TD5 – Pointeurs et allocation dynamique

Langage C (LC4)

semaine du 5 mars

1 Chaînes de caractères

Dans la biliothèque `<string.h>`, les chaînes de caractères sont définies comme des tableaux de caractères dans lesquels la fin de la chaîne est signalée par le caractère de code 0, noté `'\0'`.
Quelques remarques préliminaires :
- pour stocker une chaîne de n caractères, il est nécessaire de réserver $n+1$ caractères, pour pouvoir stocker le `'\0'` ;
- ne pas confondre :
  - le caractère `'\0'` ;
  - une chaîne vide, qu'on peut noter `""` et qui est représentée en mémoire par une chaîne dont la première case a pour valeur `'\0'` ;
  - la valeur `NULL`, qui peut être affectée à une variable de type **`char`** `*` et qui indique qu'elle ne pointe vers aucune zone de mémoire.

L'objectif de cet exercice est de manipuler des chaînes de caractères *sans utiliser les fonctions de la bibliothèque `string`*. En particulier, vous devrez récrire vous-mêmes certaines fonctions de cette bibliothèque.
Vous implémenterez les fonctions suivantes.

Question 1. **`int`** `strlen(`**`const char`** `*)` renvoie le nombre de caractères d'une chaîne. Le **`const char`** `*` spécifie que le contenu de la chaîne ne sera pas modifié par la fonction.

Question 2. **`int`** `strcmp(`**`const char`** `*s1,` **`const char`** `*s2)` compare deux chaînes `s1` et `s2` : elle renvoie un nombre négatif si la chaîne `s1` est avant `s2` (pour l'ordre lexicographique, l'ordre d'un dictionnaire usuel), 0 si elles sont égales, et un nombre positif sinon.

Question 3. **`int`** `palindrome(`**`const char`** `*)` (qui n'est pas dans `<string.h>`) teste si une chaîne est un palindrome, c'est-à-dire une chaîne identique qu'on la lise de la gauche vers la droite ou de la droite vers la gauche (par exemple les mots *laval* ou *subitomotibus*).

Question 4. **`char`** `*strchr(`**`const char`** `*s,` **`int`** `c)` renvoie l'adresse de la première occurrence du caractère `c` dans la chaîne `s` en partant du début de la chaîne.

Question 5. **`char`** `*strsep(`**`char`** `**stringp,` **`const char`** `*separateurs)` coupe une chaîne en deux à la première occurrence d'un caractère séparateur.
La chaîne à couper est `*stringp`, la chaîne `separateurs` contient tous les caractères séparateurs. Le premier caractère séparateur trouvé est remplacé par `'\0'` dans `*stringp` et la valeur de `*stringp` est modifiée (d'où l'intérêt de passer l'adresse de la chaîne) pour pointer sur le caractère suivant. L'ancienne valeur de `*stringp` est renvoyée.

2 Allocation dynamique de la mémoire

Dans certains cas, le compilateur ne sait pas combien d'espace mémoire il doit réserver pour l'emplacement désigné par un pointeur. C'est notamment le cas lorsque la taille des données dépend des arguments fournis à l'exécution du programme (les fameux `argc` et `argv` du TP5), ou encore lorsqu'on travaille sur des types construits (union, structure).

Il est alors nécessaire de réserver soi-même l'espace mémoire dont on a besoin. À cet effet, `<stdlib.h>` contient les 4 fonctions suivantes (où `size_t` est un alias de **`unsigned long int`**) :

- **`void`** `*malloc(size_t size)` qui alloue `size` octets et renvoie un pointeur vers le début de la zone allouée ;
- **`void`** `*calloc(size_t nmemb, size_t size)` qui alloue un tableau de `nmemb` éléments, chacun de taille `size` octets, y initialise tous les bits à 0, et renvoie un pointeur vers le début de la zone allouée ;
- **`void`** `*realloc(`**`void`** `*ptr, size_t size)` qui change l'espace mémoire alloué pour `ptr` pour une nouvelle taille `size` et y recopie le contenu de l'ancien espace mémoire dans la limite du minimum des deux tailles ;
- **`void`** `free(`**`void`** `*ptr)` qui libère l'espace mémoire pointé par `ptr`.

