

Programmation 1: Fondamentaux

- Structures
- Pointeurs
- Allocations dynamiques
- fonctions et passages de pointeurs en paramètres

Structures

- Elles permettent de regrouper différents types de données
- La taille de la structure est la somme de la taille de chacun de ses membres

```
struct position {  
    int x;  
    int y;  
} pos1, pos2;
```

Structures

On peut accéder à chacun des membres en utilisant un point :

```
pos1.x = 12;  
pos1.y = 34;
```

L'opérateur d'affectation permet la copie:

```
pos2 = pos1;
```

Structures

Définir une variable structurée

```
struct position {  
    int X ;  
    int Y ;  
};
```

```
struct position Pos1, Pos2 = {0 , 0};  
Pos1.X = 10;  
Pos1.Y = 20;  
Pos2 = Pos1;
```

Structures

Définir un type structuré

```
typedef struct position {  
    int X ;  
    int Y ;  
} Position;
```

Position Pos1, Pos2;

Introduction aux variables de type **pointeur**

Variables de type pointeur

A quoi servent les pointeurs ? :


- Créer un lien, une référence vers des données en mémoire (variables, tableaux, structures, mémoire allouée dynamiquement).

Concrètement les pointeurs ? :

- Ce sont des variables qui stockent des adresses
- La taille ou **sizeof()** d'un pointeur est de 4 octets si le programme est compilé en 32 bits ou de 8 octets pour un programme compilé en 64 bits.
- La déclaration d'une variable de type pointeur précise **le type des données pointées** et le nom du pointeur, l'étoile indique qu'il s'agit d'un pointeur :

```
int *ptr1;
```

Déclaration d'une variable... unsigned char MyVarChar = 255



ADRESSE (héxa)	VALEUR (héxa)
...	
0x0027FDB0	0xFF
0x0027FDB1	
0x0027FDB2	
0x0027FDB3	

la valeur 255 est rangée en mémoire à l'adresse 0x0027FDB0 quand on affecte la variable MyVarChar: `MyVarChar= 255; // soit 0xFF`

l'expression « `&MyVarChar` » représente l'**ADRESSE** de MyVarChar et vaut **0x0027FDB0**.

Déclaration d'une variable... unsigned short int MyShortInt = 0xABCD

	ADRESSE (héxa)	VALEUR (héxa)
	...	
MyShortInt →	0x0027FDB0	0xCD
	0x0027FDB1	0xAB
	0x0027FDB2	
	0x0027FDB3	

la valeur 43981 est rangée en mémoire à l'adresse 0x0027FDB0 quand on affecte la variable MyShortInt: `MyShortInt = 0xABCD; // soit 43981`

l'expression « `&MyShortInt` » représente l'**ADRESSE** de MyShortInt et vaut **0x0027FDB0**.

