



# Programmation orientée objet en C++

Les classes

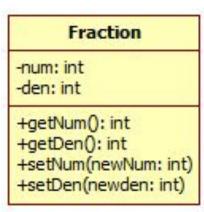




• Une classe est un ensemble d'objets qui partagent une même sémantique, des mêmes attributs et des opérations communes.

### Exemple :

Toutes les fractions ont un numérateur et un dénominateur.





## Définition d'une classe en C++

- Le mot clé class
- Les délimiteurs { } et un séparateur ; à la fin
- Une liste de membres
- Les membres d'une classes peuvent être des variables ou des fonctions, représentant les données et les opérations
- On appelle ces variables : attributs
- On appelle ces fonctions : méthodes
- Les membres (attributs et méthodes) peuvent être : private, public, ou protected

```
class Fraction
   private:
   int num;
   int den;
   public:
   int getNum();
   int getDen();
   void setNum(int newNum);
   void setDen(int newDen);
};
```





- On distingue trois types d'accesseurs : private, public, ou protected
- Les membres privées ne sont accessibles que pour la classe.
- Les membres publics sont accessibles pour toute autre classe ou fonction (Ex. dans le programme principal via la fonction main())
- Les membres protégés sont accessibles pour la classe et ses classes enfants (A détailler dans le chapitre « Héritage »)





- Le mot clé private est la clé d'un mécanisme appelé : encapsulation
- L'encapsulation sépare :
  - La structure interne de l'objet
  - De l'interface (la manière de l'utiliser)

```
class Fraction
   private:
   int num;
   int den;
   public:
   int getNum();
   int getDen();
   void setNum(int newNum);
   void setDen(int newDen);
};
```



# Objectifs de l'encapsulation

- Abstraction : l'utilisateur final n'a pas de vue sur l'implémentation des classes et leur structure interne. La manipulation se fait via les méthodes (interfaces).
- Protéger l'intégrité des objets: les méthodes peuvent vérifier que les différentes opérations effectuées préservent l'intégrité des objets.

Exemple : le dénominateur ne doit pas être nul dans une fraction. Cette vérification se fera via la méthode setDen(...).

 Robustesse du code : la classe peut être modifiée sans que son utilisateur final soit impacté.





- Les méthodes sont écrites (définies) dans des fichiers .cpp
- On les fait précéder par le nom de la classe et l'opérateur de portée :: selon la syntaxe

```
return_type ClassName::methodName(type1 arg1, ...)
{

/* function body here */
}
```





### fraction.h

```
class Fraction {
      private:
      int num;
      int den;
      public:
      int getNum();
      int getDen();
      void setNum(int);
      int setDen(int);
};
```





# fraction.cpp

```
#include "fraction.h"
#include <iostream>
using namespace std
int Fraction::getNum(){
     return num;
int Fraction::getDen(){
     return den;
void Fraction::setNum(int newnum){
     num = newNum;
```

```
int Fraction::setDen(int newden){
 if(newden == 0)
    cerr << "den is 0!"
       << endl;
    return -1;
  else
    den = newden;
    return 0;
```



# Manipulation des instances

• Les instances sont les variables de type classe.

Allocation statique:

Fraction f;

Allocation dynamique:

```
Fraction* pf = new Fraction();
```

Analogie avec les types de base

int i;

int\* pi = new int();



## L'accès aux membres de la classe

- La syntaxe est similaire à celle des structures :
- Pour les instances : instance.membre

```
Fraction f;
f.setNum(3);
```

Pour les pointeurs vers les instances : pinstance->membre

```
Fraction* pf1 = &f;
float num = pf1->getNum();

Fraction* pf2 = new Fraction();
pf2->setNum(num);
```





- 3<sup>ème</sup> usage du mot clé const
- Permet de spécifier des méthodes constantes : elles ne peuvent pas modifier les attributs de la classe (ils sont en « lecture seule »).
- Eviter au développeur de changer les attributs accidentellement.
- La syntaxe :

```
class MyClass
{
    ...
    returnType methodName(...) const
    ...
};
```

Attention :

Le mot clé const doit être présent dans la déclaration et la définition.





```
Fraction f;
f.setNum(1);
cout << f.getNum();</pre>
```

```
class Fraction
{
    ...
    int getNum() const
    {num++; return num; };
    ...
};
```

### Compiler output (g++)

```
fractionTest.cpp: In member function 'float
Fraction::getNum() const':
fractionTest.cpp:15:5: error: increment of data-member
'Fraction::num' in read-only structure
```





- Il s'agit d'un pointeur passé automatiquement à toutes les méthodes d'une classe
- Il pointe l'adresse de l'instance ayant fait l'appel

### • Exemple:

```
ThisDisplay* dp =
    new ThisDisplay();

cout << dp << endl;
dp->printThis();
```

### Standard output

```
0x8242008
0x8242008
```

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

class ThisDisplay
{
    public:
    void printThis()
    { cout << this << endl; };
};</pre>
```





 Le pointeur this permet de supprimer les conflits des identifiants entre les paramètres et les variables locales

```
void Fraction::setNum(int newnum) {
    num = newnum;
}

void Fraction::setNum(int num) {
    num = num;
}

void Fraction::setNum(int num) {
    num = num;
}

Le paramètre num reçoit sa
propre valeur

L'attribut num reçoit la valeur du
paramètre num
}
```



• Retourner la valeur de ce pointeur dans les méthodes permet de réaliser des appels chaînés (method chaining / fluent interface)

```
• Example :
```

```
Fraction* Fraction::setNum(int num) {
        this->num = num;
        return this;
}
```

### Permet d'écrire :

```
Fraction* f = new Fraction();
f->setNum(3)->setDen(2);
```