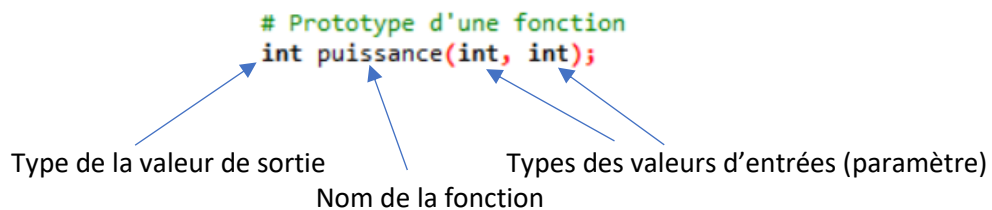


LC Fonctions Tableaux

I/ Fonctions

Une fonction offre un moyen efficace d'enfermer certains traitements dans des « boîtes noires », dont on peut ensuite se servir sans se soucier de la façon dont elle est programmée. Avec des fonctions conçues correctement, il suffit donc de les appeler en utilisant les **paramètres** demandés.

Voici la structure d'une fonction en langage C :



Paramètres : types attendus par une fonction.

Arguments : valeurs passées lorsque l'on appelle une fonction. Les types doivent bien sûr correspondre aux paramètres et dans le même ordre.

En langage C, il faut **déclarer** les fonctions avant de les utiliser selon le prototype ci-dessus. On le fait habituellement au-dessus du `main()`.

Par convention, une fonction s'écrit en lettres minuscules et l'underscore permet de séparer celles qui ont un nom composé (par exemple `calcul_salaire()`).

Voici un exemple de programme, il calcule la puissance d'un nombre entier.

```
#include <stdio.h>

// Prototype d'une fonction
int puissance(int, int);

int main()
{
    int i;

    /* Les accolades ne sont pas obligatoires si
    un bloc ne contient qu'une instruction */
    for (i = 0; i < 10; ++i) // ++i équivaut à i = i + 1
        printf("%d %d %d\n", i, puissance(2,i), puissance(-3,i));

    return 0;
}

// Elève base à la puissance n
int puissance(int base, int n) {
    int i,p;

    p = 1;
    for(i = 1; i <= n; ++i)
        p = p*base;

    return p;
}
```

Important : en langage C, il faut impérativement préciser les paramètres et le type de la valeur de retour d'une fonction : ils ne sont pas à titre informatif comme en Python.

Si une fonction ne renvoie aucune valeur, on indiquera *void* en type de retour.

Voici un exemple (pas vraiment utile, on est d'accord 😊) :

```
#include <stdio.h>

// Prototype d'une fonction ne renvoyant rien
void affiche(int);

int main()
{
    int i = 10;

    affiche(i);

    return 0;
}

// affiche n
void affiche(int n) {

    printf("%d", n);

    return;
}
```

Exercice : Ecrire une fonction permettant de donner le prix d'un ticket de cinéma en fonction de l'âge sachant qu'il est de 6 euros pour les mineurs et de 9 euros sinon.

Ne pas hésiter à reprendre le cours précédent !

II/ Pointeurs et tableaux

Un **pointeur** est une variable qui contient l'adresse d'une autre variable. Le langage C fait un grand usage des pointeurs, d'une part parce qu'ils sont parfois le seul moyen d'exprimer et que d'autre part ils contribuent à l'obtention de programmes plus compacts et efficaces que par d'autres méthodes.

Pointeurs et tableaux sont étroitement liés, cette partie montre comment les exploiter.

1/ Pointeurs et adresses

Au niveau de la mémoire, une machine classique possède un tableau de cases consécutives numérotées -ou adressées- que l'on peut exploiter individuellement ou par groupe de cases contiguës.

En général, un octet peut représenter un *char*, deux octets un entier *short* et quatre octets adjacents forment un *long*.

Un pointeur est un groupe de cases (deux ou quatre) pouvant contenir une adresse et ce quelque soit le type pointé qu'il faut préciser à la déclaration.

En langage C :

- L'opérateur unaire **&** donne l'**adresse** d'un objet.
- L'opérateur unaire ***** permet d'accéder à l'objet pointé, on parle d'opérateur de déférencement ou d'indirection.

