

# Algorithmique et Structures de Données

Année universitaire  
2020-2021

Dr. Marwa CHAIEB

# Chapitre 4

## Fonctions et procédures

# Plan

3

- Partie 1: Les sous-programmes
- Partie 2: Les fonctions
- Partie 3: Les procédures
- Partie 4: Portée d'une variable
- Partie 5: Paramètres formels/effectifs
- Partie 6: Transmission de paramètres
- Partie 7: De l'algorithmique au langage C

# Plan

4

- Partie 1: Les sous-programmes
- Partie 2: Les fonctions
- Partie 3: Les procédures
- Partie 4: Portée d'une variable
- Partie 5: Paramètres formels/effectifs
- Partie 6: Transmission de paramètres
- Partie 7: De l'algorithmique au langage C

# Sous-programmes - Décomposer pour résoudre

5

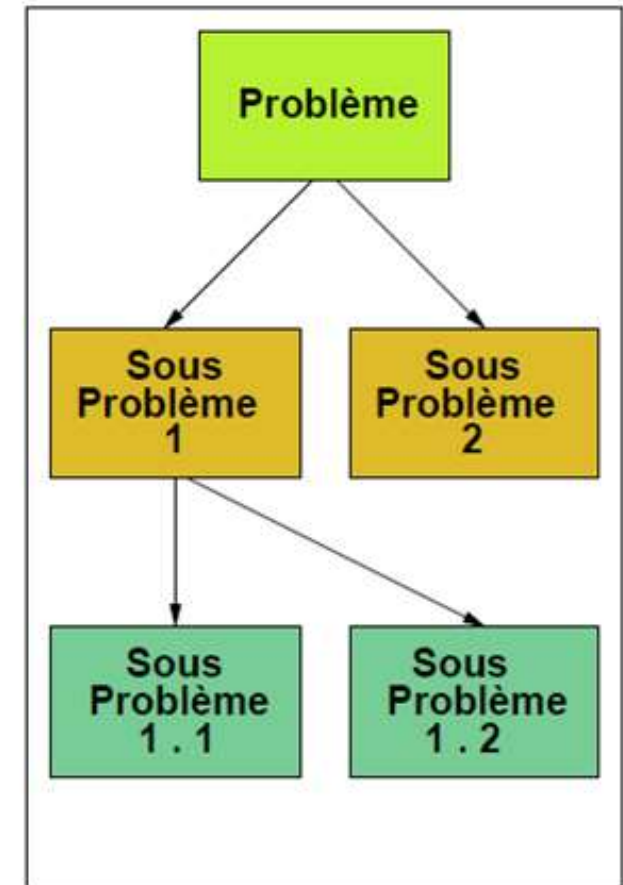
## ■ Analyse descendante

Approche incrémentale consistant à analyser la spécification d'un problème selon son arborescence

## ■ Décomposition / combinaison

à chaque niveau de l'arborescence :

- ◆ **Décomposer** le problème en un certain nombre de sous-problèmes qu'on supposera résolus
- ◆ **Combiner** les solutions des sous-problèmes pour résoudre le problème courant
- ◆ Résoudre les sous-problèmes par la même approche



# Sous-programmes - Décomposer pour résoudre

6

- Par exemple, pour résoudre le problème suivant :

**Ecrire un programme qui affiche les nombres parfaits compris entre 0 et une valeur n saisie par l'utilisateur.**

ex:  $28 = 1 + 2 + 4 + 7 + 14$ .

Cela revient à résoudre :

1. Demander à l'utilisateur de saisir un entier n
2. Savoir si un nombre donné est parfait
  - 2. 1. Savoir si un nombre est diviseur d'un autre nombre
  - 2. 2. Calculer la somme des diviseurs d'un nombre
3. Afficher les nombres parfaits entre 0 et n

# Sous-programmes - Décomposer pour résoudre

7

- Chacun de ces sous-problèmes devient un nouveau problème à résoudre.
- Si on considère que l'on sait résoudre ces sous-problèmes, alors on sait “quasiment” résoudre le problème initial.
- **Donc écrire un programme qui résout un problème revient toujours à écrire des sous-programmes qui résolvent des sous parties du problème initial.**
- En algorithmique il existe deux types de sous-programmes :
  - **Les fonctions**
  - **Les procédures**
- Un sous-programme est obligatoirement caractérisé par un nom (identifiant) unique
- Le programme qui utilise un sous-programme est appelé **le programme appelant.**

# Sous-programmes - Décomposer pour résoudre

8

- ❑ Une fonction rend un et un seul résultat



- ❑ Une procédure rend zéro ou plusieurs résultats





# Plan

9

- Partie 1: Les sous-programmes
- Partie 2: Les fonctions
- Partie 3: Les procédures
- Partie 4: Portée d'une variable
- Partie 5: Paramètres formels/effectifs
- Partie 6: Transmission de paramètres
- Partie 7: De l'algorithmique au langage C

# Les fonctions — Déclaration d'une fonction

10

**Fonction** *NomDeLaFonction (paramètres et leurs types) : type\_valeur\_retour\_fonction*

**Var** *variable locale 1 : type 1; . . .*

**début**

*instructions de la fonction*

**Retourner** *expression*

**FinFonction**

# Les fonctions — Déclaration d'une fonction

11

- **type\_valeur\_retour\_fonction** est le type du résultat renvoyé par la fonction
- **liste de paramètres** est la liste des **paramètres formels** donnés en entrée avec leurs types de la forme **p\_1: type\_1, ..., p\_n: type\_n,**
- ❖ Le corps de la fonction doit comporter une instruction de la forme :  
**retourner(expression);**

cette instruction met fin à l'exécution de la fonction et retourne *expression* comme résultat

# Les fonctions – Exemple 1

12

Écrire une fonction qui renvoie la moyenne de deux entier.

