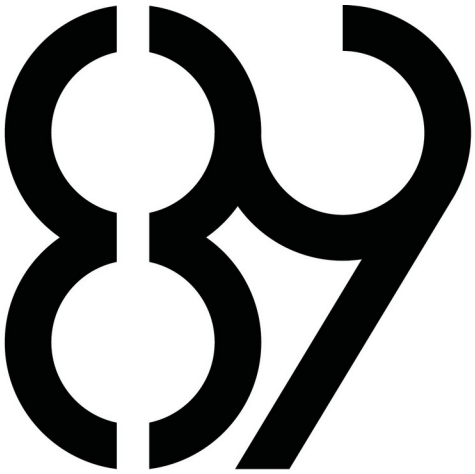
DAEMONLAB



スプリートの試験

La mémoire tu apprendras à manipuler

- DaemonLab -
pedagogie@ecole-89.com

Vous l'avez déjà fait deux fois, mais c'est pas fini.

INDEX

Avant-propos :

01 – Détails administratifs

Votre travail doit être envoyé via l'interface de ramassage de **l'Infosphère** :

Pour cette activité, vous rendrez votre travail sous la forme d'une archive au format `.tar.gz`. Cette archive devra contenir l'ensemble de votre travail tel que demandé dans la section 4.

Pour créer cette archive `.tar.gz`, il vous suffit d'utiliser la commande suivante :

```
$ tar cvfz mon_archive.tar.gz fichier1 ficher2
fichier3
```

Le nom « mon_fichier.tar.gz » étant à remplacer par le nom que vous souhaitez donner votre fichier archive, et « fichier1 », « fichier2 », « fichier3 » par les fichiers ou dossiers que vous souhaitez mettre dans cette archive. Vous pouvez vérifier le contenu de votre archive à l'aide de la commande `tar -t mon_archive.tar.gz`.

Ce travail est à effectuer seul. Vous pouvez bien sur échanger avec vos camarades, néanmoins vous devez être l'auteur de votre travail. Utiliser le code d'un autre, c'est **tricher**. Et tricher annule **toutes** les médailles que vous avez reçus sur l'activité. La vérification de la triche est réalisée de la même manière que la correction : de manière **automatique**. Prenez garde si vous pensez pouvoir passer au travers.

Médailles accessibles :

Réussir à rendre :
Vous avez réussi à envoyer votre travail au système de correction.

02 – Propreté de votre rendu

Votre rendu, c'est-à-dire le contenu de l'archive ou du dépôt que vous entrez sur l'interface du TechnoCentre, doit respecter **strictement** l'ensemble des règles suivantes :

- Il ne doit contenir **aucun** fichier objet. (*.o)
- Il ne doit contenir **aucun** fichier tampon. (*~, #*#)
- Il ne doit pas contenir votre production finale (programme ou bibliothèque)

La présence d'un fichier interdit mettra immédiatement fin à votre évaluation.

Médailles accessibles :

Rendu propre
Votre rendu respecte les règles de propretés imposées.

03 – Règlement quant à la rédaction du code C

Votre programme doit respecter la **Table des Normes**.

04 – Construction de votre rendu

Le programme de correction va construire une sous-partie déterminée de votre rendu afin d'effectuer des tests dessus. En voici les paramètres :

- Les fichiers qui seront compilés sont ceux qui auront l'extension `*.c`.
- Seuls les fichiers dans le dossier `./` seront compilés.
- Les fichiers seront compilés avec les drapeaux `-W -Wall -Werror`.
- Tous les fichiers seront compilés de façon **individuelle**, avec un `main` de test, sauf mention contraire dans l'exercice.

L'ensemble du code que vous rendez doit pouvoir être compilé.
En cas d'échec de la compilation, vous ne serez pas évalué.

Médailles accessibles :

Construction partielle
Les éléments requis de votre projet se construisent séparément.

05 – Fonctions autorisées

Vous n'avez le droit à aucune autre fonction que celle précisée dans la liste ci-dessous :

`malloc` `free`

L'utilisation d'une fonction interdite est assimilée à de la triche.
La triche provoque l'arrêt de l'évaluation et la perte des médailles.

Au cours de la l'examen, si vous estimez qu'une fonction interdite est nécessaire, vous pouvez demander à ce qu'elle soit autorisée, *en vous justifiant*.

