Objects avec allocation dynamique

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

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);
    ...
}
```

```
Point c(2,3);
```

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

Constructeur vide

Le **constructeur vide** est un constructeur par défaut qui ne fait rien. On peut le **redéfinir**.

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

    ...
};
```

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

Point a; // "constructeur vide"
```

Attention !

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

Créer un tableau d'objets requiert l'existence du constructeur vide.

Objets temporaires

Il est préférable d'utiliser des objets temporaires anonymes pour éviter les recopies inutiles :

```
void f(Point p){
    ...
}
Point g(){
    return Point(1,2);
    // Plutôt que:
    // p = Point(1,2);
    // return p;
}

...
f(Point(5,6));
// Plutôt que :
// Point p2 = g();
// f(g());
```

Destructeur

La suppression 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).

Propriétés des destructeurs

Un destructeur est :

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

Retour sur le partiel

Faire des tests

Ce qui fonctionne dans sa tête n'est pas toujours correct en pratique :

```
bool operator<=(Bloc B, Bloc C){
    if(B.x>=C.x and B.y<=C.y and B.h<=C.h and B.l<=C.l){
        return true;
    }
    else {
        return false;
    }
}
```

Écrire un test !

```
Bloc bloc1 = ...
Bloc bloc2 = ...
bool inclus = bloc1 <= bloc2
cout << inclus << endl;
// pareil pour le cas bloc1 pas inclus dans bloc3
```

Noms de variables/fonctions

```
// Bof
struct Bloc {
    int x,y; // Coin en haut à gauche du bloc
    int l,h; // Largeur et hauteur
};

// Pire
struct Bloc {
    int g;
    int d;
    int h;
    int b;
};
```

Casse

Les minuscules et majuscules ont une importance. On préférera les majuscules pour les **classes** et les minuscules pour tout le reste.

Éviter les fautes de goût

```
bool operator<=(Bloc A,Bloc B){

    ↪  if((B.chg[0]<=A.cbd[0]<=B.cbd[0])and(B.cbd[1]<=A.cbd[1]<=B.chg
        return true;
    }
    else
        return false;
}

// Un peu mieux
bool operator<=(Bloc A,Bloc B){
    bool inclus_x = ...
    bool inclus_y = ...

    return inclus_x && inclus_y;
}
```

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 constante

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
}

...
```

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 au niveau du **return** :

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

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.