Programmation Orientée Objet: C++ - Cours 6

M1 IM/MF-MPA - Année 2023-2024

Les patrons de classes

- Il existe un mécanisme similaire aux fonctions template pour les classes.
- Bien que semblable aux patrons de fonctions sur de nombreux points, il existe des différences avec les templates de classe.
- On va voir que cela permet d'implémenter un code générique et réutilisable.

Classe template Vector

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 1: Vector.hpp

On se souvient de la classe *Vector* que nous avions définie dans ce cours, dans la partie *Surcharge d'opérateurs*.

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

Perte de temps, d'énergie ...

Classe template Vector

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 1: Vector.hpp

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: à l'inverse des classes 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.

Classe template Vector

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 1: Vector.hpp

En résumé, on adoptera les conventions suivantes dans ce cours:

- Déclaration d'une classe → fichier .hh
- Définition d'une classe → fichier .cpp
- Déclaration d'un template de classe → fichier .hpp

Classe template Vector

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 1: Vector.hpp

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

Classe template Vector

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 1: Vector.hpp

```
#include <iostream>
#include "Vector.hpp"
using namespace std;

int main()
{
  Vector<double> vect(10);

  for (int i=0;i<10;i++)
    vect[i] = i*0.5;

  for (int i=0;i<10;i++)
    cout << vect[i]
         << endl;

  return 0;
}
```

Listing 2: main_Vector.cpp

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

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

Classe template Vector

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 1: Vector.hpp

```
#include <iostream>
#include "Vector.hpp"
using namespace std;

int main()
{
  Vector<double> vect(10);

  for (int i=0;i<10;i++)
    vect[i] = i*0.5;

  for (int i=0;i<10;i++)
    cout << vect[i]
         << endl;

  return 0;
}
```

Listing 2: main_Vector.cpp

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.

Plus compliqué

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 1: Vector.hpp

```
#ifndef __POINT_HPP__
#define __POINT_HPP__
#include<iostream>
using namespace std;

template <typename T>
struct Point
{
  T _x;
  T _y;
};

template <typename T>
ostream & operator<< (ostream& c,
      Point<T>& x) {
  c << x._x << ";" << x._y;
  return c;
}

#endif
```

Listing 3: Point.hpp

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

int main()
{
 Vector<double> vect(10);

 for (int i=0;i<10;i++)  {
  vect[i] = i*0.5;
  cout << vect[i] << endl; }

 Vector< Point<int> > vect2(10);

 for (int i=0;i<10;i++) {
  vect2[i]._x = i;
  vect2[i]._y = 10-i;
  cout << vect2[i] << endl; }
}
```

Listing 4: main_Vector2.cpp

Dans cet exemple, on a ajouté la structure template *Point*, qui s'utilise comme une classe template, avec les particularités liées aux structures.

On peut ainsi définir un type *Point* dont le type générique est un entier.

On peut alors aussi définir un vecteur de *Point* d'entiers!

Plus compliqué

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 1: Vector.hpp

```
#ifndef __POINT_HPP__
#define __POINT_HPP__
#include<iostream>
using namespace std;

template <typename T>
struct Point
{
  T _x;
  T _y;
};

template <typename T>
ostream & operator<< (ostream& c,
      Point<T>& x) {
  c << x._x << ";" << x._y;
  return c;
}

#endif
```

Listing 3: Point.hpp

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

int main()
{
 Vector<double> vect(10);

 for (int i=0;i<10;i++)  {
  vect[i] = i*0.5;
  cout << vect[i] << endl; }

 Vector< Point<int> > vect2(10);

 for (int i=0;i<10;i++) {
  vect2[i]._x = i;
  vect2[i]._y = 10-i;
  cout << vect2[i] << endl; }
}
```

Listing 4: main_Vector2.cpp

La structure *Point* est très simple, elle ne contient que deux variables de type T générique.

On notera aussi la surcharge de l'opérateur << pour pouvoir afficher un objet *Point*.