**Fonction** Moyenne(A : entier, B : entier) : réel

**Var** : Moy : réel

Moy  $\leftarrow (A + B) / 2$

**Retourner** Moy

**FinFonction**

# Les fonctions – Exemple 2

13

Écrire une fonction qui renvoie la factorielle d'un entier.

**Fonction** Factorielle(N : entier) : entier

**Var** : Fact, i : entier

Fact  $\leftarrow$  1

**Pour** i  $\leftarrow$  2 à N **Faire**

Fact  $\leftarrow$  Fact \* i

**Fin Pour**

**Retourner** Fact

**FinFonction**

# Les fonctions – Exemple 3

14

Écrire une fonction qui renvoie le maximum de deux réels.

**Fonction** Maximum(A : réel, B : réel) : réel

**Si** ( $A \geq B$ ) **Alors**

**Retourner** A

**Sinon**

**Retourner** B

**Fin Si**

**FinFonction**

# Les fonctions — Appel d'une fonction

15

- ❖ L'**appel** d'une fonction se fait par simple écriture de son nom dans le programme principal suivi des paramètres entre parenthèses ( les parenthèses sont toujours présentées même lorsqu'il n'y a pas de paramètre).

**NomDeLaFonction**(<liste de paramètres)

- ❖ **liste de paramètres** est la liste des **paramètres effectifs** de la forme  $q_1, \dots, q_n$ , avec  $q_i$  un paramètre de type *type*  $p_i$
- ❖ L'appel de la fonction retourne une valeur calculée en fonction des paramètres effectifs
- ❖ La liste des **paramètres effectifs** doit être **compatible** avec la liste des **paramètres formels** de la déclaration de la fonction
- ❖ Lors de l'appel, chacun des paramètres formels **est remplacé** par le paramètre effectif

# Les fonctions — Appel d'une fonction

16

- L'appel d'une fonction se fait par simple écriture de son nom dans le programme principale. Le résultat étant une valeur, devra être affecté ou être utilisé dans une expression

## Exemple:

**Fonction** Pair (n : entier ) : booléen

**Debut**

**retourner** (n%2=0)

**FinFonction**

## Algorithme AppelFonction

**Var:** c : réel, b : booléen

**Début**

b ← Pair(3)

....

**Fin**

- Lors de l'appel Pair(3), le **paramètre formel** n est remplacé par le **paramètre effectif** 3



# Les fonctions – Appel d'une fonction

17

```
fonction minimum2 (a,b : Entier) : Entier
```

```
debut
```

```
  si  $a \geq b$  alors
```

```
    retourner b
```

```
  sinon
```

```
    retourner a
```

```
  finsi
```

```
finFonction
```

```
fonction minimum3 (a,b,c : Entier) : Entier
```

```
debut
```

```
  retourner minimum2(a,minimum2(b,c))
```

```
finFonction
```

# Plan

18

- Partie 1: Les sous-programmes
- Partie 2: Les fonctions
- Partie 3: Les procédures
- Partie 4: Portée d'une variable
- Partie 5: Paramètres formels/effectifs
- Partie 6: Transmission de paramètres
- Partie 7: De l'algorithmique au langage C

# Les procédures – Déclaration

19

- Une **procédure** est un sous-programme semblable à une fonction mais qui **ne retourne rien**.

- Une procédure s'écrit en dehors du programme principal sous la forme :

**Procédure** *NomDeLaProcédure (paramètres et leurs types)*

**Var** *variable locale 1 : type 1; ...*

**début**

*instructions de la procédure*

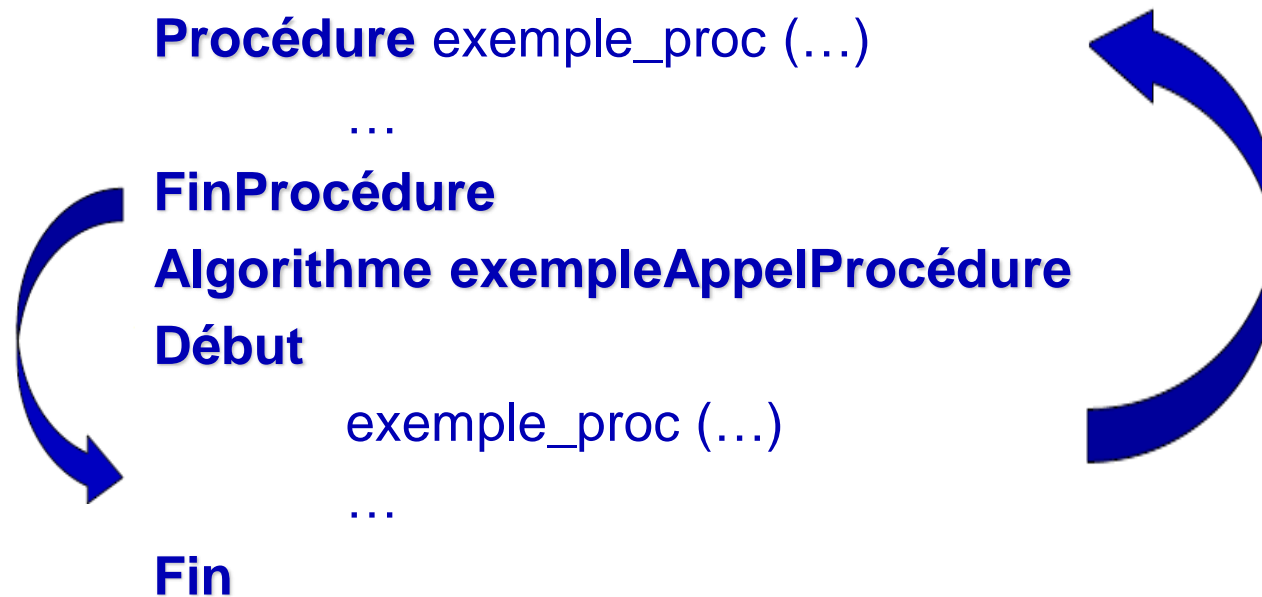
**FinProcédure**

- Remarque : une procédure peut ne pas avoir de paramètres.

