Constructeurs – destructeurs

Nicolas Audebert

Vendredi 27 novembre

Avant toute chose

Rendus de TP et des exercices

Les rendus se font sur **Educnet**, même en cas de retard.

1. Le code rendu **doit compiler**.
2. Le code rendu doit **être propre** (indentation, noms de variables clairs).
3. Le code rendu doit **être commenté** (réponses aux questions, fonctionnement du code).
4. Rassembler le code dans une seule archive (`.zip`, `.rar`, `.tar.gz`, etc.).
5. Le code doit contenir **les noms des deux binômes** le cas échéant.

Un exercice ou un TP rendu en retard ou ne respectant pas une des consignes ci-dessus sera pénalisé.

Rappels

Les objets

```
// Structure + fonctions
struct Obj1{
    int x;
};
int f(Obj1 &x);
int g(Obj1 &x, int y);

...
Obj1 a;
cout << f(a) << endl;
int i = g(a,10);
```

```
// Objet
struct Obj1{
    int x;
    int f();
    int g(int y);
};

...
Obj1 a;
cout << a.f() << endl;
int i = a.g(10);
```

On met simplement les déclarations **dans** la structure. **On ne met plus en argument** l'objet en question.

Manipulation des objets

Les objets se manipulent de façon similaire aux structures.

```
struct Obj1{
  int x;
  void double_x();
  int renvoie_4x();
};

int main(){
  // En dehors de l'objet, on
  // utilise . pour accéder
  // aux champs et méthodes
  Obj1 a;
  a.x = 3;
  cout << a.renvoie_4x() <<
  ↪  endl;
}
```

```
void Obj1::double_x(){
  // À l'intérieur du
  ↪  namespace
  // Obj1, on peut modifier
  ↪  ses champs
  x = 2*x;
}
int Obj1::renvoie_4x(){
  // À l'intérieur du
  ↪  namesapce
  // Obj1, on peut utiliser
  ↪  ses méthodes
  double_x();
  return x;
}
```

Classes

Les classes permettent de protéger l'accès à certains champs et méthodes afin de restreindre **l'interface** de l'objet.

```
// matrice.h
class Matrice{
  // privé par défaut
  int m,n;

public: // public à partir d'ici
  // Méthodes
  void cree(int m1,int n1);
  void detruit();
  double get(int i,int j);
  void set(int i,int j,double x);
  void affiche();

private: // privé à partir d'ici
    double* t;
};
```

Construction des objets

Classes et structures

Séance précédente : structure + fonctions $\longrightarrow$ objets

Par exemple :

```
struct Point{
    double x,y;
};
...
Point a;
a.x = 2; a.y = 3;
i = a.x; j = a.y;
```

```
class Point{
    double x,y;
public:
    double get(double& x, double&
    ↪  y);
    void set(double valX, double
    ↪  valY);
}
...
Point a;
a.set(2,3);
a.get(i,j);
```

Initialisation d'une structure vs. initialisation d'un objet

```
class Point{
    double x,y;
public:
    double get(double& x, double& y);
    void set(double valX, double valY);
}
...
Point a; // OK
a.set(2,3);
a.get(i,j);
...
Point b = {2,3};
// ERREUR
// x et y sont privés :
// ils sont inaccessibles en
// dehors de la classe
```

Notion de constructeur

L'initialisation d'un objet fait appel au **constructeur** de sa classe.

Définition

Un **constructeur** est une méthode :

- qui **n'a pas** de type de retour,
- qui porte **le même nom** que la classe,
- qui décrit comment initialiser les instances de la classe.

Manipulation

Un constructeur :

- est **systématiquement** appelé à la création d'un objet,
- ne peut pas être appelé **après** la création de l'objet.

Notion de constructeur

Définition

Un **constructeur** est une méthode :

- qui **n'a pas** de type de retour,
- qui porte **le même nom** que la classe,
- qui décrit comment initialiser les instances de la classe.

```
class Point{
    double x,y;
public:
    Point(double valX, double valY);
    ...
}

// Définition du constructeur
Point::Point(double valX, double
↪  valY){
    x = valX; y = valY;
}
```

```
...
Point b = {2,3}; //
↪  ERREUR
Point c(2,3); // OK
// Cette syntaxe
↪  appelle
// le constructeur
```

Le constructeur vide

À la création de l'objet il y a **toujours** un appel à un constructeur.

