PROGRAMMATION C
TP3

LICENCE MATHS-INFO
30 JANVIER 2012

L'objectif de ce TP est de créer et manipuler de *nouveaux types*, et de continuer à programmer de façon *modulaire*, via une architecture logicielle à base de fonctions et de fichiers séparés. On utilisera la notion de *bibliothèque*[1].

Le scénario est le suivant : nous voulons mettre en œuvre la possibilité de faire des calculs arithmétiques sur différents types de nombres – entiers, flottants, rationnels, complexes –, le tout à l'aide de fonctions « mixtes ». Par exemple, une unique fonction « mixte »d'addition pourra ajouter des nombres de tous les types et renvoyer un nombre du type approprié.

Pour cela, le sujet est découpé en plusieurs parties[2] :

- dans la partie 1, vous serez guidés pour concevoir l'architecture générale du code ;
- dans la partie 2, vous créerez le code permettant de manipuler des rationnels, étant donné qu'il n'existe pas de type rationnel en C ;
- dans la partie 3, vous créerez le code permettant de manipuler des nombres complexes, pour la même raison ;
- dans la partie 4, vous créerez le code permettant de manipuler un type générique de nombre, regroupant les entiers, flottants, rationnels et complexes.

Partie 1 : conception de l'architecture générale

Avant de vous lancer dans l'écriture du code C, vous devez comprendre le sujet avec suffisamment de recul pour pouvoir concevoir l'architecture générale du code. En particulier, les deux questions ci-dessous nécessitent : 1/ de comprendre quels sont les fichiers à créer ; et 2/ d'avoir une idée de leur contenu futur.

Pour commencer, lisez une fois en entier le texte des parties 2, 3 et 4, puis passez à la question suivante.

Diagramme des dépendances. A partir du premier paragraphe de chacune des parties 2, 3 et 4, déterminez quels sont les fichiers à écrire et dessinez un diagramme des dépendances (sur du papier, avec un vrai stylo).

Ecriture du makefile. A partir du diagramme, écrivez le `makefile` en entier.

---

1. En programmation, une *bibliothèque* (*library* en anglais) est un code contenant un ensemble d'outils (fonctions, types, etc.) utilisables par d'autres programmes. Une bibliothèque est généralement thématique : par exemple, la bibliothèque stdio fournit des outils d'entrée/sortie. Une bibliothèque ne contient pas de programme exécutable en général.
2. Le sujet lui-même suit donc cette logique modulaire !

Partie 2 : écriture d'une bibliothèque pour les nombres rationnels

Les nombres rationnels, c'est-à-dire les fractions, n'ont pas un type dédié en C. Dans cette partie, vous écrirez une *bibliothèque* pour les nombres rationnels en définissant un nouveau type `Rational` et en créant des fonctions pour les manipuler. Le type sera défini dans un fichier en-tête `lib_rational.h`, qui contiendra également la *déclaration* des fonctions de la bibliothèque. Un fichier `lib_rational.c` contiendra la *définition* des fonctions. Un fichier `test_lib_rational.c` contiendra un programme de test de la bibliothèque. Un utilisateur pourra donc : soit utiliser l'exécutable obtenu en compilant le programme de test pour avoir une idée du fonctionnement de la bibliothèque ; soit utiliser la bibliothèque seule (sans le programme de test) dans un autre programme.

Création du type Rational. Les rationnels peuvent être considérés comme un couple d'entiers : la fraction $3/4$ est représenté par le couple d'entiers $(3, 4)$. Nous allons mettre en œuvée l'arithmétique des fractions rationnelles et pour ce faire, il nous faut définir un type `Rational`. Nous pourrions utiliser un tableau de deux entiers mais nous allons plutôt utiliser une structure avec deux champs : `num` (numérateur) et `den` (dénominateur).

Définissez ce type `Rational`. Testez la création de variables de ce type.

Affichage d'un rationnel. Déclarez, définissez et testez une fonction `printRational` qui prend en argument une variable de type `Rational` et l'affiche.

