

TP06

February 20, 2022

Vous êtes libre de la structure de vos fichiers pour ce tp mais si vous vous sentez perdus, voici une marche à suivre plus guidée :

- ouvrir un terminal
 - `mkdir tp06`
 - `cd tp06`
 - `curl -O http://marc.de-falco.fr/assets/Makefile`
- créer un fichier `tp06.c` qui contient une fonction `main` rudimentaire
- on va créer des fichiers autours et compiler le programme avec `Makefile` pour l'exécuter ce sera juste `./tp06`

1 Structures

Il est possible de définir ses propres structures en C qui agrègent d'autre types. Ainsi, on pourra définir une structure `point` permettant de représenter des points du plan :

```
struct point {  
    double x;  
    double y;  
};
```

On peut alors définir un élément de cette structure avec le type `struct point` et on peut accéder à ses champs par l'utilisation de `..`. Pour obtenir le point (3,4) on écrit :

```
struct point p;  
p.x = 3;  
p.y = 4;
```

On peut passer en paramètres une telle structure et la renvoyer en valeur de retour comme ici :

```
struct point cree_point(double x, double y)  
{  
    struct point p;  
    p.x = x;  
    p.y = y;  
    return p;  
}
```

```
void affiche_point(struct point p)  
{  
    printf("(%.f,%.f)\n", p.x, p.y);  
}
```

```

}

struct point milieu(struct point a, struct point b)
{
    struct point m;
    m.x = (a.x + b.x)/2;
    m.y = (a.y + b.y)/2;
    return m;
}

```

```

// exemple
struct point p1 = cree_point(0, 10);
struct point p2 = cree_point(6, 4);
affiche_point(milieu(p1, p2));

```

On remarque que c'est un peu pénible d'écrire `struct point` partout, il est possible d'utiliser un `typedef` pour définir un type synonyme. Si jamais on a une structure `struct t_s` on peut écrire `typedef struct t_s t;` pour pouvoir utiliser `t` à la place. On peut même avoir le même nom comme ici :

```
typedef struct point point;
```

```

point p;
p.x = 2;
p.y = 3;
affiche_point(p);

```

Quand on sépare en `.c` et `.h`, il faut mettre les types et les déclarations (ou prototypes) dans le `.h` et les définitions dans le `.c`. Par exemple, on pourrait écrire un fichier `point.h` contenant

```

struct point {
    double x;
    double y;
};

```

```
typedef struct point point;
```

```
point cree_point(double x, double y);
```

puis un fichier `point.c` contenant

```
#include "point.h"
```

```

point cree_point(double x, double y)
{
    point p;
    p.x = x;
    p.y = y;
    return p
}

```

1.1 Raytracer - Partie 1

Je vous demande ici de créer une bibliothèque complète permettant de travailler sur des vecteurs (ou de manière équivalente des points) dans l'espace. On s'en servira pour construire, au fur et à mesure des TPs, un raytracer.

Je vous donne le type :

```
struct vec3 {
    double x;
    double y;
    double z;
};
```

```
typedef struct vec3 vec3;
```

et vous devez faire une série de fonctions complètes :

- `vec3_create` qui crée un tel vecteur
- `vec3_add` qui renvoie la somme de deux vecteurs
- `vec3_diff` qui renvoie la différence de deux vecteurs/points.
- `vec3_dot` qui renvoie le produit scalaire de deux vecteurs
- `vec3_cross` qui renvoie le produit vectoriel de deux vecteurs (go go wikipedia)
- `vec3_smul` tel que `vec3_smul(a, v)` où `a` est un double et `v` un vecteur renvoie le vecteur `a v` (comme `2v`, `-3.5v`, ...)
- `vec3_norm` qui renvoie la norme d'un vecteur **on pourra utiliser `sqrt` déclaré dans `math.h` pour la racine**
- `vec3_normalize` qui prend un vecteur `v` et renvoie $v/||v||$ si `v` est non nul, sinon on renvoie un vecteur de norme quelconque comme `(1,0,0)`.

