



GRANDE ÉCOLE D'INGÉNIEUR GÉNÉRALISTE

# Programmation Embarquée

## 2021-2022

[www.hei.fr](http://www.hei.fr)



# VII. Pointeurs

## Introduction:

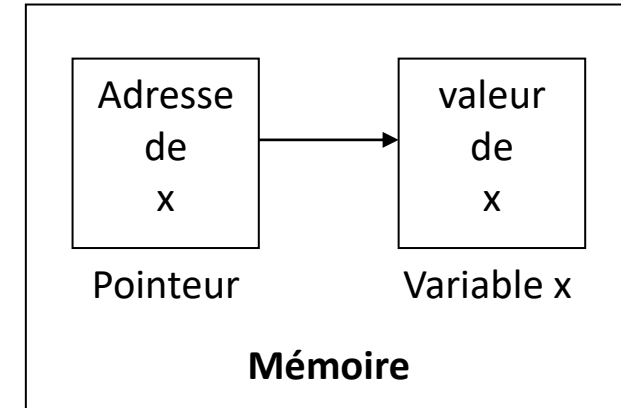
L'accès à une variable peut se faire par l'intermédiaire de son **nom** car une variable n'est rien d'autre qu'une **zone mémoire** portant un nom défini par le programmeur. On peut également choisir un chemin d'accès indirect à la variable, par le biais de son **adresse**. Pour cela, on utilise un pointeur (*pointer* en anglais).

**Un pointeur est une donnée constante ou non, qui mémorise l'adresse d'une variable.**

On dit que le pointeur renvoie (ou pointe) vers la variable concernée, cela via son contenu consistant en une adresse de variable. Les pointeurs sont parfois appelés *indirections*. On dit également que le pointeur fait référence à la variable.

Un pointeur dont le contenu est modifiable, donc qui peut mémoriser les adresses de différentes données, est une variable pointeur. De telles variables, également appelées variables d'adresses, sont d'un type particulier, adapté à la mémorisation d'adresses.

**Il existe par ailleurs des pointeurs constants. Par exemple, les noms des tableaux sont des pointeurs constants, qui équivalent à l'adresse (de début) du tableau concerné.**



# VII. Pointeurs

## Définition:

On définit une variable pointeur selon la syntaxe suivante :

**<type données> \*<pointeur>**

## Exemple :

```
int *toto ;  
char *pc ;  
short *ps ;  
long *pl ;  
float *pf ;  
double *pd ;
```

- La variable « **toto** » est un pointeur vers une donnée de type **int**,
- La variable définie « **toto** » a le type **int\***
- La variable définie « **toto** » peut mémoriser l'adresse d'une donnée de type **int**
  
- **pc** est un pointeur vers un **char**.
- **pc** a le type **char\***.
- **pc** peut contenir l'adresse d'une donnée de type **char**.
- ...
- **pf** est un pointeur vers un **float**.
- **pf** a le type **float\***.
- **pf** peut contenir l'adresse d'une donnée de type **float**.


# VII. Pointeurs

## Initialisation:

La valeur d'une variable pointeur est indéfinie lors de la définition, et le demeure tant qu'on ne lui en affecte pas explicitement une. Cela peut se faire, par exemple, par le biais d'une initialisation dès la définition.

### Exemple :

```
short s ;  
float f ;  
char c  ;  
short *ps = &s ;    // ps est un pointeur vers short,  
                    // initialisé avec l'adresse de la variable s  
float *pf = &f ;    // pf est un pointeur vers float,  
                    // initialisé avec l'adresse de la variable f  
char *pc = &c ;    // pc est un pointeur vers char,  
                    // initialisé avec l'adresse de la variable c
```



Dans ces définitions, on affecte aux variables pointeurs, comme valeurs initiales, des adresses sous forme d'expressions constituées par le nom de la variable concernée et par l'opérateur d'adressage &.

# VII. Pointeurs

## Affichage:

La chaîne de format d'une adresse mémoire est "%X". On affiche donc une adresse mémoire (très utile pour déboguer)!!

## Exemple:

```
#include <stdio.h>
int main ( )
{
    int x = 3 ;
    int *p ;
    p = &x ;
    printf ("p contient la valeur %X, qui n'est autre
    que l'adresse %X de x\n", p , &x ) ;
    return 0 ;
}
```

# VII. Pointeurs

## Accès indirect aux variables:

Pour pouvoir accéder, via un pointeur, à une autre donnée, l'opérateur \* est nécessaire mais ici avec la signification d'opérateur d'indirection (ou opérateur de contenu ou opérateur de référence).

