

STRUCTURES DE DONNÉES

MIP — S4

Pr. K. Abbad & Pr. Ad. Ben Abbou & Pr. A. Zahi
Département Informatique
FSTF
2019-2020

Objectifs du cours

- ⊙ Acquérir une méthodologie de programmation — **diviser pour régner**
- ⊙ Sensibiliser les étudiants à l'importance de la représentation des données dans un programme
- ⊙ Savoir manipuler les fichiers en C
- ⊙ Maîtriser les structures de données élémentaires — **tableaux, listes chaînées, piles, files et arbres.**

Plan du cours

- ⊙ Rappels
- ⊙ Programmation Structurée
- ⊙ Allocation dynamique
- ⊙ Fichiers
- ⊙ Piles
- ⊙ Files
- ⊙ Listes chaînées
- ⊙ Arbres

Séance 1

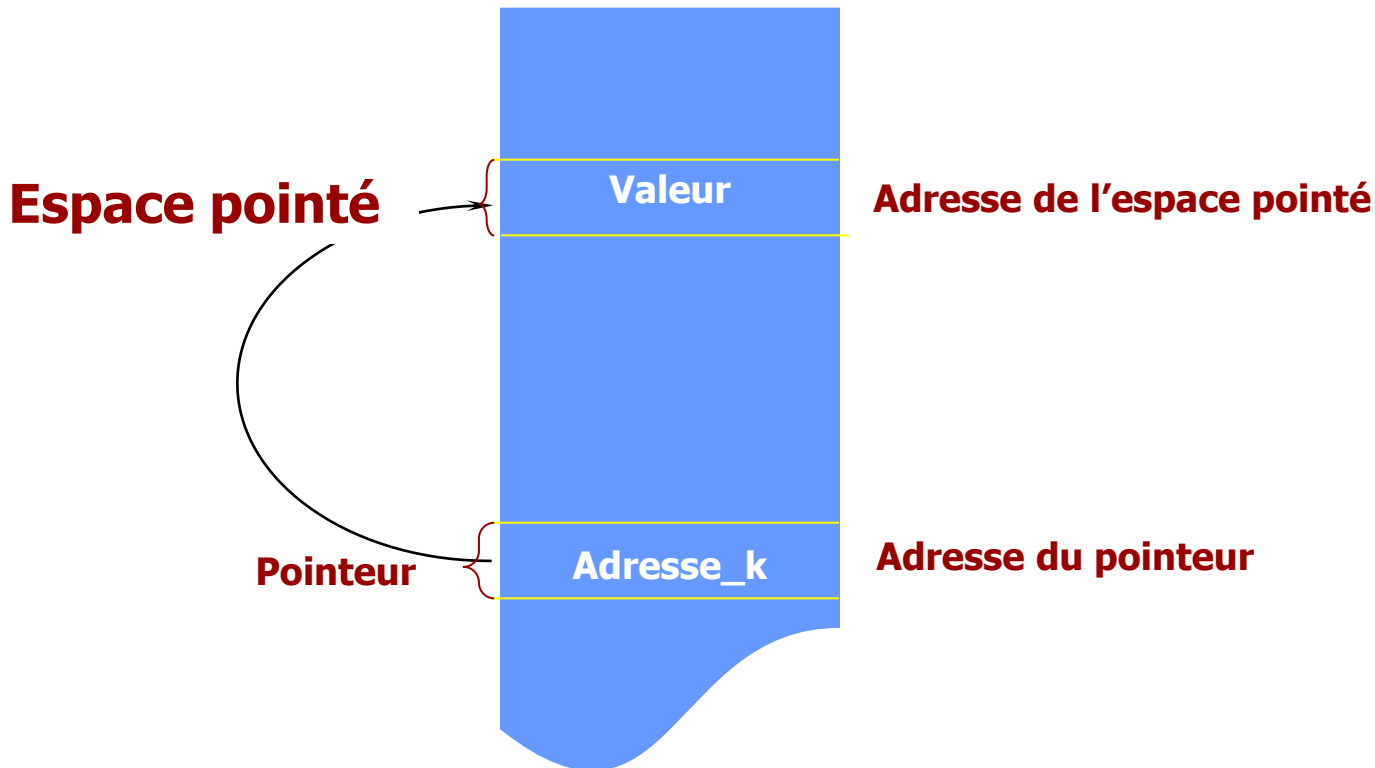
- ⊙ Rappels
- ⊙ Programmation Structurée

Rappels

- ⊙ Pointeurs
- ⊙ Fonctions
- ⊙ Tableaux
- ⊙ Structures

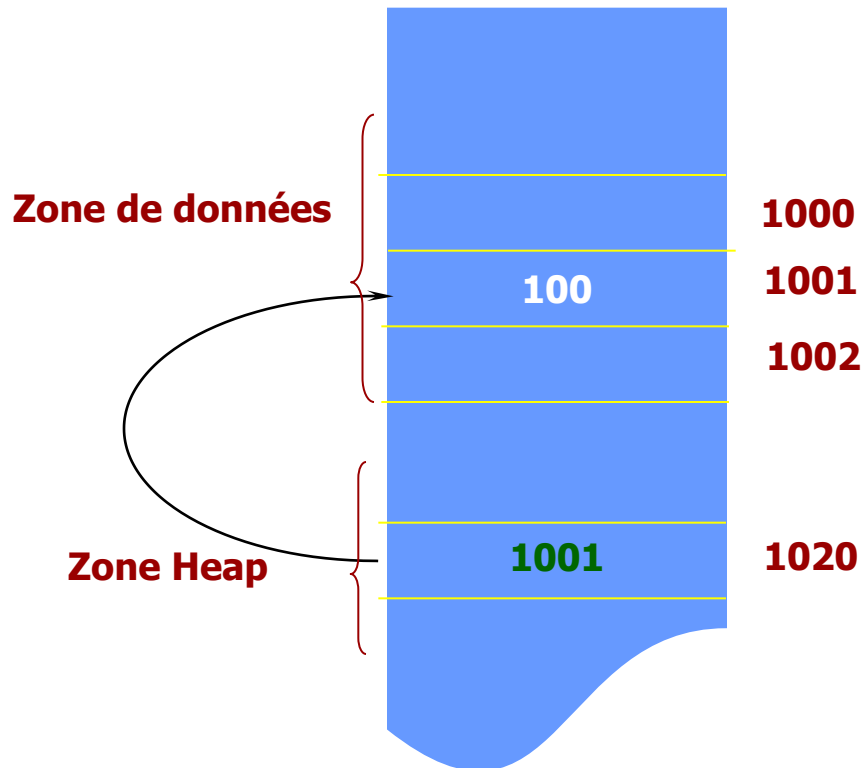
Pointeur — *Définition*

- Un **pointeur** est une **variable** ou une **constante** dont la valeur est une **adresse** d'un emplacement mémoire



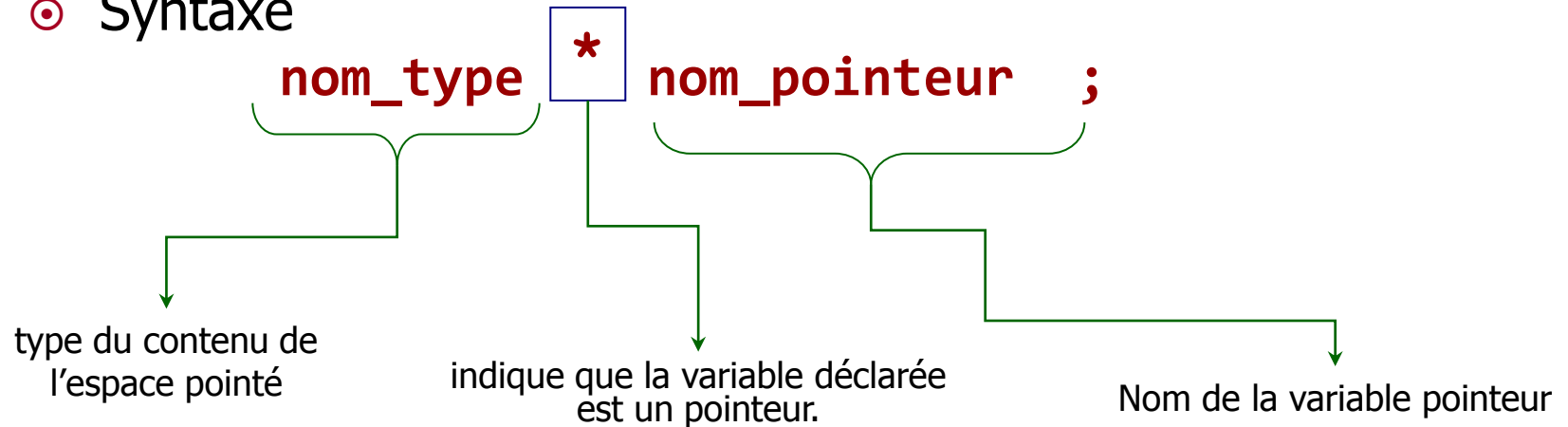
Pointeur — *Exemple*

- Le **pointeur** qui se trouve à l'adresse **1020** pointe sur l'espace d'adresse **1001** qui contient la valeur **100**



