

Cinquième partie



Les fonctions

Présentation générale



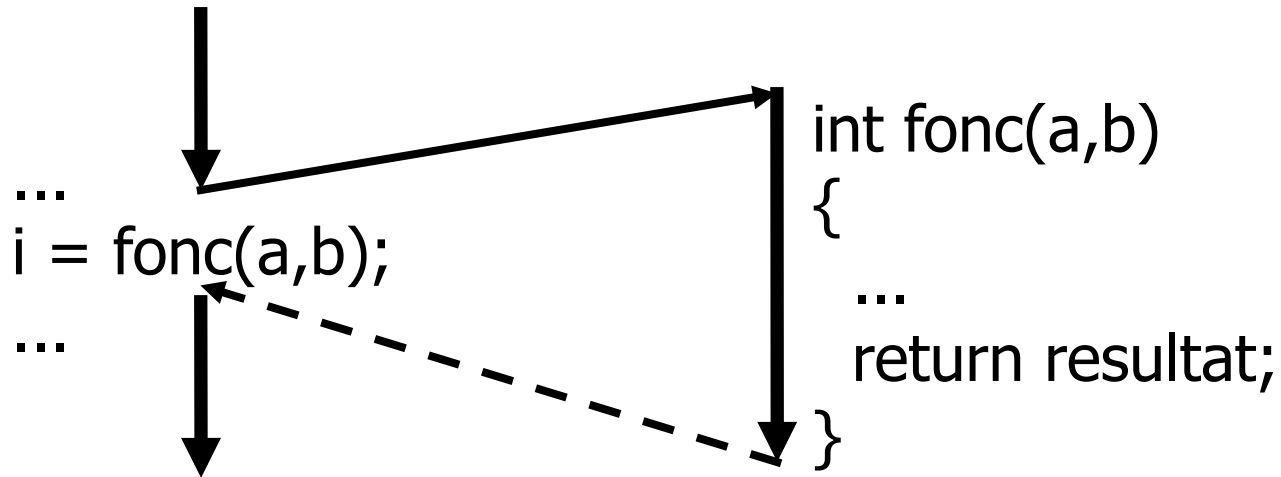
- *Une fonction est une portion de programme formant un tout homogène, destiné à remplir une certaine tâche bien délimitée*
- *Lorsqu'un programme contient plusieurs fonctions, l'ordre dans lequel elles sont écrites est indifférent, mais elles doivent être indépendantes.*

Présentation générale



- *En règle générale, une fonction appelée a pour rôle de traiter les informations qui lui sont passées depuis le point d'appel et de retourner une valeur.*
- *La transmission de ces informations se fait au moyen d'identificateurs spécifiques appelés arguments et la remontée du résultat par l'instruction return.*

Présentation générale



- *Certaines fonctions reçoivent des informations mais ne retournent rien (par exemple `printf`), certaines autres peuvent retourner un ensemble de valeurs (par exemple `scanf`).*

Définition de fonction

- *La définition d'une fonction repose sur trois éléments :*
 - *son type de retour*
 - *la déclaration de ses arguments formels*
 - *son corps*
- *De façon générale, la définition d'une fonction commence donc par :*
 - *type_retour nom (type_arg1 arg1, type_arg2 arg2 ...)*

Définition de fonction

- *L'information retournée par une fonction au programme appelant est transmise au moyen de l'instruction return, dont le rôle est également de rendre le contrôle de l'exécution du programme au point où a été appelée la fonction.*
- *La syntaxe générale de l'instruction return est la suivante :*
 - *return expression;*

Définition de fonction

■ Exemples :

```
int sum_square(int i, int j)
{
    int resultat;
    resultat = i*i + j*j;
    return resultat;
}
```

```
int max(int i, int j)
{
    if(i>j)
        return i;
    else
        return j;
}
```

