Programmation Orientée Objet

Surcharge de méthodes et d'opérateurs, et polymorphisme

Guillaume Revy

guillaume.revy@univ-perp.fr

Université de Perpignan Via Domitia

Plan du cours

1. Surcharge de méthodes et d'opérateurs

2. Polymorphisme en C++
   - Différents types de polymorphisme
   - Méthode virtuelle et classe abstraite

3. Suite et fin d'éléments importants du C++

Plan du cours

Qu'est ce que la surcharge de méthode ?

- Définition : la surcharge d'une méthode consiste à définir deux méthodes ayant le même nom, mais pas les mêmes arguments

```
// Sans surcharger les methodes
double methode_double(double, double);
int methode_int(int, int);
```

```
// En surchargeant les methodes
double methode(double, double);
int methode(int, int);
```

- Le compilateur choisit la méthode à utiliser en fonction du type des paramètres :
  - la première méthode est appelée lorsque les paramètres sont des nombres flottants,
  - la seconde méthode est appelée lorsque les paramètres sont des entiers.

Exemple de surcharge de méthode

```
#include <iostream>

double methode(double a, double b)
{
  if(a > b)
    std::cout << "Type double : a(" << a << ") > b(" << b << ")." << std::endl;
  else if(a <= b)
    std::cout << "Type double : a(" << a << ") <= b(" << b << ")." << std::endl;
}

int methode(int a, int b)
{
  if(a > b)
    std::cout << "Type int : a(" << a << ") > b(" << b << ")." << std::endl;
  else if(a <= b)
    std::cout << "Type int : a(" << a << ") <= b(" << b << ")." << std::endl;
}

int
main( void )
{
  methode(4,2);       // Type int : a(4) > b(2).
  methode(1.0,2.0);   // Type double : a(1) <= b(2).
  methode(1.0,2);     // ERREUR : call of overloaded 'methode(double, int)' is
                      //          ambiguous
                      //          candidates are: double methode(double, double)
                      //                          int methode(int, int)
}
```

Qu'est ce qu'un opérateur ?

- Définition : un opérateur est une opération entre un ou deux opérandes, c'est-à-dire, entre une ou deux variables ou expressions

- On distingue plusieurs types d'opérateurs, comme par exemple :
  - les opérateurs arithmétiques $\rightsquigarrow$ $+, -, *, /$, ...
  - les opérateurs logiques $\rightsquigarrow$ $\&\&, !!, !$, ...
  - les opérateurs de comparaisons $\rightsquigarrow$ $>, >=, <, <=, ==, !=$, ...
  - les opérateurs d'incrémentation et de décrémentation $\rightsquigarrow$ $++$ et $--$,
  - l'opérateur d'affectation $\rightsquigarrow$ $=$, ...

Qu'est ce qu'un opérateur ?

- Définition : un opérateur est une opération entre un ou deux opérandes, c'est-à-dire, entre une ou deux variables ou expressions

- On distingue plusieurs types d'opérateurs, comme par exemple :
  - les opérateurs arithmétiques $\rightsquigarrow$ +, −, *, /, ...
  - les opérateurs logiques $\rightsquigarrow$ &&, !!, !, ...
  - les opérateurs de comparaisons $\rightsquigarrow$ > , >=, < , <=, ==, !=, ...
  - les opérateurs d'incrémentation et de décrémentation $\rightsquigarrow$ ++ et −−,
  - l'opérateur d'affectation $\rightsquigarrow$ =, ...

- En pratique, l'appel à un opérateur est similaire à un appel de méthode.

```
ObjetA a, b;
// ...

a OP_BINAIRE b; //                <===> a.operatorOP_BINAIRE(b);
OP_UNAIRE b;    //                <===> b.operatorOP_UNAIRE();
```

Qu'est ce qu'un opérateur ?

