Programmation Orientée Objet: C++

MAM4 - Année 2024-2025

Classes et Objets

Introduction

Nous entrons donc maintenant dans la partie **Programmation Orientée Objet** de ce cours.

Nous avons vu les structures: ce sont des types définis par l'utilisateur qui contiennent à la fois des données, mais aussi des fonctions membres.

En fait, en C++, ces structures sont un cas particulier d'un mécanisme plus général, les Classes.

Introduction

Pourquoi ne pas simplement travailler sur des structures ?

Les structures existent en C++ par souci de compatibilité avec C. Les Classes peuvent être considérées comme les structures mais leur comportement par défaut vis-à-vis de l'encapsulation est différent. De plus, le terme 'structure' ne renvoie pas à la terminologie Objet aussi bien que le terme 'classe', plus commun.

L'encapsulation fait partie intégrante de la POO. Elle permet de masquer certains membres et fonctions membres au monde extérieur.

Dans l'idéal, on ne devrait jamais accéder aux données d'une classe, mais agir sur ses méthodes (fonctions membres).

Déclaration

```
#ifndef __POINT2D_V1_HH__
#define __POINT2D_V1_HH__

class Point2D
{
private:
  float _x;
  float _y;

public:
  void initialise(float, float);
  void affiche();
};

#endif
```

Listing 1: Point2D_V1.hh

Une classe se déclare comme une structure.
Les étiquettes *private* et *public* sont utiles pour définir le niveau de visibilité des membres à l'extérieur de la classe.

Les membres déclarés *private* ne sont pas accessibles par des objets d'un autre type.
Les membres dits publics sont accessibles partout.
Ici, on ne pourra donc plus écrire, dans la fonction *main*:

```
Point2D p;
p._x = 5; //erreur-membre privé!
```

Par convention, les noms des membres privés commencent par '_' (tiret du bas).

private/public

```
#ifndef __POINT2D_V1_HH__
#define __POINT2D_V1_HH__

class Point2D
{
private:
  float _x;
  float _y;

public:
  void initialise(float, float);
  void affiche();
};

#endif
```

Listing 1: Point2D_V1.hh

Par défaut, si on ne spécifie rien, les membres d'une classe ont le statut privé. C'est le contraire pour une structure.

Dans une structure, on peut utiliser le mot-clé *private* pour rendre privés des attributs ou des fonctions.

Constructeurs

```
#ifndef __POINT2D_V2_HH__
#define __POINT2D_V2_HH__

class Point2D
{
private:
  float _x;
  float _y;

public:
  Point2D(float,float);
  void affiche();
};

#endif
```

Listing 2: Point2D_V2.hh

Au lieu d'utiliser une fonction *initialise*, il existe un autre mécanisme en C++: les **constructeurs**.
Les fonctions comme *initialise* ont en effet des inconvénients:

- On ne peut pas forcer l'utilisateur de la classe à l'appeler.
- Elle n'est pas appelée au moment de la création de l'objet ce qui peut être ennuyeux si des opérations doivent être effectuées dès le début de la vie de notre instance.

```
#include<iostream>
#include "Point2D_V2.hh"
using namespace::std;

Point2D::Point2D(float abs, float ord){
  _x = abs, _y = ord;
}
void Point2D::affiche(){
  cout << _x << " : " << _y << endl; }
```

Listing 3: Point2D_V2.cpp

Constructeurs

```
#ifndef __POINT2D_V2_HH__
#define __POINT2D_V2_HH__

class Point2D
{
private:
  float _x;
  float _y;

public:
  Point2D(float,float);
  void affiche();
};

#endif
```

Listing 2: Point2D_V2.hh

Un constructeur est une fonction membre, ne renvoyant rien, qui porte le même nom que la classe.
Elle est appelée lors de la déclaration d'un objet.
Si nous voulions déclarer maintenant un objet de type *Point2D*, nous serions obligés de fournir les deux coordonnées.

Point2D p(3, 4); par exemple, déclare l'objet p de type *Point2D* et appelle dans la foulée le constructeur avec les paramètres 3 et 4.

```
#include<iostream>
#include "Point2D_V2.hh"
using namespace::std;

Point2D::Point2D(float abs, float ord){
  _x = abs, _y = ord;
}
void Point2D::affiche(){
  cout << _x << " : " << _y << endl; }
```

Listing 3: Point2D_V2.cpp

Constructeurs

```
#ifndef __POINT2D_V2_HH__
#define __POINT2D_V2_HH__

class Point2D
{
private:
  float _x;
  float _y;

public:
  Point2D(float,float);
  void affiche();
};

#endif
```

Listing 2: Point2D_V2.hh

Il peut y avoir autant de constructeurs que l'on veut, en fonction des besoins.

Il faudra les distinguer les uns des autres par des paramètres distincts.
On peut également, si cela est nécessaire, placer un constructeur dans la partie *private* de notre classe. Ainsi, son utilisation sera bloquée à l'extérieur de la classe.

```
#include<iostream>
#include "Point2D_V2.hh"
using namespace::std;

Point2D::Point2D(float abs, float ord){
  _x = abs, _y = ord;
}
void Point2D::affiche(){
  cout << _x << " : " << _y << endl; }
```

Listing 3: Point2D_V2.cpp

Destructeur

```
#ifndef __POINT2D_V3_HH__
#define __POINT2D_V3_HH__

class Point2D
{
private:
  float _x, _y;

public:
  Point2D(float,float);
  ~Point2D();
};

#endif
```

Listing 4: Point2D_V3.hh

La classe est déclarée dans un fichier header .hh

On voit les deux données membres déclarées *private*, ainsi qu'un constructeur de la classe ayant deux paramètres de type float.

Le destructeur de la classe porte le même nom que celle-ci précédé du caractère ~ (tilde).

Il est appelé **lors de la destruction de l'instance courante**. Son objectif est de permettre au programmeur de libérer de l'espace mémoire si nécessaire.
De même si des fichiers ont été ouverts ou des connexions (vers des bases de données par exemple) ont été initiées durant la vie de l'instance.

Objets et dynamique

```
#ifndef __POINTND_HH__
#define __POINTND_HH__

class PointND {
private:
  int _n;
  double *_vals;

public:
  PointND(int);
};

#endif
```

Listing 5: PointND.hh

```
#include <iostream>
#include "PointND.hh"

using namespace std;

PointND::PointND(int n) {
  _n = n;
  _vals = new double[n];

  for(int i=0;i<_n;i++)
    _vals[i] = 0;

  cout <<"Constructeur: "
       << "n = " << n
       << endl;
}
```

Listing 6: PointND.cpp

Notre objet contient un pointeur sur *double* qui contiendra toutes les coordonnées du n-point.

