REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

Université de M'sila
Faculté des Mathématiques et de l'Informatique
Département d'informatique

جامعة المسيلة
كلية الرياضيات والإعلام الآلي
قسم الإعلام الآلي

Chapitre 3: Partie 1

Les pointeurs et les listes chaînées

Algorithmique et structure de données 2

Présenté par: Dr. Benazi Makhlouf
Année universitaire: 2022/2023

Contenu du chapitre 03:

1. Introduction
2. Les pointeurs
3. Opérations sur les pointeurs
4. Gestion dynamique de la mémoire
5. …

1. Introduction

Un **programme** est un ensemble de données et un ensemble d'instructions où ces données sont stockées en mémoire sous forme de variables

Une **variable** est un espace mémoire qui a un nom, un type, une valeur et une adresse de stockage.

Adresse: indique son emplacement. C'est un nombre naturel qui identifie le premier octet dans lequel se trouve la variable. Habituellement, il est écrit dans le système hexadécimal 16, tel que : 0x5A63

Lorsque l’instruction de déclaration d’une variable est rencontrée lors de l’exécution, le programme demande de système d’exploitation de réserver une place en mémoire de la taille x (selon le type). Et après réservation, le système renvoie l'adresse mémoire qu’on peut l'utiliser comme variable.

L’adresse

- Pour obtenir la valeur d’une variable, il suffit d'écrire son nom.
- Pour obtenir son adresse, c'est-à-dire son emplacement en mémoire, dans algorithme on met le symbole **@** avant le nom de la variable, et en **C** on met le symbole **&** avant le nom de la variable.

Exemple:

```
écrire("valeur de age =",  age, " son adresse =", @age);
printf("valeur de age = %d son adresse = %p",age, &age);
```

- **%p** est un format voir la valeur &age comme une adresse, c'est-à-dire un nombre écrit en hexadécimal 16. On peut utiliser %d pour le voir en décimal.
- L’adresse de age peut etre changer chaque fois qu’on exécute le programme.

2. Les pointeurs

Un **pointeur** est une variable dont sa valeur est une adresse mémoire.

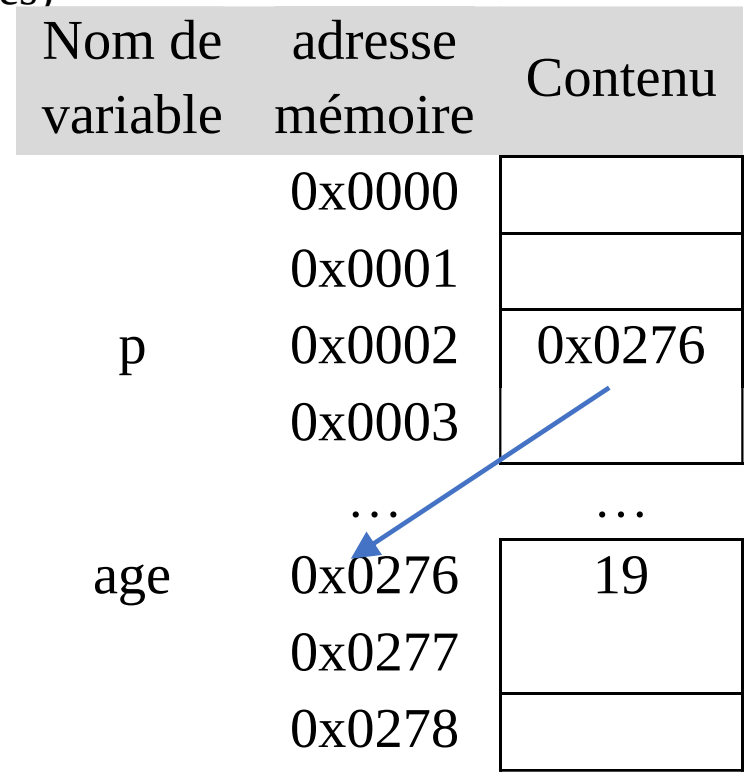
Cette adresse est soit une variable, soit un programme.