Les arguments

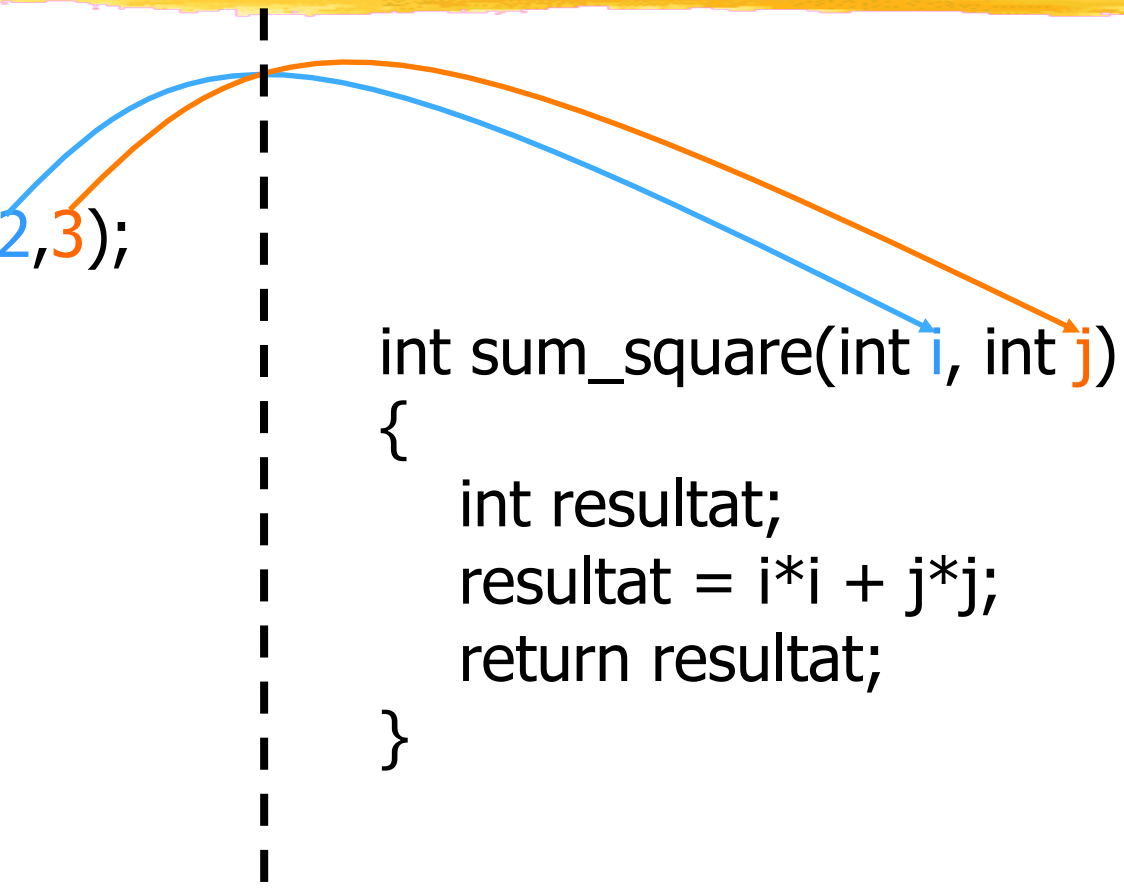


- *Les arguments formels permettent le transfert d'informations entre la partie appelante du programme et la fonctions.*
- *Ils sont locaux à la fonction, et lors de l'appel ils seront mis en correspondance avec les arguments effectifs.*

Les arguments



```
...  
val = sumsquare(2,3);  
...
```



```
int sum_square(int i, int j)  
{  
    int resultat;  
    resultat = i*i + j*j;  
    return resultat;  
}
```

Les arguments

- *La liste des arguments formels peut être vide s'il n'y a aucun argument à passer à la fonction.*
- *La syntaxe des déclarations pour les paramètres formels et les variables n'est pas la même.*
- *Pour déclarer des variables :*
int i; est équivalent à :
int i,j;
int j;
- *Pour les arguments d'une fonction :*
int max(int i,j) / est incorrecte*
**/*
{
...

Les arguments



- *Contrairement à d'autres langages, le C ne permet pas la modification des arguments d'une fonction.*
- *Le seul mode de passage des paramètres est le mode par valeur. Cela signifie que les valeurs des paramètres effectifs sont copiées dans les paramètres formels.*

Appel d'une fonction

- *nom_fonction (liste_d'expressions)*
- *Les expressions de liste_d'expressions sont évaluées, puis passées en tant qu'arguments effectifs à la fonction de nom nom_fonction, qui est ensuite exécutée.*
- *L'appel d'une fonction est une expression et non une instruction. La valeur rendue par la fonction est l'évaluation de l'expression appel de fonction.*
- *Exemples :*
s = sum_square(a,b);
m = max(a,b);

Appel d'une fonction

- *Dans le cas d'une fonction sans paramètre, la liste des paramètres doit être vide :*

```
d = fonc(void);          /* appel incorrect
*/
```

- *L'ordre d'évaluation des paramètres effectifs n'est pas spécifiés :*

```
sum_square(f(x),g(y));
```

-> La fonction g sera peut-être exécutée avant f.

Déclaration de fonction

- *Lorsque l'appel d'une fonction figure avant sa définition, la fonction appelante doit contenir une déclaration de la fonction appelée. On appelle cela prototype de la fonction.*
- *Attention a ne pas confondre définition et déclaration de fonctions. La déclaration est une indication pour le compilateur quant au type de résultat renvoyé par la fonction et éventuellement au type des arguments, et rien de plus.*

Déclaration de fonction

- Dans sa forme la plus simple la déclaration d'une fonction peut s'écrire :
 - `type_de_retour nom();`
- Les prototypes de fonctions sont souvent regroupés par thèmes dans des fichiers dont l'extension est généralement h (ex `stdio.h`).
- Ces fichiers sont inclus dans le source du programme par une directive du préprocesseur : `#include`.

Récusivité

- *La récursivité est la caractéristique des fonctions capables de s'appeler elles-mêmes de façon répétitive, jusqu'à ce que soit vérifiée une condition d'arrêt.*
- *Il n'y a rien de spécial à faire pour qu'une fonction puisse être appelée de manière récursive.*
- *Exemple :*

```
int facto(int n)
{
    if (n==1) return 1;
    else return (n*facto(n-1));
}
```


Les procédures

- *Il n'y a pas de concept de procédure à proprement parler en C.*
- *Pour cela on déclare une fonction qui ne retourne aucune valeur grâce au mot-clé `void` comme type de retour.*
- *Exemple :*

```
void print_add(int a, int b)
{
    printf("%i", a+b);
}
```

Les fonctions imbriquées

- *A l'inverse de certains langages, les fonctions imbriquées n'existent pas dans le langage C. Il n'est donc pas possible qu'une fonction ne soit connue qu'à l'intérieur d'une autre fonction.*