- Définition : un opérateur est une opération entre un ou deux opérandes, c'est-à-dire, entre une ou deux variables ou expressions

- On distingue plusieurs types d'opérateurs, comme par exemple :
  - les opérateurs arithmétiques $\rightsquigarrow$ $+, -, *, /, ...$
  - les opérateurs logiques $\rightsquigarrow$ $\&\&, !!, !, ...$
  - les opérateurs de comparaisons $\rightsquigarrow$ $>, >=, <, <=, ==, !=, ...$
  - les opérateurs d'incrémentation et de décrémentation $\rightsquigarrow$ $++$ et $--$,
  - l'opérateur d'affectation $\rightsquigarrow$ $=$, ...

- En pratique, l'appel à un opérateur est similaire à un appel de méthode.

```
ObjetA a, b;
// ...
a + b;          //          <===> a.operator+(b);
!b;             //          <===> b.operatornot();
std::cout << a; //          <===> cout.operator<<(a);
```

La surcharge d'opérateurs

- La surcharge d'opérateurs consiste donc à définir deux méthodes `operatorOP` avec des arguments différents
  - définition dans une classe A de la méthode `operatorOP` avec de nouveaux arguments
- En C++, on peut surcharger presque tous les opérateurs
  - sauf les opérateurs `::` (résolution de portée), `.`, `.*`, `?:`, `sizeof`, `typeid`, et les opérateurs de conversions du C++
- La surcharge peut se faire à l'intérieure ou à l'extérieure de la classe

La surcharge d'opérateurs

- La surcharge d'opérateurs consiste donc à définir deux méthodes `operatorOP` avec des arguments différents
  - définition dans une classe A de la méthode `operatorOP` avec de nouveaux arguments
- En C++, on peut surcharger presque tous les opérateurs
  - sauf les opérateurs `::` (résolution de portée), `.`, `.*`, `?:`, `sizeof`, `typeid`, et les opérateurs de conversions du C++
- La surcharge peut se faire à l'intérieure ou à l'extérieure de la classe
- Intérêt de surcharger un opérateur

```
// Sans surcharger l'operateur +
int
main( void )
{
  Matrix A, B, C;
  C = A.addition(B);
}
```

```
// En surchargeant l'operateur +
int
main( void )
{
  Matrix A, B, C;
  C = A + B;
}
```

Surcharger un opérateur à l'intérieure de la classe

⇝ par une méthode membre de la classe

- Pour surcharger un opérateur OP à l'intérieure d'une classe A, il faut ajouter la définition de la méthode prédéfinie `operatorOP` dans la classe A.

```
// A.hpp
class A
{
  // ...

public:
  // declaration de l'operateur
  typeDeRetour operatorOP(type1 parametre1, type2 parametre2, ...);
};
```

```
// A.cpp
// definition de l'operateur
typeDeRetour A::operatorOP(type1 parametre1, type2 parametre2, ...)
{
  // ...
}
```

Exemple de surcharge à l'intérieure de la classe

- On définit une classe Intervalle, qui représente un intervalle de valeurs

$$I = [b_{\text{inf}}, b_{\text{sup}}], \quad \text{avec } b_{\text{inf}} \text{ et } b_{\text{sup}} \text{ deux flottants.}$$

- On souhaite surcharger les opérateurs + et +=.

Exemple de surcharge à l'intérieure de la classe

⇝ dans la déclaration de la classe

- On définit une classe Intervalle, qui représente un intervalle de valeurs

$$I = [b_{\text{inf}}, b_{\text{sup}}], \quad \text{avec } b_{\text{inf}} \text{ et } b_{\text{sup}} \text{ deux flottants.}$$

- On souhaite surcharger les opérateurs + et +=.

```
class Intervalle
{
private:
  float binf, bsup;

public:
  Intervalle();
  Intervalle(float, float);
  Intervalle(Intervalle const&);

  void print(void);

  Intervalle& operator+=(Intervalle const&);          // Utilisation : I1 += I2;
  Intervalle operator+(Intervalle const&) const;      // Utilisation : I3 = I1 + I2;
};
```

