جامعة محمد الأول بوجدة
UNIVERSITÉ MOHAMMED PREMIER OUJDA
ⵜⴰⵙⴷⴰⵡⵜ ⵎⵓⵃⴰⵎⴷ ⵏⵜ ⵎⵣⵡⴰⵔ ⵓⵊⴷⴰ

المدرسة الوطنية للذكاء الاصطناعي والرقمنة - بركان
ÉCOLE NATIONALE DE L'INTELLIGENCE ARTIFICIELLE ET DU DIGITAL - BERKANE
ⵜⴰⵙⴷⴰⵡⵜ ⵏⵜⵜⵣⵔⵉⵡⵉⵏ ⵏⵜⵜⵓⵎⵓⵔⵉⵜ ⵏ ⵜⵣⵔⵓⵜ - ⴱⴻⵔⴽⴰⵏ

PROGRAMMATION ORIENTÉE OBJET EN C++

1ÈRE ANNÉE DU CYCLE INGÉNIEUR – ENIADB

Filière : GI – IRSI – ROC – IA

Prof. Mohamed BOUDCHICHE
Email : m.boudchiche0@gmail.com

Année universitaire 2024-2025

1

08 Les patrons de fonctions

2

LES PATRONS DE FONCTIONS

Nous allons maintenant introduire une fonctionnalité très puissante 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.

3

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 = 2.0, bd = 3.0;

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

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

En fait, le compilateur va générer de manière transparente pour l'utilisateur, autant de fonction qu'il est nécessaire en fonction des types de paramètres qu'on passera à la fonction.

Une seule différence à cela en comparaison de la surcharge de fonction : l'algorithme ne varie pas en fonction du type des paramètres.

4

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 = 2.0, bd = 3.0;

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

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 un type générique T qu'on pourra utiliser au sein de notre fonction.

Prof. Mohamed BOUDCHICHE

5

5

PARAMÈTRES EXPRESSIONS

```
#include <iostream>


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[] = {2.0, 3.0, 4.0, 6.0, 2.0, 1.0};
  double max = maxtab(tab, 6);
  std::cout << "plus grand element : "
            << max << std::endl;
}
```

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.

Prof. Mohamed BOUDCHICHE

6

6

PARAMÈTRES EXPRESSIONS

```
#include <iostream>

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[] = {2.0, 3.0, 4.0, 6.0, 2.0, 1.0};
  double max = maxtab(tab, 6);
  std::cout << "plus grand element : "
            << max << std::endl;
}
```

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 de notre tableau.

On remarque par ailleurs que des types non « templaté » peuvent entrer comme paramètres d'une fonction template, il s'agit de paramètre expressions.

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

7

SURDÉFINITIONS DE FONCTIONS

```
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;
}
```

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

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

8

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;
}

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

On peut également définir un template pour un algorithme général qui est valable quelque 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 char *, l'opérateur < n'aurait pas le sens auquel on s'attendrait : il comparerait les valeurs des pointeurs et non des chaines de caractère. Pour cela, on spécialise la fonction et on utilise la fonction strcmp pour comparer les chaines une à une.

9

LES PATRONS DE FONCTIONS

Enfin, il n'est pas forcément évident d'écrire une et de spécialiser une fonction template. En effet, il faut faire attention aux cas ambigües – c'est-à-dire où le compilateur ne sait pas si 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.

10

09 Les patrons de classe

11

LES PATRONS DES CLASSES

Comme pour les fonctions template, il existe un mécanisme similaire pour les classes.

Bien que semblable aux patrons de fonctions sur de nombreux points, il existe des différences avec les tempate de classe.

On va voir que cela permet d'implémenter un code générique et réutilisable.

12

LE RETOUR DE LA CLASSE VECTOR

```
#ifndef__VECTOR_HPP
#define__VECTOR_HPP
template <typename T>
class Vector {
    private:
        T* _data;
        unsigned int _n;
    public:
        Vector(unsigned int n):_n(n){
                _data = new T[_n];
        }
        ~Vector(){
                if (_data) delete [] _data;
        }
        T& operator[] (unsigned int n){
                return _data[n];
        }
};
#endif
```

On se souvient de la classe Vector que nous avions définie dans ce cours.

Celle-ci bien que déclarant une surcouche « objet » à des tableaux d'entier n'était pas utilisable si nous voulions stocker d'autres types. Il aurait alors fallu tout refaire.

Perte de temps, d'énergie …. Et d'argent !

13

LE RETOUR DE LA CLASSE VECTOR

```
#ifndef__VECTOR_HPP
#define__VECTOR_HPP
template <typename T>
class Vector {
    private:
        T* _data;
        unsigned int _n;
    public:
        Vector(unsigned int n):_n(n){
                _data = new T[_n];
        }
        ~Vector(){
                if (_data) delete [] _data;
        }
        T& operator[] (unsigned int n){
                return _data[n];
        }
};
#endif
```

En pratique, comme on le voit ci contre, on définit les types dans l'entête de la déclaration de la classe, de la même façon que pour les fonctions template.

On remarquera aussi un point important :

A l'inverse des classe que nous déclarons d'habitude, ici tout est dans le même fichier!

En effet, le code, ainsi que la déclaration de la classe ne sont, en somme, qu'une déclaration. Le code n'est vraiment généré qu'à la compilation, à partir de ce template, en fonction des besoins.

14

LE RETOUR DE LA CLASSE VECTOR

```
#ifndef__VECTOR_HPP
#define__VECTOR_HPP
template <typename T>
class Vector {
    private:
        T* _data;
        unsigned int _n;
    public:
        Vector(unsigned int n):_n(n){
                _data = new T[_n];
        }
        ~Vector(){
                if (_data) delete [] _data;
        }
        T& operator[] (unsigned int n){
                return _data[n];
        }
};
#endif
```

En résumé,

Déclaration d'une classe → fichier .hh

Définition d'une classe → fichier .cpp

Déclaration d'un template de classe → fichier .hpp

Il ne s'agit que d'une convention. Ce sera celle adoptée dans ce cours.

15

LE RETOUR DE LA CLASSE VECTOR

```
#ifndef__VECTOR_HPP
#define__VECTOR_HPP
template <typename T>
class Vector {
    private:
        T* _data;
        unsigned int _n;
    public:
        Vector(unsigned int n):_n(n){
                _data = new T[_n];
        }
        ~Vector(){
                if (_data) delete [] _data;
        }
        T& operator[] (unsigned int n){
                return _data[n];
        }
};
#endif
```

Comme on n'a au final que déclaré un template de classe, celui-ci ne se compile pas « séparément ». Il ne sera compilé que si il est inclus dans un fichier de code, et que ce code l'utilise !

16

LE RETOUR DE LA CLASSE VECTOR

```
#ifndef__VECTOR_HPP
#define__VECTOR_HPP
template <typename T>
class Vector {
    private:
            T* _data;
            unsigned int _n;
     public:
            Vector(unsigned int n):_n(n){
                  _data = new T[_n];
            }
            ~Vector(){
                  if (_data) delete [] _data;
            }
            T& operator[] (unsigned int n){
                  return _data[n];
            }
};
#endif
```

```
#include <iostream>
#include "Vector.hpp"

int main()
{
   Vector<double> vect(10);
   for (int i = 0; i < 10; i++)
      vect[i] = i;
   for (int i = 0; i < 10; i++)
      std::cout << vect[i] << std::endl;
}
```

Pour utiliser le patron de classe, on va devoir instancier le type, lors de la déclaration de notre variable.