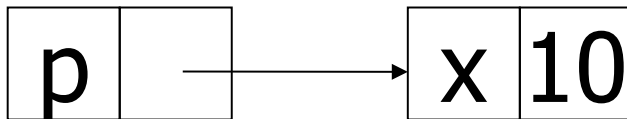
Sixième partie



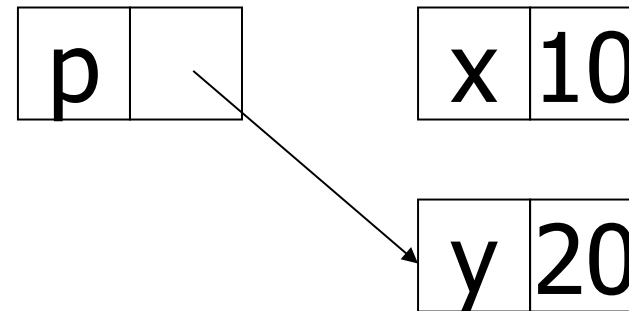
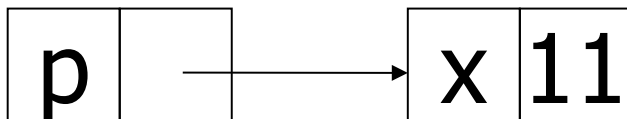
Les pointeurs et les tableaux

Pointeurs et adresses

- Une variable désigne une valeur.
- On peut être amené parfois à vouloir désigner une variable par une autre variable.
- Par exemple *p* désigne la variable *x* qui a pour valeur 10



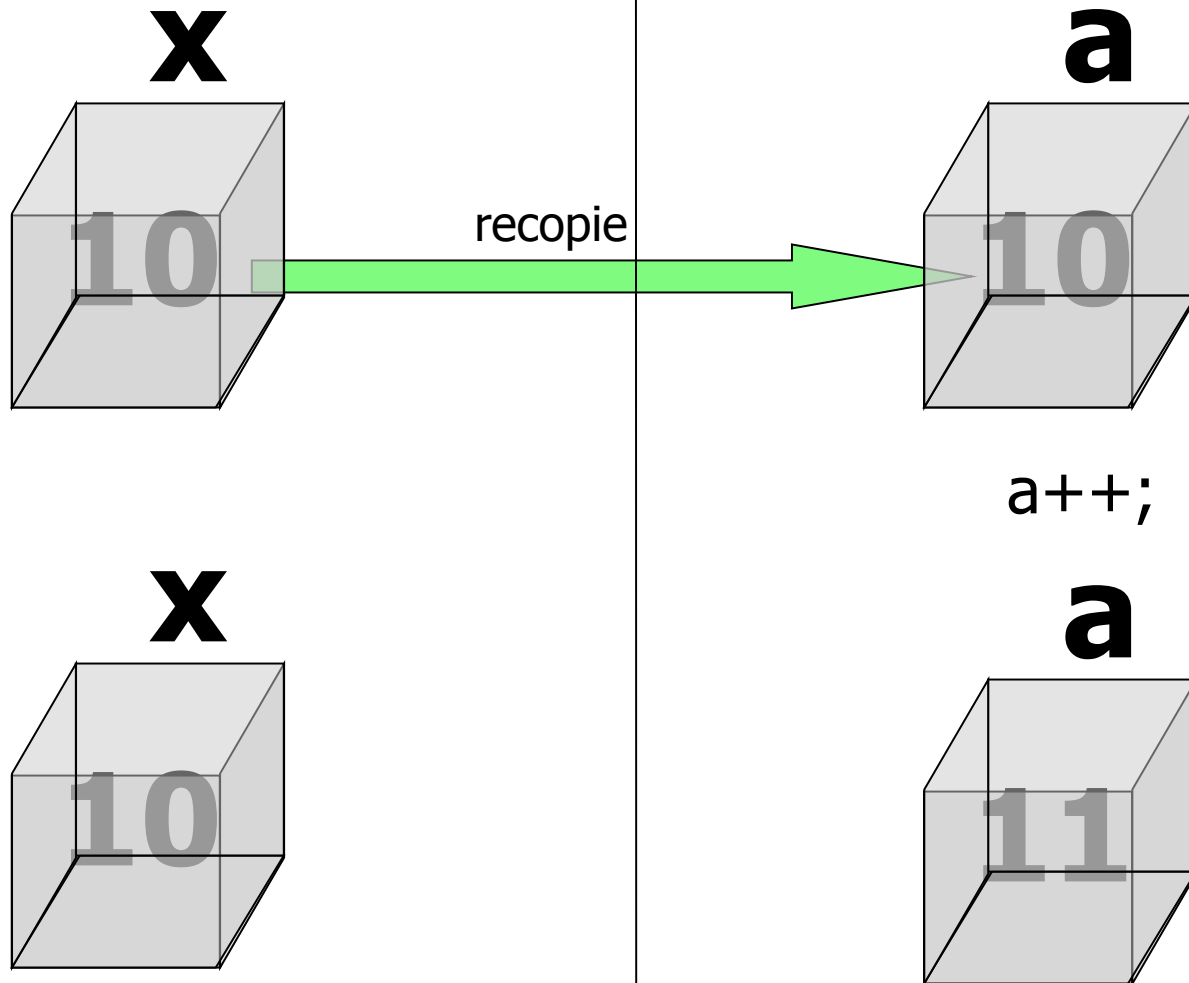
x++;