Le nombre de dimensions est un entier qui sera aussi un membre privé de notre classe.

Le constructeur initialise les deux variables.

Pour le tableau de valeurs, pas d'autre choix que d'utiliser *new* et donc une allocation dynamique. En effet, le nombre de valeurs étant variable, on ne peut pas utiliser un tableau dont la dimension serait fixée à l'avance.

Objets et dynamique: Le constructeur

```
#ifndef __POINTND_HH__
#define __POINTND_HH__

class PointND {
private:
  int _n;
  double *_vals;

public:
  PointND(int);
};

#endif
```

Listing 5: PointND.hh

```
#include <iostream>
#include "PointND.hh"

using namespace std;

PointND::PointND(int n) {
  _n = n;
  _vals = new double[n];

  for(int i=0;i<_n;i++)
    _vals[i] = 0;

  cout <<"Constructeur: "
       << "n = " << n
       << endl;
}
```

Listing 6: PointND.cpp

Celui-ci initialise $\_n$ avec la valeur entière passée en paramètre. Il alloue aussi la place en mémoire dynamiquement pour les n valeurs du n-point.

Les $\_n$ valeurs sont initialisées à 0.

Objets et dynamique: Le destructeur

```
#ifndef __POINTND_V1_HH__
#define __POINTND_V1_HH__

class PointND {
private:
  int _n;
  double *_vals;

public:
  PointND(int);
  ~PointND();
};
#endif
```

Listing 7: PointND_V1.hh

```
#include <iostream>
#include "PointND_V1.hh"
using namespace std;

PointND::PointND(int n) {
  _n = n;
  _vals = new double[n];

  for(int i=0;i<_n;i++)
    _vals[i] = 0;

  cout << "Constructeur:"
       << " n = " << n
       << endl;
}

PointND::~PointND(){
  delete [] _vals;
  cout << "Appel
      Destructeur"
       << endl; }
```

Listing 8: PointND_V1.cpp

Celui-ci contient un affichage mais ce n'est pas le but premier d'un destructeur.

Celui-ci a pour but de détruire "proprement" l'objet. Il est chargé de libérer la mémoire, de clore des fichiers ou des connexions etc.

Ici, de la mémoire a été allouée dynamiquement par un appel à *new* dans le constructeur. Quand sera-t-elle libérée? C'est au destructeur de se charger de cette tâche.

On devra donc utiliser l'opérateur *delete* sur le tableau *_vals* afin de rendre au système cet espace qu'il pourra réutiliser pour d'autres allocations par exemple.

Objets et dynamique: Le constructeur par copie

```
#ifndef __POINTND_V2_HH__
#define __POINTND_V2_HH__

class PointND {
private:
  int _n;
  double *_vals;

public:
  PointND(int);
  PointND(const PointND&);
  ~PointND();
};
#endif
```

Listing 9: PointND_V2.hh

```
#include <iostream>
#include "PointND_V2.hh"

using namespace std;

PointND::PointND(int n) {
  _n = n;
  _vals = new double[n];
  for(int i=0;i<_n;i++)
    _vals[i] = 0;

  cout << "Constructeur: "
       << "n = " << n << endl;
}

PointND::PointND(const PointND&
    pnd){
  _n = pnd._n;
  _vals = new double[_n];
  for(int i=0;i<_n;i++)
    _vals[i] = pnd._vals[i];

  cout << "Constructeur par copie
       : n=" << _n << endl;
}

PointND::~PointND(){
  delete [] _vals;
  cout << "Appel Destructeur" <<
       endl; }
```

Listing 10: PointND_V2.cpp

Ce constructeur recopie les valeurs de l'objet passé en paramètre par référence. *const* permet d'assurer que celui-ci ne sera pas modifié, ce ne serait pas une copie sinon ···
Que se passe-t-il pour la copie des valeurs?

Ici on alloue la mémoire suffisante pour un tableau de *_n* doubles. On copie les valeurs des éléments de *pnd._vals* dans le tableau *_vals*.

Il ne faut pas faire une copie de pointeurs car le résultat ne serait pas celui espéré, notamment si un objet est modifié ou détruit avant l'autre.

Objets membres

```
#ifndef __CERCLE_HH__
#define __CERCLE_HH__

class Point2D{
  float _x, _y;
public:
  Point2D(float, float);
  Point2D();
};

class Cercle {
private:
  Point2D _centre;
  float _rayon;
};

#endif
```

Listing 11: Cercle.hh

```
#include<iostream>
#include "Cercle.hh"
using namespace::std;

Point2D::Point2D(float
  cout << "Constructeu
       << x << ";"
       << y << ")"
       << endl;
  _x = x, _y = y;
}

Point2D::Point2D()
{
  cout << "Constructeur sans argument de Point2D"
       << endl;
}
```

Listing 12: Point2D_V4.cpp

```
#include "Cercle.hh"
int main()
{
  Cercle c;
  return 0;
}
```

Listing 13: code29.cpp

Soit le code ci-contre: On déclare deux classes, dont la classe *Point2D* ayant deux données membres et deux constructeurs.

La classe *Cercle* utilise le constructeur par défaut de la classe. Elle possède deux données membres dont une instance de la classe *Point2D*.

```
./a.out
Constructeur sans a
```

Objets membres

```
#ifndef __CERCLE_HH__
#define __CERCLE_HH__

class Point2D{
  float _x, _y;
public:
  Point2D(float, float);
  Point2D();
};

class Cercle {
private:
  Point2D _centre;
  float _rayon;
};

#endif
```

Listing 11: Cercle.hh

```
#include<iostream>
#include "Cercle.hh"
using namespace::std;

Point2D::Point2D(
      float x, float y
      ){
  cout << "
        Constructeur
        Point2D("
        << x << ";"
        << y << ")"
        << endl;
  _x = x, _y = y;
}

Point2D::Point2D()
{
  cout << "
        Constructeur
        sans argument
        de Point2D"
        << endl;
}
```

Listing 12: Point2D_V4.cpp

```
#include "Cercle.hh"
int main()
{
  Cercle c;
  return 0;
}
```

Listing 13: code29.cpp

A l'exécution de ce code, on remarque que construire un objet de type *Cercle* entraîne la création d'un objet de type *Point2D*.

```
./a.out
Constructeur sans a
```

Objets membres

```
#ifndef __CERCLE_HH__
#define __CERCLE_HH__

class Point2D{
  float _x, _y;
public:
  Point2D(float, float);
  Point2D();
};

class Cercle {
private:
  Point2D _centre;
  float _rayon;
};

#endif
```

Listing 11: Cercle.hh

```
#include<iostream>
#include "Cercle.hh"
using namespace::std;

Point2D::Point2D(
      float x, float y
      ){
  cout << "
        Constructeur
        Point2D("
        << x << ";"
        << y << ")"
        << endl;
  _x = x, _y = y;
}

Point2D::Point2D()
{
  cout << "
        Constructeur
        sans argument
        de Point2D"
        << endl;
}
```

Listing 12: Point2D_V4.cpp

```
#include "Cercle.hh"
int main()
{
  Cercle c;
  return 0;
}
```

Listing 13: code29.cpp

En fait, le constructeur de *Point2D* est appelé avant celui de *Cercle*.

