IN 101 - TD 08

4 novembre 2011

Corrigé

ENSTA ParisTech

Solution 1 : Bibliothèque pour gérer les rationnels

À la fin de l'exercice, le fichier `ratio.h` contient tous les prototypes de fonctions et la déclaration du type *ratio*. Il est aussi pratique de préciser tous les includes nécessaires au fonctionnement de la bibliothèque, `stdio.h` pour les `printf` et `stdlib.h` pour les `malloc`. La ligne du début `#ifndef _RATIO_H` et le `#define _RATIO_H` qui la suit servent à éviter de déclarer la structure et les fonctions deux fois si jamais le fichier `ratio.h` est inclus deux fois. Ce n'est pas forcément nécessaire ici, mais c'est une pratique à avoir : les utilisateurs de votre bibliothèque peuvent faire autant d'includes qu'ils veulent sans risques.

ratio.h

```
1  #ifndef _RATIO_H
2  #define _RATIO_H
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  typedef struct {
8    int numer;
9    int denom;
10 } ratio;
11
12 ratio* rat_init(int a, int b);
13 void rat_print(ratio* r);
14
15 ratio* rat_add(ratio* r1, ratio* r2);
16 ratio* rat_mul(ratio* r1, ratio* r2);
17 void rat_free(ratio* r);
18
19 void rat_simplify(ratio* r);
20
21 ratio* rat_clone(ratio* r);
22 double rat_eval(ratio* r);
23 void rat_inv(ratio* r);
24 ratio* rat_addint(ratio* r, int a);
25
26 #endif
```

1.a] La déclaration du type a sa place dans le `.h`, avec le prototype. Le corps de la fonction va lui dans le fichier `ratio.c`. Il faut utiliser `malloc` pour que la structure ne soit pas dans la pile et soit encore accessible après avoir quitté la fonction `rat_init`. Notez que le fichier `ratio.c` doit commencer par inclure le fichier `ratio.h` (avec des guillemets `""` et non pas les `<>` car le fichier est dans le répertoire, pas à un emplacement de bibliothèque système) pour pouvoir connaître le type *ratio*.

ratio.c

```
#include "ratio.h"

ratio* rat_init(int a, int b) {
  ratio* res = (ratio*) malloc(sizeof(ratio));
  res->numer = a;
  res->denom = b;
  return res;
}
```

1.b] Voici la fonction d'affichage et le `main` qui permet de tester.

ratio.c

```
void rat_print(ratio* r) {
  printf("%d/%d", r->numer, r->denom);
}
```

main.c

```
#include "ratio.h"

int main() {
  ratio* a = rat_init(3,5);
  rat_print(a);
  return 0;
}
```

1.c] Voici les nouvelles fonctions et de quoi les tester.

ratio.c

```
ratio* rat_add(ratio* r1, ratio* r2) {
  ratio* res = rat_init(r1->numer*r2->denom + r2->numer*r1->denom,
                                  r1->denom*r2->denom);
  return res;
}

ratio* rat_mul(ratio* r1, ratio* r2) {
  ratio* res = rat_init(r1->numer*r2->numer, r1->denom*r2->denom);
  return res;
}

void rat_free(ratio* r) {
  free(r);
}
```

main.c

```
#include "ratio.h"

int main() {
  ratio* a = rat_init(3,5);
  ratio* b = rat_init(2,7);
  ratio* c = rat_add(a,b);
  rat_print(c);
  rat_free(a);    // on libère a avant de l'écraser
  a = rat_mul(b,c);
```

```
10   rat_print(a);
11   rat_free(a); rat_free(b); rat_free(c);
12   return 0;
13 }
```

1.d] Pour simplifier on utilise la fonction interne `gcd` que l'on n'inclut pas dans le `.h`. Une fois `mat_simplify` programmé, on peut l'appeler sur `res` dans `mat_add` et `mat_mul` avant le `return`. Ici, le passage d'argument par pointeur simplifie beaucoup la vie : on modifie directement la fraction sans avoir à renvoyer quoi que ce soit.

ratio.c

```
1 int gcd(int a, int b) {
2   int r;
3   while (b != 0) {
4     r = a%b;
5     a = b;
6     b = r;
7   }
8   return a;
9 }
10
11 void rat_simplify(ratio* r) {
12   int pgcd = gcd(r->numer, r->denom);
13   r->numer /= pgcd;
14   r->denom /= pgcd;
15 }
```

1.e] Voici les dernières fonctions de la bibliothèque.

