Programmation Orientée Objet — Episode #3

La Programmation Orientée Objet

Le paradigme

Description informelle

Conception à partir d'**objets** agissant les uns sur les autres.
Le plus souvent : objets = instances de **classes**.
Permet de définir leur **type**.

Les classes

Contiennent :

- les attributs **privés** (ou...)
- les méthodes **publiques** (ou...)

et donc :

- décrivent la structure interne des données
- définissent les méthodes qui s'appliqueront aux objets de même famille

Les constructeurs/Le destructeur

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

class Trace {

  public:
    Trace() {
        cout << this << " Constructeur
             sans argument" << endl;
    }

    Trace(Trace const &) {
        cout << this << " Constructeur
             par recopie" << endl;
    }

    ~Trace() {
        cout << this << " Destructeur"
             << endl;
    }
};
```

```cpp
#include "trace.hpp"

int main() {

  Trace t;
  return 0;

}
```

Les constructeurs/Le destructeur

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

class Trace {

  public:
    Trace() {
        cout << this << " Constructeur
             sans argument" << endl;
    }

    Trace(Trace const &) {
        cout << this << " Constructeur
             par recopie" << endl;
    }

    ~Trace() {
        cout << this << " Destructeur"
             << endl;
    }
};
```

```cpp
#include "trace.hpp"

void f(Trace ) { }

int main() {

  Trace t;
  f(t);
  return 0;

}
```

Les constructeurs/Le destructeur

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

class Trace {

  public:
    Trace() {
        cout << this << " Constructeur
             sans argument" << endl;
    }

    Trace(Trace const &) {
        cout << this << " Constructeur
             par recopie" << endl;
    }

    ~Trace() {
        cout << this << " Destructeur"
             << endl;
    }
};
```

```cpp
#include "trace.hpp"

void fp(Trace *) { }

int main() {

  Trace t;
  fp(&t);
  return 0;

}
```

Les constructeurs/Le destructeur

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

class Trace {

  public:
    Trace() {
        cout << this << " Constructeur
             sans argument" << endl;
    }

    Trace(Trace const &) {
        cout << this << " Constructeur
             par recopie" << endl;
    }

    ~Trace() {
        cout << this << " Destructeur"
             << endl;
    }
};
```

```cpp
#include "trace.hpp"

void fr(Trace &) { }

int main() {

  Trace t;
  fr(t);
  return 0;

}
```

Les constructeurs/Le destructeur

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

class Trace {

  public:
    Trace() {
        cout << this << " Constructeur
             sans argument" << endl;
    }

    Trace(Trace const &) {
        cout << this << " Constructeur
             par recopie" << endl;
    }

    ~Trace() {
        cout << this << " Destructeur"
             << endl;
    }
};
```

```cpp
#include "trace.hpp"

Trace id(Trace t) {
  return t;
}

int main() {

  Trace t;
  id(t);
  return 0;

}
```

Les constructeurs/Le destructeur

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

class Trace {

  public:
    Trace() {
        cout << this << " Constructeur
             sans argument" << endl;
    }

    Trace(Trace const &) {
        cout << this << " Constructeur
             par recopie" << endl;
    }

    ~Trace() {
        cout << this << " Destructeur"
             << endl;
    }
};
```

```cpp
#include "trace.hpp"

Trace idr(Trace &t) {
  return t;
}

int main() {

  Trace t;
  idr(t);
  return 0;

}
```

Les constructeurs/Le destructeur

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

class Trace {

  public:
    Trace() {
        cout << this << " Constructeur
             sans argument" << endl;
    }

    Trace(Trace const &) {
        cout << this << " Constructeur
             par recopie" << endl;
    }

    ~Trace() {
        cout << this << " Destructeur"
             << endl;
    }
};
```

```cpp
#include "trace.hpp"

Trace & idr(Trace &t) {
  return t;
}

int main() {

  Trace t;
  idr(t);
  return 0;

}
```

Les concepts fondamentaux

- Encapsulation [*] : **regrouper** données et méthodes.
- Héritage : inclure les caractéristiques d'une classe dans une autre.
- Polymorphisme : permet d'utiliser les objets de différentes classes au travers d'une *interface* commune tout en s'assurant que chacun exhibe son comportement spécifique.

[*] s'accompagne souvent du **masquage** des données.

L’encapsulation

Le principe

Regrouper données et méthodes et **souvent** protéger les attributs.
Mise en place (systématique ?) d'accesseurs et de mutateurs.

La classe `Point` — TBC

