La **programmation orientée objet** et le langage **C++**

Pablo Rauzy

rauzy@enst.fr

pablo.rauzy.name/teaching.html#epu-cpp

Ce cours est en partie inspiré du polycopié d'Henri Garreta.

EISE4 @ Polytech'UPMC

3 octobre 2014 — Cours 2

- Nouveautés du C++ par rapport au C.
  - Déclarations, `bool`, références, `inline`, arguments par défaut, surcharge des fonctions, I/O, `new`/`delete`.
- Programmation orientée objet, encapsulation.
- Les classes en C++.
  - Syntaxe, politique d'accès aux membres, programmation modulaire, constructeurs, destructeurs, membres constants, membres statiques, amie.

- Rappel constructeur :
  - le constructeur est appelé immédiatement après l'obtention de la mémoire,
  - puis les constructeurs des objets membres,
  - enfin il exécute le corps du constructeur.
- Rappel destructeur :
  - l'exécution commence par le corps du destructeur,
  - ensuite les destructeurs des objets membres,
  - enfin l'espace mémoire est libéré.

Code

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

class A {
public:
  A () { cout << "Construction A" << endl; }
  A (const A &a) { cout << "Construction par copie A" << endl; }
  ~A () { cout << "Destruction A" << endl; }
};

class B {
public:
  B () { cout << "Construction B" << endl; }
  B (const B &b) : a_(b.a_) { cout << "Construction par copie B" << endl; }
  B (A a) : a_(a) { cout << "Construction B(a)" << endl; }
  B (int n, A &a) : a_(a) { cout << "Construction B(&a)" << endl; }
  ~B () { cout << "Destruction B" << endl; }
private:
  A a_;
};

class C {
public:
  C () { cout << "Construction C" << endl; }
  C (const C &c) : b_(c.b_) { cout << "Construction par copie C" << endl; }
  C (A a) : b_(a) { cout << "Construction C(a)" << endl; }
  C (int n, A &a) : b_(0, a) { cout << "Construction C(&a)" << endl; }
  ~C () { cout << "Destruction C" << endl; }
private:
  B b_;
};
```

- Quelles vont être les sorties des programmes suivants :
  1. `A a;`
  2. `B b;`
  3. `C c;`
  4. `A a; B b(a);`
  5. `A a; A &a_ref = a; B b(0, a_ref);`
  6. `A a; C c(a);`
  7. `A a; A &a_ref = a; C c(0, a_ref);`

`A a;`

- Qu'affiche le programme `A a;` ?

`A a;`

- Qu'affiche le programme `A a;` ?

```
Construction A
Destruction A
```

`B b;`

- Qu'affiche le programme `B b;` ?

- Qu'affiche le programme `B b;` ?

```
Construction A
Construction B
Destruction B
Destruction A
```

C c;

- Qu'affiche le programme `C c;` ?

- Qu'affiche le programme `C c;` ?

```
Construction A
Construction B
Construction C
Destruction C
Destruction B
Destruction A
```

`A a; B b(a);`

- Qu'affiche le programme `A a; B b(a);` ?

- Qu'affiche le programme `A a; B b(a);` ?

```
Construction A
Construction par copie A
Construction par copie A
Construction B(a)
Destruction A
Destruction B
Destruction A
Destruction A
```

`A a; A &a_ref = a; B b(0, a_ref);`

- Qu'affiche le programme `A a; A &a_ref = a; B b(0, a_ref);` ?

`A a; A &a_ref = a; B b(0, a_ref);`

- Qu'affiche le programme `A a; A &a_ref = a; B b(0, a_ref);` ?

```
Construction A
Construction par copie A
Construction B(&a)
Destruction B
Destruction A
Destruction A
```

- Qu'affiche le programme `A a; C c(a);` ?

`A a; C c(a);`

- Qu'affiche le programme `A a; C c(a);` ?