Je vous demande que l'en-tête soit `vec3.h` et le code dans `vec3.c`. Chaque déclaration dans le `.h` devra avoir un en-tête de documentation sous la forme :

```
/*
 * nomdefonction : description de ce qu'elle fait
 *                  et de ce qu'elle renvoie.
 *
 * arg1 : role de l'argument 1
 * arg2: role de l'argument 2
 * ...
 */
```

Je vous demande aussi de tester vous-même chaque fonction et de placer le code dans `vec3_test.c` il doit contenir une fonction principale `test` qui lance tous les tests que vous aurez écrits, chacun de préférence dans des fonctions séparées :

```
void test_fonction1()
{
    ...
}
```

```

void test_fonction2()
{
    // ...
}

void test()
{
    test_fonction1();
    test_fonction2();
    // ...
}

```

Dans vos tests utilisez `assert` comme dans `assert(mafonction() == resultat_attendu);` qui est défini dans `assert.h` et vous permet de vérifier que les fonctions font ce qu'on attend d'elles.

Vous pouvez faire un fichier `vec3_test.h` qui expose la fonction de test principale et l'appeler dans `main`.

Le contenu de `vec3_test.h` sera alors juste

```
void test();
```

et celui de `tp06.c` sera

```

#include "vec3_test.h"

int main()
{
    test();
    return 0;
}

```

Écrivez dans la cellule suivante le code `vec3.h` :

```
[ ]: # Ecrivez votre solution ici à la place de la ligne suivante.
raise NotImplementedError
```

Écrivez dans la cellule suivante le code `vec3.c` :

```
[ ]: # Ecrivez votre solution ici à la place de la ligne suivante.
raise NotImplementedError
```

Écrivez dans la cellule suivante le code `vec3_test.c` :

```
[ ]: # Ecrivez votre solution ici à la place de la ligne suivante.
raise NotImplementedError
```

1.2 Structure plus riche

La structure suivante permet de définir une structure `eleve` permettant de contenir un nom qui est une chaîne de longueur au plus 9 et un âge :

```

struct eleve {
    char nom[10]; // prend donc 10 octets en mémoire
    int age; // prend 8 octets sur 64 bits
}; // donc prend 18 octets
typedef struct eleve eleve;

```

On aura besoin de comparer deux chaînes, on utilise pour cela :

```
int strncmp(const char *s1, const char *s2, size_t n);
```

qui permet de comparer au plus `n` caractères pour décider si les chaînes sont égales et renvoie 0 à ce moment-là :

- `strncmp("toto", "toti", 4)`; ne renvoie pas 0
- `strncmp("toto", "toto", 5)`; renvoie 0 (pas de problèmes de dépassement)
- `strncmp("toto", "toti", 3)`; renvoie 0

Le qualificatif `const` présent ici signifie que `strncmp` ne modifiera pas les chaînes (ce qui est assez rassurant pour une fonction de comparaison). A l'avenir, on essaiera de mettre les bons qualificatifs.

On va considérer une classe qui est un tableau d'`eleve` et on pourra directement accéder aux champs des cases avec `t[i].nom` ou `t[i].age`.

On vous demande d'écrire la fonction suivante dans la prochaine cellule :

```

/*
 * age : renvoie l'age d'un eleve dans une classe en fonction de son nom
 *       s'il n'est pas présent dans la classe, renvoie -1
 *
 * classe est un tableau d'eleve
 * nb_eleves est le nombre d'éléments du tableau
 * nom est le nom de l'élève dont on cherche l'age
 */
int age(const eleve classe[], int nb_eleves, const char *nom);

```

```

[ ]: # Ecrivez votre solution ici à la place de la ligne suivante.
     raise NotImplementedError

```

Du code pour tester votre fonction

```

eleve classe[4];
strcpy(classe[0].nom, "Riri");
classe[0].age = 9;
strcpy(classe[1].nom, "Fifi");
classe[1].age = 10;
strcpy(classe[2].nom, "Loulou");
classe[2].age = 11; // le premier qui dit que ce sont des triplés a un devoir supplémentaire
strcpy(classe[3].nom, "Donald");
classe[3].age = 42; // il est en 43/2

if (age(classe, 4, "Donald") != 42) printf("Erreur 1\n");

```

```
if (age(classe, 3, "Donald") != -1) printf("Erreur 2\n");
if (age(classe, 4, "Loulou") != 11) printf("Erreur 3\n");
```

2 Pointeurs

2.1 Guide de survie en milieu hostile

ATTENTION A partir de maintenant, on va quand même rendre bien plus complexe ce qu'on fait en C. Je vous engage à tester vos codes sur <https://pythontutor.com/c.html#mode=edit> pour comprendre ce qui se passe au début. On peut voir des flèches pour les notions de pointeurs qu'on va voir ici et c'est très visuel.

