TP3 : Pointeurs, tableaux, chaînes de caractères

MP2I Lycée Pierre de Fermat

Consignes

Vous pouvez télécharger sur Cahier de Prépa une archive contenant des fichiers C pour ce TP.

Il y aura deux séances sur ce TP. Vous devez avoir fini au moins jusqu'à l'exercice 7 avant mercredi prochain, et vous devez rendre le TP avant mercredi 25/10 à 22h00.

N'oubliez pas de noter vos réponses dans un fichier texte ni de supprimer tous les exécutables avant de compresser l'archive. Certains exercices sont **optionnels**, généralement ils viennent clore une section du TP, sont un peu plus durs. Il est conseillé d'y revenir une fois que vous avez fini la partie principale du TP. Il est fortement conseillé de les faire si vous êtes à l'aise, et de vous y essayer peu importe votre niveau.

Vous devez commenter ***toutes*** vos fonctions, et mettre des assertions lorsque c'est nécessaire. Si une question vous demande d'écrire une fonction, il faudra automatiquement rajouter dans le main du programme de quoi tester cette fonction, même si la question ne le précise pas.

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 !

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\n", x)` mais que `x` est de type int, vous obtiendrez un simple avertissement car le programme peut tout de même être compilé.

En pratique, on veut toujours que nos programmes compilent **sans aucun avertissement**. On va même être plus stricts et demander à GCC de signaler des avertissements additionnels. Pour cela, il faut rajouter deux options à la compilation : `-Wall` et `-Wextra` (Warning all et Warning extra). Par exemple :

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

Exercice 1.

Jouons à un jeu des 8 erreurs. Depuis l'archive du TP, copiez-collez le fichier "plein_de_bugs.c". Comme son nom l'indique, il ne fait pas exactement ce qu'il est sensé faire. Trouvez et corrigez les différentes erreurs en utilisant `-Wall` et `-Wextra`, jusqu'à ce que la compilation n'affiche

aucun warning et que le programme s'exécute parfaitement. Notez les erreurs trouvées, ainsi que les corrections appliquées (vous devez en trouver 8).

Adresses mémoire

Lorsque l'on exécute un programme, l'ordinateur réserve une zone mémoire dans la RAM pour stocker les données du programme : le code, les valeurs des variables ainsi que d'autres informations diverses.

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**.

Par extension, en C, **l'adresse** d'une variable désigne l'adresse de son premier octet. Généralement, des variables déclarées au même endroit dans le code sont stockées consécutivement en mémoire, mais ce n'est pas une règle absolue.

On peut accéder à l'adresse d'une variable `x` en écrivant `&x`. Ici, l'opérateur `&` s'appelle opérateur de **référencement**. Si `x` est d'un certain type `T`, alors `&x` est de type `T*`. Les valeurs de type `int*`, `float*`, et généralement `T*` pour un type `T` quelconque, sont appelées des **pointeurs**. En C, pour afficher un pointeur avec `printf` on utilise le format `"%p"`, ce qui affichera la valeur du pointeur en hexadécimal.

Exercice 2.

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

```c
int main(){
  int x = 6;
  int y = 98;
  int* px = &x;
  int* py = &y;

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

Compilez et exécutez : de combien diffèrent les adresses de `x` et `y` ? Cela semble-t'il logique ?

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 un pointeur `T* p`, alors `*p` est la valeur de type `T` obtenue en lisant les données stockées à partir de l'adresse numéro `p`.

Exercice 3.

Recopiez le code suivant dans un nouveau fichier :

```
1 int main(){
2   int x = 6;
3     int* px = &x; // adresse de x
4
5   int y = *px; // valeur stockée à partir de l'octet pointé par px
6
7   printf("y vaut %d\n", y);
8   return 0;
9 }
```

Exécutez-le et vérifiez que `y` vaut bien 6.

En C, la valeur `NULL` représente 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 renvoie un pointeur, si elles renvoient `NULL` c'est qu'il y a eu une erreur.

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 instantanément 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 segmenter vos programmes en ***fonctions*** et de munir ces fonctions d'***assertions*** car cela vous aidera à mieux localiser les bugs.

Exercice 4.

Essayez de compiler et d'exécuter le code suivant. Que se passe-t'il ? Pourquoi ?