Il est utilisé

- pour passer des paramètres par adresse,
- réserver de la mémoire de manière dynamique
- définir des types récursifs (listes, piles et files)
- …

Exemple:

On a deux variables age de type entier sa valeur est 19 et se situer a l'adresse 0x0276 (630_{10}) et p de type pointeur et sa valeur est 0x0276, qui représente l'adresse de age. On dit donc que p pointe vers age.



La création

- En **algorithme**, on ajoute **^ devant** le **type**: `var p1, p2 :^type`
- En **C**, on ajoute * **avant** le **nom** de la variable `type *p1,*p2;`
- Ici ^ ou * indique que la variable est du type pointeur, c'est à dire une adresse mémoire, tandis que type est le type du contenu de cet emplacement.
- il est recommandé d'assigner la valeur **NULL** au pointeur pour indiquer qu'il pointe nulle part. (**NULL est** défini dans stdio.h **define NULL 0**)

```
p1= NULL;
```

Exemple 1

```
Var x : entier      p1, p2 : ^entire
    z : réel            pz : ^reel
```

En C

```
    int   x,*p1,*p2 ;
    float z,*pz ;
```

- p1 peut prendre l'adresse de x ou la valeur de p2 mais pas l'adresse de z ni la valeur de pz ni l'adresse de p2

Opérations valides

```
P1 = &x;          p2 = p1;          pz = &z;
```

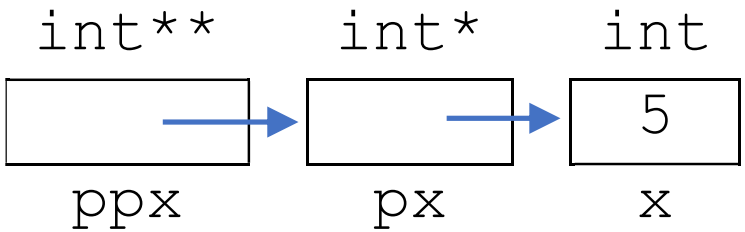
Opérations invalides

```
P1 = x;          x = p1;          p1 = &z;
pz = p1;         P2 = &p1;        p1 = &(0x0276);
```

- Il faut faire la différence entre l'adresse stockée dans le pointeur et l'adresse du pointeur lui-même

Exemple 2

x est de type entier (int), et px contient l'adresse de x, donc son type est (int*) et ppx contient l'adresse de px, donc son type est (int**) comme indiqué dans le schéma suivant :



Déclaration:

```
int x,*px,**ppx;
x=5 ;
px=&x ;
ppx=&px ;
```

Déclaration en utilisant **`typedef`** :

```
typedef int* pint;
typedef int** ppint;
pint px;
ppint ppx;
```

Utilisation

Il est rare que les adresses mémoire soient manipulées directement comme des nombres. Mais, nous manipulons les adresses des variables existantes.

En algorithme

- On utilise l'opérateur **@ avant** le nom de variable pour obtenir son adresse.
- On utilise l'opérateur **^ après** le nom de variable pour récupérer la valeur de la variable (Déréférencement) à partir de son adresse stockée dans un pointer.

En C,

- On utilise l'opérateur **& avant** le nom de variable pour obtenir son adresse.
- On utilise l'opérateur *** avant** le nom de variable pour récupérer la valeur de la variable à partir de son adresse stockée dans un pointer.
- `p←@x ⇒ p^ ⇔ x` `p=&x ⇒ *p ⇔ x`