Déclaration d'un **Pointeur SUR** unsigned short int **MyShortInt** = 0xABCD

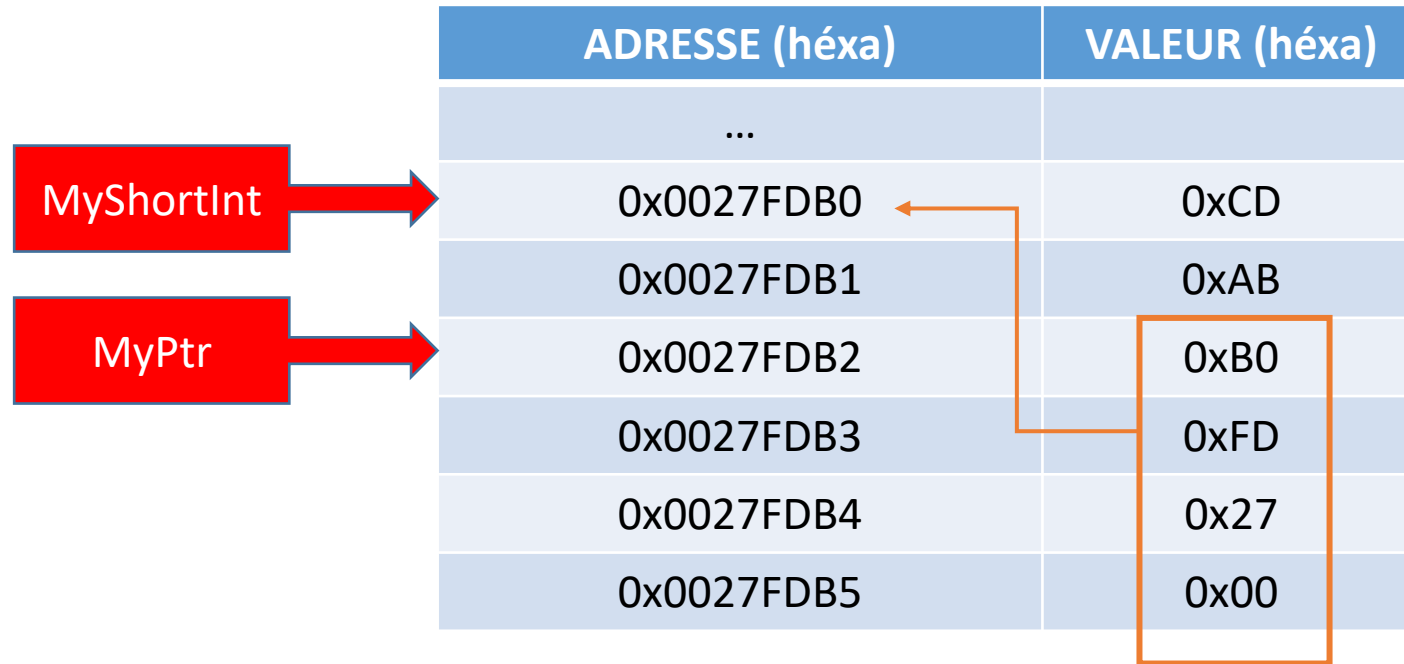
	ADRESSE (héxa)	VALEUR (héxa)
	...	
MyShortInt →	0x0027FDB0 ←	0xCD
	0x0027FDB1	0xAB
MyPtr →	0x0027FDB2	0xB0
	0x0027FDB3	0xFD
	0x0027FDB4	0x27
	0x0027FDB5	0x00

la valeur 0x0027FDB0 est rangée en mémoire à l'adresse 0x0027FDB2 quand on affecte la variable MyPtr :

MyPtr = &MyShortInt;

l'expression « **&MyShortInt** » représente l'**ADRESSE** de MyShortInt et vaut **0x0027FDB0**.

Déclaration d'un **Pointeur SUR** unsigned short int **MyShortInt** = 0xABCD



Pour manipuler la valeur référencée par le pointeur MyPtr, il faut **déréférencer** le pointeur avec l'opérateur *****

```
printf( " %hX ", *MyPtr ); // affiche la valeur en hexadécimal soit ABCD
```

Variables de type pointeur

La variable de type pointeur permet de **mémoriser une adresse de la mémoire**.

Déclaration d'un pointeur sur une valeur de type int nommé ptr1 :

```
int *ptr1;
```

Assignation d'une valeur de type adresse au pointeur :

```
ptr1 = 0x00D0F8D8;
```

ou, mieux encore avec un **cast** :

```
ptr1 = (int *) 0x00D0F8D8;
```

Cast d'une adresse
quelconque vers *référence sur
un entier*

Variables de type pointeur

Déclaration d'une valeur de type **int** nommée **valeur1** :

```
int valeur1 = 10;
```

Assignation de l'adresse de **valeur1** au pointeur :

```
ptr1 = &valeur1;
```

Utilisation de la valeur pointée par déréférencement du pointeur :

```
printf("%d", *ptr1); // affiche 10
```

```
printf("%d", valeur1); // affiche 10
```

```
*ptr1 = 20;
```

```
printf("%d", valeur1); // affiche 20
```

