Université Internationale de Casablanca

LAUREATE INTERNATIONAL UNIVERSITIES

Programmation Structurée II

ECOLE D'INGENIERIE – MIAGE

DEUXIEME ANNEE

Abdallah MOUJAHID – PMP®, ITIL®, COBIT® v5, ISO 27002

abdallah.moujahid@uic.ac.ma

CH 2 : LES FONCTIONS

- Fonctions en C – Les principes
- Déclarer une fonction
- Définition d'une fonction
- Appel d’une fonction
- Règles de visibilité des variables
- Passage des paramètres par valeur
- Les fonctions récursives
- Passer des tableaux aux fonctions

Fonctions en C – Les principes

- En C, tout est fonction (Définition, Déclaration et Appel).
- Une fonction peut avoir autant de paramètres que l'on veut, même aucun (comme void main(void))
- Une fonction renvoie une valeur ou aucune.
- Les variables déclarées dans une fonction sont locales à cette fonction
- Une fonction possède un et un seul point d'entrée, mais éventuellement plusieurs points de sortie (à l'aide du mot **return**).
- L'imbrication de fonctions n'est pas autorisée:

  une fonction ne peut pas être déclarée à l'intérieur d'une autre fonction. Par contre, une fonction peut appeler une autre fonction. Cette dernière doit être déclarée avant celle qui l'appelle.

Fonctions en C

Encapsule un traitement particulier formant un tout:

- Peut implémenter la notion de module en logique
- Notion de librairie
- Augmente la lisibilité d'un programme
- Réalise un objectif précis
- Améliore le débogage et la maintenance d'un programme

Son utilisation se décompose en trois phases :

- Définition de la fonction
- Déclaration de la fonction
- Appel de la fonction

Déclarer une fonction

Appeler une fonction

Règles de visibilité des variables

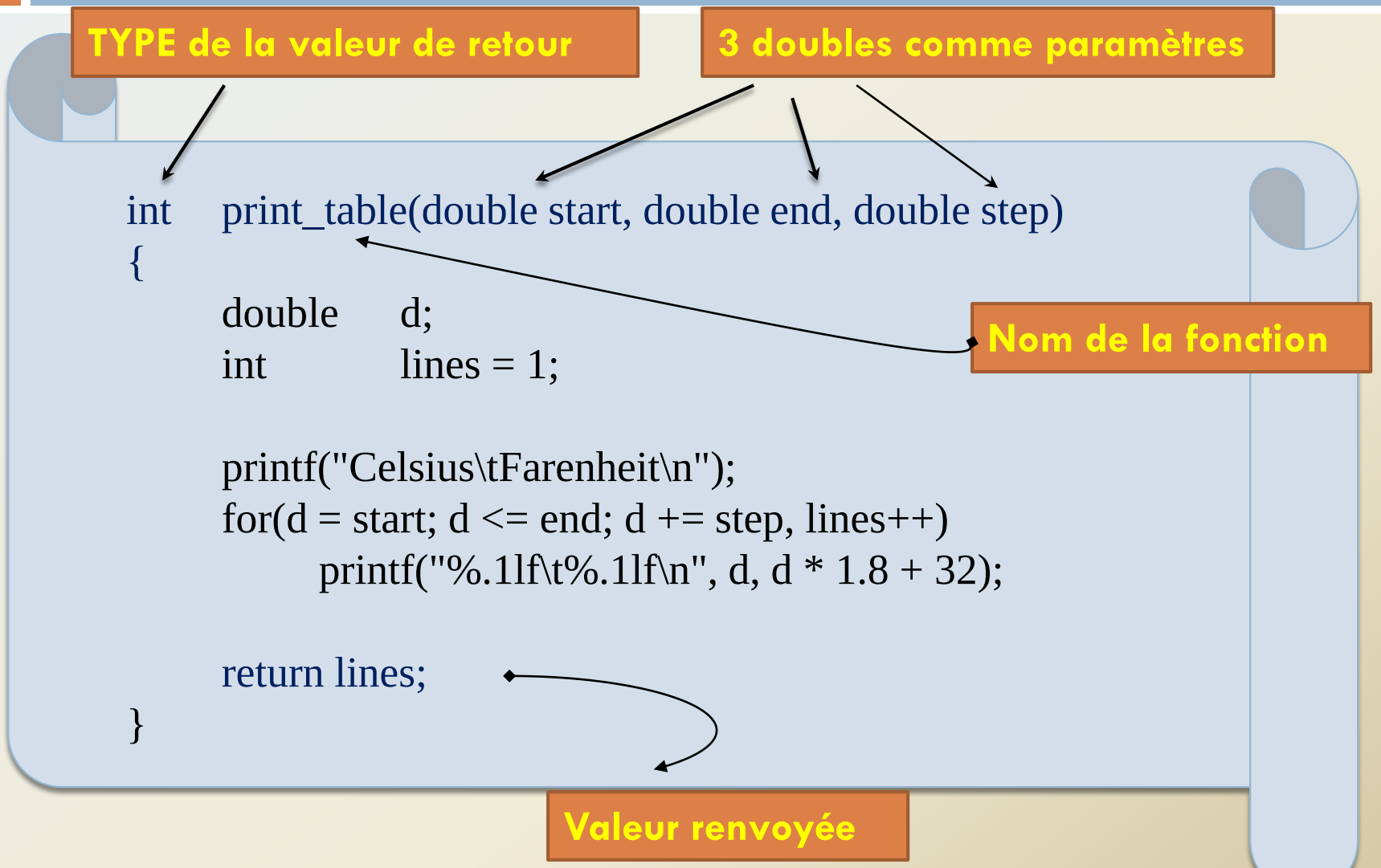
Passage des paramètres par valeur

Fonction renvoyant une valeur au programme

Passage des paramètres par valeur et par adresse

Passage des tableaux aux fonctions

Déclarer une fonction



Définition d'une fonction

- La fonction se termine par l'instruction *return* :
  - *return(expression); expression* du type de la fonction
  - *return;* fonction sans type

- Exemples :

```
int produit (int a, int b)
{
    return(a * b);
}
```

```
float calculer(float float1, float float2)
{
  return (  (float1 + float2 ) / 2) ;
}
```

Appel d'une fonction

IMPORTANT: cette instruction spécifie comment la fonction est définie

```
#include <stdio.h>

int     print_table(double, double, double);

void main(void)
{
        int             combien;
        double      end = 100.0;

        combien = print_table(1.0,end,3);
        print_table(end, 200, 15);
}
```

Le compilateur attend des doubles; les conversions sont automatiques

Ici, on ne tient pas compte de la valeur de retour

```
float calculFloat(float, float);
int addition();

void main(void)
{
  int Val ;

  Val = addition();
  printf("val = %d", Val);
}

int addition()
{
   float tmp;
   tmp = calcule(2.5,3) + calcule(5,7.2);
   return (int)tmp;
}

float calculFloat(float float1, float float2)
{
  return (  (float1 + float2 ) / 2) ;
}
```

Règles de visibilité des variables

Le C est un language structuré en blocs { } , les variables ne peuvent être utilisées que là où elles sont déclarées

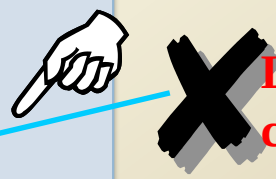
```
void    func(int x);
int glo=0;                  // variable globale
void    main(void)
{
        int       i = 5, j, k = 2;  //lacales à main
        float     f = 2.8, g;

        d = 3.7;
        func (i);
}

void    func(int v)
{
        double d, e = 0.0, f =v; //lacales à func

        i++; g--;glo++;
        f = 0.0;
}
```

Le compilateur ne connaît pas d



Le compilateur ne connaît pas i et g

C'est le f local qui est utilisé

Passage des paramètres par valeur

- Quand une fonction est appelée, ses paramètres sont copiés

  (passage par valeurs)

- La fonction travaille donc sur des copies des paramètres et ne peut donc les modifier

- En C, le passage des paramètres par référence se fait en utilisant des pointeurs (voir plus loin, dernier chapitre)

- scanf() travaille avec pointeur. C'est le pourquoi du &

Passage des paramètres par valeur

```
#include <stdio.h>

void   change(int v);

void   main(void)
{
        int var = 5;

        change(var);

        printf("main: var = %d\n", var);
}

void   change(int v)
{
        v *= 100;
        printf("change: v = %d\n", v);
}
```

```
change: v = 500
main: var = 5
```

Fonction renvoyant une valeur au programme

```
#include <stdio.h>

int change(int v);

void   main(void){
        int var = 5;
        int valeur;
        valeur = change(var);

        printf("main: var = %d\n", var);
        printf("main: valeur = %d\n", valeur);
}

int change(int v)
{
        v *= 100;
        printf("change: v = %d\n", v);
        return (v+1);
}
```

```
change: v =
main: var =
main: valeur =
```

Fonction renvoyant une valeur au programme (cont.)

