Programmation avancée en C++: rappel POO et classes

Emmanuel Franck

Bureau 225, UFR math info
mail: emmanuel.franck@inria.fr

1. Généralités

Remarque sur le TP1

- Dans le corrigé de l'exercice 3:
  - On ne connaît pas le nombre d'itérations en temps, donc la taille des tableaux à la fin de la simulation.
  - **Solution proposée**: a chaque itération on libère la mémoire et ré-alloue la nouvelle mémoire.
  - Optimal en terme de mémoire, pas en terme de coût de calcul. L'allocation mémoire a un coût non négligeable.
- Autre possibilité:
  - Allouer la mémoire au début avec un nombre maximal d'itérations très grand. Pas efficace en matière de mémoire.
  - **Compromis**: un alloue un tableau de taille m. Si le nombre d'itération dépasse m, on libère la mémoire et on ré-alloue la mémoire de taille $2m$.
  - **Structures de données adaptées**: listes etc.

Important

Le compromis entre mémoire et coût de calcul est question **importante et récurrente**. Dépend des applications.

POO: Généralités I

- La Programmation Orientée Objet est le coeur du C++ et ce qui le différencie principalement du C.

Principe

La POO est basée sur l'écriture d'objets qui peuvent être vu comme des types complexes (voiture, individu, matrice) ainsi que sur des fonctions ou opérations qui agissent sur ces objets.

- Programmation plus abstraite. On doit définir clairement ses objets en premier lieu. Exemple: "individu". Quelles données seraient contenu dans l'objet individu: "âge (double)", "sexe (char)" etc.
- **Structure d'un programme POO**: On définit les objets dont on a besoin puis les interactions entre ses objets puis le programme global utilisant les objets et interactions.
- **Applications aux mathématiques**: la description par des objets et des opérations sur les objets est très naturelle en mathématiques. Exemple: nombres complexes, matrices, vecteurs, formes géométriques....

POO: Généralités II

- **Remarque**: on peut voir les ”int”, ”double” et autres types classiques comme des objets simples.
- **Définition d'un objet**: en C++ on utilise la notion de classe.

Vocabulaire

- **Classe**: ensemble des règles définissant l'ensemble des objets. Exemple: l'ensemble des nombres pairs.
- **Objet**: (ou instance de classe) un élément de la classe (un nombre pair).

Encapsulation

En POO on essaye d'encapsuler les données. C'est-à-dire que les données d'un objet (par exemple l'âge pour l'objet ”individu”) ne sont pas accessibles directement mais uniquement par des fonctions agissant sur l'objet.

- En C++ l'encapsulation est faite par défaut. En effet, cela permet d'éviter que l'utilisateur contourne l'interface prévue. Cela peut éviter un certain nombre d'erreurs.

2. Bases: Objects, classes

Déclaration de classe

- Exemple formel de déclaration de classe.

```
1 class individu {
2   public:
3     char * prenom;
4     char * nom;
5   private:
6     char sexe;
7     int age;
8 }
```

- Les variables contenues dans l'objet sont appelées attributs de la classe.
- Une classe peut contenir tout type classique (int , double, tableau, pointeur etc) ainsi que des objets d'autres classes.

- Mot clé: class suivit du nom de la classe et des données.
- Mot clé pour les attributs:
  - "Public": ces attributs sont accessibles directement par le l'utilisateur.
  - "Private": les attributs ne sont pas accessible/modifiables directement.
  - Par défaut (si on ne spécifie rien) les membres sont privés.

Exemple: les rationnels

- On considère le corps des rationnels $\mathbb{Q}$. Un rationnel est défini par

$$x \in \mathbb{Q} \iff x = \frac{n}{d}, \text{ with } \quad n \in \mathbb{Z}, \quad d \in \mathbb{Z} \in 0$$

- On commence la classe par définir l'objet:

```
1 class rationnel {
2   private:
3   int  n;
4   int  d;
5 };
```

- Un rationnel est composé de 2 entiers: le dénominateur et le numérateur. On parle de attributs. Les attributs sont membres de la classe (pas les seuls).
- Pour un accès direct à "n/d" on peut remplacer "private" par "public".
- Déclaration d'un rationnel:

```
1 int main{
2   rationnel x;
3   return 0;
4 };
```

