **Utilisation :**

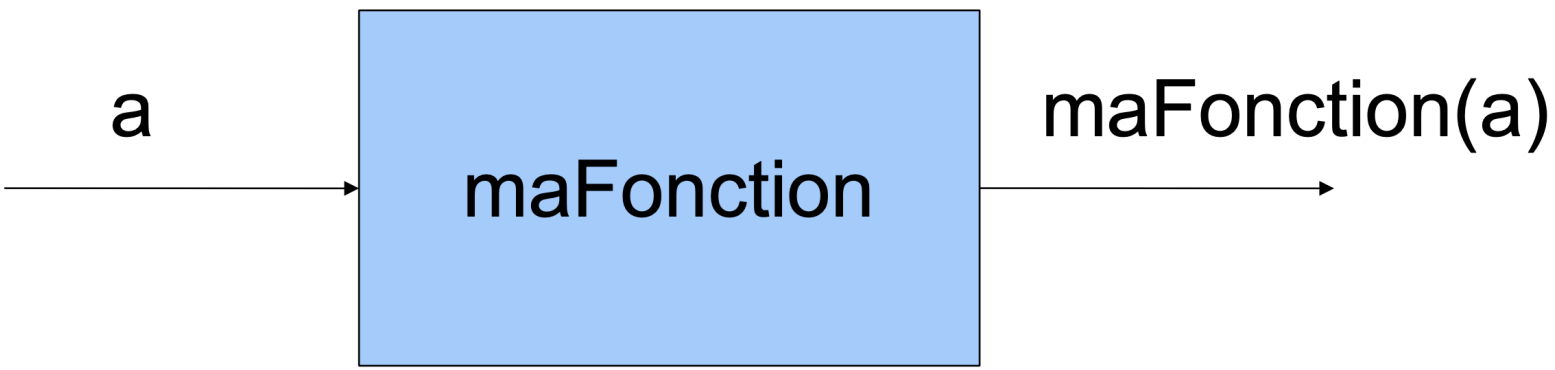
```
int x = 5;
int y = maFonction(x);
```

Type de retour, type du paramètre, nom

- **Type de retour :**
  int maFonction(int) ;
- **Type du paramètre :**
  int maFonction(**int**) ;
- **Nom :**
  int **maFonction**(int) ;


Ces trois paramètres s'appellent **Signature ou en-tête d'une fonction.**

Boites et flèches...



Exemple : Calcul du carré d'un nombre

Exemple

La fonction suivante calcule la valeur carée d'un nombre entier et retourne comme résultat un nombre entier.
Toute fonction peut faire appel à une autre fonction.

```
int carre( int a ){
    return (a*a);
}
int main(void){
    int n, x=0;
    printf("Saisir un entier n ");
    scanf("%d", &n);
    x = carre(n);
    printf("carre = %d \n", x);
return 0;
}
```

Types de retour

- **Les types pré-définis pour les variables :**

```
int, float, double, char, etc

int *, float *, char *, etc
```

- **Le type vide :**

```
void
```

Déclaration, définition, appel

Type de retour : void

Une **Procédure** ne contient pas un type de retour puisque contrairement à une fonction, une **Procédure** ne retourne rien.