06 — Avant-propos sur la méthodologie

Lisez-moi, sinon conséquences

L'objectif de cet examen est de vous faire pratiquer les notions autour de l'allocation dynamique de mémoire sur le tas et l'utilisation de tableau de caractères et de pointeurs.
C'est pour cela que la plupart des consignes qui vont suivre vous sont familières.

Vous allez être évalués sur votre capacité à reproduire les comportements demandés, dans un premier temps, et sur votre capacité à gérer votre mémoire, dans un second.

À chaque fois que vous faites un appel à `malloc`, vous devez enregistrer son retour dans un pointeur que votre code ne perdra pas, de façon à ce que cette mémoire puisse être libérée par la suite.

Avant chaque exercice, prenez bien soin de faire l'inventaire des différents cas d'utilisation de la fonction.

Avant de coder une fonction qui vous demande d'allouer de la mémoire, demandez-vous quelle quantité vous devez allouer, et dans quelles circonstances.

Toutes les informations qui suivent sont données à titre purement indicatif. N'essayez pas de reproduire ces pratiques sans ~~la présence d'un adulte~~ avoir cherché à les comprendre avant.

Dans le cas de `split`, — je vais vous aider – vous pouvez prévoir la taille de la première dimension de votre tableau, en comptant le nombre de jeton `token` dans la chaîne `str` passée en paramètre.

Si vous croisez 2 jetons, cela veut dire que vous aurez 3 sous-chaînes maximum – si tenté qu'il y ait du texte avant et après chaque jeton.

Maintenant, rappelez-vous qu'il faut stocker un pointeur `NULL` à la fin d'un tableau de pointeurs : vous arriverez à la conclusion que vous devez allouer 4 `char *` dans votre premier tableau.

Synthèse : pour deux jetons dans rencontrés dans ma chaîne de caractères, j'alloue assez de place pour trois sous-chaînes et un pointeur nul, soit un total de quatre pointeurs `char *`.

La seconde dimension de votre tableau doit être allouée pour chaque sous-chaîne. Il faut compter – avec un `strlen` spécialement modifié, par exemple – le nombre de caractères entre votre tête de lecture et le prochain jeton délimiteur.

Rappel, vous pouvez faire croire à une fonction que votre chaîne de caractère commence plus loin en lui passant un pointeur vers un caractère bien spécifique : `strlen(&str[2]);` ferait croire que à `strlen` que `str` commence à son troisième caractère.

Rappel bis, lorsque vous allouez de la place pour une chaîne de caractère, n'oubliez pas de compter l'octet `\0` final.

Rappel ter, divisez les tâches compliquées en un maximum de sous tâches (de sous fonctions.) Par exemple : le fait de créer une sous chaîne à partir de `&str[i]` et `token`, peut être délégué à une sous fonction qui renvoie un nouveau `char *`, que vous n'aurez qu'à mettre dans une case de votre premier tableau. Une seconde fonction, pourrait être utilisée pour faire avancer votre tête de lecture `i` jusqu'après le prochain jeton dans `str`, soit en passant un nouveau `i` en valeur de retour, soit en prenant un pointeur vers ce dernier et en en modifiant le contenu.

Toute implémentation reste à votre discrétion. Encore une fois, ce sont seulement des pistes que l'on vous donne.

Information complémentaire

Si vous avez déjà obtenu la médaille pour un exercice, vous n'avez pas besoin de le rendre à nouveau.

Les médailles suffixées d'un **.x** indiquent que vous avez partiellement réussi l'exercice et que vous devez le refaire jusqu'à passer tous les tests essentiels.

07 – `puts`

Fichier à rendre : **`puts.c`**

Vous devez ré-implémenter la fonction `e89_puts`. Elle doit se comporter comme `puts` de la bibliothèque standard.

Son prototype est le suivant :

```
int e89_puts(const char *s);
```

Pour rappel, la fonction doit afficher la chaîne de caractères passée en paramètre, sur la sortie standard, et la faire suivre d'un retour à la ligne.

Sa valeur de retour est le nombre de caractère total qu'elle a écrit.

Le **\n** compte comme un caractère écrit.

Pensez au comportement de votre fonction, si elle reçoit une chaîne vide.
(Notez qu'une chaîne vide et un pointeur nul, ça n'est pas la même chose.)

08 – `strcmp`

Fichier à rendre : **`strcmp.c`**

Vous devez ré-implémenter la fonction **`e89_strcmp`**. Elle doit se comporter comme **`strcmp`** de la bibliothèque standard.