Constructeurs et destructeur I

- Un objet doit être absolument initialisé. Pour cela on définit un constructeur.
- Type de constructeur:
  - Constructeur par par défaut: initialiser avec une valeur par défaut l'objet.
  - Autres constructeurs qui permettent d' initialiser de différentes façons un objet.
- On continue à définir la classe:

```
1  class rationnel {
2    private:
3      int  n;
4      int  d;
5    public:
6      rationnel();
7      rationnel(int, int);
8  };
9  rationnel::rationnel (){
10   n = 0; d = 1;
11   cout<< " objet construit "<<endl;
12 };
13 rationnel::rationnel (int num , int dem){
14   n = num; d = dem;
15   cout<< " objet construit "<<endl;
16 };
```

Constructeurs et destructeur II

```
1 int main (){
2   rationnel x;
3   rationnel y(5,2);
4   return 0;
5 };
```

```
user $  objet construit
user $  objet construit
```

- **Remarques**:
  - Constructeur par défaut: Obligatoire.
  - Les constructeurs ont le même nom que la classe.
  - Le constructeur par défaut: alloue la mémoire pour *n* et *d* et crée un rationnel ”0” qui s’écrit ”0/1”.
- Si vous ne donnez pas le constructeur par défaut il sera implicitement créé par le compilateur. Ici il allouera la mémoire pour *n* et *d*.
- **Remarque**: les opérateurs ”new” appellent le constructeur par défaut.

Constructeurs et destructeur III

- **Notion importante**: le le destructeur qui libère la mémoire associée à l'objet.
- **Destructeur**: il est nommé par le nom de classe précédé d'un tilde.

```
1  class rationnel {
2    int n;
3    int d;
4  public:
5    rationnel (){
6      n = 0;  d = 1;
7      cout<< " objet construit "<<endl;
8    }
9    rationnel (int num , int dem){
10     n = num;  d = dem;
11     cout<< " objet construit "<<endl;
12   }
13   ~rationnel (){
14     cout<< " objet détruit "<<endl;
15   }
16 };
```

- Ici: destructeur vide. Implicitement il libère la mémoire pour *d* et *n*.

Constructeurs et destructeur IV

```
user $  objet construit
user $  objet construit
user $  objet détruit
user $  objet détruit
```

- Destructeur jamais explicitement appelé. Mais les 2 objets ont été détruits.
- **Conclusion** : le destructeur est appelé en fin de programme/fonction.
- Un constructeur peut être plus élaboré que juste une initialisation.

```
1 rationnel (int num , int dem){
2 if( dem == 0){
3   cout<<" un dénominateur ne peut etre nul"<<endl; exit(0);
4 }
5 n = num;  d = dem;
6   cout<< " objet construit "<<endl;
7 };
```

```
user $ objet construit
user $ un denominateur ne peut etre nul
```

Constructeur par copie I

- **Constructeur par copie**: en général on définit un constructeur qui crée un objet par copie d'un autre.
- Ce constructeur est essentiel. Si vous ne le définissez pas, il est construit par défaut par le compilateur.

```
1 rationnel (const rationnel & x){
2   n = x.n;  d = x.d;
3   cout<< " objet construit  par copie"<<endl;
4 };
```

- **Remarques**
  - *objet*.*membre* permet d'accéder à un membre de l'objet. Possible seulement dans la classe car le membre privé.
  - Le constructeur par copie peut être appelé implicitement. C'est pour cela qu'il est essentiel.
  - **Exemple Important**: a chaque passage par copie d'un objet comme paramètre d'une fonction le constructeur par copie est appelé implicitement.

Constructeur par copie III

- Exemple : argument fonction.

```
1 void plot_objet(rationnel x){
2    cout<< " objet rationnel "<<endl;
3 };
4 int main (){
5   rationnel y(5,1);
6   plot_objet(y);
7   return 0;
8 };
```

```
user $ objet construit
user $ objet construit par copie
user $ objet rationnel
user $ objet détruit
user $ objet détruit
```

- Le passage en argument (par copie) d'un objet appelle le constructeur par copie de façon implicite.

