TP n°4 - Pointeurs, tableaux et chaînes de caractères

Notions abordées

- pointeurs, adresse d'une variable, déréférencement (opérateurs `&` et `*`)
- tableaux statiques 1D, 2D, tableaux de pointeurs
- chaînes de caractères
- arguments de la fonction `main` (`argc` et `argv`)
- découverte de `malloc` et `free` pour la réservation de mémoire dans le tas
- découverte de `getchar` pour la lecture de fichiers

Toutes les fonctions doivent être commentées et testées. Elles doivent notamment être munies d'une description faisant suite à d'éventuelles hypothèses sur leurs arguments.

Chaînes de caractères

Exercice 1 Longueur d'une chaîne de caractères

Question 1 💻

Codez une fonction `longueur` qui donne la longueur d'une chaîne de caractères. On parle de la longueur usuelle, c'est-à-dire le nombre de caractères qui la composent, sans compter le caractère nul ajouté en mémoire en C. Par exemple `""`, est de longueur 0 et `"MP2I"` est de longueur 45.

Question 2 💻

Codez une fonction `epele` qui affiche un à un les caractères d'une chaîne de caractères, avec un saut de ligne entre chaque. Dans une version améliorée, cette fonction pourra afficher "`espace`" au lieu d'un espace. (Et si vous êtes motivés "`tabulation`" pour le caractère `'\t'`, "à la ligne" pour le caractère `'\n'`,).

Question 3 ✍

Votre fonction `epele` fait-elle appel à la fonction `longueur` ? Selon vous, est-ce efficace ? Ce qui importe du point de vue de l'efficacité c'est de faire peu d'opérations, et non la concision ou la clarté du code (qui est un autre critère).

Exercice 2 Copie d'une chaîne de caractères

Question 1 💻

Sur cahier de prépa téléchargez le fichier nommé `copie_bete.c`. Complétez ce fichier en y incluant

votre définition de la fonction `longueur`, puis compilez-le. Malgré l'erreur qui apparaît à la compilation, exécutez le programme compilé.

Question 2 ✍
Grâce à l'erreur de compilation et à l'affichage obtenu à l'exécution, essayez d'expliquer ce qui s'est passé, et pourquoi la fonction de copie ne pouvait pas marcher. *Préparez votre réponse puis appelez-moi pour valider votre proposition.*

Lorsqu'on déclare une variable locale dans une fonction, un espace mémoire est réservé pour le stockage de la valeur de cette variable, mais cette réservation ne vaut que le temps de l'exécution de la fonction. Dès qu'on sort de la fonction, cette zone peut être utilisée à d'autres fins, ce qui est cohérent avec le fait qu'on ne puisse plus utiliser ladite variable locale.

Grâce à la fonction `malloc` de la librairie `stdlib.h`, on peut réserver une zone mémoire qui ne sera pas libérée à la sortie de la fonction, une zone mémoire persistante. Cette possibilité s'accompagne d'une responsabilité : il faudra préciser quand libérer cette zone mémoire.

Plus précisément la fonction `malloc` prend en argument un nombre d'octets à réserver, réserve une zone de cette taille (dans le tas), et renvoie un pointeur de type `void*` vers cette zone mémoire (comme toujours c'est l'adresse du premier octet de la zone). Il faut bien sûr enregistrer cette adresse, sinon on ne pourra pas utiliser — ni libérer — la zone qu'on vient de réserver, mais il faut aussi trans-typer cette adresse, c'est-à-dire la convertir de `void*` à `t*` afin que l'on navigue convenablement dans cette zone. En effet, on attend que l'opérateur `[1]` nous donne accès à la "case" suivante, mais on n'a pas encore précisé la taille de ces cases... En pratique, une utilisation de `malloc` suit à peu près le schéma suivant.

```
//déclaration et réservation
t* p = (t*) malloc( sizeof(t) * N);

//remplissage
int i = 0;
while (i < N){
  p[i]= <expression de type t>;
  i++;
}

//libération
free(p);
```