Ici, on remarque d'ailleurs que le constructeur par défaut de *Cercle* fait appel au constructeur sans argument de *Point2D*.

Pourquoi?

```
./a.out
Constructeur sans a
```

Objets membres

```
#ifndef __CERCLE_HH__
#define __CERCLE_HH__

class Point2D{
  float _x, _y;
public:
  Point2D(float, float);
  Point2D();
};

class Cercle {
private:
  Point2D _centre;
  float _rayon;
};

#endif
```

Listing 11: Cercle.hh

```
#include<iostream>
#include "Cercle.hh"
using namespace::std;

Point2D::Point2D(
      float x, float y
      ){
  cout << "
        Constructeur
        Point2D("
        << x << ";"
        << y << ")"
        << endl;
  _x = x, _y = y;
}

Point2D::Point2D()
{
  cout << "
        Constructeur
        sans argument
        de Point2D"
        << endl;
}
```

Listing 12: Point2D_V4.cpp

```
#include "Cercle.hh"
int main()
{
  Cercle c;
  return 0;
}
```

Listing 13: code29.cpp

Comment faire pour passer des paramètres au constructeur de *Point2D* lorsqu'on crée un objet de type *Cercle* ?

```
./a.out
Constructeur sans a
```

Objets membres

```
#ifndef __CERCLE_V2_HH__
#define __CERCLE_V2_HH__

class Point2D{
  float _x, _y;
public:
  Point2D(float, float);
  Point2D();
};

class Cercle {
private:
  Point2D _centre;
  float _rayon;
public:
  Cercle(float x, float y,
         float r);
};

#endif
```

Listing 14: Cercle_V2.hh

```
#include<iostream>
#include "Cercle_V2.hh"
using namespace::std;

Point2D::Point2D(float x, float y){
  cout << "Constructeur Point2D("
       << x << ";"
       << y << ")"
       << endl;
  _x = x, _y = y;
}

Point2D::Point2D()
{
  cout << "Constructeur sans
       argument de Point2D"
       << endl;
}

Cercle::Cercle(float x, float y,
     float r): _centre(x,y)
{
  _rayon = r;
  cout << "Constructeur de Cercle"
       << endl;
}
```

Listing 15: Point2D_V5.cpp

On modifie le code: on ajoute un constructeur qui possède 3 paramètres, à la classe *Cercle*. On va passer 2 de ces paramètres au constructeur de la classe *Point2D*.

```
#include "Cercle_V2.hh"

int main()
{
  Cercle c(2, 3, 4);
}
```

Listing 16: code30.cpp

Objets membres

```
#ifndef __CERCLE_V2_HH__
#define __CERCLE_V2_HH__

class Point2D{
  float _x, _y;
public:
  Point2D(float, float);
  Point2D();
};

class Cercle {
private:
  Point2D _centre;
  float _rayon;
public:
  Cercle(float x, float y,
         float r);
};

#endif
```

Listing 14: Cercle_V2.hh

```
#include<iostream>
#include "Cercle_V2.hh"
using namespace::std;

Point2D::Point2D(float x, float y){
  cout << "Constructeur Point2D("
       << x << ";"
       << y << ")"
       << endl;
  _x = x, _y = y;
}

Point2D::Point2D()
{
  cout << "Constructeur sans
       argument de Point2D"
       << endl;
}

Cercle::Cercle(float x, float y,
     float r): _centre(x,y)
{
  _rayon = r;
  cout << "Constructeur de Cercle"
       << endl;
}
```

Listing 15: Point2D_V5.cpp

On va utiliser une syntaxe particulière: entre l'entête de la fonction et son corps, on insère ':' et la définition de la donnée membre : *_centre(x, y)*

```
#include "Cercle_V2.hh"

int main()
{
  Cercle c(2, 3, 4);
}
```

Listing 16: code30.cpp

```
./a.out
Constructeur Point2D(2;3
Constructeur de Cercle
```

Objets membres

```
#ifndef __CERCLE_V3_HH__
#define __CERCLE_V3_HH__

class Point2D{
  float _x, _y;
public:
  Point2D(float, float);
};

class Cercle {
private:
  Point2D _centre;
  float _rayon;
public:
  Cercle(Point2D, float);
};

#endif
```

Listing 17: Cercle_V3.hh

```
#include<iostream>
#include "Cercle_V3.hh"
using namespace std;

Point2D::Point2D(float x, float y){
  cout << "Constructeur Point2D("
       << x << ";"
       << y << ")"
       << endl;
  _x = x, _y = y;
}

Cercle::Cercle(Point2D p, float r)
    : _centre(p)
{
  cout << "Constructeur de Cercle"
       << endl;
  _rayon = r;
}
```

Listing 18: Point2D_V6.cpp

Voici un autre constructeur pour *Cercle* en utilisant le type *Point2D*.

La syntaxe est proche de l'exemple précédent, l'objet *Point2D* est construit en premier via l'instruction *_centre(p)*.

Accesseurs & mutateurs

```
#ifndef __OBJET_HH__
#define __OBJET_HH__

class Objet{

private:
 double _v;

public:
 Objet(double);
 ~Objet();
 double getV() const;
 void setV(double d);
};

#endif
```

Listing 19: Objet.hh

```
#include "Objet.hh"

Objet::Objet(double v):_v(v){}

Objet::~Objet() {}

double Objet::getV() const
{
  return _v;
}

void Objet::setV(double v)
{
  _v=v;
}
```

Listing 20: Objet.cpp

On appelle accesseurs et mutateurs (ou modificateurs) des fonctions permettant l'accès à des attributs privés d'une classe.

On les appelle aussi getter et setter en Anglais.

Ces fonctions sont quasi systématiquement définies lors de la création d'une nouvelle classe.