Ainsi, on créé un objet `vect`, de type `Vector<double>`, soit un `Vector` dont le type sera des `double`.

17

LE RETOUR DE LA CLASSE VECTOR

```
#ifndef__VECTOR_HPP__
#define__VECTOR_HPP__

template <typename T>
class Vector
{
private:
   T* _data;
   unsigned int _n;
public:
   Vector(unsigned int n):_n(n){
      _data = new T[_n];
   }
   ~Vector(){
      if (_data) delete [] _data;
   }
   T& operator[] (unsigned int n){
      return _data[n];
   }
};
#endif
```

```
#include <iostream>
#include "Vector.hpp"

int main()
{
   Vector<double> vect(10);
   for (int i = 0; i < 10; i++)
      vect[i] = i;
   for (int i = 0; i < 10; i++)
      std::cout << vect[i] << std::endl;
}
```

Et pour le compiler ?

On ne compile que le fichier contenant le main, car c'est au final le seul fichier cpp de notre programme ici.

18

LE RETOUR DE LA CLASSE VECTOR

```
#ifndef__VECTOR_HPP__
#define__VECTOR_HPP__

template <typename T>
class Vector
{
private:
   T* _data;
   unsigned int _n;
public:
   Vector(unsigned int n):_n(n){
      _data = new T[_n];
   }
   ~Vector(){
      if (_data) delete [] _data;
   }
   T& operator[] (unsigned int n){
      return _data[n];
   }
};
#endif
```

```
#include <iostream>
#include "Vector.hpp"

int main()
{
   Vector<double> vect(10);
   for (int i = 0; i < 10; i++)
      vect[i] = i;
   for (int i = 0; i < 10; i++)
      std::cout << vect[i] << std::endl;
}
```

Et pour le compiler ?

On ne compile que le fichier contenant le main, car c'est au final le seul fichier cpp de notre programme ici.

Prof. Mohamed BOUDCHICHE

19

19

TYPES EXPRESSION

```
#ifndef__MATRICE_HPP__
#define__MATRICE_HPP__
#include <iostream>
using namespace std;
template <typename T, int N, int M>
class Matrice
{
  private:
    T _datas[N][M];  int _n; int _m;
  public:
    Matrice():_n(N), _m(M){}
    T* operator[] (int i){ return _datas[i]; }
    int getN() const { return _n; }
    int getM() const { return _m; }
    friend ostream& operator <<(ostream& o, Matrice& m) {
      for (int i = 0; i < m._n; ++i){
        for (int j = 0; j < m._m; ++j)
          o << m._datas[i][j] <<" ";
        o << endl;
      }
      return o;
    }
};
#endif
```

```
#include <iostream>
#include "Matrice.hpp"

int main()
{
  Matrice<double, 2, 2> mat;

  mat[0][0] = 0;
  mat[0][1] = 1;
  mat[1][0] = 1;
  mat[1][1] = 0;

  cout << mat << endl;
}
```

On peut aussi définir des types expressions. Il s'agit de donner des valeurs à travers la déclaration du template à des constantes présentes dans le code déclaré.
Ici, on définit une Matrice 2x2 dans le main, au moment de la déclaration de la variable mat.
Le compilateur va alors générer à partir du patron que nous avons déclaré une matrice dont les données seront un tableau de 2x2 double.
L'avantage est qu'on ne passe pas par un mécanisme de type allocation dynamique, toutes les données étant stockées dans l'instance courante.
Mais la dimension de celle-ci est figé et ne changera pas de manière dynamique à l'exécution.

Prof. Mohamed BOUDCHICHE

20

20

TYPES EXPRESSION

```
#ifndef__MATRICE_HPP__
#define__MATRICE_HPP__
#include <iostream>
using namespace std;
template <typename T, int N, int M>
class Matrice
{
  private:
    T _datas[N][M];  int _n; int _m;
  public:
    Matrice():_n(N), _m(M){}
    T* operator[] (int i){ return _datas[i]; }
    int getN() const { return _n; }
    int getM() const { return _m; }
    friend ostream& operator <<(ostream& o, Matrice& m) {
      for (int i = 0; i < m._n; ++i){
        for (int j = 0; j < m._m; ++j)
        o << m._datas[i][j] <<" ";
        o << endl;
      }
      return o;
    }
};
#endif
```

```
#include <iostream>
#include "Matrice.hpp"

int main()
{
  Matrice<double, 2, 2> mat;

  mat[0][0] = 0;
  mat[0][1] = 1;
  mat[1][0] = 1;
  mat[1][1] = 0;

  cout << mat << endl;
}
```

Ici, on surcharge l'opérateur [] pour pouvoir accéder aux éléments de notre matrice.
On remarquera que cela doit se faire en deux temps.
En fait on surcharge [] pour accéder aux tableaux représentant les lignes. Ensuite on accédera aux colonnes en utilisant l'opérateur [] habituel sur un tableau.

21

TYPES EXPRESSION

```
#ifndef__MATRICE_HPP__
#define__MATRICE_HPP__
#include <iostream>
using namespace std;
template <typename T, int N, int M>
class Matrice
{
  private:
    T _datas[N][M];  int _n; int _m;
  public:
    Matrice():_n(N), _m(M){}
    T* operator[] (int i){ return _datas[i]; }
    int getN() const { return _n; }
    int getM() const { return _m; }
    friend ostream& operator <<(ostream& o, Matrice& m) {
      for (int i = 0; i < m._n; ++i){
        for (int j = 0; j < m._m; ++j)
        o << m._datas[i][j] <<" ";
        o << endl;
      }
      return o;
    }
};
#endif
```

```
#include <iostream>
#include "Matrice.hpp"

int main()
{
  Matrice<double, 2, 2> mat;

  mat[0][0] = 0;
  mat[0][1] = 1;
  mat[1][0] = 1;
  mat[1][1] = 0;

  cout << mat << endl;
}
```

La surcharge de l'opérateur << pour l'affichage se fait quant à lui à l'aide d'une fonction amie.
On remarque que ici, dans le cadre des patrons de classe, la fonction amie est directement inclue dans le code de la classe.

22

LES PATRONS DE CLASSE

Les patrons de classe sont un outil très puissant et largement utilisés par les développeurs pour créer de nouvelles librairies.

On verra dans le cadre de ce cours la librairie STL pour Standard Template Library qui définie ainsi de nombreux outils rapides et puissants.

La librairie Boost, célèbre également, est une librairie basée sur les template.

En maths, par exemple, la librairie Eigen++ (http://eigen.tuxfamily.org/) est une librairie pour l'algèbre linéaire et des algorithmes très optimisés qui lui sont associés.

23

LES PATRONS DE CLASSE

La notion de template est plus vaste que celle abordée dans ce cours.

Ainsi, on n'a pas abordé la spécialisation de classe template, ou la spécialisation partielle.

Nous ne parlerons pas non plus de meta-programmation ou de template récursifs.

