Programmation C++

Constructeurs, Destructeurs et Allocation dynamique

ING2-GSI

CY Tech

2023-2024

Constructeurs, Destructeurs

Exemple : Fraction

Fraction.hpp

```
#ifndef __FRACTION_HPP_
#define __FRACTION_HPP_

class Fraction {
private :
    int numerateur;
    int denominateur;
public :
    Fraction();
    Fraction(int n);
    Fraction(int n, int d);
    ~Fraction();
    // autres methodes
};
#endif
```

Constructeurs

- Un constructeur d'une classe est une fonction membre particulière :
  - qui a le même nom que la classe
  - qui n'a pas de type de retour (même pas void)
  - dont le rôle est **d'initialiser un objet** : il va donner des valeurs aux données membres des objets de la classe.
- Il existe plusieurs types de constructeurs : par défaut, par paramètres, par copie
- Tant qu'aucun constructeur n'est défini, le compilateur suppose que la classe a un constructeur *par défaut* implicite.

Exemple

Fraction.cpp

```cpp
#include "Fraction.hpp"

Fraction::Fraction(int n, int d):numerateur{n},
    denominateur{d} { // Initializer List
}
/* Identique a :
Fraction::Fraction(int n, int d) {
    numerateur = n;
    denominateur = d;
} */

Fraction::Fraction(int n):numerateur{n},
    denominateur{1} {
}
Fraction::Fraction():Fraction(0,1) {
}
```

Destructeurs

- Un destructeur est une fonction membre :
  - déclarée du même nom que la classe mais précédé d'un tilde (~) et sans type ni paramètre
  - appelée **automatiquement** lorsqu'un objet est détruit
- Rôle : libérer l'espace occupé par l'objet

Exemple

Fraction.cpp

```
#include "Fraction.hpp"

Fraction::~Fraction(){
    // Si besoin, liberation espace memoire
    // Ici, pas la peine de le declarer/definir
}
```

Appel des constructeurs

```
void test() {
    Fraction f1; // par defaut
    Fraction f2(10); // par parametre
    Fraction f3(3,7); // par parametres
}
```

Comment ça marche ?

```
void test() {
    Fraction f(5,6);
    // 1 - creer l'espace mémoire pour un objet
    //       de type Fraction (2 entier) sur la pile
    //  2 - affecter 5 au numerateur et 6 au
    //       denominateur (constructeur)
}
// 3 - detruire f (destructeur)
// 4 - liberer l'espace memoire sur la pile
```

Constructeur par copie
Opérateur d'affectation

Exemple

```
Fraction f1(5,6);
Fraction f2 {f1}; // constructeur par copie
Fraction f3;
f3 = f1; // operateur d'affectation
```

- Pour les classes simples, cela ne pose pas de problème (copie des membres),
- Cela se complique pour les classes avec l'allocation dynamique ...

Fonction amie

Fonction amie

- En théorie, une méthode **non-membre** d'une classe n'a **pas accès** aux attributs/méthodes privées ou protégées
- MAIS, si une méthode non-membre est déclarée en tant que **fonction amie**, elle y a accès
- Mot clé : `friend`

Fonction amie pour afficher

Exemple : Fraction.hpp

```
class Fraction {
private :
    int numerateur ;
    int denominateur ;
public :
    .../...
    friend std::ostream& operator<<
        (std::ostream& out,
        const Fraction & f);
};
```

Fonction amie pour afficher

Exemple : Fraction.cpp

```
/*! Fraction.cpp */

// C'est une fonction AMIE, pas une methode membre
std::ostream& operator<< (std::ostream& out,
    const Fraction & f) {
    out << "Fraction : " << f.numerateur << "/";
    out << f.denominateur << std::endl;
    return (out);
}

/*! main.cpp */
Fraction f(5,6);
std::cout << f << std::endl;  // "Fraction : 5/6"
```

Allocation dynamique

Allocation dynamique : new et delete

```
void test() {
    Fraction * pf;
    pf = new Fraction(7,5);
    delete pf;
}
```

Comment ça marche ?

```
void test() {
    Fraction * pf;
    // 1 - Creer l'espace memoire pour le pointeur
    //     pf sur la pile
    pf = new Fraction(7,5);
    // 2 - Allouer une case memoire pour un objet
    //     de type Fraction (2 entiers) sur le tas
    // 3 - Affecter 7 au numerateur, 5 au denominateur
    delete pf;
    // 4 - Detruire la fraction pointee par pf
    // 5 - Liberer l'espace memoire pour l'objet
    //     Fraction sur le tas
}
// 6 - Liberer l'espace memoire sur la pile
```

Allocation dynamique : new[] et delete[]

```
void test() {
    int n;
    cout << "Nombre de fractions a creer ? ";
    cin >> n;
    Fraction * tf = new Fraction[n];
    for (int i=0; i<n; ++i) {
        tf[i] = Fraction(i,i+1);
    }
    delete [] tf;
}
```

Comment ça marche ?

```
void test() {
    int n;
    cout << "Nombre de fractions a creer ? ";
    cin >> n;
    Fraction * tf = new Fraction[n];
    // Tableau dynamique des fractions
    // n constructeurs par defaut sont appeles
    for (int i=0; i<n; ++i) {
        tf[i] = Fraction(i,i+1);
    }
    delete [] tf;
    // Les fractions dans le tableau tf sont
    // detruites avant la desallocation de tf
}
```

- Le constructeur par défaut de Fraction **doit exister** !

