The mathematician is entirely free,
within the limits of his imagination,
to construct what worlds he pleases.
John W. N. Sullivan

CHAPITRE XII

Exemples d'anneaux effectifs

Objectifs

- Expliciter ce qu'il faut pour rendre un anneau effectif.
- Implémenter les nombres rationnels comme un exemple de base.
- Plus généralement, implémenter le corps des fractions ou un anneau quotient.

Un anneau est *effectif* s'il existe une manière de *représenter* chacun de ses éléments sur ordinateur et des algorithmes pour *effectuer* chacune des opérations de l'anneau dans la représentation choisie. L'approche effective est une vraie restriction quant aux anneaux en question et un important défi quant à l'ingéniosité algorithmique. Ce chapitre précise ce que l'on exige d'un anneau effectif, discute quelques exemples typiques, et présente des implémentations possibles.

Notre exemple phare est, bien sûr, l'anneau $\mathbb{Z}$. La représentation des entiers et les algorithmes de base ont été discutés aux chapitres II, IX, et XI. On discute dans le présent chapitre son corps des fractions $\mathbb{Q}$ et on reconsidère les quotients $\mathbb{Z}_n$, constructions que l'on généralise à d'autres anneaux. Le projet à la fin traite de l'anneau $\mathbb{Z}[i]$ des entiers de Gauss.

Voici trois exemples importants et assez représentatifs pour illustrer l'étendue du concept :

Exemple 0.1. L'anneau $\mathbb{Q}[\sqrt{2}]$ est effectif : c'est un $\mathbb{Q}$-espace vectoriel à base $(1,\sqrt{2})$. On peut donc travailler avec des couples $(a,b)\in\mathbb{Q}^2$ pour représenter tout élément $a+b\sqrt{2}$ de manière unique. *Exercice.* — Exprimer les opérations de l'anneau $\mathbb{Q}[\sqrt{2}]$ sous cette forme.

Exemple 0.2. Si A est effectif, alors l'anneau $A[X]$ des polynômes sur A est effectif. *Exercice.* — L'idée évidente marchera : on représente tout polynôme $P\in A[X]$ par la suite *finie* de ses coefficients dans A. Détailler cette représentation et les opérations de l'anneau $A[X]$.

Exemple 0.3. Le corps $\mathbb{R}$ des nombres réels, par contre, n'est pas effectif. Ceci surprend beaucoup les débutants, mais la vie est ainsi faite. Évidemment le corps $\mathbb{R}$ est de grande importance ; toute l'analyse s'appuie sur lui, et dans des applications il est souvent indispensable d'effectuer des calculs « réels » sur ordinateur. Les difficultés *d'approcher* des calculs dans $\mathbb{R}$ sur ordinateur font l'objet du calcul numérique.

Attention. — Les types `float` et `double` *ne modélisent pas* le corps des nombres réels ! Loin de cela, ils ne représentent qu'un *sous-ensemble fini* de nombres rationnels, ce qui n'a rien à voir avec $\mathbb{R}$.

Remarque 0.4. Voici un argument intuitif mais *faux* que $\mathbb{R}$ n'est pas effectif : « il est impossible de stocker une suite infinie de décimales pour représenter un nombre irrationnel de manière exacte. » Certes, mais nul ne nous oblige de représenter nos nombres par un développement décimal ! Les sous-anneaux $\mathbb{Q}[\sqrt{2}]$ et $\mathbb{Q}[\pi]$ de $\mathbb{R}$, par exemple, contiennent des nombres irrationnels, mais on peut très bien les représenter sur ordinateur ! (Comment ?)

Si l'argument intuitif est faux, il nous met tout de même sur la bonne piste. Après réflexion, on en extrait une réponse plus solide : la représentation d'un nombre x sur ordinateur doit se faire par un objet fini (arbitrairement grand, mais toujours fini pour chaque x individuellement). On peut ainsi représenter seulement un nombre *dénombrable* d'éléments, mais le corps $\mathbb{R}$ est *non dénombrable*.

Sommaire

1. De l'anneau $\mathbb{Z}$ au corps $\mathbb{Q}$

On commence par un exemple concret et pratique, que l'on généralisera dans la suite : le passage de l'anneau $\mathbb{Z}$ des nombres entiers au corps $\mathbb{Q}$ des nombres rationnels. On révisera d'abord la construction du corps des fractions, puis on passera à l'implémentation en C++.

1.1. Qu'est-ce qu'un corps de fractions ? Certaines équations du type $a = bx$ avec $a, b \in \mathbb{Z}$ n'ont pas de solution x dans $\mathbb{Z}$. C'est le cas, par exemple, pour l'équation $2 = 3x$. Il serait pourtant très utile de disposer d'une solution ! Pour l'anneau $\mathbb{Z}$ vous connaissez bien sûr la solution de ce problème : on construit le *corps des fractions* $\mathbb{Q}$. On établit ainsi une théorie plus large mais toujours cohérente, dans laquelle le problème initial se retrouve et se résolve. Essayons d'en dégager une réponse générale, qui englobe tous les anneaux commutatif unitaires.

Exercice/M 1.1. Soit A un anneau commutatif unitaire. Optimiste que nous sommes, supposons dans un premier temps qu'il existe un homomorphisme injectif $\iota : A \to L$ dans un corps L. Afin de simplifier la notation nous pouvons identifier A avec son image $\iota(A)$, et ainsi supposer que $A \subset L$. (Pourquoi ?) Dans ce cas on peut regarder l'ensemble $\mathrm{Frac}(A) \subset L$ formé des éléments $\frac{a}{b} = ab^{-1}$ tels que $a \in A$ et $b \in A^*$. Vérifier d'abord que $\frac{a}{b} = \frac{c}{d}$ si et seulement si $ad = bc$. Établir ensuite les règles suivantes : $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$ et $\frac{a}{b} \cdot \frac{c}{d} = \frac{ac}{bd}$. En déduire que $\mathrm{Frac}(A)$ est un sous-corps de L, contenant A. Plus précisément c'est le plus petit sous-corps de L qui contienne A. On l'appelle *corps des fractions de A dans L*. Forts de cette vérification rassurante, formulons-en la définition générale :

Définition 1.2. Soit A un anneau commutatif unitaire. Un *corps de fractions* de A est la donnée d'un couple (K, ι) où K est un corps et $\iota : A \to K$ est un homomorphisme d'anneaux injectif, de sorte que tout élément $x \in K$ s'écrive comme $x = \iota(a)\iota(b)^{-1}$ avec $a \in A$ et $b \in A^*$.

Exemple 1.3. $\mathbb{Q}$ est un corps de fractions de $\mathbb{Z}$ via l'inclusion $\iota : \mathbb{Z} \subset \mathbb{Q}$. Par contre, $\mathbb{R}$ n'est pas un corps de fractions de $\mathbb{Z}$: certains éléments $x \in \mathbb{R}$ ne s'écrivent pas comme $\frac{a}{b}$ avec $a \in A$ et $b \in A^*$. Soit p premier et $\phi : \mathbb{Z} \to \mathbb{Z}_p$ l'homomorphisme canonique. Tout élément $x \in \mathbb{Z}_p$ s'écrit comme $\phi(a)\phi(1)^{-1}$ avec $a \in \mathbb{Z}$. Le couple $(\mathbb{Z}_p, \phi)$ n'est cependant pas un corps de fractions, car l'homomorphisme ϕ n'est pas injectif.

Exercice/M 1.4. Un corps de fractions (K, ι) de A se réjouit de la propriété universelle suivante : si $\phi : A \to L$ est un homomorphisme injectif dans un corps L, alors il existe un et un seul homomorphisme $\Phi : K \to L$ tel que $\Phi \circ \iota = \phi$. Il en découle que le corps des fractions (K, ι) de A est unique à isomorphisme près. Formulez précisément ce que cela veut dire, puis prouvez votre énoncé.

Exercice/M 1.5. Contrairement à l'unicité, l'existence n'est pas toujours donnée : montrer qu'un anneau non intègre n'admet pas de corps de fractions. Heureusement pour nous, c'est le seul obstacle :

Théorème 1.6. *Tout anneau intègre admet un corps de fractions.*

Exercice/M 1.7. Où chercher ce corps ? Comment prouver son existence ? Il n'y a qu'une seule possibilité : il faut le construire ! Démontrer le théorème en détaillant la construction suivante.

ESQUISSE DE PREUVE. Supposons que A est un anneau intègre. Notre but est de donner un sens aux quotients $\frac{a}{b}$ dans un corps encore à construire. En s'inspirant des propriétés souhaitées, on considère l'ensemble $F = A \times A^*$ avec les opérations $(a,b) + (c,d) := (ad + bc, bd)$ et $(a,b) \cdot (c,d) := (ac, bd)$. On constate que $(F, +, \cdot)$ *n'est pas* un anneau, encore moins un corps. (Pourquoi ? Il y a une exception : laquelle ?) Afin de sortir de cette impasse, on définit la relation $(a,b) \sim (c,d)$ si $ad = bc$. On vérifie d'abord qu'il s'agit d'une relation d'équivalence, ensuite que les opérations $+$ et $\cdot$ sont bien définies sur le quotient $K = F/\sim$, et finalement que $(K, +, \cdot)$ est un corps. (Le détailler ! Où utilise-t-on l'intégrité de l'anneau A dans ce raisonnement ?) La classe d'équivalence de (a,b) est notée $\frac{a}{b}$. L'application $\iota : A \to K$ donnée par $a \mapsto \frac{a}{1}$ est un homomorphisme d'anneaux injectif. On peut ainsi identifier l'anneau A avec son image $\iota(A)$. Ceci fait, on constate que tout élément $x \in K$ s'écrit comme $x = ab^{-1}$ avec $a \in A$ et $b \in A^*$. □

Exercice/M 1.8. Vérifier que cette construction appliquée à $\mathbb{Z}$ donne le corps $\mathbb{Q}$ comme vous le connaissez. Tout corps de caractéristique 0 contient donc $\mathbb{Q}$ comme un sous-corps.

Exercice/M 1.9. Qu'obtient-on si l'on l'applique à $\mathbb{R}[X]$? Plus généralement à $K[X]$ sur un corps K ? Que se passe-t-il lorsqu'on l'applique à un anneau non intègre, comme $\mathbb{Z}_6$ par exemple ?

