Programmation C
TP n° 2 : `enum`, `struct`

Exercice 1 :

On définit les deux fonctions `f` et `g` sur les entiers de la manière suivante :

$$\mathtt{f}(n) = \begin{cases} 2 & \text{si } n = 1 \\ 2 \cdot \mathtt{g}(n-1) & \text{sinon} \end{cases} \qquad \mathtt{g}(n) = \begin{cases} 1 & \text{si } n = 1 \\ 3 \cdot \mathtt{f}(\frac{n}{2}) & \text{sinon} \end{cases}$$

Faire un programme C pour ces deux fonctions. Donner la valeur de $\mathtt{f}(20)$.

Exercice 2 :

Dans cet exercice, on utilise les définitions suivantes :

```c
enum etat {VALIDEE, ENCOURS, EXPEDIEE};
typedef enum etat etat;
struct commande {
  int num_com;
  int prix_exp;
  int prix_prod;
  etat etat_com;};
typedef struct commande commande;
```

Une commande est donc une structure comprenant des champs pour : le numéro de la commande, son prix d'expédition, le prix du produit, et l'état de la commande (validée, en cours ou expédiée).

1. Faire une fonction d'en-tête `commande com_alea(int num)` qui renvoie une commande aléatoire : son prix d'expédition sera choisi aléatoirement entre 1 et 20, sa valeur entre 1 et 2000, son état sera choisi aléatoirement, et son numéro sera l'entier passé en argument. Remarque : pour obtenir un nombre (pseudo) aléatoire en `C`, vous pouvez utiliser la fonction `int rand()` qui retourne un entier aléatoire compris entre 0 et `RAND_MAX` (qui vaut au moins 32767). Il faut commencer par initialiser la suite de nombres aléatoires en appelant la fonction `void srand(unsigned seed)` avec par exemple comme graine de départ l'heure courante : `srand(time(NULL))`. Ensuite, à chaque appel de la fonction `rand`, celle-ci retourne un nouveau nombre aléatoire.
2. Dans le `main`, on déclarera un tableau de `NBC` commandes (où `NBC` est une constante définie par `#define`), que l'on initialisera avec la fonction `com_alea`.
3. Faire une fonction `void affiche_com(commande c)` qui affiche une description de la commande `c`. On utilisera un `switch` pour afficher proprement l'état.
4. Faire une fonction `void affiche_exp(commande t[], int taille)` qui affiche les commandes *expédiées* du tableau d'adresse `t`, supposé de taille `taille`.
5. Faire une fonction `int nbr_en_cours(commande t[], int taille)` qui renvoie le nombre de commandes *en préparation* dans le tableau d'adresse `t`.
6. Faire une fonction `int cout_validees(commande t[],int taille)` qui renvoie le coût total d'expédition des commandes *validées* dans le tableau d'adresse `t`.

Exercice 3 : Fractions avec `struct`

Dans cet exercice, on se propose de définir un type avec `struct` afin de représenter les rationnels sous forme de fractions. Pensez à tester votre code à chaque question !

1. À l'aide de `struct`, définissez un type de structure `fraction` avec deux champs entiers `num` et `den` de type `long int` qui représentent respectivement le numérateur et le dénominateur de la fraction. Utilisez le mot clé `typedef` afin de créer l'alias `fraction` pour le type `struct fraction` et rendre ainsi votre code plus lisible.
2. Écrivez une fonction d'en-tête `fraction build(long int n, long int d)` qui prend en arguments deux entiers `n` et `d` et qui retourne la fraction $\frac{n}{d}$.

La fonction définie à la question précédente à un défaut évident : elle permet de construire des fractions avec 0 comme dénominateur ! Afin de pallier ce problème, on se propose d'utiliser la fonction `assert` de la bibliothèque standard. Une commande de la forme `assert(b)` évalue la condition logique `b`. Si la condition est fausse, le programme interrompt son exécution en affichant un message d'erreur. Pour vous servir de cette fonction, il faut ajouter au début de votre programme la ligne suivante :

```
#include <assert.h>
```