Une fonction se termine et « rend la main » au code appelant lorsque son exécution rencontre l'instruction : return expression; ou return;

```
#include <stdio.h>

int return_Val(int v);

void  main(void){
        int var = 5;
        int valeur;
        valeur = return_Val(var);

        printf("main: var = %d\n", var);
        printf("main: valeur = %d\n", valeur);
}

int return_Val(int v)
{
        if (v == 10)        return (2*v);
       else                 return (3*v);
}
```

return_Val : rien
main: var =
main: valeur =

```
int max (int a, int b) {
        if (a > b)        return a ;
        else return b ;
}
int min (int a, int b) {
        if (a < b)        return a ;
        else return b ;
}
int difference (int a, int b, int c) {
      return (max (a, max(b,c)) - min (a, min (b, c))) ;
}
void main ()
{
   printf ("%d", difference (10, 4, 7)) ;
}
```

Les fonctions récursives

Une fonction est dite **récursive,** dès lors ou elle se fait appel pour calculer une sous partie de la valeur finale qu'elle retournera.

```
#include <stdio.h>
int fact(int n);
int return_Val(int v);

void   main(void){
        int var = 5,   int valeur;
        valeur = return_Val(var);
        printf("main: var = %d\n", var);
        printf("main: valeur = %d\n", valeur);
}

int fact(int n) {
  return  (n<2?1:((int)n*fact(n-1));
}
int return_Val(int v)
{
        if (v == 10)        return (2*v);
       else                 return fact(v);
}
```

```
return_Val : rien
main: var =
main: valeur =
```

(0)=0*fact(-1) ✗

```
int temp=1;
while(n>1) temp*=n--;
```

Passer des tableaux aux fonctions

- Les tableaux peuvent être passés comme paramètres d'une fonction.
- Ils ne peuvent pas être retournés comme résultat d'une fonction.
- La longueur du tableau ne doit pas être définie à la déclaration de la fonction.
- Un tableau peut être modifié dans une fonction. Il est passé par référence (adresse) et non par valeur.

Passer des tableaux aux fonctions

```
#include <stdio.h>

void    somme(int x[], int n);

void    main(void){
        int i;
        int p[6] = { 1, 2,3, 5, 7, 11 };
        somme(p, 6);
        for (i=0;i<6;i++)
                printf("%d ", p[i]);
}

void    somme(int a[], int n){
        int     i;
        for(i = n-1; i >0 ; i--)
                a[i] += a[i-1];
}
```

p avant somme	p après somme
1	1
2	3
3	5
5	8
7	12
11	18

Passer des tableaux aux fonctions - Exemple

Écrire un programme en C qui range au maximum (taille Nmax) 20 nombres entiers saisis au clavier dans un tableau. Il doit gèrer en boucle le menu de choix suivant :

A- Saisie et affichage
B- Moyenne
C- Suppression du Max et affichage
D- Suppression du Min et affichage
E- Ajout d'un entier à une position donnée
Q- Quitter

Le point A sera traité par deux fonctions :

Saisie : la saisie au clavier des entiers et leur rangement dans le tableau.
Dans cette fonction on demandera le nombre d'éléments (**NE<= Nmax**) à saisire.

Affichage : affichage des données.

Le point B sera traité par une fonction :

Moyenne : fonction de calcul de la moyenne du tableau (avec affichage du résultat).

Le point C sera traité par trois fonctions :

Max_elem : une fonction qui retourne la position du maximum des valeurs du tableau.

Supprimer : supprime le Max du tableau.

Affichage : affichage des données.

Le point D sera traité par trois fonctions :

Min_elem : une fonction qui retourne la position du minimum des valeurs du tableau.

Supprimer : supprime le Mix du tableau.

Affichage : affichage des données.

Le point E sera traité par deux fonctions :

Ajout : c'est une fonction où on demande à l'utilisateur d'introduire l'entier et la position. Puis vous insérer l'entier dans le tableau à la position indiquée.

Affichage : affichage des données.

CH 3 : LES POINTEURS & ALLOCATION DYNAMIQUE

- La notion d'adresse
- Passage « par valeur » et « par adresse »
- Les pointeurs : Définition et déclaration
- Les opérateurs Liés aux pointeurs (& ET *)
- Arithmétique des pointeurs
- Pointeur et Tableau (1D & 2D)
- Tableau de Pointeurs
- Allocation Dynamique de mémoire (malloc, calloc, realloc & free)

La notion d'adresse

Retour sur les variables :

Une variable en C a 4 caractéristiques :

- un type
- un nom
- une valeur
- une adresse

les 3 premières déjà bien exploitées. Mais la 4ème? ?

La notion d’adresse

Schématisation par une boîte ou cellule :

- L'ordinateur doit stocker cette variable quelque part : dans la RAM

- Les octets (ou cellules) de la RAM sont numérotés pour que la machine s'y retrouve : chaque cellule à une adresse, et peut contenir une valeur (son contenu)

Attention à la distinction adresse/contenu

La notion d'adresse

- La mémoire de l'ordinateur peut être vu comme une série de cases.
- Ces cases sont repérées par des adresses.
- Dans ces cases sont stockées les valeurs des variables (ou des instructions).

Adresses	valeurs
65775420	123
65775416	3.2132
65775415	'M'
65775414	'i'
65775413	'a'
65775412	'm'

Pour accéder à la valeur d'une variable
(i.e au contenu de la case mémoire)

⇨ **Il faut connaître son emplacement**
(c'est l'adresse de la variable)

Lorsqu'on utilise une variable:

Le compilateur manipule son adresse pour y accéder.

Cette adresse est alors inconnue du programmeur

Passage « par valeur » et « par adresse »

Le problème

Écrire une fonction qui échange les valeurs de deux variables a et b de type int.

Solution naïve :

```
#include <stdio.h>


void  echange( int a , int b )
{
        int temp ;
        temp = b ; b = a ; a = temp ;

}

int main( )
{
int a = 1 , b = 2 ;
echange( a, b ) ;
printf("a: %d et b: %d\n", a, b);
return 0 ;
}
```

Que valent a et b ?

Que valent a et b ?

Passage « par valeur » et « par adresse »

Le problème

Ceci ne fonctionne pas car **a et b (du main) et a et b (de echange) sont des variables différentes** bien qu'étant des homonymes !

En fait, c'est comme si on avait écrit le programme suivant :

```
#include <stdio.h>

void  echange( int n , int m )
{
        int temp ;
        temp = n ; n = m ; m = temp ;
}

int main( )
{
int a = 1 , b = 2 ;
echange( b , a ) ;
printf("a: %d et b: %d\n", a, b);
return 0 ;
}
```

Passage « par valeur » et « par adresse »

Le problème

Reprenons le déroulement du programme :

1) On définit a et b dans le main ; on leur affecte les valeurs 1 et 2.

2) On appelle la fonction echange avec les VALEURS de a et b.

3) Les paramètres m et n reçoivent les valeurs 1 et 2.

Passage par valeur.

4) La fonction echange échange les valeurs de m et n :

➔ les paramètres m et n locaux à la fonction valent 2 et 1.

5) On revient dans le main … a et b valent toujours 1 et 2 !!!

Passage « par valeur » et « par adresse »

La solution

Les pointeurs viennent à notre secours :

```
#include <stdio.h>

void  echange( int * p_a , int * p_b )
{
      int temp ;
      temp = *p_b ; *p_b = *p_a ; *p_a = temp ;
}

int main( )
{
int a = 1 , b = 2 ;
echange( &a , &b ) ;
printf("a: %d et b: %d\n", a, b);
return 0 ;
}
```

Passage « par valeur » et « par adresse »

La solution

Reprenons le déroulement du programme :

1) on définit a et b dans le main ; on leur affecte les valeurs 1 et 2.

2) on appelle la fonction echange avec les ADRESSES de a et b.