1.2. Implémentation du corps $\mathbb{Q}$. Une *classe* en C++ est un type défini par l'utilisateur. La construction de classes pour modéliser une situation donnée est le paradigme caractéristique de tout langage orienté objet. Ne citons que l'exemple qui nous a occupé dans les derniers chapitres : le type `int` de C++ ne modélise pas $\mathbb{Z}$ mais un anneau fini $\mathbb{Z}_n$, typiquement avec $n = 2^{32}$ ou $n = 2^{64}$. On a donc employé la classe `Integer`, issue de la bibliothèque GMP, qui modélise l'anneau $\mathbb{Z}$.

Jusqu'à présent nous ne disposons pas d'outils pour calculer avec des nombres rationnels. Ayant acquis suffisamment d'expérience, nous allons maintenant construire une telle classe nous-mêmes. À titre d'exemple, nous souhaitons une utilisation comme dans le programme suivant :

Programme XII.1 Exemple d'utilisation de la classe `Rationnel`

```
int main()
{
  cout << "Bienvenue au corps des nombres rationnels !" << endl
       << "Entrez deux éléments a,b sous la forme  num/den  svp : ";
  Rationnel a,b;
  cin >> a >> b;
  cout << "a   = " << a   << endl << "b   = " << b   << endl
       << "a+b = " << a+b << endl << "a-b = " << a-b << endl
       << "a*b = " << a*b << endl << "a/b = " << a/b << endl;
}
```

Nombres rationnels, version 0.1. Après s'être assuré de la construction du corps des fraction, nous allons « simplement » la traduire en C++ et ainsi construire notre classe `Rationnel`. Il faut d'abord choisir une représentation convenable, c'est-à-dire une structure des données adéquate. Dans notre cas c'est facile : tout élément $\frac{r}{s} \in \mathbb{Q}$ est caractérisé par deux entiers, un numérateur $r \in \mathbb{Z}$ et un dénominateur $s \in \mathbb{Z}^*$. En C++ ceci peut être modélisé comme suit :

Programme XII.2 Implémentation de la classe `Rationnel` – version 0.1

```
class Rationnel
{
public:
  Integer numer;
  Integer denom;
};
```

Ici le mot réservé `class` est suivi du nom de la classe, en l'occurrence `Rationnel`. Le corps de la classe est délimité par des accolades suivies d'un point-virgule. Il contient la définition des *éléments* ou *membres* de la classe : ici les deux variables `numer` et `denom` de type `Integer`.

Exemple 1.10. Avec cette définition de la classe `Rationnel` on pourrait écrire le code suivant :

```
Rationnel a;  a.numer= 4; a.denom= 6;
Rationnel b;  b.numer= 1; b.denom= 5;
```

La première ligne définit la variable `a` du type `Rationnel` ; on dit aussi que `a` est un *objet* de la classe `Rationnel`. Cet objet possède deux éléments : les variables `a.numer` et `a.denom`, auxquelles on affecte les valeurs `4` et `6` respectivement. Notre objet regroupe ces deux éléments en une seule entité. Il en est de même pour l'objet `b`. Lors de sa définition (ligne 2) sont créés ses deux éléments `b.numer` et `b.denom`, différents des éléments de `a` puisqu'il s'agit d'un *autre* objet. On dit que `a` et `b` sont deux *instances*, deux objets distincts de la classe `Rationnel`.

Remarque 1.11. Chaque objet `a`, `b`, ... mène sa vie individuelle : en particulier *l'état* d'un objet (la valeur prise par chacune des variables) est une *propriété individuelle*. Par contre, la classe décrit les *propriétés communes* : elle spécifie en l'occurrence que les deux éléments `numer` et `denom` sont toujours de type `Integer`, quelque soit l'objet en question. Comme ces deux éléments sont déclarés *publics*, on peut les utiliser comme des variables usuelles (voir l'exemple ci-dessus).

Nombres rationnels, version 0.2. Nous allons maintenant affiner notre implémentation en ajoutant des *méthodes* à la classe `Rationnel`. Commençons par la *normalisation*, qui repose sur une particularité des nombres rationnels (bien connue mais remarquable — la prouver) :

Proposition 1.12. *Tout nombre rationnel s'écrit comme une fraction rs^{-1} avec $r \in \mathbb{Z}$, $s \in \mathbb{Z}^*$, et on peut normaliser cette fraction de sorte que* $\operatorname{pgcd}(r,s) = 1$ *et* $s > 0$. *L'écriture normalisée est unique : si $rs^{-1} = uv^{-1}$ et chacune des fractions est normalisée, alors $r = u$ et $s = v$.*

La fonction `normaliser()` ci-dessous réduit toute fraction à cette forme normale.

Programme XII.3 Implémentation de la classe `Rationnel` – version 0.2

```
class Rationnel
{
public:
  Integer numer, denom;   // numérateur et dénominateur

  void normaliser(void)   // normaliser la fraction numer/denom
    { if ( denom==0 ) { cerr << "Division par zéro !\n"; exit(1); };
      Integer d= pgcd( numer, denom ); numer/= d; denom/= d;
      if ( denom<0 ) { numer= -numer; denom= -denom; }; };
};
```

Exemple 1.13 (suite)**.** On peut maintenant écrire

```
Rationnel a;  a.numer= 4; a.denom= 6;  a.normaliser();
```

Ceci réduit la représentation $4/6$ à la forme normale $2/3$, moyennant une fonction `pgcd` pour les entiers. La fonction `normaliser()` appartient à la classe `Rationnel`, ce qui est plus explicitement noté par `Rationnel::normaliser()`. Afin d'y accéder de l'extérieur de la classe, on appelle la fonction obligatoirement basée sur un objet, comme `a.normaliser()` dans l'exemple. Cet appel entraîne, naturellement, que la fonction soit appliquée à l'objet `a`. Il n'y a pas de sens d'appeler `normaliser()` sans aucun objet.

Remarque 1.14. Il serait également possible d'écrire une fonction

```
void normal( Rationnel& a )
{ if ( a.denom==0 ) { cerr << "Division par zéro !\n"; exit(1); };
  Integer d= pgcd( a.numer, a.denom ); a.numer/= d; a.denom/= d;
  if ( a.denom<0 ) { a.numer= -a.numer; a.denom= -a.denom; }; }
```

En utilisant cette fonction, notre exemple prendrait la forme

```
Rationnel a;  a.numer= 4; a.denom= 6;  normal(a);
```

Le résultat est le même, mais le point de vue est différent : la fonction `normal()` ne fait pas partie de la classe `Rationnel`, c'est une fonction extérieure. Par contre `Rationnel::normaliser()` est une méthode de la classe. Ceci peut entraîner différents droits d'accès, comme on verra plus bas.

Notez aussi la différence dans les noms des éléments : la fonction `normal()` doit adresser les éléments par `a.numer` et `a.denom`, tandis que la fonction `Rationnel::normaliser()` les appelle simplement `numer` et `denom` : évoquée par `a.normaliser()`, elle modifiera les éléments `a.numer` et `a.denom`.

Nombres rationnels, version 0.3. Afin d'*encapsuler* données et méthodes, une classe regroupe toutes les méthodes « essentielles » ou « caractéristiques » : elles font partie des propriétés communes des objets. Après avoir implémenté la normalisation nous voudrions maintenant ajouter des méthodes d'*affectation*. À titre d'exemple, on voudrait écrire

```
Rationnel a;  a.affecter(4,6);
```

ce qui devrait correspondre à `a.numer=4; a.denom=6;` suivi d'une normalisation. C'est une écriture naturelle, et la normalisation systématique évite que l'utilisateur oublie éventuellement d'appeler la fonction `normaliser()`. Voici une implémentation possible :

Programme XII.4 Implémentation de la classe `Rationnel` – version 0.3

```
class Rationnel
{
public:
  Integer numer, denom;   // numérateur et dénominateur

  void normaliser(void)   // normaliser la fraction numer/denom
    { if ( denom==0 ) { cerr << "Division par zéro !\n"; exit(1); };
      Integer d= pgcd(numer,denom); numer/= d; denom/= d;
      if ( denom<0 ) { numer= -numer; denom= -denom; }; };

  void affecter( const Integer& num, const Integer& den=1 )
    { numer= num; denom= den; normaliser(); };

  Rationnel& operator= ( const Rationnel& a )
    { numer= a.numer; denom= a.denom; return *this; };
};
```

Remarque 1.15. On voit dans la fonction `affecter()` que l'appel de la fonction `normaliser()` n'est pas précédé du nom de l'objet. Le compilateur comprend qu'il s'agit de l'objet courant sur lequel travaille la fonction `affecter()`.

Remarque 1.16. Notre implémentation se sert d'un paramètre initialisé par défaut. On peut ainsi appeler la fonction `affecter()` avec un seul argument, `num`, dans quel cas le compilateur pose `den=1` pour le deuxième argument. Ceci correspond à la conversion d'un entier de type `Integer` en `Rationnel` : on peut par exemple écrire `Rationnel a; a.affecter(5);` ce qui semble une écriture assez naturelle.

Quant à l'affectation par copie, on préfère évidemment une écriture courte et naturelle comme `a=b` à l'écriture longue est fastidieuse `a.numer=b.numer; a.denom=b.denom;` Dans ce but nous avons introduit l'opérateur `=` qui réalise cette affectation par copie. À noter que `a=b;` correspond plus explicitement à l'appel `a.operator=(b);` (le tester).

Remarque 1.17. En C++ un opérateur d'affectation renvoie une référence sur l'objet auquel on vient d'affecter la valeur. L'implémentation de la classe `Rationnel` se conforme à cette convention : Mais comment une fonction sait-elle sur quel objet elle opère ? Afin de résoudre cette crise d'identité, le mot réservé `this` fournit un pointeur sur l'objet courant, et `*this` est synonyme avec cet objet.

Nombres rationnels, version 0.4. Après avoir implémenté normalisation et affectation, nous voudrions ajouter un *constructeur*. Celui-ci est appelé exactement une fois lors de la création de l'objet, et sert à l'initialisation. En l'occurrence on souhaite que la définition `Rationnel a;` crée un objet `a` dont les deux éléments `a.numer` et `a.denom` soient initialisés à `0` et `1` respectivement. Il sera également commode de disposer des variantes à paramètres :

```
Rationnel a(4,6);                 // création de a avec initialisation simultanée
Rationnel b= Rationnel(4,6);      // création de b puis d'un objet auxiliaire anonyme
Rationnel c; c= Rationnel(4,6);   // création de c puis d'un objet auxiliaire anonyme
Rationnel d; d.affecter(4,6);     // création de d suivie d'une affectation séparée
Rationnel e; e.numer=4; e.denom=6; e.normaliser();
```

Ces définitions aboutissent toutes aux même résultat. La première est pourtant la plus naturelle et la plus efficace. L'implémentation suivante réalise un tel constructeur.