Constructeur par copie III

- Exemple : argument fonction.

```
1 void plot_objet(rationnel & x){
2    cout<< " objet rationnel "<<endl;
3 };
4 int main (){
5   rationnel y(5,1);
6   plot_objet(y);
7   return 0;
8 };
```

```
user $ objet construit
user $ objet rationnel
user $ objet détruit
```

- Ce n'est pas le cas lors d'un passage par référence.
- Nous verrons plus tard d'autres utilisations implicites du constructeur par copie.

Réécriture et Constructeur délégués

- Souvent on écrit avec un autre formalisme les constructeurs:

```
1     rationnel (): n(0), d(1) {}
2     rationnel (int num , int dem): n(num), d(dem) {}
3   }
```

- En pratique, on appelle le constructeur des entiers.
- Dans les anciennes versions du C++ un constructeur ne peut pas en appeler un autre.
- Depuis le C++11, c'est possible, on parle de constructeur délégué.

```
1     rationnel (int num , int dem): n(num), d(dem) {}
2     rationnel (): rationnel(0,0) {}
3   }
```

```
1     rationnel() {
2                 rationnel(0,0);
3     }
```

-

3. Bases: méthodes

Les méthodes I

- Les méthodes sont des membres de la classe. Il s'agit de fonctions qui permettent d'agir sur les objets de la classe.
- En général les entêtes sont définis dans la classe et le corps des méthodes en dehors.

```
1  class rationnel {
2    int n;
3    int d;
4  public:
5    ......
6   bool positif();
7   bool negatif();
8   rationnel modify_n(rationnel nn);
9   rationnel modify_d(rationnel nd);
10 };
11 bool rationnel::positif(){
12   bool res = false;
13   if( n*d > 0) { res = true; }
14   return res;
15 };
```

Les méthodes II

- **Remarque**: puisque les membres sont privés on utilise des méthodes pour accéder/modifier les objets.
- Lorsqu'on souhaite modifier un objet dans une méthode un mot-clé important: this.
- "**this**": est un pointeur qui pointe sur l'objet auquel s'applique la méthode.
- Pour accéder aux attributs de la classe:

```
1 this->x  ou  x
```

- Exemple:

```
1 rationnel rationnel::modify_n(int nn){
2   n = nn;
3   return *this;
4 };
5 rationnel rationnel::modify_d(int nd){
6   this->d = nd;
7   return *this;
8 };
```

Les méthodes III

- Pour appliquer une méthode à un objet on écrit ”objet.nomméthode”

```
1  int main (){
2    bool sign=false;
3    rationnel y(5,-1);
4    sign=y.positif();
5    cout<< ">>> signe :"<<sign<<endl;
6    y.modify_d(2);
7    sign=y.positif();
8    cout<< ">>> signe :"<<sign<<endl;
9    return 0;
10 };
```

```
user $ >>> objet construit
user $ >>> signe :0
user $ >>> objet construit par copie
user $ >>> objet détruit
user $ >>> signe :1
user $ >>> objet détruit
```

Fonctions amies I

- Parfois il est utile qu'une fonction extérieure accède à des objets d'une ou plusieurs classes. Exemple

```
1 void plot_values(rationnel x){
2   cout<< ">>> n: "<<x.n<<" d: "<<x.d<<endl;
3 };
```

```
user $ complexe.cpp:65:23: error: 'n' is a private member of 'rationnel'
user $ complexe.cpp:7:7: note: implicitly declared private here int n;
```

- **Question**: pourquoi cela ne marche pas ?