Pointeur — *Déclaration*

⊙ Syntaxe



⊙ Exemples

```
int  *ptr_entier ; /* pointeur sur un entier*/  
long int *ptr_long; /* pointeur sur un entier long*/  
double *ptr_double; /* pointeur sur un réel double*/  
char *ptr_carac  ; /* pointeur sur un caractère*/  
Complexe *Z ; /* pointeur sur un complexe qui est  
défini comme une structure*/
```


Pointeur — *Initialisation*

- ⊙ L'initialisation consiste à **affecter** une adresse à la variable pointeur
- ⊙ Un pointeur peut être initialisé par :
 - ▶ L'adresse d'une variable existante à l'aide de l'opérateur **&**
 - ▶ Une nouvelle adresse, à l'aide de la fonction **malloc**

Pointeur — *Initialisation à l'aide de l'opérateur &*

- ⊙ L'opérateur **&** permet d'obtenir l'adresse d'une variable

si **var** est une variable d'un certain type **alors** **&var** est son adresse

- ⊙ Syntaxe d'initialisation d'un pointeur avec l'adresse d'une autre variable :

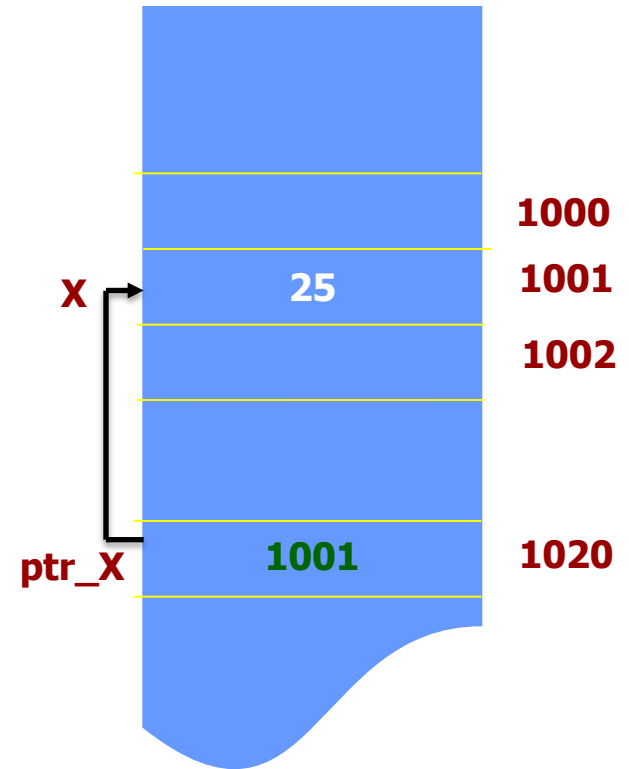
```
type X ;  
type* ptr_X ;  
ptr_X = &X ; /* affectation de l'adresse de X à ptr_X */
```

Pointeur — *Initialisation à l'aide de l'opérateur &*

◉ Exemple

```
void main()
{
    /* X contient la valeur 25    */
    int    X = 25;
    int *ptr_X ;

    ptr_X = &X ;
    /* ici ptr_X contient l'adresse de X */
}
```



Pointeur — *Initialisation à l'aide de malloc*

- ⊙ Affecte à un pointeur l'adresse d'une case mémoire **réservée au cours de l'exécution** du programme.
- ⊙ L'adresse de l'espace pointé est obtenue en utilisant:
 - ▶ La fonction **malloc()** qui réserve un espace mémoire d'une taille donnée et de retourner son **adresse**.
 - ▶ La taille de l'espace réservé est déterminée par la fonction **sizeof**.
- ⊙ Syntaxe

```
nom_type *ptr_X ;  
int t= sizeof(nom_type) ;  
ptr_X= (nom_type* ) malloc(t) ;
```

Pointeur — *Initialisation à l'aide de malloc*

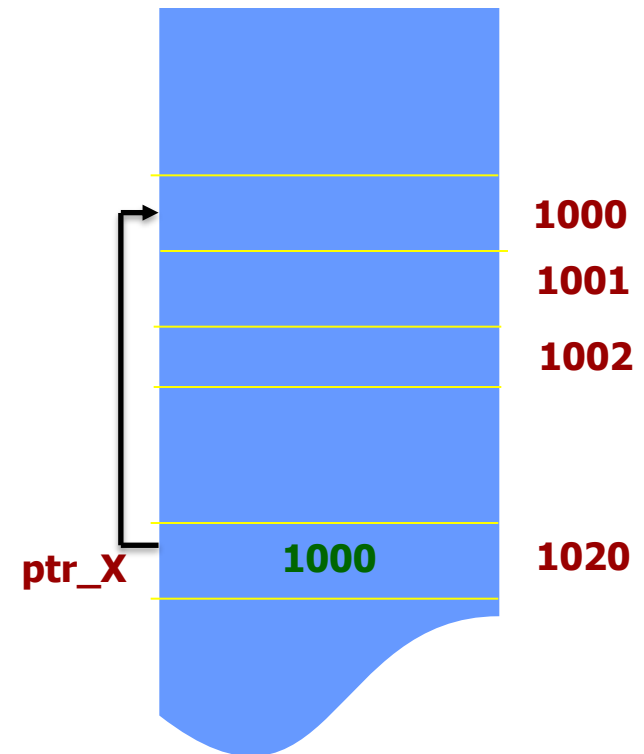
⊙ Exemple

```
void main()
{
    int *ptr_X ;

    /* allocation mémoire pour une
       variable entière X */
    ptr_X = (int *) malloc
            (sizeof(int)) ;

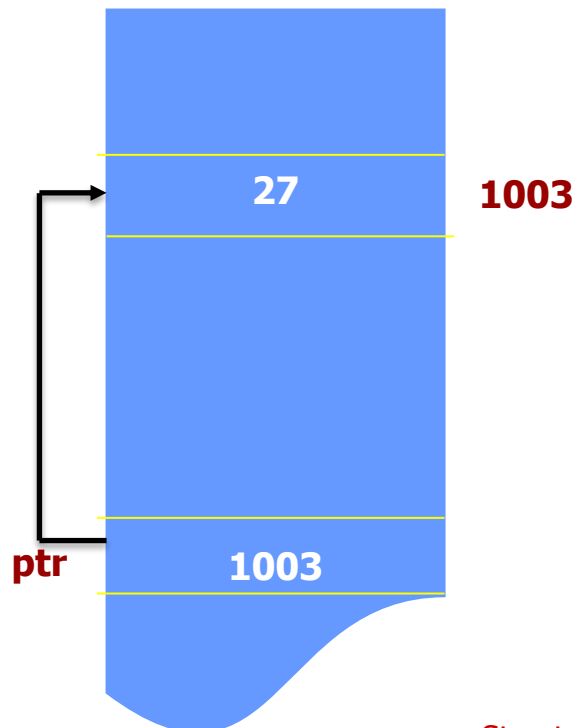
    /* on suppose que la fonction
       malloc a retourné l'adresse
       1000*/

}
```



Pointeur — *Accès à l'espace pointé*

- ⊙ L'opérateur ***** permet de délivrer la valeur contenue dans l'espace pointé par un pointeur.
- ⊙ Si **ptr** est un pointeur de type **t** alors ***ptr** est une variable de type **t** qui contient la valeur contenue dans l'adresse **ptr**



***ptr** contient la valeur 27

Pointeur — *Accès à l'espace pointé*

- ◉ Sur la variable ***ptr**, on peut effectuer toutes les opérations permises sur le type **t**:

- ▶ **Lecture** : `scanf(''%car'', ptr);`
- ▶ **Affichage** : `printf(''%car'', *p);`
- ▶ **Opérateurs** : `=, +, *, /, ==`, Etc.