Bien qu'utiles et intéressantes, ces notions sortent de l'objectif de ce cours qui est une introduction au C++.

24

10 Héritage

25

25

L'HÉRITAGE

La notion d'héritage en programmation orienté objet est une notion fondamentale. Il s'agit de créer de nouvelles classes, de nouveaux types, en se basant sur des classes déjà existantes.

On pourra alors non seulement hériter, utiliser leurs capacités, leurs données et leurs fonctions, mais aussi étendre ces capacités. Il s'agit encore ici de ne pas écrire du code qui existe déjà mais de l'utiliser et de l'étendre sans avoir à modifier quelque chose qui existe et qui fonctionne déjà.

On pourra ainsi écrire une classe dérivant d'une autre classe, mais aussi plusieurs classes héritant d'une autre classe.

De même une classe peut hériter d'une classe qui peut elle-même hériter d'une classe etc.

Prof. Mohamed BOUDCHICHE

26

26

EXEMPLE

```
#ifndef__FORME_HH_
#define__FORME_HH_
#include <string>
class Forme{
      private:
            std::string _nom;
       public:
            void setNom(const std::string&);
};
#endif
```

Forme.hh

```
#include "Forme.hh »
void Forme::setNom(const std::string& nom){
        _nom = nom;
}
```

Forme.cpp

On commence par écrire une classe très simple, Forme, qui ne contient qu'une chaine qui contiendra le nom de notre forme.

Une fonction sera chargée d'initialiser notre chaine.

Pour l'instant , on ne définie pas de constructeur ni de destructeur.

EXEMPLE

```
#ifndef__FORME_HH_
#define__FORME_HH_
#include <string>
class Forme{
      private:
            std::string _nom;
       public:
            void setNom(const std::string&);
};
#endif
```

Forme.hh

On va oublier maintenant le code de la fonction setNom. On retiendra juste que celui- ci , comme son nom l'indique, sert à initialiser la chaine donnée membre.

En fait, pour hériter d'une classe, nous n'avons besoin que de sa déclaration , c'est-à-dire du fichier header, et du code de la classe compilé (en gros un fichier .o).

EXEMPLE

```
#ifndef___FORME_HH___
#define___FORME_HH___

#include <string>
class Forme
{
private:
   std::string _nom;

public:
   void setNom(const std::string&);
};

#endif
```

Forme.hh

```
#ifndef_____ROND_HH
#define_____ROND_HH
#include "Forme.hh"

class Rond: public Forme
{
private:
   double _diametre;

public:
   void setDiametre(double);
};
#endif
```

Rond.hh

```
#include "Forme.hh"
#include "Rond.hh"

void Rond::setDiametre(double d)
{
_diametre = d;
}
```

Rond.cpp

On va donc maintenant créer une première classe fille de la classe Forme.

A savoir la classe Rond.

On remarque la ligne :

class Rond: public Forme

Dans la déclaration de la classe rond. Les « : » suivi du public Forme signifie que Rond hérite de la classe Forme.

Le mot clé « public » ici sera expliqué dans la suite de ce cours.

Prof. Mohamed BOUDCHICHE

29

29

EXEMPLE

```
#ifndef___FORME_HH___
#define___FORME_HH___

#include <string>
class Forme
{
private:
   std::string _nom;

public:
   void setNom(const std::string&);
};

#endif
```

Forme.hh

```
#ifndef_____ROND_HH
#define_____ROND_HH
#include "Forme.hh"

class Rond: public Forme
{
private:
   double _diametre;

public:
   void setDiametre(double);
};
#endif
```

Rond.hh

```
#include "Forme.hh"
#include "Rond.hh"

void Rond::setDiametre(double d)
{
_diametre = d;
}
```

Rond.cpp

```
#ifndef_____CARRE_HH
#define_____CARRE_HH
#include "Forme.hh"
class Carre: public Forme
{
private:
   double _longueur;
public:
   void setLongueur(double);
};
#endif
```

Carre.hh

```
#include "Forme.hh"
#include "Carre.hh"

void Carre::setLongueur(double l)
{
   _longueur = l;
}
```

Carre.cpp

On déclare de même une classe Carre, héritant également de la classe Forme.

Ces deux classes ont chacune leurs spécificités, comme on peut le voir, le rond ayant un diamètres, le carré une longueur.

Prof. Mohamed BOUDCHICHE

30

30

EXEMPLE

```
#ifndef__FORME_HH__
#define__FORME_HH__

#include <string>
class Forme
{
private:
   std::string _nom;

public:
   void setNom(const std::string&);
};

#endif
```

Forme.hh

```
#ifndef_____ROND_HH
#define_____ROND_HH
#include "Forme.hh"

class Rond: public Forme
{
private:
   double _diametre;

public:
   void setDiametre(double);
};
#endif
```

Rond.hh

```
#include "Forme.hh"
#include "Rond.hh"

void Rond::setDiametre(double d)
{
_diametre = d;
}
```

Rond.cpp

```
#ifndef_____CARRE_HH
#define_____CARRE_HH
#include "Forme.hh"
class Carre: public Forme
{
private:
   double _longueur;
public:
   void setLongueur(double);
};
#endif
```

Carre.hh

```
#include "Forme.hh"
#include "Carre.hh"

void Carre::setLongueur(double l)
{
   _longueur = l;
}
```

Carre.cpp

```
#include "Forme.hh"
#include "Carre.hh"
#include "Rond.hh"

int main()
{
   Carre  c;
   Rond   r;
   Forme f;
   c.setNom("carre");
   c.setLongueur(5.0);
   r.setNom("rond");
   r.setDiametre(3);
   f.setNom("forme générale.");
}
```

main.cpp

Dans la fonction main, on déclare une variable de chaque type, et on peut, sur les classes filles, appeler des fonctions de la classe Forme. L'inverse, évidemment, n'est pas possible !

31

EXEMPLE

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
class Forme
{
private:
   std::string _nom;

public:
   void setNom(const std::string&);
   void affiche();

};

#endif
```

Forme.hh

```
#ifndef __ROND_HH__
#define __ROND_HH__
#include "Forme.hh"

class Rond: public Forme
{
private:
   double _diametre;

public:
   void setDiametre(double);
};
#endif
```

Rond.hh

```
#include "Forme.hh"
#include "Rond.hh"

void Rond::setDiametre(double d)
{
_diametre = d;
}
```

Rond.cpp

```
#ifndef __CARRE_HH__
#define __CARRE_HH__
#include "Forme.hh"
class Carre: public Forme
{
private:
   double _longueur;
public:
   void setLongueur(double);
};
#endif
```

Carre.hh

```
#include "Forme.hh"
#include "Carre.hh"

void Carre::setLongueur(double l)
{
   _longueur = l;
}
```

Carre.cpp

```
void Forme::affiche()
{
   std::cout << "Je suis de type " <<  _nom << std::endl;
}
```

Forme.cpp

On définie une fonction affiche dans la classe Forme.
Celle-ci se contente d'afficher le nom de la forme.
Cette fonction est alors utilisable par toutes les classes filles, qui afficheront également ce que contient leur donnée `_nom` héritée de classe Forme.
On a donc modifié les fonctionnalité de 3 classes en en modifiant qu'une seule. Pas mal !
Mais, l'affichage ne prend pas en compte la spécificité de chaque classe…

32

EXEMPLE

Forme.hh