Addition de rationnels. Déclarez, définissez et testez une fonction `addRational` qui prend en argument deux variables de type `Rational` et retourne une variable de type `Rational` codant la somme des deux arguments.

Produit de rationnels. Déclarez, définissez et testez une fonction `mulRational` qui prend en argument deux variables de type `Rational` et retourne une variable de type `Rational` codant le produit des deux arguments.

Conversion d'un rationnel en flottant. Déclarez, définissez et testez une fonction `Rational2double` qui prend en argument un `Rational` et retourne un `double` correspondant à sa conversion.

Le codage des fonctions suivantes est optionnel et sera réalisé lorsque les parties obligatoires du TP seront finies.

Soustraction de rationnels. Déclarez, définissez et testez une fonction `subRational` qui prend en argument deux `Rational` et retourne un `Rational` codant leur soustraction.

Quotient de rationnels. Déclarez, définissez et testez une fonction `divRational` qui prend en argument deux `Rational` et retourne un `Rational` codant leur quotient.

Réduction sous forme irréductible. Déclarez, définissez et testez une fonction `normalRational` qui prend en argument un `Rational` et retourne un Rational codant sa forme irréductible (la forme irréductible de la fraction $2/4$ est $1/2$).

Egalité de rationnels. Déclarez, définissez et testez une fonction `isequalRational` qui prend en argument deux `Rational` et retourne l'entier 1 si les deux fractions sont égales d'un point de vue mathématique et 0 sinon.

Partie 3 : écriture d'une bibliothèque pour les nombres complexes

Les nombres complexes n'ont pas non plus de type dédié en C. Dans cette partie, vous écrirez une bibliothèque pour les nombres complexes en définissant, comme précédemment, un nouveau type `Complex` et en créant des fonctions pour les manipuler. Le type sera défini dans un fichier en-tête `lib_complex.h`, qui contiendra également la *déclaration* des fonctions de la bibliothèque. Un fichier `lib_complex.c` contiendra la *définition* des fonctions. Un fichier `test_lib_complex.c` contiendra un programme de test de la bibliothèque.

Création du type Complex. Les nombres complexes peuvent être considérés comme un couple de réels : le nombre complexe $3 + 4i$ est représenté par le couple $(3, 4)$. Nous allons mettre en œuvée l'arithmétique des nombres complexes et pour ce faire, il nous faut définir un type `Complex`. Vous utiliserez une structure avec deux champs : `Re` (partie réelle) et `Im` (partie imaginaire).

Définissez ce type `Complex`. Testez la création de variables de ce type.

Affichage d'un nombre complexe. Déclarez, définissez et testez une fonction `printComplex` qui prend en argument un `Complex` et l'affiche.

Addition de complexes. Déclarez, définissez et testez une fonction `addComplex` qui prend en argument deux `Complex` et retourne un `Complex` codant leur somme.

Produit de complexes. Déclarez, définissez et testez une fonction `mulComplex` qui prend en argument deux `Complex` et retourne un `Complex` codant leur produit.

Conversion d'un flottant en complexe. Déclarez, définissez et testez une fonction `double2Complex` qui prend en argument un flottant de type `double` et retourne un `Complex` correspondant à sa conversion.

Le codage des fonctions suivantes est optionnel et sera réalisé lorsque les parties obligatoires du TP seront finies.

Soustraction de complexes. Déclarez, définissez et testez une fonction `subComplex` qui prend en argument deux `Complex` et retourne un `Complex` codant leur soustraction.

Quotient de complexes. Déclarez, définissez et testez une fonction `divComplex` qui prend en argument deux `Complex` et retourne un `Complex` codant leur quotient.

Module d'un complexe. Déclarez, définissez et testez une fonction `absComplex` qui prend en argument un `Complex` et retourne son module (*absolute value*, *magnitude* ou *modules* en anglais).

Argument d'un complexe. Déclarez, définissez et testez une fonction `argComplex` qui prend en argument un `Complex` et retourne son argument (au sens des nombres complexes, et non au sens des fonctions en informatique).