Avec **car** un spécificateur de type (**d, f, s, c, etc.**)

Pointeur — *Remarques*

- ⊙ On **ne peut pas affecter** un pointeur de type **T1** à un pointeur de type **T2**

```
int      i = 3, *ptr_int;  
char     *Ptr_C;  
ptr_int=&i;  
Ptr_C = ptr_int;      /* ERREUR */
```

- ⊙ On **peut forcer** la conversion

```
int      i = 3, ptr_int;  
char     *Ptr_C;  
ptr_int=&i;  
Ptr_C = (char *) ptr_int;
```


Pointeur — *Remarques*

- ⊙ Comment obtenir une valeur à partir d'un pointeur ?

```
int *px; px =(int*) malloc(sizeof(int));  
int x = *px ;
```

- ⊙ Comment obtenir l'adresse d'une variable ?

```
int x=15;  
int* p = &x ;
```

- ⊙ L'initialisation d'un pointeur lors de la déclaration permet d'initialiser la **valeur** du pointeur et **non** pas le **contenu** de l'espace pointé.

```
int x=15;  
int *p = &x ; /* Juste*/  
int *p = 10 ; /* Faux*/
```

Pointeur — *Libération d'un espace*

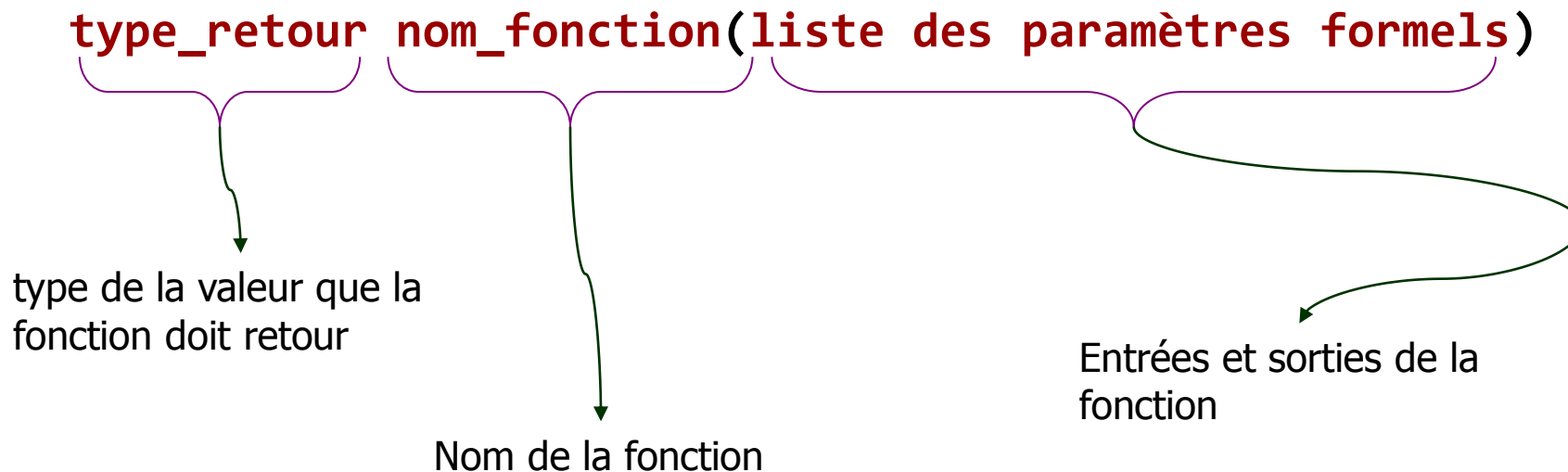
- ⊙ La fonction **free** permet de libérer un espace mémoire qui a été alloué par la fonction **malloc**,
- ⊙ Cette fonction est appelée une fois qu'on a plus besoin de cet espace.
- ⊙ Syntaxe
 - ▶ P un pointeur qui pointe vers l'espace à libérer

free(p);

Fonctions — *Définition*

- ⦿ La définition d'une fonction est constituée par trois éléments, chacun joue un rôle particulier.

► Syntaxe



Fonctions — *Définition*

◉ Liste des paramètres — Paramètres *Fixes*

► **Invariants** par la fonction — la valeur en **entrée** est la **même** qu'en **sortie**, même s'elle est modifiée à l'intérieur de la fonction.

► Syntaxe

```
type nom_fonction(type1 id,..., ..)
```

► Exemple

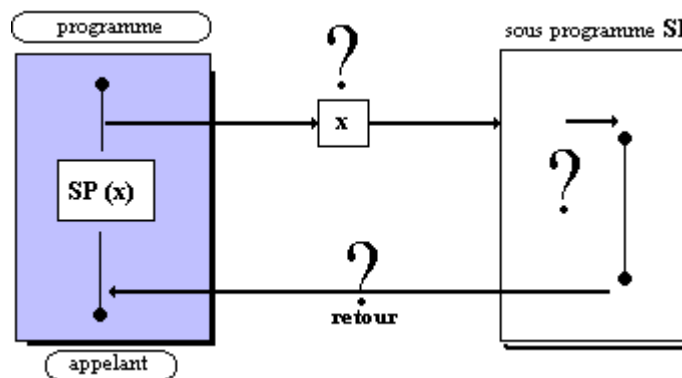
```
int somme (int a, int b)
```

Fonctions — *Définition*

- ⊙ Liste des paramètres — Paramètres *variables*
 - ▶ **Modifiables** par la fonction — la valeur en **entrée n'est pas** toujours la même que celle en sortie.
 - ▶ Syntaxe
`type nom_fonction(type1 *id,...,...)`
 - ▶ Exemple
`void permuter(int *a, int *b)`

Fonctions — *Utilisation (Appel)*

- ◉ Appliquer une fonction sur des valeurs (arguments)
- ◉ La fonction **appelante** passe ses arguments à la fonction **appelée**



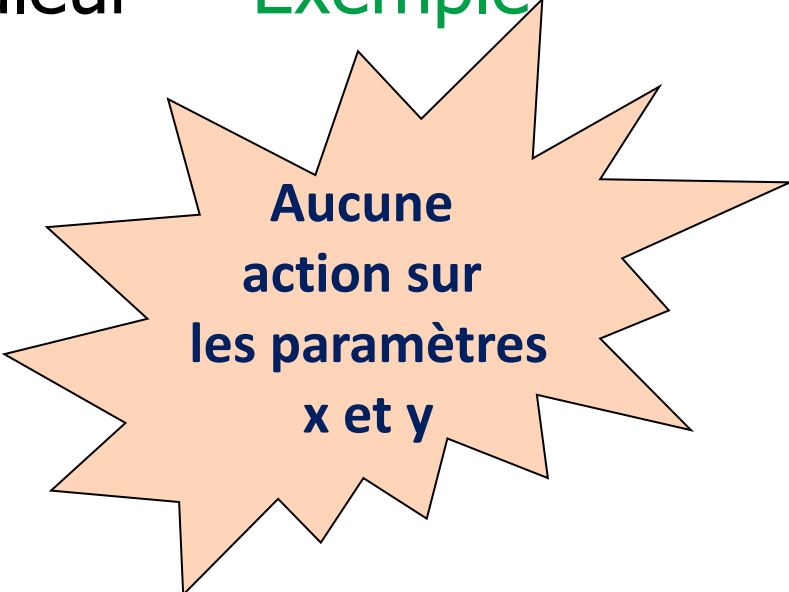
Fonctions — *Utilisation*

⊙ Passage d'arguments par valeur — Exemple

```
void permuter(int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

```
void main(){
    int x=3; int y =5;

    printf("x = %d et y = %d\n",x, y); //affiche x=3 et y=5
    permuter(x, y);
    printf("x = %d et y = %d\n",x, y); //affiche x=3 et y=5
}
```



**Aucune
action sur
les paramètres
x et y**

Fonctions — *Utilisation*

⊙ Passage d'arguments par valeur — Exemple :

```
4.1 void permuter(int a, int b){  
    4.1 int temp;  
    4.1 temp = a;  
    4.1 a = b;  
    4.1 b = temp;  
}
```

a=x et b=y