Fonctions amies I

- Parfois il est utile qu'une fonction extérieure accède à des objets d'une ou plusieurs classes. Exemple

```
1 void plot_values(rationnel x){
2   cout<< ">>> n: "<<x.n<<" d: "<<x.d<<endl;
3 };
```

```
user $ complexe.cpp:65:23: error: 'n' is a private member of 'rationnel'
user $ complexe.cpp:7:7: note: implicitly declared private here int n;
```

- **Question**: pourquoi cela ne marche pas ?
- **Solution** : les fonctions amis: il s'agit de fonctions extérieures qui ont le droit d'accéder aux membres de la classe.
- **Mot clé** : friend

Fonctions amies II

```
1  class rationnel {
2   ....
3   public:
4   ....
5   friend void plot_values(rationnel x);
6 };
7 int main (){
8   bool sign=false;
9   rationnel y(5,-1);
10  plot_values(y);
11  return 0;
12 };
```

```
user $ >>> objet construit
user $ >>> objet construit par copie
user $ >>> n: 5 d: -1
```

- **Remarques**:
  - Une méthode d'une classe peut être amis d'une autre classe.
  - Une fonction peut être amis de plusieurs classes.

Fonctions amies III

- Une classe entière peut être amie d'une autre classe.
- L'ensemble des méthodes de la classe amie a accès aux attributs de la classe.

```
1  class rationnel {
2    friend class plot_r;
3    ......
4  };
5  class plot_r {
6  public:
7    void plot(rationnel x);
8  };
9  void plot_r::plot(rationnel x){
10   cout<< ">>> plot par classe amis."<<" n: "<<x.n<<" d: "<<x.d<<endl;
11 };
```

```
user $ >>> plot par classe amis. n: 5 d: -1
```

Surcharge opérateurs I: principe

- Après avoir défini des objets comme les rationnels on a envie de les manipuler.
- **Exemple**: addition multiplication etc.
- Solutions
  - Écrire des méthodes qui font les opérations. Solution peu élégante.
  - Redéfinir les opérations pour nos objets. Surcharge opérateurs.
- On peut redéfinir les opérateurs
  - arithmétique: +,-, *, / etc,
  - de comparaison : <, >, ==, != etc,
  - d'affection = etc,
  - et d'autres encore.
- **Mot clé**: operator

Surcharge opérateurs II: opérations arithmétiques

```
1 class rationnel {
2   .....
3   rationnel operator + (rationnel);
4   rationnel operator - ( rationnel);
5   rationnel operator * ( rationnel);
6   rationnel operator / ( rationnel);
7 };
```

- Il s'agit d'opérateur binaire appliqué à un rationnel. il prend donc en argument l'autre rationnel.
- Addition:

```
1 rationnel rationnel:: operator + (rationnel x){
2   rationnel res;
3   res.d = d*x.d;
4   res.n = x.d*n + d*x.n;
5   return res; };
```

- Lorsque on utilise un membre de l'objet de gauche, on l'écrit directement. Exemple *d* et *n* sont associés à l'objet de gauche.
- Passage de paramètre par par copie.

Surcharge opérateurs II: opérations arithmétiques

```
1 class rationnel {
2   .....
3   rationnel operator + (rationnel);
4   rationnel operator - ( rationnel);
5   rationnel operator * ( rationnel);
6   rationnel operator / ( rationnel);
7 };
```

- Il s'agit d'opérateur binaire appliqué à un rationnel. Il prendra donc en argument l'autre rationnel.
- Multiplication:

```
1 rationnel rationnel:: operator + (rationnel x){
2   rationnel res;
3   res.d = d*x.d;
4   res.n = n*x.n;
5   return res; };
```

- Lorsqu'on utilise un membre de l'objet de gauche on l'écrit directement. Exemple *d* et *n* sont associés à l'objet de gauche.
- Passage de paramètre par par copie.

Surcharge opérateurs II: opérations arithmétiques

```
1 class rationnel {
2   .....
3   rationnel operator + (rationnel);
4   rationnel operator - ( rationnel);
5   rationnel operator * ( rationnel);
6   rationnel operator / ( rationnel);
7 };
```

- Il s'agit d'opérateur binaire appliqué a un rationnel. il prendre donc en argument l'autre rationnel.
- Résultat:

```
1 int main (){
2   rationnel y1(5,2), y2(4,3), y3(1,2);
3   plot_values((y1+y2)*y3);
4   return 0; };
```

```
user $ >>> n: 23 d: 12
```

- Passage de paramètre par par copie.