Pointeurs et adresses

Programme principal

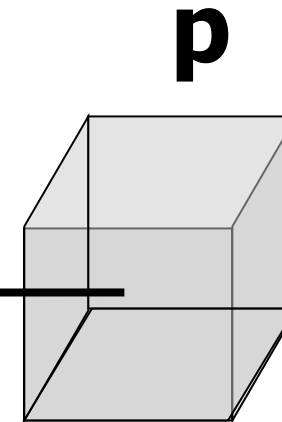
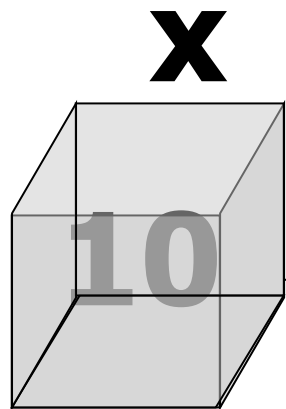
Fonction



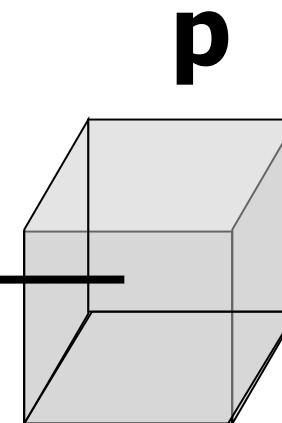
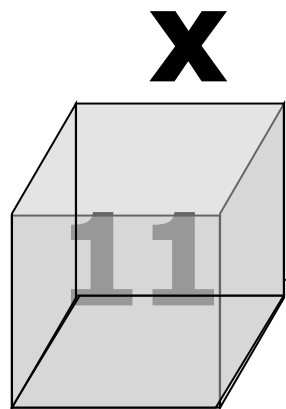
Pointeurs et adresses

Programme principal

Fonction



incrémentation



Pointeurs et adresses



- *Un pointeur est une adresse permettant de désigner un objet (une variable ou une fonction) en mémoire centrale.*
- *Par extension, on appelle pointeur la variable qui contient cette adresse.*

Pointeurs et adresses

- *Un pointeur définissant l'adresse d'un objet, l'accès à cet objet peut alors être réalisé par une indirection sur le pointeur.*
 - *L'opérateur unaire & fournit une adresse de l'objet opérande (qui doit donc être une lvalue). On dit que cet opérateur est l'opérateur de **référence**.*
 - *L'opérateur unaire * considère son opérande comme un pointeur et retourne l'objet pointé par celui-ci. On dit aussi que cet opérateur est l'opérateur **d'indirection**.*

Pointeurs et adresses

■ Exemple :

```
int x,y;  
/* Soit px un pointeur sur des int */  
px = &x;      /* px <- l'adresse de x */  
y = *px;      /* y <- l'objet pointé par px  
*/
```

■ Ceci équivaut donc à $y=x$;

Pointeurs et adresses

■ *La déclaration d'un pointeur doit spécifier le type de l'objet pointé. On dit alors que le pointeur est typé.*

■ *Exemple :*

*int *PX;*

➤ *signifie que PX pointera des objets de type int.
(Littéralement : *PX sera un int)*

Pointeurs et adresses



- *La définition d'un pointeur ne fait rien d'autre que ce qu'elle doit faire, c'est à dire qu'elle réserve en mémoire la place nécessaire pour mémoriser un pointeur, et en aucune façon de la place pour mémoriser un objet du type pointé*
- *L'objet atteint par l'intermédiaire du pointeur possède toutes les propriétés du type correspondant.*

Pointeurs et adresses

■ Exemples :

➤ $PX = \&X;$

➤ $Y = *PX + 1;$

$/* Y = X + 1 */$

➤ $*PX = 0;$

$/* X = 0 */$

➤ $*PX += 10;$

$/* X = X + 10 */$

➤ $(*PX)++;$

$/* X = X + 1 */$

Pointeurs et adresses

- *Un pointeur pouvant repérer un type quelconque, il est donc possible d'avoir un pointeur de pointeur.*

- Exemple :

```
int **ppx;  
/* ppx pointe un pointeur d'entiers  
*/
```

Opérations sur les pointeurs

- *La valeur NULL, est une valeur de pointeur, constante et prédéfinie dans stddef.h.*
- *Elle vaut 0 et signifie "Aucun objet".*
- *Cette valeur peut être affectée à tout pointeur, quel que soit son type.*
 - *Dans ce cas, ce pointeur ne pointe sur rien...*
 - *Bien entendu, l'utilisation de cette valeur dans une indirection provoquera une erreur d'exécution.*

Opérations sur les pointeurs

- *L'affectation d'un pointeur à un autre n'est autorisée que si les 2 pointeurs pointent le même type d'objet (ie ont le même type)*

- *Exemple :*

```
P = NULL;  
P = Q;                /* P et Q sont des pointeurs  
sur le même type */  
P = 0177300;          /* illégal */  
P = (int *) 0177300;  /* légal */
```

Opérations sur les pointeurs

- *L'incrémentation d'un pointeur par un entier n est autorisée.*
- *Elle ne signifie surtout pas que l'adresse contenue dans le pointeur est incrémentée de n car alors cette adresse pourrait désigner une information non cohérente : être à cheval sur 2 mots par exemple...*
- *L'incrémentation d'un pointeur tient compte du type des objets pointés par celui-ci : elle signifie "passe à l'objet du type pointé qui suit immédiatement en mémoire".*

Opérations sur les pointeurs

- *Ceci revient donc à augmenter l'adresse contenue dans le pointeur par la taille des objets pointés.*
- *Dans le cas d'une valeur négative, une décrémentation a lieu.*
- *Les opérateurs combinés avec l'affectation sont autorisés avec les pointeurs.*
- *Il en va de même pour les incrémentations / décrémentations explicites.*

Opérations sur les pointeurs

■ Exemples :

*int *Ptr, k; /* Ptr est un pointeur , k est un int */*

Ptr++;