Remarque 1.18. Ne pas confondre construction et affectation : le constructeur n'est appelé qu'une seule fois pour l'objet `a`, à savoir lors de sa création. L'affectation par contre, s'effectue sur un objet déjà construit, et peut s'effectuer plusieurs fois pendant sa vie. Il existe aussi la notion de destructeur, qui est appelé pour détruire un objet à la fin de sa vie. Ici les variables `numer` et `denom` sont déjà munies du destructeur de la classe `Integer`, qui fait le nécessaire.

Programme XII.5 Implémentation de la classe `Rationnel` – version 0.4

```
class Rationnel
{
public:
  Integer numer, denom;   // numérateur et dénominateur

  void normaliser(void)   // normaliser la fraction numer/denom
    { if ( denom==0 ) { cerr << "Division par zéro !\n"; exit(1); };
      Integer d= pgcd(numer,denom); numer/= d; denom/= d;
      if ( denom<0 ) { numer= -numer; denom= -denom; }; };

  Rationnel( const Integer& num=0, const Integer& den=1 )
    : numer(num), denom(den) { normaliser(); };

  void affecter( const Integer& num, const Integer& den=1 )
    { numer= num; denom= den; normaliser(); };

  Rationnel& operator= ( const Rationnel& a )
    { numer= a.numer; denom= a.denom; return *this; };
};
```

Remarque 1.19. Un constructeur porte toujours le nom de la classe, `Rationnel` en l'occurrence. Comme il ne sert qu'à la création d'objets, il est impossible de spécifier un type de retour ; il ne renvoie jamais de valeur. Le constructeur peut prendre une liste de paramètres, ici deux paramètres de type `Integer` qui seront interprétés comme numérateur et dénominateur. L'implémentation initialise la variable `numer` à la valeur `num` et la variable `denom` à la valeur `den`, puis appelle la normalisation. Notre implémentation se sert à nouveau des paramètres initialisés par défaut. On peut donc appeler le constructeur sans aucun argument, dans quel cas le compilateur pose `num=0` et den=1. On peut l'appeler aussi avec un seul argument `num` : le compilateur pose alors `den=1`, ce qui correspond à la conversion de `Integer` en `Rationnel`.

Nombres rationnels, version 0.5. Jusqu'à présent la classe `Rationnel` n'est pas très utile : elle regroupe deux éléments `numer` et `denom` de type `Integer` mais elle ne permet pas encore de modéliser le corps $\mathbb{Q}$. Pour cela il faut encore implémenter les *opérations* nécessaires ; le programme XII.6 présente une implémentation plus complète.

Exercice/P 1.20. Tester l'implémentation de la classe `Rationnel` par un programme similaire à XII.1. Les opérations de base fonctionnent-elles comme souhaité ? Essayer d'expliquer leur fonctionnement en détail. Tester aussi les limites de notre implémentation actuelle : élargir votre programme en ajoutant de code de plus un plus exigeant vis à vis les opérations sur les nombres rationnels… Trouvez-vous des erreurs ? des incohérences ? des comportements contre-intuitifs ? Si oui, comment remédier à ces défauts ?

Exercice 1.21. Tester les définitions/affectations suivantes puis détailler leur fonctionnement.

```
Rationnel a(20,6);                      // ok, écriture fortement conseillée
Rationnel b= Rationnel(20,6);           // acceptable, mais non optimale
Rationnel c; c.affecter(20,6);          // acceptable, mais non optimale
Rationnel d= Rationnel(20)/Rationnel(6); // acceptable, encore moins optimale
Rationnel e(20); e.denom= 6;            // écriture désormais interdite
Rationnel f(20/6);                      // écriture trompeuse, à éviter
Rationnel g= 20/6;                      // écriture trompeuse, à éviter
```

Remarque 1.22. Dans la version 0.5 on a opté pour une restriction d'accès. Précédemment l'utilisateur pouvait créer des fractions non normalisées, par exemple en affectant `a.numer=4; a.denom=6;` car ces éléments étaient publics. Nul ne l'obligeait d'appeler la normalisation.

Désormais la classe contrôle toute écriture et toute lecture de ses éléments. Pour ce faire les éléments concernés sont déclarés *privés* : ils ne sont plus accessibles de l'extérieur de la classe. Pour lire le numérateur et le dénominateur on se sert de deux fonctions `numerateur()` et `denominateur()` qui réalisent la lecture (et la lecture seule). L'affectation via la fonction `affecter` reste en place comme avant. Comme elle appelle systématiquement la normalisation, on garantit ainsi que toute fraction soit stockée sous forme normale.

Programme XII.6 Implémentation de la classe `Rationnel` – version 0.5 `rationnel0.cc`

```
#include <iostream>
#include "integer.cc"
using namespace std;

class Rationnel
{
private:
  Integer numer, denom;   // numérateur et dénominateur
  void normaliser(void)   // normaliser la fraction numer/denom
    { if ( zero(denom) ) { cerr << "Division par zéro !\n"; exit(1); };
      Integer d= pgcd(numer,denom); numer/= d; denom/= d;
      if ( denom<0 ) { numer= -numer; denom= -denom; }; };

public:
  // Constructeurs divers
  Rationnel( int num=0, int den=1 )
    : numer(num), denom(den) { normaliser(); };
  Rationnel( const Integer& num, const Integer& den=1 )
    : numer(num), denom(den) { normaliser(); };
  Rationnel( const Rationnel& a )
    : numer(a.numer), denom(a.denom) {};

  // Lire le numérateur et le dénominateur
  const Integer& numerateur() const { return numer; };
  const Integer& denominateur() const { return denom; };

  // Affectation
  void affecter( const Integer& num=0, const Integer& den=1 )
    { numer= num; denom= den; normaliser(); };
  Rationnel& operator= ( const Rationnel& a )
    { numer= a.numer; denom= a.denom; return *this; };

  // Addition
  Rationnel operator+ ( const Rationnel& a ) const
    { return Rationnel( numer*a.denom + denom*a.numer, denom*a.denom ); };
  Rationnel& operator+= ( const Rationnel& a )
    { affecter( numer*a.denom + denom*a.numer, denom*a.denom ); return *this; };

  // Opposé et soustraction
  Rationnel operator- () const
    { return Rationnel( -numer, denom ); };
  Rationnel operator- ( const Rationnel& a ) const
    { return Rationnel( numer*a.denom - denom*a.numer, denom*a.denom ); };
  Rationnel& operator-= ( const Rationnel& a )
    { affecter( numer*a.denom - denom*a.numer, denom*a.denom ); return *this; };

  // Multiplication (correcte mais non optimisée)
  Rationnel operator* ( const Rationnel& a ) const
    { return Rationnel( numer*a.numer, denom*a.denom ); };
  Rationnel& operator*= ( const Rationnel& a )
    { affecter( numer*a.numer, denom*a.denom ); return *this; };

  // Division (correcte mais non optimisée)
  Rationnel operator/ ( const Rationnel& a ) const
    { return Rationnel( numer*a.denom, denom*a.numer ); };
  Rationnel& operator/= ( const Rationnel& a )
    { affecter( numer*a.denom, denom*a.numer ); return *this; };

  // Entrée et sortie
  friend istream& operator>> ( istream& in, Rationnel& a );
  friend ostream& operator<< ( ostream& out, const Rationnel& a );
};
```

1.3. Le principe de base : encapsuler données et méthodes ! À l'instar de la classe `Integer` modélisant les entiers, nous disposons désormais d'une classe `Rationnel` modélisant les nombres rationnels. Même dans ce petit exemple on reconnaît le germe de la programmation orientée objet : la définition d'une *classe* consiste en une structure de données ainsi que des méthodes opérant sur ces données. En reprenant le paradigme de N. Wirth on pourrait dire :

☞ *« classe = données + méthodes »* ✍

L'avantage d'une telle démarche est de regrouper, d'une manière logique et naturelle, tous ce qu'il faut pour modéliser les objets en question : en l'occurrence les nombres rationnels avec leurs opérations naturelles. Ce point de vue est tellement utile, voire nécessaire, pour organiser de projets importants, que l'on en fait un principe de programmation :

☞ *Tout objet gère ses données de manière autonome, sans intervention extérieure directe.* ✍

La communication avec le monde extérieur se fait uniquement via certaines interfaces bien choisies. Elles constituent la face visible des objets, alors que la représentation interne des données reste une affaire privée. Ce principe *soulage* l'utilisateur de la responsabilité de connaître les détails de l'implémentation, et il *protège* les objets de toute manipulation nuisible.

En suivant ce principe, une classe bien construite sera facile à maintenir, à élargir, et à réutiliser dans d'autres contextes. Ces avantages réduisent considérablement le coût du développement et évitent la réécriture inutile. C'est là le succès de la programmation orientée objet. Nous en avons déjà pleinement profité en utilisant la classe `Integer`.

Exercice/P 1.23. Comme vous pouvez le constater, les opérations d'entrée-sortie ne sont pas encore implémentées pour la classe `Rationnel`, version 0.5. La sortie est plutôt évidente, par exemple $\frac{2}{3}$ s'écrit `2/3`, alors que $\frac{2}{1}$ s'écrit `2`. L'entrée est un peu plus délicate, car la syntaxe doit être analysée afin de tenir compte des deux variantes précédentes. Vous trouvez l'implémentation complète dans le fichier `rationnel.cc`. Essayez d'en comprendre les détails.

Remarque 1.24. Les opérateurs d'entrée-sortie ne sont pas des méthodes de la classe, mais des fonctions extérieures. Néanmoins la classe `Rationnel` peut leur accorder un accès direct aux éléments privés en les déclarant `friend`. C'est souvent utile, mais ici on aurait facilement pu l'éviter. Voyez-vous comment ?

Exercice/P 1.25. Notre implémentation modélise la structure de corps mais ne tient pas encore compte de la structure d'ordre. Si vous voulez, vous pouvez ajouter des opérateurs de comparaison. *Indication.* — Détailler d'abord comment comparer deux fractions $\frac{r}{s}$ et $\frac{u}{v}$. En quoi notre convention $s > 0$, $v > 0$ est-elle importante ? Vous pouvez en profiter car la classe garantit, par une gestion intelligente et des restrictions d'accès, que cette convention soit toujours respectée.