Egalité de complexes. Déclarez, définissez et testez une fonction `isequalComplex` qui prend en argument deux `Complex` et retourne l'entier 1 si les deux nombres sont égaux d'un point de vue mathématique et 0 sinon.

Partie 4 : écriture d'une bibliothèque d'arithmétique mixte

Nous souhaitons ici définir une bibliothèque permettant de manipuler un type générique de nombre regroupant tous les types de nombres. Pour ce faire, vous écrirez une bibliothèque en définissant un nouveau type `Number` et en créant des fonctions pour les manipuler. Ce type permettra de manipuler les entiers et les flottants définis par le langage, ainsi que les rationnels et les complexes via vos mises en oeuvre dans les parties précédentes. Le type sera défini dans un fichier en-tête `lib_number.h`, qui contiendra également la déclaration des fonctions de la bibliothèque. Un fichier `lib_number.c` contiendra la définition des fonctions. Un fichier `test_lib_number.c` contiendra un programme de test de la bibliothèque.

Création du type Number. En utilisant votre mise en oeuvre des rationnels et des complexes, définissez un type `Number`, une variable de ce type pouvant être soit un entier, soit un `Rational`, soit un flottant, soit un `Complex`.

Vous pouvez vous baser sur le type `Number` suivant :

```
enum TypeNumber {integer, floating, rational, complex} ;

typedef struct nb {
   enum TypeNumber tn ;
   union val {
     int intnb ;
     Rational rationalnb  ;
     double floatnb ;
     Complex complexnb ;
   } value;
} Number ;
```

Testez la création de variables de ce type.

Affichage d'un nombre quelconque. Déclarez, définissez et testez une fonction `printNumber` qui prend en argument une variable de type `Number` et l'affiche à l'écran.

Addition de nombres quelconques. Déclarez, définissez et testez une fonction `addNumber` qui prend en argument deux `Number` et retourne un `Number` codant leur somme.

Dans cette fonction et les suivantes, la mise en oeuvre devra déterminer le plus intelligemment possible la nature du résultat lorsque, par exemple, un rationnel et un flottant sont ajoutés, multipliés, etc. En particulier, on remarquera que $\mathbb{N} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$.

Produit de nombres quelconques. Déclarez, définissez et testez une fonction `mulNumber` qui prend en argument deux `Number` et retourne un`Number` codant leur produit.

Pour fixer les idées, on cherche à élaborer des fonctions permettant les manipulations décrites dans la fonction principale suivante :

```
int main(void){
  Number nb1, nb2, nb3 ;

  nb1.tn = rational ;
  nb1.value.rationalnb.num = 1 ;
  nb1.value.rationalnb.den = 4 ;

  nb2.tn = integer ;
  nb2.value.intnb= 4 ;

  nb3.tn = complex ;
  nb3.value.complexnb.Re = 2.54 ;
  nb3.value.complexnb.Im = 4.45 ;

  printNumber(nb1) ;
  printf(" et ");
  printNumber(nb2) ;
  printf(" et ");
  printNumber(nb3) ;
  printf(".\n");

  printNumber(divNumber(mulNumber(nb1,nb2),nb3)) ;
  printf(".\n");

  return 0 ;
}
```

Le codage des fonctions suivantes est optionnel et sera réalisé lorsque les parties obligatoires du TP seront finies.

Soustraction de nombres quelconques. Déclarez, définissez et testez une fonction `subNumber` qui prend en argument deux `Number` et retourne un `Number` codant leur soustraction.

Quotient de nombres quelconques. Déclarez, définissez et testez une fonction `divNumber` qui prend en argument deux `Number` et retourne un `Number` codant leur quotient.

Egalité de nombres quelconques. Déclarez, définissez et testez une fonction `isequalNumber` qui prend en argument deux `Number` et retourne l'entier 1 si les deux nombres sont égaux d'un point de vue mathématique et 0 sinon.