Voici une séquence montrant l'utilisation de ces deux opérateurs :

```
int main()
{
    // Déclaration de deux entiers x et y et d'un tableau de 10 entiers (z)
    int x = 1, y = 2, z[10];
    z[0] = 5;

    int *pi; // pi est un pointeur de type int

    pi = &x; // pi pointe sur x
    printf("Adresse de x : %d \n", pi);

    y = *pi; // y vaut désormais 1
    printf("y vaut bien 1 : %d \n", y);

    *pi = 0; // x vaut désormais 0
    printf("x vaut bien 0 : %d \n", x);

    pi = &z[0]; // pi pointe sur le premier élément du tableau z
    printf("*pi vaut bien 5 : %d \n", *pi);

    return 0;
}
```

Sortie du programme

```
Adresse de x : 6487564
y vaut bien 1 : 1
x vaut bien 0 : 0
*pi vaut bien 5 : 5
```

On peut donc utiliser les pointeurs pour effectuer des opérations arithmétiques classiques en sachant que le **déférencement** a une **priorité plus forte**.

Voici quelques exemples :

```
int main()
{
    // Déclaration de pointeurs de type int
    int *pi, *qi;
    int x = 9;

    pi = &x; // La valeur pointée est donc 9
    printf("Le pointeur vaut %d \n", *pi);

    *pi = *pi + 10; // Ajoute 10 à la valeur pointée
    printf("Le pointeur vaut %d \n", *pi);

    *pi += 10; // Même chose ici
    printf("Le pointeur vaut %d \n", *pi);

    qi = pi; // qi pointe sur pi, la valeur pointée est 29
    printf("Le pointeur vaut %d \n", *qi);

    return 0;
}
```

2/ Pointeurs et arguments de fonction

Il n'y a aucun moyen de changer la valeur d'une variable de la fonction appelante. Il s'agit d'un passage par valeur : en effet, la variable est créée localement dans la fonction.

Voici un exemple en Python et en C pour comprendre :

```
int ajoute_un(int);

int main()
{
    int x = 10;
    printf("Après ajoute_un, on trouve %d \n", ajoute_un(x));
    printf("Malheureusement, x vaut toujours %d \n", x);

    return 0;
}

int ajoute_un(int x) {
    return x + 1;
}
```

Résultat

```
Après ajoute_un, on trouve 11
Malheureusement, x vaut toujours 10
-----
Process exited after 0.2396 seconds with return value 0
Appuyez sur une touche pour continuer... █
```

```
def ajoute_un(x) :
    return x + 1;

x = 10;

print("Après ajoute_un, on trouve", ajoute_un(x))
print("Malheureusement, x vaut toujours", x)
```

Résultat

```
Après ajoute_un, on trouve 11
Malheureusement, x vaut toujours 10
```

La variable x -bien qu'elle porte le même nom- a été recréée dans la fonction *ajoute_un()* et n'a strictement rien à voir avec celle déclarée dans le programme.

Comment faire pour modifier cette variable ? Python propose le mot clé horrible **global** alors qu'il suffit de passer l'adresse de la variable x en C 😊.

```
// La fonction attend un pointeur
void ajoute_un(int *);

int main()
{
    int x = 10;
    ajoute_un(&x);
    printf("x vaut enfin %d \n", x);

    return 0;
}

void ajoute_un(int *x) {
    *x += 1;
}
```

Résultat

```
x vaut enfin 11
```

```
x = 10

def ajoute_un() :
    global x
    x += 1
    return x

print("Après ajoute_un, on trouve", ajoute_un())
print("x vaut bien", x)
```

Résultat

```
Après ajoute_un, on trouve 11
x vaut bien 11
```

3/ Pointeurs et tableaux

En C, les pointeurs et les tableaux sont très liés, on peut passer très facilement de l'un à l'autre. Un **tableau** est un bloc **consécutif** d'objets créé en mémoire.