Surcharge opérateurs III: opérations de comparaison

- Les opérateurs de comparaison peuvent être aussi surchargés.

```
1 class rationnel {
2   .....
3   bool operator < (rationnel);
4   bool operator > (rationnel);
5   bool operator == (rationnel);
6   bool operator != (rationnel);
7 };
```

- Infériorité:

```
1 bool rationnel:: operator < (rationnel x){
2   bool res = false;
3   if(n*x.d < d*x.n) { res = true;}
4   return res;
5 };
```

- Lorsqu'on utilise un membre de l'objet de gauche on l'écrit directement. Exemple *d* et *n* sont associés à l'objet de gauche.

Surcharge opérateurs III: opérations de comparaison

- Les opérateurs de comparaison peuvent être aussi surchargés.

```
1 class rationnel {
2   .....
3   bool operator < (rationnel);
4   bool operator > (rationnel);
5   bool operator == (rationnel);
6   bool operator != (rationnel);
7 };
```

- égalité:

```
1 bool rationnel:: operator == (rationnel x){
2   bool res = false;
3   if((n/x.n) == (d/x.d)) { res = true;}
4   return res;
5 };
```

- Lorsqu'on utilise un membre de l'objet de gauche on l'écrit directement. Exemple *d* et *n* sont associés à l'objet de gauche.

Surcharge opérateurs III: opérations de comparaison

- Les opérateurs de comparaison peuvent être aussi surchargés.

```
1 class rationnel {
2   .....
3   bool operator < (rationnel);
4   bool operator > (rationnel);
5   bool operator == (rationnel);
6   bool operator != (rationnel);
7 };
```

- Résultats:

```
1  rationnel x1(1,2), x2(1,3), x3(3,6);
2   bool a,b,c;
3   a = x1 < x2 ; b = (x1 == x2); c = (x1 == x3);
4   cout<<" test x1 < x2 "<< a <<endl;
5   cout<<" test x1 == x3 "<< c <<endl;
6 };
```

```
user $ test x1 < x2 0
user $ test x1 == x3 1
```

Surcharge opérateurs IV: opérateur affectation et copie

- **Opérateur essentiel**: l'affectation "=", il permet d'affecter une valeur à un objet. Il peut aussi être redéfini.

```
1 class rationnel {
2 ......
3 public:
4 ......
5 rationnel & operator = (const rationnel &);
6 };
```

- Comme le constructeur par copie l'opérateur d'affectation ne modifie pas l'objet de droite d'où le "const".
- Comme le constructeur par défaut: si on le déclare pas le compilateur en utilise un par défaut.
- Contrairement au constructeur par copie, l'objet peut déjà exister. Par conséquent en général:
  - On libère la mémoire de l'objet à qui on va affecter une valeur (s' il y a mémoire à libérer).
  - On crée un emplacement mémoire et on recopie les valeurs qu'on veut affecter.

Surcharge opérateurs IV: opérateur affectation et copie

- Définition de l'opérateur d'affectation:

```
1 rationnel & rationnel:: operator = (const rationnel & x){
2   if(&x != this){
3     d = x.d;
4     n = x.n;
5   }
6   return *this;
7 }
```

- Ici pas de mémoire à libérer avant l'affectation (cas différent: tableau).
- Un autre problème important est celui de l'auto-affectation. Affecter un objet à lui-même peut être dangereux. En effet, l'affectation risque de détruire les données membres de l'objet avant même qu'elles ne soient copiées, ce qui provoquerait en fin de compte simplement la destruction de l'objet.
- Pour éviter cela on teste l'auto-affectation avec "if(&x != this) ".

Surcharge opérateurs V: opérateurs cout

- La surcharge des opérateurs d'entrée/sortie est aussi possible.
- Les opérateurs "entrée/sortie" sont des objets des classes de la STD:
  - ostream pour les objets de sortie,
  - istream pour les objets d'entrée.
- Puisqu'il s'agit d'un opérateur sur 2 classes il ne peut pas être membre interne de la classe sauf si il était ami de la classe ostream.
- Pas possible car on ne peut modifier la classe ostream.
- il faut donc que l'opérateur soit externe et déclarée comme amie de la classe.

```
1 class rationnel {
2 public:
3 ....
4 friend ostream & operator<<(ostream &, const rationnel &);
5 };
6 ostream & operator<<(ostream & out, const rationnel & x){
7   out<<x.n<<"/"<<x.d;
8   return out;
9 }
```