Accesseurs & mutateurs

```
#ifndef __OBJET_HH__
#define __OBJET_HH__

class Objet{

private:
 double _v;

public:
 Objet(double);
 ~Objet();
 double getV() const;
 void setV(double d);
};

#endif
```

Listing 19: Objet.hh

```
#include "Objet.hh"

Objet::Objet(double v):_v(v){}

Objet::~Objet() {}

double Objet::getV() const
{
  return _v;
}

void Objet::setV(double v)
{
  _v=v;
}
```

Listing 20: Objet.cpp

Leur nom correspond généralement au nom de l'attribut précédé de *get* pour les getters et *set* pour les setters.

En général, on déclare les fonctions getters comme étant *const*, comme dans l'exemple ci-contre.

Accesseurs & mutateurs

```
#ifndef __OBJET_HH__
#define __OBJET_HH__

class Objet{

private:
 double _v;

public:
 Objet(double);
 ~Objet();
 double getV() const;
 void setV(double d);
};

#endif
```

Listing 19: Objet.hh

```
#include "Objet.hh"

Objet::Objet(double v):_v(v){}

Objet::~Objet() {}

double Objet::getV() const
{
  return _v;
}

void Objet::setV(double v)
{
  _v=v;
}
```

Listing 20: Objet.cpp

En effet, les fonctions membres déclarées comme *const* permettent à l'utilisateur de cette méthode de savoir que cette fonction ne modifiera pas les champs de l'objet.

Ce sont aussi les seules fonctions que l'on peut appeler sur des objets constants.

Accesseurs & mutateurs

```
#ifndef __OBJET_HH__
#define __OBJET_HH__

class Objet{

private:
 double _v;

public:
 Objet(double);
 ~Objet();
 double getV() const;
 void setV(double d);
};

#endif
```

Listing 19: Objet.hh

```
#include "Objet.hh"

Objet::Objet(double v):_v(v){}

Objet::~Objet() {}

double Objet::getV() const
{
  return _v;
}

void Objet::setV(double v)
{
  _v=v;
}
```

Listing 20: Objet.cpp

On pourra noter également la syntaxe du constructeur qui initialise le champ $\_v$ de l'instance en lui passant la valeur entre parenthèse, comme dans le cas des objets imbriqués.

Il est néanmoins nécessaire d'ajouter les accolades, même si le corps est vide.

Accesseurs & mutateurs

```
#ifndef __OBJET_HH__
#define __OBJET_HH__

class Objet{

private:
 double _v;

public:
 Objet(double);
 ~Objet();
 double getV() const;
 void setV(double d);
};

#endif
```

Listing 19: Objet.hh

```
#include "Objet.hh"

Objet::Objet(double v):_v(v){}

Objet::~Objet() {}

double Objet::getV() const
{
  return _v;
}

void Objet::setV(double v)
{
  _v=v;
}
```

Listing 20: Objet.cpp

```
#include <iostream>
#include "Objet.hh"
using namespace std;

int main()
{
  Objet o(3);

  o.setV(4);
  cout << o.getV()
       << endl;
  return 0;
}
```

Listing 21: main_Objet.cpp

```
g++ main_Objet.cpp Objet.cpp

./a.out
4
```

Le mot-clé this

```
#include <iostream>
using namespace std;

class Objet{
public:
  Objet();
  ~Objet();
};

Objet::Objet() {
  cout << "C: " << this << endl;
}
Objet::~Objet() {
  cout << "D: " << this << endl;
}

int main() {
  Objet o, op;
}
```

Listing 22: main_this.cpp

Chaque fonction membre d'un objet peut recevoir une information supplémentaire.

Elle permet de faire le lien entre les corps des fonctions membres et l'instance courante de la classe.

Il s'agit de *this*.

C'est un pointeur transmis à toutes les fonctions membres et qui pointe vers l'instance courante.

```
 ./a.out
C: 0xffffcbff
C: 0xffffcbfe
D: 0xffffcbfe
D: 0xffffcbff
```

Membres statiques

```
#include <iostream>
using namespace std;

class JeMeCompte
{
public:
  static int compteur; // Déclaration
  JeMeCompte();
  ~JeMeCompte();
};

int JeMeCompte::compteur = 0; // initialisation
                              // de la variable static
JeMeCompte::JeMeCompte(){
  cout << "C: " << compteur++ << endl; }

JeMeCompte::~JeMeCompte(){
  cout << "D: " << compteur++ << endl; }

int main(){
  JeMeCompte nbr;
  cout << "compteur: "<< JeMeCompte::compteur
       << endl; }
```

Listing 23: main_static.cpp

Jusqu'à présent, une donnée membre d'une classe était liée à chaque instance de la classe.

Il est également possible de lier une donnée à la classe elle-même, indépendamment d'éventuelles instances.

Et la valeur de cette donnée est partagée par toutes les instances et par la classe elle-même.

On utilise pour cela le mot clé *static* devant la ou les variable(s) désignée(s) pour ce rôle.

Membres statiques

```
#include <iostream>
using namespace std;

class JeMeCompte
{
public:
  static int compteur; // Déclaration
  JeMeCompte();
  ~JeMeCompte();
};

int JeMeCompte::compteur = 0; // initialisation
                              // de la variable static
JeMeCompte::JeMeCompte(){
  cout << "C: " << compteur++ << endl; }

JeMeCompte::~JeMeCompte(){
  cout << "D: " << compteur++ << endl; }

int main(){
  JeMeCompte nbr;
  cout << "compteur: "<< JeMeCompte::compteur
       << endl; }
```

Listing 23: main_static.cpp

Il reste la question de l'initialisation de cette variable.

Celle-ci ne peut pas être initialisée par un constructeur: en effet celui-ci est intimement lié au cycle de vie d'un objet et donc à une instance elle-même.

On ne peut pas non plus l'initialiser dans la déclaration de la classe. En effet, cela risquerait, en cas de compilation séparée, de réserver plusieurs emplacements en mémoire pour cette donnée.

Membres statiques

```
#include <iostream>
using namespace std;

class JeMeCompte
{
public:
  static int compteur; // Déclaration
  JeMeCompte();
  ~JeMeCompte();
};

int JeMeCompte::compteur = 0; // initialisation
                              // de la variable static
JeMeCompte::JeMeCompte(){
  cout << "C: " << compteur++ << endl; }

JeMeCompte::~JeMeCompte(){
  cout << "D: " << compteur++ << endl; }

int main(){
  JeMeCompte nbr;
  cout << "compteur: "<< JeMeCompte::compteur
       << endl; }
```

Listing 23: main_static.cpp

Il reste donc à l'initialiser hors de la déclaration, au côté de l'initialisation des fonctions membres.

La syntaxe est alors telle que dans l'exemple ci-contre.

On notera l'utilisation de l'opérateur de résolution de portée '::' pour signifier qu'il s'agit bien d'un membre de la classe.