3) p_a et p_b reçoivent les ADRESSES de a et b.

Passage par adresse.

4) la fonction echange échange les valeurs des cases mémoire pointées par p_a et p_b.

5) on revient dans le main … a et b valent maintenant 2 et 1 !

Passage « par valeur » et « par adresse »

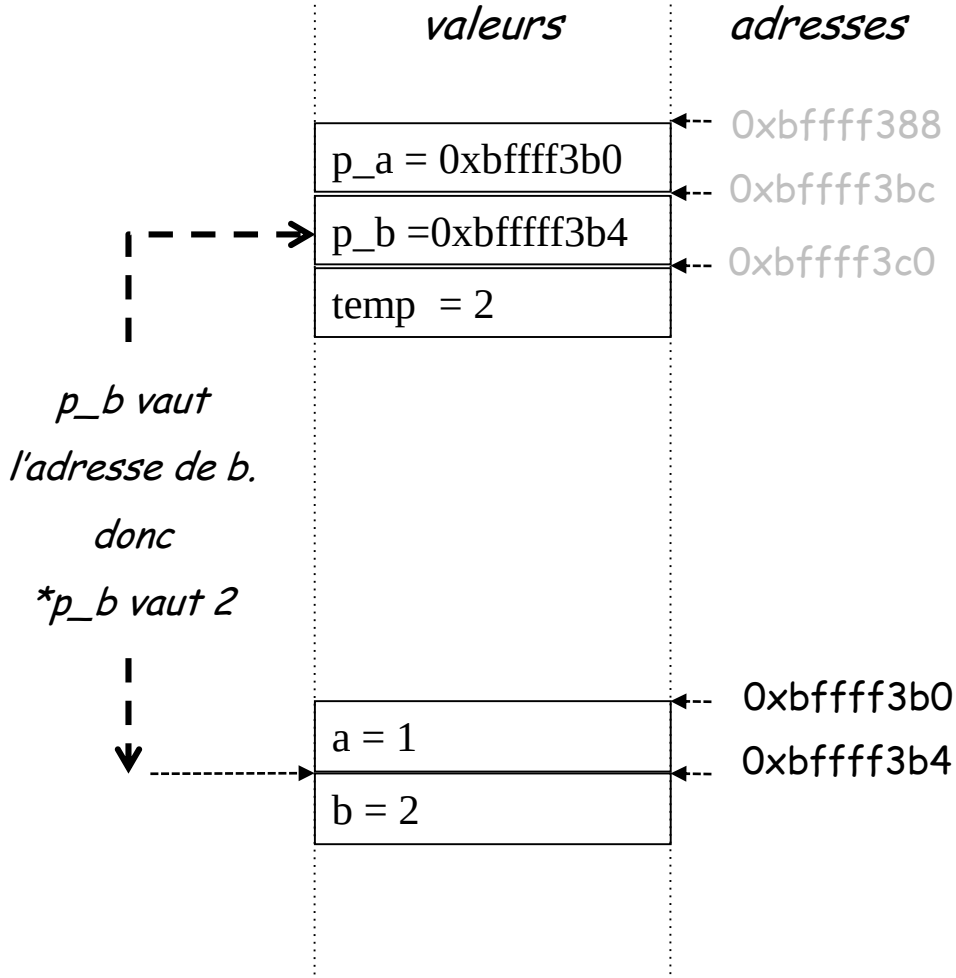
Exemple

Appel de la fonction : copie des valeurs des paramètres effectifs dans de nouvelles variables

```
void  echange( int * p_a , int * p_b )
{
   int temp ;
   temp = *p_b ;
   *p_b = *p_a ;
   *p_a = temp ;
}

int main( )
{
int a = 1 , b = 2 ;
echange( &a , &b ) ;

return 0 ;
}
```



Passage « par valeur » et « par adresse »

Exemple

Appel de la fonction : copie des valeurs des paramètres effectifs dans de nouvelles variables

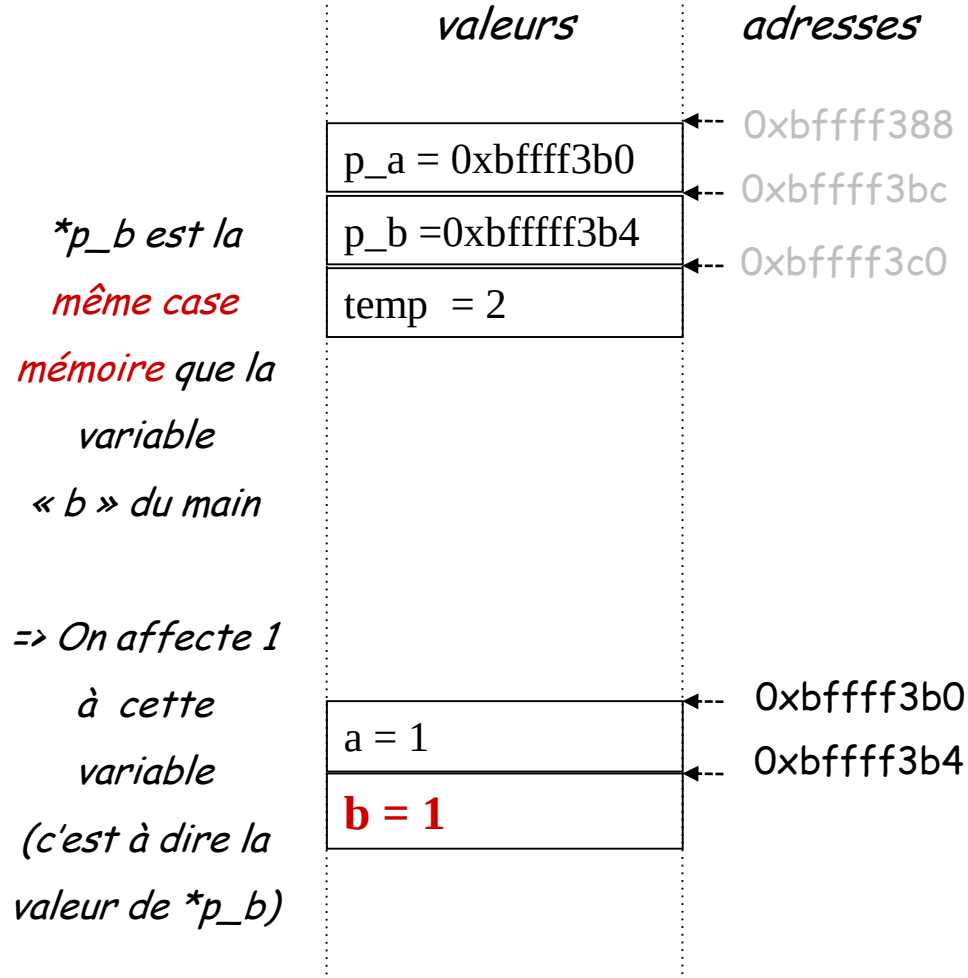
```
void  echange( int * p_a , int * p_b )
{
   int temp ;
   temp = *p_b ;
   *p_b = *p_a ;
   *p_a = temp ;
}

int main( )
{
int a = 1 , b = 2 ;
echange( &a , &b ) ;

return 0 ;
}
```

**p_b est la même case mémoire que la variable « b » du main*

*=> On affecte 1 à cette variable (c'est à dire la valeur de *p_b)*



Passage « par valeur » et « par adresse »

Une variable "a" définie dans une fonction n'est utilisable que dans cette fonction. On dit que a est locale à la fonction.

Une variable "a" définie dans une autre fonction est une variable homonyme mais différente de la première ; elle occupera une autre case mémoire lors de l'appel de la fonction.

Pour modifier la valeur d'une variable lorsqu'on appelle une fonction, il faut utiliser le mécanisme du passage "par adresse" avec des pointeurs …

Faire attention à la manipulation des & et des * !!!

Remarque : les termes "passage par valeur" et "passage par adresse" sont en fait des abus de langage… En C, on passe toujours "par valeur", valeurs qui valent parfois "des adresses"..

Les pointeurs : Définition et déclaration

- Un *pointeur est un objet (Lvalue) dont la valeur est égale à l'adresse d'un autre objet.*

- On déclare un pointeur par l'instruction :

  *type *nom-du-pointeur;*

  - *type est le type de l'objet pointé.*
  - *nom-du-pointeur est un identificateur d'adresse (la valeur est un entier long).*

Les pointeurs : Définition et déclaration

- Déclaration : *****

```
type *idPtr;
```

Nom de la variable

Ce petit signe indique que l'on ne travaille plus sur une variable normale, mais sur un pointeur

Le type peut être un type simple (**int, float**…) ou un objet (n'importe quelle classe).