Exemple de surcharge à l'intérieure de la classe

⇝ dans la définition des méthodes de la classe

```
Intervalle& Intervalle::operator+=(Intervalle const& I2)
{
  // Utilisation : I1 += I2;
  binf += I2.binf;
  bsup += I2.bsup;
  return *this;
}

Intervalle Intervalle::operator+(Intervalle const& I2) const
{
  // Utilisation : I3 = I1 + I2
  Intervalle I3(*this);
  I3 += I2;
  return I3;
}
```

```
int
main(void)
{
  Intervalle a, b(1,1), c;
  c = a + b;  // a.operator+(b)  --> 1. appel du constructeur par copie
              //                    2. appel de l'operateur +=
  c += b;     // c.operator+=(b)
  a.print(); b.print(); c.print();   // a -> [0,0],  b -> [1,1] et c -> [2,2]
}
```

Opérateur d'affectation et constructeur par copie

- L'opérateur d'affectation (=) et le constructeur par copie ont un comportement similaire
  - l'opérateur d'affectation est appelé lors d'une affectation $\rightsquigarrow$ `c = a + b;`
  - le constructeur par copie est appelé lors d'une initialisation $\rightsquigarrow$ `Point2D p2 = p1;`

- Une bonne pratique est d'implanter une méthode privée unique pour les deux, qui est appelée, à la fois, par l'opérateur d'affectation et par le constructeur par copie
  - on pourra appeler cette méthode `clone`, `copie`, ... par exemple.

Exemple de méthode `clone`

```
#ifndef __INTERVALLE_HPP__
#define __INTERVALLE_HPP__

class Intervalle
{
private:
  float binf, bsup;
  void clone(Intervalle const&);
              // Fonction de copie

public:
  Intervalle();
  Intervalle(float, float);
  Intervalle(Intervalle const&);

  void print(void);

  Intervalle& operator=(Intervalle const&);
              // Utilisation : I1 = I2;

  Intervalle& operator+=(Intervalle const&);
              // Utilisation : I1 += I2;

  Intervalle operator+(Intervalle const&)
       const;
              // Utilisation : I3 = I1 + I2;
};

#endif
```

```
Intervalle::Intervalle
      (Intervalle const& I)
{
  clone(I);
}

void Intervalle::clone
      (Intervalle const& I)
{
  binf = I.binf;
  bsup = I.bsup;
}

Intervalle& Intervalle::operator=
            (Intervalle const& I2)
{
  // Utilisation I1 = I2
  if(&I2 != this)
    clone(I2);
  return *this;
}
```

Surcharger un opérateur à l'extérieure de la classe

- La surcharge externe est utile pour des opérateurs concernés par une classe A :
  - mais pour lesquels l'opérande de gauche n'est pas un objet de la classe A,
  - mais pour lesquels l'opérande de gauche n'est pas l'objet modifié par l'opérateur.

- Exemples :
  - multiplication des éléments d'une matrice par un scalaire :
    ```
    Matrix A, B; double d; B = d * A;
    ```
  - multiplication de deux matrices :
    ```
    Matrix A, B, C; C = A * B;
    ```
  - écriture sur `cout` :
    ```
    Matrix C; std::cout << C;
    ```
    Ici, on a `std::cout.operator<<(C)`, mais on souhaitera le surcharger dans Matrix plutôt que dans la classe de `cout`.

Surcharger un opérateur à l'extérieure de la classe

- La surcharge externe est utile pour des opérateurs concernés par une classe A :
  - mais pour lesquels l'opérande de gauche n'est pas un objet de la classe A,
  - mais pour lesquels l'opérande de gauche n'est pas l'objet modifié par l'opérateur.

- Dans ces cas, on utilisera des opérateurs externes aux classes (ne faisant pas partie de ces classes)
  - déclarés avec un argument de plus que les opérateurs internes $\rightsquigarrow$ la classe sur laquelle s'applique l'opérateur

```
Intervalle operator+(Intervalle const&, Intervalle const&);

ostream& operator<<(ostream&, Intervalle const&);
```

- Il peut être utile de déclarer ces méthodes comme méthodes amies dans la classe concernée

