Programmation Orientée Objet: C++ - Cours 5

M1 IM/MF-MPA - Année 2023-2024

Classes et Objets (suite)

Objets et dynamique

```
#ifndef __POINTND_HH__
#define __POINTND_HH__

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

public:
  PointND(int);
};

#endif
```

Listing 1: 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 2: PointND.cpp

Dans une classe, on peut avoir des pointeurs comme attributs. Dans cet exemple, l'objet contient un pointeur sur *double* qui contiendra toutes les coordonnées du n-point.

La dimension est un entier, 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 taille 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 1: 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 2: 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 3: 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 4: 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 5: 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 6: 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 7: 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 8: Point2D_V4.cpp

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

Listing 9: 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
argument de
Point2D
```

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 7: 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 8: Point2D_V4.cpp

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

Listing 9: 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
argument de
Point2D
```

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 7: 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 8: Point2D_V4.cpp

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

Listing 9: 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*.

```
./a.out
Constructeur sans
argument de
Point2D
```

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 7: 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 8: Point2D_V4.cpp

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

Listing 9: code29.cpp

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

```
./a.out
Constructeur sans
argument de
Point2D
```

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 10: 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 11: 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 12: 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 10: 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 11: 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 12: 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 13: 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 << "C. de Cercle avec
       Point2D" << endl;
  _rayon = r;
}
```

Listing 14: Point2D_V6.cpp

```
#include "Cercle_V3.hh"

int main()
{
  Point2D P(2, 3);
  Cercle c(P, 4);
}
```

Listing 15: code31.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)*.

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

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 16: 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 17: 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 18: 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 16: 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 17: 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 18: 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 16: 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 17: 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 18: 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 16: 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

```cpp
#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 17: 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*.
- On va utiliser une fonction amie pour le deuxième opérateur d'addition.

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+(double, const
      Complex &);
  void affiche() const;
};

#endif
```

Listing 18: Complex_V2.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 19: 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 20: 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 21: 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 19: 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 20: 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 21: 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 19: 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 20: 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 21: 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 19: 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 20: 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 21: 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 22: 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 23: 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 22: 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 23: 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 24: 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 25: 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 26: 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 27: 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.

Les patrons de fonctions

- Nous allons maintenant introduire une fonctionnalité très intéressante de C++: les patrons, ou *template* en Anglais (ce cours utilisera indistinctement les deux appellations).
- Pour comprendre tout l'intérêt de ce concept, il faut se souvenir de la surcharge des fonctions.

  Si l'on voulait introduire une fonction *min* sur les entiers qui renverrait le plus petit de deux entiers passés en paramètres, on créerait cette fonction, mais on ne pourrait pas l'utiliser pour des *float* etc. Il faudrait créer une fonction par type de données que l'on veut comparer.

Un exemple

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

template <typename T>
T minimum(T a, T b)
{
  return (a < b) ? a : b;
}

int main()
{
  int a = 2, b = 3;
  double ad = 5.1, bd = 6.2;

  cout << a << ", " << b << " -> "
       << minimum(a,b) << endl;
  cout << ad << ", " << bd << " -> "
       << minimum(ad,bd) << endl;
  return 0;
}
```

Listing 28: code40.cpp

Dans cet exemple, on définit la fonction *minimum* une fois pour toute et on peut l'utiliser quel que soit le type passé en paramètre.

Concernant la syntaxe, on commence donc par le mot clé *template*. Ensuite, le contenu des chevrons définira le caractère générique de notre fonction.

Ici *typename T* définira donc le type générique *T* qu'on pourra utiliser au sein de notre fonction.

Un exemple

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

template <typename T>
T minimum(T a, T b)
{
  return (a < b) ? a : b;
}

int main()
{
  int a = 2, b = 3;
  double ad = 5.1, bd = 6.2;

  cout << a << ", " << b << " -> "
       << minimum(a,b) << endl;
  cout << ad << ", " << bd << " -> "
       << minimum(ad,bd) << endl;
  return 0;
}
```

Listing 28: code40.cpp

En fait, le compilateur va générer, de manière transparente pour l'utilisateur, autant de fonctions que nécessaire selon les types de paramètres qu'on passera à la fonction.

Il y a un seul bémol à cela en comparaison de la surcharge de fonction: l'algorithme ne varie pas en fonction du type des paramètres.

Paramètres expression

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

template <typename T>
T maxtab(T* arr, unsigned int n)
{
  T tmp = arr[0];
  for (unsigned int i=0; i<n; ++i)
    if (tmp < arr[i])
      tmp = arr[i];
  return tmp;
}

int main()
{
  double tab[6] = {2.0, 3.0, 4.0,
      8.0, 2.0, 1.0};
  double tmax = maxtab(tab, 6);
  cout << "Plus grand element: "
       << tmax << endl;
  return 0;
}
```

