Séance 2

Constructeur & Destructeur
Allocation dynamique

fraction.h

```
class Fraction {
    private :
        int numerateur;
        int denominateur;
    public :
        Fraction();                 // constructeur par défaut
        Fraction(int n);            // constructeur avec 1 paramètre
        Fraction(int n, int d);     // constructeur avec 2 paramètres
        ~Fraction();                // destructeur
        // autres méthodes
};
```

Constructeur

- 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.

Opérateur de résolution de portée « :: »

- Lorsque les définitions des classes deviennent complexes, la lisibilité du code diminue, car on perd la **vue d'ensemble des comportements** définit dans une classe.
- Il est toutefois possible de retrouver cette vue d'ensemble, en **écrivant les définition des méthodes à l'extérieur** de la déclaration de la classe
- Ceci est rendu possible grâce à l'opérateur de résolution de portée «::», qui permet de relier la définition d'une méthode à la classe pour laquelle elle est définie.
- Grace à cet opérateur, le programmeur peut se contenter d'indiquer les **prototypes des méthodes** dans la déclaration de la classe, et écrire les définitions correspondantes, à l'extérieur de la déclaration, sous la forme de définition de fonctions de nom:

<nom de classe>::*<nom de fonction>*(*<arg1>*,*<arg2>*, ...)

{ ... }

fraction.cpp

```
Fraction::Fraction() {
    numeroteur = 0;
    denominateur =1;
}
Fraction::Fraction() : numerateur {0}, denominateur {1} {}

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

Fraction::Fraction(int n, int d) : numerateur {n}, denominateur {d} {}

Fraction::~Fraction() {}
```

Appel des constructeurs main.cpp

```
void main() {
    Fraction f1; -- sans parentheses vides
    Fraction f2(10);
    Fraction f3(5,6);
```

```
    Fraction f4;                    f4(f3);
   f4 = f3; ou bien}
```

Destructeur

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

Destructeur

- une méthode particulière, appelée ***destructeur***, est définie.
- Cette méthode est invoquée automatiquement en fin de vie de l'instance, et son rôle principal est **d'assurer la libération des ressources** éventuellement mobilisées.

Destructeur

- La syntaxe de déclaration d'un destructeur, pour une classe nommée NomClasse, est:

```
~NomClasse()
        {
            // opérations
        }
```

- Le destructeur d'une classe est donc une méthode **sans argument** dont le nom est celui de la classe, préfixé du signe «~» (tilde)

- Comme le destructeur n'admet pas d'argument, il n'est pas possible de le surcharger.

- Toute classe admet donc exactement **un** destructeur; s'il n'est pas défini explicitement par le programmeur, c'est le compilateur qui se chargera d'en générer une version minimaliste

Destructeur

- Le problème du comptage des instances (en activité) d'une classe constitue un exemple (académique) d'utilisation des destructeurs pour lequel l'implémentation réalisée par le compilateur n'est pas acceptable:
- Pour connaître à tout moment le nombre d'instance de la classe Rectangle, il suffit d'utiliser une variable globale de type entier long comme compteur:

⇨ le constructeur incrémente la variable,

⇨ tandis que le destructeur la décrémente:

```
long counter(0); // variable globale
// ...
class Rectangle
{
// ...
   Rectangle(): hauteur(0),largeur(0) { ++counter; } // constructeur
  ~Rectangle() { --counter; } // destructeur
// ...
};
```

Destructeur

- On constate immédiatement, en considérant l'exemple précédant, que tous les cas de figure ne sont pas envisagés, et que le comptage n

void main()----------counter
{ ---------- 0
  Rectangle r1;---------- 1
   {
        Rectangle r2;---------- 2
   }---------- 1
}---------- 0

- La copie d'un rectangle en un autre, échappe au compteur d'instance, puisque nous n'avons pas définit le comportement du *constructeur de copie*