Surcharge opérateurs V: opérateurs cout

- La surcharge des opérateurs d'entrée/sortie est aussi possible.
- Les opérateurs "entrée/sortie" sont des objets des classes de la STD:
  - ostream pour les objets de sortie,
  - istream pour les objets d'entrée.
- Puisqu'il s'agit d'un opérateur sur 2 classes il ne peut pas être membre interne de la classe sauf si il était ami de la classe ostream.
- Pas possible car on ne peut modifier la classe ostream.
- il faut donc que l'opérateur soit externe et déclarée comme amie de la classe.

```
1 int main (){
2   rationnel x1(1,2), x2(1,3);
3   x2 =x1;
4   cout<<">>> print: "<<x2<<endl;
5   return 0;
6 };
```

```
user $ >>> print: 1/2
```

Surcharge opérateurs VI: opérateurs amis

- On veut parfois faire un produit entre une variable d'un type différent et un objet de notre classe.
- Pour cela on écrit un opérateur ami.

```
1 friend rationnel operator *(int, const rationnel &);
2 .....
3 rationnel operator *(int a, const rationnel & x){
4 rationnel res;
5 res.n = a*x.n;  res.d = a*x.d;
6 return res;
7 }
```

- Si on ne définit pas cet opérateur "friend". On a:

```
user $ Undefined symbols for architecture x86_64:
user $ "operator*(int,␣rationnel␣const&)", referenced from:
user $      _main in main.o
user $$ ld: symbol(s) not found for architecture x86_64
```

- Commence précédemment, la compilation indique clairement le problème.

Surcharge opérateurs VII: cast

- Comme tous les opérateurs, les opérateurs ”cast” qui permettent de convertir les types peuvent être surchargés.
- Exemple: convertir un rationnel en entier.

```
1 operator int ();
2 ....
3 rationnel:: operator int (){
4 int res;
5 res = n/d;
6 return res;}
```

- Ici il fait une troncature.
- On remarque qu’il n’y a pas de type de retour. Si on ne définit pas cet opérateur on a:

```
user $ Undefined symbols for architecture x86_64:
user $ "rationnel::operator int()", referenced from:
user $      _main in main.o
```

- Pour convertir dans l’autre sens on utilise les constructeurs.

Objet temporaire

- **Objet temporaire**: objet créé par le programme seul sans qu'on y ait accès.
  Exemple:

```
1  rationnel a;
2  a = rationnel(1,2);
```

```
user $ objet construit
user $ >>> objet construit
user $ >>> objet détruit
user $ fin programme
```

- Le constructeur par défaut construit "a" (1).
- Le constructeur crée un rationnel avec le constructeur (2).
- Ce rationnel (sans nom) est directement donné à l'opérateur d'affectation.
- l'opérateur d'affectation copie les données de l'objet temporaire dans "a".
- L'objet temporaire est immédiatement détruit (pas à la fin) (3).

4. Tests unitaires

Compilation I

```
1  CC = g++
2  CFLAGS = -Wall
3  EXEC_NAME = run
4  EXEC_NAME2 = tests
5  OBJ_FILES = main.o rationnel.o
6  TEST_FILES = tests_units.o rationnel.o
7
8  all : $(EXEC_NAME)
9
10 clean :
11         rm $(EXEC_NAME) $(OBJ_FILES)
12
13 cleantests :
14         rm $(EXEC_NAME2) tests_units.o
15
16 $(EXEC_NAME) : $(OBJ_FILES)
17         $(CC) -o $(EXEC_NAME) $(OBJ_FILES)
18
19 $(EXEC_NAME2) : $(TEST_FILES)
20         $(CC) -o $(EXEC_NAME2) $(TEST_FILES)
21
22 %.o: %.cpp %.hpp
23         $(CC) -o $@ -c $<
```

Compilation II

- **Remarques**:
  - On modifie la règle générique pour les ”.o” en incluant les ”.hpp”.
  - On construit deux exécutables: ”run” basée sur la fonction main de ”main.cpp” et un autre pour les tests unitaires.

```
user $ make
user $ c++     -c -o main.o main.cpp
user $ g++ -o rationnel.o -c rationnel.cpp
user $ g++ -o run main.o rationnel.o
```

