# TP3: Pointeurs, tableaux, chaînes de caractères

MP2I Lycée Pierre de Fermat
guillaume.rousseau@ac-toulouse.fr

## Consignes

Une archive contenant des fichiers pour ce TP est à télécharger sur Cahier de Prépa.
Pensez à noter vos réponses dans un fichier texte et à **supprimer tous les exécutables** avant de compresser l'archive.

⇓⇓**A LIRE**⇓⇓

Chaque fonction que vous écrivez doit:

- être documentée correctement (tous les paramètres de la fonction apparaissent dans le commentaire);
- avoir des assertions permettant de vérifier ses préconditions;
- être testée. Pour cela, écrivez une fonction `void test()` lançant des batteries de tests sur chaque fonction. Il est même conseillé d'écrire les tests **avant** de coder la fonction: cela vous aidera à réfléchir en avance à ce que la fonction doit faire, aux éventuels cas limites qui peuvent apparaître, etc...

⇑⇑**A LIRE**⇑⇑

Vous êtes encouragé/e/s à travailler à plusieurs, à vous expliquer les exercices entre vous, mais vous devez rendre uniquement du code que vous avez écrit vous-même !

# 1 Compilation et avertissements

GCC peut afficher des erreurs et des avertissements. Une **erreur** empêche la compilation de terminer, un **avertissement** prévient l'utilisateur mais laisse la compilation se dérouler. Par exemple, si vous écrivez `printf("%f", x)` mais que `x` est de type `int`, vous obtiendrez un simple avertissement car le programme peut tout de même être compilé, mais il risque de ne pas se comporter comme on l'attend.

On s'efforcera d'écrire des programmes qui compilent **sans avertissement**, et on va même demander à GCC de signaler des avertissements additionnels, en ajoutant deux options à la compilation: `-Wall` et `-Wextra` (Warning all et Warning extra, qui ajoutent tout un tas d'avertissements divers). Par exemple:

```
gcc fortnite.c -o fortnite -Wall -Wextra
```

# 2 Adresses mémoire

Lorsque l'on exécute un programme, l'ordinateur réserve une zone dans son espace mémoire (ce que l'on appelle la RAM), pour stocker les données du programme: le code, les variables ainsi que d'autres informations.

La RAM peut être vue comme une immense liste d'octets. Par exemple, une mémoire de 4Go (Giga-octets) contiendra $2^{32} \approx 4.10^9$ octets. Ces octets sont numérotés de 0 à la valeur maximale ($2^{32} - 1$ pour 4Go). L'indice d'un octet est appelé son **adresse**.
La RAM est donc une suite de 0 et de 1 représentant les données des programmes en cours d'exécution, et en particulier leurs variables. Par exemple, si l'on déclare une variable `int bla;`, le programme réserve 32 bits, soit 4 octets, dans la RAM.

En C, **l'adresse** d'une variable désigne l'adresse de son premier octet dans la mémoire. Généralement, des variables déclarées au même endroit dans le code sont stockées consécutivement en mémoire. Par exemple, si l'on déclare deux variables `int x, y`; on s'attend à avoir `x` stockée à une adresse $a \in \mathbb{N}$ et `y` stockée à l'adresse $a+4$: `x` occupe les octets $a, a+1, a+2, a+3$ et `y` occupe les octets $a+4$ à $a+7$.

On peut accéder à l'adresse d'une variable `x` en écrivant `&x`, on dit alors qu'on a **référencé** `x`. Le résultat est un entier non-signé, mais plutôt que d'utiliser le type `unsigned int`, chaque type C aura son propre type d'adresse. Si `x` est d'un certain type `T`, alors son adresse `&x` est de type `T*` (lu "T étoile"). Les variables de type `int*`, `float*`, etc..., sont appelées des **pointeurs**. En C, pour afficher la valeur d'un pointeur avec `printf`, on utilise le format `"%p"`, qui affiche **en hexadécimal**.