Exemple

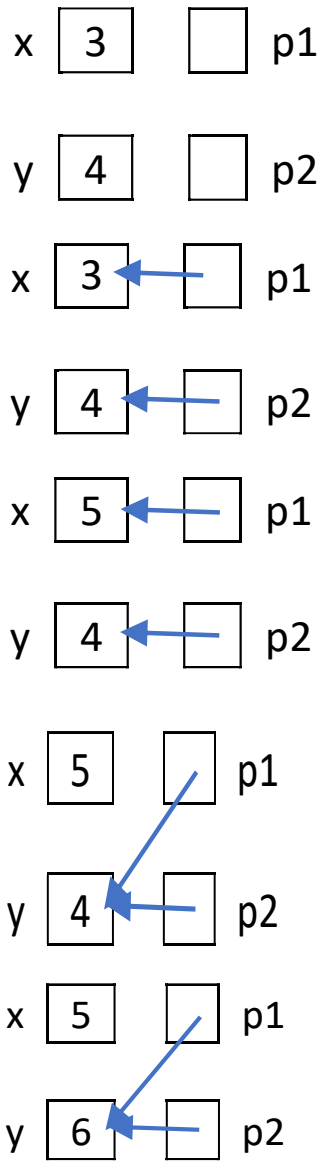
```
int x,*p1,y,*p2 ;
x=3 ;  y=4 ;

p1=&x ;
p2=&y ;

*p1=5;

 p1=p2;

  *p1=6;
```



l'algorithme

Var x, y : entier

p1, p2 : ^ entier

x←3

y←4

p1←@x

p2←@y

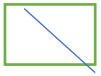
p1^←5

p1←p2

p1^←6

Observations

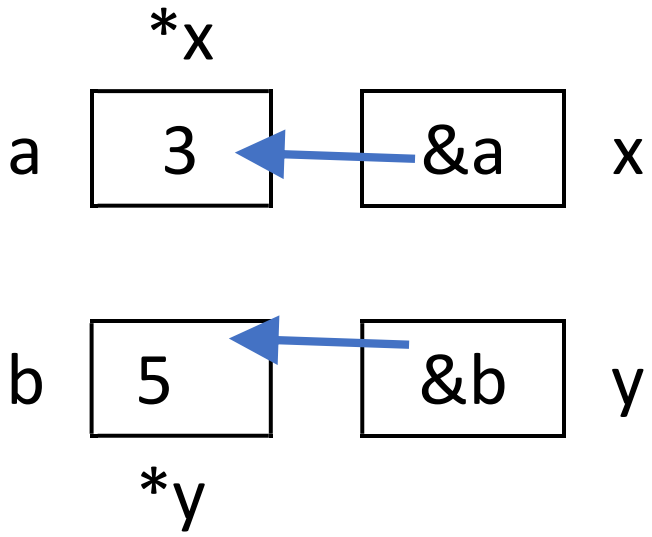
- Pour comprendre les pointeurs, il faut dessiner les variables. le pointeur porte une flèche vers la variable qui porte son adresse.

   représente un pointeur avec la valeur **NULL**
- Un pointeur est toujours de type simple, tandis que la variable dont il contient son adresse peut être de type complexe (tableau ou structure).
- Tenter de récupérer la valeur d'un pointeur non initialisé ou **NULL** entraîne l'arrêt du programme.
- Une valeur (adresse variable) doit être affectée au pointeur avant d'essayer de récupérer la valeur vers laquelle il pointe.
- Avant de récupérer la valeur sur laquelle pointe le pointeur, il faut s’assurer qu'elle ne porte pas la valeur **NULL**.

passage des paramètres par adresse

```
void echanger(int *x, int *y){
int t;
t=*x;
*x=*y;
*y=t;
}
int a=5,b=3;

echanger(&a,&b);
```



3. Opérations sur les pointeurs

Supposons que **P** et **Q** soient deux pointeurs et i un entier

Op	Type du 2e opérant	Type du résultat	
+	int	Pointeur	
P + i	Renvoie un pointeur vers le i^{iem} élément après P dans un tableau		
++		Pointeur	
P++	Renvoie un pointeur vers l'élément suivant immédiatement P dans un tableau		
-	int	Pointeur	
P – i	Renvoie un pointeur vers le i^{iem} élément avant P dans un tableau		
--		Pointeur	
P--	Renvoie un pointeur vers l'élément précédant immédiatement P dans un tableau		