Le programme suivant crée un tableau de 10 entiers et affiche sa taille en mémoire. 40 octets est normal car la taille d'un entier est de 4 octets ici :

```
#include "stdio.h"

int main() {

    // Crée un tableau de 10 entiers
    int tab[10];

    // Indique la taille du tableau
    printf("la taille du tableau est %d octets", sizeof(tab));

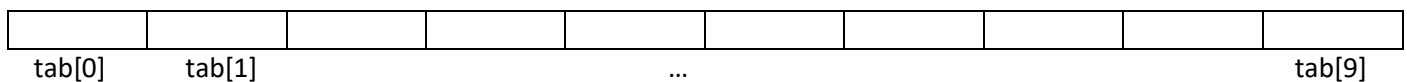
    return 0;
}
```

C:\Users\Laurent\Desktop\test.exe

```
la taille du tableau est 40 octets
-----
Process exited after 0.05602 seconds with return value 0
Appuyez sur une touche pour continuer...
```

Remarque : hormis la déclaration du type, créer un tableau en C est très proche de la syntaxe en Python.

Voici comment cela s'organise en mémoire : on définit un tableau par -dans l'exemple précédent- un bloc de 10 entiers consécutifs baptisés tab[0], tab[1] jusqu'à tab[9].



Voici un programme qui montre que les éléments sont bien contigus en mémoire :

```
#include "stdio.h"

int main() {

    // Crée un tableau de 10 entiers
    int tab[10];

    // Compteur
    int i;

    // Donne la place en mémoire de chaque élément
    for (i = 0; i < 10; i++)
        printf("l'adresse memoire de l'element %d est %p \n ", i, &tab[i]);

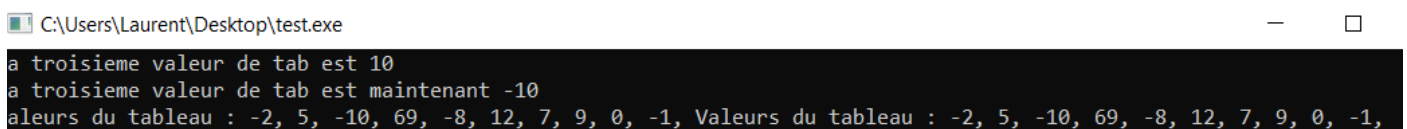
    return 0;
}
```

C:\Users\Laurent\Desktop\test.exe

```
l'adresse memoire de l'element 0 est 00000000062FDF0
l'adresse memoire de l'element 1 est 00000000062FDF4
l'adresse memoire de l'element 2 est 00000000062FDF8
l'adresse memoire de l'element 3 est 00000000062FDFC
l'adresse memoire de l'element 4 est 00000000062FE00
l'adresse memoire de l'element 5 est 00000000062FE04
l'adresse memoire de l'element 6 est 00000000062FE08
l'adresse memoire de l'element 7 est 00000000062FE0C
l'adresse memoire de l'element 8 est 00000000062FE10
l'adresse memoire de l'element 9 est 00000000062FE14
```

Il existe plusieurs méthodes pour initialiser et parcourir un tableau en C. Le programme suivant en montre quelques-unes :

```
int main() {  
  
    // Crée un tableau de 10 entiers  
    // Attention, ce sont des accolades en C, pas des crochets  
    int tab[10] = {-2, 5, 10, 69, -8, 12, 7, 9, 0, -1};  
  
    // Compteur  
    int i;  
  
    // Accès par Les [] comme en Python  
    printf("La troisieme valeur de tab est %d \n", tab[2]);  
  
    // Modification de cette troisième valeur (comme en Python)  
    tab[2] = -10;  
  
    // Accès par Les [] comme en Python  
    printf("La troisieme valeur de tab est maintenant %d \n", tab[2]);  
  
    // Affiche les valeurs du tableau  
    printf("Valeurs du tableau : ");  
    for (i = 0; i < 10; i++)  
        printf("%d, ", tab[i]);  
  
    // Déclaration d'un pointeur  
    int *val;  
  
    // Le pointeur pointe sur le premier tableau  
    val = &tab[0];  
  
    // Affichage à l'aide d'un pointeur  
    printf("Valeurs du tableau : ");  
    for (i = 0; i < 10; i++)  
        printf("%d, ", *(val++));  
  
    return 0;  
}
```



```
C:\Users\Laurent\Desktop\test.exe  
a troisieme valeur de tab est 10  
a troisieme valeur de tab est maintenant -10  
aleurs du tableau : -2, 5, -10, 69, -8, 12, 7, 9, 0, -1, Valeurs du tableau : -2, 5, -10, 69, -8, 12, 7, 9, 0, -1,
```

L'expression `*(val++)` signifie que l'on donne la valeur du pointeur `val` et que l'on déplace le pointeur d'une « case » en mémoire.