# Les procédures – Appel

20

- L'appel d'une procédure, se fait dans le programme principal ou dans une autre procédure par une instruction indiquant le nom de la procédure :



- Remarque : contrairement à l'appel d'une fonction, **on ne peut pas affecter la procédure appelée ou l'utiliser dans une expression.**

# Plan

21

- Partie 1: Les sous-programmes
- Partie 2: Les fonctions
- Partie 3: Les procédures
- Partie 4: Portée d'une variable
- Partie 5: Paramètres formels/effectifs
- Partie 6: Transmission de paramètres
- Partie 7: De l'algorithmique au langage C

# Portée d'une variable

22

- ❖ La **portée d'une variable**: l'ensemble des sous-programmes où cette variable est connue (les instructions de ces sous-programmes peuvent utiliser cette variable)
- ❖ Une variable définie au niveau du programme principal (celui qui résout le problème initial) est appelée **variable globale**
- ❖ Sa portée est totale : **tout sous-programme du programme principal peut utiliser cette variable**
- ❖ Une variable définie au sein d'un sous programme est appelée **variable locale**
- ❖ La portée d'une variable locale est uniquement le sous-programme qui la déclare. Lorsque le nom d'une variable locale est identique à une variable globale, la variable globale **est localement masquée : Dans ce sous-programme la variable globale devient inaccessible**

# Plan

23

- Partie 1: Les sous-programmes
- Partie 2: Les fonctions
- Partie 3: Les procédures
- Partie 4: Portée d'une variable
- Partie 5: Paramètres formels/effectifs
- Partie 6: Transmission de paramètres
- Partie 7: De l'algorithmique au langage C

# Paramètres formels/effectifs

24

- Les paramètres servent à échanger des données entre le programme principale (ou le sous-programme appelant) et le sous-programme appelé.
- Les paramètres placés dans la déclaration d'un sous-programme sont appelés **paramètres formels**. Ces paramètres peuvent prendre toutes les valeurs possibles mais ils sont abstraits (n'existent pas réellement)
- Les paramètres placés dans l'appel d'un sous-programme sont **appelés paramètres effectifs**. ils contiennent les valeurs pour effectuer le traitement
- Le nombre de paramètres effectifs doit être égal au nombre de paramètres formels. L'ordre et le type des paramètres doivent correspondre.



# Plan

25

- Partie 1: Les sous-programmes
- Partie 2: Les fonctions
- Partie 3: Les procédures
- Partie 4: Portée d'une variable
- Partie 5: Paramètres formels/effectifs
- Partie 6: Transmission de paramètres
- Partie 7: De l'algorithmique au langage C

# Transmission de paramètres

26

- Il existe deux modes de transmission de paramètres dans les langages de programmation :

**La transmission par valeur** : les valeurs des paramètres effectifs sont affectées aux paramètres formels correspondants au moment de l'appel de la procédure. Dans ce mode le paramètre effectif ne subit aucune modification

**La transmission par adresse (ou par référence)** : les adresses des paramètres effectifs sont transmises à la procédure appelante. Dans ce mode, le paramètre effectif subit les mêmes modifications que le paramètre formel lors de l'exécution de la procédure

**Remarque** : le paramètre effectif doit être une variable (et non une valeur) lorsqu'il s'agit d'une transmission par adresse

# Passage d'arguments dans une fonction

27

## Passage des paramètres par valeur (en Entrée)

- Les paramètres **formels** sont initialisés par une **copie** des valeurs des paramètres **effectifs**
- Modifier la valeur des paramètres formels dans le corps de la fonction ne change pas la valeur des paramètres effectifs

# Passage d'arguments dans une fonction

28

**Algorithme** *exemple 1*

**Var** *a, b* : Entier

**Fonction** *abs* (*unEntier* : Entier) : Entier

**Var** ValeurAbsolue: Entier

**Debut**

**Si**(*unEntier* ≥ 0) alors

ValeurAbsolue ← *unEntier*

**Sinon**

ValeurAbsolue ← - *unEntier*

**Finsi**

Retourner (ValeurAbsolue)

**FinFonction**

**début**

écrire("Entrez un entier:")

lire(*a*)

*B* ← *abs*(*a*)

écrire("la valeur absolue de", *a*, " est ", *b*)

**Fin**

Lors de l'exécution de la fonction *abs*, la variable *a* et le paramètre *unEntier* sont associés par un passage de paramètre en entrée : La valeur de *a* est copiée dans *unEntier*

# Passage d'arguments dans une procédure

29

- Les procédures sont des sous-programmes qui ne retournent **aucun** résultat.
- Par contre elles admettent des paramètres avec des passages:
  - ◆ En entrée, préfixés par **Entrée** (ou **E**)
  - ◆ En sortie préfixés par **Sortie** (ou **S**)
  - ◆ En entrée/sortie, préfixés par **Entrée/Sortie** (ou **E/S**)
- Les paramètres en Entrée sont considérés aussi comme paramètres **passés par valeur**.
- Les paramètre en sortie ou en Entrée/ sortie sont considéré aussi comme paramètres **passés par adresse ou par référence**.

# Déclaration d'une procédure

30

- On déclare une procédure de la façon suivante :

**Procédure** *NomDeLaProcédure* ( **E** *paramètre(s) en entrée*; **S** *paramètre(s) en sortie*; **E/S** *paramètre(s) en entrée/sortie* )

**Var** *variable(s) locale(s)*

**début**

*instructions de la procédure*

**FinProcédure**

- Et on appelle une procédure comme une fonction, en indiquant son nom suivi des paramètres entre parenthèses

# Procédures – Exemple 1

31

**Algorithme** *exemple*

**Var** entier1,entier2,entier3,min,max : **Entier**

**Fonction** minimum2 (a,b : **Entier**) : **Entier**

...