Lorsqu'aucun constructeur n'est défini par l'utilisateur, le compilateur en créé un par défaut. C'est un **constructeur vide** qui ne prend aucun argument et ne fait que créer les champs de l'objet.

```
class Point{
    double x,y;
public:
    double get(double& x, double& y);
    void set(double valX, double valY);
};
...
Point a;// Appel au constructeur par défaut
```

Redéfinir le constructeur vide

Il est possible de redéfinir le constructeur vide (dans ce cas, le constructeur par défaut est oublié).

```
class Point{
private:
    double x,y;
public:
    // Constructeur vide
    Point();

    double get(double& x,
               double& y);
    void set(double valX,
             double valY);
};
```

```
Point::Point(){
  cout << "Constructeur vide" <<
  ↪  endl;
  x = 0; y = 0;
}
```

```
...
Point a; // "constructeur vide"
a.get(i,j);
cout << i << ", " << j << endl;
// affiche "0, 0"
```

Surchage des constructeurs

Il est possible de définir plusieurs constructeurs (comme pour les méthodes) manipulant différentes combinaisons d'arguments.

```
class Point{

    double x,y;
public:

    Point(double val);
    Point(double valX,
          double valY)
    ...
};
```

```
Point::Point(double val){
    x = y = val;
}
Point::Point(double valX, double
↪  valY){
    x = valX; y = valY;
}

...
Point a(2); // 2 2
Point b(2,3); // 2 3

Point c; // ERREUR (pas de
         // constructeur vide)
```

Remplacement du constructeur vide

Attention !

Dès lors qu'un constructeur est défini, quel qu'il soit, **le constructeur par défaut n'existe plus**.

Dans ce cas, il faut définir son propre constructeur vide pour pouvoir écrire `Point c;` .

```
class Point{
    double x,y;
public:
    Point();
    Point(double val);
    Point(double valX, double valY)
};
Point::Point(){} // Constructeur vide (ne fait rien)
```

Constructeurs et tableaux d'objets

Cas général

Créer un tableau d'objets requiert l'existence du **constructeur vide** pour cette classe d'objets.

```
Point t[10]; // appelle 10 fois Point()
Point* t2 = new Point[1000]; // appelle 1000 fois Point()
//pour remplir :
for(int i=0; i<1000; i++)
    t2.set(0,0);
```

Cas particulier

Dans le cas d'une initialisation directe utilisant la syntaxe entre accolades { }, il est possible de se passer du constructeur vide.

```
Point t[3] = {Point(0), Point(1,2), Point(5, -5)};
```

Objets temporaires

Objets temporaires

Certains objets sont créés puis meurent très rapidement :

```
void f(Point p){
    ...
}
Point g(){
    Point temp(1,2);
    return temp;
}

...
Point p1(5,6);
f(p1);

Point p2 = g();

Point p3 = g();
f(p3);
```

Objets temporaires

Dans ce cas, il est préférable d'utiliser des objets temporaires anonymes pour éviter les recopies inutiles :

```
void f(Point p){
    ...
}
Point g(){
    return Point(1,2);
}

...
f(Point(5,6));
Point p2 = g();
f(g());

Point p3;
p3 = g();
p3 = Point(1,2);
```

Erreur communes

Il ne faut pas abuser des objets temporaires.

Les objets temporaires ne sont pas des accesseurs!

```
Point p3;
p3 = g();
...
p3 = Point(1,2); // moins efficace que p3.set(1,2)
```

Dans `p3.set(1,2)` il n'y a pas de création d'objet temporaire.

Les objets temporaires ne sont pas des constructeurs!

```
Point p4 = Point(1,2); // objet temporaire inutile
Point p5(1,2); // on appelle le constructeur directement
```

Exemple d'utilisation d'objet temporaire

```
class Point{
    ...
    Point operator+(Point
    ↪  b);
};
Point Point::operator+(Point
↪  b){
    Point c(x+b.x, y+b.y);
    return c;
    // on retourne la copie
    ↪  de c
}
Point p1(1,2), p2(3,4);
Point p3 = p1 + f(p2);
```

```
class Point{
    ...
    Point operator+(Point
    ↪  b);
};

Point Point::operator+(Point
↪  b){
    return Point(x+b.x,
    ↪  y+b.y);
    // on retourne l'objet
    // temporaire
}

Point p3 = Point(1,2) +
↪  f(Point(3,4));
```

Passage par référence constante

Rappel : le passage par référence

Une fonction peut recevoir ses arguments de deux manières :

Passage par valeur (ou copie)

La fonction reçoit une **copie** interne de la variable en argument.

Passage par référence

L'e**space mémoire de la variable est partagé** et modifier l'argument dans la fonction modifie la valeur de la variable initiale.

Réduire le nombre de copies

Passer un argument par valeur nécessite de réaliser une copie. Copier un objet ou une variable ajoute un surcoût négligeable pour les variables natives, mais important pour les grands objets.

```
const int N = 1000;
class Vector{
    double t[N];
    ...
};
class Matrix{
    double t[N][N];
    ...
};
```

```
void solve(Matrix A, Vector
↪  x,
           Vector& y)
{...}

...
Matrix M;
Vector a,b;
...
solve(M,a,b);
```

La matrice **M** est copiée lors de l'appel à **solve**.