Malgré tout, vous allez avoir des programmes qui plantent, c'est normal. Le `Makefile` donné plus haut compile en mode `debug`, c'est-à-dire, entre autres, qu'il sait à quelle ligne du programme correspondent les instructions. On va s'en servir de deux manières :

- Avec un débogueur :
 - Dans le terminal, on tapes `gdb ./tp06` à la place de `./tp06`
 - dans `gdb` on tape `r` pour lancer (`run`) le programme qui plante et quand il va planter, on voit la ligne correspondante. On peut alors taper `bt` pour avoir la pile complète d'appels. On tape `q` pour quitter.

Exemple :

```
$ gdb ./test
(...)
Reading symbols from ./test...
(gdb) r
Starting program: /tmp/test

Program received signal SIGSEGV, Segmentation fault.
0x000055555555129 in bosserdur () at test.c:6
6      *p = 42;
(gdb) bt
#0  0x000055555555129 in bosserdur () at test.c:6
#1  0x000055555555140 in ilvafalloir () at test.c:11
#2  0x000055555555151 in tenant () at test.c:16
#3  0x000055555555162 in main () at test.c:21
(gdb) q
A debugging session is active.
```

```
Inferior 1 [process 1325735] will be killed.
```

```
Quit anyway? (y or n) y
```

- Avec `valgrind` qui va détecter lui-même les crashes mais aussi les problèmes liés à la mémoire même quand ils ne font pas crasher le programme.
 - il n'est pas installé par défaut dans les vm donc `sudo pacman -S valgrind` pour l'installer
 - ensuite `valgrind ./tp06`

Sur l'exemple précédent, il nous dit :

```
$ valgrind ./test
==1326195== Memcheck, a memory error detector
==1326195== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1326195== Using Valgrind-3.17.0 and LibVEX; rerun with -h for copyright info
==1326195== Command: ./test
==1326195==
==1326195== Invalid write of size 4
==1326195==    at 0x109129: bosserdur (test.c:6)
==1326195==    by 0x10913F: ilvafalloir (test.c:11)
==1326195==    by 0x109150: tenant (test.c:16)
==1326195==    by 0x109161: main (test.c:21)
==1326195== Address 0x0 is not stack'd, malloc'd or (recently) free'd
==1326195==
==1326195== Process terminating with default action of signal 11 (SIGSEGV): dumping core
==1326195== Access not within mapped region at address 0x0
==1326195==    at 0x109129: bosserdur (test.c:6)
==1326195==    by 0x10913F: ilvafalloir (test.c:11)
==1326195==    by 0x109150: tenant (test.c:16)
==1326195==    by 0x109161: main (test.c:21)
```

2.2 Explications succinctes

En C, contrairement à OCaml ou Python, il est possible d'avoir un accès direct à la mémoire et à l'emplacement des variables. Ça permet plusieurs choses :

- pouvoir modifier la valeur d'une variable en dehors de sa portée en transmettant son adresse
- permettre de partager des valeurs en copiant leur adresse et pas leurs valeurs
- pouvoir parcourir la mémoire quand on connaît son organisation

On parle de pointeur vers une valeur de type `t` pour une adresse munie de l'information supplémentaire du type de valeur qui s'y trouvent. Quand on veut parler de n'importe quel type de valeur, on utilise souvent un pointeur vers une valeur de type `void`. On note `t *` le type des pointeurs vers une valeur de type `t`.

Ainsi `int *p` déclare `p` comme étant un pointeur vers une valeur entière.

En fait, en C, on ne fait pas de différence entre tableau de type `t` et pointeur sur `t` (en fait un pointeur est vu comme un tableau à une case). On peut donc dans les fonctions utiliser `int *t` à la place de `int t[]`. **Par contre** pour définir un tableau initialisé, on utilisera toujours `int t[] = { 1, 2, 3 };`

Pour que `p` pointe vers l'emplacement de la variable `x`, on écrit `p = &x;`. Plus généralement, `&v` est un pointeur vers l'emplacement de `v` du type de pointeur associé aux valeurs de `v`.