**Fonction** minimum3 (a,b,c : **Entier**) : **Entier**

...

**Procédure** calculerMinMax3 ( **E** a,b,c : **Entier** ; **S** min3,max3 : **Entier** )

**Début**

min3 ← minimum3(a,b,c)

max3 ← maximum3(a,b,c)

**Fin**

**début**

écrire("Entrez trois entiers :")

lire(entier1) ;

lire(entier2) ;

lire(entier3)

calculerMinMax3(entier1,entier2,entier3,min,max)

écrire("la valeur la plus petite est ",min," et la plus grande est ",max)

**Fin**

# Procédures – Exemple 2

32

**Algorithme** *exemple2*

**Var** a,b : entier

**Procédure** *echanger* (**E/S** val1: entier , **E/S** val2: entier )

**Var** temp: entier

**Début**

temp ← val1

val1 ← val2

val2 ← temp

**FinProcédure**

**début**

**écrire**("Entrez deux entiers :")

**lire**(a,b)

**echanger**(a,b)

**écrire**("« a= ",a," et b= ",b)

**Fin**

Lors de l'exécution de la procédure *echanger*, la variable *a* et le paramètre *val1* sont associés par un passage de paramètre en entrée/sortie : Toute modification sur *val1* est effectuée sur *a* (de même pour *b* et *val2*)



# Plan

33

- Partie 1: Les sous-programmes
- Partie 2: Les fonctions
- Partie 3: Les procédures
- Partie 4: Portée d'une variable
- Partie 5: Paramètres formels/effectifs
- Partie 6: Transmission de paramètres
- Partie 7: De l'algorithmique au langage C

# Structure d'un programme en C

34

- ✓ La décomposition d'un programme en sous programmes permet d'avoir des programmes plus lisibles, plus faciles à mettre à jour et plus simples à développer puisqu'on ne s'intéresse qu'à une seule tâche à la fois.
- ✓ Un programme en C est une collection de fonctions. L'une des fonctions doit s'appeler **main**. L'exécution d'un programme C correspond à l'exécution de la fonction **main**. Les autres fonctions sont exécutées si elles sont appelées dans la fonction **main**.
- ✓ **Il n'y a pas de procédure en C**. Pour traduire une procédure, on crée une fonction qui ne retourne pas de valeur, on utilise alors le type **void** comme type de retour.

# Déclaration d'une fonction

35

❖ ***typeRetour* NomFonction (*type1* arg1, *type2* arg2 ...*typeN* argN)**

// paramètres formels *arg1*, *arg2*, ..., *argN*

{

// variables locales

instructions ;

**return** val;

}

❖ . **L'entête de la fonction comprend :**

1. *typeRetour* : le type du résultat retourné, une fonction qui ne retourne rien est de type void.
2. Le nom de la fonction.
3. La liste de ses paramètres : arguments de la fonction. Cette liste peut être vide.

❖ **Le corps :** contient des déclarations de variables, des instructions C (conditions, boucles ...). L'instruction **return** permet de retourner le résultat de la fonction au programme qui l'a appelée

# Déclaration d'une fonction

36

```
/* la fonction surface délivre un int,  
   ses paramètres formels sont l'int largeur et l'int longueur */  
int surface(int largeur, int longueur)  
{  
    int res;          /* déclaration des variables locales */  
    res=largeur*longueur;  
    return res;       /* retourne à l'appelant en délivrant res */  
}
```

# Appel des fonctions

37

Appel d'une fonction : `Variable = Nom_fonction(arg1, arg2, ..., argN);`

Appel d'une procédure : `Nom_procedure(arg1, arg2, ..., argN);`

```
#include <stdio.h>

int surface(int largeur, int longueur)
{
    int res;
    res=largeur*longueur;
    return res;
}

void main()
{
    int l,L,surf;
    printf("\n Entrer la largeur:"); scanf("%d", &l);
    printf("\n Entrer la longueur:"); scanf("%d", &L);
    surf=surface(l,L);    // appel de la fonction surface
    printf("\n Surface = %d\n",surf);
}
```

# Appel des fonctions

38

Si la fonction est écrite **après** la fonction main, il faut donner son prototype (en-tête) avant cette dernière :

```
#include <stdio.h>
int surface(int largeur, int longueur);           // Prototype
void message();

void main()
{
    int l,L,surf;
    message(); scanf("%d", &l);
    message(); scanf("%d", &L);
    surf=surface(l,L);      /*appel avant déclaration de la fonction surface*/
    printf("\n Surface = %d\n",surf);
}

int surface(int largeur, int longueur)
{
    return largeur*longueur;
}
void message() { printf(" \n Saisir une valeur:");}
```

# Appel des fonctions

39

```
#include<stdio.h> /*inclusion d'un fichier permettant l'utilisation*/
                  /* de fonctions déjà codées */

int AuCarre(int); /* prototypage de notre fonction AuCarre */

int main()
{
    /* l'exécution commence ici */
    /* déclaration de variables locales */
    int resultat;
    int valeur = 5;

    resultat = AuCarre(valeur); /*Appel de AuCarre*/
    printf("resultat = %d \n",resultat); /*Appel fonction bibliothèque*/
}

int AuCarre(int parametre) /* code de la fonction AuCarre*/
{
    return parametre*parametre;
}
```

# Que se passe t-il en mémoire?

40

```
1  int AuCarre(int);  
2  
3  int main()  
4  {  
5      int resultat;  
6      int argument = 5;  
7  
8      resultat = AuCarre(argument);  
9  }  
10  
11 int AuCarre(int parametre)  
12 {  
13     return (parametre*parametre);  
14 }
```

## Mémoire

argument

5

resultat

?