Ptr += k;

Ptr--;

Ptr -= k;

Comparaison de pointeurs

- *Il est possible de comparer des pointeurs à l'aide des relations habituelles : < <= > >= == !=*
- *Ces opérations n'ont de sens que si le programmeur a une idée de l'implantation des objets en mémoire*
- *Un pointeur peut être comparé à la valeur NULL.*

Comparaison de pointeurs

■ *Exemples :*

```
int *Ptr1, *Ptr2;
```

```
...
```

```
if (Ptr1 <= Ptr2)
```

```
...
```

```
if (Ptr1 == 0177300)                /* illégal */
```

```
...
```

```
if (Ptr1 == (int *)0177300) /* légal */
```

```
...
```

```
if (Ptr1 != NULL)
```

Soustraction de pointeurs



- *La différence de 2 pointeurs est possible, pourvu qu'ils pointent sur le même type d'objets.*
- *Cette différence fournira le nombre d'unités de type pointé, placées entre les adresses définies par ces 2 pointeurs.*
- *Autrement dit, la différence entre 2 pointeurs fournit la valeur entière qu'il faut ajouter au deuxième pointeur (au sens de l'incrémentatation de pointeurs vue plus haut) pour obtenir le premier pointeur.*

Affectation de chaînes de caractères

- ***L'utilisation d'un littéral chaîne de caractères se traduit par :***
 - *la réservation en mémoire de la place nécessaire pour contenir ce littéral (complété par le caractère '\0')*
 - *et la production d'un pointeur sur le premier caractère de la chaîne ainsi mémorisée.*
- ***Il est alors possible d'utiliser cette valeur dans une affectation de pointeurs.***

Affectation de chaînes de caractères

- *Exemple :*

*char *Message;*

Message = "Langage C"; / est autorisé */*

- *La chaîne est mémorisée et complétée par '\0'.*

- *La variable Message reçoit l'adresse du caractère 'L'.*

Pointeurs et tableaux

- *On rappelle que la déclaration d'un tableau dans la langage C est de la forme :
 `int Tab[10];`*
- *Cette ligne déclare un tableau de valeurs entières dont les indices varieront de la valeur 0 à la valeur 9.*

Pointeurs et tableaux

- ***En fait, cette déclaration est du "sucre syntaxique" donné au programmeur. En effet, de façon interne, elle entraîne :***

- *La définition d'une valeur de pointeur* sur le type des éléments du tableau, cette valeur est désignée par le nom même du tableau (Ici Tab)
- *La réservation de la place mémoire* nécessaire au 10 éléments du tableau, alloués consécutivement.
L'adresse du premier élément (Tab[0]) définit la valeur du pointeur Tab.

Pointeurs et tableaux

- *La désignation d'un élément (Tab[5] par exemple) est automatiquement traduite, par le compilateur, en un chemin d'accès utilisant le nom du tableau (Dans notre exemple ce serait donc $*(Tab + 5)$)*

Pointeurs et tableaux

■ *Exemples :*

```
int *Ptab, Tab[10];
```

■ *On peut écrire :*

➤ *Ptab = &Tab[0];*

➤ *Ptab = Tab; /* Equivalent au précédent */*

➤ *Tab[1] = 1;*

➤ **(Tab + 1) = 1; /* Equivalent au précédent */*

■ *Donc, pour résumer, on a les équivalences suivantes :*

➤ *Tab + 1 est équivalent à &(Tab[1])*

➤ **(Tab + 1) est équivalent à Tab[1]*

➤ **(Tab + k) est équivalent à Tab[k]*

Pointeurs et tableaux

- *Aucun contrôle n'est fait pour s'assurer que l'élément du tableau existe effectivement :*
- *Si $Pint$ est égal à $\&Tab[0]$, alors $(Pint - k)$ est légal et repère un élément hors des bornes du tableau Tab lorsque k est strictement positif.*
- *Le nom d'un tableau n'est pas une lvalue, donc certaines opérations sont impossibles (exemple : $Tab++$)*

Pointeurs et tableaux

```
#include <stdio.h>
main()
{
    char Tab[32], *Ptr;
    /* Initialisation des données */
    Tab[0] = 'Q'; Tab[1] = 'W'; Tab[2] = '\0';
    Ptr = "ASDFGHJKL";
    /* Edition des chaines */
    printf("Contenu des chaines : \n");
    printf("  Tab : %s\n  Ptr : %s\n", Tab, Ptr);
    /* Utilisation des tableaux */
    printf("Edition de l'élément de rang 1 des tableaux\n");
    printf("  Tab : %c\n  Ptr : %c\n", Tab[1], Ptr[1]);
    /* Utilisation des pointeurs */
    printf("Edition du caractère pointé\n");
    printf("  Tab : %c\n  Ptr : %c\n", *Tab, *Ptr);
    /* Utilisation des pointeurs incrémentés */
    printf("Edition du caractère suivant le caractère pointé\n");
    printf("  Tab : %c\n  Ptr : %c\n", *(Tab + 1), *(Ptr + 1));
    /* Règle des priorités */
    printf ("Edition identique non parenthésée \n");
    printf("  Tab : %c\n  Ptr : %c\n", *Tab + 1, *Ptr + 1);
}
```

Passage des paramètres



- *Le seul mode de passage des paramètres à une fonction est le passage par valeur.*
- *Le problème est que lorsqu'un sous-programme doit fournir une réponse par un de ses paramètres, ou modifier ceux-ci, le mode de passage par valeur ne peut plus convenir.*