Les pointeurs : Opérateur d'adresse &

```
printf("%d",&idVar);
```

Nom d'une variable de n'importe quel type

Ce petit signe indique que l'on veux récupérer l'adresse d'une variable

Plutôt que d'afficher la valeur de la variable, on affiche l'adresse de la case mémoire

Les pointeurs : Opérateur de déréférencement *

- Opérateur de déréférencement : *
  - `Printf("%d", *ptr);`

Nom d'un pointeur

Ce petit signe indique que l'on veux récupérer la valeur située à une adresse précise

Plutôt que d'afficher l'adresse du pointeur, on affiche la valeur qui est dans la case mémoire

Les pointeurs : Opérateur de déréférencement

- Exemple :

```
char c = 'a';
char *p;
p=&c;
Printf("%d", *p);
```

On déclare une variable normale de type caractère

On déclare un pointeur sur une case de type caractère

On donne l'adresse de la variable **c** au pointeur **p**

On affiche la valeur de la case pointée par **p** ('a')

Attention !

- Il faut s'assurer que les pointeurs que l'on manipule sont bien initialisé!
  - Il doivent contenir l'adresse d'une variable valide.
  - Accéder à la variable d'un pointeur non initialisé revient à:
    - Ecrire ou lire, dans la mémoire à un endroit aléatoire.
    - Plantage à l'exécution du programme.

```
int * Var; // pour le compilateur Var pointe sur quelque chose (n'importe quoi)

int * Var = NULL; // le compilateur sait que Var ne pointe sur rien
```

Les pointeurs – Exemples (1)

int i = 3;

*int *p;*

p = &i;

On se trouve dans la configuration objet adresse valeur

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000

Les pointeurs – Exemples (2)

L'*opérateur unaire d'indirection* * *permet d'accéder directement à la valeur de l'objet pointé:*

```
main()
{
    int i = 3;
    int *p;
    p = &i;
    printf("*p = %d \n",*p);
}
```

objet	adresse	valeur
i	4831836000	3
p		
*p		

Les pointeurs – Exemples (3)

- p et *p sont deux manipulations sont très différentes:

```
main()
{
    int i = 3, j = 6;
    int *p1, *p2;
    p1 = &i;
    p2 = &j;
    *p1 = *p2;
}
```

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1		
p2		

Les pointeurs – Exemples (3)

Après l'affectation *p1 = *p2 :

objet	adresse	valeur
i		
j		
p1		
p2		

Les pointeurs – Exemples (4)

- Exemple 2:

```
main()
{
int i = 3, j = 6;
int *p1, *p2;
p1 = &i;
p2 = &j;
p1 = p2;
}
```

objet	adresse	valeur
i	4831836000	3
j	4831836004	6
p1		
p2		


SVP monsieur, j'en veux plus

Arithmétique des pointeurs

La valeur d'un pointeur étant un entier, on peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques. Les seules opérations arithmétiques valides sur les pointeurs sont :

- L'addition d'un entier à un pointeur: Le résultat est un pointeur de même type que le pointeur de départ ;

- La soustraction d'un entier à un pointeur: Le résultat est un pointeur de même type que le pointeur de départ ;

- La différence de deux pointeurs pointant tous deux vers des objets de même type: Le résultat est un entier.

Arithmétique des pointeurs

Si i est un entier et p est un pointeur sur un objet de type ***type****, l'expression p + i désigne un pointeur sur un objet de type* ***type*** *dont la valeur est égale à la valeur de p incrémentée de i * sizeof(type). Il en va de même pour la soustraction d'un entier à un pointeur, et pour les opérateurs d'incrémentation et de décrémentation ++ et –*

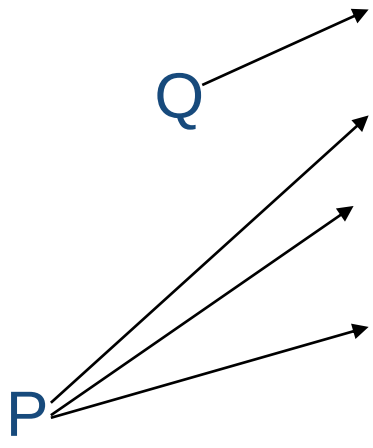
```
main()
{
double i = 3;
double *p1, *p2;
p1 = &i;
p2 = p1 + 1;
printf("p1 = %ld \t p2 = %ld\n",p1,p2);
}
```

```
p1 = 4831835984
p2 = 4831835992
```

Arithmétique des pointeurs - Exemple

- Les opérateur ++, --, +, = ….. Sont définis pour les pointeurs, mais attention, ils s'appliquent aux adresses.

```
char * P=NULL;
….  //allocation
P ++ ;
P = P – 2 ;
char *Q = P + 3;
```



65775417	123
65775416	12
65775415	156
65775414	3
65775413	9
65775412	456

Pointeur et tableau

- Nom du tableau : adresse du tableau

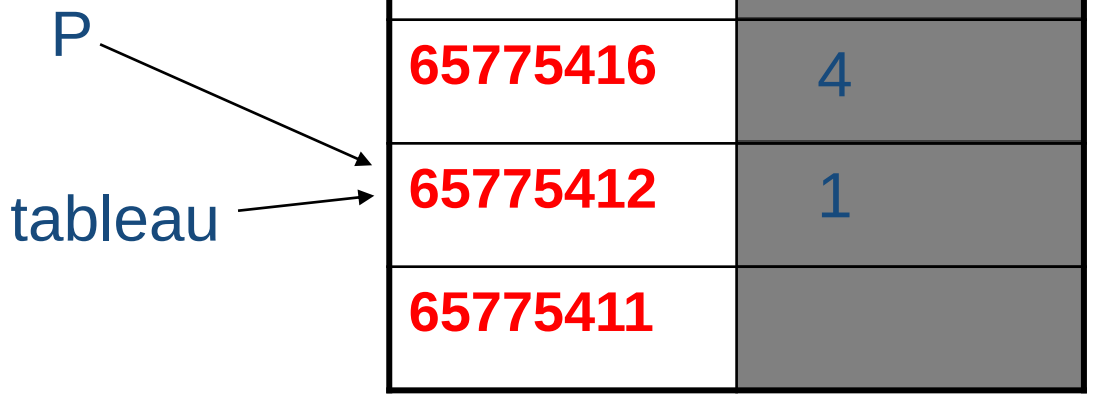
- Accès à un élément t[i] :
  - on part de l'adresse de début du tableau, on ajoute i et on obtient l'adresse du ième élément.
  - L'élément est obtenu par l'opérateur d'indirection *

- Conséquences
  - L'élément t[i] s 'écrit aussi *(t+i)
  - L'adresse de t[i] s'écrit &t[i] ou bien (t+i)

Pointeur et tableau

- Un tableau est en fait un pointeur!

```
int tableau[3];
int * P = tableau ;
Tableau [2] = 5 ;
*(tableau+1) = 4;
P[0]=1;
```

