Correction
Révisions, exceptions, debugging.

Exercice 1 | Exercice 2 | Exercice 4

Exercice 1 : arithmétique rationnelle

> Exercice n°40 (pages 93 et 276) de l'ouvrage *C++ par la pratique*.

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(const 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(const 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(const 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(pgdc(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 = pgdc(-r.p, r.q);
  else
    s = pgdc( r.p, r.q);
  if (s != 1) {
    r.p /= s;
    r.q /= s;
  }
}
```

L'addition vient ensuite très facilement :

```
Rationnel addition(const Rationnel& r1, const Rationnel& r2)
{
  Rationnel rez = { r1.p*r2.q + r2.p*r1.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(const Rationnel& r);
Rationnel addition(const Rationnel& r1, const Rationnel& r2);
void reduction(Rationnel& r);
unsigned int pgdc(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(const 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 pgdc(unsigned int a, unsigned int b)
{
  unsigned int r;
  while (b != 0) {
    r = a % b;
    a = b;
    b = r;
  }
  return a;
}

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

// ========================================================================
Rationnel addition(const Rationnel& r1, const 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 posent aucun problème particulier :

```
Rationnel soustraction(const Rationnel& r1, const Rationnel& r2)
{
```

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

Rationnel multiplication(const Rationnel& r1, const 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(const Rationnel& r1, const 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,  *
     * mais ce n'est pas le sujet de ce chapitre.         */
    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>
#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(const Rationnel& r);
Rationnel addition      (const Rationnel& r1, const Rationnel& r2);
Rationnel soustraction  (const Rationnel& r1, const Rationnel& r2);
Rationnel multiplication(const Rationnel& r1, const Rationnel& r2);
Rationnel division      (const Rationnel& r1, const Rationnel& r2);
void reduction(Rationnel& r);
unsigned int pgdc(unsigned int a, unsigned int b);

/* fonctions en extra pour faciliter l'écriture du main(); *
 * pas du tout nécéssaires 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(const Rationnel& r)
{
  cout << r.p;
  if (r.q != 1) cout << "/" << r.q;
}

// ============================================================================
unsigned int pgdc(unsigned int a, unsigned int b)
{
  unsigned int r;
  while (b != 0) {
    r = a % b;
    a = b;
    b = r;
  }
  return a;
}

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

```
    r.p /= s;
    r.q /= s;
  }
}

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

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

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

// ===========================================================================
Rationnel division(const Rationnel& r1, const 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 !
    cerr << "Erreur: division par zéro.") << endl;
    rez.p = 0; rez.q = 1; // arbitraire
  }
  reduction(rez);
  return rez;
}
```

Exercice 2 : exceptions

> Exercice n°68 (pages 179 et 375) de l'ouvrage *C++ par la pratique*.

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(const Rationnel& r1, const Rationnel& r2);
...
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(const Rationnel& r1, const 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 {
    throw "Erreur: division par zéro."s;
  }
  reduction(rez);
  return rez;
}
```

Exercice 4 : compression RLE

> Exercice n°68 (pages 179 et 375) de l'ouvrage *C++ par la pratique*.

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

constexpr char FLAG('#');

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

string compresse(string_view data, char flag);
string decompresse(string_view rledata, char flag);

int main()
{
  string dta;
  cout << "Entrez les données à comprimer : ";
  getline(cin, dta);
  string rle(compresse(dta, FLAG));
  cout << "Forme compressée:   " << rle << endl
       << "[ratio = " << rle.size()*100.0/dta.size() << "%]" << endl;
  string dcp(decompresse(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 = decompresse(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 compresse(string_view data, char flag)
{
  size_t p(0); // position dans la chaine à encoder
  string out;  // résultat de l'encodage

  while (p < data.size()) {
    int  l(1);       // le nombre de répétitions
    char c(data[p]); // le caractère à encoder
    if (c == flag) {
      // séquence spéciale: flag 0
      ++p;
      out += c;
      out += to_string(0);
    } else {
      while ((p++ < data.size()) and (l < 9) and (data[p] == c)) ++l;
      if (l >= 3) {
        out += c;
        out += flag;
        out += to_string(l);
      }
      else while (l-- > 0) out += c;
    }
  }
  return out;
}

string decompresse(string_view rledata, char flag)
{
  string out;     // chaine décompressée

  for (size_t p(0); p < rledata.size(); ++p) {
    char c(rledata[p]);          // caractère extrait
    if ((p+1 < rledata.size()) and (rledata[p+1] == flag)) {
      // flag en p+1 détecté ?
      p += 2;
      if (p >= rledata.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;
        err.suite += rledata.substr(p);
        throw err;
      } else if ((rledata[p] >= '0') and (rledata[p] <= '9')) {
        int l(rledata[p] - '0'); // 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 += rledata[p];
          err.message += " incorrect après le flag ";
          err.message += flag;
          err.decode = out + c;
          err.suite = flag;
          err.suite += rledata.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 inférieur 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 :

  1. `L = 1` -> codage sur par exemple 2 digits => on passe à une longueur max de 99 caractères
  2. `L = 2` -> codage sur par exemple 3 digits => `L`max = 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`), `L`max 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 caracteres '`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+biais = 95+10 = 105)

- On peut utiliser la même technique pour coder des répétions 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 répétitons 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 avantage de la compression RLE est justement sa faible complexité algorithmique (i.e. temps consommé pour réaliser la compression).

Dernière mise à jour : $Date: 2012-11-09 14:16:46 +0100 (ven 09 nov 2012) $ ($Revision: 191 $)