Question 3 🖳
Codez une fonction `copie` qui retourne un pointeur de type `char*` pointant vers une zone mémoire persistante contenant la chaîne qu'elle prend en entrée. *Lorsque que vous testez la fonction `copie`, vous faites des `malloc` cachés dans ces appels, pensez à faire les `free` correspondants.*

Exercice 3 Comparaison de chaînes de caractères

Question 1 🖳 ✍
On considère le programme suivant. Essayez de décrire ce qui va se passer. Récupérez ensuite ce programme sur cahier de prépa (fichier `test_comparaison_chaine.c`), compilez-le et exécutez-le pour voir si vos prévisions étaient justes. *Je ne vous demande pas de prédire les adresses affichées,*

mais tout ce que vous pouvez déjà attendre, en particulier le résultat du test d'égalité ligne 17.

```c
#include <stdio.h>

int main(){

  char* a = "salut";
  printf("a vaut %lu où se trouve la chaîne %s.\n",a,a);

  //copie de la chaine vers laquelle pointe a à la main
  char b[6];
  int i=0;
  while(i<5){
    b[i]=a[i];
    i++;
  }
  printf("b vaut %lu où se trouve la chaîne %s.\n",b,b);

  printf("(a==b) = %d \n",a==b);

  return 0;
}
```

Question 2 ✍
Comment expliquer le constat fait à la question précédente. Que retenir ? Appelez-moi pour valider votre conclusion.

La comparaison entre chaînes de caractères peut être plus précise qu'un test d'égalité : elle peut révéler quelle chaîne est la plus petite pour l'ordre lexicographique, c'est-à-dire l'ordre utilisé dans un dictionnaire, ou encore l'ordre sur les mots obtenu par produit de l'ordre sur l'alphabet.

Définissons formellement cette notion pour les mots sur un alphabet Σ ordonné par $\leqslant$.
L'**ordre lexicographique** induit par $\leqslant$ est la relation $\preccurlyeq$ définie sur Σ^* par $\varepsilon = \min_{\preccurlyeq} \Sigma^*$ et

$$\forall u = u_1 \dots u_l \in \Sigma^* \setminus \{\varepsilon\},\ \forall v = v_1 \dots v_k \in \Sigma^* \setminus \{\varepsilon\},\ \ u \preccurlyeq v \iff \Big(u_1 < v_1 \text{ ou } \Big(u_1 = v_1 \text{ et } u_{i\in[2..l]} \preccurlyeq v_{i\in[2..k]}\Big)\Big)$$

Question 3 ✍
Comparez deux à deux les chaînes "beau", "bon", "bonjour" et "" en utilisant l'ordre lexicographique (pour l'ordre habituel sur l'alphabet).

Question 4 💻
Codez une fonction `comparaison` qui prend en entrée deux chaînes de caractères s et t et qui renvoie 0 si elles sont égales, -1 si $s \preccurlyeq t$, 1 sinon, où $\preccurlyeq$ désigne l'ordre lexicographique usuel. *Notez que la comparaison entre caractères est possible en C grâce aux opérateurs usuels : `==`, `!=`, `<`, `<=`, `>`, `>=`.*

Tableaux statiques

Exercice 4 Tableaux simple (*i.e.* à une dimension)

Question 1 💻
Codez une fonction `moyenne` qui calcule la moyenne d'un tableau de flottants. *À vous de fixer les arguments et les hypothèses adéquats.*

Question 2 💻
Si vous avez pris de bonnes habitudes de programmation vous avez dû tester votre fonction avec des `assert` et des tests d'égalité par `==`. Si ça a marché, vous avez eu de la chance. Testez si la moyenne du tableau `1/3.0,1,1/3.0` est bien égale à `5/9.0`.

Question 3 💻
En vous inspirant de ce qui était demandé au TP précédent, proposez une solution pour tester "*l'égalité*" entre flottants.