```
void main(){  
  ① int x,y;  
  ② x=3;  
  ③ y=5;  
  ④ permuter(x, y);  
}
```

Avant permuter

x=3
y=5

après permuter

x=3
y=5


x	3	1000
		1001
y	5	1003
		1004
		1005
	⋮	
a	5	1030
		1031
b	3	1032
		1033
temp	3	1034
		1035
	⋮	

Fonctions — *Utilisation*

⊙ Passage d'arguments par adresse — Exemple

```
void permuter(int *a, int *b){  
    int temp;  
    temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

```
void main(){  
    int x=3; int y=5;  
    printf("x = %d et y = %d\n",x, y); //affiche x=5 et y=3  
    permuter(&x, &y);  
    printf("x = %d et y = %d\n",x, y); //affiche x=5 et y=3  
}
```



Action sur
les paramètres
x et y

Fonctions — *Utilisation*

⊙ Passage d'arguments par adresse — Exemple

```
4.1 void permuter(int *a, int *b){  
    4.1 int temp;  
    4.1 temp = *a;  
    4.1 *a = *b;  
    4.1 *b = temp;  
}
```

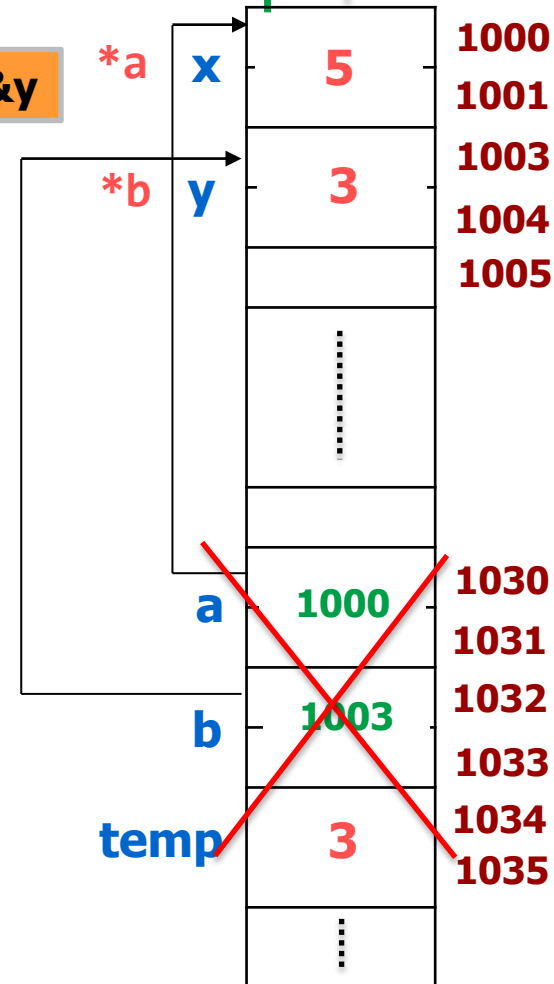
```
void main(){  
  ① int x,y;  
  ② x=3;  
  ③ y=5;  
  ④ permuter(&x, &y);  
}
```

Avant permuter

x=3
y=5

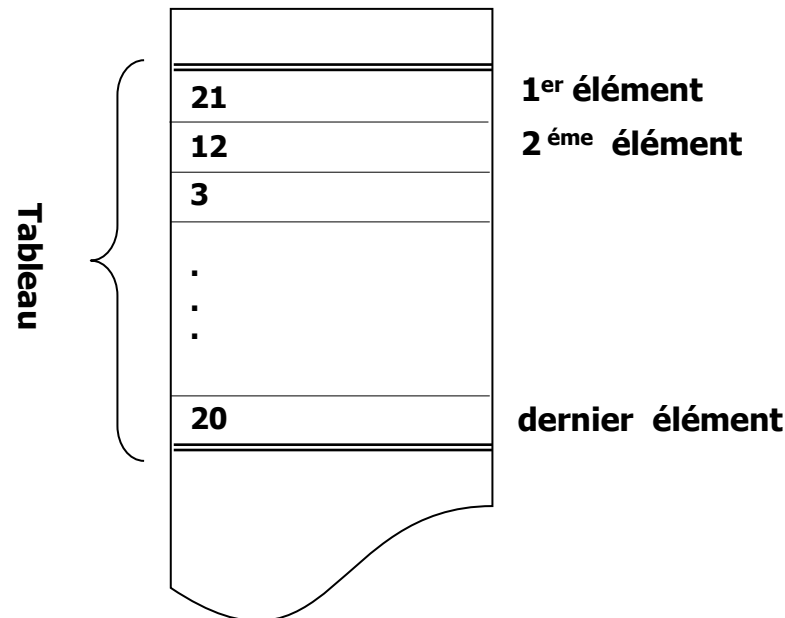
Après permuter

x=5
y=3



Tableaux — *Définition*

- ⊙ Un tableau est un ensemble **fini** de variables de **même type**, situées dans un espace **contigu** en mémoire.
- ⊙ Un espace **contigu** : Les éléments sont stockés les un à la suite des autres.



Tableaux — *Déclaration*

⊙ Syntaxe

type_element nom_tableau[CAP];

- ▶ **nom_tableau:** nom de la variable tableau
- ▶ **type_element:** type des éléments du tableau,
- ▶ **CAP:** constante qui indique le nombre maximal d'éléments qu'on peut stocker dans le tableau.

Tableaux — *Déclaration*

⊙ Remarques

- ▶ La **capacité** d'un tableau doit être connue au moment de la déclaration, on ne peut donc pas faire les déclarations suivantes:

```
int t[ ];  
int t[n]; avec n une variable.
```

- ▶ La **taille** d'un tableau est le nombre d'éléments effectivement utilisés pendant une exécution, la déclaration devient :

```
#define CAP 10    /* Capacité du tableau */  
int N ;          /* Taille du tableau   */  
int t[CAP];
```

- ▶ Les tableaux en C sont indexés de 0 à CAP-1.

Tableaux — *Initialisation*

- ⊙ Initialisation de l'intégralité ou une partie d'un tableau, par une **liste de constantes** au moment de la déclaration.
- ⊙ L'initialisation permet d'affecter aux éléments, respectivement et à partir de **0**, les constantes entre accolades.
- ⊙ Dans le cas où la taille (capacité) n'est pas indiquée, le compilateur crée un tableau avec autant d'éléments que de valeurs entre accolades.

- ⊙ Exemples

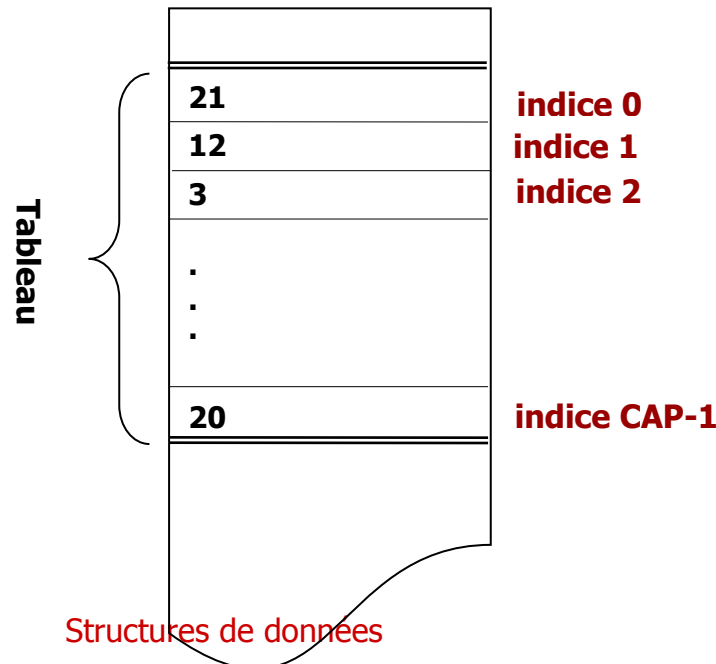
```
#define CAP 5
/* initialisation de l'intégralité du tableau*/
    int t[CAP] = {1,2,3,4,8};

/* initialisation d'une partie du tableau*/
    int k[CAP] = {1,2};

/* initialisation sans indiquer la taille */
    int l[] = {4,7,9,76};
```

Tableaux — *Accès à un élément*

- ⊙ Chaque élément du tableau est identifié par son **indice**
- ⊙ L'opérateur d'indexation **[.]** permet de renvoyer l'élément du tableau d'un indice donné
- ⊙ Syntaxe — **nom_tableau [exp]**
 - ▶ **exp**: une expression entière qui indique l'indice de l'élément dans le tableau.