Exemple de surcharge à l'extérieure de la classe

```
#ifndef __INTERVALLE_HPP__
#define __INTERVALLE_HPP__

class Intervalle
{
private:
  float binf, bsup;

public:
  Intervalle();
  Intervalle(float, float);
  Intervalle(Intervalle const&);

  Intervalle& operator+=(Intervalle const&);
      // Utilisation : I1 += I2;

  // Methode amie
  friend std::ostream& operator<<(std::ostream& , Intervalle const&);
      // Utilisation : std::cout << I1;
};

// Methode externe
Intervalle operator+(Intervalle const&, Intervalle const&);
     // Utilisation : I3 = I1 + I2;

#endif
```

Exemple de surcharge à l'extérieure de la classe

```
Intervalle operator+(Intervalle const& I1, Intervalle const& I2)
{
  // Utilisation : I3 = I1 + I2
  Intervalle I3(I1);
  I3 += I2;
  return I3;
}

std::ostream& operator<<(std::ostream& out, Intervalle const& I1)
{
  // Utilisation : std::cout << I1;
  out << "I = [ " << I1.binf << " , " << I1.bsup << " ]";
  return out;
}
```

```
int
main(void)
{
  Intervalle a, b(1,1), c;
  c = a + b;  // c = operator+(a,b)  --> 1. appel du constructeur par copie
              //                        2. appel de l'operateur =
  std::cout << c << std::endl;  // c --> [1,1]
}
```

Surcharge d'opérateur et héritage

- Les méthodes surchargeant les opérateurs dans une classe sont héritées dans les classes dérivées, sauf celles surchargeant l'opérateur d'affectation.

```
class A
{
public:
  A();
  A& operator+=(A const& a)
  { std::cout << "Operateur += dans A" << std::endl; }
  // ...
};

class B : public A
{ /* declaration de B */ };
```

```
int
main( void )
{
  B b1, b2;
  // ... initialisation de b1 et b2
  b2 += b1;                              // ===> Operateur += dans A
}
```

Plan du cours

1. Surcharge de méthodes et d'opérateurs

2. Polymorphisme en C++
   - Différents types de polymorphisme
   - Méthode virtuelle et classe abstraite

3. Suite et fin d'éléments importants du C++

Introduction au polymorphisme

- Le polymorphisme est le dernier des trois grands principes de la POO que l'on étudiera dans ce cours

- On distingue deux types de polymorphisme :
  - le polymorphisme de traitement (ad-doc) $\rightsquigarrow$ mécanisme de surcharge des méthodes (1 identificateur = plusieurs séquences d'instructions)
  - le polymorphisme de données (universel)
    - polymorphisme d'inclusion : un même code peut être appliqué à des données de type différents liées entre elles par une relation d'héritage
    - polymorphisme paramétrique : un même code peut être appliqué à n'importe quel type $\rightsquigarrow$ généricité

- Maintenant, on va s'intéresser au polymorphisme d'inclusion

Qu'est ce que le polymorphisme d'inclusion ?

- En C++, les instances d'une classe dérivée de A sont substanciables aux instances de la classe A, elle-même, en argument d'une méthode ou lors d'une affectation, tout en gardant leurs propriétés et leur nature propre.

- Mise en œuvre :
  - mécanisme d'héritage
  - résolution dynamique des liens $\rightsquigarrow$ le choix de la méthode à invoquer se fait à l'exécution du programme et va dépendre de la nature (type) des instances concernées

Résolution statique des liens

- En C++, le type de la variable détermine la méthode à exécuter.

```
class A
{
public:
  void print(void)
  {
    std::cout << "A" << std::endl;
  }

  // ...
};
```

```
class B : public A
{
public:
  void print(void)
  {
    std::cout << "B" << std::endl;
  }

  // ...
};
```

```
void execution(A const& obj)
{
  std::cout << "On est dans la classe : ";
  obj.print();
}

int
main( void )
{
  B b1; // ...
  execution(b1);
  // la methode de la classe A est appelee
}
```

Résolution dynamique des liens

- Il pourrait pourtant sembler plus naturel de vouloir exécuter la méthode correspondant à la nature réelle de l'instance $\rightsquigarrow$ ici la classe B
- Dans ce cas, il faut permettre la résolution dynamique de liens $\rightsquigarrow$ c'est-à-dire, le choix de la méthode à exécuter se décide à l'exécution du programme, en fonction de la nature réelle de l'instance
- Pour cela, on a besoin :
  - de manipuler soit des références, soit des pointeurs
  - de définir des méthodes virtuelles

Qu'est ce qu'une méthode virtuelle ?