État de la mémoire à la ligne 7 de l'exécution

## Mémoire

argument

5

resultat

?

parametre

5

État de la mémoire à la ligne 12

## Mémoire

argument

5

resultat

25

~~parametre~~

~~5~~

État de la mémoire à la ligne 9



# Fonctions — Factorielle d'un entier

41

## Déclaration:

```
int Fact(int n)
{
    int i, f ;
    for(i=1 , f=1 ; i<=n ; i ++ )
        f=f* i ;
    return (f );
}
```

## Appel de cette fonction :

```
void main( )
{
    int x = Fact(4) ;
    int y = Fact(3) + Fact(2) ;
    printf("%d", Fact(5)) ;
}
```

# Fonctions — Maximum de deux entiers

42

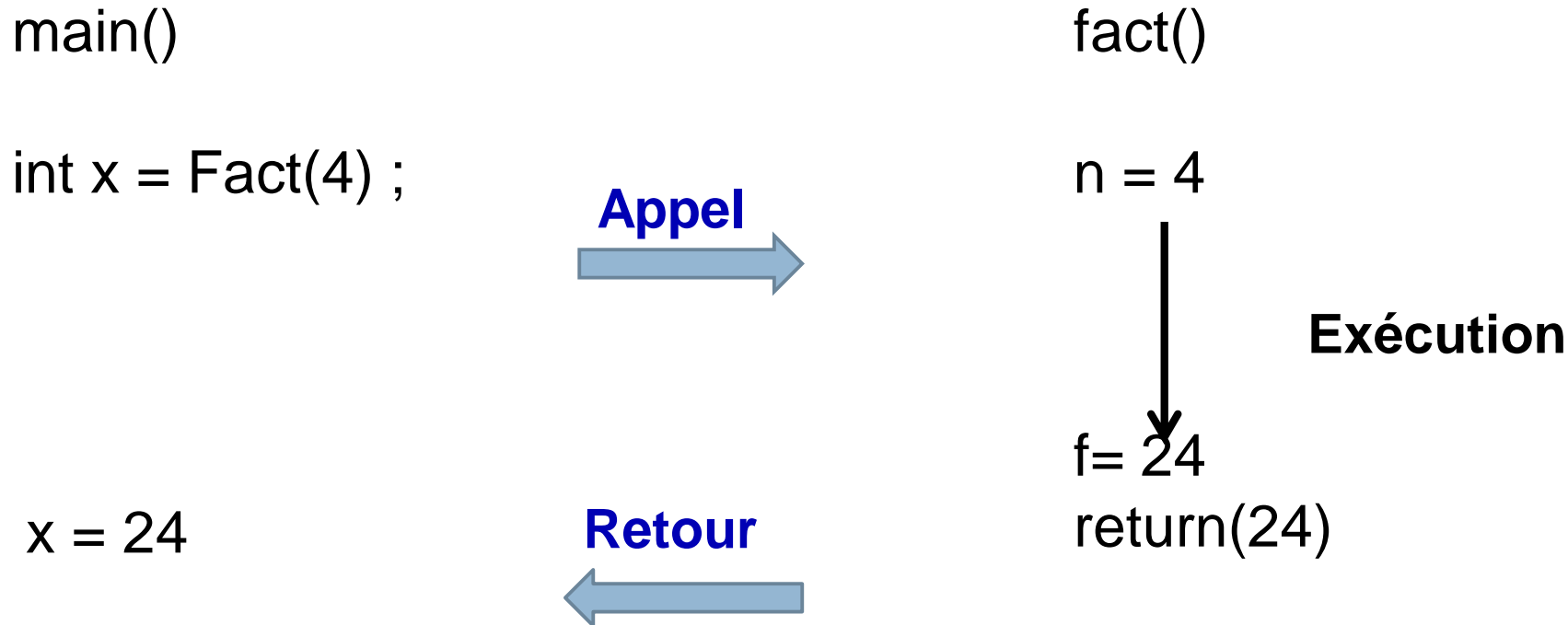
```
int max(int x, int y)
{
    int m ;
    if (x > y)
        m = x ;
    else
        m = y ;
    return (m) ;
}
```

```
int max(int x, int y)
{
    if (x > y)
        return (x) ;
    else
        return (y) ;
}
//sans aucune variable intermédiaire
```

```
void main( )
{
    int a ;
    a = Max(4, 5) ;
    printf("%d", a);           // a = 5
    a = Max(a, 10) ;
    printf("%d", a);          // a = 10
    a = Max(a, Max(a, 5)) ;
    printf("%d", a);          // a = 10
}
```

# Fonctions — Passage de paramètres par valeur

43



Lors de l'appel, il y a copie des valeurs des paramètres effectifs dans les paramètres formels. ➔ **Passage des paramètres par valeur.**

`fact(4)` ➔ Paramètre **effectif** (appel de la fonction)

`int fact(int n)` ➔ Paramètre **formel** (Déclaration de la fonction)

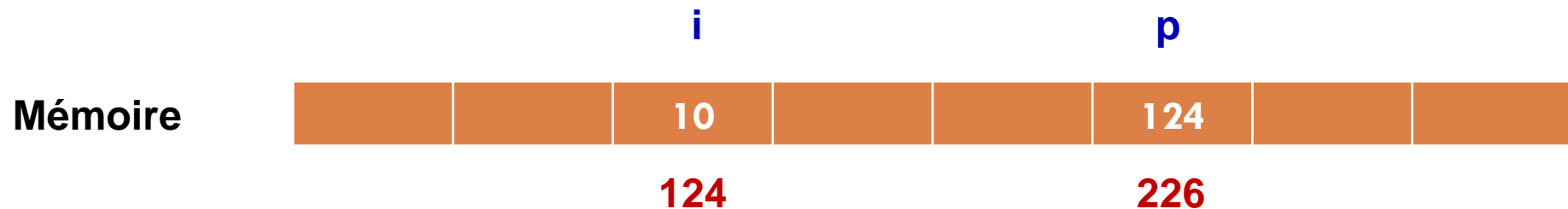
# Passage de paramètres par adresse: Les pointeurs