```
1  #ifndef POINT_HPP
2  #define POINT_HPP
3  #include <iostream>
4
5  class Point {
6
7    private: // par défaut --- TBC
8      int x;
9      int y;
10   public:
11     Point(); // aucun type de retour
12     Point(int, int); // Surcharge du constructeur
13     void afficher();
14
15     int get_x() { return x; }
16     int get_y() { return y; }
17
18 };
19 #endif
```

La classe `Point` — TBC

```cpp
#include "Point.hpp"

using namespace std;

Point::Point() {
  cout << "Constructeur par défaut." << endl;
  (*this).x = 0;
  (*this).y = 0;
}

Point::Point(int x, int y) {
  (*this).x = x;
  (*this).y = y;
}

void Point::afficher() {
  cout << "[" << (*this).x << "," << (*this).y << "]" << endl;
}
```

Pointeurs et encapsulation

Attention, danger. Attention, danger

```
1  #include <iostream>
2
3  class Tableau {
4
5    private:
6      int taille;
7      int *tableau;
8    public:
9      Tableau();
10     Tableau(int);
11     ~Tableau();
12
13     int acces(int);
14     int* get_tableau();
15
16 };
```

Attention, danger. Attention, danger

```cpp
#include "tableau.hpp"
using namespace std;
Tableau::Tableau(int taille) {
  (*this).taille = taille;
  tableau = new int[taille]();
}
Tableau::~Tableau() {
  delete[] tableau;
}
int Tableau::acces(int i) {
  return tableau[i];
}
int* Tableau::get_tableau() {
  return tableau;
}
```

Attention, danger. Attention, danger

```
#include "tableau.hpp"

int main() {
  using namespace std;

  Tableau t(10);
  for(int i = 0; i < 10; i++) cout << t.acces(i) << " ";
  cout << endl;

  int *pt = t.get_tableau();
  pt[3] = 2;

  for(int i = 0; i < 10; i++) cout << t.acces(i) << " ";
  cout << endl;

  return 0;
}
```

Bien comprendre

```
1  #include <iostream>
2  #include <vector>
3
4  class Tableau {
5
6    private:
7      int taille;
8      std::vector<int> tableau;
9    public:
10     Tableau();
11     Tableau(int);
12     ~Tableau();
13
14     int acces(int);
15     std::vector<int> get_tableau();
16
17 };
```

Bien comprendre

```cpp
#include "vecteur.hpp"
using namespace std;

Tableau::Tableau() { }

Tableau::Tableau(int taille) {
  (*this).taille = taille;
  tableau.resize(taille);
}

Tableau::~Tableau() { }

int Tableau::acces(int i) {
  return tableau[i];
}

vector<int> Tableau::get_tableau() {
  return tableau;
}
```

Bien comprendre

```
1  #include "vecteur.hpp"
2
3  int main() {
4    using namespace std;
5
6    Tableau t(10);
7    for(int i = 0; i < 10; i++) cout << t.acces(i) << " ";
8    cout << endl;
9
10   vector<int> pt = t.get_tableau();
11   pt[3] = 2;
12
13   for(int i = 0; i < 10; i++) cout << t.acces(i) << " ";
14   cout << endl;
15
16   return 0;
17 }
```

Les fonctions amies

Le concept

Une fonction **amie** d'une classe peut accéder à ses membres privés.

Mot-clé : `friend`

Pourquoi ne pas simplement être membre de la classe ?

Le fonctionnement

```
1  /* https://www.programiz.com/cpp-programming/ */
2  #include <iostream>
3  using namespace std;
4
5  class Distance
6  {
7    private:
8      int meter;
9    public:
10     Distance(): meter(0) { }
11     friend int addFive(Distance);
12 };
13
14 int addFive(Distance d) {
15   d.meter += 5;
16   return d.meter;
17 }
```

Le fonctionnement

```
1  #include "exemple.hpp"
2
3  int main()
4  {
5    Distance D;
6    cout<<"Distance: "<< addFive(D);
7    return 0;
8  }
```

Un exemple concret

Voir ici

```
#ifndef A_HPP
#define A_HPP
#include "B.hpp"

class B;
class A {
  private:
   int numA;
  public:
   A() { numA = 12; }
   friend int add(A, B);
};

#endif
```

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

int add(A a, B b) {
  return (a.numA + b.numB);
}

int main() {
  A a;
  B b;
  cout<<"Sum: "<< add(a, b) <<
    endl;
  return 0;
}
```