```
Construction A
Construction par copie A
Construction par copie A
Construction par copie A
Construction B(a)
Destruction A
Construction C(a)
Destruction A
Destruction C
Destruction B
Destruction A
Destruction A
```

`A a; A &a_ref = a; C c(0, a_ref);`

- Qu'affiche le programme `A a; A &a_ref = a; C c(0, a_ref);` ?

`A a; A &a_ref = a; C c(0, a_ref);`

- Qu'affiche le programme `A a; A &a_ref = a; C c(0, a_ref);` ?

```
Construction A
Construction par copie A
Construction B(&a)
Construction C(&a)
Destruction C
Destruction B
Destruction A
Destruction A
```

Cours précédent
Correction Mini-TD
Surcharge des opérateurs
Héritage

- En C++, on peut étendre ou modifier la *sémantique* des *opérateurs* du langage.
- On appelle ça *surcharger les opérateurs*.

- On ne peut pas inventer de nouveau opérateurs, il faut qu'il existe déjà dans C++.
- Les opérateurs surchargés conservent leur arité, leur priorité, et leur associativité.
  - `a + b * c` sera toujours `a + (b * c)`.
- Leur commutativité peut changer, ainsi que leurs liens sémantiques avec les autres opérateurs.
  - `a += b` peut être différent de `a = a + b;`

Syntaxe

- Pour surcharger l'opérateur $\star$, on surcharge la fonction `operator` $\star$.
- Cela peut être une fonction seule ou une fonction membre d'une classe.

- On peut surcharger les opérateurs suivants :

```
+ - * / % ^ & | ~ ! , = < > <= >= ++ -- <<
>> == != && || += -= /= %= ^= &= |= *= <<=
>>= [] () -> ->* new new[] delete delete[]
```

- Mais pas ceux-ci : `::` `.*` `.` `?:`

Surcharge avec une fonction membre

- Si la fonction `operator` $\star$ est membre d'une classe, elle doit avoir un paramètre de moins que l'arité de l'opérateur $\star$.
- *`foo`* $\star$ ou $\star$ *`foo`* est équivalent à *`foo`*`.operator` $\star$`()`.
- *`foo`* $\star$ *`bar`* est équivalent à *`foo`*`.operator` $\star$`(`*`bar`*`)`.

Surcharge avec une fonction seule

- Si la fonction `operator` $\star$ n'est pas membre d'une classe, elle doit avoir un nombre de paramètres égal à l'arité de l'opérateur $\star$.
- *foo* $\star$ ou $\star$ *foo* est équivalent à `operator` $\star$`(`*foo*`)`.
- *foo* $\star$ *bar* est équivalent à `operator` $\star$`(`*foo*`, `*bar*`)`.

Exemple : surcharge d'opérateur (1/3)

```
class Time
{
public:
  Time (int seconds = 0);
  Time (int minutes, int seconds);
  Time (int hours, int minutes, int seconds);

  int seconds () const;
  int minutes () const;
  int hours () const;

  int time () const;

  Time operator + (const Time t) const;
  Time operator - (const Time t) const;
  Time operator * (const int n) const;
  friend ostream &operator << (ostream &out, Time t);
  friend istream &operator >> (istream &in, Time t);

private:
  int seconds_, minutes_, hours_;
};
```

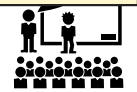
Exemple : surcharge d'opérateur (2/3)

```
Time::Time (int hours, int minutes, int seconds)
{
  int s = seconds + minutes * 60 + hours * 3600;
  seconds_ = s % 60;
  s /= 60;
  minutes_ = s % 60;
  s /= 60;
  hours_ = s;
}

//...

int Time::time () const
{
  return hours_ * 3600 + minutes_ * 60 + seconds_;
}

Time Time::operator + (const Time t) const
{
  return Time(hours_ + t.hours_, minutes_ + t.minutes_, seconds_ + t.seconds_);
}

//...

ostream &operator << (ostream &out, const Time t)
{
  out << setfill('0') << setw(2) << t.hours_
      << ':'
      << setfill('0') << setw(2) << t.minutes_
      << ':'
      << setfill('0') << setw(2) << t.seconds_;
  return out;
}
```

Exemple : surcharge d'opérateur (3/3)

```
class Distance
{
public:
  Distance (double meters = 0);

  double meters ();

  Area operator * (const Distance d) const;
  Speed operator / (const Time t) const;

private:
  double meters_;
};


class Speed
{
//...

