TP3 : Pointeurs, tableaux, chaînes de caractères

MP2I Lycée Pierre de Fermat

Consignes

Téléchargez sur Cahier de Prépa l'archive de ce TP (dans les documents à télécharger).
Ce TP est à rendre d'ici **mercredi 30 novembre** à 21h. Pensez à noter vos réponses dans un fichier texte et à supprimer tous les exécutables avant de compresser l'archive.
Certains exercices sont **optionnels**, généralement ils viennent clore une section du TP, et sont un peu plus durs. Il est conseillé de les sauter la première fois que vous y arrivez, et d'y revenir une fois que vous avez fini la partie principale du TP. Il est fortement conseillé de les faire en entier si vous êtes à l'aise en informatique, et de les essayer peu importe votre niveau.

⇓⇓Lisez le paragraphe en dessous SVP⇓⇓

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 systématiquement rajouter dans le main du programme des jeux de tests pour cette fonction, même si la question ne le précise pas.

⇑⇑Lisez le paragraphe au dessus SVP⇑⇑

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\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, qui ajoutent tout un tas d'avertissements divers). Par exemple :

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

Exercice 1

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.

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 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**.
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.

On peut accéder à l'adresse d'une variable `x` en écrivant `&x`. On appelle `&` l'opérateur de **référencement**. Si `x` est d'un certain type `T`, alors son adresse `&x` est de type `T*` (lu "T étoile"). Les valeurs de type `int*`, `float*`, etc..., 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; // variable de type "pointeur d'entier"
  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 d'octets 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 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 3

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` 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 renvoient un pointeur, et 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 : on n'a pas le droit de lire ni d'écrire dans le tout premier octet de l'ordinateur !
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 segmenter vos programmes en ***fonctions*** et de munir ces fonctions d'***assertions*** car cela vous aidera à mieux localiser les bugs.

Exercice 4

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 5

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.

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. 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.

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 :

```
/* Lit un entier dans le terminal. Si la lecture réussit, renvoie true
   et stocke l'entier lu dans res. Sinon, renvoie false.
   Dans les deux cas, l'entrée du terminal est ensuite vidée de tout
   caractère additionnel */
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.

Q1. 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
```

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 et le remplir avec une boucle :

```
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 }
```

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 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
4 int tab3[12+74]; // PAS OK: 12+74 est une somme, 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.

Exercice 7

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. Que pouvez vous en déduire sur le type des tableaux ?

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 $T[0]$, etc...

Préprocesseur, macro #define Les directives préprocesseurs sont des lignes qui ne sont pas du code C, mais des instructions lues avant la compilation, et traitées par le **préprocesseur**. Elles 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 est `#define`, qui permet de créer des **macros**. Une macro est un bout de texte réutilisable auquel on donne un nom.

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;
  printf("%d\n", 1000);
}
```

On voit donc que `N` n'est pas une variable !

Dans vos programmes, vous serez amenés à manipuler des tableaux d'une certaine taille fixée. 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. On ne peut pas utiliser de variable, car il n'est pas possible pour le moment de créer un tableau dont la taille n'est pas une constante en C. En revanche, on peut utiliser les macros et la directive ***#define*** :

```c
#define N 100000

/* 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;
  }
}
```

On peut donc simuler des tableaux de tailles variables, en n'utilisant 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

plus tard 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 adjacentes 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
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 trouverez dans l'archive du TP le fichier `jeu_tri.c`, qui contient du code à compléter. Ce fichier est là pour vous donner une structure de base si vous ne savez pas comment commencer. 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 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. 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 une structure de base si vous ne savez pas comment démarrer. 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**).

4 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. Les char dont le bit de poids fort est 1 ne représentent pas un caractère valide.

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.

Q1. 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***)

Q2. Donnez également les caractères correspondant aux entiers suivants :

a) `0x20` b) `0x28` 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 ligne 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'`. La longueur d'une chaîne de caractères n'est pas nécessairement égale à la taille mémoire qu'on lui a réservé. 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 (`bonjour[9]`) contient le caractère nul, marquant la fin de la chaîne. Les cases `bonjour[10]` jusqu'à `bonjour[19]` sont inutilisées.

Exercice 11

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 aux tableaux généraux, on peut effectuer certaines opérations sur les chaînes de caractères sans connaître leurs longueurs au préalable, 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++;
  }
}
```

On dit qu'une chaîne de caractères est **bien formée** si se termine par le caractère nul. Le format pour afficher les chaînes de caractère avec `printf` est `%s`. Si l'on essaie d'afficher une chaîne mal formée, `printf` essaiera de lire dans la mémoire jusqu'à trouver un octet nul, au delà de la mémoire réservée pour la chaîne, et si l'on a pas de chance au delà de la mémoire autorisée. Ce type de comportement peut faire totalement bugger un programme, ou même forcer son arrêt. D'où l'importance de toujours créer des chaînes bien formées !

Exercice 12

Pour toutes les fonctions suivantes, on suppose que les chaînes de caractères données en entrée sont bien formées, et que les tableaux ont été réservés avec assez d'espace pour que les fonctions s'exécutent correctement. On ne le vérifiera pas dans les assertions (car c'est impossible).

Q1. Écrire une fonction `int strlen(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 strcmp(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 strcpy(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";
strcpy(str1, str2); // copie str2 dans str1
printf("%s\n", str1); // affiche "Au revoir"
```

Q5. Écrire une fonction `void strcat(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 !";
strcat(str1, str2); // concatène str2 après str1
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`. Vous devez connaître l'existence de cette librairie, ainsi que de ces 4 fonctions !

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, un espace, ou tout autre caractère blanc.

Exercice 13

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 ? 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 chaÃ®ne 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;
}
```

Q2. 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;
}
```

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, où chaque case représente une ligne du tableau, et est donc elle-même un 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 14

"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 si vous souhaitez encore vous entraîner, vous pouvez 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...