```
#ifndef___FORME_HH___
#define___FORME_HH___

#include <string>
class Forme
{
private:
   std::string _nom;

public:
   void setNom(const std::string&);
   void affiche();

};

#endif
```

Rond.hh

```
#ifndef_____ROND_HH
#define_____ROND_HH
#include "Forme.hh"

class Rond: public Forme
{
private:
   double _diametre;

public:
   void setDiametre(double);
};
#endif
```

Carre.hh

```
#ifndef_____CARRE_HH
#define_____CARRE_HH
#include "Forme.hh"
class Carre: public Forme
{
private:
   double _longueur;
public:
   void setLongueur(double);
};
#endif
```

Rond.cpp

```
#include "Forme.hh"
#include "Rond.hh"

void Rond::setDiametre(double d)
{
_diametre = d;
}
```

Carre.cpp

```
#include "Forme.hh"
#include "Carre.hh"

void Carre::setLongueur(double l)
{
   _longueur = l;
}
```

Forme.cpp

```
void Forme::affiche()
{
   std::cout << "Je suis de type " <<  _nom << std::endl;
}
```

main.cpp

```
#include "Forme.hh"
#include "Carre.hh"
#include "Rond.hh"

int main()
{
   Carre  c;
   Rond   r;
   Forme f;
   c.setNom("carre");
   c.setLongueur(5.0);
   r.setNom("rond");
   r.setDiametre(3);
   f.setNom("forme générale.");
   f.affiche();
   c.affiche();
   r.affiche();
}
```

```
$ ./a.exe
Je suis de type forme générale.
Je suis de type carre
Je suis de type rond
```


33

REDÉFINITION D'UNE FONCTION HÉRITÉE

Forme.hh

```
#ifndef___FORME_HH___
#define___FORME_HH___

#include <string>
class Forme
{
private:
   std::string _nom;

public:
   void setNom(const std::string&);
   void affiche();

};

#endif
```

Rond.hh

```
#ifndef_____ROND_HH
#define_____ROND_HH
#include "Forme.hh"
class Rond: public Forme
{
private:
   double _diametre;
public:
   void setDiametre(double);
   void affiche();
};
#endif
```

Carre.hh

```
#ifndef_____CARRE_HH
#define_____CARRE_HH
#include "Forme.hh"
class Carre: public Forme
{
private:
   double _longueur;
public:
   void setLongueur(double);
   void affiche();
};
#endif
```

Rond.cpp

```
#include "Forme.hh"
#include "Rond.hh"

void Rond::setDiametre(double d
{
_diametre = d;
}
void Rond::affiche()
{
   std::cout << "Mon diamètre est de
"
   << _diametre << std::endl;
}
```

Carre.cpp

```
#include "Forme.hh"
#include "Carre.hh"

void Carre::setLongueur(double l)
{
   _longueur = l;
}
void Carre::affiche()
{
   std::cout << "Ma longueur est de "
   << _longueur << std::endl;
}
```

Pour afficher les spécificité de chaque enfant, on va déclarer puis définir une fonction affiche dans chaque classe fille…

```
Je suis de type forme générale.
Ma longueur est de 5
Mon diamètre est de 3
```

En créant une fonction affiche dans la classe fille, on masque la fonction affiche de la classe mère.
On a donc bien une fonction affiche par classe, mais si on voulait aussi le nom de la forme dans les classe fille, il faudrait aussi l'écrire ??
C'est pas très orienté objet ca …

34

REDÉFINITION D'UNE FONCTION HÉRITÉE

Forme.hh

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
class Forme
{
private:
   std::string _nom;

public:
   void setNom(const std::string&);
   void affiche();

};

#endif
```

Rond.hh

```
#ifndef _____ROND_HH
#define _____ROND_HH
#include "Forme.hh"
class Rond: public Forme
{
private:
   double _diametre;
public:
   void setDiametre(double);
   void affiche();
};
#endif
```

Rond.cpp

```
#include "Forme.hh"
#include "Rond.hh"

void Rond::setDiametre(double d)
{
_diametre = d;
}
void Rond::affiche()
{
Forme::affiche();
std::cout << "Mon diamètre est de "
   << _diametre << std::endl;
}
```

Carre.hh

```
#ifndef _____CARRE_HH
#define _____CARRE_HH
#include "Forme.hh"
class Carre: public Forme
{
private:
   double _longueur;
public:
   void setLongueur(double);
   void affiche();
};
#endif
```

Carre.cpp

```
#include "Forme.hh"
#include "Carre.hh"

void Carre::setLongueur(double l)
{
   _longueur = l;
}
void Carre::affiche()
{
Forme::affiche();
std::cout << "Ma longueur est de "
   << _longueur << std::endl;
}
```

On va plutôt essayer d'appeler dans la fonction affiche fille, la fonction affiche parente.
Si on écrit juste affiche() dans la fonction affiche de la classe fille, le compilateur va croire à un appel récursif de la fonction.
Il faut donc distinguer la fonction de la classe Forme avec la fonction de la classe fille, Rond ou Carre.
Pour cela, on va utiliser l'opérateur :: qui va nous permettre de se placer dans le contexte « parent » quand on le désirera.

Prof. Mohamed BOUDCHICHE

35

35

REDÉFINITION D'UNE FONCTION HÉRITÉE

Forme.hh

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
class Forme
{
private:
   std::string _nom;

public:
   void setNom(const std::string&);
   void affiche();

};

#endif
```

Rond.hh

```
#ifndef _____ROND_HH
#define _____ROND_HH
#include "Forme.hh"
class Rond: public Forme
{
private:
   double _diametre;
public:
   void setDiametre(double);
   void affiche();
};
#endif
```

Rond.cpp

```
#include "Forme.hh"
#include "Rond.hh"

void Rond::setDiametre(double d)
{
_diametre = d;
}
void Rond::affiche()
{
Forme::affiche();
std::cout << "Mon diamètre est de "
   << _diametre << std::endl;
}
```

Carre.hh

```
#ifndef _____CARRE_HH
#define _____CARRE_HH
#include "Forme.hh"
class Carre: public Forme
{
private:
   double _longueur;
public:
   void setLongueur(double);
   void affiche();
};
#endif
```

Carre.cpp

```
#include "Forme.hh"
#include "Carre.hh"

void Carre::setLongueur(double l)
{
   _longueur = l;
}
void Carre::affiche()
{
Forme::affiche();
std::cout << "Ma longueur est de "
   << _longueur << std::endl;
}
```

```
$ ./a.exe
Je suis de type forme générale.