- **Déclaration :**

```
void maProcedure(char);
```

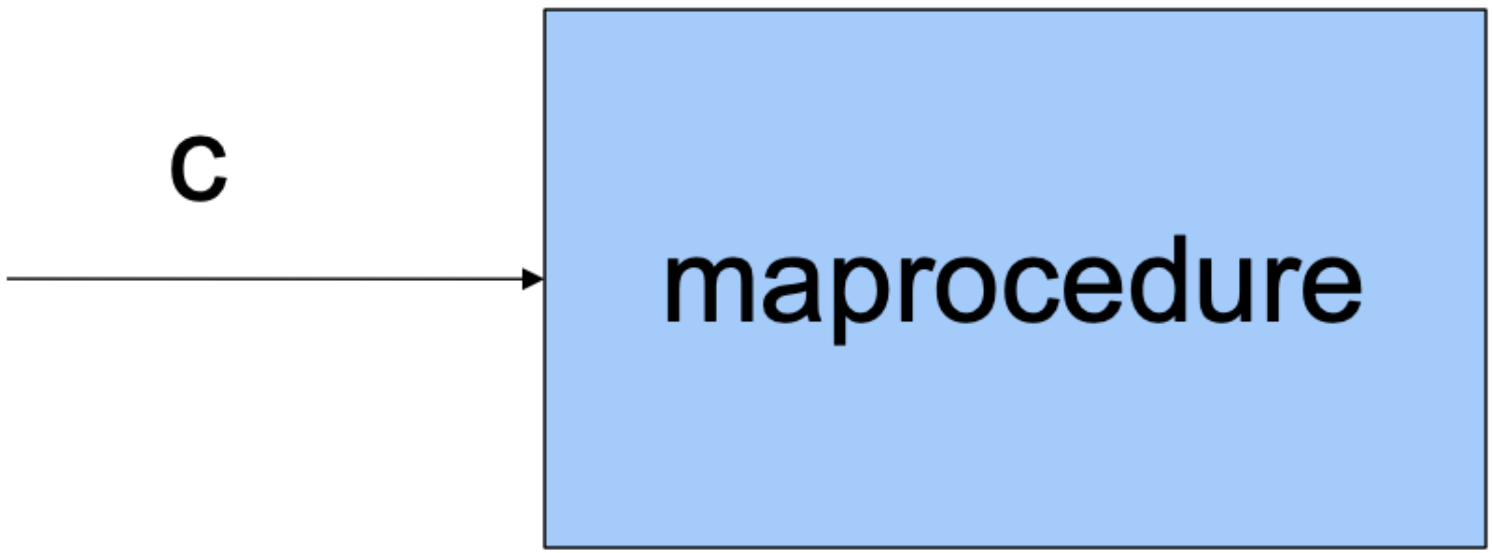
- **Définition :**

```
void maProcedure(char c)
{
    printf("%c", c);
}
```

- **Appel :**

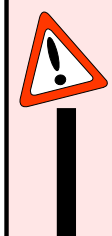
```
char c;
maProcedure(c);
```

Boites et flèches...



Type de retour, Structuration du programme

- **Type de retour :**
  void maProcedure(char) ;
- **Structuration du programme**
  Un traitement donné est écrit une fois et une seule dans le programme *(comme pour les fonctions).*



Possibilité d'avoir des paramètres en sortie

Une procédure peut donner un résultat en utilisant un *Passage de paramètre par adresse*.

Paramètres en entrée ou sortie

- **Paramètre en entrée**
  - La fonction ou la procédure en a besoin pour fonctionner.
- **Paramètre en sortie**
  - La fonction ou la procédure a pour objectif de lui donner une valeur.
  - Si le type du retour n'est pas void, une fonction C possède un paramètre de sortie obligé : le paramètre de retour.

Langage C et Paramètres en sortie


En langage C, il n'existe pas d'autre paramètre de sortie que le paramètre de retour.


Mais il existe le passage par adresse.

Passage de paramètres par valeur et par adresse

- **Passage de paramètre par valeur**
  - On passe la valeur du paramètre.
  - Implique que le paramètre qui est en entrée.
- **Passage de paramètre par adresse**
  - Aussi appelé passage par référence.
  - On passe l'adresse du paramètre.
  - Cela permet d'avoir un paramètre en sortie.

Which better

Si vous souhaitez ne transmettre que la valeur de la variable utilisez le **Passage par valeur** et si vous souhaitez avoir la modification de la valeur originale de la variable, utilisez le **Passage par référence ou adresse**.

Exemple : passage de paramètres par adresse

Exercice 1

Essayez de décrire ce qui se passe avec le code suivant. Que donne le dernier printf si la variable **n** est égale 6 :

```
void carre2(int a, int * b){
    *b = a*a;
}
int main(void){
    int n, x=0;
    printf("Saisir un entier n \n");
    scanf("%d", &n);
    carre2(n, &x);
    printf("carre2 = %d\n", x);
    return 0;
}
```

Boites et flèches...

