

Correction

Remise en route, debugging, exceptions.

Exercice 1

Assez naturellement la structure de donnée à utiliser est une struct contenant un entier pour le numérateur (p) et un entier non signé pour le dénominateur (q) :

```
struct Rationnel {  
    int p;  
    unsigned int q;  
};
```

La fonction affiche ne présente aucune difficulté, si ce n'est une attention spéciale au cas particulier où le dénominateur vaut 1 :

```
void affiche(Rationnel r) {  
    cout << r.p;  
    if (r.q != 1) cout << "/" << r.q;  
}
```

La première version du programme est donc :

```
#include <iostream>  
using namespace std;  
  
// --- structure de données pour représenter les rationnels  
struct Rationnel {  
    int p;  
    unsigned int q;  
};  
  
void affiche(Rationnel r);  
  
int main()  
{  
    Rationnel r1 = { 1, 2 },  
               r2 = { -3, 5 },  
               r3 = { 2, 1 };  
  
    affiche(r1); cout << endl;  
    affiche(r2); cout << endl;  
    affiche(r3); cout << endl;  
  
    return 0;
```

```

}

/*
=====
*
*
=====
*/
void affiche(Rationnel r) {
    cout << r.p;
    if (r.q != 1) cout << "/" << r.q;
}

```

Il nous faut ensuite coder les opérations.

Concentrons nous avant tout sur la réduction d'un nombre rationnel.

Assez facilement nous avons :

```

void reduction(Rationnel& r) {
    unsigned int s(pgcd(r.p, r.q));
    if (s != 1) {
        r.p /= s;
        r.q /= s;
    }
}

```

Mais il faut juste faire attention au signe de p :

```

void reduction(Rationnel& r) {
    unsigned int s;
    if (r.p < 0)
        s = pgcd(-r.p, r.q);
    else
        s = pgcd(r.p, r.q);
    if (s != 1) {
        r.p /= s;
        r.q /= s;
    }
}

```

L'addition vient ensuite très facilement :

```

Rationnel addition(Rationnel r1, Rationnel r2) {
    Rationnel rez = { r1.p*r2.q + r2.p*r1.q,
                     r1.q*r2.q };
    reduction(rez);
    return rez;
}

```

On peut aussi écrire :

```

Rationnel addition(Rationnel r1, Rationnel r2) {
    Rationnel rez;

```

```

    rez.p = r1.p*r2.q + r2.p*r1.q;
    rez.q = r1.q*r2.q;
    reduction(rez);
    return rez;
}

```

On arrive donc à ce stade à quelque chose comme :

```

#include <iostream>
#include <string> // pour le message d'erreur
using namespace std;

// --- structure de données pour représenter les rationnels
struct Rationnel {
    int p;
    unsigned int q;
};

void affiche(Rationnel r);
Rationnel addition(Rationnel r1, Rationnel r2);
void reduction(Rationnel& r);
unsigned int pgcd(unsigned int a, unsigned int b);

// fonctions en extra pour faciliter l'écriture du main()
// pas du tout nécessaires pour le corrigé
void afficheNL(const Rationnel& r) { affiche(r); cout << endl; }
void afficheADD(const Rationnel& r1, const Rationnel& r2) {
    affiche(r1); cout << " + "; affiche(r2); cout << " = ";
    afficheNL(addition(r1, r2));
}

int main()
{
    Rationnel r1 = { 1, 2 },
               r2 = { -3, 5 },
               r3 = { 2, 1 };

    afficheNL(r1);
    afficheNL(r2);
    afficheNL(r3);

    afficheADD(r1, r2);
    afficheADD(r3, r2);
    afficheADD(r3, r3);

    return 0;
}

```

```

/*
=====
*
*
=====
*/
void affiche(Rationnel r) {
    cout << r.p;
    if (r.q != 1) cout << "/" << r.q;
}

/*
=====
*
* version simplifiée par rapport à la série 11.
*
* mais la version donnée dans le corrigé de cette série va aussi
bien.  *
*
=====
*/
unsigned int pgcd(unsigned int a, unsigned int b)
{
    int x(a), y(b), r;

    while (y != 0) {
        r = x%y;
        x = y;
        y = r;
    }

    return x;
}

/*
=====
*
*
=====
*/
void reduction(Rationnel& r) {
    unsigned int s;
    if (r.p < 0)
        s = pgcd(-r.p, r.q);
    else
        s = pgcd(r.p, r.q);
    if (s != 1) {
        r.p /= s;