L'utilisation de l'opérateur d'indirection \* doit être différenciée de son utilisation dans les définitions de variables pointeur:

- **On décrit l'opérateur \* dans une définition de pointeur par la formulation « est un pointeur vers » ;**
- **En revanche, sa signification dans une instruction d'un programme s'exprime « contenu de ».**

# VII. Pointeurs

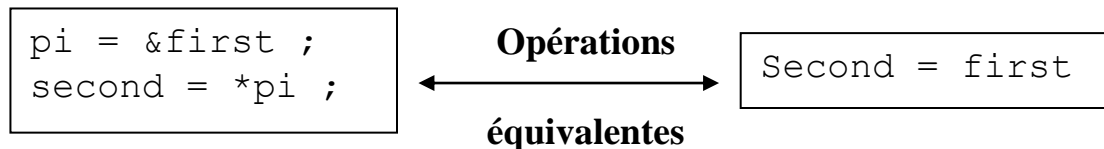
## Accès indirect aux variables:

### Exemple:

```
int first, second ;  
int *pi ; // pi est un pointeur vers un int.  
          // pi a le type int*.  
          // pi peut contenir l'adresse d'une donnée int.  
first = 1234 ;  
pi = &first ; // pi mémorise l'adresse de first  
  
second = first ; // second prend la valeur de first  
  
second = *pi ; // on accède indirectement au contenu de  
               // first qu'on affecte à second
```

**\*pi** signifie :

- valeur (ou contenu) de la donnée pointée par la variable pointeur **pi**,
- valeur (ou contenu) de la donnée dont l'adresse est stockée dans la variable pointeur **pi**

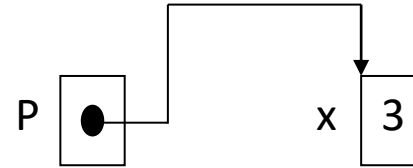


# VII. Pointeurs

## Accès indirect aux variables:

### Exemple :

```
#include<stdio.h>
int main ()
{
    int x=3;
    int *p;
    p=&x;
    return 0;
}
```



x est de type int. p est de type int\*, c'est à dire de type pointeur de int,  
p est donc faite pour contenir l'adresse mémoire d'un int.  
&x est l'adresse mémoire de la variable x,  
et **l'affectation p = &x place l'adresse mémoire de x dans le pointeur p.**  
A partir de cette affectation, **p pointe sur x.**



# VII. Pointeurs

## Initialisation:

### Exemple :

```
int first, second ;
int *p1, *p2; /* deux pointeurs vers int */
first = 1234 ;
p1 = &first ; // p1 mémorise l'adresse de first
p2 = &second ; // p2 mémorise l'adresse de second

second = *p1 ; /* ces trois instructions... */
second = first ;
*p2 = *p1 ; /* ... sont identiques */
```

## Remarques :

- Ici, on accède indirectement aux deux variables. La valeur de la donnée pointée par **p1** (**first**) est reportée à l'emplacement mémoire pointé par **p2** (**second**).
- **first** et **second** ont donc maintenant, tous les deux, la valeur 1234.
- **\*p1** et **\*p2** représentent des données du type de l'objet pointé. Ils peuvent remplacer les variables **first** et **second**, non seulement dans des affectations, mais également dans toutes les autres occasions, par exemple dans des opérations arithmétiques

```
first = first - 1232 ; /* initialise
'first' à la valeur 2 */
second = first * second ; /*initialise
'second' à la valeur 2468 */
```

# VII. Pointeurs

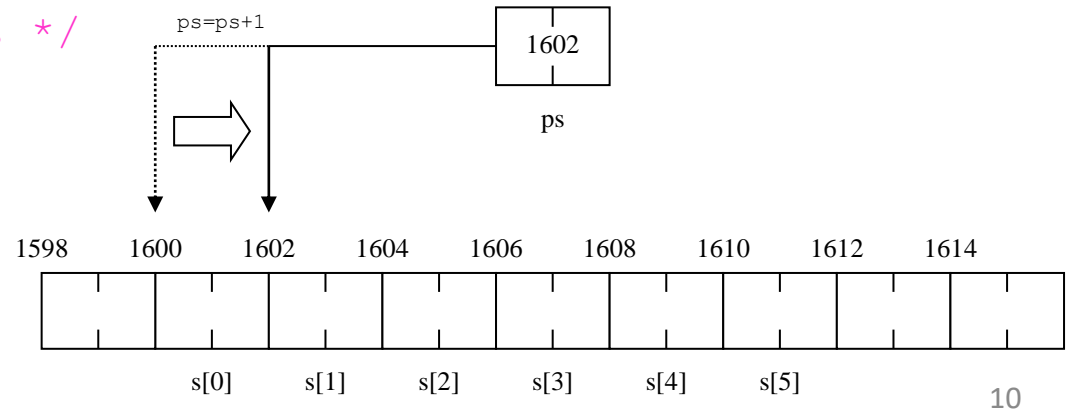
## Arithmétique des pointeurs:

**Addition:** La valeur d'un pointeur peut être augmentée d'un nombre entier. Mais **ATTENTION** :

### Exemple 1 :

```
short s[6];      /* tableau de type short */
short *ps;       /* ps est un pointeur vers short */
ps = &s[0];      /* ps pointe sur le 1er élément du tableau s */
```

```
ps = ps + 1;     /* l'adresse contenu dans ps (1600 par exemple)
                  est incrémentée de 1 * 2 = 2
                  car la dimension short = 2 octets */
```



# VII. Pointeurs

## Arithmétique des pointeurs:

### Exemple 2:

```
char c[6]= "12345", *pc = &c[0] ;  
short s[6]= {1,2,3,4,5}, *ps = &s[0] ;  
float f[6]= {1.0,2.0,3.0,4.0,5.0}, *pf = &f[0] ;
```

```
/* on suppose pc=1600 ; ps=1606 ; pf=1618 */
```

```
pc = pc + 4 ; /* pc devient 1600 + 4*1 = 1604 (char : 1 octet ) */  
ps = ps + 4 ; /* ps devient 1606 + 4*2 = 1614 (short: 2 octets) */  
pf = pf + 4 ; /* pc devient 1618 + 4*4 = 1634 (float: 4 octets) */
```

# VII. Pointeurs

## Arithmétique des pointeurs:

Opérations permises et interdites sur les pointeurs:

Opération	Permise ?	Exemple
Addition	Oui	$pa + 4 ;$
Soustraction		$pa - 2 ; pb - pa ;$
Incrémentation		$*px++ : \text{incrémente } px$ $(*px)++ : \text{incrémente } *px$
Décrémentation		$*px-- : \text{décrémente } px$ $(*px)-- : \text{décrémente } *px$
Multiplication	NON	
Division		

# VII. Pointeurs

## Récapitulatif:

### Qu'affiche, à votre avis, les 2 programmes suivants?

```
#include<stdio. h>
int main ()
{
    int x = 3 ;
    int *p ;
    p = &x ;
    printf ( "x = %d\n" , x ) ;
    *p = 4 ;
    printf ( "x = %d\n" , x ) ;
    return 0 ;
}
```

```
#include<stdio.h>
int main () {
    int x = 3 ;
    int y = 5 ;
    int *p ;
    p = &x ;
    printf ( "x = %d\n" , x ) ;
    *p = 4 ;
    printf ( "x = %d\n" , x ) ;
    p = &y ;
    printf ( "*p = %d\n" , *p ) ;
    *p = *p + 1 ;
    printf ( "y = %d\n" , y ) ;
    return 0 ;
}
```

# VII. Pointeurs: TD

1. Déterminez ce qu'affiche ce programme, exécutez-le ensuite pour vérifier:

```
#include<stdio.h>
int main ( ){
    int i = 4 ;
    int j = 10 ;
    int * p ;
    int * q ;
    p = &i ;
    q = &j ;
    printf ( "i = %d, j = %d, p = %d, q = %d\n" , i , j , *p , *q ) ;
    *p = *p + *q ;
    printf ( "i = %d, j = %d, p = %d, q = %d\n" , i , j , *p , *q ) ;
    p = &j ;
    printf ( "i = %d, j = %d, p = %d, q = %d\n" , i , j , *p , *q ) ;
    *q = *q + *p ;
    printf ( "i = %d, j = %d, p = %d, q = %d\n" , i , j , *p , *q ) ;
    q = &i ;
    printf ( "i = %d, j = %d, p = %d, q = %d\n" , i , j , *p , *q ) ;
    i = 4 ;
    printf ( "i = %d, j = %d, p = %d, q = %d\n" , i , j , *p , *q ) ;
    *q = *q + 1 ;
    printf ( "i = %d, j = %d, p = %d, q = %d\n" , i , j , *p , *q ) ;
    return 0;
}
```

# VII. Pointeurs

## Pointeurs et tableaux:

Soient les variables :

```
int x[10] ;    /* x : tableau d'éléments int */
int *px ;      /* px : un pointeur int */
int n ;        /* n : entier servant d'indice */
```

alors **&x[0]** fournit l'adresse du 1<sup>er</sup> élément de x, donc l'adresse du début de tableau.