44

## Définition :

- ✓ Un pointeur est une variable qui permet de stocker une **adresse**
- ✓ Lorsque , l'on déclare une variable, par exemple un entier i, l'ordinateur réserve un espace mémoire pour y stocker la valeur de i
- ✓ L'emplacement de cet espace dans la mémoire est nommé adresse

Par exemple si on déclare une variable de type entier (initialisée a 10) et que l'on déclare un pointeur p dans lequel on range l'adresse de i (on dit que p pointe sur i), on a le schéma suivant :



# Les pointeurs

45

- ▶ On déclare une variable de type pointeur en préfixant le nom de la variable par le caractère `*`.

**Type \*p;**



p: p est un pointeur sur une variable de type Type. C'est une adresse  
\*p: permet d'accéder à la valeur de la variable pointée.

## Exemple 1:

On veut déclarer un pointeur p qui va contenir l'adresse d'un entier

`int *p ;` // cette déclaration signifie que p va contenir l'adresse d'un entier

`int A = 20 ;`

**p = &A ;** p contient l'adresse mémoire de la variable A.

**\*p** : le contenu de la case mémoire pointée par p (contenu de la variable A).

donc **\*p est égal à A**

# Les pointeurs

46

## Exemple 2 :

```
int *p ;  
int x = 5 ;  
int y ;  
p = &x ;  
y = *p + 1 ;           // ➔  $y = x + 1 \Leftrightarrow y = 6$   
*p = *p + 10 ;         // ➔  $x = x + 10 \Leftrightarrow x = 15$   
*p += 2                // ➔  $x = x + 2 \Leftrightarrow x = 17$   
(*p)++ ;              // ➔  $x++ \Leftrightarrow x = 18$ 
```

Si on a :

```
int *p ;  
int A = &p ;  
Alors   p contient l'adresse mémoire de l'entier A ;  
        *p est un entier ➔ *p est égal à A
```

# Les pointeurs

47

## Exemple 3 :

La taille d'un entier en octets: 4

```
int i=3;  
int *p;  
p = &i;
```

Variable	Adresse	Valeur
i	4830000	3
p	4830004	4830000
*p	4830000	3

# Les pointeurs

48

## Exercice :

Considérons les variables suivantes :

```
int A = 1 ;
```

```
int B = 2 ;
```

```
int C = 3 ;
```

```
int *p1 ;
```

```
int *p2 ;
```

Le tableau suivant contient l'effet de chaque commande sur les variables.



# Les pointeurs

49

	A	B	C	p1	p2
p1 = &A ;	1	2	3	&A	-
p2 = &C ;	1	2	3	&A	&C
*p1 = (*p2)++ ;	3	2	4	&A	&C
p1 = p2 ;	3	2	4	&C	&C
p2 = &B ;	3	2	4	&C	&B

# Arithmétique des pointeurs

50

- ❖ La valeur d'un pointeur est un entier.
- ❖ On peut appliquer à un pointeur quelques opérations arithmétiques :
  - Addition d'un entier a un pointeur.
  - Soustraction d'un entier a pointeur.
  - Différence entre deux pointeurs (de même type).
- ❖ Soit  $i$  un entier et  $p$  un pointeur sur un élément de type `Type` ,  
L'expression  $p' = p + i$  (resp.  $p = p - i$ ) désigne un pointeur sur un  
élément de type `Type` ,  
la valeur de  $p'$  est égale a la valeur de  $p$  incrémenté (resp.  
décrémenté) de  $i * \text{sizeof}(\text{Type})$ .

# Arithmétique des pointeurs

51

```
main()
{
  int i=5;
  int * p1;
  int * p2;
  p1=&i+2;
  p2=p1-2;
}
```

Si  $\&i = 4830000$  alors :

Variable	Adresse	Valeur
i	4830000	5
p1	4830004	4830008
p2	4830008	4830000

# Initialisation des pointeurs

52

- ❑ Par défaut, lorsque l'on declare un pointeur, on ne sait pas sur quoi il pointe.
- ❑ Comme toute variable, il faut l'initialiser. On peut dire qu'un pointeur ne pointe sur rien en lui affectant la valeur NULL.

## Exemple :

```
Int i;  
Int *p1, *p2;  
p1 = &i;  
p2 = NULL;
```

# Initialisation des pointeurs

53

❑ Deux initialisations possibles pour les pointeurs:

❑ l'affectation de l'adresse d'une autre variable.

Par exemple

```
int x;  
int * p;  
p=&x;
```

❑ l'allocation dynamique d'un espace mémoire.

# Initialisation des pointeurs :

## Allocation dynamique

54

- ❑ L'allocation dynamique est l'opération qui consiste à réserver un espace mémoire d'une taille définie.
- ❑ L'allocation dynamique en C se fait à travers de la fonction de la librairie standard `stdlib` :

```
void *malloc(size_t size);
```

- ❑ Cette fonction retourne un pointeur de type `void *` pointant vers espace mémoire de taille `size`.