Passage des paramètres

- *De plus, dans le cas de paramètres structurés de taille importante, la copie de ceux-ci entraîne une consommation mémoire parfois superflue.*
- *C'est à la charge du programmeur de gérer tous ces problèmes en fournissant un pointeur sur l'objet en question.*
- *Ceci permet au sous-programme :*
 - *de travailler directement sur l'objet réel et donc de le modifier,*
 - *mais aussi réduit la taille des échanges, puisqu'il n'y a copie que d'une adresse.*

Passage des paramètres

<pre>void Echange_1 (int A,int B) { int C = A; A = B; B = C; }</pre>	<pre>void Echange_2 (int *A,int *B) { int C = *A; *A = *B; *B = C; }</pre>
<pre>main() { int X, Y; printf("Valeur de X : "); scanf ("%d",&X); printf("\nValeur de Y : "); scanf ("%d",&Y); Echange_1 (X,Y) printf("\nAprès appel de Echange_1 :\n X = %d\n Y = %d\n", X, Y); Echange_2 (&X,&Y) printf("Après appel de Echange_1 :\n X = %d\n Y = %d\n", X, Y); }</pre>	

Passage des paramètres

- *Quand un tableau est transmis comme paramètre effectif à un sous-programme, c'est en fait l'adresse de base de ce tableau qui est transmise par valeur.*
- *Ceci a pour effet de transmettre le tableau par référence sans intervention du programmeur.*
- *De plus, à l'intérieur de la fonction appelée, le paramètre tableau devient une lvalue, certaines opérations sont donc possibles.*

Passage des paramètres

```
#include <stdio.h>
void Test_Tab (int T[5])
{
    printf(" Editions dans la fonction \n");
    printf(" Valeur du paramètre tableau : %o \n",T);
    printf(" Valeur du contenu : %d\n",T[0]);
    printf(" Test d'incrémentatation paramètre \n");
    T++; /* T est devenue une lvalue */
    printf(" Valeur du paramètre tableau : %o \n",T);
}

main()
{
    int Tab[5];
    Tab[0] = 10;
    printf("Editions avant appel de fonction\n");
    printf(" Valeur du paramètre tableau : %o\n",Tab);
    printf(" Valeur du contenu : %d\n",Tab[0]);
    Test_Tab(Tab);
}
```

Pointeurs et tableaux multidimensionnels

- *Un tableau unidimensionnel peut se représenter grâce à un pointeur (le nom du tableau) et un décalage (l'indice).*
- *Un tableau à plusieurs dimensions peut se représenter à l'aide d'une notation similaire, construite avec des pointeurs.*
- *Par exemple, un tableau de dimension 2, est en fait un ensemble de deux tableaux à une seule dimension.*
- *Il est ainsi possible de considérer ce tableau à deux dimensions comme un pointeur vers un groupe de tableaux unidimensionnels consécutifs.*

Pointeurs et tableaux multidimensionnels

- ***On peut donc écrire la déclaration suivante:***

- *type-donnée (*varpt)[expression 2];*

- ***au lieu de la déclaration classique:***

- *type-donnée tableau [expression 1][expression 2];*

- ***Ce style de déclaration peut se généraliser à un tableau de dimension n de la façon suivante:***

- *type-donnée (*varpt)[expression 2][expression 3] ...
[expression n];*

- ***qui remplace la déclaration équivalente:***

- *type-donnée tableau[expression 1][expression 2] ...
[expression n];*

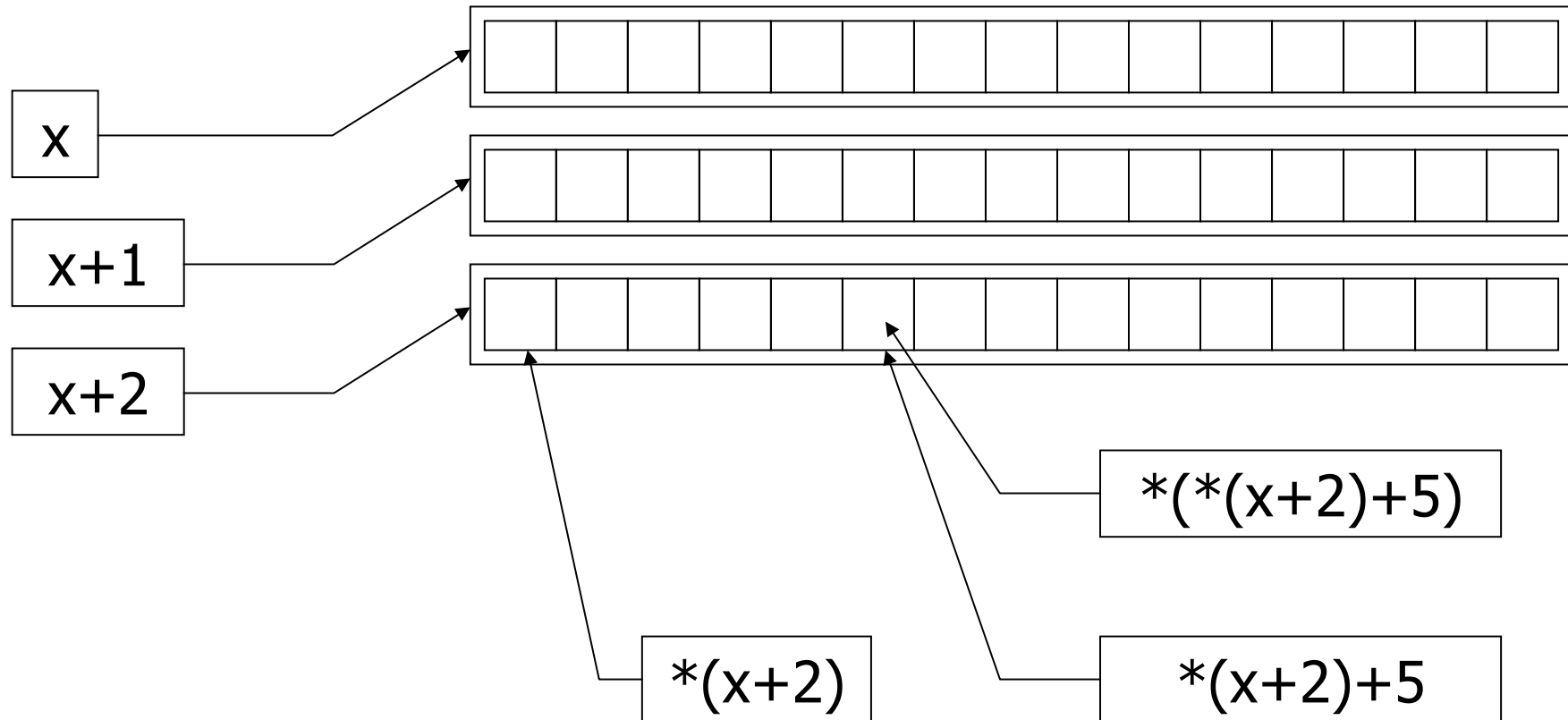
Pointeurs et tableaux multidimensionnels

