HESTIM
ENGINEERING & BUSINESS SCHOOL

Allocation dynamique de la mémoire

Filière : Ingénierie Informatique & Intelligence Artificielle

Par : Youssef Ouassit

Les pointeurs

Gestion de la mémoire des variables en C :

Compilation :

- Le compilateur analyse le code et détermine l'emplacement mémoire nécessaire pour chaque variable (taille, type, portée).
- Mais la mémoire n'est pas encore allouée à ce stade : il prépare seulement le plan.

Chargement du programme en mémoire (quand le programme est exécuté):

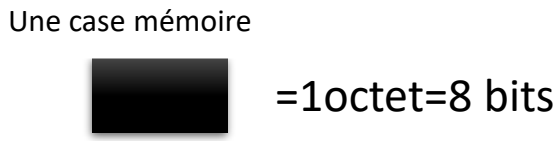
- Zone du code : pour les instructions compilées.
- Zone des données statiques : pour les variables globales et statiques.
- Pile (stack) : pour les variables locales et les paramètres de fonctions.
- Tas (heap) : pour les allocations dynamiques (malloc, calloc, etc.).

Pendant l'exécution:

- Variables globales / statiques → réservées une seule fois au démarrage.
- Variables locales → créées sur la pile quand une fonction est appelée et libérées à sa fin.
- Variables dynamiques → créées dans le tas avec malloc() et libérées par free().

Mémoire

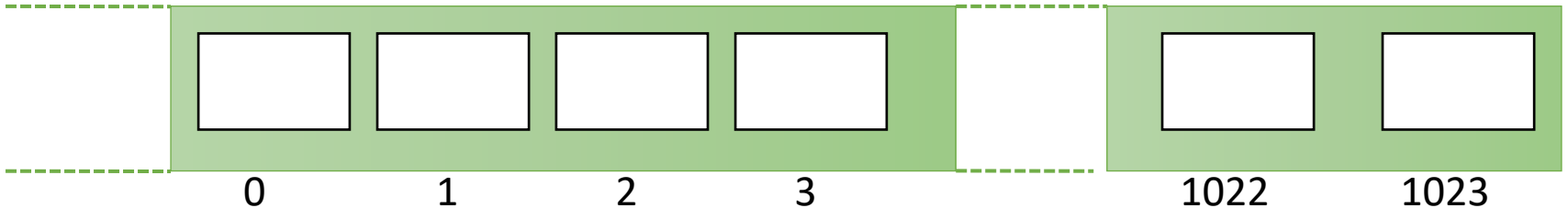
- La mémoire est constituée d’un certain nombre de cases appelées entités adressables.
- Dans la plupart des machines, le contenu d'une case est un octet (au max 2^8-1).
- Le nombre d'entités adressables(de cases) de la mémoire centrale est la capacité de la mémoire.



Adressage de la mémoire

- Chacune des cases de la mémoire centrale est numérotée, et **ce numéro est l'adresse de la case** ou de l'entité adressable(ensemble de cases).

- On numérote à partir de zéro, de sorte que s'il y a 2^{10} = 1024 entités adressables, elles sont numérotées de 0 à 2^{10} - 1 =1023.



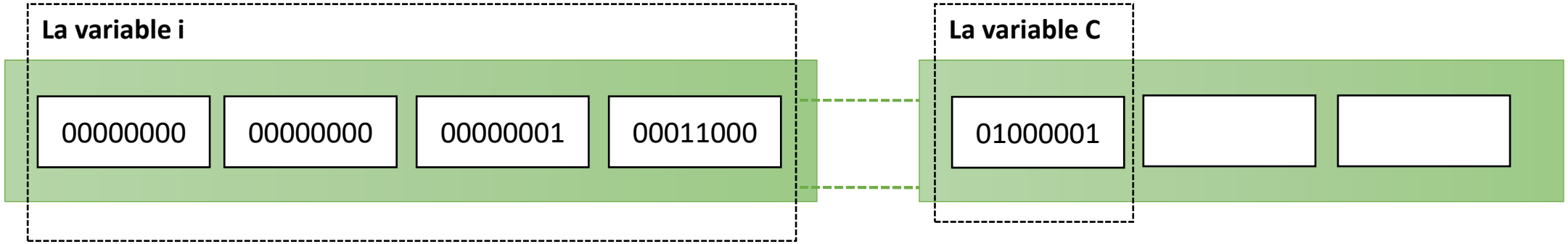
Stockage des données

En langage C les entiers (**int**) sont généralement codés sur 4 octets, donc pour stocker un entier on a besoin de 4 entités adressables. Les caractères (**char**) sont codés sur un octet, donc on a besoin d'une entité adressable. Les réels (**float**) sont codés sur 4 octets, donc on a besoin de 4 entités adressables.

Les valeurs des variables sont stockées dans la mémoire sous la forme binaire.