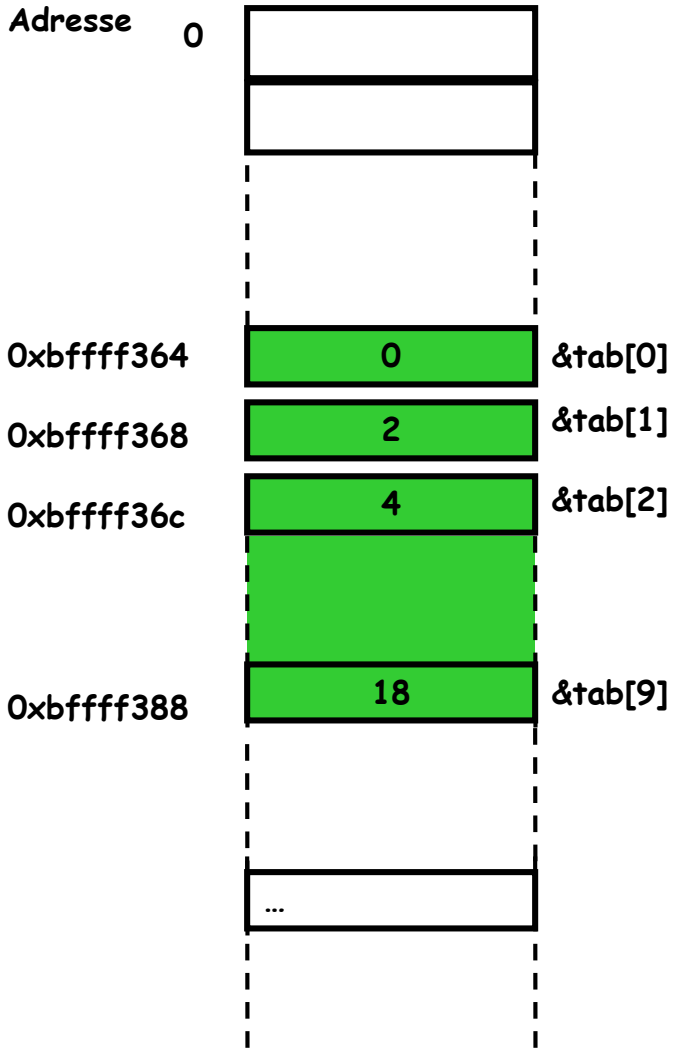


Pointeur et tableau

```
main() { int tab[10]; int i;
  for (i=0; i<10; i++) tab[i]= 2*i;
  puts("Voici l'element d'indice 2 : ");
  printf("%d %d",tab[2],*(tab+2));
  puts(""); puts("Voici les adresses : ");
  for (i=0; i<5; i++)
     printf("%p %p",tab+i, &tab[i]);
}
```

2 manières différentes d'ecrire l'adresse de t[i]

```
Terminal — bash — 59×10
macbook-pro-de-administrateur:exemples desvignes$ gcc p3.c
macbook-pro-de-administrateur:exemples desvignes$ ./a.out
Voici l'element d'indice 2 :
4 4
Voici les adresses :
0xbffff364 0xbffff364
0xbffff368 0xbffff368
0xbffff36c 0xbffff36c
0xbffff370 0xbffff370
0xbffff374 0xbffff374
```

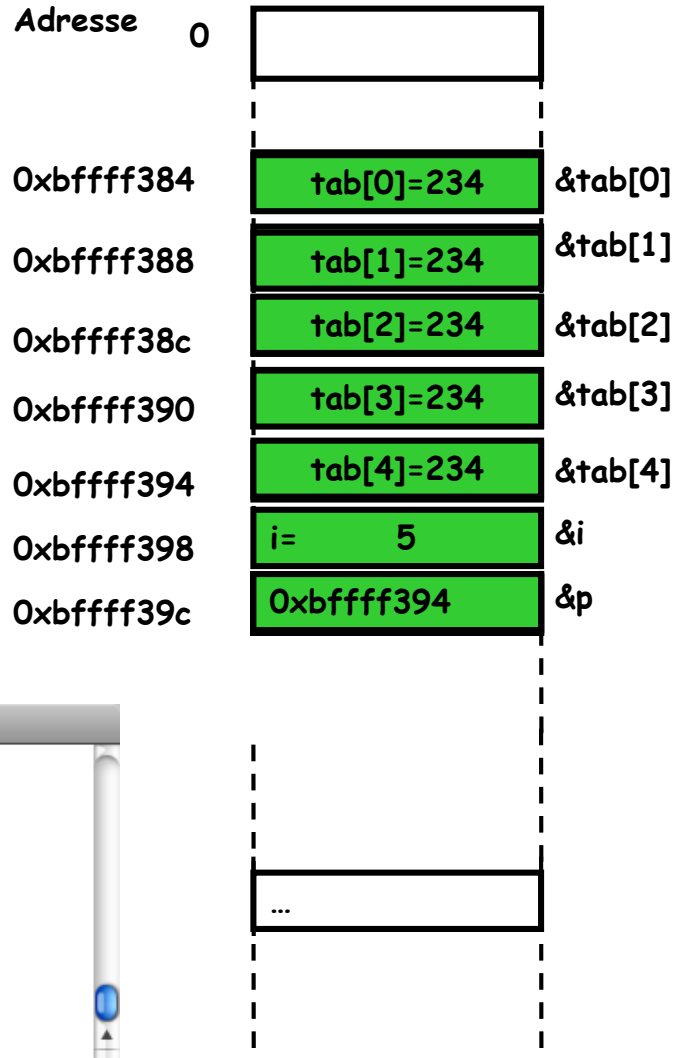


Pointeur et tableau – Exemple (1)

```
#define N 5
int tab[N] = {1, 2, 6, 0, 7};
main()
{
    int *p = NULL;
    printf("\n ordre croissant:\n");
    for (p = &tab[0]; p <= &tab[N-1]; p++)
        printf(" %d \n",*p);
    printf("\n ordre decroissant:\n");
    for (p = &tab[N-1]; p >= &tab[0]; p--)
        printf(" %d \n",*p);
}
```

Pointeur et tableau – Exemple (2)

```
main() {
    int* p=NULL; int i; int tab[5];
    for (i=0; i<5; i++) tab[i]=2*i+1;
    p=tab;
   while (p<tab+5) {
     printf(" Pointeur: %p ",,p);
     printf(" Valeur %d",*p);
     *p=234;
     printf("Valeur modifiee %d\n",*p);
     p++;
   }
```




```
Terminal — bash — 73×9
macbook-pro-de-administrateur:exemples desvignes$ ./a.out
Adresse du tableau:0xbffff384, adresse de i=0xbffff398
adresse de p=0xbffff39c
Boucle 0 : Pointeur : 0xbffff384 Valeur avant 1  Valeur modifiée 234
Boucle 1 : Pointeur : 0xbffff388 Valeur avant 3  Valeur modifiée 234
Boucle 2 : Pointeur : 0xbffff38c Valeur avant 5  Valeur modifiée 234
Boucle 3 : Pointeur : 0xbffff390 Valeur avant 7  Valeur modifiée 234
Boucle 4 : Pointeur : 0xbffff394 Valeur avant 9  Valeur modifiée 234
macbook-pro-de-administrateur:exemples desvignes$
```

Pointeur et tableau à deux dimensions

- Un tableau M à deux dimensions peut être définis:

```
int M[4][10] = { { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                 {10,11,12,13,14,15,16,17,18,19},
                 {20,21,22,23,24,25,26,27,28,29},
                 {30,31,32,33,34,35,36,37,38,39} };
```

- Le nom du tableau **M** représente l'adresse du premier élément du tableau et pointe sur le tableau M[0] qui a la valeur {0,1,2,3,4,5,6,7,8,9}. **M[0] pointe sur la rangée 0.**

- L'expression **(M+1)** est l'adresse du deuxième élément du tableau M et pointe sur M[1] qui a la valeur {10,11,12,13,14,15,16,17,18,19}.

M[1] pointe sur la rangée 1.

Pointeur et tableau à deux dimensions (2)

- L'arithmétique des pointeurs qui respecte automatiquement les dimensions des éléments conclut logiquement que **M+i désigne l'adresse du tableau M[i]**.

- Comment pouvons-nous accéder à l'aide de pointeurs aux éléments de chaque composante du tableau, c-à-d: aux éléments M[0][0], M[0][1], ... , M[3][9] ?

- Une solution consiste à convertir la valeur de M (qui est un pointeur sur *un tableau du type* ***int***) en un pointeur sur ***int***. On pourrait se contenter de procéder ainsi:

```
int *P; P = M; /* conversion automatique */
```

Pointeur et tableau à deux dimensions (3)

- Cette dernière affectation entraîne une conversion automatique de l'adresse &M[0] dans l'adresse &M[0][0].

- La solution la plus correcte:

```
int *P, B;
P = (int *)M;        /* conversion forcée */
B = *(P+(2*10) + 2); // B = M[2][2]
```

- Sachant que M est stockée ligne par ligne, il est maintenant possible traiter M à l'aide du pointeur P comme un tableau unidimensionnel de dimension 4*10.