Un exemple concret

Voir ici

```
#ifndef B_HPP
#define B_HPP
#include "A.hpp"

class A;
class B {
  private:
    int numB;
  public:
    B() { numB = 1; }
    friend int add(A, B);
};

#endif
```

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

int add(A a, B b) {
  return (a.numA + b.numB);
}

int main() {
  A a;
  B b;
  cout<<"Sum: "<< add(a, b) <<
    endl;
  return 0;
}
```

Intérêt ?

- Opérations entre classes...
- ... **Surcharge d'opérateurs**

La surcharge d’opérateurs

Opérateur ?

Common operators						
assignment	increment decrement	arithmetic	logical	comparison	member access	other
a = b a += b a -= b a *= b a /= b a %= b a &= b a |= b a ^= b a <<= b a >>= b	++a --a a++ a--	+a -a a + b a - b a * b a / b a % b ~a a & b a | b a ^ b a << b a >> b	!a a && b a || b	a == b a != b a < b a > b a <= b a >= b a <=> b	a[b] *a &a a->b a.b a->*b a.*b	a(...) a, b ? :

Les flux — cout — 1/2

Le prototype

```
1  friend ostream& operator<< (ostream &os, const Point &p);
```

L'implémentation

```
1  ostream& operator<< (ostream &os, const
       Point &p) {
2    os << "[" << p.x << "," << p.y << "]";
3  return os;
4  }
```

```
1  #include <iostream>
2  #include "Point.hpp"
3
4  int main() {
5    using namespace std;
6
7    Point p(8, 12);
8    cout << p << endl;
9    return 0;
10 }
```

Important

De **très** nombreux concepts sont pour l'instant cachés :

- Référence en paramètre...
- ... retournée par la fonction ?
- Polymorphisme
- ...

De l'utilisation vers la compréhension.

Les flux — cin — 2/2

Le prototype

```
1  friend std::istream& operator>> (std::istream &in, Point &p);
```

L'implémentation

```
istream& operator>> (istream &is, Point &p) {
  cout << "Abscisse : ";
is >> p.x;
cout << "Ordonnée : ";
is >> p.y;
return is;
}
```

```
1  #include <iostream>
2  #include "Point.hpp"
3
4  int main() {
5    using namespace std;
6
7    Point p;
8    cin >> p;
9    cout << p << endl;
10   return 0;
11 }
```

Les comparaisons — 1/6

Le prototype

```
1  friend bool operator> (const Point &p, const Point &p1);
```

L'implémentation

```
bool operator> (const Point &p, const Point &p1)
     {
  if(p.x > p1.x)
    return true;
  else if (p.x < p1.x)
    return false;
  else
    return p.y > p1.y;
}
```

```
1  #include <iostream>
2  #include "Point.hpp"
3
4  int main() {
5    using namespace std;
6
7    Point p(8, 12);
8    Point p1(5,4);
9    cout << p << "\t" <<
         p1 << "\t" << (p
       > p1) << endl;
10   return 0;
11 }
```

Les comparaisons — 1/6

Le prototype

```
1  friend bool operator> (const Point &p, const Point &p1);
```

Implémentation possible *via* une fonction de comparaison :

```
1  inline bool operator> (const X& lhs, const X& rhs)
2    { return cmp(lhs,rhs) > 0; }
```

Le prototype et l'implémentation

```
1  friend inline bool operator< (const Point &p, const Point &p1) {
2    return p1 > p;
3  }
```

```
1  friend inline bool operator>= (const Point &p, const Point &p1)
       {
2    return !(p < p1);
3  }
4
5  friend inline bool operator<= (const Point &p, const Point &p1)
       {
6    return !(p > p1);
7  }
```

Les comparaisons — */6

Le prototype

```
1  friend bool operator== (const Point &p, const Point &p1);
```

L'implémentation

```
1  bool operator== (const Point &p, const Point &p1) {
2    return p.x == p1.x && p.y == p1.y;
3  }
```

Les comparaisons — */6

Le prototype

```
1 friend bool operator== (const Point &p, const Point &p1);
```

L'implémentation

```
1 friend inline bool operator!= (const Point &p, const Point &p1)
2   { return !(p == p1); }
```

Avec cmp