3. Opérations sur les pointeurs

Supposons que **P** et **Q** soient deux pointeurs et i un entier

Op	Type du 2e opérant	Type du résultat	
-	Pointeur du même type	int	
P - Q	Renvoie le nombre d'éléments entre P et Q où P et Q doivent pointer vers le même tableau		
==	Pointeur	Booléen	
P == Q	C'est vrai si P et Q ont la même adresse, c'est-à-dire qu'ils pointent vers le même endroit		
!=	Pointeur	Booléen	
P != Q	C'est vrai si P et Q sont différents		
*		Type de valeur	
*P	Pour récupérer la valeur dont elle contient l'adresse		

4. Gestion dynamique de la mémoire

- **la réservation statique** : le compilateur réserve la mémoire nécessaire pour les variables des le déclarée de manière automatique, et elles ne sont supprimée qu'à la fin de l'exécution du programme (ou du sous-programme dans le cas d'une variable locale).
- **la réservation dynamique** : parfois, on veut réserver une quantité de mémoire durant l'exécution du programme de mémoire dynamique.
  - Pour réserver un tableau de N éléments, on déclare un pointeur et lorsque N devient disponible, nous réservons le tableau.
  - Le développeur dispose d'un ensemble de fonctions lui permettant de gérer la mémoire de manière dynamique, c'est-à-dire pendant l'exécution du programme.

La gestion en algorithme

Il existe trois procédures:

1. **allouer**() pour réserver un tableau où il prend un pointeur (nom du tableau) et le nombre des éléments

   Syntaxe: `allouer(nom_tab,nb_elements)`

   Exemple : `allouer(t,10)`

2. **réallouer**() permet de modifier la taille du tableau, que ce soit pour l'augmenter ou la diminuer.

   Syntaxe: `réallouer(nom_tab, nouvelle_taille)`

   Exemple : `réallouer(t,15)`

3. **désallouer**() pour supprimer le tableau réservé avec **allouer()**

   Syntaxe: `désallouer(nom_tab)`

   Exemple : `désallouer(t)`

accès

Après avoir créé un tableau t par allouer(), ses éléments sont accessibles par les crochets [ ] ou par l'opération de récupération ^.

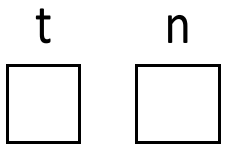
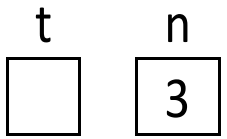
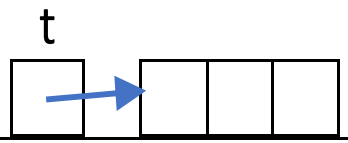
Le pointeur **t** contient l'adresse du premier élément t[0] soit:

@t[0] = t et t^ = t[0]

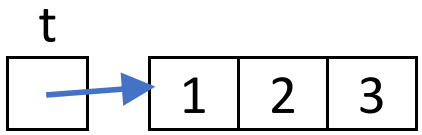
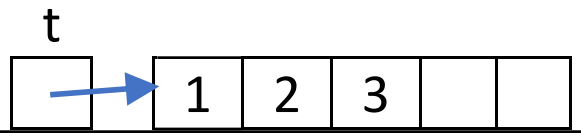
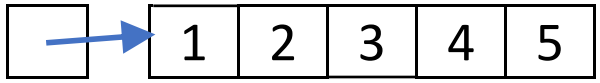
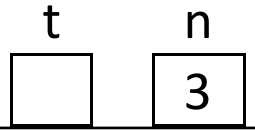
@t[1] = t+1 et (t+1)^ = t[1]

@t[i] = (t+i) et (t+i)^ ⇔t[i]

Exemple 1/2

algorithme	mémoire
var t : ^réel n :entier	
début ecrire("entrer le nombre des éléments") lire(n)	
allouer(t ,n)	

Exemple 2/2

t[0] ←1 t[1] ←2 t[2] ←3 t^ ←1 (t+1)^ ←2 (t+2)^ ←3	
reallouer(t,n+2)	
t[3] ←4 t[4] ←5 (t+3)^ ←4 (t+4)^ ←5	
désallouer(t)	