- On considère une classe A, et une méthode `foo` de A. Si
  - la méthode `foo` est définie dans les classes dérivées de A,
  - et la méthode `foo` est souvent appelée via des pointeurs ou des références sur des objets de A ou des classes dérivées,

  alors, `foo` doit être déclarée comme méthode virtuelle.

- De cette manière, on indique au compilateur qu'une méthode pourra faire l'objet d'une résolution dynamique de liens.

- Plus généralement, on considèrera que pour être redéfinie dans une classe dérivée, une méthode devra être déclarée virtuelle dans la classe de base
  - on déclare une méthode virtuelle en utilisant le mot clé `virtual`

Intérêt d'une méthode virtuelle sur un exemple

```
#ifndef __A_HPP__
#define __A_HPP__

#include <iostream>

class A
{
public:
  // ...
  void print();
};

class B: public A
{
public:
  // ...
  void print();
};

#endif // __A_HPP__
```

```
#include "A.hpp"

void A::print()
{ std::cout << "Methode print de A" << std::endl; }

void B::print()
{ std::cout << "Methode print de B" << std::endl; }
```

```
// c4-expl1.cpp
#include <iostream>
#include "A.hpp"

int
main( void )
{
  A obj_a;  obj_a.print();
  B obj_b;  obj_b.print();

  A* ptr_a = new A();   ptr_a->print();
  B* ptr_b = new B();   ptr_b->print();

  A* ptr2_a;            B* ptr2_b = new B();
  ptr2_a = ptr2_b;      ptr2_a->print();

  delete ptr_a; delete ptr_b; delete ptr2_b;

  return 0;
}
```

Intérêt d'une méthode virtuelle sur un exemple

```cpp
#ifndef __A_HPP__
#define __A_HPP__

#include <iostream>

class A
{
public:
  // ...
  void print();
};

class B: public A
{
public:
  // ...
  void print();
};

#endif // __A_HPP__
```

```cpp
// c4-expl1.cpp
#include <iostream>
#include "A.hpp"

int
main( void )
{
  A obj_a;  obj_a.print();
  B obj_b;  obj_b.print();

  A* ptr_a = new A();   ptr_a->print();
  B* ptr_b = new B();   ptr_b->print();

  A* ptr2_a;            B* ptr2_b = new B();
  ptr2_a = ptr2_b;      ptr2_a->print();

  delete ptr_a; delete ptr_b; delete ptr2_b;

  return 0;
}
```

```
Methode print de A
Methode print de B
Methode print de A
Methode print de B
Methode print de A   <====
```

Intérêt d'une méthode virtuelle sur un exemple

```cpp
#ifndef __A_HPP__
#define __A_HPP__

#include <iostream>

class A
{
public:
  // ...
  virtual void print();
};

class B: public A
{
public:
  // ...
  void print();
};

#endif // __A_HPP__
```

```cpp
// c4-expl1.cpp
#include <iostream>
#include "A.hpp"

int
main( void )
{
  A obj_a;  obj_a.print();
  B obj_b;  obj_b.print();

  A* ptr_a = new A();   ptr_a->print();
  B* ptr_b = new B();   ptr_b->print();

  A* ptr2_a;            B* ptr2_b = new B();
  ptr2_a = ptr2_b;      ptr2_a->print();

  delete ptr_a; delete ptr_b; delete ptr2_b;

  return 0;
}
```

```
Methode print de A
Methode print de B
Methode print de A
Methode print de B
Methode print de B   <====
```