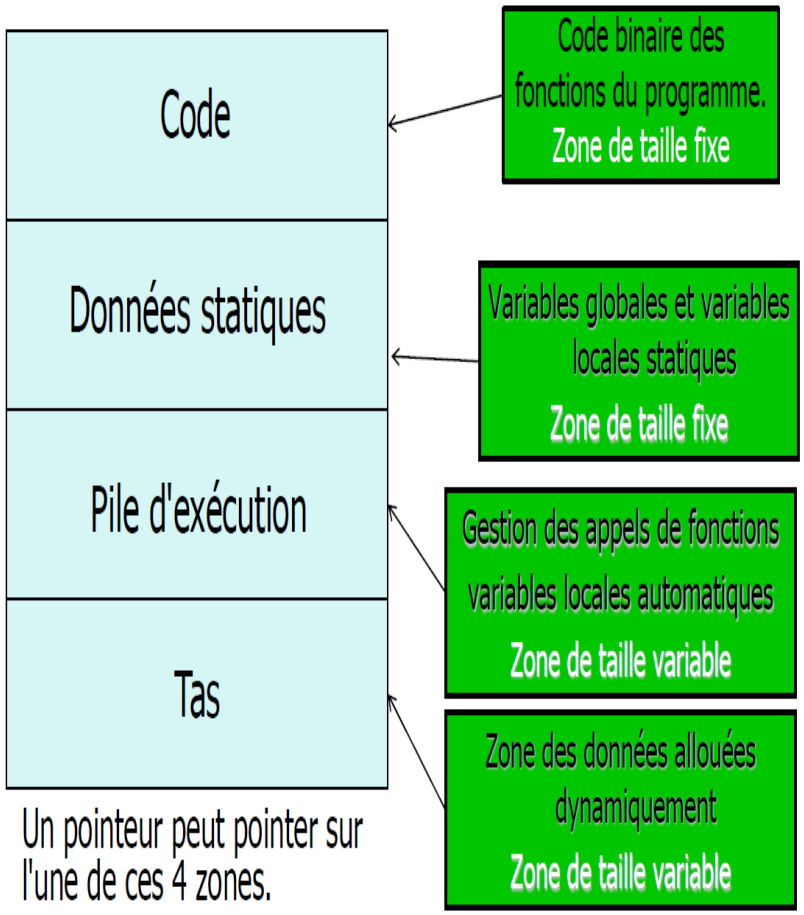
Allocation dynamique : new et delete
Comment cela marche ?

```
void test() {
    Fraction f3(5,6);
    // 1) allouer une mémoire pour un objet de type Fraction (2 entiers) sur la pile
    // 2) affecter 5 au numérateur et 6 au dénominateur (constructeur)
}
// 3) détruire f3 (destructeur)
// 4) libérer l'espace mémoire sur la pile
```

Allocation dynamique

Allocation dynamique : new et delete
Comment cela marche ?

```
void test() {
    Fraction * pf;
     // 1) créer un espace mémoire pour le pointeur pf sur la pile
    pf = new Fraction(5,6);
     // 2) allouer une mémoire pour un objet de type Fraction (2 entiers) sur le tas
    // 3) affecter 5 au numérateur et 6 au dénominateur
    delete pf;
    // 4) détruire la fraction pointée par pf
    // 5) libérer l'espace mémoire pour l'objet fraction sur le tas
}
// 6) libérer l'espace mémoire sur la pile
```



Pointeur en C

- Un tableau est une zone mémoire pouvant contenir N éléments consécutifs de même type
- Le nom d'un tableau est un pointeur constant. C'est une adresse, et plus précisément l'adresse du premier élément du tableau (d'indice 0).