Si `p` est un pointeur, on peut utiliser symétriquement `*p` pour obtenir la valeur pointée, cela peut permettre aussi de **l'obtenir en écriture** :

```
int x = 2;
int *p = &x;
```

```
*p = 3;
x
```

Une première utilisation de ce mécanisme est de permettre de renvoyer plusieurs valeurs. Par exemple, la fonction suivante renvoie l'indice et la valeur du minimum dans un tableau. Ici comme on passe le tableau avec `int *t` mais qu'on ne va faire que lire dedans alors qu'on passe `int *indice` dans lequel on va écrire, on a fait le choix de préciser que `t` était en lecture seule. On utilise pour cela le qualificatif `const` qui se rajoute devant le type. Si jamais on écrit dans `t` dans le corps, cela va produire une erreur à la compilation.

```
int cherche_minimum(const int *t, int nb, int *indice)
{
    assert(nb > 0); // fait un erreur si nb = 0
    int m = 0;
    for (int i = 0; i < nb; i++)
    {
        if (t[i] < t[m])
            m = i;
    }

    *indice = m; // on renvoie l'indice à côté
    return t[m]; // on renvoie la valeur
}

// exemple
int t[] = { 1, 3, 0, 2, 4 };
int indice;
int min = cherche_minimum(t, 5, &indice); // on passe un pointeur sur indice
printf("Le minimum est %d et il est à l'indice %d\n", min, indice);
```

Dans la fonction suivante, on vous demande de calculer pour un tableau `t` contenant au moins deux éléments le couple (i, j) où $i < j$ et pour lequel $|t[i] - t[j]|$ est minimal. C'est-à-dire la plus petite distance entre deux éléments d'indice distincts dans le tableau. On pourra utiliser `int abs(int x)`; pour calculer $|x|$.

Voici le prototype de votre fonction :

```
/*
 * t : tableau d'entiers
 * nb : nombre d'éléments du tableau
 *
 * i, j : valeurs de retours passées par pointeurs
 *
 * Renvoie dans (i,j) le couple tel |t[i]-t[j]| minimal
 * et i < j. En cas de plusieurs couples valides, renvoie dont
 * la première coordonnées est minimale.
 */
void plus_petite_distance(const int *t, int nb, int *i, int *j);
```

écrivez la définition complète ici après l'avoir testée :

```
[ ]: # Ecrivez votre solution ici à la place de la ligne suivante.  
raise NotImplementedError
```

```
// Exemple de test  
int t[] = { 5, 2, 7, 10 };  
int i, j;  
plus_petite_distance(t, 4, &i, &j);  
assert( i == 0 && j == 2 );
```

Attention, en C, on peut rapidement faire n'importe quoi et renvoyer des adresses de variables alors qu'elles n'existent plus en mémoire. Par exemple :

```
int *bug()  
{  
    int x = 3;  
    return &x;  
}
```

qui va faire un warning, mais aussi le code suivant qui ne va pas en faire :

```
int *x;  
{  
    int y = 3;  
    x = &y;  
}  
// à partir d'ici x pointe vers une variable qui n'existe plus  
// car on est sorti du bloc
```

3 Lecture et écriture de fichiers

Toutes les fonctions définies ici sont définies dans `stdio.h`

Pour accéder à des fichiers, on passe par des pointeurs vers un type `FILE` dont l'implémentation interne dépend du système (en gros on ne va jamais voir ce qu'il y a dedans).

On récupère un tel pointeur avec `fopen` :

`fp` contient un pointeur vers le fichier "test.txt" ouvert en lecture "r" pour read. `FILE *fp = fopen("test.txt", "r");` On dit que `fp` est un flux de fichier, ou **stream** en anglais.

Plus généralement

```
FILE *fopen(const char *pathname, const char *mode);
```

où `mode` est :

- "r" pour ouvrir en lecture
- "w" pour ouvrir en écriture
- "rb" ou "wb" pour lire/écrire en binaire dans des OS primitifs qui font la distinction texte/binaire (en gros, Windows)
- plus plein d'autres, faites **man fopen** sur la console pour savoir.