## Exercice 1: Adresses et pointeurs

**Q1.** Créez un fichier C et recopiez-y le code suivant:

```c
#include <stdio.h>
int main(){
  int x = 6;
  int y = 98;
  int* px = &x; // variable de type "pointeur d'entier"
  int* py = &y;

  printf("L'adresse de x est %p\n", px);
  printf("Celle     de y est %p\n", py);
  return 0;
}
```

Compilez et exécutez: de combien diffèrent les adresses de `x` et `y` ? Cela semble t-il logique (n'oubliez pas qu'on est en hexadécimal)?

**Q2.** Et en utilisant le type `long int` à la place ?

Dans l'exercice précédent, la variable `px` a pour valeur l'adresse de `x`. On dit alors que `px` **pointe vers** `x`.

L'opérateur inverse de `&` est l'opérateur `*`, appelé opérateur de **déréférencement**: si on a une variable `p` de type `T*`, alors `*p` est la valeur de type `T` obtenue en lisant les données stockées à partir de l'adresse numéro `p`. En particulier, pour n'importe quelle variable `x`, on a `*(&x) == x`: si l'on lit le contenu écrit à l'adresse de x, on trouve la valeur de x !

## Exercice 2: Déréférencement

Recopiez le code suivant:

```c
#include <stdio.h>
int main(){
  int x = 6;
  int* px = &x; // adresse de x
  int y = *px; // valeur stockée à partir de l'octet pointé par px
  printf("y vaut %d\n", y);
  return 0;
}
```

**Q1.** Exécutez ce code et vérifiez que `y` vaut bien 6.

**Q2.** Ajoutez une ligne pour modifier la valeur de `x` après l'affichage. La variable `y` est-elle affectée par cette modification ?

En C, la valeur `NULL` est le ***pointeur nul***. Cette valeur particulière est utilisée pour encoder "un pointeur vers rien", on ne peut pas la déréférencer. Par exemple, certaines fonctions de la librairie standard C renvoient un pointeur: lorsqu'elles renvoient `NULL`, c'est le signe qu'il y a eu une erreur. En pratique, `NULL` vaut 0 sur la plupart des systèmes, (mais il n'y a aucune garantie): aucune donnée ne sera stockée à l'adresse 0 !

Lorsque l'on essaie d'accéder à une zone de la mémoire à laquelle on n'a pas le droit, par exemple parce qu'elle ne correspond pas à une variable déclarée, le programme s'arrête et affiche une ***segmentation fault***, ou ***erreur de segmentation***. Vous risquez de voir cette erreur très souvent à partir de maintenant. Il est important de découper vos programmes en ***fonctions*** et de munir ces fonctions d'***assertions*** car cela vous aidera à mieux localiser les bugs.

## Exercice 3: Segfault

Recopiez le code suivant:

```c
#include <stdio.h>

int main(){
  int* p = NULL;
  printf("La valeur pointée par le pointeur NULL est %d\n", *p);
  return 0;
}
```

**Q1.** Qu'affiche le programme à l'exécution ?

**Q2.** Modifiez le programme pour qu'il essaie d'afficher le contenu de l'adresse numéro 25 de votre RAM. Que se passe-t-il ?

***Dans la suite, lorsqu'une fonction prend en argument un pointeur, et qu'elle tente de déréférencer ce pointeur, on doit s'assurer au début de la fonction que le pointeur n'est pas nul, avec un*** `assert(ptr != NULL);`.

On rappelle qu'en C, les fonctions marchent par passage par valeur, si bien qu'une fonction C ne peut pas modifier ses arguments: lors d'un appel de fonction, les valeurs des arguments sont dupliquées et écrites dans les zones mémoires des paramètres.
Les pointeurs permettent de contourner cette limitation. En effet, si l'on veut écrire une fonction qui modifie une variable, il suffit de lui faire prendre en paramètre **l'adresse** de la variable à modifier !