- Fraction tab[20];
- Fraction *pf;
- pf=&tab[0]; /* <=> pf=tab; */
- pf++; /* p pointe désormais sur le deuxième élément du tableau */
- tab++; /* n'est pas possible car tab est un pointeur constant */

&tab

```
int main(void)
{
   Fraction tab[3];
   Fraction *p;
   Fraction **pp;
   Fraction (*pf)[3];

   p = &tab;
   pp = &tab;
   pf = &tab;

   return 0;
}
```

warning: assignment to 'int *' from incompatible pointer type 'int (*)[3]'

warning: assignment to 'int **' from incompatible pointer type 'int (*)[3]'

Seule l'affectation pt = &tab ne provoque aucun avertissement. En effet, le type de pt est le même que celui de l'expression &tab

lvalue

- un tableau ne peut pas être affecté, autrement dit, il ne peut pas être la valeur à gauche (left value) d'une affectation. Et bien sûr, il ne peut pas non plus être incrémenté ni changé d'une quelconque manière. Un tableau est une lvalue non modifiable.

```
void f(void)
{
   int t1[3], t2[3], *p;
   t1 = 0;  /* erreur */
   t1 = t2; /* erreur */
   t1++;    /* erreur */
   ++t1;    /* erreur */
   t1--;    /* erreur */
   --t1;    /* erreur */
   t1 += 1; /* erreur */
   t1 -= 1; /* erreur */
}
```

Notons que si l'on remplace t1 par p dans chaque instruction ci-dessus, alors il n'y a plus d'erreur, car p est une lvalue modifiable.

Résumé

- Quand on déclare Fraction t1[3]; on obtient un tableau de trois Fractions en mémoire.
- Quand on déclare Fraction *p1; on obtient un pointeur de Fraction en mémoire.
- Quand on déclare Fraction t2[3][4]; on obtient trois tableaux de quatre Fractions, soit douze Fractions en mémoire.
- Quand on déclare Fraction **p2; on obtient un pointeur de pointeur de Fractions en mémoire.
- Quand on déclare Fraction *t3[3]; on obtient un tableau de trois pointeurs de Fractions en mémoire.
- Quand on déclare Fraction (*p3)[3]; on obtient un pointeur de tableau de 3 Fractions en mémoire.

Pointeur en C

- un tableau de pointeurs sur des fractions.

- Fraction * tableau[20]; /* crée un tableau de pointeurs sur des fractions, de taille 20 */

Allocation dynamique :new[] et delete[]
Comment cela marche ?

```
void test() {
    int n;
    cout << " Nombre de fractions à créer";
    cin >> n;
    Fraction * pf = new Fraction[n];   //Fraction *pf[n]
        // tableau dynamique des fractions
        // n constructeurs par défaut sont appelés
    for (int i = 0; i <n; i++) {
        pf[i] = Fraction(i,i+1);
    }
    delete[] pf;
    // les fractions dans le tableau pf sont détruites avant la
    désallocation de tf
}
```

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

Allocation dynamique : new et delete

```
void test() {
    // pointeur tableau des pointeurs aux fractions
    Fraction ** ppf = new Fraction * [n];  // un tableau de n pointeurs de fraction sans nom
    // tableau des pointeurs aux 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];
    }
    delete[] ppf;
```

Constructeur de copie
Opérateur d'affectation

Exemple

```
Fraction f1(5,6);
Fraction f2 (f1) ;          // constructeur de copie
Fraction f3;
f3 = f1;                    // opérateur d'affectation
```

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

Affectation

- Par défaut, affectation membre par membre.

```
class Compte {
        double actif;
        Compte(double);
        public: // liste des methodes
};
Compte C(2855.20);
Compte B(450.30);
B = C; // affectation.
```

Constructeur de recopie

- De la même manière que l'affectation (par défaut), le constructeur de recopie par défaut effectue la copie membre à membre de la source vers la destination.
- Il est appelé dans trois situations:

Initialisation d'un objet

```
Compte C(280.98);
// constructeur de recopie: création de l'objet B
// et son initialisation avec les données de C.
Compte B(C);
ou
Compte B = C;
```

Attention : il faudra différencier entre recopie et affectation.