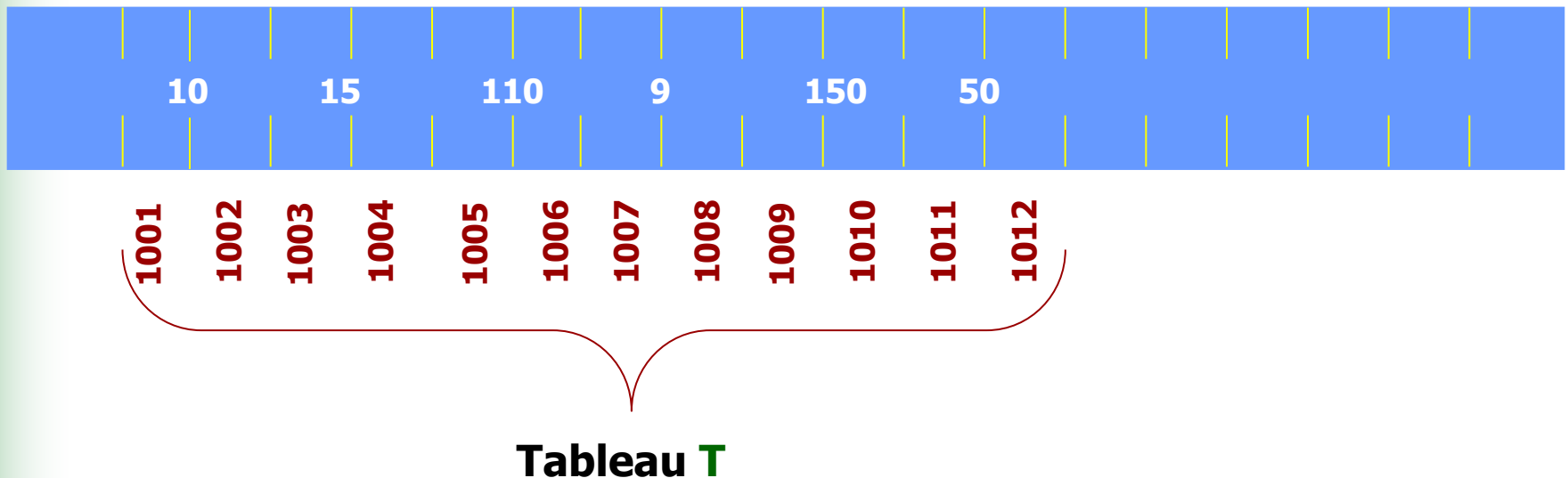


Tableaux — *Accès à un élément*

- ⊙ Soit **T** un tableau, le **(i+1)**^{ème} élément **T[i]** peut être considéré comme une variable simple sur laquelle on peut effectuer les opérations suivantes :
 - ▶ **Saisie** : `scanf("%car", &T[i]) ;` **car** l'un des spécificateurs de format
 - ▶ **Affichage** : `printf("%car", T[i]);` **car** l'un des spécificateurs de format
 - ▶ **Affectation** :
 - `/*valeur d'une constante*/`
`T[i]= Constante;`
 - `/*valeur d'une expression*/`
`T[i]= expression;`
 - `/*valeur d'une autre variable*/`
`T[i]= X ;`
 - ▶ **Élément d'une expression** :
`S=2*PI*T[i] ;`
`if(T[i] >=0)`
`T[i]= sqrt(T[i]) ;`

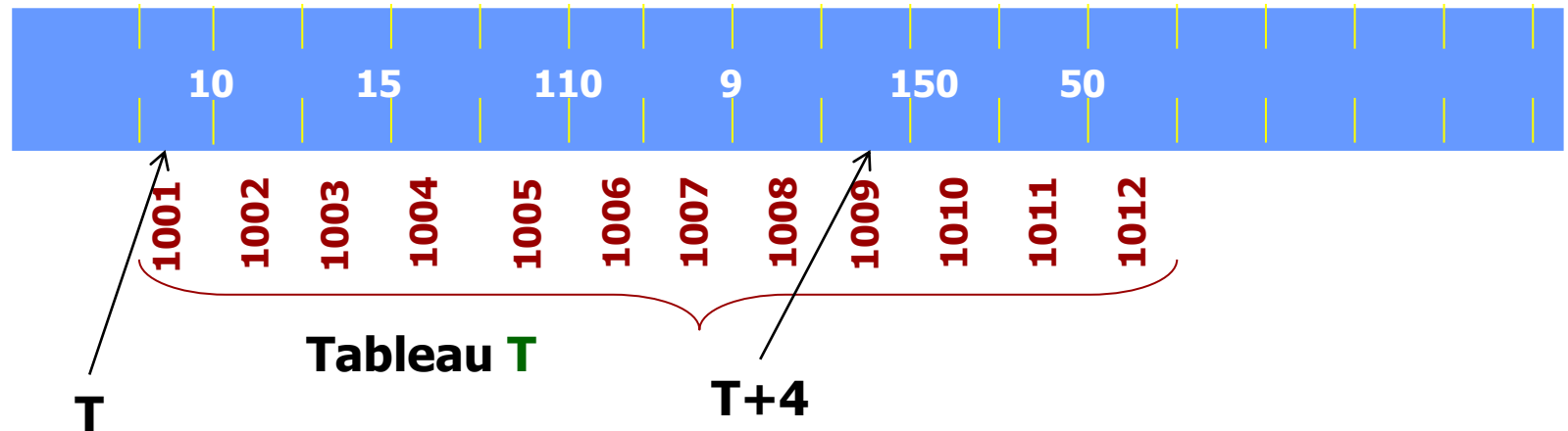
Tableaux — *le nom d'un tableau*

- ⊙ Le nom d'un tableau est un pointeur qui pointe vers le premier élément du tableau
- ⊙ Dans l'exemple ci-dessous
 - ▶ T contient l'adresse du 1^{er} élément i.e., 1001
 - ▶ *T contient la valeur 10



Tableaux — *le nom d'un tableau*

- ⊙ Soit la déclaration: **t T[CAP];**
 - ▶ ***(T+i)** contient la valeur du $(i+1)^{\text{ème}}$ élément, avec i varie de **0** à **CAP-1**
 - ▶ **T+i** contient l'adresse de $(i+1)^{\text{ème}}$ élément du tableau T.
- ⊙ Exemple



Tableaux — *Paramètres d'une fonction*

- ⊙ Pour passer un tableau comme paramètre à une fonction, on doit spécifier les informations suivantes :
 - ▶ **l'adresse du premier élément** — nom du tableau
 - ▶ la **taille effective** du tableau — la dimension du tableau

Tableaux — *Paramètres d'une fonction*

⊙ Déclaration de la fonction

- ▶ Si on ne veut pas modifier la taille du tableau

```
type nom_fonc1 ( type tab[CAP], int taille) ;  
type nom_fonc2 ( type tab[ ], int taille) ;  
type nom_fonc3 ( type *tab, int taille) ;
```

- ▶ Si on veut modifier la taille du tableau

```
type nom_fonc11 ( type tab[CAP], int *taille) ;  
type nom_fonc22 ( type tab[ ], int *taille) ;  
type nom_fonc33 ( type *tab, int *taille) ;
```

⊙ Appel de la fonction

```
type tab[10] ;  
int n ; /* taille du tableau */  
nom_fonc1(tab,n) ;  
nom_fonc11(tab,&n) ;
```

Tableaux — *Exemple*

- ⊙ On se propose d'écrire un programme qui permet de:
 - ▶ Saisir un tableau de réels
 - ▶ Calculer la moyenne des éléments du tableau
 - ▶ Remplacer chaque élément du tableau par son écart par rapport à la moyenne
 - ▶ Afficher le tableau résultant

Tableaux — *Exemple*

⊙ Prototypes des fonctions:

- ▶ Saisir un tableau de réels
 - **void Saisir (float *T, int n);**
- ▶ Calculer la moyenne des éléments du tableau
 - **float Moyenne (float *T, int n);**
- ▶ Remplacer chaque élément du tableau par son écart par rapport à la moyenne
 - **void remplacer (float *T, int n, float moy);**
- ▶ Afficher le tableau résultant
 - **void Afficher (float *T, int n);**

Structures — *Définition*

⊙ Syntaxe de définition de structures

```
typedef struct
{
    type_1 nom_champ_1 ;
    type_2 nom_champ_2 ;
    .
    .
    type_n nom_champ_n ;
} nom_structure ;
```

- **nom_structure** : nom de la structure (nouveau type).
- **nom_champ_1, nom_champ_2,...,nom_champ_n** sont les noms des membres qui caractérisent l'objet.
- **type_1, type_2,...,type_n**, sont les types des différents membres.