Pointeur et tableau à deux dimensions (4)

- Exemple pointeur VS tableau 2D ($\sum$ des éléments de M)

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
                {10,11,12,13,14,15,16,17,18,19},
                {20,21,22,23,24,25,26,27,28,29},
                {30,31,32,33,34,35,36,37,38,39}};
```

Solution ordinaire:

```
Int i;
for(int i=0;i<4;i++)
    for(int j=0;j<10;j++)
            som += M[i][j];
```

Pointeur et tableau à deux dimensions (5)

Solution avec les pointeurs:

```
int *P, som=0;
P = (int *)M;
for(int i=0;i<4;i++)
        for(int j=0;j<10;j++)
                som += *(P+(i*10)+j)
```

Tableaux de pointeurs (1)

Syntaxe :

type * identificateur du tableau[nombre de composantes];

Exemple : int * A[10]; // un tableau de 10 pointeurs
// vers des valeurs de type int.

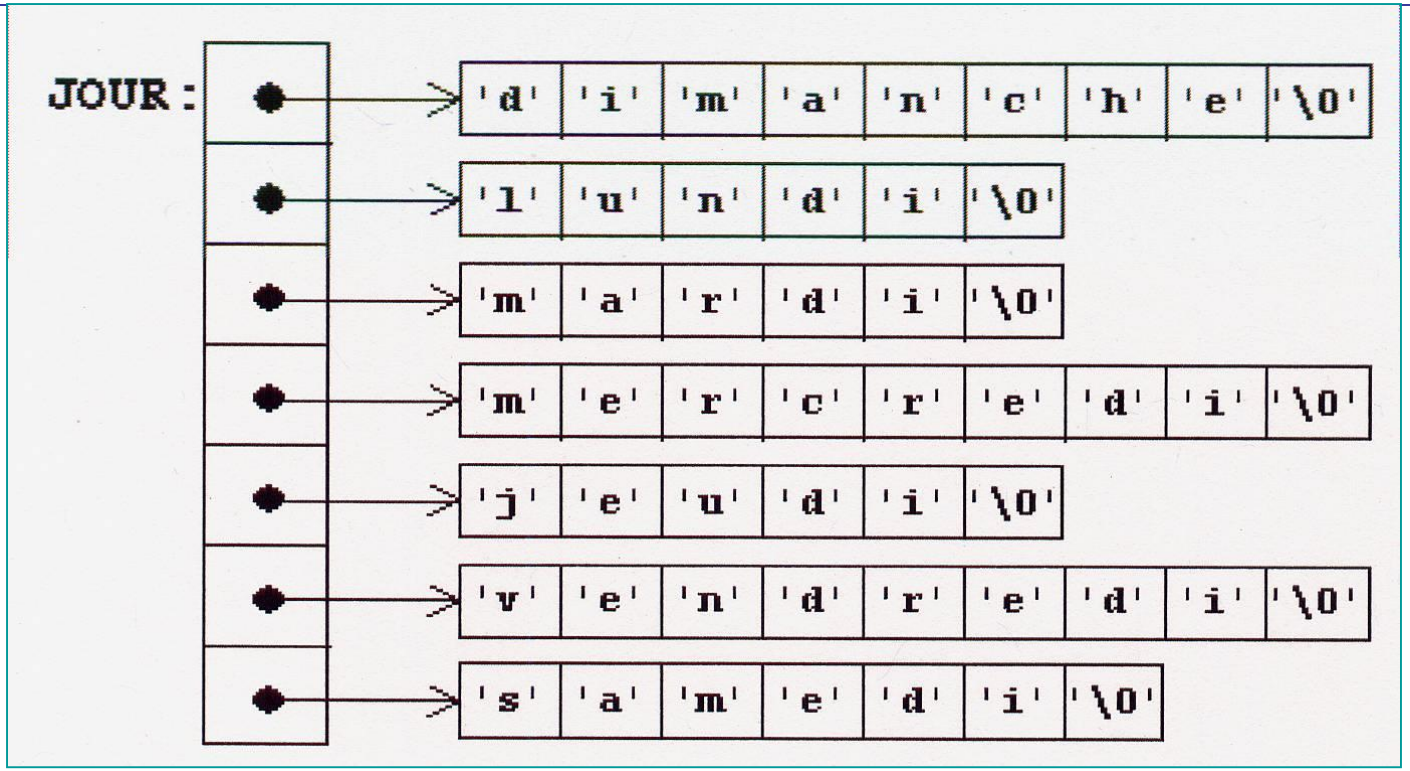
Tableaux de pointeurs vers des chaînes de caractères de différentes longueurs

Exemple

```
char *JOUR[] = {"dimanche", "lundi", "mardi",
                "mercredi", "jeudi", "vendredi",
                "samedi"};
```

Nous avons déclaré un tableau JOUR[] de 7 pointeurs de type char, chacun étant initialisé avec l'adresse de l'une des 7 chaînes de caractères.

Tableaux de pointeurs (2)



Affichage :

```
int I;
for (I=0; I<7; I++) printf("%s\n", JOUR[I]);
```

Pour afficher la 1e lettre de chaque jour de la semaine, on a :

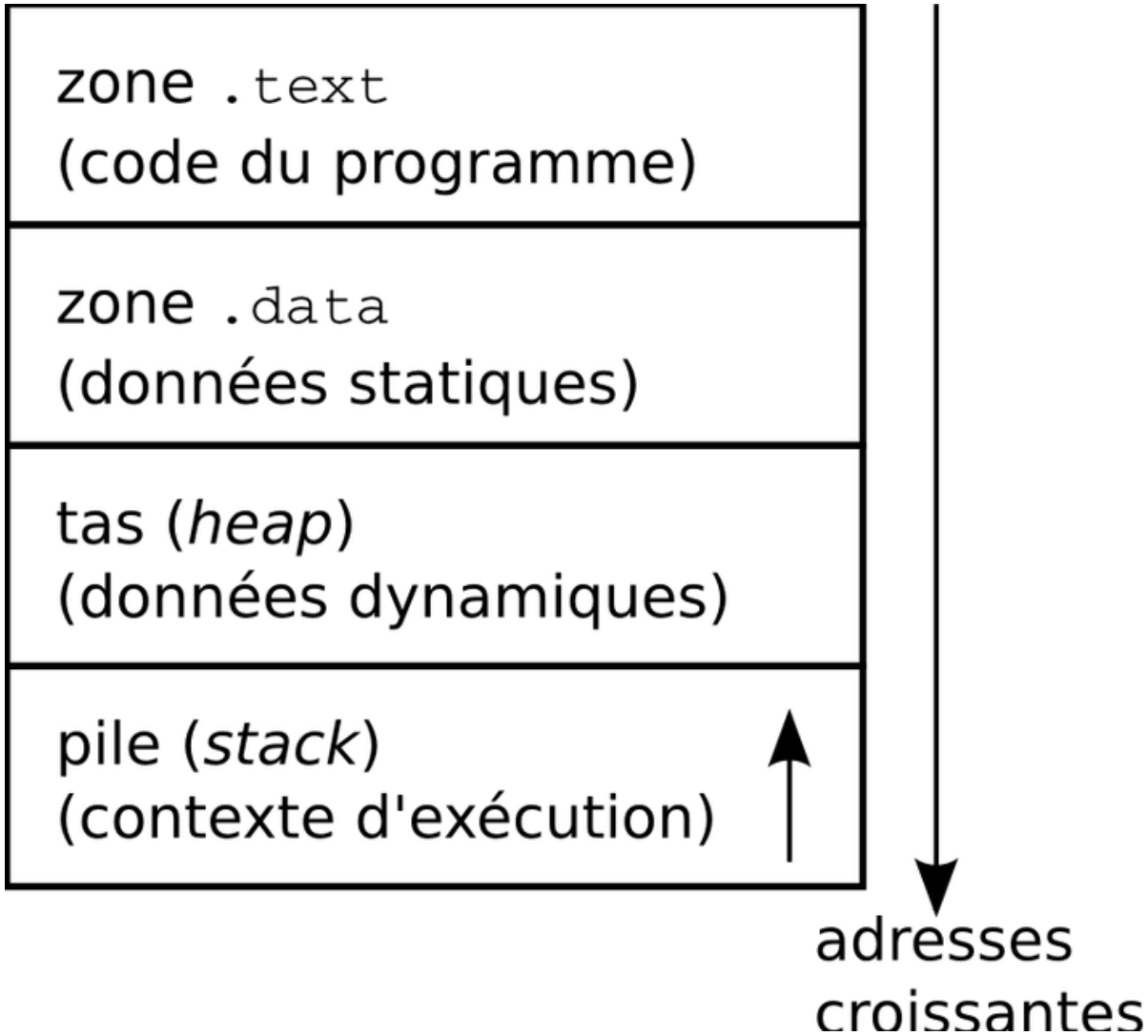
```
int I;
for (I=0; I<7; I++) printf("%c\n", *JOUR[I]);
```

Tableaux de pointeurs (3)

Si D[i] pointe dans un tableau,

`D[i]`	désigne l'adresse de la première composante
`D[i]+j`	désigne l'adresse de la j-ième composante
`*(D[i]+j)`	désigne le contenu de la j-ième composante

Comment la mémoire est gérée? (1)



Allocation statique de la mémoire (1)

- Jusqu'à maintenant, la déclaration d'une variable entraîne automatiquement la réservation de l'espace mémoire nécessaire.

- Le nombre d'octets nécessaires était connu au temps de compilation; le compilateur calcule cette valeur à partir du type de données de la variable.

Exemples d'allocation statique de la mémoire

```
float A, B, C;          /* réservation de 12  octets */
short D[10][20];        /* réservation de 400 octets */
char E[] = {"Bonjour !"};
                        /* réservation de 10  octets */
char F[][10] = {"un", "deux", "trois", "quatre"};
                        /* réservation de 40  octets */
```

Il en est de même des pointeurs.

