# Programmation Orientée Objets 2
# L3 MI
# Cours 3

Rémi Watrigant[1]


[1] basé sur le cours de Marianne Huchard

# Sommaire du cours 3

# Opérateurs

Cas d'étude:

```
class Heure {
protected:
     int nbHeures;
     int nbMinutes;
     int nbSecondes;
public:
     Heure();
     Heure(int, int int);
     Heure(string);
     ~Heure();
     ...
};
```

# Opérateurs

Cas d'étude:

```
class Heure {
protected:
     int nbHeures;
     int nbMinutes;
     int nbSecondes;
public:
     Heure();
     Heure(int, int int);
     Heure(string);
     ~Heure();
     ...
};
```

Additionner deux heures:

```
Heure h1(2, 56, 23), h2(0, 4, 43);
Heure h3 = h1.sum(h2);
```

# Opérateurs

Cas d'étude:

```
class Heure {
protected:
     int nbHeures;
     int nbMinutes;
     int nbSecondes;
public:
     Heure();
     Heure(int, int int);
     Heure(string);
     ~Heure();
     ...
};
```

Additionner deux heures:

```
Heure h1(2, 56, 23), h2(0, 4, 43);
Heure h3 = h1.sum(h2);
```

Possibilité de surcharger les opérateurs:

```
Heure h1(2, 56, 23), h2(0, 4, 43);
Heure h3 = h1 + h2;
```

# Opérateurs

Possibilité de surcharger les opérateurs:

```
Heure h1(2, 56, 23), h2(0, 4, 43);
Heure h3 = h1 + h2;
```

En fait, l'utilisation de l'opérateur + équivaut à:

```
operator+(h1, h2);
```

$\Rightarrow$ surcharge de l'opérateur operator+

# Opérateurs

Dans le fichier Heure.h :

```
class Heure {
protected:
     ...
public:
     ...
     operator+(Heure const& a);
};
```

Dans Heure.cpp :

```
Heure Heure::operator+(Heure const& a)
{
int nbSecondes = a.getNbSecondes() + b.getNbSecondes();
int nbMinutes = a.getNbMinutes() + b.getNbMinutes() + nbSecondes/60;
nbSecondes %= 60;
int nbHeures = a.getNbHeures() + b.getNbHeures() + nbMinutes/60;
nbMinutes %= 60;

Heure copie(nbHeures, nbMinutes, nbSecondes);
    return copie;
}
```

Possibilité de définir des opérateurs avec différents types:
operator+(int a);

# Opérateurs

Opérateurs pouvant être surchargés:

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

Attention : certains opérateurs ne peuvent pas être définis comme méthode de classe:
Exemple : opérateur «, qui ne fonctionne qu'avec des istream, ostream
Heure h1();
cout « h1 « endl;

# Opérateurs

Exemple : l'opérateur «
Heure.h

```
class Heure {
    ...
};
ostream &operator<<( ostream &out, Heure const& h );
```

Heure.cpp

```
ostream &operator<<( ostream &out, Heure const& h ) {
     out << h.getNbHeures() << "h " << h.getNbMinutes() << "min " << h.getNbSecondes
     return out;
}
```

Inconvénient:

- pas accès aux attributs de l'objet
- réécriture lors de l'héritage

# Opérateurs

Exemple : l'opérateur « : SOLUTION
Heure.h

```
class Heure {
public:
     void affiche(ostream& out);
};
ostream &operator<<( ostream &out, Heure const& h );
```

Heure.cpp

```
void Heure::affiche(ostream& out) {
     out << nbHeures() << "h " << nbMinutes() << "min " << nbSecondes() << "s " << e
}

ostream &operator<<( ostream &out, Heure const& h ) {
     h.affiche(out);
     return out;
}
```

Permet la redéfinition de la méthode affiche pour les classes filles

# Conversions, transtypage

# Conversions, transtypage

Plusieurs manières :

- Affectation polymorphiques : classe vers super-classe :

```
CompteBancaire ca();
CompteBancaireCarteBleue caCB();
ca = caCB;
```

- Utilisation de constructeurs
- Utilisation d'opérateurs

# Conversions, transtypage

- Utilisation de constructeurs :
  Règle : constructeur avec un unique paramètre = opérateur de conversion

```
Heure* h1 = new Heure("21 03 56");  //format hh mm ss
```

Attention :

```
void foo(Heure h) {
      ...
}

...

foo("21 03 56");       //va convertir la chaine en Heure
```