Listing 29: code41.cpp

Voici un exemple plus complexe.

On veut une fonction qui retourne le plus grand élément d'un tableau qu'on lui fournit en paramètre.

Il n'y a pas de raison de se limiter aux tableaux d'un type particulier.

En fait, tant qu'une relation de comparaison peut être définie entre deux éléments du tableau, notre algorithme peut fonctionner.

Paramètres expression

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

template <typename T>
T maxtab(T* arr, unsigned int n)
{
  T tmp = arr[0];
  for (unsigned int i=0; i<n; ++i)
    if (tmp < arr[i])
      tmp = arr[i];
  return tmp;
}

int main()
{
  double tab[6] = {2.0, 3.0, 4.0,
      8.0, 2.0, 1.0};
  double tmax = maxtab(tab, 6);
  cout << "Plus grand element: "
       << tmax << endl;
  return 0;
}
```

Listing 29: code41.cpp

On définit donc notre fonction comme une fonction *template*, prenant un pointeur sur type en paramètre ainsi qu'un entier non signé qui contiendra le nombre d'éléments du tableau.

On remarque par ailleurs que des types non "templatés" peuvent entrer comme paramètres d'une fonction *template*, il s'agit de paramètres expression.

L'algorithme ensuite est classique, en prenant soin d'utiliser le type *template* quand c'est nécessaire.

Surdéfinition de fonctions template

```
template <typename T>
T maxtab(T * arr, unsigned int n)
{
  T tmp= arr[0];
  for (unsigned int i=0; i<n; i++)
    if (tmp < arr[i])
      tmp = arr[i];
  return tmp;
}

template <typename T>
T maxtab(T* arr, T* arr2,
         unsigned int n, unsigned
             int n2)
{
  T tmp  = maxtab(arr, n);
  T tmp2 = maxtab(arr2, n2);
  return (tmp < tmp2) ? tmp2 : tmp;
}
```

Listing 30: code42_1.cpp

On peut surcharger une fonction *template* en faisant varier son nombre d'éléments ou le type de ceux-ci.

Ici nous avons surchargé la fonction *maxtab* en donnant la possibilité de renvoyer le plus grand élément de deux tableaux.

Spécialisation

```
template <typename T>
T maxtab(T * arr, unsigned int n)
{
  T tmp= arr[0];
  for (unsigned int i=0; i<n; i++)
    if (tmp < arr[i])
      tmp = arr[i];
  return tmp;
}

string maxtab(string* arr, unsigned int n)
{
  string tmp = to_lower(arr[0]);
  for (unsigned int i=0; i<n; i++)
    {
      if (to_lower(tmp)< to_lower(arr[i]))
        tmp = arr[i];
    }
  return tmp;
}
```

Listing 31: code42_2.cpp

On peut également définir un *template* pour un algorithme général qui est valable quel que soit le type, mais aussi spécialiser une fonction, c'est-à-dire définir un algorithme pour un type particulier.

Ici, dans le cas d'un tableau de *string*, l'opérateur $<$, pour comparer deux *strings*, existe mais est sensible à la casse: on ne veut pas cela ici. On spécialise ici la fonction *maxtab* et on utilise une fonction non standard *to_lower* convertissant une chaîne de caractères en minuscules.

Pour finir

Enfin, il n'est pas forcément évident d'écrire et de spécifier une fonction template. En effet, il faut faire attention aux cas ambigüs - c'est-à-dire où le compilateur ne sait pas s'il doit utiliser une fonction plutôt qu'une autre car les deux conviennent.

Par ailleurs, la règle pour les patrons de fonctions est que le type doit convenir "parfaitement", c'est-à-dire qu'un *const int* n'est pas un *int* etc.