⊙ Syntaxe de déclaration de Variables

```
nom_structure s1, s2 ;
nom_structure *s1, *s2 ; /*pointeurs sur une structure*/
```

Structures — *Exemple*

⊙ Type complexe

```
typedef struct
{
    float real ;
    float img  ;
} complexe;
```

```
/* déclaration d'un complexe */
Complexe z1;
```

```
/* déclaration d'un pointeur sur un complexe */
Complexe *z2;
```


Structures — *Accès aux champs*

- ⊙ Pour désigner un **membre** d'une structure on utilise:
 - ▶ L'opérateur de sélection point (.) pour les variables statiques
 - ▶ L'opérateur (→) pour les variables dynamiques

- ⊙ Syntaxe

`nom_variable.nom_champ; /* Variable statique */`

`nom_variable->nom_champ; /* Variable dynamique */`

- **nom_variable** le nom de la variable
- **nom_champ** le nom du membre au quel on veut accéder.

- ⊙ Exemple

`complexe z1,*z2;`

`z2= &z1 ;`

`z1 . img` accède à la partie imaginaire de z1

`z2->img` accède à la partie imaginaire de z1

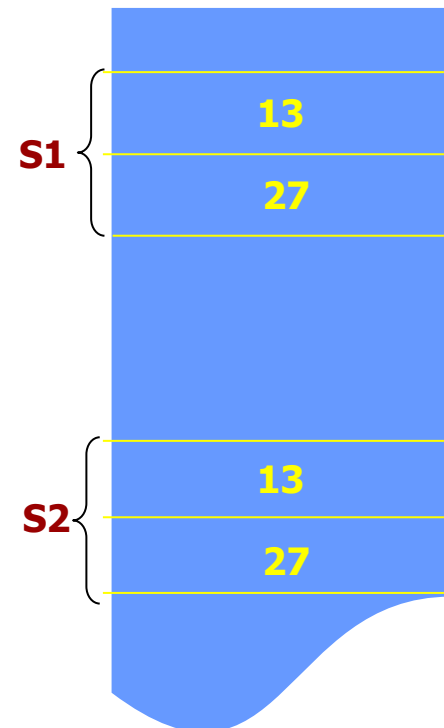
Structures — *Affectation*

- ⊙ **L'affectation** de variables de type structures consiste à transférer les valeurs des champs de la structure source dans leurs homologues de la structure destination.

- ⊙ Syntaxe

```
typedef struct
{
    type_1 nom_champ_1 ;
    type_2 nom_champ_2 ;
    .
    .
    type_n nom_champ_n ;
} nom_structure;
```

```
nom_structure s1, s2 ;
s1 = s2;
```



Structures — *Paramètres de fonctions*

⊙ Définition de la structure

```
typedef struct
{
    type_1 nom_champ_1 ;
    type_2 nom_champ_2 ;
    .
    .
    type_n nom_champ_n ;
} nom_structure;
```

⊙ Prototypes

Comme paramètre fixe	: void fonc1(nom_structure s1) ;
Comme paramètre variable	: void fonc2(nom_structure *s1) ;
Comme type de retour	: nom_structure fonc3(...) ;

⊙ Appels

```
nom_structure *s1, s2, s3 ;

fonc1(*s1);   fonc1(s2) ;
fonc2(s1) ;   fonc2(&s2) ;
s3 = fonc3(...) ;
```

Tableaux de Structures — *Déclaration*

- ⊙ Un tableau de structure est un tableau dont les éléments sont des structures.

- ⊙ **Déclaration**

```
#define CAP 100
typedef struct
{
    type_1 nom_champ_1 ;
    type_2 nom_champ_2 ;
    .
    .
    type_n nom_champ_n ;
} nom_structure;
```

```
nom_structure T[CAP];
```

- ▶ **T** designe le nom du tableau.
- ▶ **nom_structure** type des éléments du tableau.
- ⊙ **T[i]** représente l'élément d'indice **i**, c'est une **variable** de type **nom_structure**,
- ⊙ Pour accéder au **j^{ème}** champ de l'élément **i** on écrit :

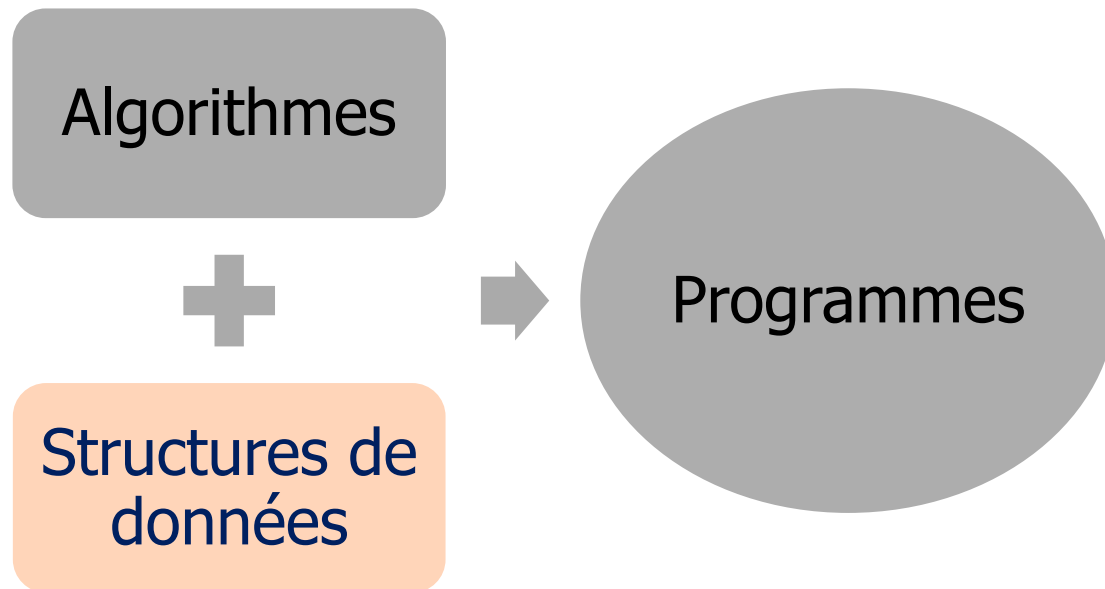
```
T[i].nom_champ_j.
```

Pogrammation

- ⊙ Définition
- ⊙ Démarche
- ⊙ Exemple

Programmation — *Définition*

- ⊙ Activité qui consiste à **exprimer** la solution d'un problème sous forme d'un **programme**.
- ⊙ Equation de Wirth, Livre publié en 1976



Programmation — *Comment ?*

◉ Trois étapes principales



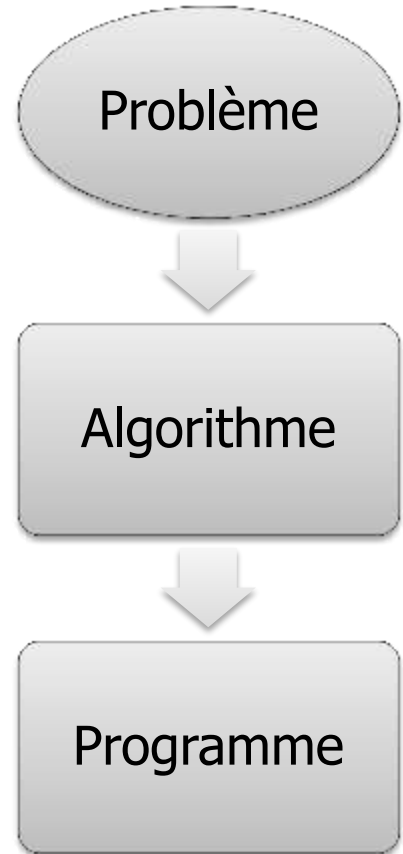
- Description du problème — Structures de données