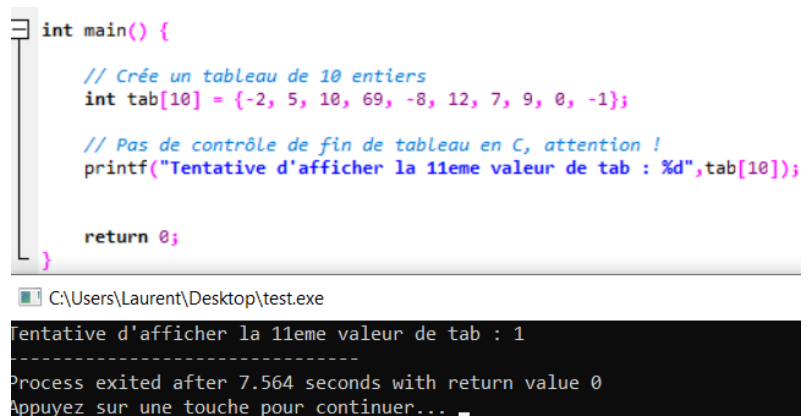
Alors, comment est-il possible que cela permette de donner les valeurs du tableau ? Pour deux raisons :

- Le **pointeur** est de type `int`, il sait qu'il faut se déplacer de 4 octets.
- Un tableau est un **bloc contigu** de données.

Exercice : dans l'exercice précédent, juste après l'instruction `val = &tab[0]`, combien vaudrait l'expression `*(val + 2)` ?

Important : il n'y a pas de contrôle de fin de tableau en C, il faut être très vigilant lors de son parcours. Le programme compilera normalement mais les résultats obtenus seront erratiques à l'exécution ...

Le programme à droite tente d'accéder au 11^{ème} élément du tableau `tab` qui ... n'existe pas et affiche un résultat inattendu.



```
int main() {  
  
    // Crée un tableau de 10 entiers  
    int tab[10] = {-2, 5, 10, 69, -8, 12, 7, 9, 0, -1};  
  
    // Pas de contrôle de fin de tableau en C, attention !  
    printf("Tentative d'afficher la 11eme valeur de tab : %d", tab[10]);  
  
    return 0;  
}
```

```
C:\Users\Laurent\Desktop\test.exe  
Tentative d'afficher la 11eme valeur de tab : 1  
-----  
Process exited after 7.564 seconds with return value 0  
Appuyez sur une touche pour continuer... █
```

Les liens entre tableaux et pointeurs sont très forts en C, on peut s'en servir pour passer les tableaux en arguments dans les fonctions. Voici un exemple :

```
#include "stdio.h"

// Ajoute un à chaque élément d'un tableau
void ajoute_un(int *, int);

int main() {

    // Crée un tableau de 10 entiers
    int tab[10] = {-2, 5, 10, 69, -8, 12, 7, 9, 0, -1};
    int taille = sizeof(tab)/sizeof(int); // Nombre d'éléments du tableau

    // On passe l'adresse du tableau et son nombre d'éléments en arguments
    ajoute_un(&tab[0], taille );

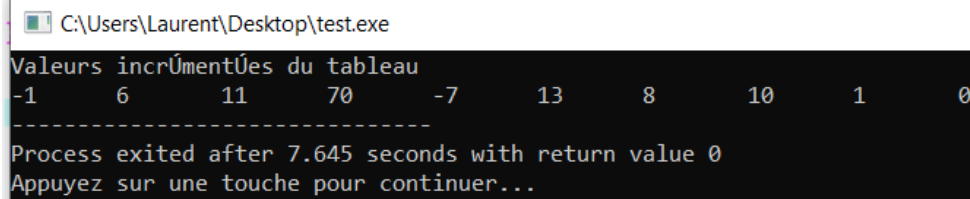
    int cpt;
    printf("Valeurs incrementées du tableau \n");

    for(cpt = 0; cpt < taille; cpt++)
        printf("%d \t", tab[cpt]);

    return 0;
}

void ajoute_un(int *lst, int taille) {
    int cpt;

    for(cpt = 0; cpt < taille; cpt++) {
        // Ajout de un à chaque élément du tableau
        *lst = *lst + 1;
        lst++; // Déplacement du pointeur
    }
}
```



```
C:\Users\Laurent\Desktop\test.exe
Valeurs incrémentées du tableau
-1    6    11   70   -7    13    8    10    1    0
-----
Process exited after 7.645 seconds with return value 0
Appuyez sur une touche pour continuer...
```