Exemples :

```
int i=280;   //280 en binaire est 100011000
char c='A';  //'A' en ASCII est 65 en binaire  1000001
```



Adresse d’une variable

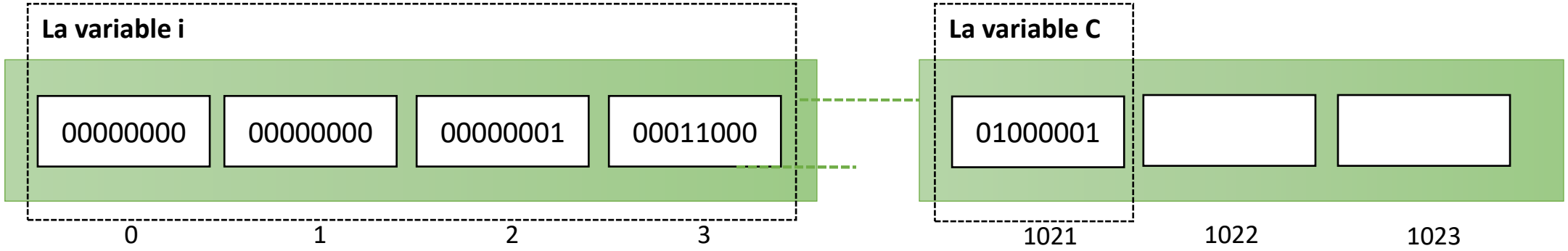
L’adresse d’une variable est l’adresse de sa première case mémoire.

<u>Exemple :</u>

```
int i=280;      //280 en binaire est 100011000
char c='A';     //'A' en ASCII est 65 en binaire  1000001
```

L’adresse de la variable i est 0

L’adresse de la variable c est 1021



L’operateur &

En langage C, Si X est une variable alors &X est l’adresse de la variable (début de l’emplacement mémoire réservé à X).

- & est appelé opérateur d’adresse.

- le format d'une adresse est **%p** (hexadécimal) ou **%d** (décimal) dans printf.

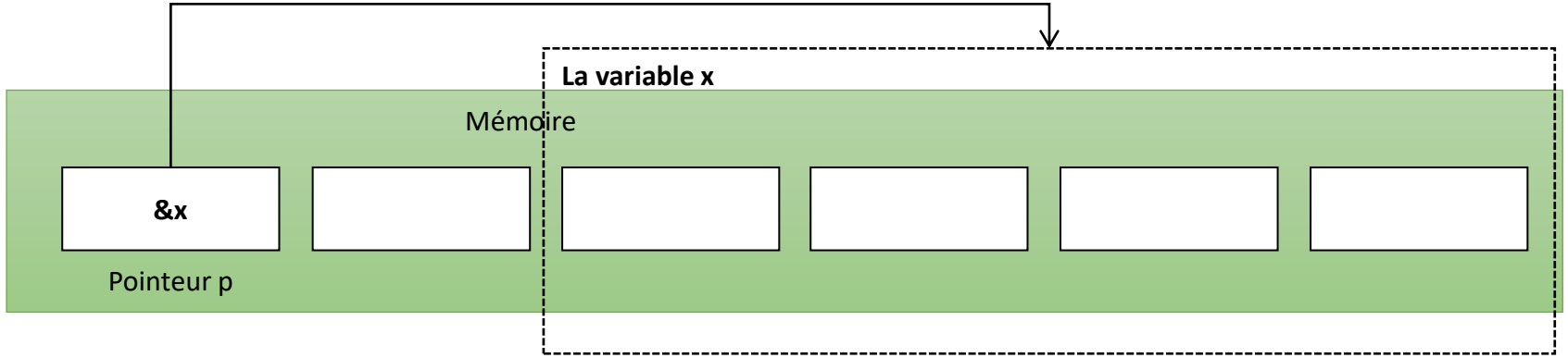
Exemple:

```
int x = 8;
printf("Voici x: %d",x);
printf("Voici son adresse en hexadecimal: %p",&x);
```

Notion de pointeur

- Un pointeur est une variable (d'un type spécial qui pourra contenir l'adresse d'une autre variable.

- On dit qu'il pointe vers cette variable.



Déclaration

La déclaration d'un pointeur devra indiquer le type d'objet vers lequel il va pointer (on dit d'un pointeur qu'il est typé).

