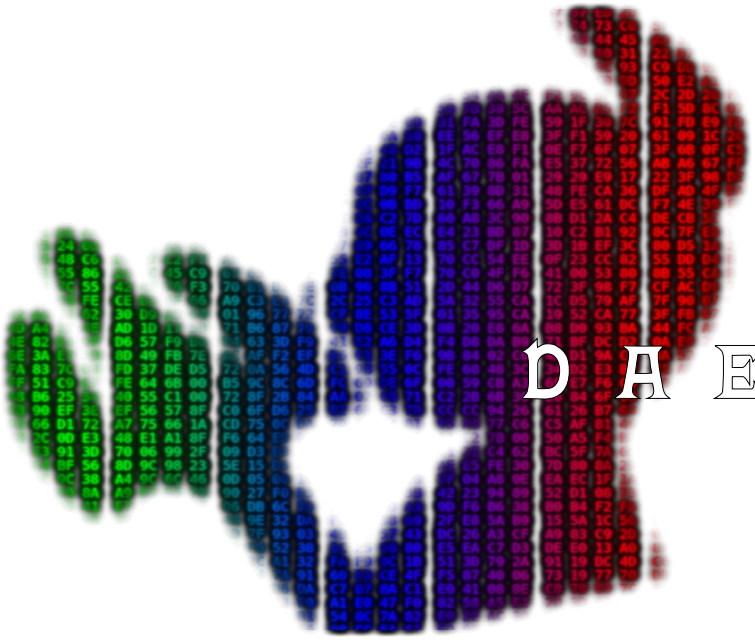

DAEMONLAB

Marathon de programmation système

- DaemonLab -
pedagogie@ecole-89.com

Deux exercices d'une heure, un de deux. Top chrono. Go !

INDEX

01 – Avant-propos

Votre travail doit être rendu via les dossiers **~/marathon/claviercran1/** dans votre espace personnel.

Si vous faites erreur et que le dossier que vous utilisez pour votre rendu est différent, vous ne serez pas évalué faute d'avoir pu trouver votre travail.

Ce travail est a effectuer en binôme. 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 reçu 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.

Votre rendu 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.

Votre programme doit respecter les Tables de la Norme dans leur intégralité. Vous êtes invité à les observer depuis **l'Infosphère**. Elles sont disponibles comme ressource de cette activité.

02 – Fonctions autorisées

La bibliothèque logicielle venant avec le C est vaste et disponible. La LibLapin, que vous utilisez dans vos projets multimédia, est également vaste… Cependant nous avons fait le choix de vous interdire leurs utilisation intégrales, afin de vous amener progressivement à reprogrammer vous même ses fonctionnalités les plus utiles.

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.

Vous n'avez le droit d'utiliser aucune fonction issue de la LibC ou de la LibLapin à l'exception de celles que nous vous autoriserons explicitement.

Pouvoir utiliser une fonction ne signifie pas nécessairement que celle-ci soit utile à votre cas.

Pour cette activité, issu de la LibC, vous n'avez le droit qu'à la liste suivante :

- open
- close
- read
- srand
- cos
- atan2
- write
- alloca
- atexit
- rand
- sin
- sqrt

Remarquez bien l'absence de **malloc** et de **free**. En effet ils sont **interdits** ! Issu de la LibLapin, vous avez le droit aux fonctions suivantes :

- bunny_start
- bunny_stop
- **bunny_malloc**
- **bunny_free**
- bunny_new_pixelarray
- bunny_delete_clipable
- bunny_blit
- bunny_display
- bunny_set_*_function
- bunny_set_*_response
- bunny_loop
- bunny_create_effect
- bunny_compute_effect
- bunny_sound_play
- bunny_sound_stop
- bunny_delete_sound
- bunny_open_configuration
- bunny_delete_configuration
- bunny_configuration_getf
- bunny_configuration_setf
- bunny_configuration_*
- bunny_set_memory_check
- bunny_release
- bunny_usleep

La suite sur la page d'après.

Pour utiliser **bunny_malloc**, vous pouvez soit programme directement avec, soit en utilisant le modèle de projet qui vous a été transmis, mettre 1 dans la variable de Makefile **BMALLOC**, qui transformera **malloc** en **bunny_malloc**.

Vous appellerez **bunny_set_memory_check** au début de votre fonction **main** de sorte à provoquer une vérification de vos allocations à la fermeture du programme.

bunny_malloc, par défaut, limitera votre consommation de RAM à 20Mo.

Pour information :
Une image en 1920*1080 fait environ 8Mo.

Une musique en 44kHz de 1 minute en stéréo fait environ 10Mo.

Vous devrez donc disposer d'une discipline de fer avec vos allocations... et probablement trouver des compromis.

L'utilisation de **bunny_malloc** parfois **cachera** des erreurs dans votre programme, et parfois en **révélera** : son principe d'allocation était différent de **malloc**, il sera parfois plus « fort » ou plus « faible ». Ne vous mentez pas à vous en disant « l'utilisation de **bunny_malloc** fait planter mon programme », ce n'est pas **bunny_malloc**, c'est *vous*.