Remarque 1.26. Pour l'implémentation de la comparaison et de toute autre fonction, plusieurs variantes sont imaginables : comme méthode de la classe, comme fonction extérieure, ou bien comme fonction déclarée `friend`. Laquelle vous semble la mieux adaptée ici ?

Conversion implicite vs explicite. Il est souvent utile d'avoir une conversion implicite entre différents types — nous devrions dire maintenant : entre différentes classes. En l'occurrence, la conversion correspond à une convention bien connue en mathématiques : on interprète tout nombre entier $a \in \mathbb{Z}$ aussi comme l'élément $\frac{a}{1}$ dans le corps de factions $\mathbb{Q}$.

En C++ il est donc naturel de convertir `int` ou `Integer` en `Rationnel` quand le contexte le demande. A priori ce sont des types bien différents, comment le compilateur saura-t-il effectuer la conversion ? Bien sûr il ne connaît rien des corps de fractions, et pourtant... un seul indice lui suffit : le constructeur à partir d'un `int` ou d'un `Integer` servira aussi bien pour la conversion.

Exemple 1.27. Les instructions suivantes devraient avoir un sens :

```
Rationnel r(2);    // conversion explicite de int en Rationnel
Rationnel s= r*3;  // acceptable car conversion implicite
```

La manière dont nous avons implémenter l'opérateur `*` exige que le premier argument soit l'objet courant, de type `Rationnel`. Si tel est le cas, le deuxième argument peut y être convertit, et le compilateur le fait tacitement. Il interprète donc le calcul précédent comme

```
Rationnel s= r * Rationnel(3);  // ok, conversion explicite
```

Exemple 1.28. À notre grande surprise, la conversion *implicite* ne s'applique plus au calcul suivant :

```
Rationnel t= 3*r;  // impossible sans conversion explicite
```

Voyez-vous pourquoi ? Bien sûr, la conversion *explicite* fonctionne toujours :

```
Rationnel s= Rationnel(3) * r;  // ok, conversion explicite
```

Le comportement serait différent si l'on implémentait l'opérateur `*` non comme une méthode de la classe, mais comme une fonction extérieure :

```
Rationnel operator* ( const Rationnel& a, const Rationnel& b )
{ return Rationnel( a.numerateur() * b.numerateur(),
                    a.denominateur() * b.denominateur() ); };
```

Cette écriture est plus longue, mais maintenant la situation est entièrement symétrique : et `3*r` et `r*3` déclenchent implicitement une conversion via le constructeur `Rationnel(3)`.

Remarque 1.29. Souvent commode, la conversion implicite est parfois indésirable voire dangereuse, car le compilateur pourrait effectuer des conversions non souhaitées par le programmeur. Pour l'éviter, on peut restreindre un constructeur en faisant précéder le mot clé `explicit`. Ainsi il ne sera appliqué qu'à la suite d'un appel explicite, comme par exemple `Rationnel(3)`. Il est en général prudent de déclarer les constructeurs `explicit`. Afin d'améliorer le confort, en peut ensuite écrire des opérateurs mixtes entre `Rationnel` et `int` ou `Integer`, là où ils sont explicitement souhaités.

2. Anneaux effectifs

Nous essayerons d'étendre nos considérations de $\mathbb{Z}$ et $\mathbb{Q}$ à des anneaux plus généraux.

2.1. Que faut-il pour implémenter un anneau ? Soit A un anneau (commutatif unitaire). En C++ on souhaiterait disposer d'un type `A` qui modélise l'anneau A dans le sens suivant :

Représentation et entrée-sortie: Tout d'abord on veut que tout élément de l'anneau A puisse être représenté comme une valeur de type `A`, et que les opérateurs d'entrée `>>` et de sortie `<<` permettent de lire et d'écrire ces valeurs dans une écriture convenable ; ils traduisent alors entre représentation externe et représentation interne.

Constructeurs: On doit être capable de construire au moins les éléments 0 et 1 de notre anneau. Ceci peut se faire par des constructeurs `A(0)` et `A(1)`. Plus généralement, pour un entier `n` on veut que `A(n)` construise l'élément $n \cdot 1_A$ dans A. Afin d'atteindre tous les éléments de A, d'autres constructeurs seront parfois souhaitables.

Comparaison: On voudra utiliser les opérateurs de comparaison `==` et `!=` avec la syntaxe usuelle : ils prennent deux arguments de type `A` et renvoient un résultat de type `bool`. Bien sûr on exige qu'ils correspondent à la comparaison dans l'anneau A. (Les comparaisons `<`, `<=`, `>`, `>=` n'ont un sens naturel que dans un anneau ordonné.)

Affectation: On voudra utiliser l'opérateur d'affectation `=` avec la syntaxe et la sémantique usuelle : il prend deux arguments, une variable de type `A` à gauche et une valeur de type `A` à droite, puis il affecte la valeur à la variable. (C'est une opération assez triviale mais omniprésente car elle gère la copie d'objets pendant l'exécution d'un programme.)

Opérateurs arithmétiques: Les opérateurs arithmétiques `+`, `-`, `*` prennent deux arguments de type `A` et renvoient un résultat de type `A`. Quant à leur sémantique, on exige que le résultat corresponde au résultat de l'opération respective dans A.

Opérateurs mixtes: Les opérateurs `+=`, `-=`, `*=` devraient s'utiliser d'une manière naturelle, de sorte que `a+=b` équivaille à `a=a+b` etc. L'intérêt est une meilleure lisibilité, et dans certains cas une légère optimisation de performance.

Inversibilité et divisibilité: Étant donnés deux éléments $a, b \in A$ on doit souvent répondre aux questions suivantes : Déterminer si a est inversible, et le cas échéant trouver son inverse. Déterminer si a et divisible par b, et le cas échéant trouver c tel que $a = bc$. Déterminer si a et b sont associés, et le cas échéant trouver $u \in A^\times$ tel que $ua = b$.

Exercice/P 2.1. Le code en C++ ci-dessous explicite les méthodes et fonctions requises pour un anneau effectif. Essayez de vous convaincre que nous avons trouvé un modèle convenable sur lequel on pourra baser toutes les constructions ultérieures : lesquelles pouvez-vous envisager ? Ceci est un point important : bien que l'implémentation concrète puisse changer, les interfaces, elles, devraient rester les mêmes. Tout changement ultérieur d'interfaces sera coûteux, car il nécessitera une révision entière des applications déjà écrites.

Programme XII.7 Modèle minimal pour implémenter un anneau `anneau0.cc`

```
 1  class Anneau
 2  {
 3  public :
 4    Anneau( int n=0 );                              // constructeur/conversion
 5    Anneau( const Anneau& source );                 // constructeur par copie
 6    Anneau& operator=  ( int n );                   // affectation/conversion
 7    Anneau& operator=  ( const Anneau& a );         // affectation par copie
 8    bool    operator== ( const Anneau& a ) const;   // test d'égalité
 9    bool    operator!= ( const Anneau& a ) const;   // test de différence
10    Anneau  operator+  ( const Anneau& a ) const;   // addition
11    Anneau& operator+= ( const Anneau& a );         // addition en place
12    Anneau  operator-  () const;                    // opposé
13    Anneau  operator-  ( const Anneau& a ) const;   // soustraction
14    Anneau& operator-= ( const Anneau& a );         // soustraction en place
15    Anneau  operator*  ( const Anneau& a ) const;   // multiplication
16    Anneau& operator*= ( const Anneau& a );         // multiplication en place
17  };
18
19  // Opérateurs d'entrée-sortie
20  ostream& operator<< ( ostream& out, const Anneau& a );
21  istream& operator>> ( istream& in, Anneau& a );
22
23  // Reconnaître les éléments 0 et 1
24  bool    zero       ( const Anneau& a );
25  bool    one        ( const Anneau& a );
26
27  // Déterminer si a est inversible, si oui calculer son inverse
28  bool    inversible ( const Anneau& a );
29  bool    inversible ( const Anneau& a, Anneau& inverse );
30
31  // Déterminer si a est divisible par b, si oui calculer le quotient q
32  bool    divisible  ( const Anneau& a, const Anneau& b );
33  bool    divisible  ( const Anneau& a, const Anneau& b, Anneau& q );
34
35  // Déterminer si a et b sont associés, si oui calculer u tel que ua=b
36  bool    associes   ( const Anneau& a, const Anneau& b );
37  bool    associes   ( const Anneau& a, const Anneau& b, Anneau& u );
38
39  // Choisir un représentant préféré r=ua parmi les éléments associés à a
40  Anneau  assopref   ( const Anneau& a );
41  Anneau  assopref   ( const Anneau& a, Anneau& u );
```

Remarque 2.2. Dans ce modèle on a inclus certaines fonctions non strictement nécessaires mais pratiques, par exemple la reconnaissance des éléments 0 et 1. On pourrait, bien sûr, écrire `a == A(0)` ou `a == A(1)`, mais il y a souvent des méthodes plus efficaces qui évitent la création d'objets auxiliaires `A(0)` et `A(1)`.

Aucune implémentation n'est fournie dans ce modèle. Pour une implémentation concrète il faut remplacer le nom `Anneau` par le nom de la classe à implémenter, puis définir chacune des fonctions déclarées ci-dessus. Ceci est fait pour la classe `Integer` dans le fichier `integer.cc`, puis pour la classe `Rationnel` dans le fichier `rationnel.cc`. Vérifier que ce sont des traductions fidèles des anneaux en question.

2.2. Vers une formulation mathématique. On dit que A est un *anneau effectif* s'il satisfait aux exigences détaillées ci-dessus : on peut représenter chacun de ses éléments dans une structure de données convenable, puis effectuer les opérations de l'anneau dans la représentation choisie. Après réflexion, le fait qu'un anneau soit effectif ne dépend pas du langage de programmation, ni de la machine qui exécutera le programme : c'est l'existence des algorithmes qui compte.

Remarque 2.3. Bien sûr, on sous-entend ici que l'on travaille sur un ordinateur « usuel ». La seule idéalisation que l'on fait habituellement est de supposer que la mémoire vive de notre ordinateur soit toujours suffisamment grande. Il est clair que notre approche pragmatique laisse tous les détails dans le flou.

Afin d'être rigoureux, on devrait à ce point expliciter le modèle de la machine qui stocke les données et exécute les algorithmes. Une approche éprouvée est de définir un ordinateur abstrait, la machine de Turing ou une de ses variantes équivalentes. On explicite ainsi la structure essentielle d'un ordinateur : tout ce que peut calculer un ordinateur « usuel » peut être calculé par une machine de Turing.