Question 4 💻
Codez une fonction `moyenne_ponderee` qui calcule la moyenne pondérée à partir d'un tableau de valeurs flottantes et d'un tableau de coefficients flottants. *À vous de fixer les arguments et les hypothèses adéquats.*

Exercice 5 Tableaux à deux dimensions

Question 1 💻
Proposez une fonction qui calcule les sommes de chaque sous-tableau d'un tableau à deux dimensions, et les stocke dans un tableau à une dimension. *Au choix cous créerez un nouveau tableau pour les sommes ou bien vous remplirez seulement un tableau passé en paramètre. Dans un cas comme dans l'autre, l'importance est que ce soit clair dans la description.*

Tableaux de pointeurs

Exercice 6 Tableaux de `char*` pour la ligne de commande

On a vu comment passer des arguments à une fonction, on va maintenant voir comment passer des paramètres à un programme. Cela signifie qu'après que le programme a été compilé, on peut le lancer avec différentes valeurs pour ces paramètres, et que l'exécution du programme dépendra de ces valeurs.
Si les arguments d'une fonction ont un type et un nom, ce qui facilite leur usage, les paramètres d'un programme sont juste des chaînes de caractères, qu'on identifie par leur position dans la ligne de commande. En effet, on passe des paramètres en les ajoutant à la ligne de commandes, après la commande `./mon_programme`, séparées par des espaces.
Dans le programme, on récupère ces chaînes en argument de la fonction `main`. Plus précisément, on

récupère d'une part le nombre total de mots dans la ligne de commande, sous la forme d'un entier nommé `argc`, et ces mots dans un tableau de `char*` nommé `argv`. La signature de la fonction `main` est alors : `int main(int argc, char* argv[]);`

Par exemple, pour la ligne de commande `./mon_programme a b 12`, `argc` vaut 4 (la commande de lancement et les 3 paramètres), et

- `argv[0]` pointe vers la chaîne `"./mon_programme"`
- `argv[1]` pointe vers la chaîne `"a"`
- `argv[2]` pointe vers la chaîne `"b"`
- `argv[3]` pointe vers la chaîne `"12"`

Question 1 💻
Créez un programme `echo` qui affiche la ligne de commande qui l'a lancé, hormis le premier mot `./echo`. *Ce nom n'est pas choisi au hasard, on recode ce que fait la commande `echo` de bash, que vous pouvez essayer dans le terminal.*

Question 2 💻 bonus
Créez un programme qui affiche la valeur d'un expression arithmétique écrite en notation polonaise inversée, c'est-à-dire où les opérateurs arithmétiques sont placés après leurs deux opérandes (notation post-fixe) et non au milieu comme d'habitude (notation infixe). Par exemple, $(2 + 3) \times 4$ s'écrit $4\,2\,3\,+\,\times$. On se limitera ici aux expressions dont les opérations sont l'addition et la multiplication, notées respectivement $+$ et $\times$ (et surtout pas `*` dans la ligne de commande pour éviter les mauvaises surprises) et dont les nombres sont entiers.
Vous aurez besoin de la fonction `atoi` de la bibliothèque `stdlib.h`, qui prend en argument une chaîne de caractères, et qui renvoie l'entier auquel elle correspond. Par exemple, `atoi("123")` vaut 123, de même que `atoi(" 123 ")`.

Exercice 7 Tableaux de `char*` pour les lignes d'un fichier texte

A Fichiers et manipulation de flux

On s'intéresse dans cet exercice à la lecture d'un fichier texte par un programme C. Cela permet d'importer des chaînes de caractères dans notre programme pour les manipuler, sans avoir à les copier dans le programme avant compilation, ni avoir à les saisir à la main lors de l'exécution.

Pour cela il faut savoir que le fichier texte va être traité comme un flux de caractères : toutes les lignes du fichier sont comme mises bout à bout, séparées par des caractères de saut de ligne (`'\n'`), et suivies d'un caractère spécial pour marquer la fin du fichier. Dans le code, on notera `EOF` (comme *end of file* en anglais) pour désigner ce caractère de fin de fichier, le compilateur se chargera de remplacer `EOF` par le caractère effectivement placé à la fin des fichier sur notre machine.