Comme *Point* est un type template, il est "logique" qu'une fonction l'utilisant soit aussi template $\cdots$ sinon quel type de *Point* le compilateur instancierait-il?

Pour finir sur 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 définit 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++ (https://eigen.tuxfamily.org/) est une librairie pour l'algèbre linéaire et contient des algorithmes très optimisés.

Pour finir sur les patrons de classe

- La notion de template est plus vaste que celle abordée dans ce cours.
- Ainsi, on n'a pas abordé les notions de type expression, ni 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++ et à la POO.

Héritage

- La notion d'héritage en programmation orientée 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...

Exemple

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
using namespace std;

class Forme
{
private:
 string _nom;

public:
 void setNom(const string&);
};
#endif
```

Listing 5: Forme.hh

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

void Forme::setNom(const
    string& nom) {
  _nom = nom;
}
```

Listing 6: Forme.cpp

On commence par écrire une classe très simple, *Forme*, constituée seulement d'une chaîne qui contiendra le nom de notre forme.

Une fonction sera chargée d'initialiser notre chaîne.

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

Exemple

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
using namespace std;

class Forme
{
private:
 string _nom;

public:
 void setNom(const string&);
};
#endif
```

Listing 5: 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 changer la valeur de la chaîne *_nom*, qui est une 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 objet d'extension .o, obtenu après compilation).

Exemple

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
using namespace std;

class Forme
{
private:
 string _nom;

public:
 void setNom(const string&);
};
#endif
```

Listing 5: Forme.hh

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

class Rond: public Forme
{
private:
  double _diametre;
public:
  void setDiametre(double);
};
#endif
```

Listing 7: Rond.hh

```
#include "Forme.hh"
#include "Rond.hh"
#include <iostream>
void Rond::setDiametre(
    double d) {
_diametre = d;
}
```

Listing 8: Rond.cpp

On crée maintenant une première classe fille de la classe *Forme*: à savoir la classe *Rond*. On remarque la ligne:

class Rond: public Forme

Dans la déclaration de la classe *Rond*, ":" suivi de *public Forme* signifie que *Rond* hérite de la classe *Forme*.

Le mot clé *public* sera expliqué dans la suite de ce cours.

Exemple

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
using namespace std;

class Forme
{
private:
 string _nom;

public:
 void setNom(const string&);
};
#endif
```

Listing 5: Forme.hh

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

class Rond: public Forme
{
private:
  double _diametre;
public:
  void setDiametre(double);
};
#endif
```

Listing 7: Rond.hh

```
#include "Forme.hh"
#include "Rond.hh"
#include <iostream>
void Rond::setDiametre(double d) {
_diametre = d;
}
```

Listing 8: Rond.cpp

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

class Carre: public Forme
{
private:
 double _longueur;
public:
 void setLongueur(double);
};
#endif
```

Listing 9: Carre.hh

```
#include "Forme.hh"
#include "Carre.hh"
#include <iostream>

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

Listing 10: 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 diamètre et le carré une longueur.

Exemple

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
using namespace std;

class Forme
{
private:
 string _nom;

public:
 void setNom(const string&);
};
#endif
```

Listing 5: Forme.hh

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

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

Listing 11: main_Forme.cpp

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

class Rond: public Forme
{
private:
  double _diametre;
public:
  void setDiametre(double);
};
#endif
```

Listing 7: Rond.hh

```
#include "Forme.hh"
#include "Rond.hh"
#include <iostream>
void Rond::setDiametre(double d) {
_diametre = d;
}
```

Listing 8: Rond.cpp

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

void Forme::setNom(const string&
     nom) {
  _nom = nom;
}
```

Listing 6: Forme.cpp

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

class Carre: public Forme
{
private:
 double _longueur;
public:
 void setLongueur(double);
};
#endif
```

Listing 9: Carre.hh

```
#include "Forme.hh"
#include "Carre.hh"
#include <iostream>

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

Listing 10: Carre.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!