## Exercice 4: Passage par adresse

**Q1.** Recopiez le code suivant:

```c
// ajoute 1 à l'entier pointé par p
void incrementer(int* p){
  assert(p != NULL);
    *p = *p + 1;
}

int main(){
  int x = 3;
  printf("Avant: %d\n", x);
  incrementer(&x);
  printf("Après: %d\n", x);
}
```

Compilez et exécutez: vérifiez que la valeur de `x` a bien été incrémentée.

**Q2.** Dans le même fichier, écrivez une fonction `void echange(float* pa, float* pb)` en suivant le même principe, qui permet d'échanger le contenu de deux zones mémoires correspondant à des flottants. Exemple d'utilisation:

```c
float x = 3.0, y = 1.5;
echange(&x, &y);
assert (x == 1.5 && y == 3.0);
```

**Q3.** Écrivez maintenant une fonction ayant la spécification suivante:

```c
/* Résout l'équation quadratique aX^2 + bX + c = 0
   et renvoie le nombre de solutions réelles (0, 1 ou 2).
   Stocke également la ou les racines réelles dans les zones
   pointées par x1 et x2. */
int quad_solve(float a, float b, float c, float* x1, float* x2);
```

La dernière question de l'exercice précédent monte une autre utilisation intéressante des pointeurs: on peut les utiliser pour "renvoyer" **plusieurs** valeurs dans une fonction. Cette utilisation est très courante dans les fonctions de la librairie standard de C: l'utilisateur fournit des pointeurs à la fonction, et la fonction va y écrire le résultat, ne renvoyant qu'un code donnant des informations sur le résultat.

### Exercice 5: Valeur de retour de scanf

Pour l'instant, nous avons utilisé `scanf` comme si elle ne renvoyait rien. En réalité, la fonction renvoie **le nombre de valeurs lues**, en plus de stocker les valeurs lues dans les adresses fournies. Par exemple, considérons le code suivant:

```
1 int x, y;
2 int nb_lus = scanf("%d %d", &x, &y);
```

La variable `nb_lus` peut valoir:

- 2 si l'utilisateur a rentré deux nombres correctement (par exemple "`55 12`"). Dans ce cas, `x` vaut 55 et `y` vaut 12.
- 1 si l'utilisateur a rentré un nombre puis n'importe quoi (par exemple "`55 bla`"). Dans ce cas, `x` vaut 55 mais `y` n'a pas été modifiée.
- 0 si l'utilisateur a rentré n'importe quoi dès le début (par exemple "`m.p.2`"). Dans ce cas, ni `x` ni `y` n'a été modifiée.

**Q1.** Écrire une fonction `bool lire_int(int* p)` qui demande à l'utilisateur de rentrer un entier. En cas de succès, la fonction stockera la valeur lue à l'adresse pointée par `p` et renverra `true`; en cas d'échec elle renverra `false`.

# 3 Tableaux

Un **tableau** est une suite de cases mémoires successives. Un tableau en C ne peut stocker qu'un seul type d'élément. On peut avoir des tableaux de float, des tableaux d'int, mais pas des tableaux mélangeant les deux.

En C, on peut déclarer un tableau d'entiers de la manière suivante:

```
1 int nom_tableau[50];
```

Cette ligne de code a pour effet de réserver 50 cases mémoires successives de 4 octets chacune.

On peut accéder à la $i$-ème case d'un tableau `tab` avec la syntaxe `tab[i]`. Les tableaux sont indexés à partir de 0, donc si `tab` est un tableau de taille $n$, on peut lire et écrire dans `tab[0]` jusqu'à `tab[n-1]`.

On peut initialiser un tableau directement à la déclaration:

```
1 int tab[3] = {12, 63, 268};
```

ou bien seulement déclarer le tableau, puis le remplir (avec une boucle par exemple):

```
1 // creer un tableau contenant les 500 premiers carrés
2 int carres[500];
3 for (int i = 0; i < 500; i++){
4     carres[i] = i*i;
5 }
```

On ne peut pas, dans le cas général, connaître la longueur d'un tableau. Ainsi, lorsqu'une fonction manipule un tableau, elle doit prendre en entrée la taille de ce tableau. De plus, on ne peut pour l'instant créer que des tableaux dont la taille est une constante numérique:

```
1 int tab[48]; // OK
2 int n = 23;
3 int tab2[n]; // PAS OK: n est une variable, pas une constante
```

Pour l'instant, on ne peut pas non plus faire de fonction qui crée et ***renvoie*** un tableau, on verra pourquoi en cours, et surtout comment y remédier. Pour l'instant, on donne les tableaux en paramètre des fonctions, ce qui permet de les modifier.