Tout passer par référence ?

Une solution serait de passer tous les arguments par référence pour éviter les copies.

```
const int N = 1000;
class Vector{
    double t[N];
    ...
};
class Matrix{
    double t[N][N];
    ...
};
```

```
void solve(Matrix &A,Vector
↪  &x,Vector& y)
{...}

...
Matrix M;
Vector a,b;
...

solve(M,a,b);
```

Risque

Rien ne garantit que **`solve`** ne modifie pas les arguments.

Référence constantes

Solution

On souhaite passer les objets par référence en spécifiant que l'argument ne doit pas être modifié : **on ajoute le mot clé `const`**.

```
const int N = 1000;
class Vector{
    double t[N];
    ...
};
class Matrix{
    double t[N][N];
    ...
};
```

```
void solve(const Matrix &A,
    const Vector &x, Vector&
    ↪  y)
{...}

...
Matrix M;
Vector a,b;
...
solve(M,a,b);
```

Méthodes constantes

Lorsqu'on utilise une référence constante, on ne peut accéder qu'aux méthodes définies comme **constantes**, *i.e.* qu'on a déclaré comme ne modifiant pas l'objet.

```
const int N = 1000;
class Vector{
    double t[N];
public
    double get(int i);
    void set(int i, double
    ↪  v);
    ...
};
class Matrix{
    double t[N][N];
    ...
};
```

```
void solve(const Matrix &A,
    const Vector &x, Vector&
    ↪  y)
{
    ...
    x.set(10, 8); // ERREUR: x
    ↪  est
                  // non
                  ↪  modifiable
    x.get(5); // ERREUR

    y.set(1, 5.6); // OK
}
...
```

Méthodes constantes

Lorsqu'on utilise une référence constante on accès qu'au méthodes définies comme **constantes** : *i.e.* qu'on a déclaré comme ne modifiant pas l'objet.

```
const int N = 1000;
class Vector{
    double t[N];
public
    double get(int i) const;
    void set(int i, double
    ↪  v);
    ...
};

double Vector::get(int i)
↪  const{
    return t[i];
}
```

```
void solve(const Matrix &A,
    const Vector &x, Vector&
    ↪  y)
{
    ...
    x.set(10, 8); // ERREUR: x
    ↪  est
                  // non
                  ↪  modifiable
    x.get(5); // OK, get est
    ↪  const

    y.set(1, 5.6); // OK
}

...
```

Destructeurs

Propriétés d'un destructeur

La création d'un objet appelle un constructeur.

La supression d'un objet appelle un **destructeur**.

Définition et propriétés

Un destructeur est une méthode qui :

- n'a pas de type de retour,
- n'a pas d'argument,
- porte le nom de la classe précédé de `~` (tilde).

Implémentation des destructeurs

Propriétés des destructeurs

Un destructeur est :

- unique pour chaque classe,
- fourni par défaut par remplaçable,
- **JAMAIS** appelé explicitement.

```
class Obj{
    ...
public:
    Obj(); // constructeur
    ↪  vide
    Obj(int i);

    ~Obj(); // destructeur
    ...
};
```

```
Obj::~Obj(){

    cout << "Destruction";
    cout << endl;

}
```

Destructeurs et tableaux

Le destructeur est appelé autant de fois qu'il y a d'éléments dans le tableau.

```
{
    Obj tab[100]; // appel 100 fois au constructeur vide
    ...
} // sortie de bloc : destruction de tab -> 100 appels au
↪  destructeur
```

En allocation dynamique, le destructeur est appelé lors du **delete** .

```
Obj* tab2 = new Obj[10000]; // 10000 appels au constructeur
↪  vide
...
delete[] tab2; // 10000 appels au destructeur de Obj
```

Destructeurs et tableaux

```
Obj* tab2 = new Obj[10000]; // 10000 appels au constructeur
↪  vide
...
delete[] tab2; // 10000 appels au destructeur de Obj
```

Attention erreur

Il est possible d'écrire `delete tab2` . Cela désalloue la mémoire mais n'appelle pas le destructeur sur les objets !

Constructeur de copie

Constructeur de copie

Le constructeur de copie permet de créer un objet à partir d'un autre.

```
class Obj{
    ...
    Obj(const Obj& o);
};
Obj::Obj(const Obj& o){...}
```

Il s'utilise quand on écrit :

```
Obj a;
Obj b(a);
```

```
Obj a;
Obj b = a;
// c'est la même syntaxe
↪  que Obj b(a);
```

Il est aussi implicitement utilisé lorsque l'on passe un objet par copie en argument d'une fonction.

Constructeur de copie

Définition

- Un constructeur de copie est fourni par défaut
- Par défaut il recopie les champs de **a** dans **b**
- Une fois redéfini, il fait **uniquement** ce qu'il y a dans la méthode.