Exemple

```
#ifndef __FORMEV2_HH__
#define __FORMEV2_HH__

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

class Forme
{
private:
 string _nom;

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

#endif
```

Listing 11: Forme_V2.hh

```
#ifndef __RONDV1_HH__
#define __RONDV1_HH__
#include "Forme_V2.hh"

class Rond: public Forme
{
private:
  double _diametre;
public:
  void setDiametre(double);
};
#endif
```

Listing 12: Rond_V1.hh

```
#include "Rond_V1.hh"
void Rond::setDiametre(double d)
      {
 _diametre = d;
}
```

Listing 13: Rond_V1.cpp

```
#include "Forme_V2.hh"
void Forme::setNom(const string&
      nom) {
  _nom = nom;
}
void Forme::affiche() {
  cout << "Je suis de type " <<
       _nom << endl; }
```

Listing 14: Forme_V2.cpp

```
#ifndef __CARREV1_HH__
#define __CARREV1_HH__
#include "Forme_V2.hh"

class Carre: public
     Forme
{
private:
 double _longueur;
public:
 void setLongueur(
      double);
};
#endif
```

Listing 15: Carre_V1.hh

```
#include "Carre_V1.hh"

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

Listing 16: Carre_V1.cpp

On définit 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 ce que contient la donnée _nom héritée de la classe *Forme*.

On a donc modifié les fonctionnalités de 3 classes après modification d'une seule. Pas mal! Mais l'affichage ne prend pas en compte la spécificité de chaque classe ...

Exemple

```
#include "Carre_V1.hh"
#include "Rond_V1.hh"

int main(void)
{
  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();
}
```

Listing 17: main_FormeV2.cpp

```
#ifndef __RONDV1_HH__
#define __RONDV1_HH__
#include "Forme_V2.hh"

class Rond: public Forme
{
private:
  double _diametre;
public:
  void setDiametre(double);
};
#endif
```

Listing 12: Rond_V1.hh

```
#include "Rond_V1.hh"
void Rond::setDiametre(double d)
       {
 _diametre = d;
}
```

Listing 13: Rond_V1.cpp

```
#include "Forme_V2.hh"
void Forme::setNom(const string&
       nom) {
  _nom = nom;
}
void Forme::affiche() {
  cout << "Je suis de type " <<
       _nom << endl; }
```

Listing 14: Forme_V2.cpp

```
#ifndef __CARREV1_HH__
#define __CARREV1_HH__
#include "Forme_V2.hh"

class Carre: public
      Forme
{
private:
 double _longueur;
public:
 void setLongueur(
       double);
};
#endif
```

Listing 15: Carre_V1.hh

```
#include "Carre_V1.hh"

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

Listing 16: Carre_V1.cpp

```
./a.out

Je suis de
  type forme
  générale
Je suis de
  type carre
Je suis de
  type rond
```

Redéfinition d'une fonction héritée

```
#ifndef __FORMEV2_HH__
#define __FORMEV2_HH__

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

class Forme
{
private:
 string _nom;

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

#endif
```

Listing 11: Forme_V2.hh

```
#ifndef __RONDV2_HH__
#define __RONDV2_HH__
#include "Forme_V2.hh"

class Rond: public Forme
{
private:
  double _diametre;
public:
  void setDiametre(double);
  void affiche();
};
#endif
```

Listing 18: Rond_V2.hh

```
#include "Rond_V2.hh"

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

void Rond::affiche() {
  cout << "Mon diametre est de "
       << _diametre << endl;
}
```

Listing 19: Rond_V2.cpp

```
#ifndef __CARREV2_HH__
#define __CARREV2_HH__
#include "Forme_V2.hh"

class Carre: public
     Forme
{
private:
  double _longueur;
public:
  void setLongueur(
       double);
  void affiche();
};
#endif
```

Listing 20: Carre_V2.hh

```
#include "Carre_V2.hh"
void Carre::setLongueur
     (double l) {
  _longueur = l;   }

void Carre::affiche() {
  cout << "Ma longueur
       est de "
       << _longueur
       << endl;    }
```

Listing 21: Carre_V2.cpp

Pour afficher les spécificités de chaque enfant, on va déclarer puis définir une fonction *affiche* dans chaque classe fille ...