```

```

        r.q /= s;
    }
}

/*
=====
*
*
=====
*/
Rationnel addition(Rationnel r1, Rationnel r2) {
    Rationnel rez = { r1.p*r2.q + r2.p*r1.q,
                     r1.q*r2.q };
    reduction(rez);
    return rez;
}

```

La soustraction et la multiplication ne pose aucun problème particulier :

```

/*
=====
*
*
=====
*/
Rationnel soustraction(Rationnel r1, Rationnel r2) {
    Rationnel rez = { r1.p*r2.q - r2.p*r1.q,
                     r1.q*r2.q };
    reduction(rez);
    return rez;
}

/*
=====
*
*
=====
*/
Rationnel multiplication(Rationnel r1, Rationnel r2) {
    Rationnel rez = { r1.p*r2.p, r1.q*r2.q };
    reduction(rez);
    return rez;
}

```

Quant à la division, il faut juste faire attention au signe de p2 comme indiqué dans l'énoncé :

```

Rationnel division(Rationnel r1, Rationnel r2) {
    Rationnel rez;
    if (r2.p < 0) {
        rez.p = (-r1.p) * r2.q;
    }
}

```

```

    rez.q = (-r2.p) * r1.q;
} else if (r2.p > 0) {
    rez.p = r1.p * r2.q;
    rez.q = r2.p * r1.q;
} else {
    // il faudrait normalement lancer une exception ici
    cout <<"Erreur: division par zéro." << endl;
    rez.p = 0; rez.q = 1; // arbitraire
}
reduction(rez);
return rez;
}

```

On aboutit donc finalement au code :

```

#include <iostream>
using namespace std;

// --- structure de données pour représenter les rationnels
struct Rationnel {
    int p;
    unsigned int q;
};

void affiche(Rationnel r);
Rationnel addition(Rationnel r1, Rationnel r2);
Rationnel soustraction(Rationnel r1, Rationnel r2);
Rationnel multiplication(Rationnel r1, Rationnel r2);
Rationnel division(Rationnel r1, Rationnel r2);
void reduction(Rationnel& r);
unsigned int pgcd(unsigned int a, unsigned int b);

// fonction en extra pour faciliter l'écriture du main()
// pas du tout nécessaires pour le corrigé
void afficheNL(const Rationnel& r) { affiche(r); cout << endl; }
void afficheADD(const Rationnel& r1, const Rationnel& r2) {
    affiche(r1); cout << " + "; affiche(r2); cout << " = ";
    afficheNL(addition(r1, r2));
}
void afficheSOUS(const Rationnel& r1, const Rationnel& r2) {
    affiche(r1); cout << " - "; affiche(r2); cout << " = ";
    afficheNL(soustraction(r1, r2));
}
void afficheMULT(const Rationnel& r1, const Rationnel& r2) {
    affiche(r1); cout << " * "; affiche(r2); cout << " = ";
    afficheNL(multiplication(r1, r2));
}
void afficheDIV(const Rationnel& r1, const Rationnel& r2) {

```

```

    affiche(r1); cout << " / ("; affiche(r2); cout << ") = ";
    afficheNL(division(r1, r2));
}

```

```

int main()

```

```

{
    Rationnel r1 = { 1, 2 },
                r2 = { -3, 5 },
                r3 = { 2, 1 };

```

```

    afficheNL(r1);
    afficheNL(r2);
    afficheNL(r3);

```

```

    afficheADD(r1, r2);
    afficheADD(r3, r2);
    afficheADD(r3, r3);

```

```

    afficheSOUS(r1, r2);
    afficheSOUS(r3, r2);
    afficheSOUS(r2, r2);

```

```

    afficheMULT(r1, r2);
    afficheMULT(r3, r2);
    afficheMULT(r2, r2);

```

```

    afficheDIV(r1, r2);
    afficheDIV(r3, r2);
    afficheDIV(r2, r2);

```

```

return 0;
}

```

```

/*
=====
*
*
=====
*/
void affiche(Rationnel r) {
    cout << r.p;
    if (r.q != 1) cout << "/" << r.q;
}

```

```

/*
=====
*
*

```

```

=====
*/
unsigned int pgcd(unsigned int a, unsigned int b)
{
    int x(a), y(b), r;

    while (y != 0) {
        r = x%y;
        x = y;
        y = r;
    }

    return x;
}

/*
=====
*
*
=====
*/
void reduction(Rationnel& r) {
    unsigned int s;
    if (r.p < 0)
        s = pgcd(-r.p, r.q);
    else
        s = pgcd(r.p, r.q);
    if (s != 1) {
        r.p /= s;
        r.q /= s;
    }
}

/*
=====
*
*
=====
*/
Rationnel addition(Rationnel r1, Rationnel r2) {
    Rationnel rez = { r1.p*r2.q + r2.p*r1.q,
                      r1.q*r2.q };
    reduction(rez);
    return rez;
}

/*
=====

