D A E M O N L A B

Journée 1

Variables, conditions, boucles

- DaemonLab -

Durant cette journée, vous allez pour la plupart d'entre vous programmer pour la première fois. Nous verrons ensemble les éléments fondamentaux d'un programme.

Ce document est strictement personnel et ne doit en aucun cas être diffusé.

INDEX

01 – Avant-propos

Votre travail doit être rendu via le dossier **~/hyperespace/j1/** 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 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 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. Nous avons cependant fait le choix de vous interdire son utilisation, 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 à l'exception de celles que nous vous autoriserons explicitement. Dans le cadre de cette activité, la seule fonction à laquelle vous aurez le droit est l'appel système **write**.

Vous pouvez observer le fonctionnement de **write** en tapant « **man 2 write** » dans votre terminal, néanmoins, afin de vous faciliter son utilisation au départ, nous vous avons préparé une fonction que vous êtes libre d'intégrer à votre code.

Cette fonction est **tc_putchar**, elle tente d'écrire le caractère passé en paramètre sur la sortie standard (le terminal, par défaut). Elle renvoi 1 si elle a réussie à écrire le caractère, sinon elle renvoi 0. Ci-dessous, le code de **tc_putchar** suivi d'un programme de test.

```
#include <unistd.h>

int tc_putchar(char c)
{
     return (write(1, &c, 1) > 0);
}
```

```
int main(void)
{
     tc_putchar('A');
     tc_putchar('\n');
     return (0);
}
```

03 – write_alphabet

Fichier à rendre : write_alphabet.c

Pour réussir cet exercice, vous devez afficher l'alphabet en minuscule sur le terminal, suivi d'un saut de ligne. Ci-dessous, vous trouverez les commandes pour compiler votre programme, pour exécuter votre programme ainsi que la sortie du programme telle qu'attendue.

```
$> ls
write_alphabet.c
$> gcc write_alphabet.c -W -Wall -Werror
$> ./a.out
abcdefghijklmnopqrstuvwxyz
$>
```

« a.out » est le nom par défaut donné par gcc au programme qu'il produit en l'absence de l'option « -o ».

Médailles accessibles :

Premier programme !
Vous venez de réaliser le premier programme informatique de votre scolarité.

putchar
Vous avez su utiliser la fonction putchar.

Boucle fixe
Vous êtes parvenu à programmer une boucle simplement bornée.

04 – write_descending_digits

Fichier à rendre : write_descending_digits.c

Cette fois, vous devrez afficher les chiffres de 9 à 0, suivi d'un saut de ligne.

```
$> gcc write_descending_digits.c -W -Wall -Werror
$> ./a.out
9876543210
$>
```

05 – write_halfabet

Fichier à rendre : write_halfabet.c

Nous sommes de retour sur l'alphabet, mais cette fois, nous ne désirons afficher qu'une seule lettre sur deux en partant de la lettre a. Nous finirons encore sur un saut de ligne.

```
$> gcc write_halfabet.c -W -Wall -Werror -o halfabet
$> ./halfabet
acegikmoqsuwy
$>
```

Remarquez l'utilisation de l'option « **-o** » avec un paramètre « **halfabet** », faisant générer un fichier nommé **halfabet** au lieu de **a.out** à gcc.

Médailles accessibles :

Condition simple
Vous êtes parvenu à écrire une condition simple.

06 – write_waving_alphabet

Fichier à rendre : write_waving_alphabet.c

Dernier exercice sur l'alphabet : cette fois, au lieu de faire disparaître les lettres pair, nous allons les mettre en majuscule. Nous terminerons sur un saut de ligne.

```
$> gcc write_waving_alphabet.c -W -Wall -Werror
$> ./a.out
aBcDeFgHiJkLmNoPqRsTuVwXyZ
$>
```

07 – repeat_char

Fichier à rendre : repeat_char.c

Nous avons écrit assez de programme pour aujourd'hui. Désormais, nous ferons des fonctions à la place. Pour commencer, écrivez la fonction **repeat_char** : cette fonction affiche un caractère passé en paramètre autant de fois qu'indiqué par un second paramètre.

Le prototype de la fonction **repeat_char** est le suivant :

```
void std_repeat_char(char c, int nbr)
```

Cette fonction doit fonctionner pour toute valeur de c et pour toute valeur positive de **nbr**. Si **nbr** est négatif, la fonction ne doit rien afficher.

Le fait que ce programme ne comporte pas de fonction **main** a pour conséquence que vous ne pouvez pas le compiler tout seul. Ce n'est pas un problème pour la correction qui apportera sa propre fonction **main**, mais cela va vous empêcher de tester.