  Distance operator * (const Time t) const;
  Acceleration operator / (const Distance d) const;

//...

private:
  double speed_;
};
```

Quel est l'intérêt de faire cela ?

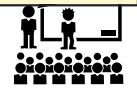


Affectation

- Pour les mêmes raisons qu'on a besoin d'un constructeur par copie, il est souvent utile de surcharger l'opérateur d'*affectation*.

```
class Position
{
public:
  Position (int x, int y, char *label);
  Position (const Position &p);
  ~Position ();

  int x ();
  int y ();
  char *label ();

  Position &operator = (const Position &p);

  Distance getDistanceTo (const Position &p);
  Distance getDistanceToOrigin ();

private:
  int x_, y_;
  char *label_;
};
```

Pourquoi on a besoin de redéfinir l'affectation ici ?



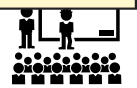
Conversions de types

- On a déjà vu comment faire une conversion (*cast*) d'un autre type vers une classe en utilisant un constructeur à un seul argument.
- Rappel : cela peut être dangereux, n'hésitez pas à rendre vos constructeur `explicit`.

- Une conversion d'une classe vers un autre type se fait en déclarant l'autre type comme opérateur.

```
class Distance
{
  //...

  operator Area ()
  {
    return Area(meters_ * meters_);
  }

  operator double ()
  {
    return meters_;
  }

  //...
};
```

[Rappel] Comment le cast vers sa classe fonctionne ?



- Une classe qui hérite d'une autre récupère ses membres et peut en définir de nouveaux ou redéfinir ceux qui sont hérités.
- Une classe peut hériter de plusieurs classes.

Vocabulaire

- La classe héritante est appelée *classe dérivée* (ou *sous classe*).
- Les classes héritées sont appelées *classes de base directes* (ou *super classe*).
- Une *classe de base* d'une classe dérivée est soit une classe de base directe, soit une classe de base directe d'une classe de base.

- On parle d'*héritage simple* quand il n'y a qu'une seule classe de base directe.
- Sinon, on parle d'*héritage multiple*.

Syntaxe

```
class Derived : policy1 Base1, policy2 Base2,...
{
  //...
}
```

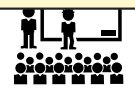
- `Derived` est la classe dérivée.
- `Base1`, `Base2`, ... sont les classes de base directes.
- `policy1`, `policy2`, ... sont les *politiques de dérivation* (cela sera détaillé plus tard).

Exemple : héritage

```
class Array
{
public:
  Array (length)
    : length_(length)
  {
    array_ = new int[length_];
  }

  ~Array ()
  {
    delete[] array_;
  }

  int &operator [] (int i)
  {
    if (i < 0 || i >= length_) {
      cerr << "index out of bound" << endl;
      exit(EXIT_FAILURE);
    }
    return array_[i];
  }

  int size () { return length_; }

private:
  int *array_;
  int length_;
};
```

```
class Stack : private Array
{
public:
  Stack (int max_size)
    : Array(max_size)
    , size_(0)
  {}

  void push (int n)
  {
    (*this)[size_++] = n;
  }

  int pop ()
  {
    return (*this)[--size_];
  }

  bool isEmpty () { return size_ == 0; }

  int size () { return size_; }

private:
  int size_;
};
```

Pourquoi l'opérateur [] retourne une référence ?



Héritage et accessibilité des membres

- Souvenez-vous, nous avons utilisé les *politiques d'accès aux membres* pour l'*encapsulation*.
- Nous avions alors des membres publics (`public`) et des membres privés (`private`).
- Ce concept s'étend aux membres protégés (`protected`) lors de l'héritage.

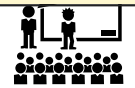
- Les membres protégés d'une classe sont accessibles par ses fonctions membres et amies (comme les privés), mais aussi par les fonctions membres et amies de ses classes dérivées directes.
- Ils ne font pas partie de l'interface de la classe, mais sont jugés utiles aux concepteurs de classes dérivées.

Exemple : héritage et accessibilité des membres