Il faut utiliser l'opérateur de cast afin de modifier et d'adapter le type du pointeur selon le besoin

Exemple :

```
int *t;
```

```
t = (int*) malloc (10* sizeof(int));
```

# Les pointeurs: Libération de l'espace mémoire

55

- ❑ C'est l'opération qui consiste à la libération de l'espace mémoire alloué
- ❑ En C, la libération de l'espace mémoire se fait par l'intermédiaire de la fonction de la librairie standard `stdlib.h`

**`void free(nom_pointeur) ;`**

- ❑ Tout espace mémoire alloué par la fonction `malloc` (ou équivalent) doit être libéré par la fonction `free`.
- ❑ Exemple:

```
int x;  
x=(int*)malloc (sizeof(int));  
.....  
free(x);
```

# Passage de paramètre en entrée

56

## ... par un passage de paramètre par valeur

On copie la valeur du paramètre effectif dans le paramètre formel :

```
int carre (int x){  
    return (x*x);  
}  
  
void exemple (){  
    int i = 3;  
    int j;  
  
    j = carre (i);  
    printf ("%d\n",j);  
}
```

	exemple	carre
Avant appel de carre	i = 3 j = ?	
Appel de carre	i = <span style="border: 1px solid black; padding: 0 2px;">3</span> j = ?	x = <span style="border: 1px solid black; padding: 0 2px;">3</span>
Après appel de carre	i = 3 j = 9	<span style="border: 1px solid black; padding: 0 2px;">9</span>

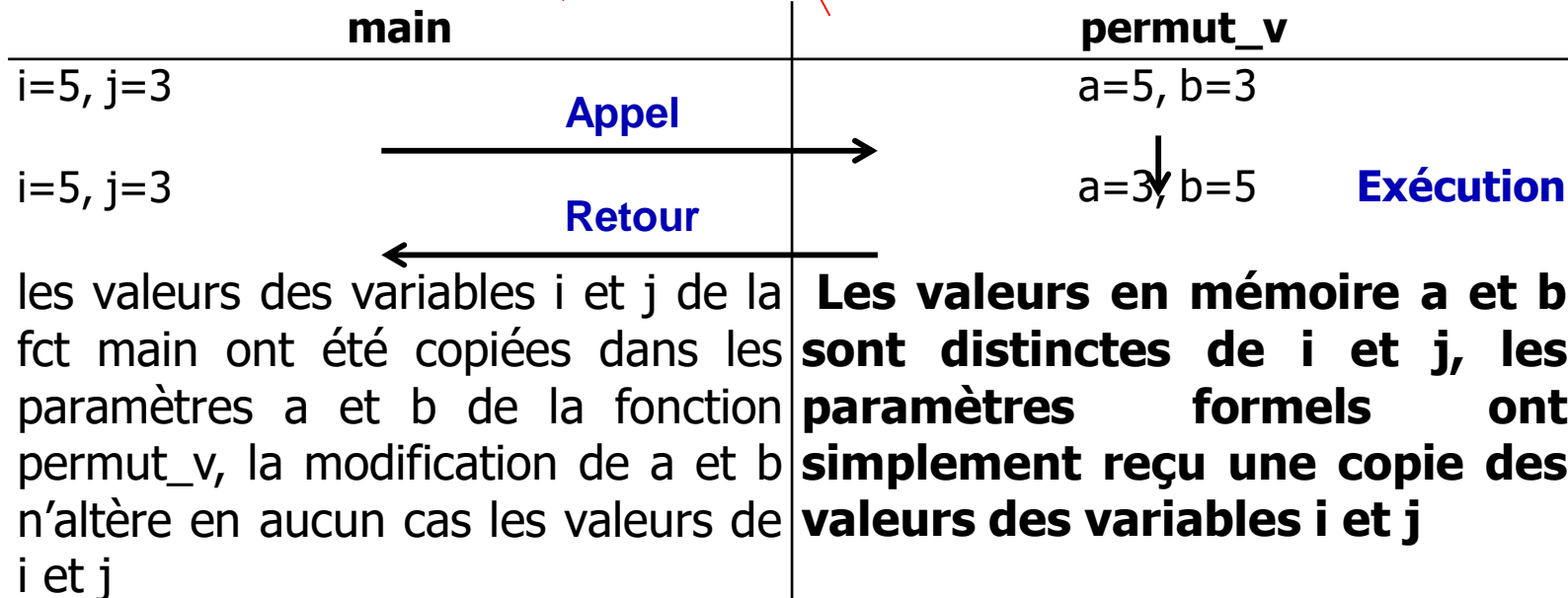


# Passage de paramètres en entrée/sortie ou sortie

57

```
void permut_v(int a, int b)
{
    int x=a ;
    a=b ;
    b=x ;
}
```

```
void main()
{
    int i=5,j=3 ;
    permut_v(i,j) ;
    printf("i=%d, j=%d", i,j) ;
}
```