- Description de la solution — méthode de résolution



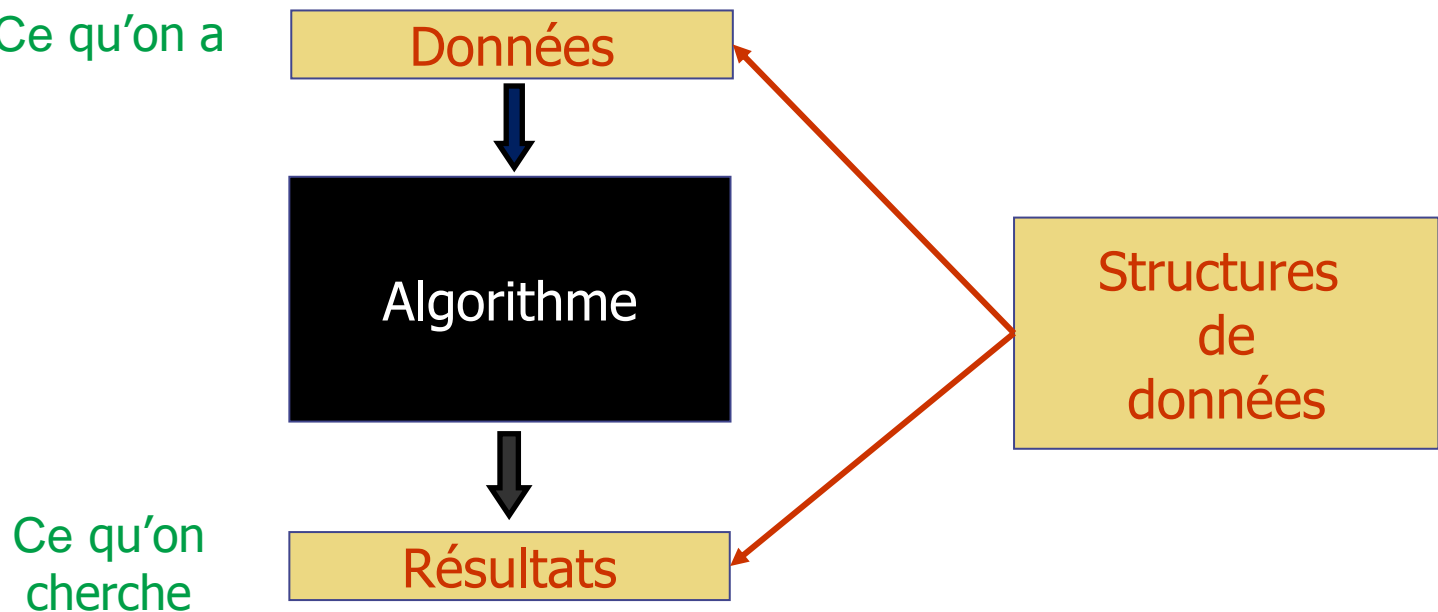
- Traduction dans un langage de programmation



Programmation — *Comment?*

- ⊙ Description du problème — *structures de données*
 - ▶ Consiste à *représenter* et *structurer les données* et les *résultats* du problème

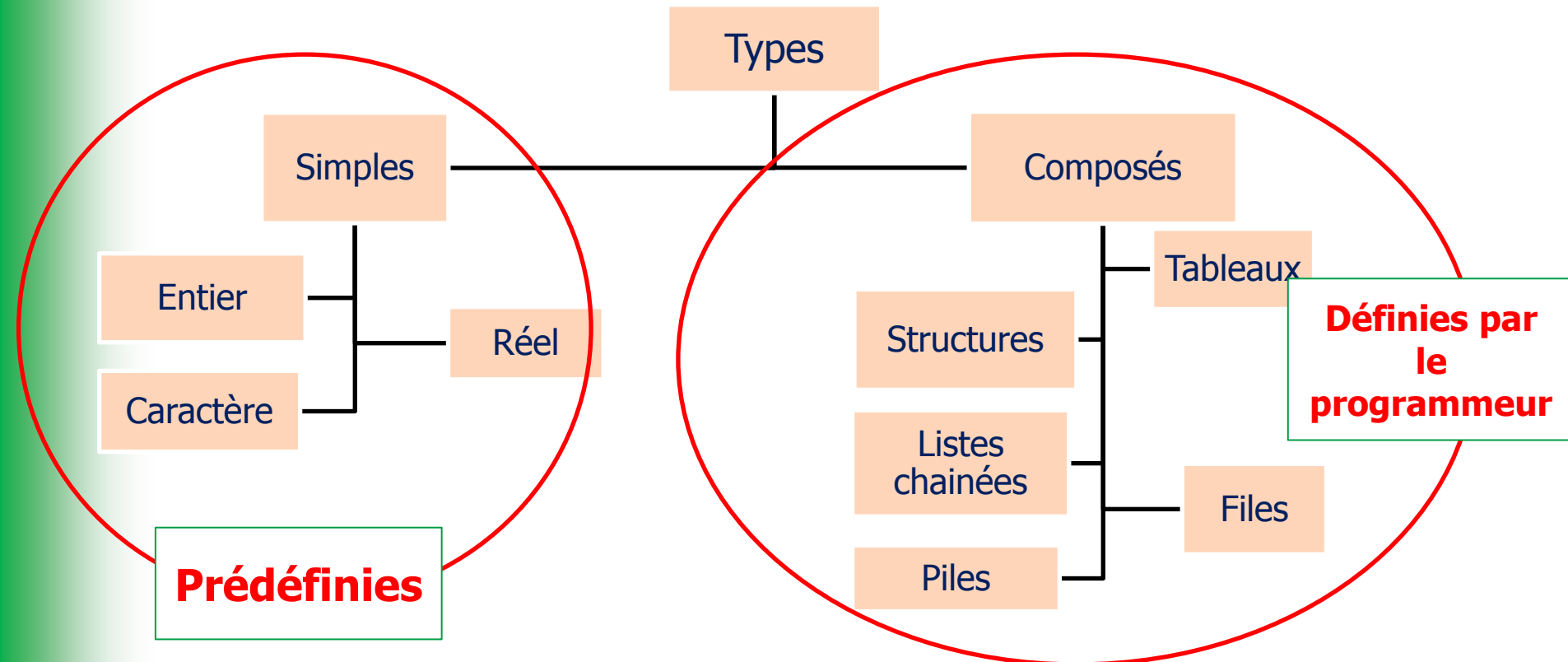
Ce qu'on a



Ce qu'on
cherche

Programmation — *Comment?*

- ◉ Description du problème — *structures de données*
 - Construits à l'aide de la notion de type



Programmation — *Comment?*

- ⊙ Description de la solution — *méthode de résolution*

- ▶ Agit sur les structures de données
- ▶ Exprime la solution à l'aide des éléments algorithmiques : tests, boucles, opérateurs, fonctions, etc.
- ▶ Repose sur des briques de base — les fonctions (sous programmes)



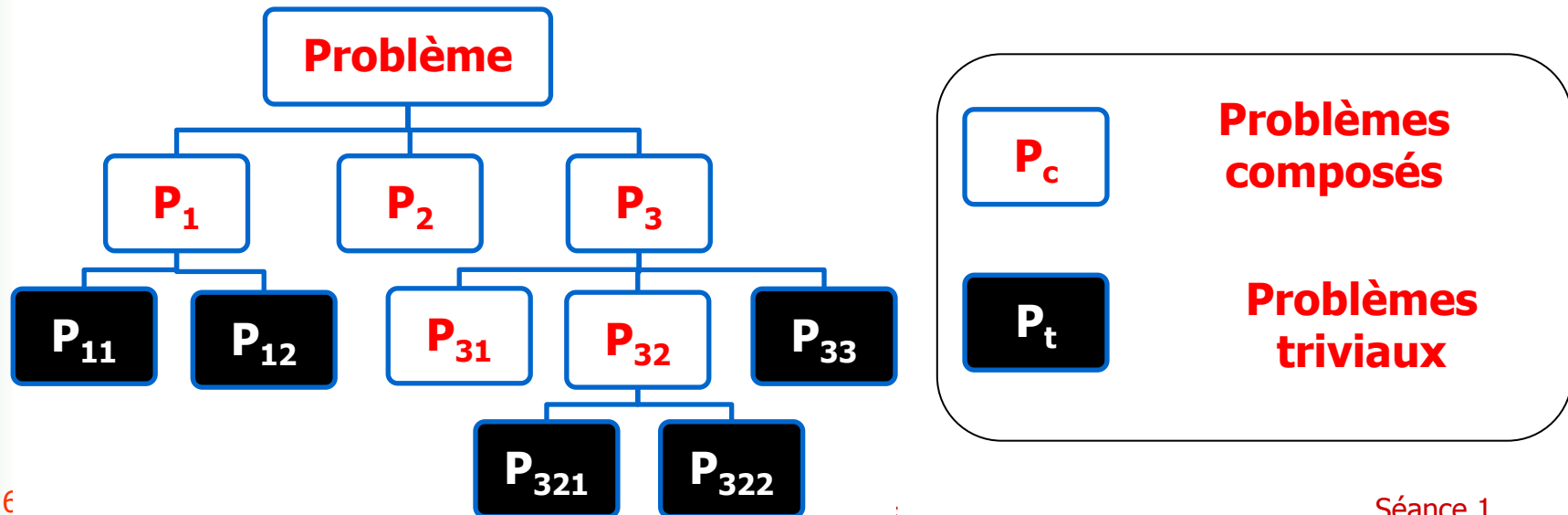
Analyse descendante pour obtenir les fonctions