Règles s'appliquant sur les méthodes virtuelles

- Les constructeurs ne peuvent pas être déclarés virtuels, alors qu'il est fortement recommandé de déclarer les destructeurs virtuels $\rightsquigarrow$ cf. transparent suivant.
- Déclarer une méthode comme virtuelle dans une classe de base n'impose pas de redéfinir cette méthode dans toutes les classes dérivées, sauf si elle est déclarée virtuelle pure $\rightsquigarrow$ cf. transparent suivant.
- Une méthode déclarée virtuelle dans une classe de base la rend virtuelle dans toute la chaîne d'héritage.
- La signature de la méthode virtuelle dans les classes dérivées doit être la même que celle dans la classe de base, sinon, la méthode de la classe dérivée masque celle de la classe de base $\rightsquigarrow$ il n'y a pas de polymorphisme

Règles s'appliquant sur les méthodes virtuelles

- Si une méthode n'est pas déclarée virtuelle dans la classe de base, la même méthode dans une classe dérivée masque celle de la classe de base $\rightsquigarrow$ il n'y a pas de polymorphisme

- Une méthode déclarée virtuelle dans la zone privée d'une classe de base peut être redéfinie dans une classe dérivée, bien qu'elle ne pourra pas être appelée depuis la classe dérivée. Un appel à cette méthode depuis la classe de base utilise le comportement polymorphique, et la méthode de la classe dérivée sera bien appelée.

Exemple sur le polymorpisme

```
class ClasseA
{
public:
  void call_print(void){ print(); }
private:
  virtual void print(void){ std::cout << "Dans la classe ClasseA" << std::endl; }
};

class ClasseB : public ClasseA
{
private:
  void print(void){ std::cout << "Dans la classe ClasseB" << std::endl; }
};
```

```
int main(void)
{
  ClasseA a, *pta = new ClasseA(), *pt2a;
  ClasseB b, *ptb = new ClasseB();

  a.call_print();      b.call_print();

  pta->call_print();   ptb->call_print();

  pt2a = ptb;          pt2a->call_print();

  delete pta; delete pt2a; delete pta;
}
```

Exemple sur le polymorpisme

```
int main(void)
{
  ClasseA a, *pta = new ClasseA(), *pt2a;
  ClasseB b, *ptb = new ClasseB();

  a.call_print();      b.call_print();

  pta->call_print();   ptb->call_print();

  pt2a = ptb;          pt2a->call_print();

  delete pta; delete pt2a; delete pta;
}
```

```
On est dans la classe ClasseA
On est dans la classe ClasseB
On est dans la classe ClasseA
On est dans la classe ClasseB
On est dans la classe ClasseB
```

Pourquoi doit-on déclarer un destructeur virtuel ?

```
#ifndef __A_HPP__
#define __A_HPP__

#include <iostream>

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

class B: public A
{
public:
  B();
  ~B();
};

class C: public B
{
public:
  C();
  ~C();
};

#endif // __A_HPP__
```

```
int
main( void )
{
  C * tmp = new C();
  A * tmp2 = tmp;
  delete tmp2;
  return 0;
}
```

Pourquoi doit-on déclarer un destructeur virtuel ?

```
#ifndef __A_HPP__
#define __A_HPP__

#include <iostream>

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

class B: public A
{
public:
  B();
  ~B();
};

class C: public B
{
public:
  C();
  ~C();
};

#endif // __A_HPP__
```

```
int
main( void )
{
  C * tmp = new C();
  A * tmp2 = tmp;
  delete tmp2;
  return 0;
}
```

```
Constructeur de A
Constructeur de B
Constructeur de C
Destructeur de A
```

- Les destructeurs de B et C ne sont pas automatiquement appelés

Pourquoi doit-on déclarer un destructeur virtuel ?

```
#ifndef __A_HPP__
#define __A_HPP__

#include <iostream>

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

class B: public A
{
public:
  B();
  ~B();
};

class C: public B
{
public:
  C();
  ~C();
};

#endif // __A_HPP__
```

```
int
main( void )
{
  C * tmp = new C();
  A * tmp2 = tmp;
  delete tmp2;
  return 0;
}
```

```
Constructeur de A
Constructeur de B
Constructeur de C
Destructeur de A
```

- Les destructeurs de B et C ne sont pas automatiquement appelés
  - il faut déclarer le destructeur de A virtuel