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 classes filles, il faudrait l'écrire en dur??

Ce n'est pas très orienté objet ça ...

Redéfinition d'une fonction héritée

```
#ifndef __FORMEV2_HH__
#define __FORMEV2_HH__

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

class Forme
{
private:
 string _nom;

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

#endif
```

Listing 11: Forme_V2.hh

```
#ifndef __RONDV2_HH__
#define __RONDV2_HH__
#include "Forme_V2.hh"

class Rond: public Forme
{
private:
  double _diametre;
public:
  void setDiametre(double);
  void affiche();
};
#endif
```

Listing 18: Rond_V2.hh

```
#include "Rond_V2.hh"

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

void Rond::affiche() {
  Forme::affiche();
  cout << "Mon diametre est de "
       << _diametre << endl;
}
```

Listing 22: Rond_V22.cpp

```
#ifndef __CARREV2_HH__
#define __CARREV2_HH__
#include "Forme_V2.hh"

class Carre: public
     Forme
{
private:
  double _longueur;
public:
  void setLongueur(
       double);
  void affiche();
};
#endif
```

Listing 20: Carre_V2.hh

```
#include "Carre_V2.hh"
void Carre::setLongueur
     (double l) {
  _longueur = l;  }

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

Listing 23: Carre_V22.cpp

On va plutôt essayer d'appeler dans la fonction *affiche* fille, la fonction *affiche* parente.
Si on écrit simplement *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.

Redéfinition d'une fonction héritée

```
#ifndef __FORMEV2_HH__
#define __FORMEV2_HH__

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

class Forme
{
private:
 string _nom;

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

#endif
```

Listing 11: Forme_V2.hh

```
#ifndef __RONDV2_HH__
#define __RONDV2_HH__
#include "Forme_V2.hh"

class Rond: public Forme
{
private:
  double _diametre;
public:
  void setDiametre(double);
  void affiche();
};
#endif
```

Listing 18: Rond_V2.hh

```
#include "Rond_V2.hh"

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

void Rond::affiche() {
  Forme::affiche();
  cout << "Mon diametre est de "
       << _diametre << endl;
}
```

Listing 22: Rond_V22.cpp

```
#ifndef __CARREV2_HH__
#define __CARREV2_HH__
#include "Forme_V2.hh"

class Carre: public
     Forme
{
private:
  double _longueur;
public:
  void setLongueur(
       double);
  void affiche();
};
#endif
```

Listing 20: Carre_V2.hh

```
#include "Carre_V2.hh"
void Carre::setLongueur
     (double l) {
  _longueur = l;  }

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

Listing 23: Carre_V22.cpp

Nous avons maintenant un affichage adapté selon 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!

```
./a.out

Je suis de
  type forme
  générale
Je suis de
  type carre
Ma longueur
  est de 5
Je suis de
  type rond
Mon diametre
  est de 3
```

Redéfinition d'une fonction héritée

```
#ifndef __FORMEV2_HH__
#define __FORMEV2_HH__

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

class Forme
{
private:
 string _nom;

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

#endif
```

Listing 11: Forme_V2.hh

```
#ifndef __RONDV2_HH__
#define __RONDV2_HH__
#include "Forme_V2.hh"

class Rond: public Forme
{
private:
  double _diametre;
public:
  void setDiametre(double);
  void affiche();
};
#endif
```

Listing 18: Rond_V2.hh

```
#include "Rond_V2.hh"

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

void Rond::affiche() {
  Forme::affiche();
  cout << "Mon diametre est de "
       << _diametre << endl;
}
```

Listing 22: Rond_V22.cpp

```
#ifndef __CARREV2_HH__
#define __CARREV2_HH__
#include "Forme_V2.hh"

class Carre: public
     Forme
{
private:
  double _longueur;
public:
  void setLongueur(
       double);
  void affiche();
};
#endif
```