ratio.c

```
1 ratio* rat_clone(ratio* r) {
2   ratio* res = rat_init(r->numer, r->denom);
3   return res;
4 }
5
6 double rat_eval(ratio* r) {
7   double res = (double) r->numer/r->denom;
8   return res;
9 }
10
11 void rat_inv(ratio* r) {
12   int tmp = r->numer;
13   r->numer = r->denom;
14   r->denom = tmp;
15 }
16
17 ratio* rat_addint(ratio* r, int a) {
18   ratio* res = rat_init(r->numer + a*r->denom, r->denom);
19   return res;
20 }
```

Solution 2 : Utilisation de votre bibliothèque de rationnels

2.a] Voici à quoi peut ressembler la fonction pour la calcul de $\sqrt{2}$ et un `main` pour la tester. Notez bien l'utilisation de `rat_free` dans la boucle pour libérer un rationnel à chaque tour de boucle : il est remplacé par le nouveau rationnel que `rat_addint` a alloué juste avant.

main.c

```
#include "ratio.h"

ratio* sqrt_deux(int n) {
  int i;
  ratio *r, *res;
  // on démarre avec 1/2 et on applique la transfo : x -> 1/(2+x)
  r = rat_init(1,2);
  for (i=1; i<n; i++) {
    res = rat_addint(r,2);
    rat_inv(res);
    rat_free(r);
    r = res;
  }
  res = rat_addint(r,1);   // le +1 de la fin
  rat_free(r);      // bien libérer tout ce qui ne sert plus avant le return
  return res;
}

int  main() {
  int i;
  ratio* r;
  for (i=1; i<30; i++) {
    printf("précision %d => ",i);
    r = sqrt_deux(i);
    rat_print(r);                        // on affiche la fraction
    printf(" = %.10f\n", rat_eval(r));  // et son évaluation
    rat_free(r);
  }
  return 0;
}
```

2.b] La fonction est très simple (il faut quand même faire les `mat_free` au bon moment), mais on manque très vite de précision avec des simples *`int`* pour représenter un rationnel :

main.c

```
ratio* sum_carres(int n) {
  int i;
  ratio *r1, *r2, *res;
  r1 = rat_init(0,1);
  for (i=1; i<=n; i++) {
    r2 = rat_init(1, i*i);
    res = rat_add(r1,r2);
    rat_free(r1);
    rat_free(r2);
    r1 = res;
  }
  return res;
}
```

En utilisant des *long long int* au lieu d'*int* dans la structure *ratio* on peut aller un peu plus loin, mais on reste loin d'une bonne approximation de $\frac{\pi^2}{6}$. Il faudrait utiliser des entiers multi-précision (comme pour le 100 ! du TD 02) pour faire mieux, ou alors directement tout calculer en *double* si une valeur approchée suffit.

2.c] L'équation $x = \frac{b}{2a+x}$ a pour solution une des racines de $x^2+2ax-b$ qui sont $\frac{-2a\pm\sqrt{4a^2+2b}}{2}$. La racine positive est $\sqrt{N}-a$. Il suffit donc de calculer cette fraction continue et d'y ajouter a pour obtenir la racine carrée de N. La principale difficulté est de savoir quand arrêter les calculs car il n'est pas évident de savoir quand les *int* utilisés pour la représentation vont déborder. Sinon, le code ressemble beaucoup au code de la racine de 2.

main.c

```
1  double racine(int n) {
2    int a, b, i;
3    ratio *r1, *r2;
4    double res;
5
6    // on calcule le plus grand carré plus petit que n
7    a=1;
8    while ((a+1)*(a+1) <= n) {
9      a++;
10   }
11   b = n-a*a;
12
13   // si n est un carré, le travail est fini
14   if (b == 0) {
15     return a;
16   }
17
18   // sinon on démarre à b/2a et on itère la fonction : x -> b/(2a+x)
19   r1 = rat_init(b,2*a);
20   while ((r1->numer < 10000000) && (r1->denom < 10000000)) {
21     r2 = rat_addint(r1, 2*a);
22     rat_free(r1);
23     rat_inv(r2);
24     // pour faire propre, il faudrait un fonction rat_mulint dans ratio.c
25     r2->numer *= b;
26     r1 = r2;
27   }
28   r2 = rat_addint(r1,a); // on ajoute le a de la fin
29   res = rat_eval(r2);
30   rat_free(r1);          // on libère tous les rationnels
31   rat_free(r2);
32   return res;
33 }
```