Celle-ci peut être affectée à la variable pointeur appropriée :

```
px = &x[0] ;
```

Accès aux éléments du tableau (remplissage du tableau avec les valeurs 0 à 9) :

Via l'indice	Via le pointeur
for (n=0 ; n<10 ; n++ ) x[n] = n ;	for (n=0 ; n<10 ; n++ ) *(px+n) = n ;

# VII. Pointeurs

## Pointeurs et tableaux:

### Equivalence d'expressions:

**&x[0] et x ( le nom du tableau équivaut toujours à l'adresse du 1<sup>er</sup> élément du tableau)**

```
px = &x[0] ;           /* px reçoit l'adresse du début de tableau x */  
px = x ;               /* px reçoit l'adresse du début de tableau x */
```

<b>x</b>	OU	<b>&amp;x[0]</b>	: adresse du 1 <sup>er</sup> élément
<b>x+1</b>	OU	<b>&amp;x[1]</b>	: adresse du 2 <sup>ème</sup> élément
<b>x+2</b>	OU	<b>&amp;x[2]</b>	: adresse du 3 <sup>ème</sup> élément
...			
<b>x+i</b>	OU	<b>&amp;x[i]</b>	: adresse de l'élément d'indice i

Accès aux éléments du tableau (remplissage du tableau avec les valeurs 0 à 9) :

Via le contenu

```
for (n=0 ; n<10 ; n++ )  
    *(x+n) = n ;
```




# VII. Pointeurs

## Pointeurs et tableaux:

La programmation en C ne permet pas de retourner un tableau entier en argument à une fonction.

Cependant, vous pouvez retourner un pointeur vers un tableau en spécifiant le nom du tableau sans index. Si vous voulez retourner un tableau dans une fonction, il faut déclarer une fonction retournant un pointeur comme dans l'exemple suivant:

```
int * maFonction() {  
.  
.  
.  
}
```



Quand vous renvoyez un tableau à une fonction, vous envoyez l'adresse de la première valeur du tableau, donc les éventuelles modifications que vous faites sur votre tableau dans la fonction sont faites directement sur le tableau, pas sur une copie.  
Pour renvoyer vraiment un tableau il faut renvoyer l'adresse de la première valeur du tableau. Utile pour les chaînes de caractères

# VII. Pointeurs

## Chaînes de caractères constantes et pointeurs:

L'adresse d'une donnée est habituellement affectée à un pointeur au moyen de l'opérateur **&** ou comme contenu d'un autre pointeur. Il existe en outre une forme spéciale d'expression d'adressage, requise lorsque le pointeur correspond à l'adresse d'une chaîne constante.

Les opérations sur les chaînes constantes se déroulant toujours au moyen d'un pointeur, ce n'est pas la constante elle-même qui est transmise à la fonction, mais un pointeur vers son premier élément.

Il est possible de définir des affectations de la forme :

```
char *pstring = «toto»;
```

Le pointeur *pstring* reçoit l'adresse de la chaîne constante.

On affecte ici à la variable pointeur ***pstring*** de type ***char\**** une valeur initiale, qui est l'adresse de la chaîne «toto»

# VII. Pointeurs

## Allocation dynamique:

Lorsque l'on déclare un tableau, il est obligatoire de préciser sa taille. Cela signifie que la taille d'un tableau doit être connue à la compilation. *Alors que faire si on ne connaît pas cette taille?*


La seule solution qui se présente pour le moment est le surdimensionnement, on donne au tableau une taille très (trop) élevée de sorte qu'aucun débordement ne se produise.

Nous aimerions procéder autrement, c'est à dire **préciser la dimension du tableau au moment de l'exécution**.

## Solution

Il serait donc souhaitable de pouvoir gérer des tableaux dont la taille serait exactement adaptée au nombre de valeurs à saisir. Mais, comme celui-ci n'est connu qu'au cours de l'exécution du programme, les tableaux statiques, possédant un nombre fixe d'éléments, ne répondent pas au problème.

Au contraire, on a besoin ici **de tableaux de tailles variables**, créés lors de l'exécution du programme et **dont les dimensions varient selon les exigences du moment**. On parle alors de gestion dynamique de la mémoire, réalisée au moyen de fonctions prédéfinies.