Listing 20: Carre_V2.hh

```
#include "Carre_V2.hh"
void Carre::setLongueur
     (double l) {
  _longueur = l;  }

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

Listing 23: Carre_V22.cpp

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

Dans le fichier cpp contenant le *main*, il faut alors un peu modifier la syntaxe de l'appel de la fonction *r.Forme::affiche()*,

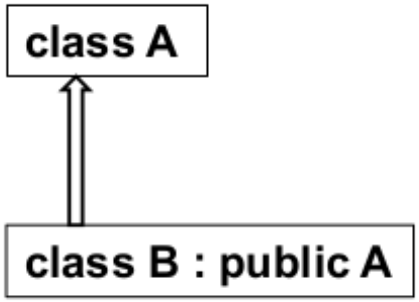
```
./a.out

Je suis de
  type forme
  générale
Je suis de
  type carre
Ma longueur
  est de 5
Je suis de
  type rond
```

Surcharge et définition

On fera attention sur un point:

Une fonction membre redéfinie dans une classe masque automatiquement les fonctions héritées, 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 paramètre 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.

Constructeurs & destructeurs

```
#ifndef __AB1__HH__
#define __AB1__HH__
class A
{
private:
  int _a;
public:
  A(int);
};

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

Listing 24: AB_1.hh

```
#include "AB_1.hh"
A::A(int a)
{
  _a = a;
}

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

Listing 25: AB_1.cpp

Pour construire un objet B, on va rencontrer une difficulté si on doit fournir des paramètres au constructeur de A. En effet, a priori, les paramètres fournis au constructeur de B sont sensés être utilisés par le constructeur de A.

Or ce dernier doit être appelé avant.

$\rightarrow$ Lors de la définition du constructeur de B, on utilise la même syntaxe que pour les objets membres.

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 un troisième, lié 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.

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 pouvoir 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.

Contrôle des accès

```
#ifndef __AB2__HH__
#define __AB2__HH__
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();
};
#endif
```

Listing 26: AB_2.hh

```
#include "AB_2.hh"
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;
}
```

Listing 27: AB_2.cpp

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 cette partie sur l'héritage, nous avons déclaré qu'une classe dérivait 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 publics de la classe parente, comme s'ils faisaient partie de la classe fille.

Dérivation publique & dérivation privée

```
#ifndef __AB3__HH__
#define __AB3__HH__
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);
};
#endif
```

Listing 28: AB_3.hh

```
#include "AB_3.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) {}
```

Listing 29: AB_3.cpp

```
#include "AB_3.hh"

int main(void){
  A a(7);
  B b(5);
  C c(5);

  a.setA(3);
  //  b.setA(6);
  c.setA(6);
}
```

Listing 30: main_AB3.cpp

(Compilation avec la ligne *b.setA(6);*)

```
g++ AB_3.cpp main_AB3.cpp
In file included from main.cpp:1:0:
AB.hh: Dans la fonction 'int main()':
AB.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 publics de la classe *A* de l'extérieur.

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.

On n'utilisera pas cette forme d'héritage dans le cours.

Classe de base & classe dérivée

- En Programmation Orientée Objet, on considère qu'un objet d'une classe dérivée peut "remplacer" un objet d'une classe de base. Un objet dérivé d'une classe A peut être utilisé quand celui d'une classe A est attendue.
- 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 près. Elle ne s'applique que dans le cas d'un héritage public.
- Il existe donc une conversion implicite d'une classe fille vers le type de la classe parente.

Conversion

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

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

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

Listing 31: AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

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

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 32: AB_4.cpp

```
#include "AB_4.hh"

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

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

Listing 33: main_AB4.cpp

Dans cet exemple - cas d'école - on déclare deux classes: la classe A et la classe 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.

Conversion

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

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

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

Listing 31: AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

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

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 32: AB_4.cpp

```
#include "AB_4.hh"

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

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

Listing 33: main_AB4.cpp

```
 ./a.out
On construit a, b
Constr A
Constr A
Constr 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 la variable *b* est convertie, elle 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 la variable *a*, sa donnée privée a bien pris la valeur de la partie *A* de *b*.

Conversion, pointeurs & références

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

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

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

Listing 31: AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

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

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 32: AB_4.cpp

```
#include "AB_4.hh"

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

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

Listing 34: main_AB42.cpp

```
./a.out

On construit b
Constr A
Constr 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.

Conversion, pointeurs & références

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

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

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

Listing 31: AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

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

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 32: AB_4.cpp

```
#include "AB_4.hh"

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

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

Listing 34: main_AB42.cpp

```
./a.out

On construit b
Constr A
Constr 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
```

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, mais si on l'initialise avec un autre type, les valeurs supplémentaires sont perdues lors de la conversion.