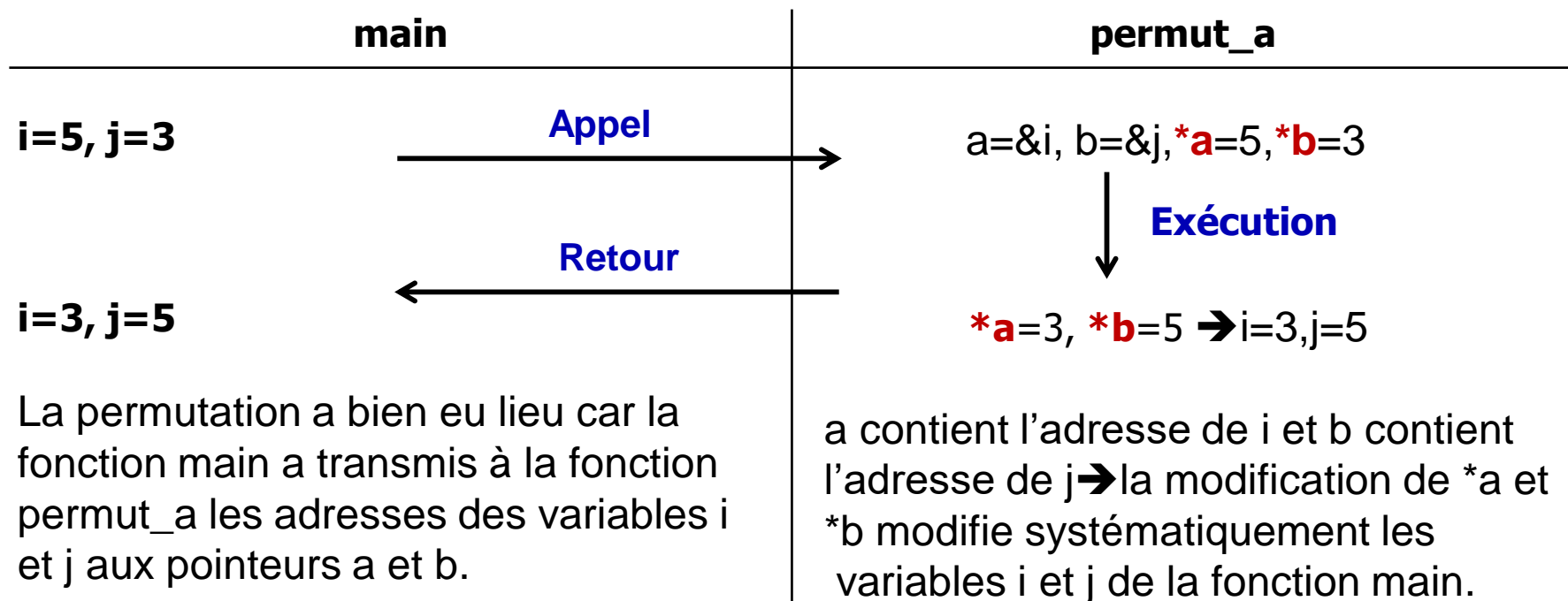


# Passage de paramètres en entrée/sortie ou sortie

58

```
void permut_a(int *a, int *b)
{
    int x = *a ;
    *a = *b ;
    *b = x ;
}
```

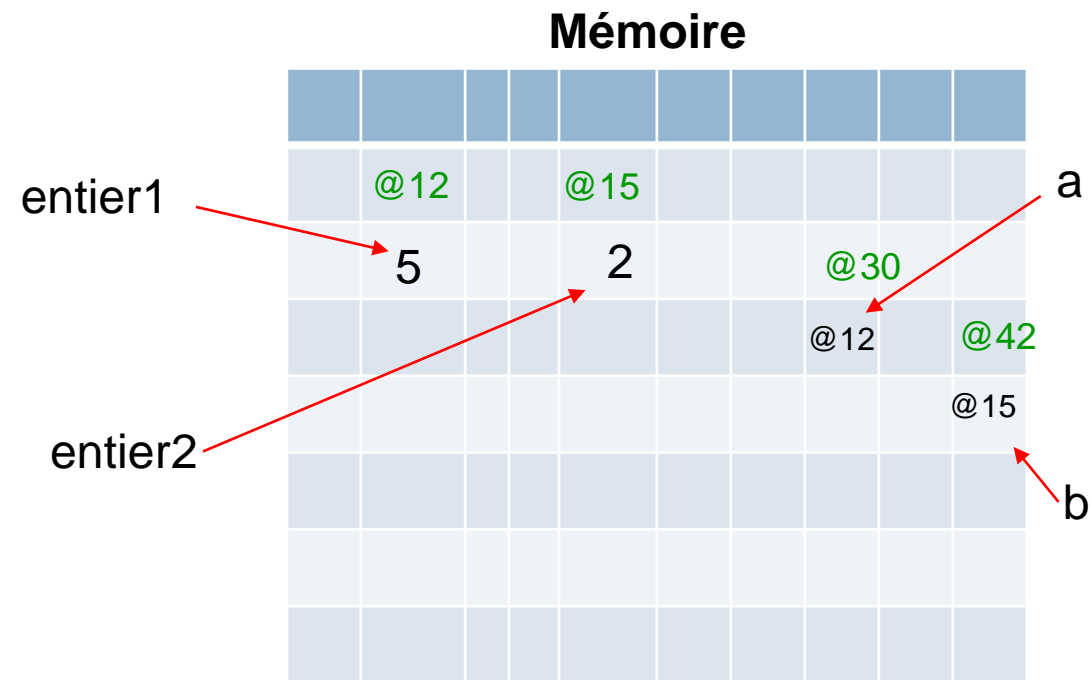
```
void main()
{
    int i=5,j=3 ;
    permut_a(&i,&j) ;
    printf("i=%d, j=%d", i,j) ;
}
```



# Exemple illustratif

59

```
#include<stdio.h>
void permutation( int * a, int* b)
{
    a= @12    *a=5
    b= @15    *b=2
    int aux;
    aux = *a;
    *a = *b;
    *b = aux;
}
int produit( int x, int y){
    return(x*y);      x=2 y=5
}
int main( )
{
    /*1*/int entier1, entier 2;
    /*2*/ printf("Saisir les deux entiers\n");
    /*3*/ scanf("%d%d",&entier1,& entier2);
    /*4*/ permutation(& entier1,& entier2); //permutation(@12,@15)
    /*5*/ printf("Après permutation : a =%d b =%d",a,b);
    /*6*/ printf("%d*%d=%d\n",a,b,produit(a,b)) ; produit(2,5)
}
```



Instruction

Affichage

1

2

3

4

5

6

Saisir les deux entiers

5 2

Après permutation a=2 b=5

2\*5= 10

# Passage de paramètres en entrée/sortie ou sortie

60

```
#include <stdio.h>

void surface(int largeur, int longueur, int *res)
{
    *res=largeur*longueur;
}

void main()
{
    int l,L,surf;
    printf("\n Entrer la largeur:"); scanf("%d", &l);
    printf("\n Entrer la longueur:"); scanf("%d", &L);
    surface(l,L, &surf);    // appel de la fonction surface
    printf("\n Surface = %d\n",surf);
}
```