Variables de type pointeur

Déclaration d'un pointeur sur un type non-défini:

```
void *ptr;
```

Initialisation d'un pointeur vers **rien** :

```
ptr = NULL;
```

Duplication d'un pointeur :

```
int *debut, *fin;  
fin = debut;
```

Duplication d'un pointeur avec **cast** : void * -> int *

```
int *debut;  
debut = (int *) ptr
```

Variables de type pointeur

Opération sur un pointeur:

```
int *ptr;  
  
ptr++; // avance d'un entier dans la mémoire  
ptr+=2; // avance de deux entiers
```

Attention, la comparaison de deux pointeurs n'a pas toujours un sens. Pour la comparaison de chaînes de caractères, on utilisera la fonction **strcmp()**.

```
if ( ptr1 < ptr2 ) { ... }; // adresse contenue dans ptr1 est avant celle contenue dans ptr2  
if ( ptr1 == ptr2 ) { ... }; // les pointeurs ont la même valeur et désignent à priori les mêmes données.  
if ( ptr1 == NULL ) { ... }; // usage recommandé pour tester un pointeur qui ne référence pas de  
// données.
```

Variables de type pointeur

Les paramètres de type pointeur dans les appels de fonction:

Exemple : Permutation de deux valeurs

```
void permut(int *a, int*b) {  
    int tmp;  
    tmp = *a;  *a = *b;  *b = tmp;  
}
```

```
int X = 7; int Y = 14;  
permut(&X , &Y);
```


Variables de type pointeur

Utilisation du nom du tableau comme un pointeur :

```
#define TABSIZE 10  
int MyTab[TABSIZE];  
int i;
```

```
for (i=0; i<TABSIZE; i++) *(MyTab+i)=0;
```

MyTab, écrit sans les [], équivaut à un pointeur sur le premier élément du tableau.



Variables de type pointeur

Utilisation du nom du tableau comme un pointeur:

$\text{*(MyTab)} \Rightarrow \text{MyTab}[0]$

$\text{*(MyTab+1)} \Rightarrow \text{MyTab}[1]$

$\text{*(MyTab + TABSIZE - 1)} \Rightarrow \text{MyTab}[\text{TABSIZE}-1]$

Variables de type pointeur

Exercice :

Ecrire une fonction **MyStrlen** qui calcule la longueur d'une chaîne de caractères

Remarque : `nom[5] = {'J', 'O', 'H', 'N', '\0'}`
`nom[5] = { 74, 79, 72, 78, 0 }`

En langage C, une chaîne de caractères est construite avec un simple tableau de codes de caractère et se termine par la valeur 0.

La valeur 0 est équivalente au caractère spécial '`\0`'

Variables de type pointeur

Expliquer les différents cas :

```
char prenom[20] = { 'E', 'L', 'O', 'D', 'I', 'E', '\0' };
```

```
char nom[] = "JOHN";
```

```
*(nom + 2) = 'E';
```

```
*(nom + 3) = 'L';
```

```
char *message = "JE SUIS INVARIANT";
```

```
*(message + 1) = 'E'; // interdit
```

Variables de type pointeur

Expliquer les différents cas :

```
char prenom[20] = { 'E', 'L', 'O', 'D', 'I', 'E', '\0' }; // le tableau de capacité 20 est initialisé avec les six caractères + le caractère de fin
```

```
char nom[] = "JOHN"; // La taille du tableau est déduite de la chaîne "JOHN", la chaîne est placée // dans le tableau avec sa fin de chaîne.  
*(nom + 2) = 'E';    // On modifie les lettres du tableau.  
*(nom + 3) = 'L';
```

```
char *message = "JE SUIS INVARIANT"; // Un pointeur de caractère est dirigé vers un message constant // en mémoire.  
*(message + 1) = 'E'; // interdit: message pointe sur une chaîne de caractères constante
```

Variables de type pointeur

Utiliser le nom du tableau comme un pointeur, répercussion pour un tableau à **deux dimensions**:

- En utilisant un pointeur sur le début du tableau, celui-ci peut-être manipulé comme un tableau à une dimension.
- La taille du tableau en mémoire est égale à :

```
sizeof(type)* nb_de_ligne * nb_de_colonne
```

- Pour accéder à un élément en position (ligne, colonne) :

```
*(tab + nb_colonnes*ligne + colonne)
```

Variables de type pointeur

Utiliser le nom du tableau comme un pointeur: répercussion pour un tableau à deux dimensions

```
#define NBLIGNE 5
#define NBCOL 11
int tableau[NBLIGNE][NBCOL] = { 0 };
int ligne = 2;
int colonne = 5;
tableau[ligne][colonne] = 9;
// ou , accès à l'aide d'un pointeur, cela sera utile afin de rendre génériques
// les fonctions qui manipulent des tableaux à deux dimensions

int *ptrTab = tableau; // ptrTab pointe sur le 1er élément du tableau à 2 dimensions
*(ptrTab + NBCOL*ligne + colonne) = 99;

printf("\n valeur modifiée dans le tableau = %d", tableau[ligne][colonne]);
```

Variables de type pointeur

Utiliser le **nom du tableau** comme un **pointeur**, répercussion pour un tableau à **deux dimensions**:

En utilisant un pointeur sur le début du tableau, celui-ci peut-être manipulé comme un tableau à une dimension

```
#define nbCol 10
void InitTab(int tab[][nbCol], int sizeY, int sizeX) {
    for (int l = 0; l < sizeY; l++)
        for (int c = 0; c < sizeX; c++)
            tab[l][c] = 0; // remplissage avec valeur 0
}

void InitTab2(int *tab, int sizeY, int sizeX) {
    // sizeY, sizeY sont les dimensions du tableau à deux dimensions
    int tabsz = sizeX * sizeY;
    for (int i = 0; i < tabsz; i++) *(tab + i) = 0;
}
```


Variables de type pointeur

Il est possible de créer ou de manipuler des variables de type **pointeur sur pointeur** :

Pour accéder à la valeur pointée, il faut **déréférencer** le pointeur **plusieurs fois**

```
int **ptr2; // déclaration d'un pointeur sur pointeur
```

```
int *ptr1;
```

```
int valeur = 123;
```

```
ptr1 = &valeur;
```

```
ptr2 = &ptr1; // assignation d'une valeur de pointeur à ptr2
```

```
printf("\n %d \n", **ptr2); // double déréférencement de ptr2
```

Pointeurs et allocations dynamiques de mémoire

- l'allocation dynamique fournit à la demande une zone de mémoire disponible pour y stocker des données. l'accès à cette mémoire se fait à l'aide d'un pointeur :

```
ptr = malloc( taille en octets )

int *buffer;
buffer = (int *) malloc( 10 * sizeof(int) );

if (buffer == NULL) {
    printf("\nMémoire insuffisante\n");
    return(-1);
}
buffer[5]=12571;
```

- La fonction `calloc(nb_elts, sizeof(elt))` réalise une allocation et initialise la mémoire à zéro

Pointeurs et allocations dynamiques de mémoires

Dès qu'une mémoire allouée dynamiquement n'est plus utilisée, il faut la restituer au système.

```
free( ptr )
```

Il faut éviter de faire beaucoup de petites allocations dynamiques, cela est couteux pour le système. Cependant, le mécanisme d'allocation prend en charge les petites allocations différemment des grandes afin d'optimiser le processus.



Pointeurs et allocations dynamiques de mémoires

Exercice :

Ecrire la fonction

```
char * cloneStr(char *)
```

qui clone une chaîne de caractères en allouant la mémoire nécessaire et renvoie un pointeur sur la copie réalisée, ou un pointeur NULL si l'opération a échoué.