Conversion, pointeurs & références

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

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

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

Listing 31: AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

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

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 32: AB_4.cpp

```
#include "AB_4.hh"

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

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

Listing 34: main_AB42.cpp

```
./a.out

On construit b
Constr A
Constr 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
```

Alors qu'un pointeur (ou une référence) n'est qu'une adresse qui pointe vers un type en vérité plus "grand" que le type réel du pointeur.

Limitations

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

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

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

Listing 31: AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

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

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 32: AB_4.cpp

```
#include "AB_4.hh"

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

  a->affiche();

  return 0;
}
```

Listing 35: main_AB43.cpp

```
./a.out
Constr A
Constr B
A : 5
```

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

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 $\cdots$ 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.

Limitations

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

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

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

Listing 31: AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

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

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 32: AB_4.cpp

```
#include "AB_4.hh"

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

  a->affiche();

  return 0;
}
```

Listing 35: main_AB43.cpp

```
./a.out
Constr A
Constr 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 ce sont 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 dans la suite un mécanisme qui permet de passer outre cette difficulté.

Polymorphisme

Définition

Nous avons vu qu'un pointeur sur une classe parente pouvait recevoir l'adresse de n'importe quel objet dérivant la classe parente.

Il y avait néanmoins une contrainte: lorsque l'on appelait une fonction de l'objet pointé, c'était la fonction de la classe parente qui était appelée et pas la fonction de l'objet réellement pointé.

Cela provient du fait qu'à la compilation, le compilateur ne connaît pas le type de l'objet réellement pointé, et se base uniquement sur le type du pointeur. Il inclura dans le code compilé les appels aux fonctions de ce type-là, qui correspond au type de la classe parente. Il s'agit d'un typage statique.

Définition

C++ sait faire mieux que cela et permet un typage dynamique de ces objets.

Lors de la compilation, il sera alors mis en place un mécanisme permettant de choisir au moment de l'exécution la fonction qui sera appelée.

Il s'agit du Polymorphisme.

Des objets de types différents peuvent être pointés par le même pointeur et l'exécution du code se fait de manière cohérente avec les types réellement pointés.

Pour cela, nous allons voir un nouveau type de fonctions membres: les fonctions virtuelles.

Fonctions virtuelles

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

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

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

Listing 31: AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

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

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 32: AB_4.cpp

```
#include "AB_4.hh"

int main()
{
  A* pa;
  B b(6,7);

  pa = &b;
  pa->affiche();
}
```

Listing 36: main_AB44.cpp

```
./a.out

Constr A
Constr B
A : 6
```

Dans cet exemple, nous rappelons le problème.

Dans le *main*, on crée un pointeur sur un objet de type A. Mais celui-ci pointe en réalité sur un objet de type *B* par le jeu des conversions implicites.

Lorsqu'on appelle la fonction *affiche*, c'est celle de *A* qui est appelée et non celle de B.

Fonctions virtuelles

```cpp
#ifndef __AB7__HH__
#define __AB7__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  virtual void affiche();
        // fonction
  // virtuelle
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche(); //
       définie pour B
};
#endif
```

Listing 37 : AB_7.hh

```cpp
#include "AB_7.hh"

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

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

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

void B::affiche()
{
  A::affiche();
  cout << "B : ";
  cout << _b << endl;
}
```

Listing 38 : AB_7.cpp

```cpp
#include "AB_7.hh"

int main()
{
  A* pa;
  B b(6,7);

  pa = &b;
  pa->affiche();
}
```

Listing 39 : main_AB7.cpp

```
./a.out

A : 6
B : 7
```

On se contente ensuite d'ajouter le mot clé *"virtual"* à la déclaration de la fonction *affiche*.