Programmation — *Comment?*

- ⊙ Description de la solution — *Analyse descendante*
 - ▶ Basée sur la technique **diviser pour régner** qui se déroule en trois étapes :
 - Diviser
 - Régner
 - Résoudre

Programmation — *Comment?*

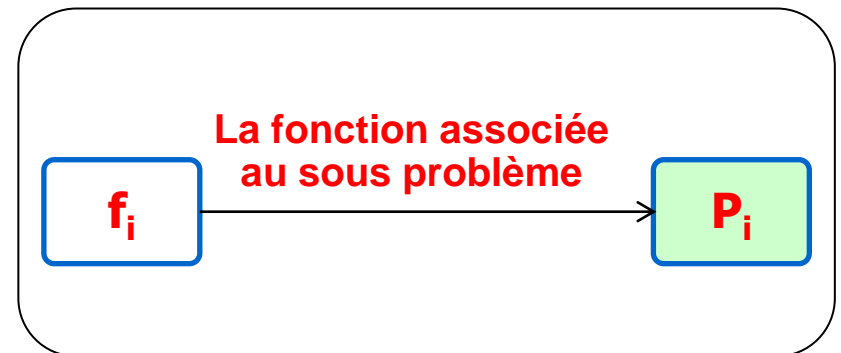
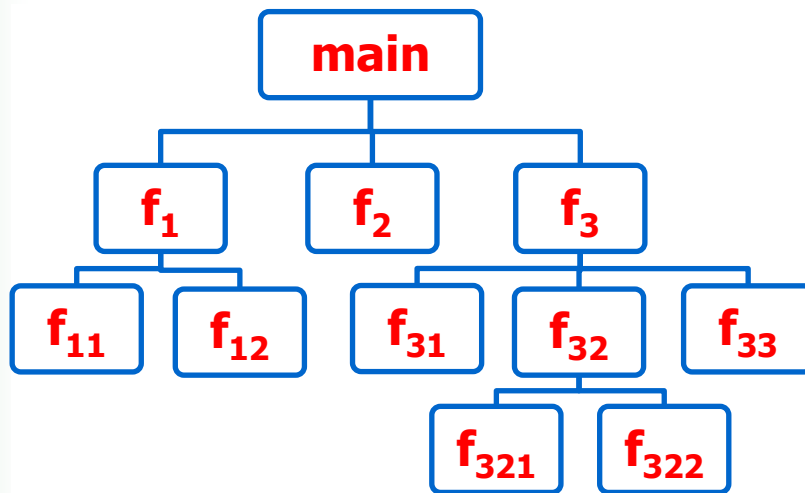
- ◉ Description de la solution — *Diviser*
 - ▶ **Décomposer** le problème en sous problèmes **indépendants**,
 - ▶ Un sous problème peut être :
 - **Composé** : nécessite une nouvelle décomposition
 - **Trivial** : facile à résoudre (la solution est triviale)



Programmation — *Comment?*

◉ Description de la solution — *Analyse descendante*

- ▶ **Régner** — résoudre séparément les sous problèmes
 - Pour chaque sous problème on conçoit une **fonction**.



- ▶ **Résoudre** — il s'agit de combiner les fonctions pour construire la solution globale du problème

Programmation — *Comment?*

⊙ Exemple 1: (P) Calcul de la fonction

$$f_n(x) = \frac{1}{n!} \sum_{i=1}^{i=n} x^i$$

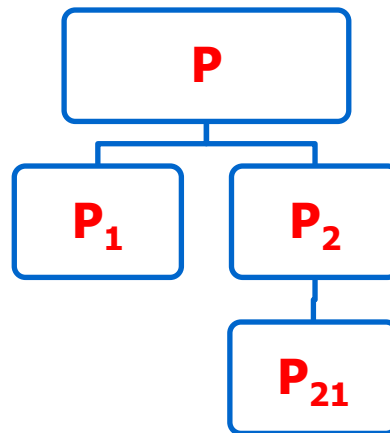
► Structures de données :

- Données : Un entier n et un réel x
- Résultats : Un réel f_x

Programmation — *Comment?*

⊙ Exemple 1: (P) Calcul de la fonction $f_n(x)$

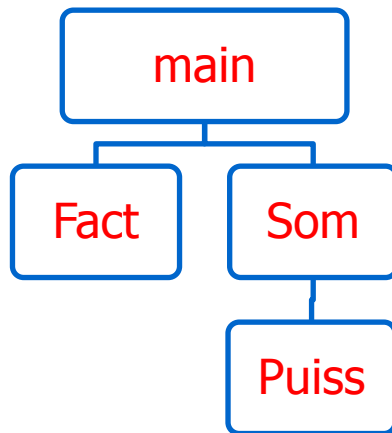
- **Diviser** — Le problème peut être décomposé en trois sous-problèmes :



- **P1** : calculer $n!$, pour n donné.
- **P2** : calculer la somme des puissances croissantes de x
- **P21** : calculer x^n , pour x et n donnés.

Programmation — *Comment?*

- ⊙ Exemple 1: (P) Calcul de la fonction $f_n(x)$
 - Régner — une fonction à chaque sous-problème



Fonction	Paramètres	Sous problème
Fact	int n	P1
Som	float x, int n	P2
Puiss	float x, int n	P21

- Résoudre — la solution globale du problème est :

$$f_x = (1/\text{Fact}(n)) * \text{Som}(x, n)$$

Programmation — *Comment?*

⊙ **Exemple 2 — (P)** Calculer et afficher l'âge d'un étudiant à une date donnée

► **Structures de données :**

- Données : un étudiant , une date
- Résultats : l'âge



Représentation

```
typedef struct{  
  
    int  jour ;  
    int  mois ;  
    int  annee ;  
  
} date ;
```

```
typedef struct{  
  
    char nom[40] ;  
    date d_naissance;  
  
} etudiant;
```

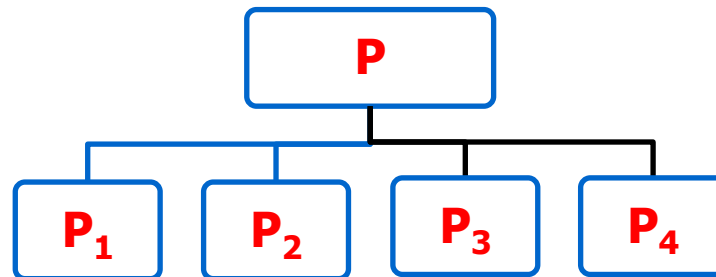
```
typedef struct{  
  
    int  nb_ann ;  
    int  nb_mois;  
    int  nb_jours;  
  
} age;
```

Programmation — *Comment?*

⊙ Exemple 2: (P) Calcul de l'âge d'un étudiant

► Diviser

- P1: saisir une date
- P2 : saisir un étudiant
- P3: calculer l'âge entre deux dates
- P4: afficher l'âge



Programmation — *Comment?*

- ⊙ Exemple 2: (P) Calcul de l'âge d'un étudiant
 - ▶ Régner — une fonction pour chaque sous-problème

Fonction	Paramètres	Sous problème
saisirDate	date * d	P1
saisirEtudiant	etudiant* e	P2
entreDeuxDates	date d1, date d2	P3
afficherAge	Age a	P4

- ▶ Résoudre — la solution globale du problème est :

```
saisirEtudiant(&e);  
saisirDate(&d);  
a=entreDeuxDates(e.d_naissance, d);  
afficherAge(a);
```