```
1 #include <stdio.h>
2
3 int main()
4 {
5   int* p = NULL;
6   printf("La valeur pointée par le pointeur NULL est %d\n", *p);
7
8     return 0;
9 }
```

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 une assertion.

On rappelle qu'en C, les fonctions marchent par passage par valeur, si bien qu'une fonction C ne peut pas modifier ses arguments. Les pointeurs permettent de contourner cette limitation. En effet, si on donne à une fonction un pointeur `T* p` comme argument, la fonction ne connaît pas l'adresse à laquelle `p` est stockée, mais elle connaît la valeur de `p` et peut aller écrire en mémoire à cet endroit.

Exercice 5.

Question 1. 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 !

Question 2. Dans le même fichier, en suivant le même principe, écrivez une fonction `void echange(float* pa, float* pb)` qui permet d'échanger le contenu de deux zones mémoires correspondant à des flottants.

Question 3. 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 dans les zones pointées
   par x1 et x2.
    */
int quadratic_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. Par exemple, `scanf` stocke les entiers lus dans les cases mémoires pointées par les arguments qu'on lui fournit, et elle ne renvoie qu'un code, qui indique le nombre d'objets distincts lus (ou bien un code EOF (End Of File), que l'on verra lorsque l'on manipulera des fichiers en C).

Exercice 6. **Optionnel** Nous avons vu dans un TP précédent que si l'on exécute `scanf("%d", &x)` dans un programme, mais que l'on écrit autre chose qu'un entier dans le terminal (par exemple des lettres ou de la ponctuation), alors le texte du terminal n'est pas lu et scanf ne modifie pas son entrée. Le but de cet exercice est d'écrire une fonction ayant la spécification suivante :

```
1 /* Lit un entier dans le terminal. Si la lecture réussit, alors la fonction
2 renvoie true et stocke l'entier lu dans res. Si la lecture est infructueuse,
3 alors la fonction renvoie false. Dans les deux cas, l'entrée du terminal est
4 ensuite vidée de tout caractère additionnel */
5 bool read_int_and_flush(int* res);
```

Pour cela, nous allons utiliser le type `char`, dont le format pour scanf et print est `"%c"` et permet de lire exactement un caractère. De plus, on notera que la fonction scanf **renvoie** un entier, indiquant le nombre d'objets lus avec succès. Par exemple, `scanf("%d", x)` renverra 1 en cas de succès car on lira un entier.

Le schéma proposé pour la fonction `read_int_and_flush` est le suivant :

1. Lire un entier avec scanf dans le terminal ;
2. En cas de succès, on peut stocker l'entier dans le pointeur donné en entrée ;
3. Afin de vider l'entrée du terminal, on lit un caractère dans le terminal jusqu'à lire un retour à la ligne, i.e. le caractère `'\n'` (mettez bien des guillemets simples !) ;
4. On renvoie la valeur booléenne correspondant au résultat.

Question 1. Depuis l'archive du TP, copiez-collez le fichier `int_flush.c`, qui contient déjà une fonction main permettant de tester le programme. Implémentez la fonction `read_int_and_flush` selon le schéma décrit ci-dessus et testez son comportement soigneusement. Vous devez pouvoir observer l'exécution suivante :

```
Rentrez un entier (0 pour finir): 12tutu
Entier lu: 12
Rentrez un entier (0 pour finir): 95
Entier lu: 95
Rentrez un entier (0 pour finir): bla78
Erreur de lecture
```

Tableaux

Un **tableau** est une suite de cases mémoires successives qui ont été réservées. Contrairement aux listes Python, un tableau en C ne peut stocker qu'un seul type d'élément. On peut avoir des tableaux de float, des tableaux de 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 de 4 octets, contigües dans la mémoire.

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 avec des valeurs :

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

ou bien seulement déclarer le tableau et le remplir avec une boucle :

```
1 int carres[500];
2 for (int i = 0; i < 500; i++){
3     carres[i] = i*i;
4 }
```

En C, il n'y a aucun moyen d'obtenir la taille d'un tableau. Ainsi, lorsqu'une fonction manipule un tableau, on doit fournir la taille du tableau en paramètre. De plus, on ne peut pour l'instant créer que des tableaux de taille constante (i.e. écrire `int tab[50];`) et pas de taille variable (i.e. écrire `int tab[n];`.

Pour l'instant, on ne peut pas non plus faire de fonction qui ***renvoie*** un tableau : on verra pourquoi en cours, et surtout comment y remédier.

Exercice 7.

Question 1. Lisez et recopiez le code suivant dans un nouveau fichier :

```
1  #include <stdio.h>\\
2
3  // calcule la somme des n premiers entiers de t. n doit être <= 100
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. Que pouvez vous en déduire sur le type des tableaux ?

Question 2. Sur le même modèle, écrivez une fonction qui permet d'afficher les n premières cases d'un tableau.