Son prototype est le suivant :

```
int e89_strcmp(const char *s1, const char *s2);
```

Pour rappel, la fonction doit renvoyer :
— 0 lorsque les deux chaînes sont égales ;
— une valeur non-nulle quand les chaînes sont différentes – peu importe la valeur pour la moulinette, étant donné que le standard ne donne pas de consignes à ce sujet.

Pensez au comportement de votre fonction lorsqu'on lui passe une ou des chaînes vides.
Exemple : **`e89_strcmp("", "");`**

09 – `strndup`

fichier à rendre : **`strndup.c`**

Vous devez implémenter la fonction `e89_strndup`, qui alloue une nouvelle chaîne de la taille `n` demandée, et la remplit avec le contenu de `str`, sans jamais copier plus de `n` caractères. Une fois le `\0` de `str` atteint, même si `n` caractères n'ont pas encore été copiés, il convient de s'arrêter et de renvoyer la chaîne résultante.

Pensez *toujours* à terminer vos chaînes par un octet nul, **`'\0'`**.

Prototype de la fonction :

```
char *e89_strndup(const char *src, int n);
```

10 – substrs_len

fichier à rendre **substrs_len.c**

Implémentez la fonction `substrs_len`. Elle doit renvoyer un tableau d'entiers, alloué dynamiquement, qui contient la longueur de chaque sous chaîne rencontrée dans `str`, en utilisant pour délimiteur le caractère `token`. La dernière case du tableau doit être mise à `-1`.

Pour la chaîne `"bonjour;miaou;au revoir;4"` le résultat serait un tableau d'entiers contenant : *7*, *5*, 9, 1.

Un test de la fonction pourrait alors ressembler à :

```
char *str = "bonjour;miaou;au revoir;4"
int *lens = substrs_len(str, ';');

assert(lens);
assert(lens[0] == 7);
assert(lens[1] == 5);
assert(lens[2] == 9);
assert(lens[3] == 1);
assert(lens[4] == -1);
free(lens);
```

11 – `split`

Fichier à rendre : **`split.c`**

Vous devez implémenter la fonction **`split`**. Elle doit éclater la chaîne passée en paramètre en plusieurs sous-chaînes. En utilisant le délimiteur `token`.

Chaque sous-chaîne doit être allouée dynamiquement.

Son prototype est :

```
char **split(const char *str, char token);
```

Exemple

Soit :
— une chaîne `str` contenant la succession de caractères : `"abc;def;ghi"` ;
— un caractère `token` contenant ';'.

Un appel à `split` avec pour paramètres la chaîne `str` et le caractère `token` renverrait : un tableau de chaînes de caractères contenant :
— `"abc"`,
— `"def"`,
— `"ghi"`,
— `NULL`.

12 – `split_len`

Fichier à rendre : **`split_len.c`**

Vous devez implémenter la fonction **`split_len`**. Elle doit prendre en paramètre un double tableau de caractères et renvoyer le nombre de sous-chaînes qu'il contient, en excluant le dernier pointeur nul.

Son prototype est le suivant :

```
int split_len(const char **);
```

Pour l'exemple précédent, cette fonction renvoie 3.

13 – `split_print`

Fichier à rendre : **`split_print.c`**

Vous devez implémenter la fonction **`split_print`**. Elle doit prendre en paramètre un double tableau de caractères et afficher, ligne par ligne, chacune des entrées du tableau.

Son prototype est le suivant :

```
void split_print(const char **);
```

Exemple

Pour un tableau contenant les chaînes `"abc"`, `"def"`, `"ghi"`, `NULL` l'affichage serait :

```
abc
def
ghi
```

N'oubliez pas les sauts de ligne après chaque ligne !

14 – `split_free`

Fichier à rendre : **`split_free.c`**

La fonction `split_free` a pour objectif de libérer toute la mémoire allouée pour un double tableau de caractères.

Son prototype est le suivant :

```
void split_free(char **);
```

Pour valider cet exercice, vous devez faire en sorte qu'aucune zone de mémoire allouée avec `malloc` ne soit oubliée.

Cela veut aussi dire que votre fonction `split` ne doit pas avoir de pertes mémoires (soit de variables contenant des pointeurs issus de `malloc`, qui auraient été réassignés ou perdues)

Pour libérer de la mémoire, allouée dynamiquement, on utilise **`free`**. Autant d'appels à **`free`** sont nécessaires que d'appels à **`malloc`**.