Je suis de type carre
Ma longueur est de 5

Je suis de type rond
Mon diamètre est de 3
```

Nous avons maintenant un affichage adapté en fonction de, si on appelle la fonction affiche d'une classe Forme, Rond ou Carre, et ce, sans réécrire la fonction affiche de la classe Forme.
Une fois c'est suffisant !

Prof. Mohamed BOUDCHICHE

36

36

REDÉFINITION D'UNE FONCTION HÉRITÉE

Forme.hh

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
class Forme
{
private:
   std::string _nom;

public:
   void setNom(const std::string&);
   void affiche();

};

#endif
```

Rond.hh

```
#ifndef _____ROND_HH
#define _____ROND_HH
#include "Forme.hh"
class Rond: public Forme
{
private:
   double _diametre;
public:
   void setDiametre(double);
   void affiche();
};
#endif
```

Rond.cpp

```
#include "Forme.hh"
#include "Rond.hh"

void Rond::setDiametre(double d)
{
_diametre = d;
}
void Rond::affiche()
{
Forme::affiche();
std::cout << "Mon diamètre est de "
   << _diametre << std::endl;
}
```

Carre.hh

```
#ifndef _____CARRE_HH
#define _____CARRE_HH
#include "Forme.hh"
class Carre: public Forme
{
private:
   double _longueur;
public:
   void setLongueur(double);
   void affiche();
};
#endif
```

Carre.cpp

```
#include "Forme.hh"
#include "Carre.hh"

void Carre::setLongueur(double l)
{
   _longueur = l;
}
void Carre::affiche()
{
Forme::affiche();
std::cout << "Ma longueur est de "
   << _longueur << std::endl;
}
```

```
$ ./a.exe
Je suis de type forme générale.

Je suis de type carre
Ma longueur est de 5

Je suis de type rond
Mon diamètre est de 3
```

Et si on avait voulu appeler la fonction affiche de Rond héritée de Forme, et non la fonction affiche redéfinie ?

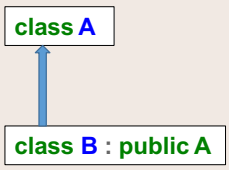
Il faut un peu modifier la syntaxe de l'appel de la fonction :
r.Forme::affiche();

SURCHARGE ET REDÉFINITION

On fera attention sur un point :

Une fonction membre redéfinie dans une classe masque automatiquement les fonctions héritée, même si elles ont des paramètres différents. La recherche d'un symbole dans une classe se fait uniquement au niveau de la classe, si elle échoue, la compilation s'arrête en erreur, même si une fonction convenait dans une classe parente.

CONSTRUCTEURS & DESTRUCTEURS



Soit une classe A, et une classe B héritant de A.

On va maintenant s'intéresser à la construction et à la destruction de la classe B et à son lien avec la classe A.

Pour construire la classe B, le compilateur va devoir d'abord créer la classe A. Il va donc appeler un constructeur de la classe A. Puis il va appeler celui de B.

Dans le cas où il n'y a pas de paramètres au constructeur de A, ou dans le cas d'un constructeur par défaut, il n'y a rien à faire celui-ci est appelé automatiquement.

Les destructeurs, eux, sont appelés dans le sens inverse des appels des destructeurs. C'est-à-dire que B sera détruite avant A.

39

CONSTRUCTEURS & DESTRUCTEURS

```
class A
{
private:
   int _a;
public:
   A(int);
};

class B : public A
{
private:
   int _b;
public:
   B(int, int);
};
```

```
A::A(int a)
{
   _a = a;
}

B::B(int a, int b) : A(a)
{
   _b = b;
}
```

On va rencontrer une difficulté si on doit fournir des paramètres au constructeur de A. En effet, à priori, les paramètres fournis au constructeur de B sont sensés être utilisés par lui. Or le constructeur de A doit être appelé avant. On va avoir recours à la même syntaxe que pour les objets membres lors de la définition du constructeur de B.

40

CONTRÔLE DES ACCÈS

Nous avons déjà vu private et public comme statuts possibles pour une donnée ou une fonction membre. Il en existe en fait une troisième liée à la notion d'héritage : **protected**.

On rappelle d'abord que :

→ private : le membre n'est accessible qu'aux fonctions membres et aux fonctions amies d'une classe.

→ public : le membre est accessible aux fonctions membres et fonctions amies, mais également à l'utilisateur de la classe.

41

CONTRÔLE DES ACCÈS

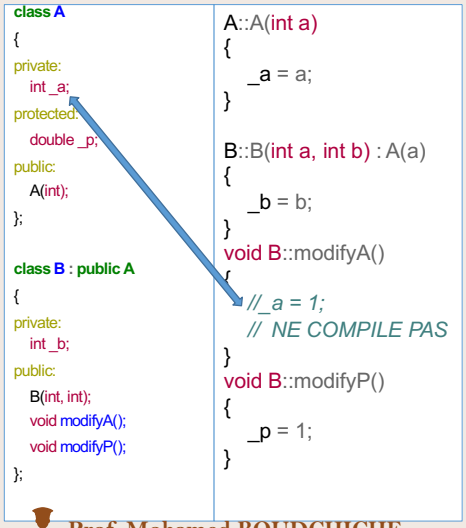
Le mot clé **protected** va quant à lui permettre de protéger une donnée ou une fonction membre d'un usage utilisateur, mais le membre reste accessible à partir de fonctions membres de classes dérivées.

Il constitue donc un intermédiaire entre le concepteur d'une classe, qui a tout pouvoir sur elle, et un « simple » utilisateur de celle-ci.

Il va permettre à un développeur qui souhaite étendre les fonctionnalités d'une classe d'avoir plus de pouvoirs qu'un utilisateur extérieur de la classe. Il aurait été obligé sinon de passer par « l'interface de la classe » c'est-à-dire les fonctions membres « accesseur » et « modificateur », au risque d'une baisse de performance et de praticité.

42

CONTRÔLE DES ACCÈS



```
class A
{
private:
   int _a;
protected:
   double _p;
public:
   A(int);
};

class B : public A
{
private:
   int _b;
public:
   B(int, int);
   void modifyA();
   void modifyP();
};
```

```
A::A(int a)
{
   _a = a;
}

B::B(int a, int b) : A(a)
{
   _b = b;
}
void B::modifyA()
{
   //_a = 1;
   // NE COMPILE PAS
}
void B::modifyP()
{
   _p = 1;
}
```

On voit donc ici que la variable _p déclarée en protected dans la classe A est accessible depuis la classe B. La variable _a, privée, reste inaccessible et un accès depuis B ne compilera pas.

DÉRIVATION PUBLIQUE & DÉRIVATION PRIVÉE

Depuis le début de ce cours sur l'héritage, nous avons déclaré qu'une classe dérivée d'une autre classe de la façon suivante :

class B : public A

En écrivant le mot clé public ici, nous avons en fait déclaré une dérivation publique.

Une dérivation publique permet aux utilisateurs d'une classe dérivée d'accéder aux membres publiques de la classe parente, comme si elles faisaient parties de la classe fille.