L’opération « sizeof »

Pour connaître la taille d’un type ou d’une variable en octet

Exemple:

```
float t[20] ;
printf("char   : %d octets\n", sizeof(char));
printf("int    : %d octets\n", sizeof(int));
printf("double : %d octets\n", sizeof(double));
printf("la taille de t: %d octets\n", sizeof(t));
printf("la taille de t: %d octets\n", 20*sizeof(float));
```

qui affiche sur l'écran

```
char   : 1 octets
int    : 4 octets
double : 8 octets
la taille de t: 80 octets
la taille de t: 80 octets
```

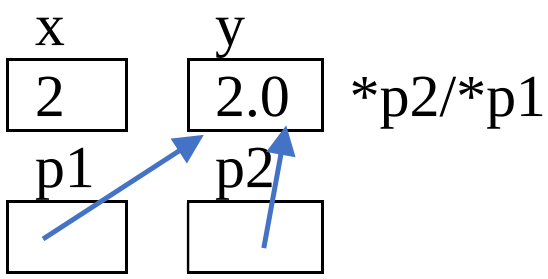
Changement de type : transtypage/casting

pour forcer le compilateur à changer le type d'une valeur spécifique, nous utilisons la formule suivante :

```
(type) expression
```

Exemple 1

```
int A=8,B=3 ;
printf("no casting %f \n", A/B );
printf("with   casting   R=%f   \n",
(float)A/B);
```



Exemple 2

```
int x,*p1 ;
float y=2,*p2 ;
x=(int)y ;
p2=&y ;
p1=(int*)p2 ;
printf("x=%d \n", x);
printf("*p2=%f \n",*p2);
printf("*p1=%d \n",*p1);
```

La gestion en C 1/2

La gestion dynamique de la mémoire en C se fait à l'aide de quatre fonctions définies dans la librairie stdlib :

1. malloc()) ‹**m**emory **alloc**ation pour réserver de la mémoire. Prend la taille en octet et renvoie un pointeur vers cette adresse ou **NULL** en cas d'échec .

Exemple

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

t=	(float *)	malloc(	10*	sizeof(	float	));
Nom du tableau	Convertir en type du pointeur	Pour réserver tableau	Nombre de éléments	La taille de chaque élément	Type de chaque élément	

N.B.: La conversion du type de pointeur n'est pas necessaire en C++

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

La gestion en C 2/2

2. free(), pour retourner la mémoire réservée par le malloc au système. Elle prend un pointeur précédemment réservée. Il est recommandé d'affecter **NULL** au pointeur après avoir appelé **free**

   Exemple

   ```
   free(t);
   ```

3. realloc(), pour changer la taille de la mémoire réservée

   Exemple

   ```
   t=(float*)realloc(t, 20*sizeof(float));
   ```

4. calloc(), comme malloc, sauf qu'il met des zéros dans la mémoire réservée. Elle prend le nombre d'éléments du tableau, et la taille d'une cellule, et renvoie un pointeur vers l'espace réservé.

   Exemple

   ```
   t=(float*)calloc(10,sizeof(float));
   ```

Observations

- Dans la leçon sur les fonctions, nous avons vu que void signifie que la fonction ne renvoie rien, mais void* signifie que la fonction renvoie un pointeur de type indéfini.
- Le type void* doit être convertit vers le type du pointeur qui contiendra l'adresse en plaçant le type pointeur entre parenthèses avant les noms de fonction malloc, calloc et realloc, mais cette conversion n'est pas nécessaire en C++.
- Pour utiliser ces fonctions, vous devez appeler la bibliothèque stdlib ou alloc à l'aide de l'instruction suivante :

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

- Lorsqu’on réserve de la mémoire, on suit ces étapes :
  1. On réserve de la mémoire avec malloc.
  2. On s'assure que le processus de réservation s'est terminé avec succès en utilisant : `if (pointeur != NULL)`
  3. Lorsqu’on a fini d'utiliser l'espace réservé, on retourne la mémoire au système via: `free(pointeur)`