```

```

*
*
=====
*/
Rationnel soustraction(Rationnel r1, Rationnel r2) {
    Rationnel rez = { r1.p*r2.q - r2.p*r1.q,
                      r1.q*r2.q };
    reduction(rez);
    return rez;
}

/*
=====
*
*
=====
*/
Rationnel multiplication(Rationnel r1, Rationnel r2) {
    Rationnel rez = { r1.p*r2.p, r1.q*r2.q };
    reduction(rez);
    return rez;
}

/*
=====
*
*
=====
*/
Rationnel division(Rationnel r1, Rationnel r2) {
    Rationnel rez;
    if (r2.p < 0) {
        rez.p = (-r1.p) * r2.q;
        rez.q = (-r2.p) * r1.q;
    } else if (r2.p > 0) {
        rez.p = r1.p * r2.q;
        rez.q = r2.p * r1.q;
    } else {
        // il faudrait normalement lancer une exception ici
        cout <<"Erreur: division par zéro." << endl;
        rez.p = 0; rez.q = 1; // arbitraire
    }
    reduction(rez);
    return rez;
}

```

Exercice 2

Cet exercice ne devrait présenter aucune difficulté une fois le cours assimilé et/ou l'exercice 0 fait.

Si vous ne comprenez pas la solution, refaites [l'exercice 0](#) de cette série.

Solution :

```
#include <string> // pour le message d'erreur
...
Rationnel division(Rationnel r1, Rationnel r2) throw(string);
...
int main()
{
    Rationnel r1 = { 1, 2 };
    Rationnel zero = { 0, 1 };
    try {
        afficheDIV(r1, zero);
    }
    catch (string& erreur) {
        cerr << erreur << endl;
        cout << "pas défini" << endl;
    }

    return 0;
}
...
/*
=====
*
*
=====
*/
Rationnel division(Rationnel r1, Rationnel r2) throw(string) {
    Rationnel rez;
    if (r2.p < 0) {
        rez.p = (-r1.p) * r2.q;
        rez.q = (-r2.p) * r1.q;
    } else if (r2.p > 0) {
        rez.p = r1.p * r2.q;
        rez.q = r2.p * r1.q;
    } else {
        throw string("Erreur: division par zéro.");
    }
    reduction(rez);
    return rez;
}
```

Exercice 4 : Compression RLE

```
#include <string>
#include <iostream>
#include <sstream>
using namespace std;

const char FLAG('#');

struct Erreur {
    string message;
    string decode;
    string suite;
};

string compresse(const string& data, char flag);
string decompresse(const string& rldata, char flag) throw(Erreur);

string toStr(int data)
{
    /*
     * conversion d'entier sous forme de string
     *
     * Exemple:
     *     int i = 4;
     *     string s = toStr(i);
     */
    ostringstream os;
    os << data;
    return os.str();
}

int fromStr(const string& str)
{
    /*
     * conversion de string vers entier
     *
     * Exemple:
     *     string s = "41";
     *     int i = fromStr(s);
     */
    int extracted;
    istringstream is(str);
    is >> extracted;
```

```

        return extracted;
    }

int fromStr(char c)
{
    /*
     * conversion de char vers entier
     *
     * Exemple:
     *     char c = '4';
     *     int i = fromStr(c);
     */
    if ((c >= '0') && (c <= '9'))
        return c - '0';
    else
        return 0;
}

int main()
{
    string dta;
    cout << "Entrez les données à compresser : ";
    getline(cin,dta);
    string rle(comprime(dta, FLAG));
    cout << "Forme compressée:  " << rle << endl
         << "[ratio = " << rle.size()*100.0/dta.size() << "%]" <<
endl;
    string dcp(decomprime(rle, FLAG));
    if (dcp != dta)
        cout << "Erreur - données corrompues:" << endl
             << dcp << endl;
    else cout << "décompression ok!" << endl;
    // teste la decompression
    cout << "Entrez les données à décompresser : ";
    getline(cin,dta);
    try {
        dcp = decomprime(dta, FLAG);
        cout << "décompressé : " << dcp << endl;
    }
    catch (Erreur& erreur) {
        cout << "Erreur de décompression : " << erreur.message
             << endl;
        cout << "décodé à ce stade : " << erreur.decode << endl;
        cout << "non décodé          : " << erreur.suite << endl;
    }
}

string compress(const string& data, char flag)