Question 3. 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.

Question 4. 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 $T[0]$, etc...

Préprocesseur, macro #define Dans vos programmes, vous serez amenés à manipuler des tableaux d'une certaine taille. Par exemple, si vous manipulez un tableau de taille 100, vous devrez écrire 100 à de nombreux endroits du programme. Pour changer 100 en 1000, il faudra changer CHAQUE instance. Afin d'éviter cela, on utilise la directive ***#define*** qui n'est pas une instruction, mais permet de donner un nom à un bout de code. Par exemple, si au début de votre programme, vous écrivez :

```
1 #define N 1000
```

Après cette ligne, dans le reste du programme, à chaque fois que vous écrirez `N`, le compilateur verra les caractères "1000". Par exemple, le code suivant :

```
1 #define N 1000
2 int n = N;
3 N = 50;
```

sera lu par votre compilateur comme suit :

```
1 int n = 1000;
2 1000 = 50; // gros problème !
```

Il faut donc bien faire attention que `N` n'est pas une variable ! Il vaut mieux le voir comme une constante mathématique définie pour le programme. En C, on dit que `N` est une **macro**. Lors de la compilation, une première phase va remplacer toutes les macros créées par `#define` par leur valeur. Pour cette raison, `#define` est appelée une ***directive préprocesseur***. Toutes les directives préprocesseur commencent par # (par exemple, les inclusions de librairies).

Comme on ne peut pas encore déclarer de tableau de taille variable en C, on va se contenter pour le moment d'écrire des programmes qui réservent des tableaux immenses, quitte à n'utiliser que les premières cases, par exemple :

```c
#define N 100000
int t[N];

int main(){
    int n;
    scanf("%d", &n);
    assert(n <= N);
    for (int i = 0; i < n; i++){
      t[i] = i*i;
    }
}
```

Ici on n'utilise que les n premières cases du tableau. 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 en cours comment réserver de la mémoire de manière plus adaptée dans ce type de situation.

Exercice 8.

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 du tableau. Le but est de trier le tableau en utilisant le moins de coups.

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
[5, 12, 8, 21]
Rentrez les indices à échanger: 1 2
[5, 8, 12, 21]
Bravo !! Vous avez gagné en 2 coup(s) !
```

Vous trouverez dans l'archive du TP le fichier `jeu_tri.c`, qui contient du code à compléter. Ce fichier est là pour vous donner des indices sur la bonne direction à prendre. Si vous souhaitez implémenter le programme à votre manière sans vous en inspirer, vous pouvez le faire (du moment que votre code est clair et **bien commenté**).

Exercice 9. Optionnel

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 quelles cases n'ont 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. En pratique, on pourra utiliser un tableau de booléens dont chaque case indiquera si la case de même indice dans la mémoire est encore non-initialisée.

Enfin, au tout début du programme, l'utilisateur rentre une taille de mémoire maximale $n \in [\![0, 1024]\!]$, et le programme devra toujours s'assurer que les adresses rentrées par l'utilisateur sont valides.

Les exécutions devront ressembler à :

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

Vous trouverez dans l'archive du TP le fichier `memory.c`, qui contient du code à compléter. Ce fichier est là pour vous donner des indices sur la bonne direction à prendre. Si vous souhaitez implémenter le programme à votre manière sans vous en inspirer, vous pouvez le faire (du moment que votre code est clair et **bien commenté**).

Chaîne de caractère

Le type `char` sert en réalité à représenter non seulement des entiers signés 8 bits, mais aussi des caractères (comme son nom l'indique), en associant à chaque symbole un nombre entier entre 0 et 127.

Exercice 10. Le fichier `code_to_char.c` dans l'archive du TP est un programme qui, une fois compilé, permet à l'utilisateur de rentrer un entier entre 0 et 127 et affiche le caractère correspondant.

Le fichier `char_to_code.c` implémente la fonctionnalité inverse : on y rentre des caractères, et le programme nous affiche l'entier dans $[\![0, 127]\!]$ correspondant.

Question 1. Compilez ces programmes (donnez leur des noms distincts !), et donnez les entiers correspondant aux caractères suivants

a) Les lettres majuscules 'A' - 'Z' et les lettres minuscules 'a' - 'z'
b) Les chiffres '0'-'9'
c) Le caractère '\'
d) `'\n'` (le caractère de retour à la ligne, aussi appelé ***Line Feed*** ou ***LF***)

Question 2. Donnez également les caractères correspondant aux entiers suivants :

a) `0x20`
b) `0x28` et
c) `0x29`

Ainsi, en C, lorsque l'on écrit `'a'`, c'est ***exactement*** comme si l'on écrivait `97` ou `0x41`. Vous pouvez consulter en lignes la table de correspondance caractère/code, que l'on appelle la **table ASCII** (En fait, ce système d'encodage entier s'appelle l'ASCII). 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 !

Un texte est donc représenté par une ***chaîne de caractères***, c'est à dire un tableau de caractères. En anglais on parle également de *character strings* ou **strings**. Toutes les chaînes de caractère finissent par le caractère `0x00`, appelé caractère nul. En C, on peut le noter `'\0'`. Ainsi, contrairement au cas général des tableaux, il est possible de connaître la longueur d'une chaine de caractères, en la lisant jusqu'à rencontrer le caractère nul. En revanche, la longueur d'une chaîne de caractères n'est pas nécessairement égale à la taille mémoire qu'on lui a réservé. Si l'on déclare une chaîne de 500 caractères, et que l'on écrit `'\0'` dans la case 3, la longueur de la chaîne est 3, alors que la taille mémoire réservée est 500.

Par exemple, si l'on considère le code suivant :