```
double *G;        /* réservation de p    octets */
char *H;          /* réservation de p    octets */
float *I[10];     /* réservation de 10*p octets */
```

Allocation statique de la mémoire (2)

Il en est de même des chaînes de caractères constantes.

Exemples

```
char *J = "Bonjour !";
          /* réservation de   p+10      octets */
float *K[] = {"un", "deux", "trois", "quatre"};
          /* réservation de 4*p+3+5+6+7 octets */
```

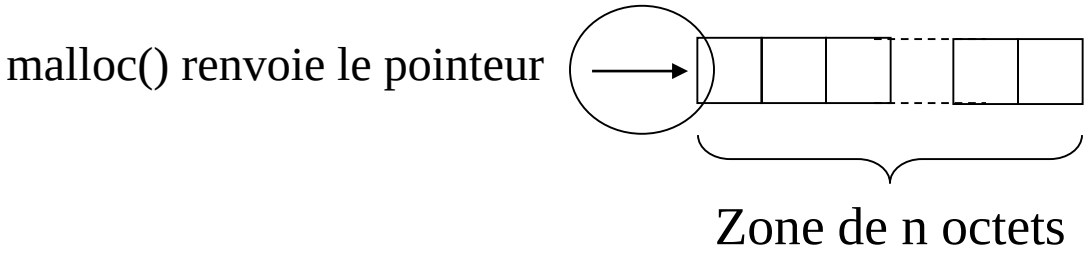
Allocation dynamique – malloc (1)

- Un tableau est alloué de manière statique : nombre d'éléments constant.
- Alloué lors de la compilation (avant exécution)
- Problème pour déterminer la taille optimale, donnée à l'exécution
- Surestimation et perte de place de plus, le tableau est un pointeur constant.
- **Il faudrait un système permettant d'allouer un nombre d'éléments connu seulement à l'exécution : c'est l'allocation dynamique.**

Allocation dynamique – malloc (2)

- Faire le lien entre le pointeur non initialisé (le panneau vide) et une zone de mémoire de la taille que l'on veut.
- On peut obtenir cette zone de mémoire par l'emploi de **malloc**, qui est une fonction prévue à cet effet.
- malloc est une fonction qui fait partie de ***stdlib.h***
- Il suffit de donner à **malloc** le nombre d'octets désirés (attention, utilisation probable de **sizeof**), et **malloc renvoie un pointeur de type *void** sur la zone de mémoire allouée**.
- Si **malloc** n'a pas pu trouver une telle zone mémoire, il renvoie NULL.
- Appel par : **malloc(nombre_d_octets_voulus)**;

Allocation dynamique – malloc (3)



• Pour accéder à cette zone, il faut impérativement l'affecter à un pointeur existant. On trouvera donc **toujours** malloc à droite d'un opérateur d'affectation.

• Il ne faut pas oublier de transtyper le résultat de **malloc()** qui est de type void* en le type du pointeur auquel on affecte le résultat.

Allocation dynamique – malloc (4)

Exemples d'utilisation:

allocation dynamique pour un pointeur vers des entiers, (analogue à un tableau d'entiers). On demandera à l'utilisateur le nombre d'éléments qu'il souhaite, puis on fait l'allocation dynamique correspondante : tableau de la taille requise, pas de perte de mémoire !

```
#include <stdio.h>
void main()
{
        int *pt_int = NULL;
        int nbElem;

        printf("combien d'elements dans le tableau ?:");
        scanf("%d",&nbElem);

        pt_int = (int *)malloc(nbElem*sizeof(int));
}
```

Allocation dynamique – malloc (5)

Détail de la dernière ligne : analyse de l'expression à droite de l'opérateur d'affectation :

• **(int *)** : transtypage : car malloc() donne un pointeur void *, et pt_int est de type int *

• **malloc** : appel à la fonction

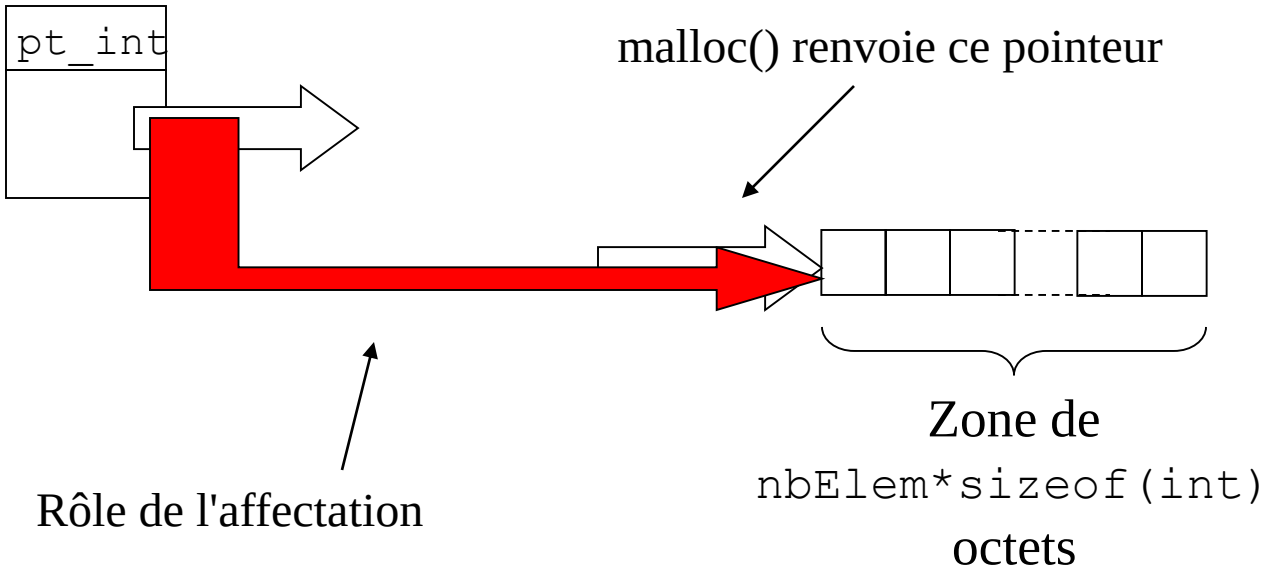
•**nbElem*sizeof(int)** : n'oublions pas que malloc reçoit un nombre d'octets a allouer ! Ici, on veut allouer nbElem élements, qui sont chacun de type int ! Or un int occupe plus d'un octet. Il occupe sizeof(int) octets !

• Donc le nombre total d'octets à demander est : **nombre d'éléments * taille de chaque élément en octets**.

Allocation dynamique – malloc (6)

pt_int = (int *)malloc(nbElem*sizeof(int));

avec la symbolisation déjà vue pour les pointeurs :



Allocation dynamique – malloc (7)

Vérifier le fonctionnement : si malloc() à donné NULL, l'allocation a échoué. Toujours tester la valeur du pointeur affecté après l'emploi de malloc().