Il est interdit en C de créer un tableau en indiquant sa taille à l'aide d'une variable. Ça peut fonctionner sur certains compilateurs mais uniquement dans des cas précis, il est recommandé de ne pas l'utiliser!

# VII. Pointeurs

## Allocation dynamique:

### Allocation de memoire avec *malloc*:

Lorsque vous déclarez un pointeur p, vous allouez un espace mémoire pour y stocker l'adresse mémoire d'un entier.

Et p, jusqu'à ce qu'on l'initialise, contient n'importe quoi. Vous pouvez ensuite faire pointer p sur l'adresse mémoire que vous voulez (une zone contenant un int, double...).

Vous pouvez demander au système d'exploitation de l'espace mémoire pour y stocker des valeurs. La fonction qui permet de réserver n octets est malloc(n). Si vous écrivez malloc(10), l'OS réserve 10 octets, cela s'appelle une allocation dynamique, c'est à dire une allocation de la mémoire au cours de l'exécution.

Si vous voulez réserver de l'espace mémoire pour stocker un int par exemple, il suffit d'appeler malloc(2), car un int occupe 2 octets en mémoire.



malloc, sizeof, free,  
NULL se trouve dans la  
bibliothèque stdlib.h.

A ne pas oublier:  
#include <stdlib.h>

# VII. Pointeurs

## Allocation dynamique:

Allocation de memoire avec ***malloc***:

L'instruction `p = malloc(2)` ne peut pas passer la compilation. Le compilateur vous dira que les types `void` et `int` sont incompatibles (incompatible types in assignment). (Ca dépend des compilateurs)

Pour votre culture générale, `void` est le type "adresse mémoire" en C.

Alors que `int` est le type "adresse mémoire d'un int".

Il faut donc dire au compilateur que vous saviez ce que vous faites, et que vous êtes sûr que c'est bien un `int` que vous allez mettre dans la variable pointée.

Pour ce faire, il convient d'effectuer ce que l'on appelle un cast, en ajoutant, juste après l'opérateur d'affectation, le type de la variable se situant à gauche de l'affectation entre parenthèses. Dans l'exemple ci-avant, cela donne :

```
int *p=(int*)malloc(2) .
```

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

int main () {
    int *p ;
    p=(int *) malloc(sizeof(int));
    *p = 28;
    printf ( "%d\n" , *p ) ;
    return 0 ;
}
```

# VII. Pointeurs

## Allocation dynamique:

### La fonction *free*:

Lorsque que l'on effectue une allocation dynamique, l'espace réservé ne peut pas être alloué pour une autre variable.

Une fois que vous n'en avez plus besoin, vous devez le libérer explicitement si vous souhaitez qu'une autre variable puisse y être stockée.

La fonction de libération de la mémoire est *free*. *free(v)* où *v* est une variable contenant l'adresse mémoire de la zone à libérer.

**A chaque fois que vous allouez une zone mémoire, vous devez la libérer !**

### Exemple :

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

int main (){
    int *p ;
    p=(int *) malloc(sizeof(int));
    *p = 28;
    printf ( "%d\n" , *p ) ;
    free(p) ;
    return 0 ;
}
```

# VII. Pointeurs

## Allocation dynamique:

### La valeur *NULL*:

Le pointeur p qui ne pointe aucune adresse a la valeur NULL.

Attention, il n'est pas nécessairement initialisé à NULL,

NULL est la valeur que, conventionnellement, on décide de donner à p s'il ne pointe sur aucune zone mémoire valide!

**Par exemple**, la fonction malloc retourne NULL si aucune zone mémoire adéquate n'est trouvée.

**Il convient, à chaque malloc, de vérifier si la valeur retournée par malloc est différente de NULL.**

### Exemple :

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

int main (){
    int *p ;
    p=(int *) malloc(sizeof(int));
    if(p==NULL ){
        exit (0);
    }
    *p = 28;
    printf ( "%d\n" , *p ) ;
    free(p) ;
    return 0 ;
}
```

# VII. Pointeurs

## Allocation dynamique:

Lors de l'allocation dynamique d'un tableau, il est nécessaire de déterminer la taille mémoire de la zone à occuper.


Par exemple, si vous souhaitez allouer dynamiquement un tableau de 10 variables de type char. Il suffit d'exécuter l'instruction `malloc(10)`, car un tableau de 10 char occupe 10 octets en mémoire.

Si par contre, vous souhaitez allouer dynamiquement un tableau de **10 int**, il conviendra d'exécuter **`malloc(20)`**, car chaque int occupe 2 octets en mémoire!

Pour se simplifier la vie, le compilateur met à notre disposition la fonction **`sizeof`**, qui nous permet de calculer la place prise en mémoire par la variable d'un type donné. Par exemple, soit T un type, la valeur **`sizeof(T)`** est la taille prise en mémoire par une variable de type T.

Si par exemple on souhaite allouer dynamiquement un int, il convient d'exécuter l'instruction **`malloc(sizeof(int))`**.

**Attention, sizeof prend en paramètre un type !**



La fonction `realloc` modifie la dimension d'un bloc de mémoire existant.  
Voici la forme générale de la fonction :