Syntaxe : **type *identificateur;**

Exemple: Pour déclarer un pointeur qui va pointer vers un entier on écrira:

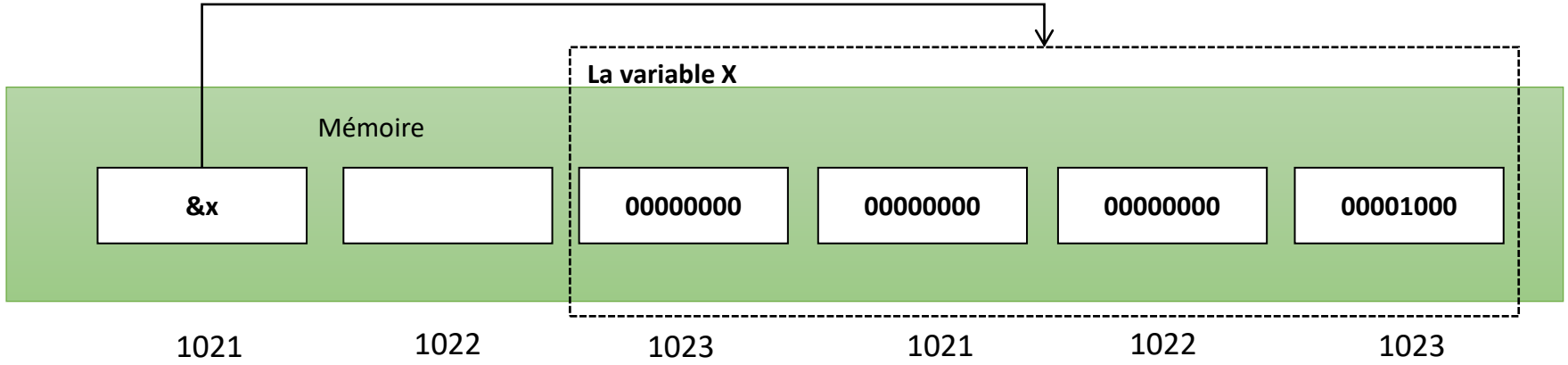
```
int *p;
int* p;
```

Initialisation

Pour initialiser un pointeur de type T on lui affecte l'adresse d'une variable de qui porte le même type T.

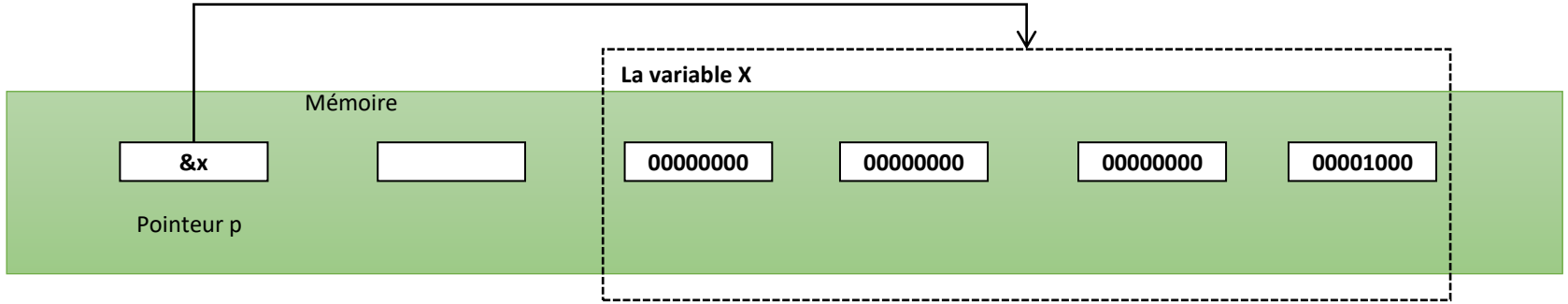
Exemple:

```
int x = 8;
int* p;
p=&x;   //On dit que le pointeur P pointe sur la variable X
```

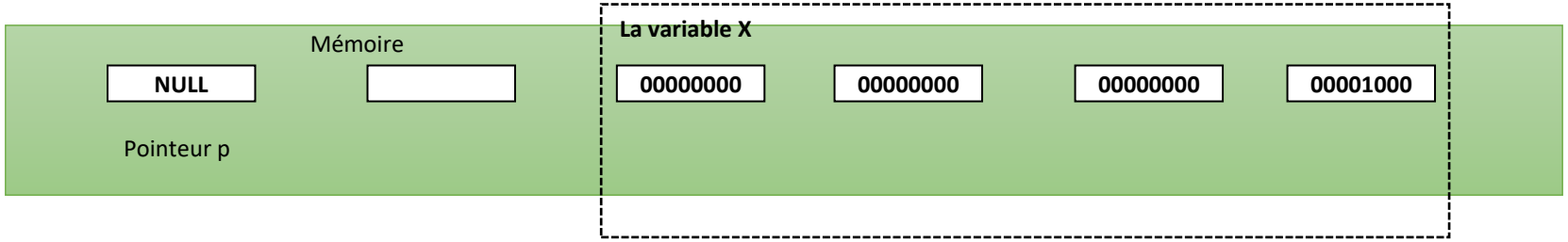


Pointeur NULL

- Un pointeur qui ne pointe sur rien contient la valeur NULL
  - int x = 8;
  - Int* p;
  - p=&x ; //On dit que le pointeur P pointe sur la variable X



  - p=NULL; // le pointeur ne pointe sur rien



L’opérateur *

- On accède à la valeur stockée dans l’emplacement qui porte l’adresse contenue dans un pointeur grâce à l’operateur unaire dit de référencement ou d’indirection *** .**

- Si **p** est un pointeur sur **x** (p==&x) alors ***p** est la variable qui est associé à cette adresse mémoire .