Fonctions membres statiques

```
#include <iostream>
using namespace std;

class JeMeCompte{
public:
  static int compteur;
  JeMeCompte();
  ~JeMeCompte();
  static void AfficheCompteur();
};

int JeMeCompte::compteur = 0;

JeMeCompte::JeMeCompte(){
  cout << "C: " << compteur++ << endl; }
JeMeCompte::~JeMeCompte(){
  cout << "D: " << compteur++ << endl; }
void JeMeCompte::AfficheCompteur(){
  cout << compteur << " objets" << endl;}

int main(){
  JeMeCompte nbr;
  JeMeCompte::AfficheCompteur();  }
```

Listing 24: main_static_V2.cpp

Il est également possible de déclarer des fonctions membres statiques.

Comme les données, elles ne dépendent plus d'une instance mais de la classe elle-même. On peut donc les appeler en dehors de tout objet, comme dans l'exemple ci-contre.

Pour signaler que l'on utilise la fonction de la classe *JeMeCompte*, on utilise l'opérateur ::

Fonction amie

```
#ifndef __OBJET_V2_HH_
#define __OBJET_V2_HH_

class Objet{

private:
  double _v;

public:
  Objet(double _v);
  ~Objet();
  double getV() const;
  void setV(double d);
  friend bool egale(Objet o,
                    Objet p);
};

#endif
```

Listing 25: Objet_V2.hh

```
#include "Objet_V2.hh"
#include<cmath>
Objet::Objet(double v): _v(v) {}

Objet::~Objet() {}

double Objet::getV() const {
  return _v;
}

void Objet::setV(double v) {
  _v=v;
}

bool egale(Objet o, Objet p) {
  return abs(o._v-p._v)<1e-10;
}
```

Listing 26: Objet_V2.cpp

```
#include <iostream>
#include "Objet_V2.hh"
using namespace std;
int main()
{
  Objet o(3);
  o.setV(4);
  cout<< o.getV() << endl;
  cout << boolalpha
       << egale(o,o) << endl;
}
```

Listing 27: main_Objet_V2.cpp

Il existe un autre moyen pour une fonction d'accéder aux membres privés d'une classe. Il s'agit d'une fonction **amie**.

Celle-ci se déclare dans le corps de la classe, précédée du mot clé *friend*.

Elle ne fait pas partie de la classe et ne reçoit donc le pointeur *this* d'aucune instance.

Elle accède, par contre, aux données membres sans l'utilisation des getters ou des setters.

Fonction amie

```
#ifndef __OBJET_V2_HH_
#define __OBJET_V2_HH_

class Objet{

private:
  double _v;

public:
  Objet(double _v);
  ~Objet();
  double getV() const;
  void setV(double d);
  friend bool egale(Objet o,
                    Objet p);
};

#endif
```

Listing 25: Objet_V2.hh

```
#include "Objet_V2.hh"
#include<cmath>
Objet::Objet(double v): _v(v) {}

Objet::~Objet() {}

double Objet::getV() const {
  return _v;
}

void Objet::setV(double v) {
  _v=v;
}

bool egale(Objet o, Objet p) {
  return abs(o._v-p._v)<1e-10;
}
```

Listing 26: Objet_V2.cpp

```
#include <iostream>
#include "Objet_V2.hh"
using namespace std;
int main()
{
  Objet o(3);
  o.setV(4);
  cout<< o.getV() << endl;
  cout << boolalpha
       << egale(o,o) << endl;
}
```

Listing 27: main_Objet_V2.cpp

Son utilisation se justifie quand on recherche un code optimisé. Car l'utilisation des getters et des setters génère du code moins optimisé.

On réservera donc son usage pour des cas particuliers de recherche de performance.

La plupart du temps et, pour un code plus lisible, on utilisera les modificateurs et accesseurs.

Fonction amie

```
#ifndef __OBJET_V2_HH_
#define __OBJET_V2_HH_

class Objet{

private:
  double _v;

public:
  Objet(double _v);
  ~Objet();
  double getV() const;
  void setV(double d);
  friend bool egale(Objet o,
                    Objet p);
};

#endif
```

Listing 25: Objet_V2.hh

```
#include "Objet_V2.hh"
#include<cmath>
Objet::Objet(double v): _v(v) {}

Objet::~Objet() {}

double Objet::getV() const {
  return _v;
}

void Objet::setV(double v) {
  _v=v;
}

bool egale(Objet o, Objet p) {
  return abs(o._v-p._v)<1e-10;
}
```

Listing 26: Objet_V2.cpp

```
#include <iostream>
#include "Objet_V2.hh"
using namespace std;
int main()
{
  Objet o(3);
  o.setV(4);
  cout<< o.getV() << endl;
  cout << boolalpha
       << egale(o,o) << endl;
}
```

Listing 27: main_Objet_V2.cpp

On notera également l'utilisation de *boolalpha* qui est un modificateur de *cout*.

Il permet l'affichage de booléens de manière plus lisible.

Ici, on obtient true/false à la place de 1/0.

Surcharge d'opérateurs

Introduction

Imaginons que nous définissions une classe *Complex* pour gérer les nombres complexes dans un programme.

On va certainement être amené à définir les opérations courantes sur ces nombres. Par exemple, l'addition entre deux nombres complexes pourrait se définir comme une fonction ayant comme prototype:

```
Complex add(const Complex &) const;
```

Pour utiliser cette fonction dans un programme, on écrirait par exemple:

```
Complex c(1, 1), c2(2, 2);
Complex c3 = c.add(c2) ;
```

Introduction

C'est bien, mais on est habitué à une écriture qui nous semble plus naturelle:
On aurait envie d'écrire:

```
Complex c4 = c + c2;
```

Que nous faut-il pour cela ?
L'opérateur '+' existe bien en C++, mais il n'existe que pour des types prédéfinis, de base, tels que *int*, *float*, *double*, etc.
Pour que nous puissions l'utiliser dans le cadre des nombres complexes que nous avons définis, il faudrait le définir dans ce contexte.

Introduction

Le C++ permet de telles définitions supplémentaires. On appelle ce mécanisme la surcharge ou surdéfinition d'opérateurs. Ce mécanisme vient compléter la surcharge de fonctions que nous avons déjà vue.

Il ne faut pas croire qu'il s'agit là de quelque chose de naturel et que tous les langages le permettent. Le C, par exemple, ne permet pas cela.

De plus, il existe une contrainte importante en C++ à cette possibilité. Il n'est pas possible de redéfinir un opérateur qui s'appliquerait à des types de base.
Hors de question, donc, de redéfinir l'addition des entiers.

Mais qui aurait envie de faire cela, au risque de rendre son programme incompréhensible?