DÉRIVATION PUBLIQUE & DÉRIVATION PRIVÉE

```
class A
{
private:
   int _a;
protected:
   double _p;
public:
   A(int);
   void setA(int);
};

class B : private A
{
public:
   B(int);
};
class C : public A
{
public:
   C(int);
};
```

```
#include "A.hh"

A::A(int a)
{
   _a = a;
}
void A::setA(int a)
{
   _a = a;
}

B::B(int a) : A(a) {}
C::C(int a) : A(a) {}
```

```
#include "A.hh"

int main()
{
B b(5);
C c(5);

b.setA(6);
c.setA(6);
}
```

```
 g++ A.cpp  main.cpp
In file included from main.cpp:1:0:
A.hh: Dans la fonction ‘int main()’:
A.hh:10:10: erreur : ‘void A::setA(int)’ is inaccessible
     void setA(int);
          ^
main.cpp:8:10: erreur : à l'intérieur du contexte
  b.setA(6);
          ^
main.cpp:8:10: erreur : ‘A’ is not an accessible base of ‘B’
```

La classe B hérite en privé de la classe A. Elle masque alors son héritage à l'extérieur de la classe. Un utilisateur ne peut pas accéder à partir de B à des membres même publics de la classe A. La classe C par contre, hérite publiquement de la classe A, et on peut utiliser les membres publiques de la classe A de l'extérieur.

45

DÉRIVATION PUBLIQUE & DÉRIVATION PRIVÉE

```
class A
{
private:
   int _a;
protected:
   double _p;
public:
   A(int);
   void setA(int);
};

class B : private A
{
public:
   B(int);
};
class C : public A
{
public:
   C(int);
};
```

```
#include "A.hh"

A::A(int a)
{
   _a = a;
}
void A::setA(int a)
{
   _a = a;
}

B::B(int a) : A(a) {}
C::C(int a) : A(a) {}
```

```
#include "A.hh"

int main()
{
B b(5);
C c(5);

b.setA(6);
c.setA(6);
}
```

```
 g++ A.cpp  main.cpp
In file included from main.cpp:1:0:
A.hh: Dans la fonction ‘int main()’:
A.hh:10:10: erreur : ‘void A::setA(int)’ is inaccessible
     void setA(int);
          ^
main.cpp:8:10: erreur : à l'intérieur du contexte
  b.setA(6);
          ^
main.cpp:8:10: erreur : ‘A’ is not an accessible base of ‘B’
```

L'intérêt de la dérivation privée, par exemple, existe quand une classe redéfinie une fonction d'une classe parente. Il n'y a probablement plus de raison d' accéder à cette fonction dans la classe parente. Et on peut ainsi en interdire l'utilisation.

46

DÉRIVATION PROTÉGÉE

Comme il existe le mot clé **protected** pour les membres d'une classe, il existe aussi une notion de dérivation protégée.

Les membres de la classe parente seront ainsi déclarés comme protégés dans la classe fille, et lors des dérivations successives.

47

CLASSE DE BASE & CLASSE DÉRIVÉE

En Programmation orienté objet, on considère qu'un objet d'une classe dérivée peut « remplacer » un objet d'une classe de base. Un objet dérivée d'une classe A peut intervenir quand une classe A est attendu.

En effet, tout se qui se trouve dans une classe A se trouve également dans ses classes dérivées.

En C++, on retrouve également cette notion, à une nuance prés. Elle ne s'applique que dans le cas d'un héritage publique.

Il existe donc une conversion implicite d'une classe fille vers un type de la classe parente.

48

CONVERSION

```
class A
{
private:
   int _a;
public:
   A(int);
   void affiche();
};

class B : public A
{
private:
   int _b;
public:
   B(int, int);
   void affiche();
};
```

.hh

```
A::A(int a)
{
   std::cout << "Cons A"
   << std::endl;
   _a = a;
}
void A::affiche()
{
   std::cout << "A : ";
   std::cout << _a
   << std::endl;
}

B::B(int a, int b) : A(a)
{
   std::cout << "CONS B"
   << std::endl;
   _b = b;
}
void B::affiche()
{
   std::cout << "B : ";
   A::affiche();
   std::cout << _b
   << std::endl;
}
```

.cpp

```
int main()
{
   std::cout << "On construit a, b"
      << std::endl;
   A a(5);
   B b(6,7);

   std::cout << "On affiche a, b"
      <<std::endl;
   a.affiche();
   b.affiche();
   std::cout << "on dit a = b"
      << std::endl;
   a = b;
   std::cout << "On affiche a, b"
      <<std::endl;
   a.affiche();
   b.affiche();
}
```

main.cpp

Dans cet exemple – cas d'école – on déclare deux classes, A, et B dérivant publiquement de A.

Dans le main, on construit deux objets : 'a' de type A, et 'b' de type B.

Ensuite on dit a = b.

On a le droit de le faire car 'b' est de type B, dérivant de A, donc peut faire l'affaire dans le cas où un type A est attendu.

Ainsi, a = b est accepté par le compilateur. Dans ce cas, 'b' est converti en un objet de type A.

49

CONVERSION

```
class A
{
private:
   int _a;
public:
   A(int);
   void affiche();
};

class B : public A
{
private:
   int _b;
public:
   B(int, int);
   void affiche();
};
```

.hh

```
A::A(int a)
{
   std::cout << "Cons A"
   << std::endl;
   _a = a;
}
void A::affiche()
{
   std::cout << "A : ";
   std::cout << _a
   << std::endl;
}

B::B(int a, int b) : A(a)
{
   std::cout << "CONS B"
   << std::endl;
   _b = b;
}
void B::affiche()
{
   std::cout << "B : ";
   A::affiche();
   std::cout << _b
   << std::endl;
}
```

.cpp

```
int main()
{
   std::cout << "On construit a, b"
      << std::endl;
   A a(5);
   B b(6,7);

   std::cout << "On affiche a, b"
      <<std::endl;
   a.affiche();
   b.affiche();
   std::cout << "on dit a = b"
      << std::endl;
   a = b;
   std::cout << "On affiche a, b"
      <<std::endl;
   a.affiche();
   b.affiche();
}
```

main.cpp

```
$ ./a.exe
On construit a, b
Cons A
Cons A
CONS B
On affiche a, b
A : 5
B : A : 6
7
on dit a = b
On affiche a, b
A : 6
B : A : 6
7
```

Néanmoins, comme le montre l'affichage de notre programme, lorsque 'b' est converti, il perd une partie de ses données membres – celles de B – pour ne garder que les données membres héritées : celles de A. Ce qui est normal car 'a' est de type A.

Comme on le voit, quand on réaffiche 'a', sa donnée privée a bien pris la valeurs de la partie A de b.

50

CONVERSION, POINTEURS & RÉFÉRENCES

```
class A
{
private:
   int _a;
public:
   A(int);
   void affiche();
};

class B : public A
{
private:
   int _b;
public:
   B(int, int);
   void affiche();
};
```

.hh

```
A::A(int a)
{
   std::cout << "Cons A"
   << std::endl;
   _a = a;
}
void A::affiche()
{
   std::cout << "A : ";
   std::cout << _a
   << std::endl;
}

B::B(int a, int b) : A(a)
{
   std::cout << "CONS B"
   << std::endl;
   _b = b;
}
void B::affiche()
{
   std::cout << "B : ";
   A::affiche();
   std::cout << _b
   << std::endl;
}
```

.cpp

```
int main()
{
   std::cout << "On construit b"
      << std::endl;
   A  *pa;
   B  b(6,7);

   pa = &b;
   std::cout << "On affiche pa, b"
      <<std::endl;
   pa->affiche();
   b.affiche();
   std::cout << "ra : reference sur b"
      << std::endl;
   A &ra = b;
   std::cout << "On affiche ra, b"
      <<std::endl;
   ra.affiche();
   b.affiche();
}
```

main.cpp

```
$ ./a.exe
On construit b
Cons A
CONS B
On affiche pa, b
A : 6
B : A : 6
7
ra : reference sur b
On affiche ra, b
A : 6
B : A : 6
7
```

Le mécanisme avec les pointeurs et les références reste similaire.
Si on définit un pointeur sur A, on peut l'initialiser avec une adresse de type pointeur sur B.
De même, une référence sur A peut prendre l'adresse d'un objet de type B.