```
1 void main(){
2     char bonjour[20] = "Bonjour !";
3 }
```

Le tableau `bonjour` contient 20 cases, dont seules les 10 premières sont réellement utiles. Les 9 premières cases contiennent les caractères 'B', 'o', 'n', 'j', 'o', 'u', 'r', ' ', '!', et la 10ème case (celle d'indice 9) contient le caractère nul, marquant la fin de la chaîne.

On dit qu'une chaîne de caractères est **bien-formée** si elle se finit par le caractère nul, et qu'elle est **mal-formée** sinon.

En C, le format `"%s"` sert à afficher une chaîne de caractères, et a pour effet d'afficher tous les caractères dans l'ordre jusqu'à rencontrer un `'\0'`. Si la chaîne de caractère est mal-formée, alors le programme va continuer à lire des cases mémoires par delà la mémoire réservée pour la chaîne de caractère jusqu'à trouver un octet valant 0 (avec de la malchance, on pourrait ainsi lire l'intégralité de la mémoire!!).

Lorsque l'on utilise le format `"%s"` avec la fonction scanf, le programme va lire les caractères dans le terminal jusqu'à rencontrer un saut de ligne ou un espace.

Exercice 11.

Question 1. 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 ? Expliquez en quoi ce comportement peut être exploité pour faire dysfonctionner un programme.

```c
#include <stdio.h>

/* Répète ce que l'utilisateur écrit tant que le programme
 * n'est pas interrompu*/
int main()
{
  char buf[20]; // sert à stocker la chaîne lue
  while (1){
    scanf("%s", buf); // On n'écrit pas &buf car
                      // buf est déjà un pointeur !!

    printf("Vous avez écrit: %s\n", buf);
  }
  return 0;
}
```

Question 2. Le code suivant est dans le fichier string.c. Lisez-le et, sans l'exécuter, écrivez ce qu'il devrait afficher. Exécutez le programme pour vérifier.

```c
#include <stdio.h>

int main(){
  char str[20];
  str[2] = 'U';
  str[3] = 'L';
  str[4] = 'T';
  str[8] = 'R';
  str[1] = 'A';
  str[0] = 'S';
  str[5] = '\0';
  str[6] = '!';
  str[7] = '!';

  printf("%s\n", str);
  return 0;
}
```

Question 3. Compilez et exécutez le fichier coucou_bonjour.c dans l'archive fournie avec le TP. Décrivez son comportement, et formulez une hypothèse sur pourquoi il se comporte de la sorte :

```c
#include <stdio.h>

int main(int argc, char const *argv[])
{
  char str1[8] = "BONJOUR"; // 7 lettres + 1 caractère nul = 8
  char str2[7] = "COUCOU"; // 6 lettres + 1 caractère nul = 7
  str1[7] = ' ';
  str2[6] = ' ';

  printf("%s\n", str1);
  printf("%s\n", str2);

  return 0;
}
```

Une des utilités du caractère nul, comme nous l'avons déjà évoqué plus haut, est que l'on peut déterminer la longueur d'une chaîne de caractères bien-formée. Dans certains cas, on peut même effectuer certaines opérations sur les strings sans déterminer la longueur, en utilisant le caractère nul comme condition de fin :

```c
// affiche chaque caractère de s sur une ligne
void epeler(char* s){
  int i = 0;
  while (s[i] != '\0'){
      printf("%c\n", s[i]);
      i++;
  }
}
```

ou alternativement :