Lors de l'appel dans *main*, c'est maintenant la fonction de *B* qui est appelée!

Que s'est-il passé?

Fonctions virtuelles

Le mot clé "virtual" placé dans la déclaration d'une fonction permet de rendre une fonction "virtuelle".

Même si visuellement, il semblerait que peu de chose ait changé dans le code, en vérité un système relativement complexe a été mis en place par le compilateur pour obtenir un comportement cohérent dans le cas du polymorphisme : c'est en effet la fonction membre du type réel de l'objet qui est appelée et plus celle du type du pointeur.

Fonctions virtuelles - Limitations

Les fonctions virtuelles ont néanmoins quelques règles à respecter:

- Seule une fonction membre peut être virtuelle. Les fonctions "ordinaires" ou amies sont exclues de ce mécanisme.
- Un constructeur ne peut pas être virtuel. En effet, un constructeur est appelé pour construire une classe. Cela n'aurait pas trop de sens qu'en réalité, il construise une autre classe $\cdots$
- En revanche, un destructeur peut être virtuel.

Fonctions virtuelles - Destructeur

```
#ifndef __AB8__HH__
#define __AB8__HH__

#include<iostream>
using namespace std;

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

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

Listing 40: AB_8.hh

```
#include "AB_8.hh"

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

A::~A()
{
  cout << "Destr A" << endl;
}

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

B::~B()
{
  cout << "Destr B" << endl;
}
```

Listing 41: AB_8.cpp

```
#include "AB_8.hh"

int main()
{
  A* pa = new B(2,3);

  delete pa;
}
```

Listing 42: main_AB8.cpp

```
./a.out

Destr A
```

Que se passe-t-il lorsqu'un destructeur n'est pas virtuel dans cet exemple?

On construit un objet de type *B* avec *new*. Il faudra donc le détruire.

Mais son adresse est stockée dans un pointeur de type A*.

C'est donc le destructeur de *A* qui est appelé! Et l'objet *B* n'est pas entièrement détruit ...

Fonctions virtuelles - Destructeur

```cpp
#ifndef __AB9__HH__
#define __AB9__HH__

#include<iostream>
using namespace std;

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

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

Listing 43: AB_9.hh

```cpp
#include "AB_9.hh"

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

A::~A()
{
  cout << "Destr A" << endl;
}

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

B::~B()
{
  cout << "Destr B" << endl;
}
```

Listing 44: AB_9.cpp

```cpp
#include "AB_9.hh"

int main()
{
  A* pa = new B(2,3);

  delete pa;
}
```

Listing 45: main_AB9.cpp

```
./a.out

Destr B
Destr A
```

On déclare maintenant le destructeur de A comme étant virtuel.

Lors de l'exécution, c'est donc bien le destructeur de B qui est appelé.

Intérêt

```
#ifndef __AB7__HH__
#define __AB7__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  virtual void affiche(); //
      fonction
  // virtuelle
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche(); // définie
      pour B
};
#endif
```

Listing 37 : AB_7.hh

```
#include "AB_7.hh"

int main()
{
  A* tab[2]; // tableau de 2
      pointeurs sur A
  tab[0] = new A(5);
  tab[0]->affiche();

  tab[1] = new B(7,9);
  tab[1]->affiche();

  delete tab[0];
  delete tab[1];
}
```

Listing 46 : main_AB72.cpp

Un des intérêts du polymorphisme est de pouvoir créer des tableaux de pointeurs sur une classe.

Ici, à l'intérieur du tableau *tab*, on peut avoir une adresse d'un objet *A* ou d'un objet B, ce qui permet de manipuler différents types de données.