Pour pouvoir tester quand même, vous écrirez un « **main de test** » qui appellera la fonction que vous avez écrit pour pouvoir la tester. Vous ne devrez pas le rendre. Voici par exemple, un main de test pour la fonction **repeat_char** :

```
int main()
{
	std_repeat_char('a', 4); // affiche aaaa
	std_repeat_char('z', 3); // affiche zzz
	std_repeat_char('$', 0); // n'affiche rien
	std_repeat_char('_', -1); // n'affiche rien
	return (0);
}
```

Médailles accessibles :

Procédure
Vous venez d'écrire une fonction ne renvoyant aucune valeur.

Boucle paramétrée
Vous venez d'écrire une boucle dont l'une des bornes n'était pas connue à l'avance :

08 – abs

Fichier à rendre : abs.c

Écrivez la fonction **std_abs** qui renvoi la valeur absolue de la valeur reçu en paramètre. Voici quelques exemples de valeur absolue :

- La valeur absolue de 5 est 5.
- La valeur absolue de -8 est 8.

En plus d'écrire la fonction **std_abs**, vous allez devoir écrire une fonction de test pour la fonction **tc_abs** ! Qu'est ce que ca veut dire ? Que le **TechnoCentre**, le système de correction va fournir une fonction **tc_abs** dont vous devrez vous assurer du fonctionnement !

Le système de correction s'arrangera pour que parfois, sa fonction marche et parfois pas… votre fonction de test, **test_abs** devra déceler quand est ce qu'elle marche et quand est ce qu'elle ne marche pas ! Elle renverra **1** quand la fonction semble fonctionner, et **0** quand elle detectera qu'elle ne fonctionne pas.

Le prototype des **std_abs** est le suivant. Nous vous fournissons aussi une version incomplète de **test_abs**.

```
int std_abs(int nbr);

int test_abs(void)
{
	if (tc_abs(-5) != 5)
		return (0);
	// Compléter ici
	return (1);
}
```

Évidemment, vous pouvez utiliser cette fonction **test_abs** pour tester votre propre fonction **std_abs** ! L'idée étant ici de vous apprendre à tester !

Pendant que vous travaillez, disposez de votre fonction **main** pour pouvoir tester, ainsi que de fonctions de test testant vos fonctions, et avant de rendre votre travail, effacez votre main et modifiez vos fonctions de test pour qu'elle corrigent celle du TechnoCentre à la place.

Vos fonctions de tests seront alors évalués comme les autres.

Médailles accessibles :

Fonction
Vous venez d'écrire une fonction, avec des paramètres et une valeur de retour.

Test
Vous venez de construire votre premiere fonction de test ! Votre premier pas vers la qualité.

12 – is_printable

Fichier à rendre : is_printable.c

Écrivez la fonction **std_is_printable** et la fonction d'évaluation **test_is_printable** qui évaluera **tc_is_printable**. La fonction **std_is_printable** doit renvoyer vrai si le caractère reçu en paramètre est imprimable. Un caractère est imprimable si il dispose d'un symbole graphique associé, ou si c'est le caractère espace.

Voici le prototype des fonctions :

```
int std_is_printable(char c);
int test_is_printable();
```

13 – is_between

Fichier à rendre : is_between.c

Ecrivez la fonction **std_is_between** et la fonction d'évaluation **test_is_between** qui évaluera **tc_is_between**. La fonction **std_is_between** doit renvoyer vrai si le premier de ses paramètres est situé entre son second et son troisième, égalité incluse.

Voici le prototype des fonctions :

```
int std_is_between(int a, int b, int c);
int test_is_between();
```

Attention ! Le paramètre b peut-être plus grand ou plus petit que c…
Votre fonction is_between doit fonctionner dans les deux cas !

14 – clamp

Fichier à rendre : clamp.c

Vous commencez à avoir l'habitude : ecrivez les fonctions pour l'exercice « clamp ». Vous l'aurez deviné, il s'agit de **std_clamp** et de **test_clamp** (testant toujours **tc_clamp**). Par la suite, nous ne préciserons plus le nom de la fonction de test ni le nom de la fonction testée par la fonction de test : nous supposerons que vous comprendrez par vous-même comment les choses doivent fonctionner.

La fonction **clamp** doit s'assurer qu'une valeur, passée en première en paramètre est bien située dans une intervalle donné ensuite. Si cette valeur est plus petite que la valeur minimale, alors la valeur minimale sera renvoyée. Si cette valeur est plus grande que la valeur maximale, alors la valeur maximale sera renvoyée. Si min est plus grand que max, alors la valeur sera toujours renvoyée directement sans traitement.