L'avantage d'une approche rigoureuse est la possibilité de prouver des énoncés négatifs : on peut montrer que certaines fonctions ne sont pas calculables. Le développement d'une telle *théorie de la calculabilité* est un important et vaste domaine, que nous n'aborderons pas ici.

Exercice/M 2.4. L'anneau $\mathbb{Z}$ des nombres entiers est effectif. Nous en avons déduit que le corps $\mathbb{Q}$ des nombres rationnels est aussi effectif. Le corps $\mathbb{R}$ des nombres réels, par contre, n'est pas effectif. (Le détailler.) Pour un point de vue plus optimiste (certes idéalisé), on consultera avec profit le livre récent de L. Blum, F. Cucker, M. Shub, S. Smale, *Complexity and real computation*, Springer Verlag, New York 1998.

Exercice/M 2.5. En reprenant notre modèle pragmatique, on veut interpréter les valeurs prises par le type `A` comme des éléments de l'anneau A. Ce n'est rien d'autre qu'une application $I\colon \{\text{valeurs du type } \texttt{A}\} \to A$. Tout d'abord on souhaite que le type `A` puisse représenter n'importe quel élément de A, autrement dit, on veut que I soit surjective. Essayez de reformuler précisément toutes nos exigences ci-dessus à l'aide de l'application I. Pourquoi exige-t-on la surjectivité de I mais on n'insiste pas sur l'injectivité ? L'opérateur `==` induit-il une relation d'équivalence ? Dans quel sens peut-on dire que I induit un isomorphisme entre le modèle informatique `A` et l'objet mathématique A ?

Exercice/M 2.6. Comme un cas simple mais déjà très intéressant regardons $d \in \mathbb{Z}$ sans facteur carré et $\mathbb{Z}[\sqrt{d}] = \{x+y\sqrt{d} \mid x,y \in \mathbb{Z}\}$. Montrer que $\mathbb{Z}[\sqrt{d}]$ est un anneau et que $(1,\sqrt{d})$ en est une $\mathbb{Z}$-base. Expliquer pourquoi $\mathbb{Z}[\sqrt{d}]$ est un anneau effectif et esquisser une implémentation. On implémentera l'anneau $\mathbb{Z}[i]$ dans le projet à la fin de ce chapitre, ce qui correspond au cas particulier $d=-1$.

Exercice/M 2.7. En généralisant l'exemple précédent on peut considérer l'anneau $\mathbb{Z}[\theta]$ où $\theta \in \mathbb{C}$ est racine d'un polynôme unitaire $P = X^n + c_{n-1}X^{n-1} + \cdots + c_1X + c_0$ à coefficients entiers. On peut supposer que P est le polynôme minimal de θ. Montrer que $\mathbb{Z}[\theta] = \{\sum_{i=0}^{n-1} z_i\theta^i | z_i \in \mathbb{Z}\}$ est un anneau et que $(1,\theta,\ldots,\theta^{n-1})$ en est une $\mathbb{Z}$-base de $\mathbb{Z}[\theta]$. Est-ce un anneau effectif ?

2.3. Anneaux euclidiens. Rappelons qu'un anneau intègre A est euclidien s'il existe un stathme $\nu\colon A \to \mathbb{N}$ et une division $\delta\colon A \times A^* \to A \times A$, $(a,b) \mapsto (q,r)$ telle que $a = bq + r$ et $\nu(r) < \nu(b)$. Au delà de l'existence nous souhaitons désormais un calcul effectif.

On dit que A est un *anneau euclidien effectif* s'il est effectif dans le sens précédent et que la division δ est effectivement calculable. En C++ nous exigeons l'implémentation suivante :

```
void   eudiv( const Anneau& a, const Anneau& b, Anneau& q, Anneau& r );
Anneau eudiv( const Anneau& a, const Anneau& b );
Anneau eumod( const Anneau& a, const Anneau& b );
```

Ici la fonction `eudiv(a,b,q,r)` implémente la division euclidienne $\delta\colon (a,b) \mapsto (q,r)$. Les deux autres fonctions renvoient q et r séparément, dans le but d'un usage plus commode. Remarquons toutefois que la fonction `eumod` devrait être définie aussi pour $b=0$, dans quel cas elle renvoie a. Par contre `eudiv` n'est définie que pour $b \neq 0$, sinon elle provoque une erreur.

Exemple 2.8. L'anneau $\mathbb{Z}$ est un anneau euclidien effectif. Il en est de même pour $\mathbb{Q}[X]$: comme on verra dans le prochain chapitre, si K est un corps effectif, alors l'anneau $K[X]$ des polynômes sur K est un anneau euclidien effectif. (Esquisser pourquoi.)

Exercice 2.9. Étant donné un anneau euclidien effectif, montrer que les algorithmes d'Euclide et de Bézout sont applicables. En particulier on sait ainsi calculer le pgcd et des coefficients de Bézout pour tout $a,b \in A$. Le programme ci-dessous présente des fonctions génériques. (Pour l'usage des fonctions génériques, ou « patrons de fonctions », voir le chapitre V, §4.)

Programme XII.8 Modèle générique de l'algorithme d'Euclide `euclide0.hh`

```
1   // L'algorithme d'Euclide pour calculer le pgcd de a et b
2   template <typename Anneau>
3   Anneau pgcd( Anneau a, Anneau b )
4   {
5     Anneau r;
6     while( !zero(b) ) { r= eumod(a,b); a= b; b= r; };
7     return assopref(a);
8   }
9
10  // L'algorithme d'Euclide étendu pour calculer pgcd(a,b) = au + bv
11  template <typename Anneau>
12  Anneau pgcd( Anneau a, Anneau b, Anneau& u, Anneau& v )
13  {
14    u= Anneau(1); v= Anneau(0);
15    Anneau x(0), y(1), q, r;
16    while( !zero(b) )
17      {
18        eudiv(a,b,q,r); a= b; b= r;
19        r= u - q * x;   u= x; x= r;
20        r= v - q * y;   v= y; y= r;
21      };
22    a= assopref(a,q); u*= q; v*= q;
23    return a;
24  }
```

2.4. Anneaux principaux. Dans un anneau principal A tout couple d'éléments $a,b \in A$ admet un pgcd et il existe $u,v \in A$ vérifiant l'identité de Bézout : $\text{pgcd}(a,b) = au + bv$.

Au delà de l'existence nous souhaitons un calcul effectif. On dit que A est un *anneau principal effectif* s'il est effectif et admet un algorithme qui calcule pour tout $a,b \in A$ leur pgcd avec des coefficients de Bézout. En C++ nous exigeons donc l'implémentation suivante :

```
Anneau pgcd( const Anneau& a, const Anneau& b, Anneau& u, Anneau& v );
Anneau pgcd( const Anneau& a, const Anneau& b, Anneau& u );
Anneau pgcd( const Anneau& a, const Anneau& b );
```

À nouveau on inclut deux fonctions spécialisées pour un usage commode. La première sert notamment à calculer l'inverse de a modulo b : si $\text{pgcd}(a,b) = 1 = au + bv$ il suffit de connaître u. La deuxième sert à calculer le pgcd sans coefficients de Bézout, ce qui arrive assez souvent. Notez qu'une fonction spécialisée est en général plus efficace, car un calcul restreint peut être optimisé.

Exemple 2.10. Tout anneau euclidien effectif est un anneau principal effectif. En particulier, ceci est le cas pour l'anneau $\mathbb{Z}$ des nombres entiers et l'anneau $K[X]$ des polynômes sur un corps K.

Exercice/M 2.11. Une application importante des anneaux principaux réside dans la résolution des équations linéaires. Expliquez comment résoudre l'équation $a_1x_1 + a_2x_2 = b$ dans un anneau principal effectif : comment déterminer si elle admet de solution ? comment en trouver une ? comment en trouver toutes ? Essayez de formuler un algorithme, puis généralisez-le à l'équation linéaire $a_1x_1 + \cdots + a_nx_n = b$.

Si vous êtes courageux, vous pouvez ensuite regarder un système d'équations linéaires, disons sous forme matricielle, et formuler l'élimination de Gauss sur un anneau principal effectif. Ce n'est pas immédiat, mais tout fonctionnera comme vous l'espérez !

Remarque 2.12. Il existe des anneaux qui sont principaux mais non euclidiens. Il n'est pas évident d'exhiber un tel exemple, mais $\mathbb{Z}[\xi]$ avec $\xi = \frac{1}{2}(1+i\sqrt{19})$ en est un. On vérifie d'abord que ξ a pour polynôme minimal $X^2 - X + 5$; le couple $(1,\xi)$ constitue donc une $\mathbb{Z}$-base de $\mathbb{Z}[\xi]$. Vous pouvez ainsi vous convaincre qu'il s'agit d'un anneau effectif ; son caractère principal mais non euclidien est moins facile à prouver.

2.5. Anneaux factoriels. Étant donné un anneau factoriel A on peut définir le pgcd et le ppcm (rappeler comment). Au delà de la définition abstraite nous souhaitons un calcul effectif. On dit que A est un *anneau à pgcd effectif* s'il est effectif et admet un algorithme pour calculer le pgcd. Pour l'implémentation en C++ on exige une fonction

```
Anneau pgcd( const Anneau& a, const Anneau& b );
```

Exemple 2.13. Tout anneau principal effectif est aussi un anneau à pgcd effectif.

Exemple 2.14. L'anneau $\mathbb{Z}[X]$ n'est pas principal : il suffit de se convaincre que l'idéal $(2,X)$, par exemple, n'est pas principal. Néanmoins, $\mathbb{Z}[X]$ est factoriel et permet un calcul effectif du pgcd.

Remarque 2.15. La factorialité de $\mathbb{Z}[X]$ est un célèbre théorème de Gauss ; le temps venu votre cours d'algèbre vous le présentera. Il affirme plus généralement que $A[X]$ est factoriel si et seulement si A est factoriel. Si vous regarder la preuve sous l'angle algorithmique, vous prouverez en même temps : si A est un anneau intègre à pgcd effectif, alors $A[X]$ est un anneau intègre à pgcd effectif. On trouve ainsi maints exemples d'anneaux factoriels qui ne sont pas euclidiens, ni principaux, mais qui permettent néanmoins de calculer effectivement le pgcd. Pour en savoir plus, on consultera avec profit le développement dans [**9**], § II-8.

Remarque 2.16. L'anneau A étant factoriel, on voudrait certes disposer des fonctions permettant de tester l'irréductibilité d'un élément $a \in A$, et le cas échéant de décomposer a en un produit d'éléments irréductibles. Malheureusement ces deux questions peuvent être assez difficiles. Pour nos besoins modestes on se contentera de calculer le pgcd, ce qui est souvent plus facile.