Le flux sur lequel un programme C lit s'appelle `stdin` (comme *standard input* en anglais, soit entrée standard), tandis que celui sur lequel il écrit via `printf` s'appelle `stdout` (comme *standard output* en anglais, soit sortie standard). Dans un terminal on peut gérer ces flux. Certaines commandes `bash` sont capables d'écrire sur la sortie standard (et donc d'afficher), notamment les commandes `echo` et `cat`.

Question 1 💻
Ouvrez un terminal dans un répertoire `test` pour éviter d'écraser vos fichiers par mégarde. Tapez

`echo "salut"` puis entrée. Qu'observez-vous ? En déduire ce que fait la fonction `echo`. *Si besoin faites plusieurs essais.*

Question 2 💻
Toujours dans ce dossier `test`, tapez `echo "salut" >> res.txt`. Que remarquez-vous ? Tapez-ensuite `cat res.txt`, qu'observez-vous ? Vous pouvez ouvrir le fichier `res.txt` dans un éditeur de texte pour vous aider à comprendre ce qui se passe. Renouvelez les deux commandes et expliquer le résultat.

En fait `>>` est un opérateur de **redirection**, il redirige le flux `stdout` généré par la commande à sa gauche vers le fichier indiqué à sa droite, à la suite de ce qui est déjà écrit dans le fichier. L'opérateur `>` lui fait la même redirection sauf qu'il écrase (*i.e.* efface) ce qui se trouve éventuellement dans le fichier. La commande `cat` quant à elle fait un peu le contraire : elle affiche sur la sortie standard le contenu du fichier qu'on lui indique en argument.

Dans cet exercice on va utiliser la fonction `getchar` de la librairie `stdio.h` qui permet de lire le premier caractère du flux stdin, et de se décaler pour qu'au prochain appel le caractère lu soit le deuxième. On peut voir ça comme un décalage de curseur, ou penser qu'on a effacé ce premier caractère en le lisant, on l'a mangé, et on ne pourra plus y revenir.

Pour lire le texte contenu dans un fichier avec un programme C, il nous faut donc être capable de le faire passer dans le `stdin`. Rappelons que grâce à la commande `cat` on sait le faire passer dans le `stdout`. C'est là qu'intervient l'opérateur de redirection `|` (altGR+6) (appelé *pipe*, *i.e.* tuyaux en anglais). Cet opérateur permet de rediriger le flux `stdout` de sa commande de gauche vers le `stdin` de sa commande de droite. Il nous suffit donc d'utiliser `cat` à gauche de ce symbole, et de lancer l'exécution de notre programme à droite.

Question 3 💻
Téléchargez le fichier `lecture_ligne.c` sur cahier de prépa, placez le dans le dossier `test` puis compilez-le. Toujours dans le même fichier, créez un fichier `test.txt` avec le texte de votre choix. Tapez-ensuite `cat test.txt | ./lecture_ligne`. Qu'observez-vous ? Ouvrez le fichier `lecture_ligne.c` et essayez de comprendre ce qu'il s'est passé.

Question 4 💻
Que se passera-t-il si on tape `echo "bonjour" | ./lecture_ligne` ? Prédire puis tester.

B Lire et afficher un fichier ligne à ligne

Maintenant qu'on a vu comment rediriger le texte d'un fichier vers notre programme C, et comment le lire caractère par caractère, on va essayer de stocker ce texte ligne par ligne grâce à un tableau de pointeurs de type `char*`. Chaque ligne est une chaîne de caractères, donc de type `char*`, et on rassemble ces lignes dans un tableau.

On se donnera un nombre maximum de caractère par ligne (`nbc_max`) et un nombre maximum de lignes par fichier (`nbl_max`), afin de pouvoir réserver en mémoire un espace suffisant avant de connaître l'espace réellement nécessaire (*i.e.* la taille réelle de chaque ligne, la taille du fichier).

Vous coderez dans un fichier nommé `exo_7.c` les fonctions demandées dans cette partie. Vous pouvez déjà y copier la fonction `lit_ligne` donnée dans le fichier `lecture_ligne.c`.