4/ Déclaration d'un pointeur

Jusqu'à présent, on a simplement utilisé des pointeurs sur des variables initialement créées. En réalité, déclarer une variable est simplement une réservation par le programme d'une partie de mémoire pour stocker la valeur souhaitée.

Le programme gère seul ce type de variables et nettoie automatiquement la mémoire lorsqu'elles ne sont plus utilisées (variables locales d'une fonction, fin de programme ...).

Variable statique : celles qui sont allouées par le compilateur dans la zone des données statiques (elles sont souvent qualifiées de **statiques**). Ce sont toutes les variables de fichier et les variables de bloc de classe statique. Elles ont une durée de vie **permanente** et sont **initialisées à 0** par défaut.

Variable automatique : Ce sont les variables de bloc de classe automatique et les paramètres formels. Elles ont une durée de vie limitée à celle du bloc où elles sont définies et n'ont aucune initialisation par défaut.

Il existe des variables situées dans une troisième zone de la mémoire, appelée le **tas**. Ce sont les variables **dynamiques**. Elles sont utilisées pour l'allocation dynamique de la mémoire.

Source : https://public.iutenligne.net/informatique/algorithme-et-programmation/priou/LangageC/197_variables__statiques__et__automatiques_.html

Dans la plupart des langages, il existe un *garbage collector* qui désalloue automatiquement ces variables. En langage C, c'est au programmeur de s'en charger !

Des variables mal gérées peuvent provoquer des fuites de mémoires et un ralentissement progressif du programme qui peut devenir inutilisable au bout d'un certain temps.

Fuites de mémoire, lien ici : <https://blog.itgs-solutions.ch/quest-ce-quune-fuite-de-memoire-sur-un-ordinateur/>

Pour allouer et désallouer de la mémoire en langage C, il existe deux instructions : *malloc()* et *free()*.

Règle : tout appel à *malloc()* doit obligatoirement correspondre à un appel à *free()*.

Voici un exemple de programme :

```
#include "stdio.h"
#include "stdlib.h"
#include "assert.h"

#define TAILLE 10

int main() {

    // Réserve de l'espace mémoire nécessaire pour 10 entiers
    // On notera la conversion en pointeurs d'entiers.
    int *tab = (int *)malloc(TAILLE*sizeof(int));

    // Test de la création de la variable
    assert(tab != NULL);

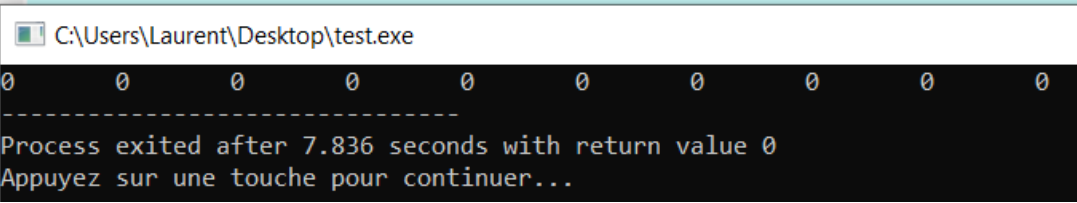
    int cpt;

    // Initialisation à zéro
    for(cpt=0; cpt < TAILLE; cpt++)
        tab[cpt] = 0;

    // Affichage des valeurs du tableau
    for(cpt=0; cpt < TAILLE; cpt++)
        printf("%d \t", tab[cpt]);

    // Libération de la ressource et annulation du pointeur
    free(tab);
    tab = NULL;

    return 0;
}
```



Quelques explications au sujet du code.