On peut traiter les tableaux comme des pointeurs. Plus précisément, si l'on a défini `int tab[20];`, alors `tab` se comporte comme un pointeur vers sa propre première case. Inversement, si `int* p` est un pointeur, alors écrire `p[i]` lit dans la mémoire $i$ cases après l'adresse pointée par `p`.

Donc, pour faire une fonction qui prend en entrée un tableau, on utilise simplement le type pointeur.

## Exercice 6: Tableaux

**Q1.** Lisez et recopiez le code suivant dans un nouveau fichier:

```
1  #include <stdio.h>
2
3  /* calcule la somme des éléments de t, tableau de taille n */
4  int somme(int* tab, int n){
5    int res = 0;
6    for (int i = 0; i < n; i++){
7        res = res + tab[i];
8    }
9    return res;
10 }
11
12 int main(){
13     int t[6] = {2, 3, 5, 1, -3, 3};
14     printf("%d\n", somme(t, 6));
15 }
```

Compilez et exécutez.

**Q2.** Sur le même modèle, écrivez une fonction qui permet d'afficher les cases d'un tableau d'entiers.

**Q3.** Écrivez une fonction qui prend en entrée un tableau $T$ et une taille $n$, et qui remplit les $n$ premières cases de $T$ avec des entiers aléatoires entre $-10$ et 10.

**Q4.** Écrivez une fonction qui prend en entrée deux tableaux flottants $T$ et $U$, ainsi qu'une taille $n$, et qui remplit les $n+1$ premières cases de $U$ en stockant dans chaque case $U[i]$ la somme des $i$ premières cases de $T$. Attention au décalage: la case $U[0]$ doit contenir 0, la case $U[1]$ doit contenir $T[0]$, la case $U[2]$ doit contenir $T[0]+T[1]$, etc...

**Préprocesseur, macro #define** Une **directive préprocesseur** est une instruction C qui est traitée directement par le préprocesseur, la première étape de compilation. Les directives préprocesseurs commencent toujours par #, et nous en avons déjà rencontré une: l'inclusion de librairie. Lorsque l'on écrit `#include<bla.h>`, le préprocesseur remplace cette ligne par l'intégralité du code écrit dans le fichier `bla.h`.

Une autre directive préprocesseur importante est `#define`, qui permet de créer des **macros**, c'est à dire des bouts de code réutilisables auxquels on donne des noms.

Par exemple, si l'on écrit `#define BLA 1000` au début d'un programme, lors du passage du préprocesseur, toutes les occurrences de `BLA` sont remplacées par 1000. Par exemple, voici un programme C écrit par un humain (à gauche), et le programme que verra le compilateur une fois les directives préprocesseur exécutées (à droite):

```c
#define N 1000
int main(){
  int n = N;
  N = 50;
  printf("%d\n", N);
}
```

```c
int main(){
  int n = 1000;
  1000 = 50; // ne compile pas
  printf("%d\n", 1000);
}
```

Le programme ci-dessus serait donc invalide. On voit que `N` n'est **pas une variable**, c'est plutôt comme si l'on donnait un deuxième nom à l'entier 1000.

En pratique, pour le moment, nous allons utiliser les macros pour définir une taille maximale autorisée pour les tableaux. Par exemple:

```c
#define N 10000

/* Demande un entier n à l'utilisateur, et remplit les n premières
   cases d'un tableau t avec les carrés parfaits. */
int main(){
  int t[N];
  int n;
  scanf("%d", &n);
  assert(n <= N);
  for (int i = 0; i < n; i++){
    t[i] = i*i;
  }
}
```

Le programme ci-dessus réserve systématiquement 100000 cases pour `t`, mais n'utilise que les `n` premières. On peut alors faire comme si `n` était la taille de `t`, et ignorer les cases non-utilisées. C'est évidemment une méthode peu économe car il se peut que $n$ soit bien plus petit que $N$, auquel cas la quasi-totalité du tableau a été réservée pour rien. On verra plus tard en cours comment réserver de la mémoire de manière plus adaptée dans ce type de situation.