On n'oublie pas le *delete* ici (sans [])!

```
#include "AB_7.hh"

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

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

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

void B::affiche()
{
  A::affiche();
  cout << "B : ";
  cout << _b << endl;
}
```

Listing 38 : AB_7.cpp

```
./a.out

A : 5
A : 7
B : 9
```

Classes abstraites - Fonctions virtuelles pures

```
#ifndef __AB12__HH__
#define __AB12__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  virtual void affiche() = 0; //
       fonction virtuelle pure
};

class B : public A
{
private:
  int _b;
public:
  void affiche(); // définie pour B
};
#endif
```

Listing 47: AB_11.hh

Les classes abstraites en POO sont des classes qui n'ont pas pour but d'être instanciées directement.

Il s'agira alors pour l'utilisateur de la classe de créer une classe et d'hériter de celle-ci en créant le code supplémentaire si besoin.

Pour cela, le C++ introduit des fonctions virtuelles dites "pures", c'est-à-dire qu'on ne donne pas de définition à cette fonction et c'est la classe fille qui devra définir cette fonction.

Classes abstraites - Fonctions virtuelles pures

```
#ifndef __AB12__HH__
#define __AB12__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  virtual void affiche() = 0; //
       fonction virtuelle pure
};

class B : public A
{
private:
  int _b;
public:
  void affiche(); // définie pour
       B
};
#endif
```

Listing 48 : AB_12.hh

```
#include "AB_12.hh"

int main()
{
  A* a = new A(); // ko!
  a->affiche();

  A* b = new B();
  b->affiche();
}
```

Listing 49 : main_AB12.cpp

```
  g++ AB_12.cpp main_AB12.cpp -o main_AB12
main_AB12.cpp: In function 'int main()':
main_AB12.cpp:5:16: error: invalid
new-expression of abstract class type 'A'
    5 |   A* a = new A();
      |                ^
In file included from main_AB12.cpp:1:
AB_12.hh:7:7: note:
because the following virtual functions
are pure within 'A':
    7 | class A
      |       ^
      AB_12.hh:12:16: note:
      'virtual void A::affiche()'
      12 |   virtual void affiche() = 0;
      // fonction virtuelle pure
```

Dans cet exemple, dans la fonction *main*, on essaie d'instancier un objet de type A.

Le compilateur refuse car on essaie d'instancier une classe abstraite.

Classes abstraites - Fonctions virtuelles pures

```
#ifndef __AB12__HH__
#define __AB12__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  virtual void affiche() = 0; //
       fonction virtuelle pure
};

class B : public A
{
private:
  int _b;
public:
  void affiche(); // définie pour
       B
};
#endif
```

Listing 48 : AB_12.hh

```
#include "AB_12.hh"

int main()
{
  A* a = new A(); // ko!
  a->affiche();

  A* b = new B();
  b->affiche();
}
```

Listing 49 : main_AB12.cpp

```
  g++ AB_12.cpp main_AB12.cpp -o main_AB12
main_AB12.cpp: In function 'int main()':
main_AB12.cpp:5:16: error: invalid
new-expression of abstract class type 'A'
    5 |   A* a = new A();
      |                ^
In file included from main_AB12.cpp:1:
AB_12.hh:7:7: note:
because the following virtual functions
are pure within 'A':
    7 | class A
      |       ^
      AB_12.hh:12:16: note:
      'virtual void A::affiche()'
      12 |   virtual void affiche() = 0;
      // fonction virtuelle pure
```

Une classe abstraite est une classe contenant au moins une fonction virtuelle pure.

On ne peut pas instancier cette classe directement.

La classe *B* hérite publiquement de *A* et redéfinit la fonction *affiche*. On pourra alors instancier un objet de type *B*.

Classes abstraites - Fonctions virtuelles pures

```
#ifndef __AB12__HH__
#define __AB12__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  virtual void affiche() = 0; // fonction
        virtuelle pure
};

class B : public A
{
private:
  int _b;
public:
  void affiche(); // définie pour B
};
#endif
```

Listing 48 : AB_12.hh

```
#include "AB_12.hh"

int main()
{
  A* b = new B();
  b->affiche();
}
```

Listing 50 : main_AB12ok.cpp

```
  ./a.out

B : 0
```