Possibilité d'interdire ce comportement : mot clé explicit

```
class Heure {
protected:
       ....
public:
      explicit Heure(string);
}
```

# Conversions, transtypage

- Utilisation d'opérateurs:

```
class Heure {
protected:
        ....
public:

          virtual operator string();


}
```

Utilisation :

```
 Heure h1();

 cout << string(h1) << endl;
```

# Surcharge versus Redéfinition : redéfinition, virtual

# Surcharge versus Redéfinition : redéfinition, virtual

```
class Base {
public:
     void f();
     virtual void g();
};
void Base::f() { cout << "Base::f()" << endl;}
void Base::g() { cout << "Base::g()" << endl;}

class Derive : public Base {
public:
     void f();
     virtual void g();
};
void Derive::f() { cout << "Derive::f()" << endl;}
void Derive::g() { cout << "Derive::g()" << endl;}

int main(int argc, char** argv) {
    Base* d1 = new Derive();
    Derive* d2 = new Derive();

    d1->f();   //Base::f()
    d1->g();   //Derive::g()
    d2->f();   //Derive::f()
    d2->g();   //Derive::g()
```

# Surcharge versus Redéfinition : surcharge

```
class Base {
public:
     virtual void f();
};
void Base::f() { cout << "Base::f()" << endl;}


class Derive : public Base {
public:
     virtual void f( int );
};
void Derive::f(int x) { cout << "Derive::f( int )" << endl;}


int main(int argc, char** argv) {
    Derive* d = new Derive();

    d->f( 50 );       //OK
    d->f()            //ERREUR DE COMPILATION ! surcharge => masquage des méthodes hé
```

# Surcharge versus Redéfinition : surcharge

```
class Base {
public:
     void f();
};
void Base::f() { cout << "Base::f()" << endl;}


class Derive : public Base {
public:
     using Base::f();          ///permet d'utiliser la méthode héritée
     virtual void f( int );
};
void Derive::f(int x) { cout << "Derive::f( int )" << endl;}


int main(int argc, char** argv) {
    Derive* d = new Derive();

    d->f( 50 );       //OK
    d->f()            //OK
```

# Elements statiques et amitié

# Elements statiques et amitié

Attributs et Méthodes statiques d'une classe : accessibles sans instancier d'objets :
Exemple.h

```
class Exemple {
public:
     static void uneMethode();
     static int unAttribut;
};
```

Exemple.cpp

```
int Exemple::unAttribut = 1337;

void Exemple::uneMethode() {
     cout << "hello world!" << endl;
}
```

main.cpp

```
int main(int argc, char** argv) {
    cout << Exemple::monAttribut << endl;
    Exemple::uneMethode();

    return 0;

}
```

# Elements statiques et amitié

Exemple : compteur d'instances:
Personne.h

```
class Personne {
private:
    string nom;
    static int nbPersonnes;
public:
     Personne(string);
     ~Personne();
     static int getNbPersonnes();
};
```

Personne.cpp

```
int Personne::nbPersonnes = 0;
int Personne::getNbPersonnes() {
     return nbPersonnes;
}
Personne::Personne(string nom) {
     this-> nom = nom;
     nbPersonnes++;
}
Personne::~Personne() {
     nbpersonnes--;
}
```

# Elements statiques et amitié

Exemple : compteur d'instances:
main.cpp

```
int main(int argc, char** argv) {
    Personne* p1 = new Personne("toto");
    Personne* p2 = new Personne("tata");

    cout << "Il y a " << Personne::getNbPersonnes() << " personnes instanciées" << e

    return 0;
}
```

# Elements statiques et amitié

Amitié : permet la définition de fonctions (hors classe) ayant accès aux attributs et méthodes privées d'une classe

```
class maClasse {
protected:
   int attribut;
public:
   ...

friend void fonctionAmie();
};
```

- La fonction *fonctionAmie* a accès aux attributs et classes privées de la classe : /!\ Plus d'encapsulation !
- Pas de notion de public private ou protected pour les fonctions amies
- Possibilité de déclarer toute une classe amie

# Elements statiques et amitié

Utilité des fonctions amies :
Heure.h

```
class Heure {
    ...

    friend ostream &operator<<( ostream &out, Heure const& h );

};
```

Heure.cpp

```
ostream &operator<<( ostream &out, Heure const& h ) {
     out << h.nbHeures << "h " << h.nbMinutes << "min " << h.nbSecondes << "s " << e
     return out;
}
```