Allocation dynamique : new[] et delete[]

```
void test() {
    ...
    Fraction ** ppf = new Fraction*[n];
    // Tableau de pointeurs de fractions
    for (int i=0; i<n; ++i) {
        ppf[i] = new Fraction(i,i+1);
    }
    ...
    for (int i=0; i<n; ++i) {
        delete ppf[i];
        // Liberation de chaque pointeur
    }
    delete[] ppf;
    // Liberation du tableau
}
```

Allocation dynamique
Constructeur par copie
Opérateur d'affectation

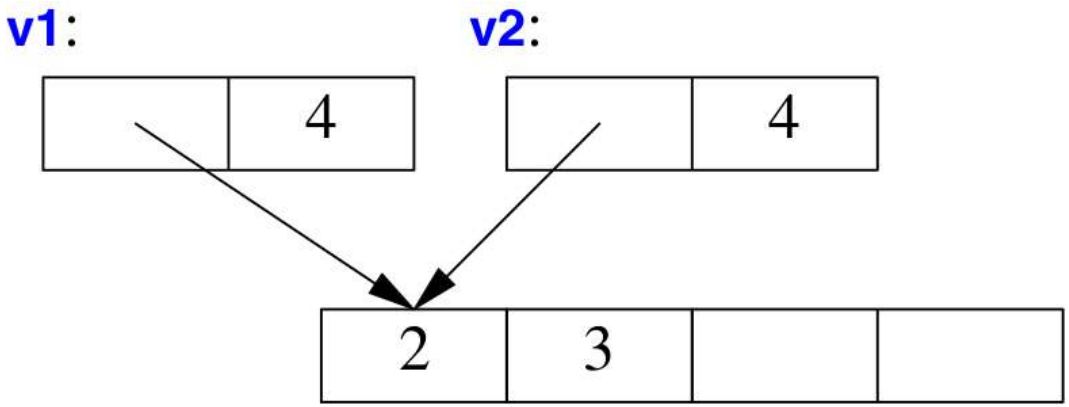
Exemple : classe Vector

```
/*! Vector.hpp */
class Vector {
private:
    double* elem;
    int sz;
public:
    // constructeur avec allocation dynamique
    Vector(int s);
    ~Vector();
};

/*! Vector.cpp */
Vector::Vector(int sz) : elem{new double[s]}, sz{s}{
}
```

Attention : shallow copy !

```
void bad_copy (Vector v1) {
    Vector v2 = v1; // Constructeur par copie
                    // copie v1 dans v2
    v1[0] = 2; // v2[0] est aussi 2
    v2[0] = 3; // v1[0] est aussi 3
}
```

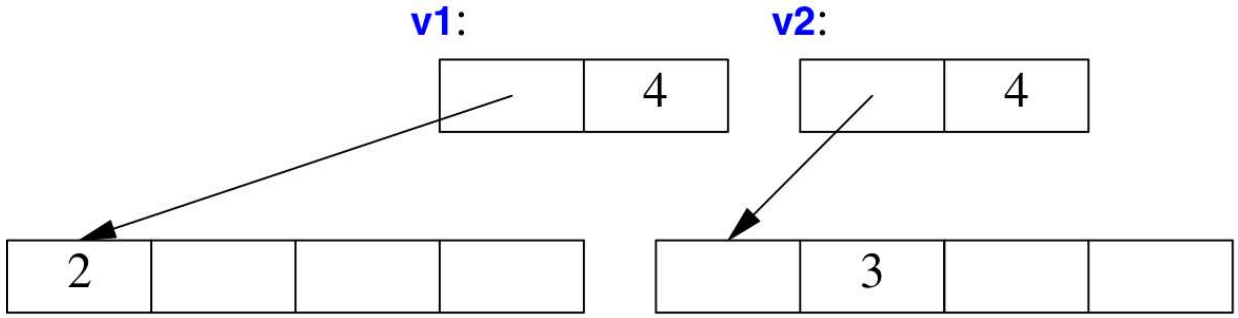


Classe Vector

```
class Vector {
private :
    double* elem;
    int sz;
public :
    // constructeur avec allocation dynamique
    Vector(int s);
    //constructeur par copie
    Vector(const Vector& v);
    //operateur d'affectation
    Vector& operator= (const Vector& v);
    ~Vector();
};
```

Constructeur par copie

```
Vector::Vector (const Vector& v) :
   elem{new double[v.sz]},sz{v.sz} {
      // copie terme a terme
      for (int i=0; i<sz; ++i) {
            elem[i] = v.elem[i];
      }
}
```



Opérateur d'affectation

```
Vector& Vector::operator=(const Vector& v) {
    sz = v.sz;
    delete[] elem; // supprime les anciens elements
    elem = new double[sz];
    // copie terme a terme
    for (int i=0; i<sz; ++i) {
        elem[i] = v.elem[i];
    }
    return *this;
}
```

- Chaque objet en C ++ a accès à sa propre adresse via un pointeur appelé **this**
- Seules les fonctions membres ont le pointeur **this**