Compte B; // création de l'objet.

B = C; // pas de recopie mais uniquement l'affectation

// car l’objet est déjà créé.

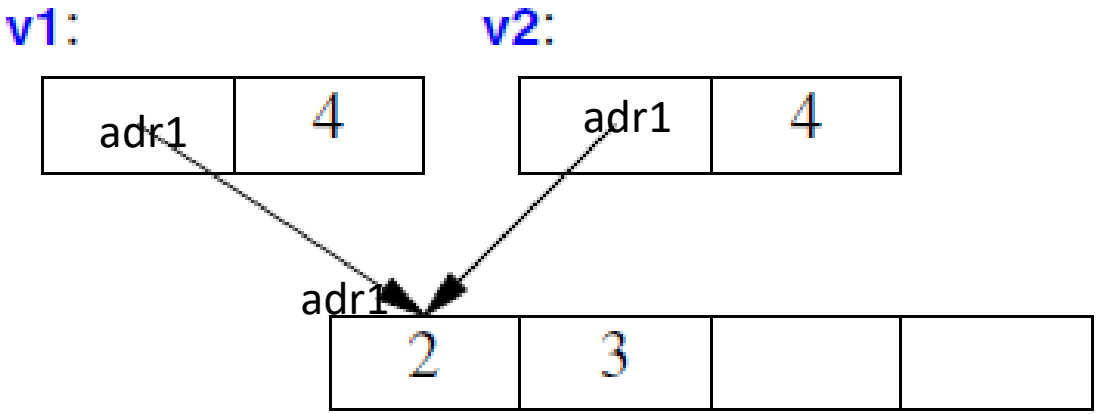
Classe Vector

```
// vector.h
class Vector {
        private:
                double* elem;
                unsigned int sz;
        public:
                Vector(unsigned int s);  //constructeur avec l'allocation dynamique
                ~Vector();
};

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

Vector::Vector(unsigned int s) {
        elem = new double[s];
        sz = s;
}
```

Attention !
Ne pas confondre affectation et recopie

Affectation

```
void bad_copy(Vector v1){
Vector v2 = v1; // constructeur par recopie
                 //copie de v1 dans v2
v1[0] = 2;       //v2[0] est aussi 2
v2[1] = 3; ;     //v1[0] est aussi 3
}
```