- **x** est équivalente à ***p**

- **p** est équivalente à **&x**

L’opérateur *

Exemple :

```
int x = 8;
Int *p;
p=&x;
```

```
printf(‘’l’adresse de la variable x est %d’’,&x); //ou bien printf(‘’l’adresse de la variable x est %d’’,p)
printf(‘la valeur de la variable x est %d’’,x);  //ou bien printf(‘’la valeur de la variable x est %d’’,*p)
```

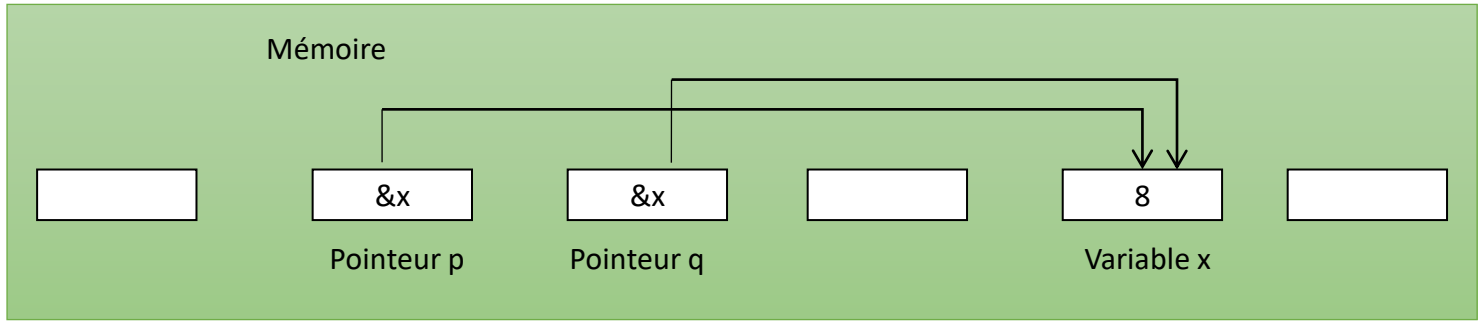
Copie d'un pointeur

Soit la déclaration suivante :

```
int x = 8;
int* p;
int* q;
p=&x;
q=p;
```

Les deux pointeurs p et q pointent sur le *même* emplacement mémoire, donc on a:

- p==q==&x
- et on a également *p==*q==x



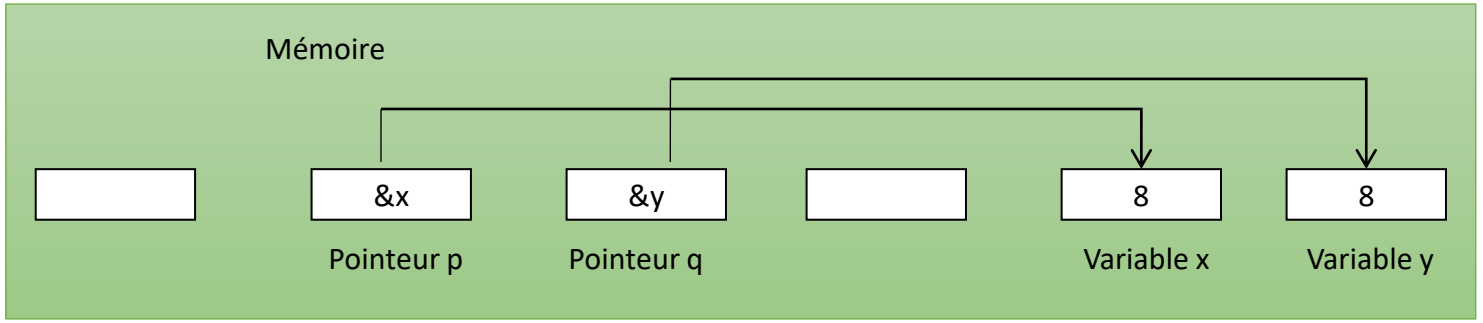
Copie d'une valeur pointée

Soit la déclaration suivante

```
int x = 8;
int y=10;
int* p;
int* q;
p=&x;
q=&y
*q=*p;
```

Les valeurs pointées sont identiques mais les pointeurs restent *différents*, donc on a:

- p==&x et q==&y
- Donc p!=q
- et on a également *p==*q==x==y



Exercice

Soit la déclaration suivante :

```
int a=8;
int b=10;
int *p;
int *q;
p=&a;
q=&b;
```

Donnez le contenu des variables a et b après l ’exécution de chaque instruction :

```
*p=*q+2;
*q++;
*p+=a;
b=*p+*q;
p=q
*p=*(&b);
*p*=*q
```

Solution:

a=24

b=35*35

Remarque

- Les pointeurs et les noms de variables ont le même rôle: Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur.