- La ligne **#define TAILLE 10** permet de définir une **constante** valant 10. Lors de la compilation, chaque occurrence de TAILLE sera remplacée par 10. Cette méthode est très fréquente en C lorsque l'on déclare des constantes.

- La ligne `int *tab = (int *)malloc(TAILLE*sizeof(int))` permet d'allouer l'espace mémoire suffisant pour y stocker 10 nombres entiers. La fonction renvoie un pointeur générique (`void *`) qu'il faut convertir en pointeur d'entiers naturels, ceci est effectué grâce à l'opérateur (`int *`).
- L'instruction `assert(tab != NULL)` permet de s'assurer que l'espace mémoire a bien été réservé et que -par conséquent- le pointeur créé n'est pas NULL.
Il faut bien comprendre que dans les années 70, l'espace mémoire était réduit et il fallait s'assurer que l'on puisse allouer assez de mémoire.
- La boucle `for(cpt=0 ; cpt < TAILLE ; cpt++) tab[cpt]` ; permet d'initialiser le tableau à zéro pour chaque élément (on aurait pu mettre d'autres valeurs) mais **allouer de la mémoire n'initialise pas le tableau avec des valeurs par défaut**.
- L'instruction `free(tab)` désalloue la mémoire, elle sera disponible pour d'autres processus. On invalide le pointeur par `tab = NULL`.

Exercice : Ecrire un programme allouant un tableau de 200 nombres flottants et l'initialiser par défaut.

Aller plus loin : Ecrire un programme allouant un nombre très élevé de mémoire sans la désallouer. On pourra utiliser une boucle infinie pour se faire.

A l'exécution, suivre la consommation en mémoire à l'aide d gestionnaire de tâches.

Attention, l'ordinateur peut planter.

5/ Pointeurs et chaînes de caractères

Une constante de type chaîne écrite de la façon suivante :

`"Je suis une chaîne"`

est un tableau de caractères. Dans la représentation interne, le tableau se termine par le **caractère nul** `'\0'` afin que les programmes puissent en détecter la fin. Par conséquent, il occupe une unité de plus en mémoire.

Ce type de constantes apparaît dans des fonctions d'affichage par exemple :

`printf("tu es un boulet \n")`

Quand on les rencontre dans un programme, on accède à de telles chaînes de caractères à l'aide d'un pointeur de caractères ; `printf()` reçoit un pointeur sur le début de la chaîne de caractères. Ainsi, on accède à une constante de type chaîne via un pointeur sur son premier élément.

Les **constantes de type chaîne ne peuvent pas être des arguments de fonctions** : il ne s'agit pas d'une copie de chaîne de caractères, seuls les pointeurs interviennent.

Important : la déclaration « `a` » et « `a` » n'a pas du tout la même signification en langage C : le premier est une chaîne de caractères et le second un caractère dont la valeur est celle de la norme ASCII.

Le programme définit de deux façons une chaîne de caractères :

```
#include "stdio.h"

int main() {

    // Création d'un tableau de caractères
    char tab_mess [] = "nous sommes des milliers";

    // Création d'un pointeur
    char *p_mess = "nous sommes des milliers";

    // Taille du tableau et du pointeur
    printf("Taille du tableau : %d\n", sizeof(tab_mess));

    // Taille du tableau et du pointeur
    printf("Taille du pointeur : %d\n", sizeof(p_mess));

    return 0;
}
```

C:\Users\Laurent\Desktop\test.exe

```
Taille du tableau : 25
Taille du pointeur : 8
```

- La variable `tab_mess` est un tableau, de taille juste suffisante pour contenir la suite de caractères et le caractère `'\0'`. Il y a en effet 24 caractères et celui de fin. On peut bien sûr changer des caractères individuels du tableau mais cette **variable représentera toujours le même espace mémoire**.
- La variable `p_mess` est un pointeur initialisé de façon à pointer sur une **constante** de type chaîne : on peut le faire pointer ailleurs mais la **chaîne n'est théoriquement pas modifiable** (comme en Python). **La taille de 8 correspond à celle d'un pointeur classique et non au nombre d'éléments.**