```
class Array
{
public:
  Array (length)
    : length_(length),
    , access_counter_(0)
  {
    array_ = new int[length_];
  }

  //...

  int &operator [] (int i)
  {
    if (i < 0 || i >= length_) {
      cerr << "index out of bound" << endl;
      exit(EXIT_FAILURE);
    }
    access_counter_++;
    return array_[i];
  }

  //...

protected:
  int access_counter_;

private:
  int *array_;
  int length_;
};
```

```
class Stack : private Array
{
public:
  //...

  bool isEmpty () {
    access_counter_++;
    return size_ == 0;
  }

  //...

private:
  int size_;
};
```

Pourquoi n'a-t-on pas redéfini `push` et `pop` ?



Héritage et politique de dérivation

- Trois politiques de dérivation.
  - Publique (`public`) :
    Les membres privés de la classe de base sont inaccessibles.
    Les membres publics et protégés gardent leur statut.
  - Protégée (`protected`) :
    Les membres privés de la classe de base sont inaccessibles.
    Les membres publics et protégés deviennent protégés.
  - Privée (`private`) :
    Les membres privés de la classe de base sont inaccessibles.
    Les membres publics et protégés deviennent privés.
- Notez qu'on ne peut pas augmenter l'accessibilité d'un membre.
- Quelque soit la politique de dérivation, on dit qu'un objet instance d'une classe dérivée a des *sous-objets* instances de ses classes de base.

Redéfinition des membres

- Si un membre d'une classe dérivée a le même nom qu'un membre d'une de ses classes de base, alors il $masque$ ce dernier.
- De manière générale, éviter les masquages de mêmes noms mais de signature différentes.
- En revanche, redéfinir seulement certains comportements d'une classe de base peut être très utile.

Construction d'objets instances de classes dérivées

- La construction d'un objet d'une classe dérivée se déroule comme suit :
  1. le constructeur est appelé immédiatement après l'obtention de la mémoire,
  2. il commence par appeler les constructeurs de chacune des classes de base directes *dans l'ordre de leur déclaration*,
  3. puis les constructeurs des objets membres *dans l'ordre de leur déclaration*,
  4. enfin il exécute le corps du constructeur.
- Par défaut les constructeurs sans arguments sont appelés, si ce n'est pas possible ou pas désiré, on peut les invoquer explicitement avec la même syntaxe que pour l'instanciation des objets membres.

Destruction d'objets instances de classes dérivées

- La destruction d'un objet d'une classe dérivée se déroule comme suit :
  1. l'exécution commence par le corps du destructeur,
  2. ensuite les destructeurs des objets membres *dans l'ordre inverse de leur déclaration*,
  3. puis les destructeurs de chacune des classes de base directes *dans l'ordre inverse de leur déclaration*,
  4. enfin l'espace mémoire est libéré.
- La destruction d'un objet a lieu à la fin de son espace de portée ou lors de l'appel explicite à son destructeur.

- Si une classe `D` dérive publiquement de `B` alors l'interface de `B` est entièrement accessible sur les objets de type `D`.
- On peut donc utiliser un `D` là où un `B` est prévu.

- Si l'objet est un pointeur ou une référence, on l'utilise alors comme un `B` mais il reste un `D`.
- Sinon, un "vrai" `B` est construit par copie.

Exemple : polymorphisme

```
class Shape
{
  //...