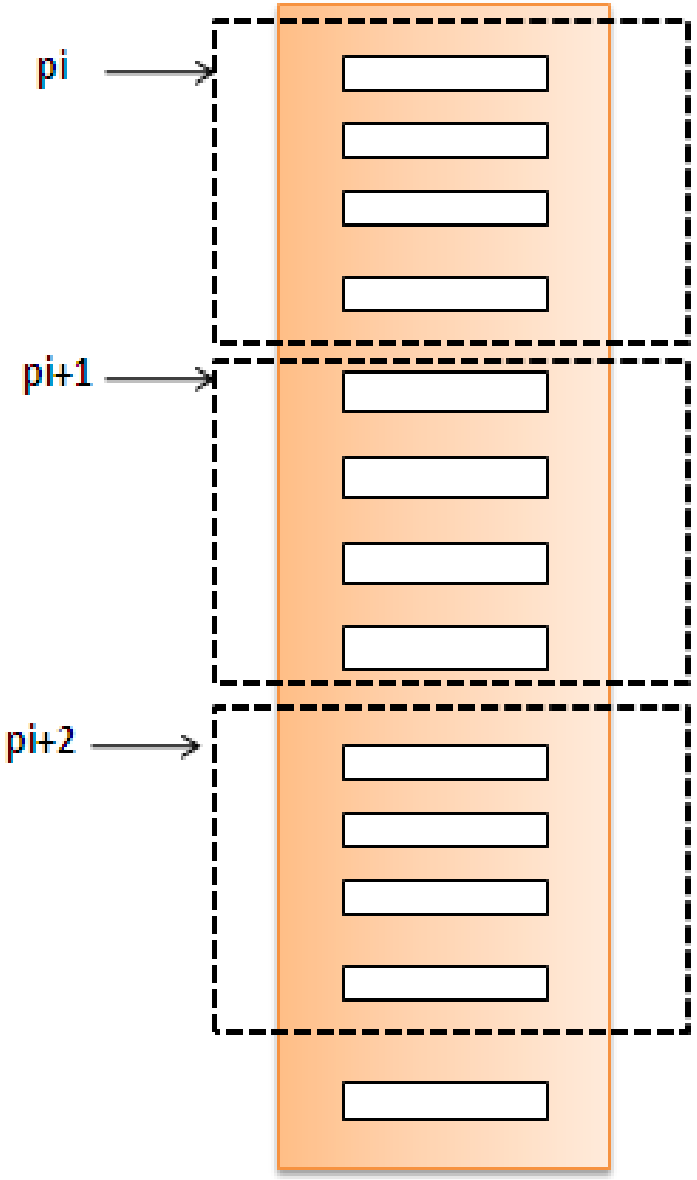
- Il faut quand même bien faire la différence:
  - Un ***pointeur*** est une variable qui peut 'pointer' sur différentes adresses.
  - Le ***nom d'une variable*** reste toujours lié à la même adresse d'un emplacement mémoire où on peut stocker plusieurs fois de nouvelles valeurs.

Arithmétique des pointeurs

- On peut **déplacer** un pointeur dans un plan mémoire à l'aide des opérateurs d'addition, de soustraction, d'incrémentation, et de décrémentation.

- Si **pi** est un pointeur de type int (int *pi) alors **pi+1** est **l'adresse de l'entier suivant**

- On ne peut déplacer **pi** que **d'un nombre de cases mémoire multiple du nombre de cases réservées en mémoire pour la variable sur laquelle il** pointe(pour le cas des entiers c'est 4)**.**