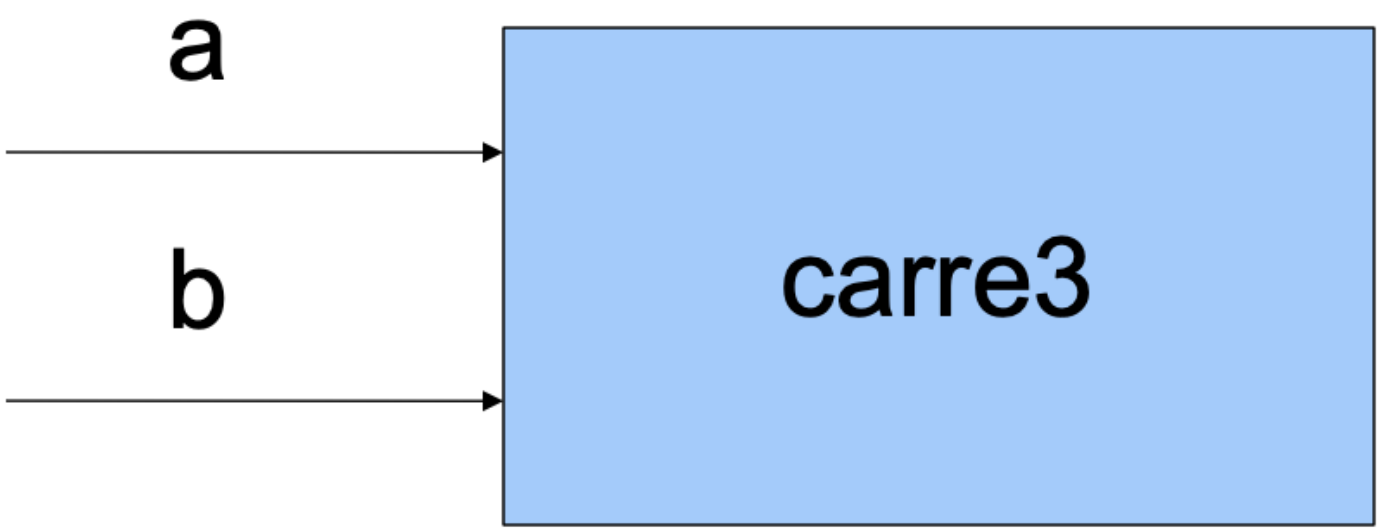
Exemple : passage de paramètres par valeur

Exercice 2

Essayez de décrire ce qui se passe avec le code suivant. Que donne le dernier printf si la variable **n** est égale 6 :

```
void carre3(int a, int b){
    b = a*a;
}
int main(void){
    int n, x=0;
    printf("Saisir un entier n \n");
    scanf("%d", &n);
    carre2(n, &x);
    printf("carre2 = %d\n", x);
    return 0;
}
```

Boites et flèches...



Définition, Exemple

- **Définition :**
  - Une méthode de description d'algorithmes qui permet à une procédure (ou une fonction) de s'appeler **elle-même**.
  - Permet généralement l'écriture des fonctions sous une forme concise et plus simple à comprendre.
  - Moins naturelle à concevoir.
  - Lorsque le problème traité peut se décomposer en une succession de sous-problèmes identiques, la récursivité est généralement bien indiquée
- **Exemple :**
  - Prenons l'exemple de la fonction factorielle() qui calcule la factorielle d'un entier. On rappelle ici le calcul de la factorielle de n :

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

Forme itérative : factorielle

Exemple

La forme itérative est l'implémentation classique (sans récursivité). Voici le code de la fonction factorielle() sans récursivité :

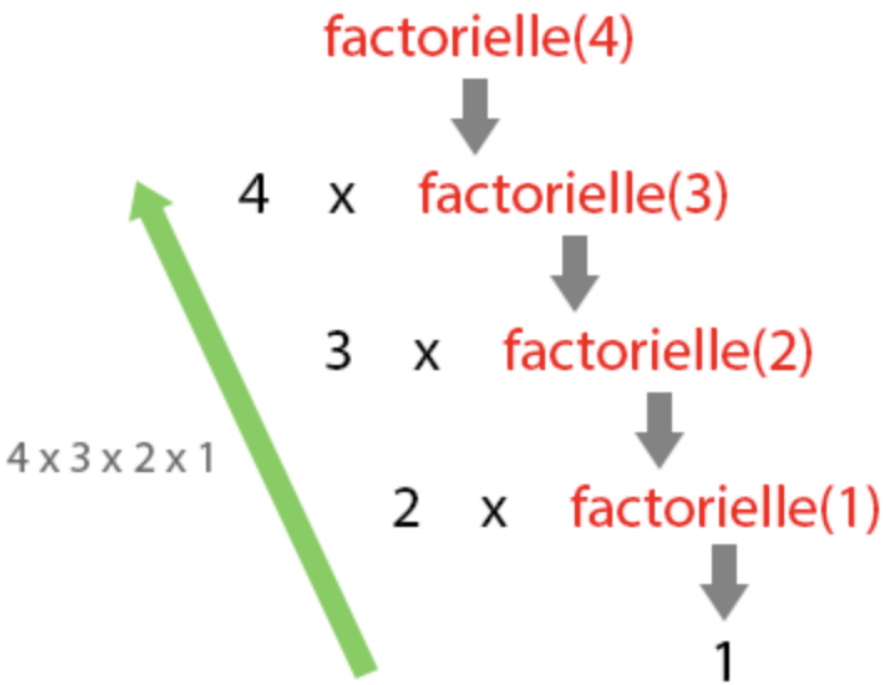
```
int factorielle (int n) {

  int i;
  int fact=1;
  for (i=2;i<=n;i++) {
    fact = fact*i;  /* Parcourt tous les termes
                       et multiplie fact par i */
  }
  return fact;
}
```