Voici le prototype des fonctions, dont la fonction de test, précisé pour la dernière fois :

```
int std_clamp(int a, int min, int max);
int test_clamp();
```

15 – xor

Fichier à rendre : xor.c

Écrivez la fonction suivante **et les tests associés :**

```
int std_xor(int a, int b);
```

Cette fonction doit effectuer une opération logique « OU EXCLUSIF » entre a et b. Pour que cette opération soit vraie (1), il faut qu'un seul des deux paramètres soit vrais. Dans tous les autres cas, la fonction renverra faux (0).

Médailles accessibles :

Logique XOR
Vous avez saisi le sens du OU EXCLUSIF.

16 – rot_minuschar

Fichier à rendre : rot_minuschar.c

Écrivez la fonction suivante **et les tests associés :**

```
char std_rot_minuschar(char c, int n);
```

Cette fonction fait avancer ou reculer le caractère dans c de n lettres. Par exemple :

- Si c vaut ‘e’ et n vaut 3, le résultat sera ‘h’.
- Si c vaut ‘e’ et n vaut -3, le résultat sera ‘b’.
- Si c vaut ‘z’ et que n vaut 1, le résultat sera ‘a’.
- Si a vaut ‘a’ et que n vaut -1, le résultat sera ‘z’.

Si c n’est pas une lettre en minuscule, la fonction renverra immédiatement la valeur contenu dans c.

Médailles accessibles :

Opérateur modulo
Vous avez saisi le sens de l’opérateur modulo.

17 – rclamp

Fichier à rendre : rclamp.c

Ecrivez la fonction suivante **et les tests associés :**

```
char std_rclamp(int a, int min, int max);
```

Cette fonction renvoi max si a est plus petit que min. Elle renvoi min si a est plus grand que max et renvoi a dans tous les autres cas.

18 – write_ascii_table

Fichier à rendre : write_ascii_table.c

Écrivez un programme affichant la table ASCII de la façon suivante :

```
$> gcc write_ascii_table.c -W -Wall -Werror
$> ./a.out
 !"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~
$>
```

Vous remarquerez que le premier symbole est l'espace ' ' et que le dernier est '~'. Faites attention à l'endroit ou sont situés les sauts de lignes.

19 – mod

Fichier à rendre : mod.c

Écrivez la fonction **std_mod**, ainsi que sa fonction de test **test_mod**, testant la fonction **tc_mod** du système de correction. L'objectif de la fonction **mod** est de renvoyer le reste de la division entre les deux paramètres.

Voici le prototype des fonctions :

```
int std_mod(int a, int b);
int test_mod();
```

Entre nombres entiers, les divisions sont dites entières, cela signifie que par exemple, 3 divisé par 2 donne comme résultat 1 et non 1,5. Comme le résultat est 1 et non 1,5, il y a donc un reste après la division. En l'occurrence, le reste ici est de 2, car 3 – 3 / 2 vaut… 2.

Il existe un opérateur permettant d'effectuer cette opération. Vous pouvez également vous en passer, bien entendu, mais gardez à l'esprit que cet opérateur vous sera fort utile par la suite.

20 – write_truth

Fichier à rendre : write_truth.c

Écrivez la fonction **std_write_truth**. Cette fonction écrit 'y' si son paramètre est vrai, sinon elle écrit 'n'.

```
void std_write_truth(int n);
```

21 – write_subpart

Fichier à rendre : write_subpart.c

Écrivez la fonction **std_write_subpart**.

```
void std_write_subpart(char a, char b);
```

Cette fonction écrit tous les caractères situés entre a et b. Si b est plus grand que a, aucun caractère ne doit être écrit.

22 – is_even

Fichier à rendre : is_even.c

Écrivez la fonction **std_is_even** ainsi que sa fonction de test.

```
int std_is_even(int n);
```

Cette fonction renvoi 1 si n passé en paramètre est pair. Sinon elle renvoi 0.

23 – affine

Fichier à rendre : affine.c

Écrivez la fonction **std_affine** ainsi que sa fonction de test associé.

```
int std_affine(int a, int x, int b);
```

Cette fonction renvoi le résultat de l'opération a * x + b.

24 – distance

Fichier à rendre : distance.c

Écrivez la fonction **std_distance** ainsi que sa fonction de test associé.

```
int std_distance(int a, int b);
```

Cette fonction renvoi la différence absolue entre a et b. Par exemple, entre 0 et 3, il y a 3 de distance. Entre -2 et 2, il y a 4.