Plus généralement, on ne décrit ici qu'un usage simple des fonctions mais la doc accessible dans le terminal linux avec `man` est bien plus complète.

Lorsqu'on n'a plus besoin d'un flux de fichier, on le ferme avec `fclose(fp)`;

ATTENTION c'est à la fermeture qu'on effectue une partie des écritures, avant cela, votre fichier n'est pas valide. Il ne se ferme pas automatiquement à la fermeture du programme contrairement à Python.

3.1 Lecture caractère par caractère

On va ici utiliser un fichier de tests. Vous pouvez le télécharger ainsi dans un terminal dans le répertoire `tp06` :

- `curl -O http://marc.de-falco.fr/assets/test_occ.txt`

Pour lire un caractère depuis le flux de fichier on utilise `char c = fgetc(fp)`; Quand on a atteint la fin du fichier, on reçoit le caractère spécial EOF. On peut aussi utiliser la fonction `feof(fp)` qui renvoie un booléen permettant de déterminer si on est à la fin du fichier. On peut donc faire :

```
while (!feof(fp))
{
    char c = fgetc(fp);
    // des trucs utilisant c
}
```

On vous demande d'écrire une fonction qui lit un fichier encodé en ascii (c'est-à-dire que seuls les caractères 0 à 127 s'y trouvent) et affiche les 20 caractères les plus fréquents ainsi que leur fréquence.

Le prototype est :

```
/*
 * plus_frequent : affiche les 20 caractères les plus fréquents dans le fichier f
 *                  où f est encodé en ascii donc les caractères sont entre 0 et 127
 *
 * f : nom de fichier
 */
void plus_frequent(const char *f);
```

On rappelle qu'on peut utiliser `printf` (depuis `stdio.h`) pour faire un affichage sophistiqué. Notamment, si ici le caractère est `c`, que sa fréquence est `freq` et que c'est le `i`ème plus fréquent on pourra écrire :

```
printf("Le %dème caractère le plus fréquent est %c de fréquence %f\n", i, c, freq);
```

Comme 20 est grand, il faudra utiliser un tri sur un tableau plutôt que de maintenir 20 variables. Vous risquez de perdre l'information sur le caractère si on utilise un seul tableau `occ` comme dans le TP4. Il faudrait un autre tableau `carac` tel que `carac[i]` donne le caractère dont le nombre d'occurrences est dans `occ[i]`. Quand vous faites un échange pour trier, échangez dans les deux tableaux en même temps pour maintenir le lien (oui c'est un invariant ;-)).

```
[ ]: # Ecrivez votre solution ici à la place de la ligne suivante.
      raise NotImplementedError
```

Votre code devra produire le résultat suivant sur le fichier `test_occ.txt`

```
Le 1eme caractère est   de fréquence 0.234752
Le 2eme caractère est e de fréquence 0.074419
Le 3eme caractère est t de fréquence 0.053447
Le 4eme caractère est o de fréquence 0.051462
Le 5eme caractère est a de fréquence 0.045168
Le 6eme caractère est h de fréquence 0.040318
Le 7eme caractère est n de fréquence 0.040070
Le 8eme caractère est s de fréquence 0.039660
Le 9eme caractère est r de fréquence 0.038917
Le 10eme caractère est i de fréquence 0.036464
Le 11eme caractère est l de fréquence 0.026885
Le 12eme caractère est d de fréquence 0.024812
Le 13eme caractère est
  de fréquence 0.022916
Le 14eme caractère est u de fréquence 0.021260
Le 15eme caractère est m de fréquence 0.017412
Le 16eme caractère est y de fréquence 0.015346
Le 17eme caractère est , de fréquence 0.015114
Le 18eme caractère est . de fréquence 0.013964
Le 19eme caractère est w de fréquence 0.013407
Le 20eme caractère est f de fréquence 0.012933
```

3.2 Écriture caractère par caractère

On va ici utiliser un fichier dans cette partie. Vous pouvez le télécharger ainsi dans un terminal dans le répertoire `tp06` :

```
curl -O http://marc.de-falco.fr/assets/test_chiffre.txt
```