## Exercice 7: Jeu du tri

Le but de ce programme est d'implémenter le **jeu du tri**. Le principe est le suivant: le programme affiche un tableau à l'utilisateur, et celui-ci doit le trier dans l'ordre croissant en un minimum de coups. Pour cela, il a le droit d'échanger successivement les valeurs de deux cases adjacentes du tableau. Un coup invalide (qui échange une case avec elle-même, ou qui échange deux cases non-adjacentes) ne doit pas être pris en compte.

Une partie de jeu ressemblera à :

```
Bienvenue au jeu du tri. Choisissez la taille du tableau: 4
Voici votre tableau:
[8, 12, 5, 21]
Rentrez les indices à échanger: 0 2
Échange invalide: les cases doivent être adjacentes.
Rentrez les indices à échanger: 1 2
[8, 5, 12, 21]
Rentrez les indices à échanger: 0 1
[5, 8, 12, 21]
Bravo !! Vous avez gagné en 2 coup(s) !
```

Vous pouvez faire cet exercice de zéro, ou bien partir du fichier `jeu_tri.c` de l'archive, qui contient une structure basique à compléter.

## Exercice 8: (optionnel) Mémoire

Le but de cet exercice est d'écrire un programme qui simule une petite mémoire RAM. Ce programme possèdera un tableau en variable globale, et proposera en boucle à l'utilisateur le menu suivant:

```
Que voulez vous faire ?
1. Accéder à une valeur    2. Stocker une valeur
3. Faire une addition      4. Quitter le programme
```

Dans le cas 1, l'utilisateur rentre une adresse mémoire, et le programme affiche la valeur de la case mémoire correspondante Dans le cas 2, l'utilisateur rentre une adresse mémoire puis une valeur, et le programme stocke la valeur dans la case correspondante Dans le cas 3, l'utilisateur donne trois adresses $a, b, c$, et le programme stocke la somme des valeurs des cases $a$ et $b$ dans la case $c$.

De plus, le programme gardera en mémoire les cases n'ayant jamais été initialisées, afin d'empêcher que l'on puisse les lire. Par exemple, si la première commande de l'utilisateur est d'accéder à la case mémoire 1, sans jamais avoir écrit dedans, le programme refusera de lire le contenu. Enfin, au tout début du programme, l'utilisateur rentre une taille de mémoire maximale $n \in [\![1, 1024]\!]$, et le programme devra toujours s'assurer que les adresses rentrées par l'utilisateur sont valides.

Exemple d'exécution:

```
Choisir la taille de la mémoire: 512
(ici afficher les choix)
Choix: 2
Rentrer l'adresse mémoire puis la valeur à stocker: 13 85
85 stocké dans la case 13
(ici afficher les choix)
Choix: 1
Rentrez l'adresse mémoire: 13
La case 13 contient 85
(etc...)
```

Comme pour l'exercice précédent, vous pouvez faire cet exercice de zéro, ou partir du fichier `memory.c` de l'archive.

# 4 Chaîne de caractères

Le type `char` sert non seulement à représenter des entiers signés 8 bits, mais aussi des caractères littéraux (comme son nom l'indique), en associant à chaque symbole un nombre entier entre 0 et 127. Les `char` dont le bit de poids fort est 1 ne représentent pas un caractère valide.

## Exercice 9: Codes ASCII

Les fichiers `code_to_char.c` et `char_to_code.c` de l'archive permettent de traduire un entier $n \in [\![0, 127]\!]$ en le caractère $c$ correspondant, et inversement.