On commence à entrevoir une chose intéressante : un pointeur ou une référence peuvent pointer vers des objets qui ne sont pas de leur type, mais d'un type dérivé.
C'est plus intéressant que pour les objets, car le type d'un objet ne varie pas, même si on l'initialise avec un autre type, les valeurs supplémentaires sont perdues lors de la conversion.

51

LIMITATIONS

```
class A
{
private:
   int _a;
public:
   A(int);
   void affiche();
};

class B : public A
{
private:
   int _b;
public:
   B(int, int);
   void affiche();
};
```

```
A::A(int a)
{
   std::cout << "Cons A"
   << std::endl;
   _a = a;
}
void A::affiche()
{
   std::cout << "A : ";
   std::cout << _a
   << std::endl;
}

B::B(int a, int b) : A(a)
{
   std::cout << "CONS B"
   << std::endl;
   _b = b;
}
void B::affiche()
{
   std::cout << "B : ";
   A::affiche();
   std::cout << _b
   << std::endl;
}
```

```
int main()
{
   A* a = new B(5,6);

   a->affiche();
}
```

```
$ ./a.exe
Cons A
CONS B
A : 5
```

On reprend les classes A et B de l'exemple précèdent.

Dans le main, on déclare un pointeur sur A, qu'on initialise avec un objet de type B. C'est donc le constructeur de B qui est appelé. C'est valide comme on l'a vu précédemment.

On appelle alors la fonction affiche de notre pointeur et … déception, c'est la fonction affiche d'un objet de type A qui est appelée, et non pas celle d'un objet de type B, même si c'est bien un objet de ce type qui a été créé.

52

LIMITATIONS

```
class A
{
private:
   int _a;
public:
   A(int);
   void affiche();
};

class B : public A
{
private:
   int _b;
public:
   B(int, int);
   void affiche();
};
```

```
A::A(int a)
{
   std::cout << "Cons A"
   << std::endl;
   _a = a;
}
void A::affiche()
{
   std::cout << "A : ";
   std::cout << _a
   << std::endl;
}

B::B(int a, int b) : A(a)
{
   std::cout << "CONS B"
   << std::endl;
   _b = b;
}
void B::affiche()
{
   std::cout << "B : ";
   A::affiche();
   std::cout << _b
   << std::endl;
}
```

```
int main()
{
  A* a = new B(5,6);

  a->affiche();
}
```

```
$ ./a.exe
Cons A
CONS B
A : 5
```

Cela vient du fait que les fonctions appelées sont « figées » lors de la compilation. Et pour le compilateur, un pointeur sur un objet correspond au type de cet objet, et c'est donc les fonctions de la classe de cet objet qui seront appelées. Même si lors de l'exécution, l'objet pointé est en réalité « plus grand ».

On verra un peu plus tard un mécanisme qui permet de passer outre cette difficulté.

CONSTRUCTEUR DE RECOPIE & HÉRITAGE

Nous avons déjà parlé du constructeur de recopie. Celui-ci est appelé dans le cas d'une initialisation d'un objet par un objet de même type, ou d'une transmission par valeur ou par retour d'une fonction.

Il y a plusieurs cas possible, suivant si le constructeur de recopie est déclaré ou non dans la classe dérivée. Et des nuances un peu délicates comme on va le voir.

CONSTRUCTEUR DE RECOPIE & HÉRITAGE

```
class A
{
private:
   int _a;
public:
   A(int);
};

class B : public A
{
private:
   int _b;
public:
   B(int, int);
   B(const B&);
};
```

```
A::A(int a)
{
   std::cout << "Cons A"
   << std::endl;
   _a = a;
}

B::B(int a, int b) : A(a)
{
   std::cout << "CONS B"
   << std::endl;
   _b = b;
}

B::B(const B& b) : A(b)
{
   std::cout << "Recopie B"
   << std::endl;
   _b = b._b;
}
```

```
void affiche(B b){
}

int main()
{
   B b(5,6);
   affiche(b);
}
```

```
$ ./a.exe
Cons A
CONS B
Cons A
Recopie B
```

Si l'objet b de type B définit un constructeur de recopie, celui-ci n' appellera pas automatiquement celui de A, comme c'est déjà le cas pour les constructeurs. Il faut donc appeler le constructeur de recopie de A dans le constructeur de recopie de B, en lui passant … 'b' en paramètre. Celui-ci sera converti au passage et la partie A de B sera copiée.

Pour le reste, on copie les données membres de 'B privée de A' dans le constructeur de recopie de B.

CONSTRUCTEUR DE RECOPIE & HÉRITAGE

```
class A
{
private:
   int _a;
   A(const A&);
public:
   A(int);
};

class B : public A
{
private:
   int _b;
public:
   B(int, int);
   B(const B&);
};
```

```
A::A(int a)
{
   std::cout << "Cons A"
   << std::endl;
   _a = a;
}

B::B(int a, int b) : A(a)
{
   std::cout << "CONS B"
   << std::endl;
   _b = b;
}

B::B(const B& b) : A(b)
{
   std::cout << "Recopie B"
   << std::endl;
   _b = b._b;
}
```

```
void affiche(B b){
}

int main()
{
   B b(5,6);
   affiche(b);
}
```

```
$ g++ A.cpp main.cpp
In file included from A.cpp:2:0:
A.hh: Dans le constructeur de copie 'B::B(const B&)':
A.hh:8:5: erreur : 'A::A(const A&)' is private
     A(const A&);
     ^
A.cpp:18:23: erreur : à l'intérieur du contexte
 B::B(const B& b) : A(b)
                       ^
```

Que se passe t il si on déclare un constructeur de recopie de A en privé ?

On interdit alors aux objets héritant de A de se copier !

OPÉRATEUR D'AFFECTATION & HÉRITAGE

Si la classe dérivée ne surdéfinit pas l'opérateur d'affectation '=', l'affectation se déroule en utilisant l'opérateur par défaut.

La partie héritée de A est alors traitée par l'affectation prévue par A si elle existe et qu'elle est public, ou par l'affectation par défaut de A si elle n'est pas redéfinie.

Si la classe A surdéfinit son opérateur d'affectation en privé, alors l'affectation est interdite pour elle, ainsi que pour ses classes dérivées, comme dans le cas des constructeurs de recopie.

