TP 3 : struct et union

1 Manipulation de rationnels, ou comment utiliser struct

Il existe principalement deux types numériques en C : les entiers (`char`, `short`, `int`, `long`, `long long`, etc.) et les flottants (`float`, `double`). Nous allons définir un nouveau type pour représenter des rationnels sous forme de fraction.

Exercice 1

1. Dans un fichier `types_numeriques.h`, définir une structure `fraction` avec deux champs entiers `denominateur` et `numerateur`.
2. Dans un fichier `fractions.c`, qui importe la définition de la structure `fraction` à l'aide de la directive `#include "types_numeriques.h"`, implémenter une fonction :
   `void affiche_fraction(struct fraction f)` qui a□che une fraction sur la sortie standard, suivie d'une valeur approchée de cette fraction.
3. Définir un tableau `ex_fractions` contenant les fractions : $\frac{1}{1}, \frac{1}{2}, \frac{2}{4}, \frac{15}{10}, \frac{-9}{3}, \frac{8}{-20}, \frac{-5}{-10}, \frac{1}{-3}$. Testez votre fonction `affiche` sur chacune de ces fractions.
4. Implémenter une fonction `struct fraction reduit(struct fraction f)` qui renvoie la fraction irréductible [1] correspondant à la fraction passée en argument. Pour cela vous écrirez une fonction `long long pgcd(long long a, long long b)` qui renvoie le plus grand diviseur commun à deux entiers.
   Testez votre fonction sur les fractions du tableau `ex_fractions`.
5. Implémenter les fonctions :
   - `struct fraction add_frac(struct fraction f1, struct fraction f2)`
     qui renvoie la somme des deux fractions passées en argument
   - `struct fraction sub_frac(struct fraction f1, struct fraction f2)`
     qui renvoie la di□érence des deux fractions passées en argument
   - `struct fraction mult_frac(struct fraction f1, struct fraction f2)`
     qui renvoie le produit des deux fractions passées en argument
   - `struct fraction div_frac(struct fraction f1, struct fraction f2)`
     qui renvoie le quotient des deux fractions passées en argument
   - `int frac_eq(struct fraction f1, struct fraction f2)`
     qui renvoie si deux fractions sont égales.
   - `float to_float(struct fraction f)`
     qui renvoie un entier approximant la fraction passée en argument
   - `struct fraction of_int(int i)`
     qui renvoie la fraction irréductible correspondant à l'entier passé en argument.

   Exportez ensuite leurs définitions dans le fichier `fractions.h`. Ce fichier devra contenir la directive `#include "types_numeriques.h"`.
6. Ajouter la directive `#include "fractions.h"` dans le fichier `fractions.c`. A-t-on toujours besoin de la directive `#include "types_numeriques.h"` dans `fractions.c` ?

1. Si le dénominateur est 0, la forme irréductible peut être $\frac{1}{0}$, $\frac{0}{0}$ ou $\frac{-1}{0}$.

7. La *Méthode de Héron* permet de calculer des approximations rationnelles des racines carrées d'entiers. Elle se base sur le fait que la suite U_n définie par :

$$\begin{aligned} U_0 &= 1 \\ U_{n+1} &= \frac{U_n + \frac{A}{U_n}}{2} \end{aligned}$$

tend vers $\sqrt{A}$ quand n tend vers $+\infty$.
Dans un fichier `heron.c`, écrire une fonction `struct fraction heron(int a, int n)` qui renvoie le n-ième terme de la suite de Héron pour un entier a. Vous utiliserez le type `fraction` pour faire tous les calculs. Quel(s) fichier(s) "`.h`" faut-il importer ?

8. Écrire une fonction main [2] qui demande à l'utilisateur deux entiers a et n, et qui affiche les n premiers termes de la suite de Héron paramétrée par a. Que constatez-vous ?
`heron.c`

9. (**bonus**) Écrire une fonction `struct fraction heron_mieux(int a, int n)` qui renvoie le n-ième terme de la suite de Héron paramétrée par a et initialisée avec $\lfloor\sqrt{a}\rfloor$.

2 Types numeriques abstraits (avec struct et union)

En C, lorsqu'on effectue une division sur des entiers, le résultat renvoyé est un entier. Ici, on cherche à faire en sorte que la division de deux entiers nous renvoie une `fraction`.

Exercice 2

1. Dans le fichier `types_numeriques.h` définir un type union nommé `valeur` qui peut être soit un `long long`, un `float` ou un `struct fraction`.
2. Afin de nous "souvenir" quel est le type d'un objet de type `union valeur`, nous allons l'encapsuler dans une structure `num`, qui a deux champs : un champ `val` qui contiendra la `valeur` et un champ `t` qui contiendra le type de la valeur. Quels sont les types de ces deux champs ? Écrire la définition du type `num` [3].
3. Dans un fichier `nums.c` [4] écrire une fonction `affiche_num` qui affiche un `num`.
4. Écrire une fonction `struct num retype(struct num n)` qui retype l'argument :
   - Si sa valeur est un `struct fraction`, et que celle-ci a un dénominateur égal à 1, alors il est reconverti en un `num` dont la valeur est un `long long`. Sinon, la fraction est réduite. Enfin, si le dénominateur ou la valeur absolue du numérateur est supérieure à 2.10^9, alors il est converti en un `num` dont la valeur est un `float`. Cela en vue d'éviter un débordement d'entier [5].
   - Si sa valeur est un `float` tel que `(float) ((int) f) == f` alors, il est reconverti en un `num` dont la valeur est un entier.
   - Si sa valeur est un `long long`, alors on ne fait rien.
5. Dans `nums.c` écrire les fonctions d'addition, de soustraction, de multiplication et de division sur les `num`. Exportez ces définition dans `nums.h`.
6. Modifiez votre fichier `heron.c` [6] pour qu'il utilise les fonctions sur les nums au lieu des fonctions sur les fractions.

---

2. dans le fichier `heron.c`, qui devra être compilé avec `gcc heron.c fractions.o -o heron`
sachant que `fractions.o` aura été obtenu avec la commande `gcc -c fractions.c`

3. `num` doit vous évoquer "type numérique abstrait", rien à voir avec `enum` qui doit vous évoquer le terme anglais "enumeration"

4. Qui sera compilé avec `gcc nums.c fractions.o -o nums`

5. On rappelle qu'un `long long` est codé sur 64 bits, il prend donc des valeurs entre -2^{63} et $2^{63}-1$, si tous les entiers sont inférieurs (en valeur absolue) à $2.10^9 \approx 2^{31}$, alors on est assuré que le produit de deux entiers sera inférieur (en valeur absolue) à 2^{62} et que la somme de deux tels produits sera inférieur (en valeur absolue) à 2^{63}, qu'ainsi nous n'aurons pas de débordements d'entiers.

6. Il devra maintenant être compilé avec `gcc heron.c nums.o fractions.o -o heron`