Exemple 2.17. On a déjà rencontré les difficultés de la factorisation dans l'anneau $\mathbb{Z}$ au chapitre XI. Dans certains anneaux la situation est encore plus compliquée, dans d'autres encore, par exemple $\mathbb{F}_q[X]$ sur un corps fini $\mathbb{F}_q$, la question d'irréductibilité et de factorisation sont beaucoup plus faciles que dans $\mathbb{Z}$. Nous ne poursuivons pas cette approche ici. Pour en savoir plus, vous pouvez consulter [**11**].

2.6. Corps des fractions. Reprenons la construction du corps des fractions. À partir de la classe `Integer` modélisant l'anneau $\mathbb{Z}$ nous avons déjà implémenté la classe `Rationnel` modélisant le corps $\mathbb{Q}$. Nous avons aussi prouvé que cette construction se généralise à n'importe quel anneau intègre.

Précisons pourtant que, pour une implémentation efficace, nous avons besoin du pgcd : on a tout intérêt à réduire systématiquement toute fraction $\frac{r}{s}$ afin d'assurer $\operatorname{pgcd}(r,s) = 1$. (Pourquoi ?) Le programme XII.9 en donne une implémentation générique.

Exercice/P 2.18. Comme vous le constatez, le programme XII.9 n'implémente pas encore d'entrée-sortie ; vous trouverez une implémentation plus complète dans le fichier `fraction.hh`. Vérifier soigneusement cette implémentation et effectuer quelques tests sur la classe `Fraction<Integer>` pour vous convaincre qu'elle modélise bien le corps des fraction $\operatorname{Frac}(\mathbb{Z}) = \mathbb{Q}$. Le programme `fraction.cc` en donne un exemple d'utilisation. Expliquez l'intérêt d'une classe générique `Fraction`. Quel pourrait être l'intérêt d'une classe spécialisée comme `Rationnel` ?

2.7. Anneaux quotients. Étant donné un anneau A implémenté par une classe `A` en C++, nous souhaitons implémenter l'anneau quotient A/I où I est un idéal de A. Ce problème général est trop dur, mais il devient facile quand nous exigeons que A soit un anneau principal effectif : dans ce cas $I = (m)$ pour un certain $m \in A$.

Dans un souci d'efficacité nous allons même supposer que A est un anneau euclidien effectif. Dans ce cas on représentera la classe $a + (m)$ par le reste $r = a \bmod m$ de la division euclidienne dans A. Le principal intérêt du passage au reste et de réduire la taille des données.

Exercice/M 2.19. Expliquer comment représenter les éléments de $A/(m)$ et comment effectuer les opérations d'anneau. Analysez ensuite l'implémentation proposée dans le fichier `quotient.hh` et vérifier soigneusement sa correction. Compléter les fonctions manquantes :

- Déterminer si a est inversible, et le cas échéant trouver son inverse.
- Déterminer si a et divisible par b, et le cas échéant trouver c tel que $a = bc$.
- Déterminer si a et b sont associés, et le cas échéant trouver $u \in A^\times$ tel que $ua = b$.
- Étant donné un élément a, choisir un élément associé préféré $r = ua$ avec $u \in A^\times$.
  (En absence de préférences on prendra $r := a$ et $u := 1$.)

Programme XII.9 Corps des fractions d'un anneau à pgcd effectif `fraction0.hh`

```
template <typename Anneau>
class Fraction
{
private:
  Anneau numer, denom;   // numérateur et dénominateur
  void normaliser(void)  // normaliser numer/denom
    { if ( zero(denom) ) { cerr << "Division par zéro !\n"; exit(1); };
      Anneau d= pgcd(numer,denom);
      divisible(numer,d,numer); divisible(denom,d,denom);
      Anneau u; denom= assopref(denom,u); numer*= u; };

public:
  // Constructeurs divers
  Fraction( int num=0, int den=1 )
    : numer(num), denom(den) { normaliser(); }
  Fraction( const Anneau& num, const Anneau& den )
    : numer(num), denom(den) { normaliser(); };
  Fraction( const Fraction& a )
    : numer(a.numer), denom(a.denom) {};

  // Affectation
  void affecter( int num=0, int den=1 )
    { numer= num; denom= den; normaliser(); };
  void affecter( const Anneau& num, const Anneau& den )
    { numer= num; denom= den; normaliser(); };
  Fraction& operator= ( const Fraction& a )
    { numer= a.numer; denom= a.denom; return *this; };

  // Lire le numérateur et le dénominateur
  const Anneau& numerateur() const { return numer; };
  const Anneau& denominateur() const { return denom; };

  // Comparaison
  bool operator== ( const Fraction& a ) const
    { return ( numer*a.denom == denom*a.numer ); };
  bool operator!= ( const Fraction& a ) const
    { return ( numer*a.denom != denom*a.numer ); };

  // Addition
  Fraction operator+ ( const Fraction& a ) const
    { return Fraction( numer*a.denom + denom*a.numer, denom*a.denom ); };
  Fraction& operator+= ( const Fraction& a )
    { affecter( numer*a.denom + denom*a.numer, denom*a.denom ); return *this; };

  // Opposé et soustraction
  Fraction operator- () const
    { return Fraction( -numer, denom ); };
  Fraction operator- ( const Fraction& a ) const
    { return Fraction( numer*a.denom - denom*a.numer, denom*a.denom ); };
  Fraction& operator-= ( const Fraction& a )
    { affecter( numer*a.denom - denom*a.numer, denom*a.denom ); return *this; };

  // Multiplication (correcte mais non optimisée)
  Fraction operator* ( const Fraction& a ) const
    { return Fraction( numer*a.numer, denom*a.denom ); };
  Fraction& operator*= ( const Fraction& a )
    { affecter( numer*a.numer, denom*a.denom ); return *this; };

  // Division (correcte mais non optimisée)
  Fraction operator/ ( const Fraction& a ) const
    { return Fraction( numer*a.denom, denom*a.numer ); };
  Fraction& operator/= ( const Fraction& a )
    { affecter( numer*a.denom, denom*a.numer ); return *this; };
};
```

Programme XII.10 Quotient d'un anneau euclidien effectif `quotient0.hh`

```
template <typename Anneau>
class Quotient
{
private:
  Anneau mod, rep; // le module et le représentant

public:
  // Constructeurs
  Quotient( const Anneau& r=Anneau(0), const Anneau& m=Anneau(0) )
    : mod( assopref(m) ), rep( eumod( r, mod ) ) {};
  Quotient( const Quotient& a )
    : mod(a.mod), rep(a.rep) {};

  // Affectation
  void affecter( const Anneau& r=0, const Anneau& m=0 )
    { mod= assopref(m); rep= eumod( r, mod ); };
  Quotient& operator= ( const Quotient& a )
    { mod= a.mod; rep= a.rep; return *this; };

  // Lire le module et le représentant
  const Anneau& module() const { return mod; };
  const Anneau& representant() const { return rep; };

  // Comparaison
  bool operator== ( const Quotient& a ) const
    { return associes( mod, a.mod ) && divisible( rep-a.rep, mod ); };
  bool operator!= ( const Quotient& a ) const
    { return !associes( mod, a.mod ) || !divisible( rep-a.rep, mod ); };

  // Addition
  Quotient operator+ ( const Quotient& a ) const
    { return Quotient( rep + a.rep, pgcd( mod, a.mod ) ); };
  Quotient& operator+= ( const Quotient& a )
    { affecter( rep + a.rep, pgcd( mod, a.mod ) ); return *this; };

  // Opposé et soustraction
  Quotient operator- () const
    { return Quotient( -rep, mod ); };
  Quotient operator- ( const Quotient& a ) const
    { return Quotient( rep - a.rep, pgcd( mod, a.mod ) ); };
  Quotient& operator-= ( const Quotient& a )
    { affecter( rep - a.rep, pgcd( mod, a.mod ) ); return *this; };

  // Multiplication
  Quotient operator* ( const Quotient& a ) const
    { return Quotient( rep * a.rep, pgcd( mod, a.mod ) ); };
  Quotient& operator*= ( const Quotient& a )
    { affecter( rep * a.rep, pgcd( mod, a.mod ) ); return *this; };
};
```

Remarque 2.20. La classe `Quotient` stocke tout élément $a+(m)$ du quotient $A/(m)$ par le représentant $r = a \bmod m$. Soulignons à titre d'avertissement que le passage au reste $r = a \bmod m$ n'est en général pas équivalent à la projection $\pi\colon A \to A/(m)$, car on ne peut pas garantir l'unicité du reste de la division euclidienne : il peut y avoir $a \equiv a' \pmod m$ avec $r = a \bmod m$ différent de $r' = a' \bmod m$. Il est donc prudent de tester l'égalité entre $r+(m)$ et $r'+(m)$ non par $r = r'$, mais par la divisibilité $m \mid r - r'$. Le programme XII.10 tient compte de cette subtilité.

Exercice/M 2.21. Montrer pour l'anneau $A = K[X]$ des polynômes sur un corps K, que la projection $\pi\colon A \to A/(m)$ est équivalente au passage au reste $a \mapsto a \bmod m$: ceci est possible parce qu'il n'existe qu'un seul représentant r avec $\deg(r) < \deg(m)$. Détailler un contre-exemple dans $\mathbb{Z}$ en explicitant une division euclidienne convenable, $a = bq + r$ telle que $|r| < |b|$. Discuter en particulier l'opérateur `%` du C++, qui est souvent une source d'erreurs. (Comment peut-on rendre le reste unique dans ce cas concret ?)

Remarque 2.22. Dans l'implémentation précédente, tout objet stocke son propre module `mod` ainsi qu'un représentant `rep` de la classe modulo `mod`. Très souvent on ne calcule que modulo un seul idéal (m) de A, il est donc inutile de stocker la même valeur m pour chacun des objets — quel gaspillage de ressources !

Le programme XII.11 ci-dessous montre comment implémenter un module *commun* pour tous les objets de la classe `Quot<Anneau>`. En C++ un tel élément se dit `static` et se comporte comme une variable globale, seul le nom `Quot<Anneau>::mod` rappelle qu'il appartient à la classe `Quot<Anneau>`. Alors que l'élément `rep` est une variable individuelle de chaque objet, l'élément `mod` n'existe qu'en un seul exemplaire, ce qui est le comportement souhaité. Pour le manipuler on implémente deux fonctions, également déclarées `static`, ce qui permet un accès via la classe, indépendemment de tout objet.