```
1  inline bool operator==(const X& lhs, const X& rhs)
2    { return cmp(lhs,rhs) == 0; }
3  inline bool operator!=(const X& lhs, const X& rhs)
4    { return cmp(lhs,rhs) != 0; }
5  inline bool operator< (const X& lhs, const X& rhs)
6    { return cmp(lhs,rhs) < 0; }
7  inline bool operator> (const X& lhs, const X& rhs)
8    { return cmp(lhs,rhs) > 0; }
9  inline bool operator<=(const X& lhs, const X& rhs)
10   { return cmp(lhs,rhs) <= 0; }
11 inline bool operator>=(const X& lhs, const X& rhs)
12   { return cmp(lhs,rhs) >= 0; }
```

Les comparaisons — */6

Une autre méthode (de classe)

```
1 bool operator< (const Point &p1)
```

L'implémentation

```
1 bool Point::operator< (const Point &p1) {
2   return p1 > (*this);
3 }
```

L'instruction `p < p1` s'interprète en :

`p.operator<(p1)`

L'opérateur []

Le prototype

```
1 // Qu'implique la référence ?
2 int& operator[](std::size_t id);
```

Méthode `friend` ?

L'opérateur []

Le prototype

```
1 // Qu'implique la référence ?
2 int& operator[](std::size_t id);
```

Méthode **de classe** : les opérateurs =, [], () et -> **doivent** être de classe.

L'opérateur []

Le prototype

```
1 // Qu'implique la référence ?
2 int& operator[](std::size_t id);
```

Les opérateurs << et >> ne **peuvent pas** être de classe.

L'opérateur []

Le prototype

```
1  // Qu'implique la référence ?
2  int& operator[](std::size_t id);
```

L'implémentation

```
int& Point::operator[](std::size_t id) {
  if(id == 0) return (*this).x;
  else return (*this).y;
}
```

```
1  #include <iostream>
2  #include "Point.hpp"
3  int main() {
4    using namespace std;
5    Point p(8, 12);
6    cout << p[0] << "\t"
         << p[1] << endl;
7    p[0] = 2;
8    cout << p << endl;
9  }
```

L'affectation

Le prototype

```
1  Point& operator=(const Point& p);
```

L'implémentation

```
Point& Point::operator=(const Point& p) {
  (*this).x = p.x;
  (*this).y = p.y;
  return *this;
}
```

```
1  #include <iostream>
2  #include "Point.hpp"
3
4  int main() {
5    using namespace std;
6
7    Point p(8,12);
8    Point p1(5,4);
9    p = p1;
10   cout << p << "\t" <<
         p1 << endl;
11   return 0;
12 }
```

Méthode amie ou méthode de classe ?

Quelques règles :

- Les opérateurs `=`, `[]`, `()` et `->` **doivent** être de classe.
- Opérateur unaire : méthode de classe.
- Opérateur binaire :
  modifiant la partie gauche : méthode de classe (si possible).
  ne modifiant pas la partie gauche (`+`) : méthode amie ou normale.

Les opérations (+) — 1/?

```
1  #include <iostream>
2  class Simple {
3    int n;
4    public:
5      Simple(int _n) { (*this).n = _n; }
6      Simple operator+(int _y) { return Simple((*this).n + _y); }
7
8      friend std::ostream& operator<<(std::ostream& os, const
       Simple &s)
9        { os << s.n; return os; }
10 };
11 int main() {
12   Simple s(10);
13   std::cout << s + 3 << std::endl;
14 }
```

Les opérations (+) — 1/?

```
1  #include <iostream>
2  class Simple {
3    int n;
4    public:
5      Simple(int _n) { (*this).n = _n; }
6      Simple operator+(int _y) { return Simple((*this).n + _y); }
7
8      friend std::ostream& operator<<(std::ostream& os, const
       Simple &s)
9        { os << s.n; return os; }
10 };
11 int main() {
12   Simple s(10);
13   std::cout << 3 + s << std::endl;
14 }
```

Les opérations (+) — 1/?

```
1  #include <iostream>
2  class Simple {
3    int n;
4    public:
5      Simple(int _n) { (*this).n = _n; }
6      Simple operator+(int _y) { return Simple((*this).n + _y); }
7
8      friend std::ostream& operator<<(std::ostream& os, const
       Simple &s)
9        { os << s.n; return os; }
10     friend Simple operator+(int _n, const Simple &s)
11       { return Simple(s.n + _n); }
12 };
13 int main() {
14   Simple s(10);
15   std::cout << 3 + s << std::endl;
16 }
```