TD/TP 7 — corrigés partiels

1 Utilisation de malloc

Écrire une fonction qui a le même prototype que `malloc`*, qui appelle* `malloc` *et qui arrête le programme en signalant à l'utilisateur un manque de mémoire si* `malloc` *échoue. Un nom classique pour une telle fonction est* `xmalloc` *; on définit de même* `xrealloc`*.*

```
#include <stdlib.h>
#include <stdio.h>
void *xmalloc(size_t n) {
    void *p = malloc(n);
    if (p == NULL) {
        fputs("Memoire insuffisante.\n", stderr);
        exit(EXIT_FAILURE);
    }
    return p;
}
void *xrealloc(void *p, size_t n) {
    void *q = realloc(p, n);
    if (q == NULL) {
        fputs("Memoire insuffisante.\n", stderr);
        exit(EXIT_FAILURE);
    }
    return q;
}
```

Dans beaucoup de programmes de petite ou moyenne taille, arrêter le programme en cas de manque de mémoire est un comportement acceptable. Dans un gros programme, on peut vouloir se rabattre un algorithme plus lent mais demandant moins de mémoire ; ou (pour un programme interactif) suggérer à l'utilisateur de libérer de la mémoire ; ou sauvegarder l'état du programme pour pouvoir reprendre là où on en était.

Utiliser systématiquement `xmalloc` et `xrealloc` à la place des fonctions standards permet de changer leur comportement si un jour un petit programme s'agrandit et on ne veut plus se contenter de s'arrêter en cas de manque de mémoire.

Il arrive que `malloc` et `realloc` soient utilisées par une bibliothèque qu'on ne maîtrise pas. Avec certaines bibliothèques, on peut changer une variable pour dire d'utiliser une autre fonction à la place de `malloc`. D'autre part, certains systèmes permettent de fournir sa propre version de `malloc` ; voir par exemple `dlopen(3)` et compagnie sur la plupart des Unix modernes.

2 Duplication d'une chaîne de caractères

Écrire une fonction `dupstr` *qui fait une copie d'une chaîne de caractères vers un espace mémoire alloué dynamiquement.*

```
#include <stdlib.h>
#include <string.h>
void *xmalloc(size_t);
char *dupstr(char *s) {
    size_t n = strlen(s);
    char *p = xmalloc(n);
    do { --n; p[n] = s[n]; } while (n != 0);
    return p;
}
```

Cette fonction existe sous le nom `strdup` sous Unix et Posix.

Si on définit sa propre fonction dans un programme qui doit être portable au maximum, il vaut mieux ne pas l'appeler `strdup`. En effet, sur certains systèmes, ceci créerait un conflit entre un `strdup` défini par la bibliothèque C du système et le `strdup` du programme.

D'une manière générale, la norme du langage C réserve certains noms de variables et de fonctions à la bibliothèque standard, et il vaut mieux éviter de les redéfinir. En simplifiant, il s'agit :

- des noms commençant par un trait de soulignement (`_`) ;
- des noms de variables, fonctions et macros fournis par la bibliothèque standard ;
- de quelques familles de noms, par exemple les noms commençant par `E` suivi d'une lettre majuscule ou d'un chiffres (valeurs de `errno`), les noms commençant par `str` ou `is` suivi d'une lettre, etc.

3 Lecture d'une ligne complète

Écrire une fonction `getline` *qui lit en mémoire une ligne d'un fichier texte et la stocke dans un espace mémoire alloué dynamiquement.*

La fonction standard `fgets` lit une ligne, mais il faut prévoir à l'avance une taille maximale ; si la ligne est trop longue, `fgets` s'arrête au milieu. La fonction `getline` lit une ligne entière quelle que soit sa taille.

Le principe général du code de `getline` est : on lit un caractère, et si ce n'est pas un saut de ligne, on le place dans le tampon. Quand on trouve un saut de ligne, on le lit mais on ne le place pas dans le tampon ; et on place un caractère nul à la fin du tampon. Noter que `getline` renvoie `NULL` si on est à la fin du fichier, mais renvoie le contenu de la dernière ligne même si celle-ci n'est pas terminée par un saut de ligne (normalement, un fichier *texte* non vide se termine par un saut de ligne).

Si le tampon s'avère trop petit, on augmente sa taille avec `xrealloc`. La variable `taille` contient la taille du tampon, tandis que la variable `n` contient le nombre de caractères déjà lus. La stratégie adoptée ici est de doubler la taille à chaque fois. Ceci a l'inconvénient d'occuper jusqu'à deux fois plus de mémoire que nécessaire, mais l'avantage de ne pas faire beaucoup d'appels à `realloc` même si la ligne est très longue. On pourrait aussi augmenter la taille d'une quantité fixe, ou adopter une stratégie plus compliquée.

Noter qu'on a l'invariant de boucle `taille` > `n`. C'est vrai au début parce qu'on commence avec un tampon non vide, et ça reste vrai parce qu'on teste la taille du tampon dans la boucle après avoir ajouté le caractère. Cette propriété assure qu'on a la place de stocker le caractère nul à la fin.

Rappel : la variable `x` doit être un `int`, et pas un `char`, sous peine de croire qu'on est arrivé à la fin du fichier quand on rencontre un caractère dont la valeur est `EOF` (typiquement, ÿ en latin1).

```
#include <stdlib.h>
#include <stdio.h>
void *xmalloc(size_t);
void *xrealloc(void*, size_t);
char *getline(FILE * f) {
    size_t taille = 4096, n = 0;
    char *tampon = xmalloc(taille);
    int x;
    if (feof(f)) return NULL;
    while ((x = getc(f)) != EOF &&
           x != '\n' && x != 0) {
        tampon[n++] = x;
        if (n == taille) {
            taille *= 2;
            tampon = xrealloc(tampon, taille);
        }
    }
    tampon[n++] = 0;
    return xrealloc(tampon, n);
}
```

4 Recherche d'une sous-chaîne

Écrire une fonction qui recherche une chaîne `mot` *dans une autre* `s`*. Renvoyer un tableau contenant la liste de toutes les positions (pointeurs sur un caractère de* `s`*), suivies d'un* `NULL` *pour marquer la fin du tableau.*

On reprend la fonction de recherche d'une sous-chaîne du TP 6, et on la modifie pour stocker dans un tableau l'adresse de chaque occurence.

Dans le tableau `occurences`, on prévoit de la place pour `ls`+1 occurences (plus un `NULL`), ce qui est le maximum (atteint si `mot` est vide)). C'est perdre beaucoup de mémoire dans le cas fréquent ou `s` peut être long mais il n'y a pas beaucoup d'occurences. Exercice complémentaire : modifier la fonction pour qu'elle ne consomme pas beaucoup plus de mémoire que nécessaire (pour une définition raisonnable de « pas beaucoup »).

```
#include <string.h>
void *xmalloc(size_t);
char **indice_sous_chaine (char *s, char *mot) {
    size_t i, k, o;
    size_t lm = strlen(mot), ls = strlen(s);
    char **occurences =
        xmalloc((ls+2)*sizeof(*occurences));
    for (i = 0; i + lm <= ls; i++) {
        for (k = 0; mot[k] != 0; k++)
            if (s[i+k] != mot[i]) break;
        if (mot[k] == 0) {
            occurences[o] = s + i;
            ++o;
        }
    }
    occurences[o] = NULL;
    return xrealloc(occurences,
                    (o+1)*sizeof(*occurences));
}
```