Question 2.23. En quoi est-ce périlleux de changer le module en cours du calcul ? Sous quelle condition les valeurs gardent-elles un sens après changement de module ? (Quand est-ce une opération « bien définie » ?)

Programme XII.11 Quotient d'un anneau (à compléter en exercice) `quot.hh`

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

// Définition d'une classe générique modélisant un anneau quotient
template <typename Anneau>
class Quot
{
private:
  static Anneau mod;  // le module commun de tous les objets de la classe
  Anneau rep;         // le représentant particulier de l'objet courant

public:
  // Lire et redéfinir le module commun
  static const Anneau& module() { return mod; };
  static void module( const Anneau& m ) { mod= assopref(m); };

  // Lire et réduire le représentant modulo mod
  void reduire() { rep= eumod( rep, mod ); };
  const Anneau& representant() const { return rep; };

  // Constructeurs
  Quot( const Quot& a ) : rep( a.rep ) {};
  Quot( const Anneau& r=Anneau(0) ) : rep( eumod( r, mod ) ) {};

  // S'ajoutent ici d'autres méthodes encore à implémenter ...
};

// On définit ensuite la variable statique de la classe Quot<Anneau>
template <typename Anneau>
Anneau Quot<Anneau>::mod(0);

// Exemple d'utilisation
#include "integer.cc"
int main()
{
  Quot<Integer> a(11), b(12);
  cout << "a = " << a.representant() << " mod " << a.module() << endl;
  cout << "b = " << b.representant() << " mod " << b.module() << endl;
  Quot<Integer>::module(5);
  cout << "a = " << a.representant() << " mod " << a.module() << endl;
  cout << "b = " << b.representant() << " mod " << b.module() << endl;
  Quot<Integer> c= a+b;
  cout << "c = " << c.representant() << " mod " << c.module() << endl;
}
```

Exercice/P 2.24. Compléter la classe `Quot<Anneau>` conformément au modèle XII.7.

One Ring to rule them all,
One Ring to find them.
J.R.R. Tolkien, *The Lord of the Rings*

PROJET XII

Entiers de Gauss et sommes de deux carrés

Objectifs

- Étudier l'anneau $\mathbb{Z}[i]$ et expliciter son caractère euclidien.
- En déduire une application classique aux sommes de deux carrés.

Une question classique de la théorie des nombres est la suivante : quels entiers peuvent être écrits comme somme de deux carrés ? Voici un premier constat :

$0 = 0^2 + 0^2$	$1 = 1^2 + 0^2$	$2 = 1^2 + 1^2$	$3 = ?$
$4 = 2^2 + 0^2$	$5 = 2^2 + 1^2$	$6 = ?$	$7 = ?$
$8 = 2^2 + 2^2$	$9 = 3^2 + 0^2$	$10 = 3^2 + 1^2$	$11 = ?$
$12 = ?$	$13 = 3^2 + 2^2$	$14 = ?$	$15 = ?$
$16 = 4^2 + 0^2$	$17 = 4^2 + 1^2$	$18 = 3^2 + 3^2$	$19 = ?$
$20 = 4^2 + 2^2$	$21 = ?$	$22 = ?$	$23 = ?$

Ce projet a pour but de résoudre ce problème. On considère d'abord le point de vue théorique : ici l'anneau $\mathbb{Z}[i] \subset \mathbb{C}$ des entiers de Gauss nous rendra d'excellents services. Ensuite vient l'aspect algorithmique : en sachant que $10^{100} + 949$ admet une unique décomposition en somme de deux carrés, comment la trouver de manière efficace ? C'est le caractère euclidien de $\mathbb{Z}[i]$ qui se révélera un merveilleux outil.

Sommaire

1. Analyse mathématique du problème

Exercice/P 1.1. Écrire une fonction qui prend comme argument un entier n et renvoie tous les couples $(x,y) \in \mathbb{Z}^2$ tels que $x^2 + y^2 = n$ et $x \geq y \geq 0$. Pour l'instant une méthode exhaustive suffira, néanmoins on effectuera l'optimisation évidente moyennant la fonction `sqrt`. Combien d'itérations sont nécessaires ? Est-ce raisonnable pour $10^6 + 33$? pour $10^{12} + 61$? pour $10^{50} + 577$? pour $10^{100} + 949$? Quel est le plus petit n admettant deux telles sommes ? trois ? quatre ? cinq ?

Exercice/M 1.2. L'expérience montre que beaucoup d'entiers s'écrivent comme sommes de deux carrés. On constate aussi que $4k+3$ n'est jamais une somme de deux carrés. Donner une preuve.

Pour simplifier on ne regardera dans la suite que les nombres premiers. Le but de ce projet et de montrer le théorème suivant et en même temps de développer un algorithme efficace.

Théorème 1.3. *Tout nombre premier $p = 4k+1$ s'écrit de manière unique comme somme de deux carrés, $p = x^2 + y^2$ avec $x > y > 0$ dans $\mathbb{Z}$.*

Le développement qui suit consiste en deux parties. La première donne une preuve du théorème en étudiant l'anneau $\mathbb{Z}[i]$ des entiers de Gauss. La deuxième est consacrée à l'implémentation d'une classe `Gauss` et d'une méthode efficace pour trouver la décomposition $p = x^2 + y^2$.

1.1. Unicité. Nous commençons notre étude de la décomposition $p = x^2 + y^2$ par l'unicité : si p est premier, alors $p = x^2 + y^2 = u^2 + v^2$ entraîne $\{x^2, y^2\} = \{u^2, v^2\}$.

Exercice/M 1.4. La norme $N\colon \mathbb{Z}[i] \to \mathbb{N}$ est définie par $N(z) = z\bar{z}$, ou encore $N(x+yi) = x^2 + y^2$ pour $x, y \in \mathbb{Z}$. Montrer que la norme est multiplicative : $N(ab) = N(a)N(b)$ pour tout $a, b \in \mathbb{Z}[i]$. En déduire que $a \in \mathbb{Z}[i]$ est inversible si et seulement si $N(a) = 1$, ce qui équivaut à $a \in \{\pm 1, \pm i\}$. Montrer que a est irréductible dans $\mathbb{Z}[i]$ si $N(a)$ est irréductible dans $\mathbb{Z}$. *Attention.* — la réciproque est fausse.

Exercice/M 1.5. Montrer que $\mathbb{Z}[i]$ est euclidien par rapport à la norme N, donc principal, donc factoriel. *Indication.* — Regarder l'anneau $\mathbb{Z}[i]$ et son corps des fractions $\mathbb{Q}[i]$ dans le plan complexe $\mathbb{C}$. Pour a et $b \neq 0$ dans $\mathbb{Z}[i]$ approcher leur quotient $\frac{a}{b} \in \mathbb{Q}[i]$ de manière optimale par $q \in \mathbb{Z}[i]$. (Faites un dessin !) Conclure que $a = bq + r$ avec un reste r vérifiant $N(r) \leq \frac{1}{2}N(b)$.

Exercice/M 1.6. Pour $p \in \mathbb{N}$ premier montrer que $p = x^2 + y^2 = u^2 + v^2$ avec $x, y, u, v \in \mathbb{Z}$ implique $\{x^2, y^2\} = \{u^2, v^2\}$. *Indication.* — Regarder les décompositions $p = (x+yi)(x-yi) = (u+vi)(u-vi)$.

1.2. Existence. Après l'unicité montrons l'existence d'une décomposition $p = x^2 + y^2$.

Exercice/M 1.7. Soit $p \in \mathbb{N}$ premier. Montrer que -1 admet une racine carrée modulo p si et seulement si p est de la forme $p = 4k+1$ ou $p = 2$. Dans ce cas il existe donc $q \in \mathbb{Z}$ tel que p divise $q^2 + 1$. En déduire que p n'est pas premier dans $\mathbb{Z}[i]$. *Indication.* — regarder le produit $q^2 + 1 = (q+i)(q-i)$ dans $\mathbb{Z}[i]$.

Exercice/M 1.8. En supposant que p est premier dans $\mathbb{Z}$ mais non dans $\mathbb{Z}[i]$, on sait qu'il s'écrit comme $p = ab$ avec deux facteurs $a, b \in \mathbb{Z}[i]$ non inversibles. En déduire que $N(a) = N(b) = p$, puis $\bar{a} = b$, et terminer la preuve du théorème 1.3.

Exercice/M 1.9. Reste la question pratique : comment calculer la décomposition $p = ab$ dans $\mathbb{Z}[i]$? Montrer que, quitte à échanger a et b, on a $a \sim \operatorname{pgcd}(p, q+i)$ et $b \sim \operatorname{pgcd}(p, q-i)$. Expliquer l'intérêt pratique.

1.3. Cas général. On vient de prouver qu'un nombre $p = 4k+1$ qui est premier dans $\mathbb{Z}$ devient réductible dans $\mathbb{Z}[i]$. On peut se poser la question de savoir ce qui se passe avec les autres nombres premiers, ceux de la forme $p = 4k+3$. Plus généralement, quels sont les éléments irréductibles dans $\mathbb{Z}[i]$?

Exercice/M 1.10. Commencez par montrer que les éléments suivants sont irréductibles dans $\mathbb{Z}[i]$:

- $x+yi$ avec $p = x^2 + y^2 \in \mathbb{N}$ un nombre premier de la forme $p = 4k+1$ ou $p = 2$.
- p avec $p \in \mathbb{N}$ un nombre premier de la forme $p = 4k+3$.

À noter que 2 se décompose dans $\mathbb{Z}[i]$ en deux facteurs irréductibles associés, $(1+i)$ et $(1-i)$. Un nombre premier $p = 4k+1$ se décompose dans $\mathbb{Z}[i]$ en deux facteurs irréductibles conjugués, $x+yi$ et $x-yi$, qui ne sont pas associés dans $\mathbb{Z}[i]$. Un nombre premier $p = 4k+3$ reste irréductible dans $\mathbb{Z}[i]$.

Exercice/M 1.11. Montrer que tout élément irréductible dans $\mathbb{Z}[i]$ est associé à un des précédents. *Indication.* — Étant donné $z \in \mathbb{Z}[i]$ décomposer $N(z) = z\bar{z}$ en éléments irréductibles, d'abord dans $\mathbb{Z}$, puis dans $\mathbb{Z}[i]$.

Exercice/M 1.12. Après avoir résolu la question pour les nombres premiers, caractériser les nombres naturels s'écrivant comme somme de deux carrés. Que peut-on dire du nombre de telles décompositions ? Votre résultat correspond-il aux observations empiriques de l'exercice 1.1 ?

2. Implémentation de la classe `Gauss`

Afin de calculer commodément dans $\mathbb{Z}[i]$ nous allons implémenter une classe `Gauss` modélisant cet anneau. Pour faciliter cette tâche, le programme XII.12 ci-dessous déclare les méthodes de base nécessaires ainsi que quelques fonctions supplémentaires.

Exercice/P 2.1. Relire notre modèle d'un anneau effectif (§2) puis implémenter la classe `Gauss` comme déclarée dans le programme XII.12. Vous pouvez prendre les fichiers `integer.cc` et `rationnel.cc` comme modèle, mais veillez à implémenter correctement les opérations de $\mathbb{Z}[i]$. *Remarque.* — La fonction `assopref` sert à rendre le pgcd unique. Pour les entiers de Gauss il n'y a pas de choix canonique. Pour un élément non nul on pourrait choisir l'associé $x+yi$ avec $x > 0$ et $y \geq 0$.

Conseil. — Comme toujours, testez puis relisez soigneusement votre implémentation. Pour les opérations de base choisissez des méthodes simples mais efficaces. Par exemple, tester si $x+yi$ est inversible dans $\mathbb{Z}[i]$ en calculant $x^2 + y^2$ est inefficace pour x, y grands ; essayez de faire mieux.

Programme XII.12 Déclaration de la classe `Gauss` (à compléter en exercice) `gauss.cc`