**Q1.** Compilez ces programmes (donnez leur des noms distincts !), et donnez les entiers correspondant aux caractères suivants en hexadécimal, décimal, et binaire (n'oubliez pas que la traduction hexadécimal - binaire est triviale).

a) Les lettres majuscules 'A' - 'Z', les chiffres '0'-'9'

b) Les parenthèses, et le retour à la ligne `'\n'` (vous devez simplement appuyer sur entrée dans le terminal, pas taper "\n")

**Q2.** Donnez également les caractères correspondant aux entiers `0x61` à `0x7a` (traduisez-les en décimal avant de les taper dans le terminal).

Ainsi, en C, lorsque l'on écrit `'a'`, c'est ***exactement*** comme si l'on écrivait `97` ou `0x61`. En particulier, on peut faire `'a' + 1` et obtenir `'b'`.

Vous pouvez consulter en ligne la table de correspondance caractère/code, que l'on appelle la **table ASCII** (l'ASCII étant le nom de cet encodage). Attention, il faut utiliser des guillemets ***simples*** pour les caractères, et des guillemets ***doubles*** pour les chaînes de caractères. En C, `'a'` et `"a"` ne sont pas équivalents !

Dans un programme C, on représente le texte par des tableaux de caractères. Une **chaîne de caractères** est un tableau de caractères dont un des octets est `0x00`, aussi appelé le **caractère nul**. Ce caractère **marque la fin** des chaînes de caractères, et permet de différencier la longueur d'une chaîne de la taille mémoire qui lui a été allouée. Par exemple, si l'on considère le code suivant:

```
1 char bonjour[20] = "Bonjour !";
```

Le tableau `bonjour` contient 20 cases, dont seules les 10 premières sont réellement utilisées. Les 9 premières cases contiennent les caractères 'B', 'o', 'n', 'j', 'o', 'u', 'r', ' ', '!', et la 10ème case (`bonjour[9]`) contient le caractère nul, marquant la fin de la chaîne. Notons que quand on écrit une chaîne de caractère directement dans le code, le caractère nul est ajouté automatiquement, mais il faut penser à lui réserver une case. Il aurait été incorrect d'écrire `char bonjour[9] = "bonjour";`.

En code, on note `'\0'` le caractère nul, plutôt que `0x00`.

## Exercice 10: Message brouillé

Lisez le code du fichier `string.c` de l'archive et, sans l'exécuter, écrivez ce qu'il devrait afficher. Exécutez le programme pour vérifier.

Contrairement à un tableau quelconque, on peut manipuler une chaîne de caractères sans connaître sa longueur au préalable, en utilisant le caractère nul comme condition de fin. Par exemple, pour épeler une chaîne de caractère et en afficher chaque caractère sur une ligne:

```c
void epeler(char* s){
  assert(s != NULL);
  int i = 0;
  while (s[i] != '\0'){
      printf("%c\n", s[i]);
      i++;
  }
}
```

```c
// Version avec une boucle for

void epeler(char* s){
  assert(s != NULL);
  for (int i = 0; s[i] != '\0'; i++){
      printf("%c\n", s[i]);
  }
}
```

Pour qu'une chaîne de caractères soit bien formée, elle **doit** se terminer par le caractère nul. Le format pour afficher et scanner les chaînes de caractère avec `printf` et `scanf` est `%s`, et l'instruction `scanf("%s", s)`, lit dans l'entrée du terminal jusqu'à rencontrer un saut de ligne, un espace, ou tout autre caractère blanc.

## Exercice 11: Fonctions sur les chaînes de caractères

Pour toutes les fonctions suivantes, on suppose que les entrées des fonctions sont des chaînes de caractères terminées par le caractère nul, et que les tableaux ont été réservés avec assez d'espace pour que les fonctions s'exécutent correctement. Lorsqu'une fonction doit créer une nouvelle chaîne de caractères, on prendra soin de toujours la finir avec un caractère nul.

**Q1.** Écrire une fonction `int str_len(char* s)` qui renvoie la longueur de $s$, c'est à dire son nombre de caractères (caractère nul **exclus**).