Exemple

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  char *str;
  str = (char *) malloc(4*sizeof(char));
  str[0]='A'; str[1]='S'; str[2]='D'; str[3]='\0';
  //ou
  *str='A'; *(str+1)='S'; *(str+2)='D'; *(str+3)='\0';
  printf("String is %s\n  Address is %p\n", str, str);
  str = (char *) realloc(str, 5*sizeof(char));
  str[3]='2'; str[4]='\0';
  //ou
  *(str+3)='2'; *(str+4)='\0';
  printf("String is %s\n  New address is %p\n", str, str);
  free(str);
  return 0;
}
```

Pointeurs et matrices en C 1/3

- Les matrices en C sont un tableau dont chaque élément est un tableau.

Par exemple on va créer une matrice M[3][4] avec 3 lignes et 4 elements dans chaque ligne (4 colonnes).

- Supposons qu'on a 3 tableaux M0, M1, M2

```
float M0[4],M1[4],M2[4] ;
```

- Ces tableaux peuvent être créés à l'aide de pointeurs

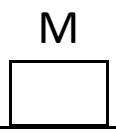
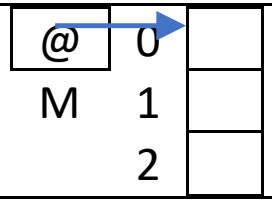
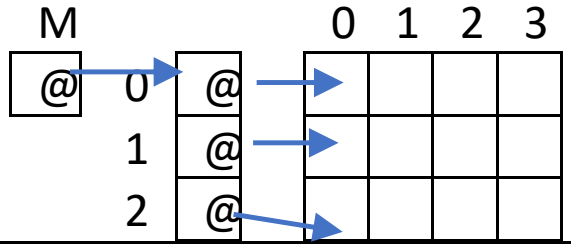
```
float *M0,*M1,*M2 ;
M0=(float *)malloc(4*sizeof(float));
M1=(float *)malloc(4*sizeof(float));
M2=(float *)malloc(4*sizeof(float));
```

- Notons que M0, M1 et M2 sont tous du même type (float *), ils peuvent donc être remplacés par un tableau M de type (float *).

```
float * M[3] ;
for(int i=0;i<3;i++)
  M[i]=(float *)malloc(4*sizeof(float));
```

Pointeurs et matrices en C 2/3

Maintenant, les pointeurs peuvent être utilisés pour créer la table M

C	mémoire
`float **M ;`	
`M=(float**) malloc( 3*sizeof(float*));`	
	
`for(int i=0;i<3;i++)` `  M[i]=(float*) malloc(4*sizeof(float));`	
	

Pointeurs et matrices en C 3/3

en utilisant typedef

```
typedef float ** matrix;
typedef float *  table;
matrix M ;
M=(matrix)malloc(3* sizeof(table));
for(i=0;i<3;i++)
  M[i]=(table) malloc(4*sizeof(float));
```

Tout élément de la matrice est accessible en utilisant [] ou en utilisant l'opérateur de récupération * où

`M[i][j]` ⇔ `*(M[i]+j)`

`M[i][j]` ⇔ `*(*(M+i)+j)`

Remarque : un tableau statique en C est une adresse mémoire constante qui ne peut pas être modifiée.

Exemple: `int *p,t[10];`

```
p=t;  //Correct car t est l'adresse du premier élément
t=p; // Incorrect car t est une constante
```

La gestion de la mémoire en C++

se fait à l'aide de deux **operateurs** :

Il existe deux syntaxes : une pour les types simples et l'autre pour les tableaux.

1. **new** : pour réserver de la mémoire. renvoie un pointeur vers cette adresse ou **NULL** en cas d'échec .

   Exemple

   ```
   float *p, *t;
   p=new float;
   t=new float[n];
   ```

2. **delete** : pour retourner la mémoire réservée par **new** au système.

   Exemple

   ```
   delete p;
   delete[] t;
   ```

Fin partie 1 du Chapitre 03