Boites et flèches...

Nouvelle logique !

Pour la **forme récursive**, nous allons nous appuyer sur une autre écriture de la factorielle :

$$n! = n \times (n-1)$$



Forme recursive : factorielle

Exemple

La forme récursive est généralement plus simple à comprendre et plus élégante.
Mais les appels récursifs occasionnent la sauvegarde du contexte (les valeurs des variables) avant chaque appel et sa restitution au retour de l'appel, ce qui peut légérement diminuer l'efficacité du programme.

```
int factorielle (int n) {

  if (n<=1)
    return 1;    //Si n <= 1, retourne 1 car
                 //!0=1 et !1=1
  return n*Factorielle(n-1); // Retourne n*!(n-1)
}
```

Définition, appel

- **Définition :**

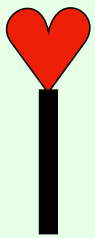
```
#define MA_MACRO(X, Y) X+19*Y
```

- **Utilisation :**

```
int i = MA_MACRO(4, 13);
```

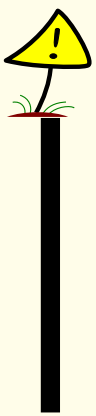
- **Fonctionnement :**
  gcc le pre-processeur remplace **MA_MACRO(a, b)** par **a + 19*b** partout dans le code source.

Macros : Avantages et Inconvénients



Avantages

- Rapidité d'exécution.
- Lisibilité du source (comme pour les fonctions).



Inconvénients

- Effet de bords indésirés tels que la surchage de mémoire.
- Expansion du code exécutable, puisque les macro sont créés par le preprocesseur.

Exercice général

Step - 1

Ecrire une fonction dont la signature est :

```
int deIntervalleANombre(int a, int b);
```

qui permet de :

- Demander à l'utilisateur de saisir un nombre entier dans l'interval [a, b].
- Retourner ce nombre.
- Renouveler la demande tant que ce nombre n'est pas dans l'intervalle [a, b].

Exercice général

Step - 1 : correction

```
int deIntervalleANombre(int a, int b){
  int x;
  do {
    printf("x ? (%d<=x<=%d) ", a, b);
    scanf("%d", &x);
  } while (x<a || x>b);
  return x;
}
```

Exercice général

Step - 2

Ecrire un programme principal main qui permet de :

- Demander un nombre N dans [0, 12] via la fonction

  `deIntervalleANombre.`

- Afficher la valeur de N ! via une fonction récursive.

Exercice général

Step - 2 : correction

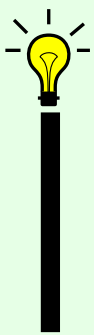
```
#include <stdio.h>
// ici les definitions des 2 fonctions
int main(void) {
    int n;
    n = deIntervalleANombre(0,12);
    printf("n = %d\n", n);
    printf("n! = %d\n", factorielle(n));
    return 0;
}
```

Définition, Structures de contrôle, Structures de données

- **Définition :**
  - **P** : un problème.
  - **M** : Solution pour résoudre le problème **P**.
  - Algorithme : description de la méthode **M** dans un algorithme.

- **Structures de contrôle :**
  - Branchement conditionnel.
  - Boucle (ou itération).
  - Le saut (goto)

- **Structures de données :**
  - Constantes.
  - Variables.
  - Tableaux.
  - Structures récursives .