**Q2.** Écrire une fonction `int occurrences(char* s, char c)` qui renvoie le nombre de fois où le caractère $c$ apparaît dans la chaîne $s$.

**Q3.** Écrire une fonction `int str_cmp(char* s1, char* s2)` qui renvoie 0 si les deux chaînes en entrée sont égales, un entier strictement négatif si $s_1$ est plus petite dans l'ordre alphabétique, et un entier strictement positif si $s_2$ est plus petite. L'ordre des lettres est simplement l'ordre naturel `<=` sur le type `char`.

**Q4.** Écrire une fonction `void str_cpy(char* dst, char* src)` qui prend la chaîne écrite dans la source `src` et la recopie dans la destination `dst`. Par exemple:

```c
char str1[30] = "Bonjour";
char str2[40] = "Au revoir";
str_cpy(str1, str2); // copie str2 dans str1
printf("%s\n", str1); // affiche "Au revoir"
```

**Q5.** Écrire une fonction `void str_cat(char* dst, char* src)` qui concatène (c'est à dire rajoute à la suite) la chaîne `src` à la fin de la chaîne `dst`. Par exemple:

```c
char str1[50] = "Bonjour";
char str2[20] = " tout le monde !";
str_cat(str1, str2);
printf("%s\n", str1); // Affiche "Bonjour tout le monde !"
```

En réalité, les fonctions `strlen`, `strcpy`, `strcat` et `strcmp` existent en C, dans la librairie `string.h` (notez qu'elles ne s'appellent pas exactement comme dans l'exercice, il n'y a pas de tiret du bas). Vous devez connaître l'existence de cette librairie, ainsi que de ces 4 fonctions, et être capable de les recoder !

La fonction `printf` affiche les chaînes de caractères en affichant chacun de leurs caractères, jusqu'à tomber sur un caractère nul. Pour une chaîne mal-formée, le programme essaiera de lire dans la mémoire jusqu'à trouver un octet nul, au delà de la mémoire réservée pour la chaîne, et même au delà de la mémoire réservée pour le programme, causant une erreur de segmentation. D'où l'importance de toujours créer des chaînes bien formées !

## Exercice 12: Dysfonctionnements

**Q1.** Le programme scan_string.c dans l'archive du TP demande une entrée à l'utilisateur, et écrit dans le terminal ce que l'utilisateur a entré. Lisez le code, exécutez-le, et testez d'écrire des mots, des phrases, etc... pour comprendre comment le programme fonctionne. Essayez d'écrire un mot de plus de 20 lettres: que se passe-t'il ? A votre avis, en quoi ce comportement peut-il être exploité pour faire dysfonctionner un programme ?

**Q2.** Compilez et exécutez le fichier coucou_bonjour.c dans l'archive fournie avec le TP. Décrivez son comportement et, en faisant un schéma du contenu de la mémoire, formulez une hypothèse sur pourquoi il se comporte de la sorte.

En C, on peut créer des tableaux à plusieurs dimensions. Par exemple, en tapant `float grille[30][40];`, on crée un grille 2D (une matrice): `grille` est un tableau de 30 cases, où chaque case est un autre tableau de 40 cases. Ainsi, on peut accéder aux différents flottants de la grille avec `grille[i][j]` pour $i \in [\![0, 29]\!]$ et $j \in [\![0, 39]\!]$.

## Exercice 13: Randonnée

On considère une région montagneuse représentée par une grille $G$ carrée de dimensions $n \times n$. Chaque case $G[i][j]$ contient un entier représentant l'altitude du terrain dans une unité arbitraire. Voici un exemple de grille d'altitude:

| | | | | |
|---|---|---|---|---|
| 0 | 2 | 4 | 5 | 5 |
| 1 | 1 | 1 | 1 | 4 |
| 5 | 2 | 6 | 4 | 3 |
| 5 | 2 | 4 | 5 | 2 |
| 4 | 3 | 4 | 3 | 1 |

Une randonneuse se situant initialement sur la case en haut à gauche, de coordonnées $(0, 0)$, veut rejoindre la case en bas à droite, de coordonnées $(n-1, n-1)$. Attention, on compte les coordonnées comme dans une matrice: la coordonnée $(i, j)$ se situe à la $i$-ème ligne et à la $j$-ème colonne. La randonneuse ne peut se déplacer que d'une case à la fois, vers le haut, le bas, la gauche ou la droite, et ne peut pas monter ou descendre de plus qu'une unité en un mouvement.