```
<Pointeur> = realloc (<pointeur>,  
    <nouvelle_dimension>);
```



# VII. Pointeurs

## Passage par références (fonctions): *cf. cours chapitre VI*

si la procédure/fonction reçoit, non pas la valeur de la donnée comme paramètre, mais son adresse, alors on parle de **transmission (ou passage) par adresse**.

Lorsque l'on veut qu'une fonction puisse modifier la valeur d'une donnée passée comme paramètre, il faut lui transmettre non pas la valeur de l'objet concerné, mais son adresse. Cette technique de transmission des paramètres est dite "passage par adresse".

La conséquence en est que la fonction appelée ne travaille plus sur une copie de l'objet transmis, mais sur l'objet lui-même (car la fonction en connaît l'adresse). Le passage des paramètres par adresse permet donc à une fonction de modifier les valeurs des variables d'autres fonctions.

```
#include <stdio.h>

void show(int *x, int *y); /*déclaration de show */

int main()
{
    int a=3, b=5; /* variables locales à main */
    printf("Valeur1:%d\tValeur2 : %d\n", a, b);
    show(&a, &b); /* appelle show: passage par ref */
    printf("Valeur1:%d\tValeur2:%d\n",a,b);
    return 0;
}

void show(int *x, int *y)/* définition de show */
{
    (*x)++ ;
    (*y)++ ;
}
```

Quel est l'affichage du programme ci-haut?

## VII. Pointeurs: TDs (suite)

2. Allocation dynamique:
  - a) Créez dynamiquement un int, affectez-y la valeur 5, affichez-le, libérez la mémoire.
  - b) En utilisant les variables de type pointeur et l'allocation dynamique, remplir un tableau d'entiers dont le nombre d'éléments est saisie à partir du clavier. Afficher le premier élément du tableau tout seul puis tout le tableau a l'aide d'une boucle.
3. Ecrire une fonction ***void swap*** qui prend en paramètre 2 variables x et y et effectue une permutation . Ecrire le programme principal correspondant.
4. Ecrire une fonction qui inverse une chaine de caractères donnée passée en paramètres à cette fonction. Cette dernière ne doit utiliser que des pointeurs. Exemple: bonjour devient roujnob

## VII. Pointeurs: TDs

5. Ecrire une fonction qui détermine le nombre de caractères d'une chaîne de caractères passée en paramètres à cette fonction. Cette dernière ne doit utiliser que des pointeurs.
6. Ecrire une fonction qui copie une chaîne de caractères dans une autre. Cette fonction ne doit utiliser que des pointeurs.
7. Ecrire un programme qui place dans un tableau `t` les `N` premiers nombres impairs, puis qui affiche le tableau. Vous accéderez à l'élément d'indice `i` de `t` avec l'expression `*(t + i)`.

## VII. Pointeurs: TDs

8. Déterminez ce qu'affiche ce programme, exécutez-le ensuite pour vérifier:

```
#include<stdio.h>
void affiche2int ( int a , int b ){
    printf ( "%d, %d\n" , a , b ) ;}
void incr1 ( int x ){
    x = x + 1 ;}
void incr2 ( int * x ){
    *x = *x + 1 ;}
void decr1 ( int * x ){
    x = x - 1 ;}
void decr2 ( int * x ){
    *x = *x - 1 ;}
int main ( ){
    int i = 1 ;
    int j = 1 ;
    affiche2int ( i , j ) ; //inst1
    incr2(&i ) ;
    affiche2int ( i , j ) ; //inst 2
    decr1(&j ) ;
    affiche2int ( i , j ) ; // inst 3
    decr2(&j ) ;
    affiche2int ( i , j ) ; // inst 4
    while ( i != j ){
        incr1 ( j ) ;
        decr2(&i ) ;}
    affiche2int ( i , j ) ; // inst 5
    return 0 ;
}
```