- *Exemple : On suppose que x est un tableau d'entiers de dimension 2, ayant 10 lignes et 20 colonnes.*
- *On peut le déclarer de la façon suivante:*
*int (*x) [20];*
- *au lieu de :*
int x[10] [20];

Pointeurs et tableaux multidimensionnels

- *Dans la première déclaration, on dit que x est un pointeur vers un groupe de tableaux d'entiers contigus, possédant chacun une dimension et 20 éléments.*
- *Ainsi, x pointe le premier tableau de 20 éléments, qui constitue en fait la première ligne (ligne 0) du tableau d'origine de dimension 2.*
- *De manière analogue, $(x+1)$ pointe vers le deuxième tableau de 20 éléments, qui représente la seconde ligne (ligne 1) du tableau d'origine, et ainsi de suite*

Pointeurs et tableaux multidimensionnels



Pointeurs et tableaux multidimensionnels

- *Si l'on considère à présent un tableau tridimensionnel de réels, t, on peut le définir par:
 `float (*t) [20] [30];`*
- *au lieu de :
 `float t[10] [20] [30];`*
- ***La première forme définit t comme un pointeur vers un groupe contigu de tableaux de réels, de dimension 20 x 30.***
- ***Dans ce cas t pointe le premier tableau 20 x 30, (t+1) pointe le second tableau 20x30, etc.***

Pointeurs et tableaux multidimensionnels

- *Pour accéder à un élément donné d'un tableau à plusieurs dimensions, on applique plusieurs fois l'opérateur d'indirection. Mais cette méthode est souvent plus pénible d'emploi que la méthode classique.*
- *Exemple : x est un tableau d'entiers de dimension 2, à 10 lignes et 20 colonnes. L'élément situé en ligne 2 et colonne 5 peut se désigner aussi bien par :*
$$x[2][5] \qquad \text{que par} \qquad * (* (x+2) + 5)$$

Pointeurs et tableaux multidimensionnels

- *Les programmes mettant en jeu des tableaux multidimensionnels peuvent s'écrire de différentes manières.*
- *Choisir entre ces formes relève surtout des goûts personnels du programmeur.*

Tableaux de pointeurs

- *Les pointeurs étant des informations comme les autres, ils peuvent être mis dans des tableaux.*
 - *Exemple : `char *Ptr_Char[7];`*
- *Cette déclaration se lit comme : `Ptr_Char` est un tableau dont les éléments sont du type `char*`.*
- *Les 7 éléments de `Ptr_Char` sont donc des pointeurs vers des caractères.*

Tableaux de pointeurs

- *D'après ce qui a été vu précédemment cela signifie que chaque élément du tableau peut pointer vers une chaîne de caractères.*
- *On conçoit qu'avec une telle représentation, une permutation de chaîne est très facile puisqu'il suffit de permuter les pointeurs correspondants.*
- *De plus, les différentes chaînes peuvent avoir des tailles différentes, ce qui ne serait pas possible si l'on déclarait un tableau de caractères à 2 dimensions comme :*
char Tab_Char[7][10];

Initialisation des tableaux multidimensionnels

- *On peut initialiser un tableau d'entiers à deux dimensions de la façon suivante :*

➤ *`int t[2][3] = {{1,2,3},{4,5,6}};`*

- *Ou encore :*

➤ *`int t[2][3] = {1,2,3,4,5,6};`*

- *Car les éléments sont placés dans l'ordre suivant : `t[0][0]`, `t[0][1]`, `t[0][2]`, `t[1][0]`, `t[1][1]`, `t[1][2]`, c'est-à-dire ligne après ligne.*

Pointeurs de fonctions

- *Une fonction ne peut être considérée comme une variable; cependant elle est implantée en mémoire et son adresse peut être définie comme l'adresse de la première instruction de son code.*
- *Lorsque le compilateur reconnaît, dans une expression, l'identificateur d'une fonction qui ne correspond pas à un appel de fonction (absence de parenthèses), il engendre alors l'adresse de cette fonction.*
- *Il est donc possible de définir un pointeur sur une fonction, ce qui sera particulièrement utile lorsqu'on désirera passer une fonction en paramètre.*