Pourquoi doit-on déclarer un destructeur virtuel ?

```
#ifndef __A_HPP__
#define __A_HPP__

#include <iostream>

class A
{
public:
  A();
  virtual ~A();
};

class B: public A
{
public:
  B();
  ~B();
};

class C: public B
{
public:
  C();
  ~C();
};

#endif // __A_HPP__
```

```
int
main( void )
{
  C * tmp = new C();
  A * tmp2 = tmp;
  delete tmp2;
  return 0;
}
```

```
Constructeur de A
Constructeur de B
Constructeur de C
Destructeur de C
Destructeur de B
Destructeur de A
```

- À partir de maintenant, déclarer systématiquement le destructeur des classes de base en tant que virtuel

Qu'est ce qu'une classe abstraite ?

- Une méthode virtuelle est souvent introduite à un niveau élevé de la hiérarchie d'héritage
  - difficulté de donner une implantation à cette méthode
  - permet de définir une interface, que l'utilisateur devra compléter

- Une classe abstraite est une classe qui comporte au minimum une méthode virtuelle pure
  - méthode virtuelle pure $\rightsquigarrow$ ne possède pas d'implantation

```
// ClasseAbstraite.hpp
class ClasseAbstraite
{
public:
  ClasseAbstraite();
  virtual ~ClasseAbstraite();

  virtual void print() = 0;      // methode virtuelle pure
};
```

Remarques sur les classes abstraites

- Une classe abstraite ne peut pas être instanciée

```
// ClasseAbstraite.hpp
class ClasseAbstraite
{
public:
  ClasseAbstraite();
  virtual ~ClasseAbstraite();

  virtual void print() = 0;      // methode virtuelle pure
};
```

```
int
main(void)
{
  ClassesAbstraite ca1;                                  // Erreur de compilation
  ClassesAbstraite *ca2 = new ClasseAbstraite();  // Erreur de compilation
  // ...
}
```

Remarques sur les classes abstraites

- Une classe abstraite ne peut pas être instanciée

```
// ClasseAbstraite.hpp
class ClasseAbstraite
{
public:
  ClasseAbstraite();
  virtual ~ClasseAbstraite();

  virtual void print() = 0;      // methode virtuelle pure
};
```

```
int
main(void)
{
  ClassesAbstraite ca1;                                  // Erreur de compilation
  ClassesAbstraite *ca2 = new ClasseAbstraite();  // Erreur de compilation
  // ...
}
```

- Une classe dérivée doit définir les méthodes virtuelles pures de sa classe de base, ou également les déclarer virtuelles pures.

Exemple de classe abstraite

```
class Figure
{
protected:
  int x, y;
public:
  Figure(int, int);
  virtual void seDessiner(void) = 0;   // methode virtuelle pure... doit etre
                                       // definie dans Rectangle et Cercle
};
```

```
class Rectangle : public Figure
{
private:
  int largeur, hauteur;
public:
  Rectangle(int, int, int, int);
  void seDessiner(void);
};
```

```
class Cercle : public Figure
{
private:
  int rayon;
public:
  Cercle(int, int, int);
  void seDessiner(void);
};
```

Exemple de classe abstraite

```
Figure::Figure(int x, int y){ this->x = x; this->y = y; }
```

```
Rectangle::Rectangle(int x, int y, int l, int h):Figure(x,y)
{
  largeur = l;
  hauteur = h;
}

void Rectangle::seDessiner()
{
  std::cout << "Dessiner un rectangle de centre (" << x << "," << y << "), largeur
       " << largeur << " et hauteur " << hauteur << "." << std::endl;
}
```

```
Cercle::Cercle(int x, int y, int r):Figure(x,y)
{
  rayon = r;
}

void Cercle::seDessiner()
{
  std::cout << "Dessiner un cercle de centre (" << x << "," << y << ") et rayon "
       << rayon << "." << std::endl;
}
```

Exemple de classe abstraite

```
int
main(void)
{
  Rectangle * f1 = new Rectangle(1,2,3,4);
  Cercle* f2 = new Cercle(4,5,6);

  f1->seDessiner();
  f2->seDessiner();

  delete f1; delete f2;
}
```

```
Dessiner un rectangle de centre (1,2), largeur 3 et hauteur 4.
Dessiner un cercle de centre (4,5) et rayon 6.
```