Opérateur d’affectation (=)

L'opérateur d'affectation

Il est aussi possible de redéfinir l'opération d'affectation, le =.
Par défaut, il recopie les champs d'un objet dans l'autre.

```
class Obj{
    ...
    void operator=(const Obj& o);
};
void Obj::operator=(const Obj& o){...}
```

Opérateur =

```
class Obj{
    ...
    void operator=(const Obj& o);
};
void Obj::operator=(const Obj& o){...}

Obj a,b;
b = a; // OK
Obj c;
c = b = a; // ERREUR
```

Il faut lire :

```
Obj a,b,c;
c = b = a; // équivalent à
c = (b = a); // ou encore
c.operator=(b.operator=(a));
```

Opérateur =

```
class Obj{
    ...
    Obj operator=(const Obj&
    ↪  o);
};
Obj Obj::operator=(const Obj&
↪  o){
    ...
    return o;
}
```

```
Obj a,b;
b = a; // OK
Obj c;
c = b = a; // OK
```

Pour éviter une recopie supplémentaire au niveau du **return** :

```
class Obj{
    ...
    const Obj& operator=(const Obj& o);
};
const Obj& Obj::operator=(const Obj& o){
    ...
```

Avertissement

Attention !

Il ne faut jouer aux apprentis sorciers. Ce n'est pas la peine de recoder le destructeur et le construteur par copie lorsque cela n'est pas nécessaire.

Objets avec allocation dynamique

Une classe de vecteur

```
class Vect{
    int n;
    double* t;
public:
    Vect(int taille);
    ~Vect();
};
```

```
Vect::Vect(int taille){
    n = taille;
    t = new double[n];
}
Vect::~Vect(){
    delete[] t;
}
```

```
void f(){
    Vect v(1000); // contructeur -> allocation
    ...
} // destructeur -> desallocation
```

Plus besoin de faire les **new** et les **delete[]** à la main.

Une classe de vecteur

Petit problème : si on veut faire des tableaux de `Vect`, ou si on donne une taille négative ou nulle.

```
class Vect{

    int n;
    double* t;

public:

    Vect();

    Vect(int taille);

    ~Vect();

};
```

```
Vect::Vect(){
    n = 0;
}
Vect::Vect(int taille){
    if(taille > 0){
        n = taille;
        t = new double[n];
    } else {
        n = 0;
    }
}
Vect::~Vect(){
    if(n > 0){
        delete[] t;
    }
}
```

Une classe de vecteur

Un autre problème : le code suivant ne fonctionne pas.

```
int main(){
    Vect v1(100), v2(100);
    v1 = v2; // fuite de
    ↪  mémoire
    return 0;
}

class Vect{
    int n;
    double* t;
public:
    Vect();
    Vect(int taille);
    ~Vect();
    const Vect& operator=(
            const Vect&
            ↪  v);
};
```

```
const Vect& Vect::operator=
            (const Vect& v){
    if(n>0)
        delete[] t;
    n = v.n;
    if(n>0){
        t = new double[n];
        for(int i=0; i<n; i++)
            t[i] = v.t[i];
    }
    return v;
}
```

Une classe de vecteur

```
class Vect{
    int n;
    double* t;
    void alloue(int
    ↪  taille);
    void detruit();
    void copie(const Vect&
    ↪  v);
public:
    Vect();
    Vect(const Vect& v);
    ~Vect();
    const Vect& operator=(
        const Vect& v);
    Vect(int taille);
};
```

```
void Vect::alloue(int taille){
    if(taille > 0){
        n = taille;
        t = new double[n];
    } else {
        n = 0;
    }
}
void Vect::detruit(){
    if(n > 0)
        delete[] t;
}
void Vect::copie(const Vect&
↪  v){
    alloue(v.n);
    for(int i=0; i<n; i++)
        t[i] = v.t[i];
}
```

Une classe de vecteur

```
class Vect{
    int n;
    double* t;
    void alloue(int
    ↪  taille);
    void detruit();
    void copie(const Vect&
    ↪  v);
public:
    Vect();
    Vect(const Vect& v);
    ~Vect();
    const Vect&
    ↪  operator=(const
                 Vect& v);
    Vect(int taille);
};
```

```
Vect::Vect(){
    alloue(0);
}
Vect::Vect(const Vect& v){
    copie(v);
}
Vect::~Vect(){
    detruit();
}
const Vect& Vect::operator=(
        const Vect& v){
    if(this != &v){
        detruit();
        copie(v);
    }
    return v;
}
Vect::Vect(int taille){
    alloue(taille);
}
```

Serpent

Un serpent qui se déplace et s'allonge tout les x pas de temps.

Tron

Un serpent deux joueurs qui s'allonge à tous les pas de temps.