Pointeurs de fonctions

■ *Exemple :*

*int (*f)(); /* pointeur de fonction retournant un entier */*

- *Ici (*f) signifie : f est un pointeur; *f définit l'objet pointé*
- *(*f)() signifie : l'objet pointé est une fonction*
- *int (*f)() signifie : l'objet pointé est une fonction qui retourne un entier*

■ **ATTENTION !!! à ne pas confondre :**

*int (*f)() avec : int *f ()*

Pointeurs de fonctions



```
int fonc1()
{
    return 1;
}
int fonc2()
{
    return 2;
}

int main()
{
    int (*f)();

    f=fonc1;
    printf("resultat : %i\n",f());
    f=fonc2;
    printf("resultat : %i\n",f());
    return 0;
}
```


Pointeurs de fonctions

```
int add(int a, int b)
{
    return a+b;
}
int mul(int a, int b)
{
    return a*b;
}

int main()
{
    int (*f)(int, int);

    f=add;
    printf("resultat : %i\n",f(2,3));
    f=mul;
    printf("resultat : %i\n",f(2,3));
    return 0;
}
```

```
int max(int a,int b)
{
    return(a>b?a:b);
}
int min(int a,int b)
{
    return(a<b?a:b);
}
void main(void);
{
    int (*calcul)(int,int);
    char c;
    printf("utiliser mAx (A) ou mIn (I) ?");
    do
        c=getchar();
    while ((c!='A')&&(c!='I'));
    calcul=(c=='A')?&max:&min;
    printf("%d\n",(*calcul)(10,20));
}
```

Allocation dynamique

- *La bibliothèque standard C dispose de 3 fonctions d'allocation : malloc, calloc et realloc qui ressemblent au new pascal, et une fonction de désallocation : free*
- *Elles ont leur prototype dans stdlib.h.*

Allocation dynamique

- `void *malloc (size_t tob) ;`
- Le type `size_t` est défini dans `stdlib.h` et est équivalent à `unsigned int` ou `unsigned long int` (ça dépend des systèmes).
- `tob` représente une taille d'objet en octets.
- `malloc` renvoie un pointeur universel (`void *`) vers le début de la zone allouée de `tob` octets.
- `malloc` renvoie `NULL` si l'allocation a échoué.

Allocation dynamique



```
#include <stdio.h>

int main()
{
    char chaine[80];
    printf("Entrez une chaîne :");
    scanf("%s",chaine);
    printf("Voici la chaîne : %s",chaine);
    return 0;
}
```

Allocation dynamique



```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char *chaine;
    chaine = (char *)malloc(sizeof(char)*80);
    if(chaine)
    {
        printf("Entrez une chaîne :");
        scanf("%s",chaine);
        printf("Voici la chaîne : %s",chaine);
    }
    else
        printf("Plus de mémoire !!!");
    return 0;
}
```

Allocation dynamique

- *`void *calloc (size_t nob, size_t tob) ;`*
- *`nob` représente un nombre d'objets.*
- *`calloc` renvoie un pointeur universel (`void *`) vers le début de la zone allouée de `nob` objets de taille `tob` octets (tableau dynamique) .*
- *`calloc` renvoie `NULL` si l'allocation a échoué.*

Allocation dynamique



```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char *chaine;
    chaine = (char *)calloc(80,sizeof(char));
    if(chaine)
    {
        printf("Entrez une chaîne :");
        scanf("%s",chaine);
        printf("Voici la chaîne : %s",chaine);
    }
    else
        printf("Plus de mémoire !!!");
    return 0;
}
```

Allocation dynamique



- ***void *realloc (void *ptr, size_t tob) ;***
- ***realloc permet de modifier la taille d'une zone précédemment allouée.***
- ***realloc renvoie NULL si l'allocation a échoué.***

Allocation dynamique



```
int main()
{
    char *chaine;
    chaine = (char *)calloc(1000,sizeof(char));
    if(chaine)
    {
        printf("Entrez une chaîne :");
        scanf("%s",chaine);
        chaine = (char*)realloc(chaine,sizeof(char)*(strlen(chaine)+1));
        printf("Voici la chaîne : %s\n",chaine);
    }
    else
    {
        printf("Plus de mémoire !!!");
    }
    return 0;
}
```

Allocation dynamique



- *void free (void *) ;*
- *free rend au système (désalloue) la zone pointée par le pointeur paramètre. Si ce pointeur a déjà été désalloué ou s'il ne correspond pas à une valeur allouée par malloc, calloc ou realloc, il y a une erreur.*
- *Le comportement du système est alors imprévisible.*

Allocation dynamique



```
int main()
{
    char *chaine;
    chaine = (char *)calloc(1000,sizeof(char));
    if(chaine)
    {
        ...
        free(chaine);
    }
    else
    {
        printf("Plus de mémoire !!!");
    }
    return 0;
}
```

Allocation dynamique

- ***Exemple : Une fonction qui reçoit en argument un pointeur à initialiser par malloc et qui renvoie un booléen indiquant si l'allocation s'est bien passée.***

```
int alloc (int **pp) /* pp est un pointeur de pointeur d'entier */
{
    *pp=(int *) malloc (sizeof (int)) ;
    return *pp != NULL ;
}
main ()
{
    ...
    int *p ;
    p=NULL ;
    if (alloc (&p)) /* adresse du pointeur */
        *p=5 ; /* p a été modifié par alloc */
    ..... /* il n'est plus à NULL */
}
```