Exemple de classe abstraite

⇝ utilisation du polymorphisme

```
int
main(void)
{
  Figure * f1 = new Rectangle(1,2,3,4);
  Figure * f2 = new Cercle(4,5,6);

  f1->seDessiner();
  f2->seDessiner();

  delete f1; delete f2;
}
```

Exemple de classe abstraite

⇝ utilisation du polymorphisme

```
int
main(void)
{
  Figure * f1 = new Rectangle(1,2,3,4);
  Figure * f2 = new Cercle(4,5,6);

  f1->seDessiner();
  f2->seDessiner();

  delete f1; delete f2;
}
```

```
Dessiner un rectangle de centre (1,2), largeur 3 et hauteur 4.
Dessiner un cercle de centre (4,5) et rayon 6.
```

Une remarque avant de finir

- Les modificateurs d'accès permettent de décider à la compilation la partie de code d'une classe (attributs/méthodes) accessible depuis l'extérieure de la classe et/ou de ces classes dérivées.

- Par contre, le mécanisme de polymorphisme permet de décider à l'exécution la méthode à appeler

Plan du cours

Attributs et méthodes `volatile`

- À la déclaration d'un attribut, le mot clé `volatile` permet d'indiquer au compilateur que cet attribut pourra être modifié par un processus externe, par la configuration matérielle, ...
  - permet d'éviter les optimisations à la compilation portant sur cet attribut
  - exemple d'optimisation : le compilateur évite de relire en mémoire la valeur d'une variable qui n'a pas été ré-affectée

- Lorsque l'on créé une instance d'une classe en utilisant le mot clé `volatile`, seules les méthodes `volatile` peuvent lui être appliquées

Attributs et méthodes `volatile`

⇝ exemple d'utilisation

```
class TestVolatile
{
public:
  static int x;
  // ...
};
```

```
int TestVolatile::x = 0;

int
main(void)
{
  while(TestVolatile::x != 1000)
    { /* ... do nothing ... */ }

  return 0;
}
```

- Remarque :
  - le test `while(TestVolatile::x != 1000)` est toujours vrai ⇝ un compilateur pourra optimiser cette partie de code : `while(true)`

Attributs et méthodes `volatile`

⇝ exemple d'utilisation

```
class ExplVolatile
{
public:
  void foo1(void)                          // declaration d'une fonction non-volatile
  { std::cout << "FOO 1" << std::endl; }
  void foo2(void) volatile                  // declaration d'une fonction volatile
  { std::cout << "FOO 2" << std::endl; }
};
```

```
int
main(void)
{
  ExplVolatile v1;
  volatile ExplVolatile v2;

  v1.foo1();                  // OK : v1 et foo1 ne sont pas volatile
  v1.foo2();                  // OK : v1 est non-volatile, et foo2 est volatile

  v2.foo1();                  // ERREUR : v2 est volatile, et foo1 non-volatile
  v2.foo2();                  // OK : v2 et foo2 sont volatile

  return 0;
}
```

Attributs `mutable`

- Le mot clé `mutable` devant un attribut permet de spécifier que cet attribut pourra être modifié, même si la méthode ou l'instance est déclarée `const`

```
class TestMutable
{
private:
  int a;
  mutable int b;
public:
  // ...
  void foo1()         { a++; }
  void foo2()  const  { a++; } // ERREUR : a n'est pas mutable, et ne peut
                               //   pas etre modifie par une methode const
  void foo3()         { b++; }
  void foo4()  const  { b++; } // OK : b est mutable
};

int
main(void)
{
  const TestMutable tmp;
  tmp.foo1();      // ERREUR : tmp est const, mais foo1 n'est pas const
  tmp.foo2();      // ERREUR : foo2 ne peut pas etre compilee
  tmp.foo3();      // ERREUR : tmp est const, mais foo3 n'est pas const
                   //   meme si b est mutable
  tmp.foo4();
  return 0;
}
```

Questions ?