```

```

{
    char c;                // le caractère à encoder
    int l;                 // le nombre de répétitions
    unsigned int p = 0;    // position dans la chaîne à encoder
    string out;            // résultat de l'encodage

    while (p < data.size()) {
        l = 1;
        if ((c = data[p]) == flag) {
            // séquence spéciale: flag 0
            p++;
            out += c;
            out += toString(0);
        }
        else {
            while ((p++ < data.size()) && (l < 9) && (data[p] == c)) l++;
            if (l >= 3) {
                out += c;
                out += flag;
                out += toString(l);
            }
            else while (l-- > 0) out += c;
        }
    }
    return out;
}

```

```

string decompresse(const string& rldata, char flag) throw(Erreur)
{
    char c;                // caractère extrait
    int l;                 // nombre de répétitions à produire
    unsigned int p;        // position dans la chaîne à décompresser
    string out;            // chaîne décompressée

    for (p = 0; p < rldata.size(); p++) {
        c = rldata[p];
        if ((p+1 < rldata.size()) && (rldata[p+1] == flag)) {
            // flag en p+1 détecté ?
            p += 2;
            if (p >= rldata.size()) {
                // erreur : on devrait avoir qqchode derrière le FLAG
                Erreur err;
                err.message = "flag ";
                err.message += flag;
                err.message += " sans rien derrière";
                err.decode = out + c;
                err.suite = flag + rldata.substr(p);
                throw err;
            }
        }
    }
}

```

```

    } else if ((rldata[p] >= '0') && (rldata[p] <= '9')) {
        l = fromStr(rldata[p]); // on récupère l
        if (l >= 1)
            while (l-- > 0) out += c; // décompression des l x c
        else {
            // l=0 -> le flag était dans les données
            out += c;
            out += flag;
        }
    } else {
        // erreur : ce qui suit le FLAG n'est pas correct
        Erreur err;
        err.message = "caractère ";
        err.message += rldata[p];
        err.message += " incorrect après le flag ";
        err.message += flag;
        err.decode = out + c;
        err.suite = flag + rldata.substr(p);
        throw err;
    }
    } else {
        // caractère seul
        out += c;
        if (c == flag) p++; // saute le 0 dans le cas d'un flags au début
    }
    ("#0")
    }
    }
    return out;
}

```

Question subsidiaire :

- pour coder de grandes séquences consécutives, on peut par exemple utiliser les valeurs 'spéciales' de L (dans notre cas 0, 1 et 2, puisque les chaînes de longueurs inférieure à 3 ne sont pas compressées. Nous utilisons déjà le 0 pour le flag seul) pour indiquer une extension du codage du nombre de répétitions.

Par exemple :

- L = 1 -> codage sur par exemple 2 digits => on passe à une longueur max de 99 caractères
- L = 2 -> codage sur par exemple 3 digits => Lmax = 999

et on peut continuer le raisonnement : avec L = 1, la longueur 'minimale' codée de manière étendue devrait être 10 (sinon on code comme avant). **1b)** on peut donc encoder L avec un décalage de 10 -> L code donc de 10 (#100) à 109 (#199), Lmax devient donc 109 au lieu de 99. **1c)** on peut utiliser à nouveau tout ou une partie de ces valeurs pour prolonger l'encodage étendu...Exemple : on doit coder 105 caractères 'c' consécutifs: cas 1: c#199c#6 (1->99 puis 105-99=6 codés en normal) cas 2: c#2105 (2->105 sur 3 digits)

cas 1b: c#195 (1->95+bias = 95+10 = 105)

- On peut utiliser la même technique pour coder des répétitions du flag ; dans ce cas, on peut utiliser par exemple la valeur spéciale '2' pour indiquer que le flag est effectivement compressé, et le nombre de répétitions est encodé sur le prochain caractère.

Exemple : à compresser: aa#####bb -> aa#25bb

On constate que dans ce cas, il devient intéressant de coder des répétitions de 2 flags déjà (3 caractères (#22) contre 4 avec la version '#0' (#0#0))

En résumé, il est possible d'utiliser les valeurs 'impossibles' de L pour indiquer des cas spéciaux, ou encore encoder les valeurs avec un certain biais. On étend ainsi légèrement la plage de valeurs représentable, pour un très faible surcoût de complexité. À ce propos, l'un des avantages de la compression RLE est justement sa faible complexité algorithmique (i.e. temps consommé pour réaliser la compression).

Dernière mise à jour : 2007/03/20 16:23:00 (1.6)