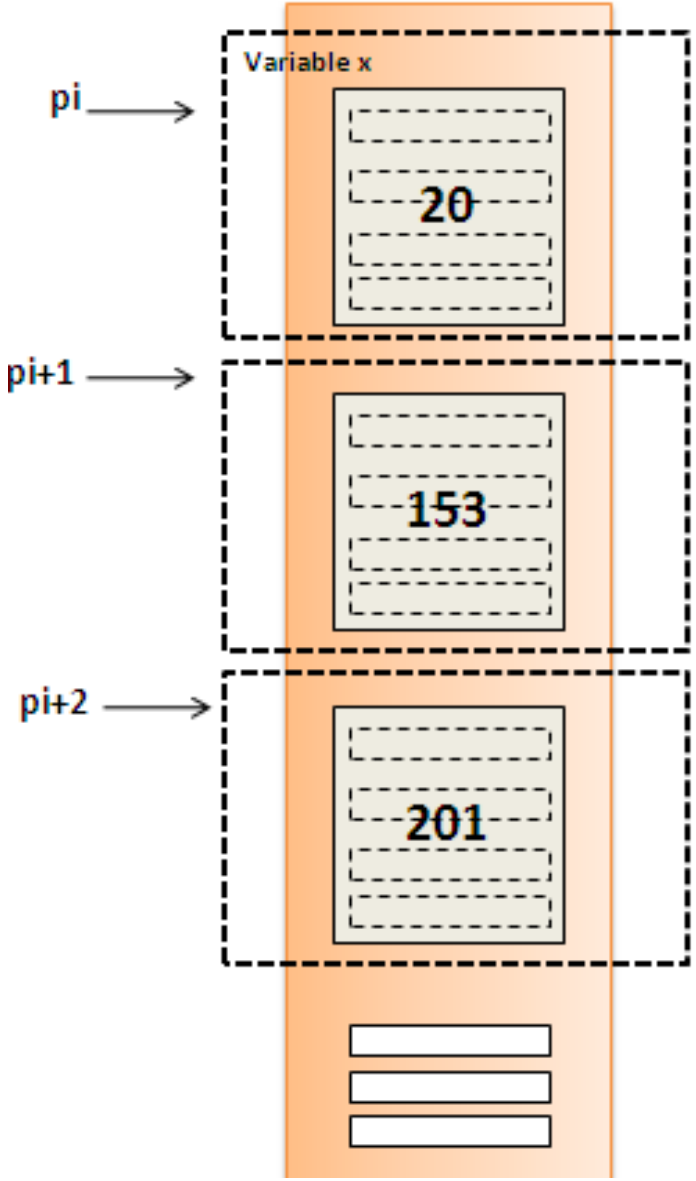


Une case mémoi

Arithmétique des pointeurs

Exemple:

```
int x;
int *pi;
pi=&x
*pi=20;
*(pi+1)=153;
*(pi+2)=201;
```

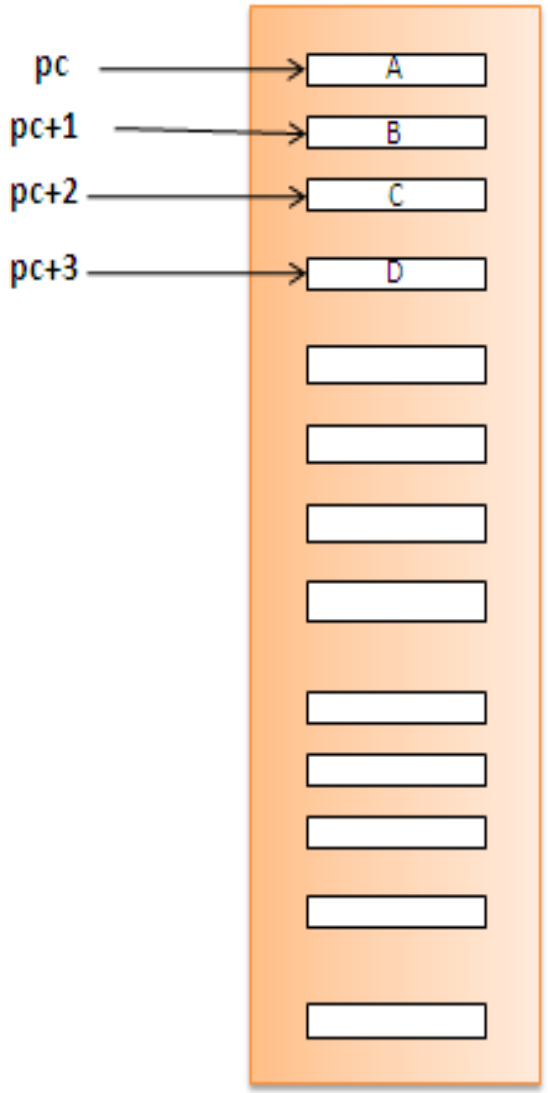


Une case mémoire = 1 octets

Arithmétique des pointeurs

Exemple:

```
 char c
 char *pc;
 pc=&c
*pc='A';
*(pc+1)='B';
*(pc+2)='C';
*(pc+3)='D';
```



Une case mémoire = 1 octets

A quoi sert les pointeurs ?

- **Passage par adresse des paramètres :**

Transmettre à une fonction l'adresse mémoire d'une variable, au lieu de sa valeur.

- **Allocation dynamique de la mémoire:**

Réserver de la mémoire pendant l'exécution du programme (et non au moment de la compilation).

Allocation dynamique de la mémoire

Introduction :

Comment résoudre ce problème ?

Ecrire un programme qui la moyenne de n étudiants entrés par l'utilisateur.

Solution 1: Déclarer un tableau statique avec un nombre de cases suffisant.

Solution 2: Ne réserver le tableau qu'après connaître la valeur exacte de n (Au moment de l'exécution).