Introduction

Presque tous les opérateurs peuvent être redéfinis. Seuls quelques-uns échappent à cette règle.

Ainsi, parmi les opérateurs que nous connaissons déjà, ceux qui suivent ne peuvent pas être redéfinis:

- :: (opérateur de résolution de portée)
- . (opérateur point, pour accéder aux champs d'un objet)
- *sizeof*
- ?: (opérateur ternaire)

Introduction

Les autres peuvent donc prendre une définition différente en fonction du contexte dans lequel ils s'appliquent.

Néanmoins, ils gardent la même priorité et la même associativité que les opérateurs que nous connaissons déjà.

Ainsi, même si nous les redéfinissons, l'opérateur * de la multiplication sera "plus prioritaire" que l'addition $+$.

De même, les opérateurs conservent leur pluralité. Un opérateur unaire le reste, un binaire le restera également.

Un exemple

```
#ifndef __COMPLEX_V1_HH__
#define __COMPLEX_V1_HH__

class Complex
{
private:
 double _real, _imag;

public:
 Complex(double real,
      double imag);
 Complex operator+(const
      Complex &) const;
 void affiche() const;
};

#endif
```

Listing 28: Complex_V1.hh

```
#include <iostream>
#include "Complex_V1.hh"

using namespace std;

Complex::Complex(double real,
     double imag):
  _real(real), _imag(imag) {}

Complex Complex::operator+(const
     Complex &c) const
{
  return Complex(_real + c._real,
       _imag + c._imag);
}

void Complex::affiche() const
{
  cout << "(" << _real << ", "
       << _imag << ")"<< endl;
}
```

Listing 29: Complex_V1.cpp

```
#include <iostream>
#include "Complex_V1.hh"

int main()
{
  Complex c(1,1);
  Complex c2(2,2);
  Complex cres = c + c2;
  cres.affiche();
}
```

Listing 30: main_Complex.cpp

On définit une classe *Complex* ayant deux données privées, de type *double*, destinées à recevoir les parties réelles et imaginaires de nos nombres complexes.

On définit aussi un constructeur pour initialiser ces deux champs.

Un exemple

```
#ifndef __COMPLEX_V1_HH__
#define __COMPLEX_V1_HH__

class Complex
{
private:
 double _real, _imag;

public:
 Complex(double real,
      double imag);
 Complex operator+(const
      Complex &) const;
 void affiche() const;
};

#endif
```

Listing 28: Complex_V1.hh

```
#include <iostream>
#include "Complex_V1.hh"

using namespace std;

Complex::Complex(double real,
      double imag):
  _real(real), _imag(imag) {}

Complex Complex::operator+(const
      Complex &c) const
{
  return Complex(_real + c._real,
       _imag + c._imag);
}

void Complex::affiche() const
{
  cout << "(" << _real << ", "
       << _imag << ")"<< endl;
}
```

Listing 29: Complex_V1.cpp

```
#include <iostream>
#include "Complex_V1.hh"

int main()
{
  Complex c(1,1);
  Complex c2(2,2);
  Complex cres = c + c2;
  cres.affiche();
}
```

Listing 30: main_Complex.cpp

La fonction *affiche* est classique et son but est simplement de produire un affichage des données membres de notre instance.

Un exemple

```
#ifndef __COMPLEX_V1_HH__
#define __COMPLEX_V1_HH__

class Complex
{
private:
 double _real, _imag;

public:
 Complex(double real,
      double imag);
 Complex operator+(const
      Complex &) const;
 void affiche() const;
};

#endif
```

Listing 28: Complex_V1.hh

```
#include <iostream>
#include "Complex_V1.hh"

using namespace std;

Complex::Complex(double real,
     double imag):
  _real(real), _imag(imag) {}

Complex Complex::operator+(const
     Complex &c) const
{
  return Complex(_real + c._real,
       _imag + c._imag);
}

void Complex::affiche() const
{
  cout << "(" << _real << ", "
       << _imag << ")"<< endl;
}
```

Listing 29: Complex_V1.cpp

```
#include <iostream>
#include "Complex_V1.hh"

int main()
{
  Complex c(1,1);
  Complex c2(2,2);
  Complex cres = c + c2;
  cres.affiche();
}
```

Listing 30: main_Complex.cpp

Il reste une dernière fonction membre, appelée *operator+*. C'est cette fonction qui redéfinit l'opérateur + pour nos nombres complexes.

La syntaxe est toujours la même quel que soit l'opérateur.

Un exemple

```
#ifndef __COMPLEX_V1_HH__
#define __COMPLEX_V1_HH__

class Complex
{
private:
 double _real, _imag;

public:
 Complex(double real, double imag);
 Complex operator+(const Complex &) const;
 void affiche() const;
};

#endif
```

Listing 28: Complex_V1.hh

On voit qu'*operator+* est une fonction membre de la classe *Complex*.
Son argument est une référence sur une autre instance de type *Complex*. Celle-ci est déclarée *const*, car cet objet n'est pas appelé à changer lors de notre addition.
De même, la fonction elle-même est déclarée comme *const* car l'instance courante n'est pas amenée à être modifiée lors de l'opération.
Cela nous permet d'utiliser notre addition sur des objets constants, ce qui semble raisonnable.
Enfin, celle-ci renverra un objet de type *Complex*. Car, pour rester cohérent, l'addition de deux nombres complexes est aussi un nombre complexe.

Un exemple

```
#include <iostream>
#include "Complex_V1.hh"

using namespace std;

Complex::Complex(double real, double imag):
  _real(real), _imag(imag) {}

Complex Complex::operator+(const Complex &c) const
{
  return Complex(_real + c._real, _imag + c._imag);
}

void Complex::affiche() const
{
  cout << "(" << _real << ", "
       << _imag << ")"<< endl;
}
```

Listing 29: Complex_V1.cpp

Intéressons-nous à l'implémentation proprement dite:

On voit que notre fonction retourne simplement un nouvel objet de type *Complex* en fixant les deux valeurs passées à son constructeur comme la somme des parties réelles et imaginaires.

Le tout est ensuite renvoyé par valeur en sortie de la fonction.

Commutativité

Si nous voulons définir une opérateur + avec un *Complex* c et un *double*, l'opérateur que nous surchargeons ne peut s'appliquer que dans l'ordre dans lequel on le définit.

$c + 3.5$ n'appellera pas la même fonction que $3.5 + c$.

Car la première porte sur un *Complex* en premier argument et un *double* en second.

Le premier argument de $3.5 + c$ est un *double* et le deuxième un *Complex*.

Commutativité

Ce constat nous amène à deux réflexions.