- Pour éviter des problèmes d’inclusion on écrit tous les ”.hpp” de la façon suivante:

Rationnel.hpp

```
1 #ifndef RATIONNEL_HPP
2 #define RATIONNEL_HPP
3 #include <cmath>
4 #include <iostream>
5 using namespace std;
6 ....
7 #endif
```

Tests unitaires I

- Exemple de tests unitaires:

```
1  int rationnel::testu_1(){  // test opération
2    rationnel x(1,2);
3    rationnel y(1,3);
4    rationnel z(-2,5);
5
6    *this = x+y; // res =5/6
7    *this = *this * z; // res = -10/30
8    *this = *this /x - y;  // res = -20/30- 1/3 = -90/90
9    if(this->num() == -90 && this->dem() == 90){ return 0; }
10   else { return 1; }
11 }
```

- Fonction ”main” des tests unitaires:

```
1  int rationnel:: all_testsu(){
2    int c=0,t=0; rationnel x;
3    c=x.testu_1(); t = t +c;
4    c=x.testu_2(); t = t +c;
5    c=x.testu_3(); t = t +c;
6    cout<<"Tests unitaires réussis, rationnel: "<<3-t<<"/3"<<endl;
```

Tests unitaires I

- Exemple de tests unitaires:

```
1  int rationnel::testu_2(){  // test signe + modification
2    rationnel x(1,2);
3    bool signe1 = false, signe2 = false, signe3 = false;
4    *this = x;
5    signe1 = this->positif();
6    this->modify_n(-1.0);
7    signe2 = this->negatif();
8    this->modify_d(-2.0);
9    signe3 = this->positif();
10   if(signe1 && signe2 && signe3){ return 0; }
11   else { return 1; }
12 }
```

- Fonction ”main” des tests unitaires:

```
1  int rationnel:: all_testsu(){
2    int c=0,t=0; rationnel x;
3    c=x.testu_1(); t = t +c;
4    c=x.testu_2(); t = t +c;
5    c=x.testu_3(); t = t +c;
6    cout<<"Tests unitaires réussis, rationnel: "<<3-t<<"/3"<<endl;
```

Tests unitaires I

- Exemple de tests unitaires:

```
1  int rationnel::testu_3(){ // test comparaison
2    rationnel x1(1,2);
3    rationnel x2(1,3);
4    rationnel x3(3,6);
5    bool a,b,c;
6    *this = x1;
7    a = x2 < *this ; b = *this > x2; c = (*this == x3);
8
9    if(a && b & c){ return 0; }
10   else { return 1; }
11 }
```

- Fonction ”main” des tests unitaires:

```
1  int rationnel:: all_testsu(){
2    int c=0,t=0; rationnel x;
3    c=x.testu_1(); t = t +c;
4    c=x.testu_2(); t = t +c;
5    c=x.testu_3(); t = t +c;
6    cout<<"Tests unitaires réussis, rationnel: "<<3-t<<"/3"<<endl;
```

Tests unitaires II

- Ces tests unitaires vérifient toutes les fonctions/méthodes de la classe ”rationnel”.

```
user $ make tests
user $ c++    -c -o tests_units.o tests_units.cpp
user $ g++ -o tests tests_units.o rationnel.o
user $ ./tests
user $ Tests unitaires réussis, classe rationnel: 3/3
user $ make cleantests
user $ rm tests tests_units.o
```

- On compile les tests séparément, on les exécute et on a un ”make clean” spécifique.
- **Construction de projets**: il est utile de construire des tests unitaires pour chaque classe, pour chaque fonction et créer un exécutable qui lance tous les tests.