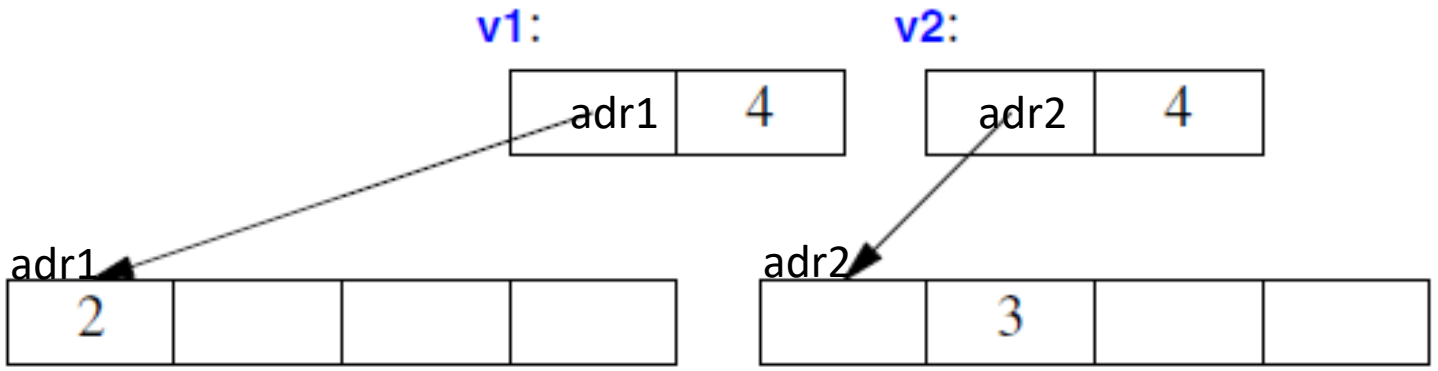


Classe Vector

```
// Vector.h
class Vector {
    private:
        double* elem;
        unsigned int sz=10;
    public:
        Vector();
        Vector(const Vector& a);                    //constructeur de copie
        Vector& operator=(const Vector& a);         //opérateur d'affectation
        ~Vector() {delete[] elem; }

        Vector(unsigned int s);

};
```

Constructeur de copie

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



Opérateur d'affectation

```
Vector& Vector::operator=(const Vector& a) {
    sz = a.sz;
    delete[] elem;        // supprimer anciens éléments
    elem = new double[sz];
   // copie terme a terme
    for (unsigned int i=0; i<sz; ++i)
          elem[i] = a.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**

Opérateur d'affectation

- Test

```
int main(){
Vector *v2 = new Vector (4);
Vector v1;

v1=v2;
v1.operator=(v2);

v1.Vector(v2);



return 0;
}
```

Fonction amie pour afficher :

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

```
/*! fraction.h */

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

Fonction amie pour afficher

```
/*! fraction.cpp */
// C ' est une fonction AMIE , pas  une  methode membre*
//ele accède à toutes les proprités de la classe comme une méthode membre
    de la classe
ostream& operator<< (ostream& out, const Fraction & f) {
    out << "Fraction : " << f.numerateur  << "/" << f.denominateur  << endl;
    return out;
}

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

Classe amie [friend class F or friend F]

```
namespace NS
{
    class M
   {
        friend class F;  // Introduces F but doesn't define it
    };
}

namespace NS {
  class M    {
        friend F; // error C2433: 'NS::F': 'friend' not permitted on data declarations
   };
}
```

```
class F {};
namespace NS
{
   class M
   {
      friend F;  // OK
   };
}
```

Dans l'exemple suivant, friend F fait référence à la F classe déclarée en dehors de l'étendue de NS.

Permet friend F de déclarer un paramètre de modèle en tant qu'ami :

```
class Base {
  public:
    friend void f(Base& b);
    // ...
  protected:
    virtual void do_f();
    // ...
};

inline void f(Base& b)
{
  b.do_f();
}

class Derived : public Base {
  public:
    // ...
  protected:
    virtual void do_f();  // "Redéfinit" le comportement de f(Base& b)
    // ...
};

void userCode(Base& b)
{
   f(b);
}
```

L'instruction f(b) dans la fonction userCode(Base&) va invoquer b.do_f(), qui est virtual. Ce qui veut dire que Derived::do_f() va être appelée si b est effectivement un objet de la classe Derived. Notez que Derived redéfinit le comportement de la fonction membre protected: virtual do_f(); elle, n'a pas sa propre version de la fonction friend f(Base&).

l'amitié n'est ni héritée ni transitive, ni réciproque

- **Je ne fais pas forcément confiance aux enfants de mes amis** Les privilèges de l'amitié ne sont pas hérités. Les classes dérivées d'une classe amie ne sont pas forcément des amis. Si la classe Fred déclare que la classe Base est une amie, les classes dérivées de Base n'ont pas à avoir automatiquement des droits d'accès particuliers aux objets de type Fred.

- **Je ne fais pas forcément confiance aux amis de mes amis** Les privilèges de l'amitié ne sont pas transitifs. Un ami d'un ami n'est pas forcément un ami. Si la classe Fred déclare que la classe Wilma est une amie, et que la classe Wilma déclare que Betty est une amie, la classe Betty n'a pas à avoir automatiquement des droits d'accès particuliers aux objets de type Fred.

- **L'amitié n'est pas réciproque**