3. Modifiez le code de la fonction `build` afin de provoquer une erreur si l'on essaye de construire une fraction dont le dénominateur est `0`. Ensuite, dans le `main`, créez un tableau `ex_fractions` de fractions qui contient les fractions $\frac{1}{1}$, $\frac{1}{2}$, $\frac{2}{4}$, $\frac{-9}{3}$, $\frac{8}{-20}$, $\frac{-5}{-1}$, $\frac{1}{-3}$.
4. Écrivez une fonction d'en-tête `int eq_fraction(fraction f, fraction g)` qui renvoie `1` si les deux fractions sont égales et 0 sinon. On rappelle que deux fractions $\frac{a}{b}, \frac{c}{d}$ sont égales si et seulement si $a * d = c * b$.
5. Écrivez une fonction `int is_int(fraction f)` qui renvoie `1` si la fraction `f` est un entier (c'est-à-dire peut être mis sous la forme $\frac{n}{1}$ où n est un entier) et 0 sinon.
6. Écrivez les fonctions suivantes qui calculent la somme, la soustraction et la multiplication de fractions.

```
fraction sum(fraction f, fraction g) // somme
fraction sub(fraction f, fraction g) // soustraction
fraction mul(fraction f, fraction g) // multiplication
```

7. Écrivez une fonction d'en-tête `fraction reduce(fraction f)` qui renvoie la fraction `f` sous forme irréductible. Pour cela, on pourra d'abord coder la fonction `long pgcd(long a, long b)` qui calcule le pgcd des deux entiers `a` et `b`. On rappelle que l'algorithme d'Euclide pour calculer le pgcd de deux entiers *positifs* a et b est (en pseudo-code) :

```
x <- a
y <- b
while (y !=0){
  r <- reste de la division euclidienne de x par y
  x <- y
  y <- r
}
return x
```

Vous devez également faire en sorte que lorsque la fraction renvoyée par `reduce` est négative, le signe apparaisse au numérateur et non pas au dénominateur. Testez vos fonctions sur les fractions de `ex_fractions`.

On souhaite maintenant définir un type qui représente un point (rationnel) dans le plan, c'est-à-dire un point dont les coordonnées peuvent être représentées par des fractions.

8. Définissez un type de structure `point` avec deux champs `x` et `y` de type `fraction`, qui représentent les coordonnées du point, puis utilisez `typedef` afin de créer l'alias `point` pour le type `struct point`.

9. Écrivez une fonction `int eq_point(point p1, point p2)` qui renvoie `1` si les deux points ont les mêmes coordonnées et `0` sinon.

10. Écrivez une fonction d'en-tête `double dist(point p1, point p2)` qui prend en argument deux points et calcule leur distance euclidienne en tant que valeur de type `double`.

On rappelle que pour utiliser la fonction `sqrt` de la bibliothèque standard, il faut rajouter `#include <math.h>` en haut de votre programme et il faut alors compiler avec l'option `-lm`.

On rappelle également que pour convertir une variable `n` de type entier en double, il faut utiliser l'expression `(double) n`.

Exercice 4 : Polynômes

On peut modéliser un polynôme à coefficients entiers par un tableau `t` d'entiers où l'entier `t[i]` représente le coefficient de degré `i`. Ainsi, le polynôme $3X^3 + 4X + 1$ est représenté par le tableau `{1, 4, 0, 3}`. Pour simplifier l'exercice, on fixera un entier N et on n'utilisera que des tableaux de taille N pour modéliser nos polynômes.

1. Écrivez une fonction qui affiche un tableau d'entiers représentant un polynôme. Par exemple, le tableau `{1,0,3,0... }` sera affiché de la manière suivante : `1+3X^2`.
2. Écrivez une fonction qui prend deux polynômes et retourne un nouveau polynôme résultant de leur addition. L'addition de deux polynômes s'effectue en sommant les coefficients de même degré. Par exemple, la somme de $1 + X + 2X^2$ et de $2 + 3X$ donne $3 + 4X + 2X^2$.
3. Écrivez une fonction qui prend deux polynômes P et Q, et retourne un nouveau polynôme résultant de leur multiplication. Le coefficient de degré i du polynôme résultant est la somme des produits des coefficients P_j et Q_k pour $j + k = i$. Par exemple, la multiplication de $1 + X + 2X^2$ et de $2 + 3X$ donne $2 + 5X + 7X^2 + 6X^3$.
4. Écrivez une fonction qui prend un polynôme ainsi qu'un entier et retourne le résultat de l'évaluation de ce polynôme. Par exemple, `eval({1, 0, 3}, 2)` qui représente l'évaluation de $1 + 3X^2$ en 2 retournera la valeur 13.