  friend double getDistanceBetween
    (Shape *a, Shape *b);

protected:
  double x_, y_;
  string label_;
};


class Square : public Shape
{
  //...

protected:
  double edge_length_;
};


class Circle : public Shape
{
  //...

protected:
  double radius_;
};
```

```
double getDistanceBetween
  (Shape *a, Shape *b)
{
  return sqrt(pow(a->x_ - b->x_, 2) +
              pow(a->y_ - b->y_, 2));
}

Circle *c = new Circle(42, 13, 2);
Square *s = new Square(51, 69, 3);

double d = getDistanceBetween(s, c);
```

Type statique, type dynamique

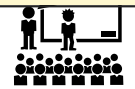
- Dans l'exemple précédent, les arguments `a` et `b` de la fonction `getDistanceBetween` sont de type `Shape`.
- Lors de l'appel de cette fonction on lui passe en fait un `Square` et un `Circle`.
- Dans ce cas on dit que
  - le *type statique* de `a` et `b` est `Shape`, et
  - le *type dynamique* de `a` est `Square` et celui de `b` est `Circle`.
- On accède à un objet par l'interface de son type statique.

Exemple : généralisation

```
class Shape
{
  //...

  void draw (Display *d);

  //...
};


class Square : public Shape
{
  //...

  void draw (Display *d);

  //...
};


class Circle : public Shape
{
  //...

  void draw (Display *d);

  //...
};
```

```
void drawShapes (Shape **list, int count)
{
  for (int i = 0; i < count; i++) {
    list[i]->draw();
}

Shape **drawing = new Shape *[2];

drawing[0] = new Circle(42, 13, 2);
drawing[1] = new Square(51, 69, 3);

drawShapes(drawing, 2);
```

Quel problème va-t-on rencontrer ici ?



Fonctions virtuelles

- Une fonction membre peut être déclarée comme étant *virtuelle* (avec le mot clef `virtual`).
- Toutes les redéfinitions dans les classes dérivées seront automatiquement virtuelles, et doivent avoir la même signature.
- À l'exécution, ce sera la version la *plus spécialisée* de la fonction qui sera appelée (par exemple celle du type dynamique).

```
class Shape
{
  //...

  virtual void draw (Display *d);

  //...
};
```

Fonctions virtuelles pures

- En fait, on va déclarer des fonctions virtuelles très haut dans la hiérarchie de classes.
- À ce niveau là, ces fonctions ne peuvent pas avoir d'implémentation (comme `draw` dans `Shape`).
- Ce qu'on veut en revanche c'est poser une contrainte sur les classes dérivées : celles qui seront effectivement utilisées devront fournir une implémentation de `draw` (ou l'hériter).
- Dans ce cas on fait de `draw` une *fonction virtuelle pure*.

```
class Shape
{
  //...

  virtual void draw (Display *d) = 0;

  //...
};
```

Classe abstraite

- Une classe qui possède au moins une fonction virtuelle pure est dites *abstraite*.
- On ne peut pas l'instancier.

Conversions dynamiques

- Syntaxe : `dynamic_cast<`*`type`*`>(`*`expression`*`)`.
- Conversion d'un type pointeur vers un autre type pointeur.
- Soit une généralisation (vers une classe de base).
- Soit une spécialisation (vers une classe dérivée plus proche du type dynamique).
- Renvoie le pointeur, ou `nullptr` quand la conversion n'est pas possible.
- Aussi utilisable avec des références (mais renvoie une exception en cas d'erreur).

- L'opérateur `static_cast` permet de faire la même chose statiquement (i.e., à la compilation), mais est moins sûr.

Identification dynamique

- Syntaxe : `typeid(`*`type`*`)` ou `typeid(`*`expression`*`)`.
- Renvoie un objet de type `type_info`.

```
class type_info
{
public:
  const char *name() const;
  int operator == (const type_info&) const;
  int operator != (const type_info&) const;
  int before (const type_info&) const;

  //...
};
```

C'est tout pour aujourd'hui :)