```
#include <iostream>
#include "integer.cc"
using namespace std;

// La classe Gauss modélisant l'anneau Z[i]
class Gauss
{
public:
  Integer re, im;
  Gauss( const Integer& x=0, const Integer& y=0 );
  Gauss( const Gauss& a );
  Gauss& operator=  ( const Gauss& a );
  bool   operator== ( const Gauss& a ) const;
  bool   operator!= ( const Gauss& a ) const;
  Gauss  operator+  ( const Gauss& a ) const;
  Gauss& operator+= ( const Gauss& a );
  Gauss  operator-  () const;
  Gauss  operator-  ( const Gauss& a ) const;
  Gauss& operator-= ( const Gauss& a );
  Gauss  operator*  ( const Gauss& a ) const;
  Gauss& operator*= ( const Gauss& a );
};

// Opérateurs d'entrée-sortie
ostream& operator<< ( ostream& out, const Gauss& a );
istream& operator>> ( istream& in, Gauss& a );

// Quelques fonctions supplémentaires
bool    zero ( const Gauss& a );
bool    one  ( const Gauss& a );
Gauss   conj ( const Gauss& a );
Integer norme( const Gauss& a );

// Inversibilité, divisibilité, éléments associés
bool  inversible( const Gauss& a );
bool  inversible( const Gauss& a, Gauss& b );
bool  divisible ( const Gauss& a, const Gauss& b );
bool  divisible ( const Gauss& a, const Gauss& b, Gauss& q );
bool  associes  ( const Gauss& a, const Gauss& b );
bool  associes  ( const Gauss& a, const Gauss& b, Gauss& u );
Gauss assopref  ( const Gauss& a );
Gauss assopref  ( const Gauss& a, Gauss& u );

// Une division euclidienne adaptée à la norme
void  eudiv( const Gauss& a, const Gauss& b, Gauss& q, Gauss& r );
Gauss eudiv( const Gauss& a, const Gauss& b );
Gauss eumod( const Gauss& a, const Gauss& b );

// L'algorithme d'Euclide-Bézout
Gauss pgcd( Gauss a, Gauss b, Gauss& u, Gauss& v );
Gauss pgcd( Gauss a, Gauss b, Gauss& u );
Gauss pgcd( Gauss a, Gauss b );
```

Exercice/P 2.2. Votre classe `Gauss` devrait se prêter aisément à la construction du corps des fractions $\mathrm{Frac}(\mathbb{Z}[i]) \cong \mathbb{Q}[i]$. Écrire un petit programme qui teste la classe `Fraction<Gauss>`. Votre classe `Gauss` est-elle conforme aux exigences ? La classe `Fraction<Gauss>` fonctionne-t-elle comme prévu ?

Exercice/P 2.3. Écrire un petit programme pour tester la classe `Quotient<Gauss>` ou `Quot<Gauss>` qui modélise les anneaux quotients de $\mathbb{Z}[i]$. Le quotient $Q = \mathbb{Z}[i]/(3)$ est-il un corps ? Quel est son cardinal ? Trouver un élément d'ordre 8 dans $Q^\times$. Généraliser cette approche à un nombre premier $p = 4k+3$ quelconque, puis écrire un programme qui trouve un élément d'ordre p^2-1 dans $(\mathbb{Z}[i]/(p))^\times$.

3. Décomposition en somme de deux carrés

Pour mener ce projet à bonne fin, il nous faut encore une méthode pour trouver $q \in \mathbb{Z}$ tel que p divise q^2+1. Ce problème a été résolu au chapitre X, §2.1.

Exercice 3.1. Écrire un programme qui lit un premier $p \equiv 1 \pmod 4$ au clavier, puis trouve $q \in \mathbb{Z}$ vérifiant $p \mid q^2+1$ et calcule $a = \operatorname{pgcd}(p, q+i)$ dans $\mathbb{Z}[i]$. En déduire la décomposition cherchée $p = x^2+y^2$. Ajouter une vérification du résultat et le tester sur suffisamment de petits exemples.

Exemple. — Trouver la décomposition en somme de deux carrés de $p = 1009$, puis 10^6+33 et 10^9+9 et $10^{12}+61$. Comparez les résultats et la vitesse de ce programme avec celui de l'exercice 1.1. Ajuster le programme pour que l'on puisse entrer p sous la forme $p = 10^e + k$. Le tester sur les nombres $10^{50}+577$ et $10^{100}+949$ et $10^{200}+357$ et $10^{500}+961$. Que peut-on dire de la performance ? Ces résultats auraient-ils été accessibles avec une recherche exhaustive ?

Exercice 3.2. On sait maintenant traiter efficacement les nombres premiers $p \in \mathbb{Z}$. Esquisser comment implémenter le cas général afin de décomposer un entier quelconque $n \in \mathbb{Z}$ en sommes de deux carrés.

4. Une preuve d'existence non constructive

Nous redémontrons ici le théorème 1.3 de manière *élémentaire*, en développant une preuve élégante due à Don Zagier. Sa note *A one-sentence proof that every prime* $p \equiv 1 \pmod 4$ *is a sum of two squares* peut être admirée dans American Mathematical Monthly 77 (1990), page 144. Dans un premier temps le but sera de comprendre la preuve ; ensuite on comprendra peut-être mieux la différence entre une preuve d'existence et une preuve constructive.

L'idée. On considère l'ensemble $S = \{(x,y,z) \in \mathbb{N}^3 \mid x^2+4yz = p\}$ avec l'involution $\tau\colon S \to S$ donnée par $\tau(x,y,z) = (x,z,y)$. L'idée est simple et géniale : tout point fixe de τ est de la forme (x,y,y) et nous fournit une solution $x^2+(2y)^2 = p$ comme souhaité. Pour montrer qu'un tel point existe il suffit de montrer que le cardinal de S est impair. (Pourquoi ?)

La preuve. On vérifie d'abord que S est un ensemble fini. On le partitionne en trois parties S_1, S_2, S_3 sur lesquelles on définit des applications affines $\sigma_i\colon S_i \to \mathbb{N}^3$ comme suit :

$$\begin{aligned} S_1 &= \{(x,y,z) \in S \mid x < y-z\}, & \sigma_1(x,y,z) &= (x+2z, z, y-x-z) \\ S_2 &= \{(x,y,z) \in S \mid y-z < x < 2y\}, & \sigma_2(x,y,z) &= (2y-x, y, x-y+z) \\ S_3 &= \{(x,y,z) \in S \mid 2y < x\}, & \sigma_3(x,y,z) &= (x-2y, x-y+z, y) \end{aligned}$$

Ensuite on vérifie que $\sigma_1(S_1) \subset S_3$ et $\sigma_3(S_3) \subset S_1$, ainsi que $\sigma_2(S_2) \subset S_2$. Les applications $\sigma_1\colon S_1 \to S_3$ et $\sigma_3\colon S_3 \to S_1$ sont inverses l'une à l'autre, et l'application $\sigma_2\colon S_2 \to S_2$ vérifie $\sigma_2^2 = \mathrm{id}$. Finalement, on vérifie que $S = S_1 \cup S_2 \cup S_3$ est une réunion disjointe, comme souhaité. En mettant tout ensemble, on obtient ainsi une involution $\sigma\colon S \to S$ définie par morceaux : $\sigma(x,y,z) = \sigma_i(x,y,z)$ pour $(x,y,z) \in S_i$.

La conclusion. Forcément tout point fixe $(x,y,z) = \sigma(x,y,z)$ est dans S_2. Dans ce cas $(x,y,z) = (2y-x, y, x-y+z)$ implique $x = y$. Par définition de S, un tel point (x,x,z) vérifie $x(x+4z) = p$, ce qui implique $x = 1$ et $z = \frac{p-1}{4}$. Autrement dit $(1,1,\frac{p-1}{4})$ est l'unique point fixe de σ. Comme σ est une involution, le cardinal $|S|$ doit être impair. On conclut que l'involution τ, elle aussi, admet un point fixe. Il existe donc $x,y \in \mathbb{N}$ tels que $x^2+(2y)^2 = p$.

Question 4.1. Nous avons vu deux démonstrations différentes du théorème $p = x^2+y^2$. En quoi la preuve de Zagier est-elle plus élémentaire ou plus élégante que la preuve développée dans le projet ci-dessus ? La preuve de Zagier nous indique-t-elle comment trouver (x,y) effectivement ? Résumer les avantages et les inconvénients des deux approches.

Remarque 4.2. La technique de prouver l'existence d'une solution par un argument de point fixe est fréquemment utilisée en mathématique, voir par exemple le théorème de point fixe de Banch et ses nombreuses applications. (Voir par exemple le chapitre XVII pour le calcul numérique.) Souvent on se contente de l'existence, et ainsi la construction ou la recherche effective ne font pas toujours partie du théorème. Par exemple le théorème de point fixe de Brouwer dit que toute application continue $f\colon [0,1]^n \to [0,1]^n$ admet au moins un point fixe, sans indiquer sa localisation. Quels sont les avantages et les inconvénients de ces deux théorèmes ?