Objectif

- Evaluer l'efficacité de la méthode **M**.
- Comparer **M** avec une autre méthode **M'**.

indépendamment de l'environnement (machine, ...).

Evaluation, Notions , Configurations caractéristiques

- **Evaluation :** Evaluation du nombre d'opérations élémentaires en fonction :
  - de la taille des données.
  - de la nature des données.

- **Notions :**
  - n : Taille des données.
  - T(n) : Nombre d'opérations élémentaires.

- **Configurations caractéristiques :**
  - Meilleur cas.
  - Pire des cas.
  - Cas moyen.

Evaluation de T(n) (séquence), Evaluation de T(n) (embranchement)

- **Evaluation de T(n) (séquence) :**
  Somme des coûts.

$$\left.\begin{array}{ll} \text{Traitement1} & T_1(n) \\ \\ \text{Traitement2} & T_2(n) \end{array}\right\} \quad T(n) = T_1(n) + T_2(n)$$

- **Evaluation de T(n) (embranchement) :**
  Max des coûts.

$$\left.\begin{array}{ll} \textbf{si } < \text{condition} > \textbf{ alors} & \\ \quad \text{Traitement1} & T_1(n) \\ \textbf{sinon} & \\ \quad \text{Traitement2} & T_2(n) \end{array}\right\} \quad max(T_1(n), T_2(n))$$

Evaluation de T(n) (boucle), Evaluation de T(n) (fonctions récursives)

Evaluation de T(n) (boucle) :

Somme des coûts des passages successifs

$$\left.\begin{array}{l}\textbf{tant que } < \text{condition} > \textbf{ faire} \\ \quad \text{Traitement} \quad T_i(n) \\ \textbf{fin faire}\end{array}\right\} \quad \sum_{i=1}^{k} T_i(n)$$

$T_i(n)$: coût de la $i^{\text{ème}}$ itération

souvent défini par une équation récursive

Evaluation de T(n) (fonctions récursives) :

fonction FunctionRecursive (n)

1 **si** $(n > 1)$ **alors**
2 FunctionRecursive($n/2$), coût $T(n/2)$
3 Traitement(n), coût $C(n)$
4 FunctionRecursive($n/2$), coût $T(n/2)$

Equation récursive

$$T(n) = 2 * T(n/2) + C(n)$$

si $C(n) = 1$ alors $T(n) = K \times n$
si $C(n) = n$ alors $T(n) = K \times n \times \log n$

Les principales classes de complexité , Exemple : permutation

- **Les principales classes de complexité :**

$O(1)$	temps constant
$O(\log n)$	logarithmique
$O(n)$	linéaire
$O(n \times \log n)$	tris (par échanges)
$O(n^2)$	quadratique, polynomial
$O(2^n)$	exponentiel (problèmes très difficiles)

- **Exemple : permutation :**

fonction permutation (S, i, j)

1	$tmp := S[i]$,	coût c_1
2	$S[i] := S[j]$,	coût c_2
3	$S[j] := tmp$,	coût c_3
4	**renvoyer** S	coût c_4

Coût total

$$T(n) = c_1 + c_2 + c_3 + c_4 = O(1)$$

Recherche sequentielle

- **Les principales classes de complexité :**

fonction recherche (x, S, n)

1 $i := 1,$
2 **tant que** $((i < n)$ **et** $(S[i] \neq x))$ **faire** (n fois)
3 $i := i + 1,$
4 **renvoyer** $(S[i] = x)$

Pire des cas : n fois la boucle

$T(n) = 1 + \sum_{i=1}^{n} 1 + 1 = O(n)$

Temps de calcul

Le temps de calcul approximatif selon la complexité et le nombre d'iterations de l'algorithme.

$Taille$	$\log_2 n$	n	$n \log_2 n$	n^2	2^n
10	$0.003\,ms$	$0.01\,ms$	$0.03\,ms$	$0.1\,ms$	$1\,ms$
100	$0.006\,ms$	$0.1\,ms$	$0.6\,ms$	$10\,ms$	$10^{14}\,siecles$
1000	$0.01\,ms$	$1\,ms$	$10\,ms$	$1\,s$	
10^4	$0.013\,ms$	$10\,ms$	$0.1\,s$	$100\,s$	
10^5	$0.016\,ms$	$100\,ms$	$1.6\,s$	$3\,heures$	
10^6	$0.02\,ms$	$1\,s$	$20\,s$	$10\,jours$	