Question 5 ✍

Donnez une représentation graphique de la structure qu'on cherche à créer pour stocker le texte d'un fichier. *Faites un petit dessin, comme ceux que je fais en cours pour illustrer ce qui se passe en mémoire. N'hésitez pas à utiliser des couleurs.*

Question 6 💻
Pour commencer, codez une fonction `copie_bis` qui va copier une chaîne de caractère dans une autre en supposant qu'elles ont la même longueur, et que celle-ci est connue. Votre fonction prend donc en entrée deux `char*` et un `int`. *Pas de `malloc` dans cette fonction, on suppose que les pointeurs passés en paramètre correspondent à des zones mémoire déjà reversées et de taille suffisante.*

Question 7 💻
Codez une fonction `lit_fichier` qui lit une à une les lignes de `stdin` et les enregistre dans un tableau `p_lignes` passé en paramètre. On considère que ce tableau est déjà déclaré. Sa taille `nbl_max` est aussi passée en argument, ainsi que `nbc_max` le nombre de caractères maximal par ligne.
Attention, chaque élément de `p_lignes` doit pointer vers une chaîne qui a pile la bonne longueur, c'est-à-dire le nombre de caractères hors `'\n'` de la ligne (+1 pour `'\0'`) et pas `nbc_max` par défaut. De plus cette chaîne doit être persistante : on veut pouvoir y accéder après être sorti de la fonction, il faudra donc utiliser `malloc`

Question 8 💻
Codez une fonction `affiche_fichier` qui affiche le texte enregistré dans un tableau de lignes comme le tableau `p_lignes` rempli dans la fonction précédente. Grâce à cette fonction vous pouvez tester si la fonction précédente fait ce que vous voulez, en lisant puis affichant un fichier texte de votre choix.

C Réordonner un fichier mélangé (bonus)

Question 9 💻
Codez une fonction `affiche_fichier_num` qui affiche un fichier à partir d'un tableau de ligne, en ajoutant au début de chaque ligne son numéro suivi d'un espace. Bien que l'usage soit de numéroter les lignes à partir de 1, pour simplifier les questions suivantes on commencera la numérotation à 0.

Question 10 💻
Testez votre code en créant une version numérotée d'un fichier texte de votre choix. *Vous devez utiliser ici deux redirections, une pour la lecture et une autre pour stocker la version numérotée de votre fichier*

Question 11 💻
On suppose qu'un problème (improbable certes, mais c'est un prétexte à la manipulation de tableau de ligne) est survenu et a mélangé (*i.e.* permuté) les lignes de votre fichier texte. Heureusement, vous les aviez numérotées au préalable. Afin de pouvoir réparer les dégâts, codez une fonction `trie` qui trie les lignes d'un tableau en supposant qu'elles commencent toutes par un numéro, c'est-à-dire qui range dans chaque case d'indice `i` du tableau la ligne commençant par le numéro `i`. On suppose bien sûr que les numéros des lignes sont tous dans l'intervalle $[0..l-1]$ où l est le nombre de ligne, et deux à deux distincts. *Notez qu'échanger deux lignes ne nécessite pas de copier leurs caractères, seulement d'intervertir les pointeurs dans les cases correspondantes du tableau de ligne.*

Question 12 💻
Afin de tester votre fonction, créez à la main un fichier permuté (par exemple en permutant les lignes d'un fichier que vous avez numéroté automatiquement). Créez un programme capable de trier ce fichier et de l'afficher ensuite. Vérifiez sur l'exemple.

Question 13 🖳

Codez une fonction `affiche_fichier_sans_num` qui affiche un fichier à partir de son tableau de lignes en enlevant les numéros qu'on suppose être inscrit au début de chaque ligne. *Attention, le numéro d'un ligne peut avoir 2 voir trois chiffres... et le premier numéro, 0, a été écrit avec un chiffre.*

Question 14 🖳

Téléchargez le fichier nommé `code_melange.c` sans l'ouvrir. Appliquez lui la procédure de tri, puis enregistrez le sans numéro de ligne sous un nouveau nom. Compilez ce fichier, puis exécutez le