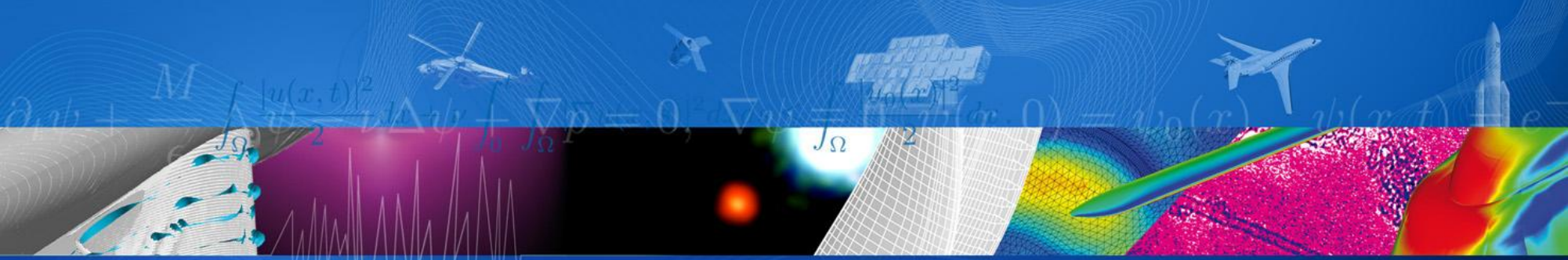

Introduction à la programmation C++

Constructeurs et destructeurs
Vector de la STL

Nicolas Audebert

Mercredi 30 novembre 2016

ONERA
THE FRENCH AEROSPACE LAB

retour sur innovation

Plan de la séance

Un petit problème avec les classes

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);
```

Un petit problème avec les classes 2

```
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 prives
// ils sont inaccessibles en
// dehors de la classe
```

Solution

Il faut utiliser un **constructeur**.
Un constructeur est une méthode :

- qui **n'a pas** de type de retour
- qui porte le même nom que l'objet

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

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

```
...

Point b = {2,3};
// ERREUR


Point c(2,3);
// OK
// appel le constructeur a la
// creation de l'objet
```

Plan de la séance

Constructeur

Un constructeur est une méthode :

- qui **n'a pas** de type de retour
- qui porte le même nom que l'objet

Un constructeur :

- est toujours appelé à la création de l'objet
- ne peut pas être appelé après la création de l'objet

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, il y a en un par défaut. C'est un **constructeur vide** : pas d'argument, il ne fait que créer les champs.

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

Redéfinir le constructeur vide

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

```
class Point{

    double x,y;
public:

    Point(); // constructeur vide

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

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

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

Plusieurs constructeurs

Il est possible de définir plusieurs constructeurs (comme pour les méthodes). Les arguments doivent être différents.

```
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)
```

Constructeur vide ?

ATTENTION

Lorsqu'un constructeur est défini, **le constructeur par défaut n'existe plus**.

Ainsi pour utiliser :

```
Point c;
```

il faut un constructeur vide.

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

Constructeur : tableaux d'objets

Cas général

Pour créer un tableau d'objet il faut un constructeur vide.

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

Cas particulier

Initialisation avec les $\{\}$

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

Plan de la séance

Destructeur

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

- le destructeur est unique
- un destructeur est fourni par défaut
- il est possible de redéfinir le destructeur
- on n'appelle **JAMAIS** explicitement le destructeur

Destructeur : implémentation

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 $\sim$

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

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

```
Obj::~Obj(){

    cout << "Destruction de l'objet";
    cout << endl;

}
```

Destructeurs : tableaux d'objets

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 -> appel 100 fois 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 des objets.

Plan de la séance

Constructeur de copie

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

est utilisé :

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

- ```
  Obj a;
  Obj b = a; // en fait equivalent a Obj b(a);
  ```

- pour construire les objets dans les arguments des fonctions

mais pas ici :

```
Obj a,b;
b = a; // c'est l'operateur =
```

Constructeur de copie 2

- 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.

Plan de la séance

Opérateur =

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; //equivalent a
c = (b = a); // ou encore
c.operator=(b.operator=(a));
```

Mais = est une méthode void.

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 aller plus loin : (comme ça on ne fait pas de recopie au moment du return)

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

Plan de la séance

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 :(plus compliqué) le code suivant ne fonctionne pas.

```
int main(){
    Vect v1(100), v2(100);
    v1 = v2; // fuite de memoire
    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;
}
```

Ce code ne fonctionne pas si on fait v=v. (désallocation puis lecture dans une zone qui n'existe plus)

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);
}
```

Plan de la séance

Standard Template Library

- Librairie disponible par défaut en C++
- De nombreux modules :
  - classes : chînes de caractères, tableaux, piles ...
  - algorithmes : tri, $n^{ième}$ éléments ...
  - lecture / écriture

Des modules connus :

- `iostream`
- `string`

vector

Une classe particulièrement intéressante de la STL est la classe `vector`.

- tableau dynamique
- pas de gestion de la mémoire
- interface type tableau

```
#include <vector>

using namespace std;
```

Classe template

La classe `vector` est une classe template, elle peut s'adapter à tous les types de données. (les templates seront évoqués dans les cours suivants).

```
// creation d'un vector
vector<T> tab;

// Exemple :
vector<int> t_int;
vector<double> t_double;
vector<Matrix> t_mat;
vector<float*> t_point;
```

vector : utilisation

Comme un tableau classique :

```
// creation d'un vector d'entier 100 cases
vector<int> t(100);

// acces au cases
for(int i=0; i<100; i++){
    t[i] = i*i;
}
cout << t[5] << endl;

// recuperer la taille
cout << t.size() << endl;
```

où l'on peut récupérer la taille.

vector : utilisation

- Redimensionner :

```
// creation d'un vector d'entier 100 cases
t.resize(1000);
```

Les éléments présents sont gardés.

- Création et remplissage :

```
// creation d'un vector d'entier 100 cases rempli avec 5.6
vector<double> t2(1000, 5.6);
```

vector : utilisation

- Premier élément :

```
cout << *t.begin() << endl;
```

C'est un iterateur (fonctionne un peu comme un pointeur)

- Fin de vector :

```
t.end(); // Attention pointe juste derriere la derniere case
```

vector : utilisation

- Concaténer :

```
vector<int> t1 (10,2);
vector<int> t2 (30, 100);
t2.insert(t2.end(), t1.begin(), t1.end());
```

C'est un iterateur (fonctionne un peu comme un pointeur)

- Trier :

```
#include <algorithm>

...

std::sort(t.begin(), t.end());
```

Serpent

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

Tron

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