```c
// affiche chaque caractère de s sur une ligne
void epeler(char* s){
  for (int i = 0; s[i] != '\0'; i++){
      printf("%c\n", s[i]);
  }
}
```

Exercice 12.

Le but de cet exercice est d'étudier quelques fonctions de la librairie `<string.h>`, qui permettent de manipuler facilement des strings. En particulier, nous allons nous intéresser à 4 fonctions :

— `int strlen(char* s)` qui renvoie la longueur de s, c'est à dire son nombre de caractères (caractère nul **exclus**) ;

— `int strcmp(char* s1, char* s2)` qui renvoie 0 si les deux chaînes en entrée sont égales, -1 si s_1 est plus petite dans l'ordre alphabétique, et 1 si s_2 est plus petite. L'ordre des lettres est simplement l'ordre naturel sur le type `char` ;

— `void strcpy(char* dst, char* src` qui perend la chaîne écrite dans la source `src` et la recopie dans la destination `dst`. Par exemple :

```
1 char str1[30] = "Bonjour";
2 char str2[40] = "Au revoir";
3 strcpy(str1, str2); // copie str2 dans str1
4 printf("%s\n", str1);
```

Ce code affichera `Au revoir`.

— `void strcat(char* dst, char* src)` qui concatène (c'est à dire ajoute) la chaîne `src` à la fin de la chaîne `dst`. Par exemple :

```
1 char str1[50] = "Bonjour";
2 char str2[20] = " tout le monde !";
3 strcat(str1, str2); // concatène str2 après str1
4 printf("%s\n", str1);
```

Ce code affichera `Bonjour tout le monde !`

Pour les deux dernières fonctions, il faut faire attention : la chaîne de destination doit avoir assez de place pour stocker le résultat. Si ce n'est pas le cas, le comportement du programme est imprévisible, car on peut aller écrire dans des emplacements mémoires où l'on est pas sensés écrire. La page de documentation de `strcar` dit même :

`Buffer overruns are a favorite avenue for attacking secure programs.`

(Un "buffer overrun" désigne un débordement d'une chaîne de caractères.)

Question 1. Créez un nouveau fichier c, et testez les 4 fonctions ci-dessus dans le main pour vérifier leur comportement.

Question 2. Dans un autre fichier c, implémentez votre propre version des 4 fonctions. Afin d'éviter les conflits de noms, vous les appellerez `ma_strlen`, `ma_strcmp`, etc... Avant de coder chaque fonction, vous écrirez dans le main des tests utilisant des assertions permettant de vérifier que vos fonctions se comportent comme les fonctions de la librairie standard. Par exemple :

```
1 //dans main():
2     char s1[50] = "Un test de strcpy";
3     char s2[50];
4     char s3[50];
5     strcpy(s2, s1); // copie s1 dans s2
6     ma_strcpy(s3, s1); // copie s1 dans s3
7     assert (strcmp(s2, s3) == 0);
```

Rien n'empêche de manipuler des tableaux de chaînes de caractères, et même de manière générale des tableaux de tableaux (ou des tableaux de tableaux de tableaux de ...). Par exemple :

```
1 float grille[30][40];
```

`grille` sera un tableau de 30 cases, et chaque case sera elle-même un tableau de 40 lignes. Ainsi, on peut accéder aux différents flottants de la grille avec `grille[i][j]` avec $i \in [\![0, 29]\!]$ et $j \in [\![0, 39]\!]$.

Exercice 13.

"Cent mille milliards de poèmes" est un ouvrage / une expérience de Raymond Queneau, un écrivain surréaliste du XX^{e} siècle membre de l'Oulipo (un groupe de recherche sur la création littéraire). L'ouvrage est un sonnet (14 vers), où chaque vers possède 10 versions différentes interchangeables. Ainsi, il y a 10^{14} combinaisons possibles, c'est à dire cent mille milliards.

Le fichier `queneau.c` de l'archive du TP contient un tableau `char* vers[14][10]`, c'est à dire un tableau de 14 cases, où chaque case est elle-même un tableau contenant les 10 versions d'un vers.

Complétez ce fichier pour obtenir un programme affichant un des 10^{14} poèmes au hasard.

Exercice libre (Optionnel)

Ce TP est déjà très copieux, mais vous pouvez si vous le souhaitez appliquer toutes les notions vues afin d'écrire un programme en étant moins guidé que dans le reste du sujet. Quelques suggestions d'idées :

- 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 jeu de bataille, de puissance 4, etc...
- Un programme qui calcule et affiche le triangle de Pascal, le triangle de Sierpinski, la spirale d'Ulam, etc...