N'hésitez pas à l'activer, à le désactiver (Et lorsqu'il est désactivé, à utiliser valgrind). Et n'oubliez pas que désormais, votre demande de RAM a de véritables chances d'échouer. Car exploiter aussi peu de RAM, cela va très vite…

Dans votre rendu, il ne devra pas y avoir la moindre trace de malloc.

03 – Méthode de construction

Il peut vous être demandé d'écrire des programmes ou des fonctions.

Dans le cas des programmes, il vous sera toujours demandé de fournir un **dossier** pour l'exercice le requérant. Un **Makefile** vous sera également demandé. Le **nom du programme** de sortie vous sera précisé à chaque fois. Un Makefile incorrect, un mauvais nom de programme, et votre correction n'aura pas lieu…

Dans le cadre des fonctions, il vous ai demandé de fournir le fichier dans votre **dossier de bibliothèque personnelle**, de sorte à ce que vous puissiez utiliser toutes les fonctions que vous avez déjà réalisé jusqu'ici. Pour rappel, le dossier de votre bibliothèque doit être placé à la racine de votre espace personnel et s'appeler **libstd/.**

N'oubliez pas d'entretenir avec soin votre dossier **libstd/** de sorte à ce qu'il soit toujours propre, respecte la norme et soit en état de compiler… sans quoi elle fera obstacle à la correction.

Votre compilation devra toujours comporter les options **-W**, **-Wall** et **-Werror**.

Dans le cadre de la programmation multimédia, le système de correction établira toujours la variable d'environnement **BMALLOC** à 1. Si vous utilisez le modèle de projet, cela provoquera l'utilisation de **bunny_malloc** dans votre bibliothèque personnelle comme dans votre projet rendu.

06 – Exercice 1

Votre programme prendra en paramètre un nombre indéterminé de fichiers.

Il devra afficher sur le terminal leur taille en **octet** avant de sauter une ligne puis d'afficher le contenu de ces fichiers. Un saut de ligne final clôturera l'affichage du fichier. Votre programme continuera tant qu'il n'aura pas épuisé ses paramètres.

L'ordre de lecture des paramètres sera fait de 1 jusqu'à **argc** non compris.

07 – Exercice 2

Votre programme prendra en paramètre des chemins de **dossiers**. Vous afficherez le contenu de ces dossiers tel qu'il sera renvoyé par les fonctions de parcours de dossiers. *Il n'est **pas** demandé de parcourir récursivement les sous-dossiers que vous rencontrerez, seulement les dossiers passés en paramètre !*

Votre affichage fonctionnera de la façon suivante : vous afficherez le paramètre suivi du symbole ':' suivi d'un saut de ligne. Ensuite, chaque fichier ou dossier sera affiché. Les fichiers et dossiers seront séparé par un unique **espace**. Il ne **devra pas** y avoir d'espace en fin de ligne, mais un **saut de ligne** qui conclura le parcours du paramètre.

Les noms des dossiers **rencontrés** seront suffixés par '/'.

Vous recommencerez tant que vous n'aurez pas parcouru **argv** de 1 jusqu'à **argc** exclu.

08 – Exercice 3

Cet exercice dure 2 heures

Votre programme prendra en paramètre deux fichiers. Sa tache sera de parcourir octet par octet les deux fichiers. Tant que les octets sont identiques dans les deux fichiers, il se contente d'afficher les informations suivantes : à gauche le premier fichier, à droite le second. Les octets sont affichés **en base 10, et seulement en positif**.

Dans un premier temps, réalisez seulement cette partie là : lisez les deux fichiers **en même temps**, en affichant les valeurs de leurs octets.

Une fois terminé cette première partie, vous allez travailler sur ce qu'il se passe lorsque la différence est rencontrée :

Si il y a une différence, il affiche un prompt '!' et attend une entrée de l'utilisateur. L'utilisateur peut alors entrer un symbole au choix parmi '<' et '>'. Tout autre symbole est rejeté et le prompt est affiché de nouveau.

Le symbole '<' indique que c'est le premier fichier, celui de gauche, qui est « bon » et celui de droite qui présente un « problème ». Cela provoque le déplacement de la tête de lecture du fichier de droite de 1 octet, sans faire progresser celle de gauche. Le symbole '>' indique que c'est le second fichier, celui de droite, qui est « bon » et donc c'est à l'inverse le fichier de gauche qui verra sa tête de lecture avancer d'un octet.

Si après avoir progressé, les octets sont de nouveau identique, on continue à avancer automatiquement jusqu'à la fin ou la prochaine différence.

Si la fin de l'un des deux fichiers devait être atteinte avant l'autre, il serait simplement affiché quel fichier est terminé et combien d'octets restait-il pour celui qui ne l'était pas. Si les deux fichiers sont terminés en même temps, rien n'est affiché.

Voici un exemple qui vous permettra de comprendre les subtilités de la méthode d'affichage attendue. En noir, ce sont les entrées utilisateurs.

```
$> ./manualdiff fichier1 fichier2
127 127
240   3
!<
240 240
 25  25
174 174
 56 100
!>
 88 100
!>
100 100
fichier1 terminated
fichier2 12 bytes before end
$>
```