57

OPÉRATEUR D'AFFECTATION & HÉRITAGE

```
class A
{
private:
   int _a;
public:
   A(int);
   A& operator=(const A&);
};

class B : public A
{
private:
   int _b;
public:
   B(int, int);
   //B& operator=(const B&);
};
```

```
A::A(int a)
{
   std::cout << "Cons A"
   << std::endl;
   _a = a;
}
A& A::operator=(const A&a)
{
   std::cout << "= A"
      << std::endl;
   _a = a._a;
   return *this;
}

B::B(int a, int b) : A(a)
{
   std::cout << "CONS B"
   << std::endl;
   _b = b;
}

/*B& B::operator=(const B&b)
{
   std::cout << "= B"
      << std::endl;
   _b = b._b;
   return *this;
}*/
```

```
int main()
{
   B b1(5,6);
   B b2(6,7);

   b2 = b1;
}
```

```
$ ./a.exe
Cons A
CONS B
Cons A
CONS B
= A
```

Ici, B ne surdéfinit pas d'opérateur d'affectation (celui-ci est en commentaire…).

L'opérateur '=' de A est par contre défini, et celui-ci est appelé lorsqu'on écrit b2 = b1.

58

OPÉRATEUR D'AFFECTATION & HÉRITAGE

```
class A
{
private:
   int _a;
public:
   A(int);
   A& operator=(const A&);
};

class B : public A
{
private:
   int _b;
public:
   B(int, int);
   B& operator=(const B&);

};
```

```
A::A(int a)
{
   std::cout << "Cons A"
   << std::endl;
   _a = a;
}
A& A::operator=(const A&a)
{
   std::cout << "= A"
      << std::endl;
   _a = a._a;
   return *this;
}

B::B(int a, int b) : A(a)
{
   std::cout << "CONS B"
   << std::endl;
   _b = b;
}

B& B::operator=(const B&b)
{
   std::cout << "= B"
      << std::endl;
   _b = b._b;
   return *this;
}
```

```
int main()
{
   B b1(5,6);
   B b2(6,7);

   b2 = b1;
}
```

```
$ ./a.exe
Cons A
CONS B
Cons A
CONS B
= B
```

Ici, l'opérateur '=' a été défini aussi dans B.

On voit alors que celui-ci n'appelle pas l'opérateur '=' de A !

On se retrouve un peu comme dans le cas du constructeur de recopie, à ceci prêt qu'on ne peut pas passer les paramètres au constructeur de A comme plus haut…

On doit donc prévoir la recopie aussi des membres de A mais …

OPÉRATEUR D'AFFECTATION & HÉRITAGE

```
class A
{
private:
   int _a;
public:
   A(int);
   A& operator=(const A&);
};

class B : public A
{
private:
   int _b;
public:
   B(int, int);
   B& operator=(const B&);

};
```

```
A::A(int a)
{
   std::cout << "Cons A"
   << std::endl;
   _a = a;
}
A& A::operator=(const A&a)
{
   std::cout << "= A"
      << std::endl;
   _a = a._a;
   return *this;
}

B::B(int a, int b) : A(a)
{
   std::cout << "CONS B"
   << std::endl;
   _b = b;
}

B& B::operator=(const B&b)
{
   std::cout << "= B"
      << std::endl;
   _a = b._a;
   _b = b._b;
   return *this;
}
```

```
int main()
{
   B b1(5,6);
   B b2(6,7);

   b2 = b1;
}
```

… mais on ne peut pas faire ca dans notre exemple !

En effet , _a est un membre privé de A et on ne peut pas l'utiliser comme cela dans B !

```
$ g++ A.cpp main.cpp
In file included from A.cpp:2:0:
A.hh: Dans la fonction membre 'B& B::operator=(const B&)':
A.hh:7:9: erreur : 'int A::_a' is private
     int _a;
         ^
A.cpp:29:5: erreur : à l'intérieur du contexte
     _a = b._a;
     ^
In file included from A.cpp:2:0:
A.hh:7:9: erreur : 'int A::_a' is private
     int _a;
         ^
A.cpp:29:12: erreur : à l'intérieur du contexte
     _a = b._a;
            ^
```

OPÉRATEUR D'AFFECTATION & HÉRITAGE

```
class A
{
private:
   int _a;
public:
   A(int);
   A& operator=(const A&);
};

class B : public A
{
private:
   int _b;
public:
   B(int, int);
   B& operator=(const B&);

};
```

```
A::A(int a)
{
   std::cout << "Cons A"
   << std::endl;
   _a = a;
}
A& A::operator=(const A&a)
{
   std::cout << "= A"
      << std::endl;
   _a = a._a;
   return *this;
}
B::B(int a, int b) : A(a)
{
   std::cout << "CONS B"
   << std::endl;
   _b = b;
}
B& B::operator=(const B&b)
{
   std::cout << "= B"
      << std::endl;
   A* pa = this;
   const A* pb = &b;
   *pa = *pb;
   _b = b._b;
   return *this;
}
```

```
int main()
{
   B b1(5,6);
   B b2(6,7);

   b2 = b1;
}
```

```
$ ./a.exe
Cons A
CONS B
Cons A
CONS B
= B
= A
```

On va donc s'arranger pour pouvoir quand même utiliser l'opérateur d'affectation de A.

On va créer un pointeur de type A* qui pointera vers this. Le pointeur this(de type B*) est alors converti implicitement vers un pointeur de type A*.

On fait de même pour b que l'on met dans un pointeur de type const A* également. (Il est passé const en paramètre, donc un pointeur const également). Ensuite on affecte la valeur pointée par le nouveau, sur l'ancien (this).

Et le tour est joué !

Prof. Mohamed BOUDCHICHE

HÉRITAGE & TEMPLATE

```
template <typename T>
class A
{
private:
   T _a;
public:
   A(T a) {_a = a;}
};
```

```
class B : public A<int>
{
private:
   int _b;
public:
   B(int, int);
};
```

L'héritage et les template se mélangent plutôt bien.

Il y a plusieurs cas possibles.

Ici, seule la classe parente est template. La classe fille qui ne l'est pas, doit donc définir lors de sa déclaration un type pour A.

Ici, A est une classe template et B ne l'est pas.

Donc B ne peut hériter que d'un type particulier de A.

Prof. Mohamed BOUDCHICHE

HÉRITAGE & TEMPLATE

```
class A
{
private:
   int _a;
public:
   A(int a);
};

template <typename T>
class B : public A
{
private:
   T _b;
public:
   B(int a, T b) : A(a)
   {
   _b = b;
   }

};
```

Dans le cas où c'est la classe dérivée qui est templaté, tout se déroule naturellement…

HÉRITAGE & TEMPLATE

```
template <typename T>
class A
{
private:
   T _a;
public:
   A(T a){ _a = a;}
};
template <typename T>
class B : public A<T>
{
private:
   T _b;
public:
   B(T a, T b) : A<T>(a)
   { _b = b; }
};
```

Enfin, dans le cas où la classe dérivée et parente sont templatées, on utilisera la syntaxe suivante.

Ici, on passe le type template de B à A. Elles seront donc templatées par le même type.

On fera également attention à l'appel du constructeur de A dans le constructeur de B.

HÉRITAGE & TEMPLATE

```
template <typename T>
class A
{
private:
   T _a;
public:
   A(T a){ _a = a;}
};

template <typename T, typename U>
class B : public A<U>
{
private:
   T _b;
public:
   B(U a, T b) : A<U>(a)
   { _b = b; }
};
```

Il n'est pas obligatoire de templater la classe dérivée et héritée par le même type !

Ici, le type T est utilisé dans B, alors que U servira à définir le type de A.