Le programme permet de copier une chaîne de caractères dans une autre. Cette fonction existe dans la bibliothèque standard et s'appelle `strcpy()`.

```
#include "stdio.h"

// Déclaration de strcpy
void strcpy(char *, char *);

int main() {

    // Création d'un tableau de caractères
    char init_mess[] = "nous sommes des milliers";

    // Allocation de la mémoire nécessaire
    char dest_mess[sizeof(init_mess)];

    // Copie de init_mess dans dest_mess
    strcpy(dest_mess, &init_mess[0]);

    // Affiche de dest_mess
    printf("Copie du message : %s\n", dest_mess);

    return 0;
}
```

```
// Copie de init dans dest
void strcpy(char *dest, char *init) {

    // Déclaration de l'indice
    int i;

    // Tant que l'on n'atteint pas le caractère d'arrêt
    i = 0;
    while(init[i] != '\0') {
        dest[i] = init[i];
        i++;
    }

    // On n'oublie pas le '\0' pour le caractère d'arrêt
    dest[i] = '\0';
}
```

C:\Users\Laurent\Desktop\test.exe

```
Copie du message : nous sommes des milliers
```

Quelques explications pour ce programme :

- L'instruction `char dest_mess[sizeof(init_message)]` permet de réserver en mémoire l'espace nécessaire pour accueillir le futur. On aurait pu utiliser `malloc()` / `free()` pour une allocation dynamique mais c'est inutile ici. Une simple déclaration d'un tableau vide aurait certainement fonctionné (`char dest_mess[] = {}`) mais cela n'est pas recommandé ...
- La boucle **while** dans la fonction `strcpy(char *dest, char *init)` permet de recopier chaque caractère de *init* dans *dest* **sauf le caractère d'arrêt**, d'où l'instruction `dest[i] = '\0'` à la fin.

Exercice 1 : Supprimer l'affectation `dest[i] = '\0'` et exécuter le programme. Justifier le caractère de fin qui s'affiche (qui peut être *bizarre*).

Exercice 2 : Réécrire la fonction `strcpy()` en utilisant les pointeurs plutôt que l'accès par tableau. On pensera si possible à des combinaisons d'opérateurs de déréférencement et arithmétiques et au fait que le caractère `'\0'` représente la fin de la chaîne soit comme un zéro pour un booléen.

Deux instructions suffisent dans l'esprit C 😊, je compte sur vous ...

Exercice 3 : Proposer une fonction `strcmp()` qui compare deux chaînes de caractères. Elle doit renvoyer 1 si elles sont égales et 0 le cas échéant.

A noter : pour afficher les caractères accentués (comme à ; é etc.), on peut utiliser le code ASCII étendu en hexadécimal pour les afficher. Eh oui, le langage C n'a pas de norme UTF-8 !

Voici un exemple :

```
#include "stdio.h"

int main() {

    // Message littéral
    char lit_mess[] = "Nous voilà";

    // La lettre 'à' correspond à \x85 en hexadécimal
    char hex_mess[] = "Nous voil\x85";

    // Affiche de lit_mess
    printf("Message littéral : %s\n", lit_mess);

    // Affiche de hex_mess
    printf("Message hexad\x82mal : %s\n", hex_mess);

    return 0;
}
```

C:\Users\Laurent\Desktop\test.exe

```
Message litt  ral : Nous voil  
Message hexad  mal : Nous voil  
```

Il suffit d'  crire `\x` devant la valeur de la lettre en hexad  cimal.

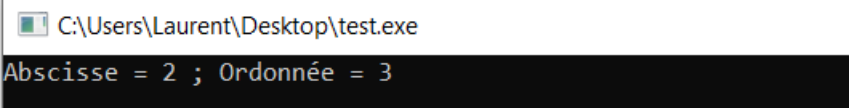
Lien vers le tableau de conversion : <http://www.table-ascii.com/>

Remarque : avec la console de Windows, il y a des bugs d'affichage et des avertissements à la compilation. On peut alors utiliser avec parcimonie cette méthode :

```
int x, y;
x=2;
y=3;

// Pas joli le printf, merci à Windows !
printf("Abscisse = %i ; Ordonnée = %i \n", x, 130, y);

return 0;
}
```



Le %c indique que l'on affiche un caractère et le 130 est le code ASCII de la lettre é.