Pointeurs et tableaux à une dimension

- L'utilisation des tableaux en langage C impose de réserver de l'espace mémoire avant de savoir le nombre exacte d'éléments dont on a besoin à traiter.
  - Exemple :

```
Int T[100];
int n;
printf(''Donner le nombre des étudiants :'');
scanf(''%d'',&n);
```

Statique

- La déclaration précédente implique que **100-N** cases mémoires seront inutiles.

- Ce qui montre une autre fois l'utilité des pointeurs et de l'allocation dynamique de la mémoire en particuliers.
  - Exemple:

```
int *T;
int n;
printf('Donner le nombre des étudiants : '');
scanf(''%d'',&n);
T=(int *)malloc(n*seizeof(int));
```

Dynamique

Allocation statique vs Allocation dynamique

Allocation statique	Allocation dynamique
Taille connue à la compilation	Taille déterminée à l'exécution
Allouée sur la pile (stack) ou dans le segment global	Allouée sur le tas (heap)
Libération automatique (fin de portée)	Liberté de libérer manuellement (free())

Les fonctions disponibles pour gérer la mémoire (bibliothèque **stdlib.h**) :

Fonction	Rôle	Exemple
malloc()	Alloue un bloc mémoire **non initialisé**	ptr = malloc(taille_en_octets);
calloc()	Alloue et **initialise à 0** un bloc mémoire	ptr = calloc(n, taille);
realloc()	Change la **taille** d'un bloc déjà alloué	ptr = realloc(ptr, nouvelle_taille);
free()	**Libère** la mémoire allouée	free(ptr);

La fonction malloc pour allouer la mémoire :

- Pour allouer de l'espace mémoire on utilise la fonction **malloc** de la bibliothèque **stdlib.h**

- Syntaxe:

  pointeur = (type*) **malloc** (nombre de case mémoires à réserver)

- Exemple

```
char *pc;
int *pi,
float *pr;
pc =(char*) malloc(10)     /* on réserve 10 cases mémoire, soit la place pour 10 caractères */
pi =(int*) malloc(16);     /* on réserve 16 cases mémoire, soit la place pour 4 entiers  */
pr =(float*)malloc(24);    /* on réserve 24 places en mémoire, soit la place pour 6 réels */
```

La fonction calloc pour allouer la mémoire :

- La fonction **calloc** permet d'allouer la mémoire comme la fonction malloc, mais elle initialise tous les octets à 0.

- Syntaxe:

  pointeur = (type*) **calloc** (size_t n, size_t size)

- Exemple

```
int *pi,
float *pr;
pi =(int*) calloc(4, sizeof(int));   /* soit la place pour 4 entiers  */
pr =(float*) calloc(6, sizeof(float));   la place pour 6 réels */
```

La fonction realloc pour allouer la mémoire :

- La fonction **realloc** permet de modifier la taille d'un bloc mémoire déjà alloué dynamiquement.

- Syntaxe:

  ptr = **realloc(void *ptr, size_t new_size);**

- Exemple

```
int * pr,
ptr =(int*) malloc(4*sizeof(int));        /* on réserve 4 cases pour 4 entiers  */
ptr =(int*) realloc(ptr, 10*sizeof(int));    /* Ajouter 6 nouvelles cases*/
```

La fonction free pour libérer la mémoire:

La fonction **free**: pour libérer l'espace mémoire alloué:

```
int *pc = (int) malloc(5*sizeof(int));

free(pc)                    /* on libère la place précédemment réservée pour pc*/
```

Pointeurs et tableaux à une dimension

Un tableau est en fait un pointeur constant, donc il peut être manipuler comme un pointeur.

	Tableaux	Pointeur
Déclaration	**int T[n];**	**int *T;** **T = (int*)malloc(n*sizeof(int));**
Le 1er élément	**T[0]**	***T**
Un autre élément	**T[i]**	***(T+i)**
Adresse du 1er élément	**&T[0]**	**T**
Adresse d'un autre élément	**&(T[i])**	**(T + i)**

Exemple 1

Les deux programmes équivalents suivants copient les éléments positifs d'un tableau T dans un deuxième tableau POS.

```
Void main()
{
   int T[10] = {-3, 4, 0, -7, 3, 8, 0, -1, 4, -9};
   int POS[10];
   int I,J;    /* indices courants dans T et POS */
   for (J=0,I=0 ; I<10 ; I++)
      if (T[I]>0)
      {
         POS[J] = T[I];
         J++;
      }
}
```

```
Void main()
{
   int *T= {-3, 4, 0, -7, 3, 8, 0, -1, 4, -9};
   int *POS;
   POS= (int*)malloc(10*sizeof(int));
   int I,J;    /* indices courants dans T et POS */
   for (J=0,I=0 ; I<10 ; I++)
      if (T[I]>0)
      {
         POS[J] = T[I];
         J++;
      }
}
```

Exercice :

Soit P un pointeur qui 'pointe' sur un tableau A:

```
int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
int *P;
P = A;
```

Quelles valeurs ou adresses fournissent ces expressions:

- a) *P+2
- b) *(P+2)
- d) &A[4]-3
- e) A+3
- f) &A[7]-P
- g) P+(*P-10)
- h) *(P+*(P+8)-A[7])

Remarques :

- Un pointeur est une variable qui peut pointer sur une variable ou un tableau qui portent le même type.
- Un tableau est un pointeur constant, alors il est impossible de changer son contenu par une affectation.
- Soit la déclaration suivante :
  - Int T1[]={1,2,3,4,5,6};
  - Int T2[]={10,11,12,13,14,15};
  - Int *P1;
  - Int *P2;
- Les instructions suivantes sont incorrecte(les tableaux sont des constantes)
  - T1=T2;
  - T2=P1;
- Les instructions suivantes sont correctes(les pointeurs sont des variables)
  - P1=T1
  - P2=T2

Les chaînes de caractères

- Ce qui est dit pour les tableaux peut être appliquer aussi aux chaînes de caractères, mais on doit prendre en compte les remarques suivantes.

1. Un tableau de caractères est un pointeurs constant, donc impossible de changer son contenue par une affectation
   - ~~char t[8];~~
   - t=''Bonjour'' ici on peut utiliser strcpy(t, ''Bonjour'')
2. Pour dépasser se problème on peut utiliser un pointeur à la place d'un tableau
   - char *t;
   - t=''Bonjour'';
3. L'utilisation de scanf(''%s'',t) pour lire la valeur de t est impossibles dans cette exemple parce qu'on n'a pas réserver de l'espace mémoire nécessaire .
   - ~~char *t;~~
   - Scanf(''%s'',t);
4. Pour dépasser ce problème on doit réserver de l'espace mémoire
   - char *t;
   - t=(char*)malloc(n*seizeof(char));
   - Scanf(''%s'',t);

Tableaux statiques à 2 Dimensions

Utilisation du formalisme tableau uniquement .

- syntaxe : Type T[N][M];
- Exemple : int T[3][4];

	0	1	2	3
0	T[0][0]	T[0][1]	T[0][2]	T[0][3]
1	T[1][0]	T[1][1]	T[1][2]	T[1][3]
2	T[2][0]	T[2][1]	T[2][2]	T[2][3]

Tableaux dynamique à 2 Dimensions

- Déclaration d'un pointeur de pointeur
- Exemple:

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

	0	1	2	3
0	T[0][0]	T[0][1]	T[0][2]	T[0][3]
1	T[1][0]	T[1][1]	T[1][2]	T[1][3]
2	T[2][0]	T[2][1]	T[2][2]	T[2][3]

Les pointeurs et les enregistrements

Pointeur sur un enregistrement

- De la même façon que les types simples, on peut déclarer des pointeurs sur des types complexes(enregistrement)

- **Exemple :**

```
typedef  struct {
    char nom[20];
    char prenom[20];
    int   age;
 } personne;
```

```
int main()
{
    personne  *p;
    p=(personne*)malloc(sizeof(personne));
    scanf(“%s”,(*p).nom);
    scanf(“%s”,(*p).prenom);
    scanf(“%d”,&(*p).age);
    printf("%s %s %d",(*p).nom, (*p).prenom, (*p).age);
    return 0
}
```

Pointeur sur un enregistrement

- Langage c simplifie l'accès aux champs de l'enregistrement dans le cas de la déclaration sous forme de pointeur et cela à l'aide de l'opérateur **→.**

(*p).nom → p->nom

(*p).prenom → p->prenom

(*p).age → p->age

```
typedef  struct {
    char nom[20];
    char prenom[20];
     int   age;
} personne;
```

```
int main()
{
    personne  *p;
    p=(personne*)malloc(sizeof(personne));
    scanf("%s", p->nom);
    scanf("%s", p->prenom);
    scanf("%d",& p->age);
    printf("%s %s %d", p->nom, p->prenom, p->age);
    free(p);
    return 0;
}
```

Enregistrement récursif

- Supposons que la structure personne contient maintenant : un nom, un prénom, l'âge , une personne père et une personne mère.

- La structure sera récursive puisqu'on décrit une personne comme contenant deux variables de type Personne qui contiennent chacune deux variables de type Personne, etc.

- **Essayer de déclarer un type structuré qui répond à cette nouvelle extension?**

Enregistrement récursif

- Avec les compétence qu'on jusqu'à maintenant, on peut écrire:

```
typedef  struct {
    char nom[20];
    char prenom[20];
    int   age;
    personne pere;
    personne mere ;
} personne;
```

- Cette déclaration est incorrecte car le compilateur ne peut déterminer a priori la taille en octets de la structure personne.

Enregistrement récursif

- Pour dépasser ce problème , le langage C propose la solution suivante

```
typedef  struct pers {
        char nom[20];
        char prenom[20];
        int   age;
        struct pers *pere;
        struct pers *mere ;
 } Personne;
```