5. Autre exemple: les complexes

Autre exemple: Le corps des complexes

- On considère l'ensemble des nombres complexes

$$x \in \mathbb{C}, \rightarrow x = a + ib, \text{ with } a, b \in \mathbb{R}^2$$

- L'ensemble $(\mathbb{C}, +, *)$ est un corps pour "+" et "*".

```
1  #define COMPLEXE_HPP
2  #include <cmath>
3  #include <iostream>
4  using namespace std;
5  class complexe {
6    double r;
7    double i;
8  public:
9    complexe (){
10     r = 0; i = 0;
11   }
12   complexe (double reel , double img){
13     r = reel;  i = img;
14   }
15   complexe (const complexe & x){
16     r = x.r;  i = x.i;
17   }
18   ~complexe (){
19   }
```

Autre exemple: Le corps des complexes II

```
1   bool reel_pure();
2   bool img_pure();
3   complexe modify_reel(double reel);
4   complexe  modify_img(double img);
5   double  module();
6   complexe conjuge();
7   double reel();
8   double img();
9   complexe operator + (complexe);
10  complexe operator - ( complexe);
11  complexe operator * ( complexe);
12  complexe operator / ( complexe);
13  complexe & operator = (const complexe &);
14  bool operator < (complexe);
15  bool operator > (complexe);
16  bool operator == (complexe);
17  bool operator != (complexe);
18  friend ostream & operator<<(ostream &, const complexe &);
19  friend complexe operator *(double, const complexe &);
20  int testu_1();
21  int testu_2();
22  int testu_3();
23  int all_testsu();
24 };
25 #endif
```

Autre exemple: Le corps des complexes III

- Les opérateurs d'accès aux membres: "reel()", "img()", de modification, de test sur les membres ne présente pas de difficulté particulière.
- Les surcharges de "+","-","=" sont aussi naturelles.
- Multiplication: $(a + ib)(c + id) = (ab - cd) + i(ac - bd)$

```
1  double complexe::module(){
2    double res=0;
3    return sqrt(r*r+i*i);
4  }
5  complexe complexe::conjuge(){
6    complexe res;
7    res.r = r;
8    res.i = -i;
9    return res;
10 }
11 complexe complexe:: operator * (complexe x){
12   complexe res;
13   res.r = r*x.r-i*x.i;
14   res.i = r*x.i+i*x.r;
15   return res;
16 }
```

Autre exemple: Le corps des complexes IV

- **Remarques**:
  - L'opération"/", les opérateurs de comparaison peuvent utiliser une méthode.

```
1  complexe complexe :: operator / (complexe x){
2    complexe res;
3    double mod;
4    mod = this->module();
5    res.r = (r*x.r+i*x.i)/(mod*mod);
6    res.i = (i*x.r-x.i*r)/(mod*mod);
7    return res;
8  }
9  bool complexe:: operator == (complexe x){
10   bool res = false;
11   if(this->module() == x.module()) { res = true;}
12   return res; }
```

  - Opérateur externe: $\alpha * (a + ib) = (\alpha a + i(\alpha b))$.

```
1  complexe operator *(double a, const complexe & x){
2    complexe res;
3    res.r = a*x.r;  res.i = a*x.i;
4    return res;
5  }
```

Rappel vocabulaire

- Les classes sont composées de membres de classe.
- Plusieurs types de membres:
  - Les attributs: données,
  - les constructeurs/destructeurs: fonctions initialisant/détruisant les objets (ou instance de classe),
  - les méthodes: fonctions agissant sur les objets,
  - les opérateurs: fonctions particulières agissant sur les objets (+, = etc).
- **Important**: toutes les fonctions membres ont accès aux attributs. Si ils sont privés seul les fonctions membres y ont accès.

Important

- Le constructeur par défaut, par copie et le destructeur sont régulièrement appelés implicitement. Si on les donne pas le compilateur les construit par défaut.
- C'est la même chose pour la surcharge de l'opérateur "=".
- Le constructeur par copie et le "=" donnés par le compilateur copie les membres.

Conclusion

- Les classes sont un outil très puissant, très utilisé qui permet de gérer facilement/efficacement des objets et problèmes complexes.
- En mathématiques, les classes très adaptées aux structures algébriques et espaces vectoriels: Corps/anneaux algébriques, polynômes, matrices, vecteurs, séries de Fourier, objets géométriques.
- **Dangers**: il y a un risque d'en faire trop, de trop utiliser les classes. Il faut réfléchir a priori sur l'utilité de la classe et sa construction.