- Tout d'abord, même si on peut définir une fonction autant de fois qu'on le veut tant que les types des paramètres sont différents, il est quand même désagréable d'avoir à le faire dans ce cas! On verra que ce n'est pas forcément nécessaire car un *double* n'est qu'un complexe particulier et on pourra convertir un *double* en complexe. Ce qui nous ramènera au problème précédent.
- Une opération $3.5 + c$ a comme premier paramètre un *double*. L'opérateur + que nous avions défini dans notre classe *Complex* recevait en effet l'instance en premier argument, car il s'agissait d'une fonction membre de notre classe *Complex*.

Opérateurs & fonctions amies

Nous avons, jusqu'à présent, défini les opérateurs surchargés dans nos classes, en tant que méthode de classe.

Cela n'est pas forcément obligatoire, on peut les définir aussi en tant que fonction amie de notre classe comme nous allons le voir dans l'exemple suivant.

Opérateurs & fonctions amies

```
#ifndef __COMPLEX_V2_HH__
#define __COMPLEX_V2_HH__

class Complex
{
private:
  double _real, _imag;

public:
  Complex(double real, double imag);
  Complex(const Complex &);
  Complex operator+(const Complex &) const;
  friend Complex operator-(const Complex &,
       const Complex &);
  void affiche() const;
};

#endif
```

Listing 31: Complex_V2.hh

On définit maintenant, dans notre classe *Complex*, une fonction **amie** *operator-* qui, comme on l'aura deviné, sera chargée de définir l'opérateur de soustraction '-' pour les nombres complexes.

Celle-ci n'est pas une fonction membre de notre classe, mais une fonction amie. Son implémentation est:

```
Complex operator-(const Complex
      & c1, const Complex & c2)
{
  return Complex(
     c1._real - c2._real,
     c1._imag - c2._imag);
}
```

Opérateurs & fonctions amies

```
#ifndef __COMPLEX_V2_HH__
#define __COMPLEX_V2_HH__

class Complex
{
private:
  double _real, _imag;

public:
  Complex(double real, double imag);
  Complex(const Complex &);
  Complex operator+(const Complex &) const;
  friend Complex operator-(const Complex &,
       const Complex &);
  void affiche() const;
};

#endif
```

Listing 31: Complex_V2.hh

REMARQUE:
Dans cet exemple, il n'est pas nécessaire de passer par une fonction amie, externe à notre classe, car le premier paramètre est du type de la classe (mais ce sera nécessaire dans l'opérateur suivant ou pour l'opérateur d'affichage standard, comme on le verra plus tard)

```
Complex operator-(const Complex
      & c1, const Complex & c2)
{
  return Complex(
     c1._real - c2._real,
     c1._imag - c2._imag);
}
```

Opérateurs & fonctions amies

```
#ifndef __COMPLEX_V2_HH__
#define __COMPLEX_V2_HH__

class Complex
{
private:
  double _real, _imag;

public:
  Complex(double real, double imag);
  Complex(const Complex &);
  Complex operator+(const Complex &) const;
  friend Complex operator-(const Complex &,
       const Complex &);
  friend Complex operator+(double, const
      Complex &);
  void affiche() const;
};

#endif
```

Listing 32: Complex_V3.hh

On souhaite ici définir, dans notre classe *Complex*, une fonction **amie** *operator+*, chargée de définir l'opérateur d'addition entre un *double* et un *Complex*.

Ici on n'a pas d'autre choix que d'utiliser une fonction amie pour l'opération souhaitée, car l'ordre des arguments est important.

Si on avait voulu définir l'opérateur d'addition entre un *Complex* et un *double*, on aurait eu le choix entre utiliser une fonction amie ou une fonction membre.

```
Complex operator+(double d,
      const Complex & c1)
{
  return Complex(d + c1._real,
        c1._imag);
}
```

Opérateur []

```
#ifndef __VECTOR_V1_HH__
#define __VECTOR_V1_HH__

#include <iostream>
using namespace std;

class Vector
{
private :
 int *_val;
 int _n;

public:
 Vector(int n);
 ~Vector();
 int & operator[] (int i);
};

#endif
```

Listing 33: Vector_V1.hh

```
#include "Vector_V1.hh"

Vector::Vector(int n) :
      _n(n){
  _val = new int[n];
}

Vector::~Vector(){
  delete [] _val;
}

int & Vector::operator[]
      (int i)
{
  return _val[i];
}
```

Listing 34: Vector_V1.cpp

```
#include "Vector_V1.hh"

int main()
{
 Vector v(5);

 for (int i=0;i<5;i++)
   v[i] = i;

 for (int i=0;i<5;i++)
   cout << v[i] << " ";
 cout << endl;
 return 0;
}
```

Listing 35: main_Vector_V1.cpp

La notation [] vue, par exemple, pour accéder aux éléments d'un tableau, est un opérateur que l'on peut redéfinir lorsqu'il s'applique à un objet.

Evidemment, son utilisation s'applique particulièrement bien aux objets qui sont la surcouche d'un tableau, comme ici pour la classe *Vector*.

Opérateur []

```
#ifndef __VECTOR_V1_HH__
#define __VECTOR_V1_HH__

#include <iostream>
using namespace std;

class Vector
{
private :
 int *_val;
 int _n;

public:
 Vector(int n);
 ~Vector();
 int & operator[] (int i);
};

#endif
```

Listing 3: Vector_V1.hh

```
#include "Vector_V1.hh"

Vector::Vector(int n) :
      _n(n){
  _val = new int[n];
}

Vector::~Vector(){
  delete [] _val;
}

int & Vector::operator[]
       (int i)
{
  return _val[i];
}
```

Listing 4: Vector_V1.cpp

```
#include "Vector_V1.hh"

int main()
{
 Vector v(5);

 for (int i=0;i<5;i++)
   v[i] = i;

 for (int i=0;i<5;i++)
   cout << v[i] << " ";
 cout << endl;
 return 0;
}
```

Listing 5: main_Vector_V1.cpp

On a déjà parlé des constructeur et destructeur. Leur rôle est simplement ici de gérer le tableau de données - un pointeur sur entier - qui est un membre privé de notre classe.

Celui-ci, bien qu'alloué, n'est pas initialisé lors du constructeur et ses valeurs sont donc considérées comme aléatoires.