Lien entre tableaux et pointeurs : En réalité, pour le langage C, un tableau est simplement un autre manière de voir un pointeur : le pointeur est une adresse, on considère que c'est l'adresse de la première case du tableau, les cases suivantes sont les cases contiguës dans la mémoire. Pour allouer dynamiquement un tableau, on est amené à le considérer explicitement comme un pointeur :

```
int taille = ...;
int *tab = calloc(taille, sizeof(int));
```

La notation `tab[i]` est simplement un raccourci pour désigner la case à distance `i` de l'adresse contenue dans la variable `tab`. Elle permet donc d'accéder à l'élément d'indice `i` du tableau si on veut le voir comme un tableau. On rappelle que le premier élément d'un tableau est d'indice 0.
Note sur l'allocation dynamique : pour éviter les erreurs d'allocation, il peut être utile de prendre le réflexe d'utiliser la syntaxe suivante pour faire les allocations dynamiques (avec malloc comme avec calloc) :

```
pointeur = malloc(sizeof(*pointeur));
```

De cette manière, on est assuré que la taille réservée est bien la taille nécessaire, quel que soit le type pointé (rappelons que l'allocation d'une quantité de mémoire insuffisante ne produira pas d'erreur, ni à la compilation, ni même au moment de l'allocation).

3 Vecteurs

Une manière d'implémenter un vecteur consiste à utiliser un tableau à une dimension.

3.1 Réels

Un nombre réel sera représenté par le type **double**.

Question 6. Écrire une fonction `double *alloue_vecteur(int taille)` qui crée un vecteur de réels de taille `taille` initialisé à 0.

Question 7. Écrire une fonction `void libere_vecteur(double *vec, int taille)` qui libère un vecteur de réels de taille `taille`. On veillera à bien libérer toute la mémoire occupée.

3.2 Complexes

Un nombre complexe sera représenté par la structure suivante :

```
struct couple_reels {
  double partie_reelle;
  double partie_imaginaire;
};
typedef struct couple_reels complexe;
```

Question 8. Écrire une fonction `complexe *alloue_vecteur_complexe(int taille)` qui crée un vecteur de complexes de taille `taille` initialisé à $1 + 2i$.

Question 9. Écrire une fonction `void libere_vecteur_complexe(complexe *vec, int taille)` qui libère un vecteur de complexes de taille `taille`. On veillera à bien libérer toute la mémoire occupée.

3.3 Pointeurs

Un pointeur vers un complexe sera représenté par le type suivant :

```
typedef complexe *ptr_complexe;
```

Question 10. Écrire une fonction `ptr_complexe *alloue_vecteur_ptr_complexe(int taille)` qui crée un vecteur de taille `taille` de pointeurs vers des complexes initialisés à $1 + 2i$.

Question 11. Écrire une fonction `void libere_vecteur_ptr_complexe(ptr_complexe *vec, int taille)` qui libère un vecteur de taille `taille` de pointeurs vers des complexes. On veillera à bien libérer toute la mémoire occupée.

4 Chaînes de caractères (suite)

Implémentez...

Question 12. la fonction `char *strcpy(char *dst, const char *src)` copie la chaîne `src` dans `dst`, y compris le caractère de fin de chaîne (et renvoie `dst`). La chaîne `dst` est supposée pointer vers une zone allouée avec une taille suffisante.

Question 13. `char *strcat(char *dst, const char *s)` concatène la chaîne `s` à la suite de `dst` (et renvoie `dst`). Plus précisément, elle écrit par-dessus le caractère `'\0'` à la fin de

`dst` puis ajoute un `'\0'` à la fin de la concaténation.
Comme pour `strcpy()`, la chaîne `dst` est supposée pointer vers une zone allouée avec une taille suffisante.