```
if (pt_int == NULL)
{
        printf("allocation n'a pas fonctionne\");
}
else
{
        /* suite du programme */
}
```

on peut maintenant utiliser pt_int comme un tableau d'entiers, avec la notation [] !

Allocation dynamique – calloc

La fonction calloc : ***void *calloc(nombre,taille);***

- Elle alloue un bloc de mémoire de (***nombre x taille)*** octets
- La zone mémoire allouée par **calloc** est initialisée avec des 0.
- Elle renvoie un pointeur sur la zone de type ***void*** (valable pour tous les types) qu'il faut convertir en un type adapté aux données.
- Si l'allocation réussit, **calloc** renvoi un pointeur sur le bloc, elle échoue si taille = 0 ou s'il n'y a pas assez de place en mémoire. Dans ce cas elle retourne un ***pointeur nul : NULL***

Ex:

```
int *p;
p = (int *) calloc(10 , sizeof(int) );      // réservation pour 10 entiers
if ( p== NULL)                              // test création du pointeur
{ printf("erreur d'allocation mémoire !!!");
  // ici traitement de l'erreur …
}
```

Allocation dynamique – realloc (1)

La fonction realloc : ***void *realloc(pointeur , newtaille);***

- Elle permet de **changer la taille d'un bloc déjà alloué**. Elle gère l'aspect dynamique des pointeurs. Utilisable à tous moment dans le programme

- Elle renvoie un pointeur sur la zone de type void (valable pour tous les types) qu'il faut convertir en un type adapté aux données.

- Si l'allocation réussit, **calloc** renvoi un pointeur sur le bloc, elle échoue si taille = 0 ou s'il n'y a pas assez de place en mémoire. Dans ce cas elle retourne un pointeur nul : NULL

Allocation dynamique – realloc (2)

Exemple d'utilisation:

```
int *p;
…
p = (int *) realloc( p , 20 * sizeof(int) );    // réservation pour 10 entiers supplémentaires
if ( p== NULL)    // test création du pointeur
{
        printf("erreur d'allocation mémoire !!!");
        // ici traitement de l'erreur …
}
```

!!! ATTENTION, les données peuvent être déplacées si l'espace n'est pas suffisant !!!

Libération de la mémoire - free (1)

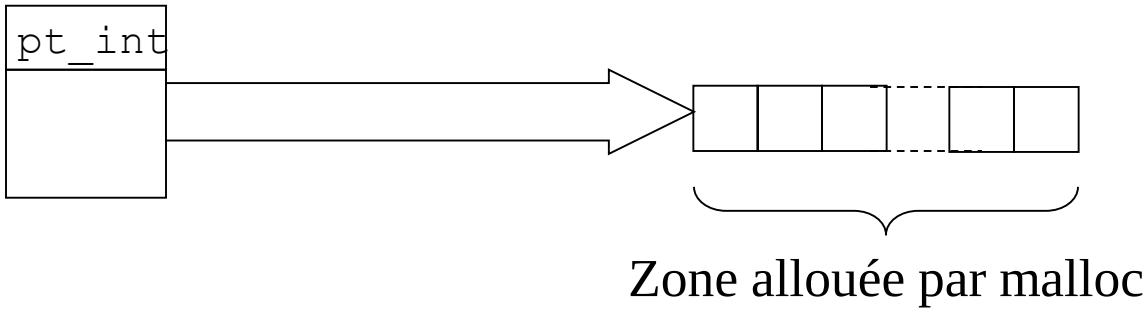
• Lorsque la mémoire allouée dynamiquement n'est plus utile (le plus souvent, à la fin d'un programme, il est nécessaire de la libérer : la rendre disponible pour le système d'exploitation.

•Fonction ***free*** qui **réalise le contraire** de ***malloc()***.

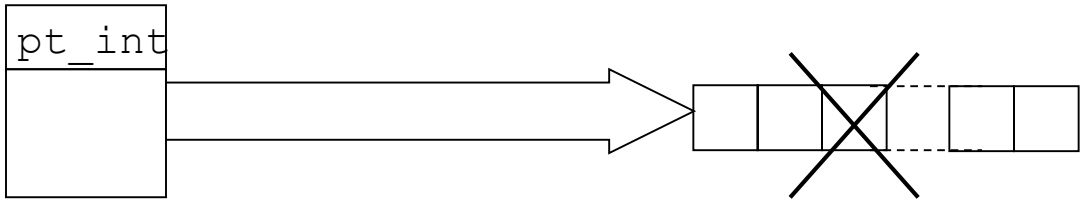
free(pointeur_vers_la_zone_allouée);

• **On ne peut libérer que des zones allouées dynamiquement** : pas de free avec un tableau statique, même si le compilateur l'accepte.

Libération de la mémoire - free (2)

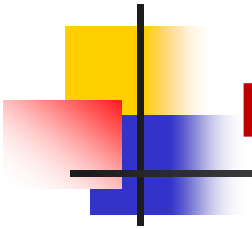


Si on écrit : ***free(pt_int);***



pt_int pointe toujours au même endroit de la mémoire, mais on ne peut plus l'utiliser.

À chaque malloc() doit correspondre un free() dans un programme !



Résumé & Illustration

Fichier <**stdlib.h**>	
`void *`**`malloc`**`(size_t)`	allocation d'un bloc
`void *`**`calloc`**`(size_t, size_t)`	allocation & initialisation d'un bloc
`void *`**`realloc`**`(void *, size_t)`	modification de la taille d'un bloc
`void` **`free`**`(void *)`	libération d'un bloc

Résumé & Illustration

```
#include <stdlib.h>

{
   void *ptr;

   ptr = malloc(5);
   ptr = realloc(ptr, sizeof(int));
   free(ptr);

   ptr = calloc(2, sizeof(short int));
   free(ptr);
}
```

Adresse	Nom	Contenu
0123456	ptr	?
0123457		
0123458		
0123459		

Résumé & Illustration

```
#include <stdlib.h>

{
   void *ptr;

   ptr = malloc(5);
   ptr = realloc(ptr, sizeof(int));
   free(ptr);

   ptr = calloc(2, sizeof(short int));
   free(ptr);
}
```

Adresse	Nom	Contenu
0123456	ptr	0065662
0123457		
0123458		
0123459		

Adresse	Contenu
0065662	?
0065663	?
0065664	?
0065665	?
0065666	?
0065667	?

Résumé & Illustration

```
#include <stdlib.h>

{
   void *ptr;

   ptr = malloc(5);
   ptr = realloc(ptr, sizeof(int));
   free(ptr);

   ptr = calloc(2, sizeof(short int));
   free(ptr);
}
```

Adresse	Nom	Contenu
0123456	ptr	0065662
0123457		
0123458		
0123459		

Adresse	Cont.
0065662	?
0065663	?
0065664	?
0065665	?
0065666	?
0065667	?

Résumé & Illustration

```
#include <stdlib.h>

{
   void *ptr;

   ptr = malloc(5);
   ptr = realloc(ptr, sizeof(int));
   free(ptr);

   ptr = calloc(2, sizeof(short int));
   free(ptr);
}
```

Adresse	Nom	Contenu
0123456	ptr	0065662
0123457		
0123458		
0123459		

Adresse	Cont.
0065662	?
0065663	?
0065664	?
0065665	?
0065666	?
0065667	?

Résumé & Illustration

```
#include <stdlib.h>

{
   void *ptr;

   ptr = malloc(5);
   ptr = realloc(ptr, sizeof(int));
   free(ptr);

   ptr = calloc(2, sizeof(short int));
   free(ptr);
}
```

Adresse	Nom	Contenu
0123456	ptr	0065790
0123457		
0123458		
0123459		

Adresse	Contenu
0065790	0
0065791	
0065792	0
0065793	
0065794	?
0065795	?

Résumé & Illustration

```
#include <stdlib.h>

{
   void *ptr;

   ptr = malloc(5);
   ptr = realloc(ptr, sizeof(int));
   free(ptr);

   ptr = calloc(2, sizeof(short int));
   free(ptr);
}
```

Adresse	Nom	Contenu
0123456	ptr	0065790
0123457		
0123458		
0123459		

Adresse	Contenu
0065790	0
0065791	0
0065792	0
0065793	0
0065794	?
0065795	?

Put it all together - Exemple

```
/*
void *realloc(current_storage_pointer,new_size);

*/

int main(){
    char *str;
    str = malloc(13);

    if(str ==NULL){
            puts("unable to allocate memory");
            }else{
            strcpy(str,"Learning Lad");
            puts(str);

            str = (char *) realloc(str,19);
            strcat(str," rocks");
            puts(str);
            free(str);

            }

    getch();
    return 0;
}
```