On représente un chemin par une chaîne de caractères, constituée de $H, B, D, G$ représentant des mouvements respectifs vers le haut, le bas, la droite, et la gauche. Par exemple, sur la grille représentée ci-dessus, le chemin `BDBBBDHDHDBB` permet d'atteindre la case $(4, 4)$.

Dans le code, on supposera les grilles de taille au plus $n_{\max} = 100$. On pourra donc utiliser des grilles de taille $n_{\max} \times n_{\max}$, et n'utiliser que les $n \times n$ premières cases.

**Q1.** Lire le fichier `randonnee.c` de l'archive pour vous familiariser avec le code. Compiler et vérifier que le programme s'exécute correctement.

**Q2.** Écrire une fonction `bool valide(int n, int i, int j)` vérifiant que $0 \leq i, j < n$.

**Q3.** Écrire une fonction `void deplacer(char c, int* i, int* j)` prenant en entrée un caractère, ainsi que deux pointeurs vers les coordonnées $(i, j)$ de la randonneuse. La fonction modifiera les coordonnées selon le caractère donné. Par exemple, si $(i, j) = (5, 6)$, alors l'appel `deplacer('H', &i, &j)` aura déplacé la randonneuse d'une case vers le haut, changeant $(i, j)$ en $(4, 6)$.

**Q4.** Écrire une fonction `bool valider_chemin(char* chemin, int G[N_MAX][N_MAX], int n)` vérifiant que le chemin donné est valide dans $G$ et permet bien d'atteindre la case $(n-1, n-1)$. Votre fonction devra aussi vérifier que le chemin ne sort pas des bornes de la grille.

On s'intéresse maintenant à un algorithme randomisé permettant de trouver un chemin dans une grille, de la manière suivante:

1. on part de $(0, 0)$;
2. on choisit aléatoirement une des cases accessibles et on s'y rend;
3. on réitère jusqu'à avoir atteint la case $(n-1, n-1)$ ou au bout de $5n^2$ étapes.

Si l'algorithme effectue $5n^2$ étapes sans atteindre l'arrivée, il échoue. Notons que l'algorithme peut tout à fait échouer alors qu'il existe un chemin.

**Q5.** Implémenter la méthode gloutonne:

```
/* Stocke dans chemin un chemin de (0, 0) à (n-1, n-1) dans G,
   et renvoie true si un tel chemin a été trouvé, false sinon. */
bool recherche_chemin(int G[N_MAX][N_MAX], char* chemin, int n);
```

**Attention**, l'algorithme peut repasser plusieurs fois au même endroit, et peut produire une chemin de taille $5n^2$ au plus. Il faudra donc fournir en argument un tableau `chemin` avec assez d'espace (sans oublier le caractère nul).

Plus tard dans l'année, nous verrons des algorithmes permettant de trouver efficacement des chemins, et même de trouver des chemins les plus courts possibles.

## Exercice 14: Exercice libre

Écrire un programme faisant intervenir des tableaux, des chaînes de caractères, des pointeurs. Quelques suggestions si vous n'avez pas d'inspiration:

- Un jeu de bataille, de puissance 4, etc...
- Un générateur de mots ressemblant à du français (ou à votre langue préférée ayant un alphabet ou un syllabaire), puis un générateur de pseudo-phrases.
- Un programme qui calcule et affiche le triangle de Sierpinski, la spirale d'Ulam, une partie de l'ensemble de Mandelbrot, etc...

Pensez à bien diviser votre programme en fonctions, afin de produire du code organisé et lisible !