Opérateur []

```
#ifndef __VECTOR_V1_HH__
#define __VECTOR_V1_HH__

#include <iostream>
using namespace std;

class Vector
{
private :
 int *_val;
 int _n;

public:
 Vector(int n);
 ~Vector();
 int & operator[] (int i);
};

#endif
```

Listing 3: Vector_V1.hh

```
#include "Vector_V1.hh"

Vector::Vector(int n) :
      _n(n){
  _val = new int[n];
}

Vector::~Vector(){
  delete [] _val;
}

int & Vector::operator[]
      (int i)
{
  return _val[i];
}
```

Listing 4: Vector_V1.cpp

```
#include "Vector_V1.hh"

int main()
{
 Vector v(5);

 for (int i=0;i<5;i++)
   v[i] = i;

 for (int i=0;i<5;i++)
   cout << v[i] << " ";
 cout << endl;
 return 0;
}
```

Listing 5: main_Vector_V1.cpp

Intéressons nous à la surcharge de [].

Que veut-on faire exactement avec cette surcharge?

On souhaite, d'une part, accéder à un élément donné du tableau, pour l'afficher par exemple.

Mais on veut également pouvoir le modifier!

Ces deux cas sont visibles dans la fonction *main*. D'abord une boucle dans laquelle les valeurs accédées sont modifiées et une autre dans laquelle elles ne sont qu'accédées.

Opérateur []

```
#ifndef __VECTOR_V1_HH__
#define __VECTOR_V1_HH__

#include <iostream>
using namespace std;

class Vector
{
private :
 int *_val;
 int _n;

public:
 Vector(int n);
 ~Vector();
 int & operator[] (int i);
};

#endif
```

Listing 3: Vector_V1.hh

```
#include "Vector_V1.hh"

Vector::Vector(int n) :
      _n(n){
  _val = new int[n];
}

Vector::~Vector(){
  delete [] _val;
}

int & Vector::operator[]
      (int i)
{
  return _val[i];
}
```

Listing 4: Vector_V1.cpp

```
#include "Vector_V1.hh"

int main()
{
 Vector v(5);

 for (int i=0;i<5;i++)
   v[i] = i;

 for (int i=0;i<5;i++)
   cout << v[i] << " ";
 cout << endl;
 return 0;
}
```

Listing 5: main_Vector_V1.cpp

En fait, tout réside dans le type de la valeur de retour de notre fonction.

On voit ici qu'elle retourne une référence sur l'élément auquel on veut accéder.

Cela permet, une fois la fonction terminée, de pouvoir modifier cette valeur.

Si nous avions travaillé avec un retour par valeur, nous n'aurions eu qu'une copie de notre valeur et celle-ci n'aurait pas pu être modifiée.

Opérateur <<

```
#ifndef __VECTOR_V2_HH__
#define __VECTOR_V2_HH__

#include <iostream>
using namespace std;

class Vector
{
private :
 int *_val;
 int _n;

public:
 Vector(int n);
 ~Vector();
 int & operator[] (int i);
 friend ostream& operator<<(
    ostream &c, Vector& v);
};

#endif
```

Listing 6: Vector_V2.hh

```
#include "Vector_V2.hh"

Vector::Vector(int n) : _n(n){
  _val = new int[n];
}

Vector::~Vector(){
  delete [] _val;
}

int & Vector::operator[](int i)
{
  return _val[i];
}

ostream& operator<<(ostream &c,
      Vector& v)
{
  for(int i=0; i<v._n; i++)
    c << v[i] << " ";
  c << endl;
  return c;
}
```

Listing 7: Vector_V2.cpp

Un opérateur que l'on peut aussi surcharger avec intérêt est l'opérateur <<, pour l'objet *ostream*.

Il ne peut pas se surcharger comme fonction membre de la classe car le premier opérande est un objet de type *ostream*. Il s'agit d'un objet de flux de sortie, comme *cout* que l'on connaît déjà.

Il peut donc être défini comme fonction amie de la classe.

Sa valeur de retour est aussi un objet de type *ostream*, renvoyé en référence. On fait cela afin de pouvoir utiliser l'opérateur en série.
Ainsi lorsqu'on utilise cet opérateur avec *cout* et un objet de type *Vector*, il est appelé et produit les affichages définis dans le corps de la fonction.

Opérateur <<

```
#ifndef __VECTOR_V2_HH__
#define __VECTOR_V2_HH__

#include <iostream>
using namespace std;

class Vector
{
private :
 int *_val;
 int _n;

public:
 Vector(int n);
 ~Vector();
 int & operator[] (int i);
 friend ostream& operator<<(
    ostream &c, Vector& v);
};

#endif
```

Listing 5: Vector_V2.hh

```
#include "Vector_V2.hh"

Vector::Vector(int n) : _n(n){
  _val = new int[n];
}

Vector::~Vector(){
  delete [] _val;
}

int & Vector::operator[](int i)
{
  return _val[i];
}

ostream& operator<<(ostream &c,
      Vector& v)
{
  for(int i=0; i<v._n; i++)
    c << v[i] << " ";
  c << endl;
  return c;
}
```

Listing 6: Vector_V2.cpp

```
#include <iostream>
#include "Vector_V2.hh"
using namespace std;

int main()
{
  Vector v(5);

  for (int i=0; i<5; i++)
    v[i] = i;

  cout << v;
  return 0;
}
```

Listing 7: main_Vector_V2.cpp

On voit ici l'intérêt de la définition de l'opérateur: la syntaxe dans le *main* pour afficher un objet de la classe devient très simple!

Les foncteurs - Opérateur ()

```
#ifndef __AFFINE_HH__
#define __AFFINE_HH__

class Affine
{
private:
 double _a, _b;
public :
 Affine(double, double);
 double operator()(double x) const;
};

#endif
```

Listing 8: Affine.hh

```
#include "Affine.hh"

Affine::Affine(double a,
        double b): _a(a), _b(b)
{}

double Affine::operator()(double
                x) const
{
  return (_a*x + _b);
}
```

Listing 9: Affine.cpp

```
#include <iostream>
#include "Affine.hh"

using namespace std;

void valeurEn0(const
       Affine & a)
{
 cout << a(0)
      << endl;
}

int main()
{
  Affine a(2, 3);
  valeurEn0(a);
  return 0;
}
```

Listing 10: main_Affine.cpp

Un opérateur qu'il peut être pratique de surcharger est l'opérateur ().

On peut ainsi transformer un objet en fonction et l'utiliser comme tel. Par exemple, ici, on construit une fonction affine *a* sous la forme d'un objet que l'on paramétrise lors de sa construction et on peut l'utiliser, par exemple comme paramètre d'une autre fonction (*valeurEn0*).

Ici, l'exemple est trivial et on aurait pu aussi utiliser un pointeur sur fonction.

Il existe de nombreux autres opérateurs qu'il est possible de surcharger et ce cours ne permet malheureusement pas de les étudier tous.

L'opérateur '=', ainsi que les opérateurs d'incrémentation ($++$ et $--$) peuvent néanmoins faire l'objet d'une étude plus approfondie en raison de leur subtilité.