Pointeurs et allocations dynamiques de mémoires

La fonction **realloc()** permet d'ajuster la taille d'un allocation dynamique.

```
int *buffer; int *tmp;
buffer = (int *)malloc(10 * sizeof(int));
if (buffer == NULL) {
    printf("\nMémoire insuffisante\n");
    return(-1);
}

// Sauvegarde de l'ancienne valeur du pointeur de la mémoire allouée
int *oldBuffer = buffer;
// On essaie de modifier la taille de la mémoire allouée
buffer = (int *) realloc( buffer, nouvelle_taille * sizeof(int));
// Si cela échoue, il faut restaurer l'ancien pointeur
if (buffer == NULL) {
    buffer = oldBuffer;
    printf("\n Mémoire insuffisante \n");
    return(-1);
}
```

Pointeurs et allocations dynamiques de mémoires

Remarque sur la fonction **realloc()** :

- Le pointeur passé en paramètre n'est pas modifié.
- Pour récupérer le nouveau pointeur il faut assigner la valeur de retour de `realloc()`.
- Si la valeur de retour est NULL, la réallocation a échoué
- Il faut penser à sauvegarder l'ancien pointeur pour le rétablir en cas d'échec car les données s'y trouvent encore
- Si la réallocation fonctionne, l'ancienne zone est automatiquement libérée le cas échéant.
- La nouvelle zone peut se trouver au même endroit que l'ancienne, ou pas.
- La fonction de réallocation doit en effet trouver un espace de mémoire contiguë suffisamment grand.
- La fonction `realloc()` permet également de réduire la taille de la mémoire allouée, avec une perte possible des données qui s'y trouvent.
- La fonction `_msize()` permet de retrouver la taille d'une mémoire allouée de manière dynamique.

Pointeurs et Structures

- L'accès aux champs d'une structure pointée se fait avec ->

```
typedef struct heure {  
    int heure;  
    int minute;  
} Heure;
```

```
void incrementMinute(Heure *now) {  
    now->heure = (now->minute + 1) / 60 + now->heure;  
    now->minute = (now->minute + 1) % 60;  
}
```

Pointeurs : Exercices

- Ecrire la fonction **cloneLog()** qui réalise le clone d'un tableau de structures Heure en utilisant l'allocation dynamique.
- Ecrire la fonction **afficheLog()** qui affiche un listing des heures contenues dans le tableau

```
Heure * cloneLog(Heure *Log, int nbLog) {    }  
void afficheLog(Heure * Log, int nbLog) {    }
```




Pointeurs : Exercices

Dans la fonction `int main(int argc, void **argv)`, afficher la liste des paramètres passés sur la ligne de commande :

- **argc** : nombre d'arguments (dont chemin complet et nom du programme)
- **argv** : pointeur sur un tableau de chaînes

	variable simple	pointeur	pointeur sur structure
déclaration	int x;	// pointeur sur un entier int *debut;	struct personne personne1 = {"Burton",51}; struct personne personne2 = {"Audiard",57}; // pointeur sur une structure personne struct personne *realisateur;
utilisation locale de la valeur	x = 12;	debut = &x; // debut pointe sur la variable x *debut = 88; // modification de la valeur pointée // x vaut 88 // debut contient l'adresse de la variable x printf("\n %d ", x); printf("\n %d ", *debut);	realisateur = &personne1; printf("\n Nom : %s, âge : %d", realisateur->nom, realisateur->age); realisateur = &personne2; printf("\n Nom : %s, âge : %d", realisateur->nom, realisateur->age);
utilisation dans une fonction : passage par valeur	void Afficher(int valeur) { printf("%d", valeur); } Afficher(x);	void Afficher(int valeur) { printf("%d", valeur); } Afficher(*debut);	void AffichePersonne(struct personne pers) { printf("\n%s %d\n", pers.nom, pers.age); } AffichePersonne(*realisateur);
utilisation dans une fonction : passage par adresse	void Doubler(int *valeur) { *valeur = *valeur * 2; } Doubler(&x);	void Doubler(int *valeur) { *valeur = *valeur * 2; } Doubler(debut);	void majuscule(struct personne *pers) { int i; int lg = strlen(pers->nom); for (i=0; i<lg; i++) { *(pers->nom+i)=toupper(*(pers->nom+i)); } } majuscule(realisateur);