Plan

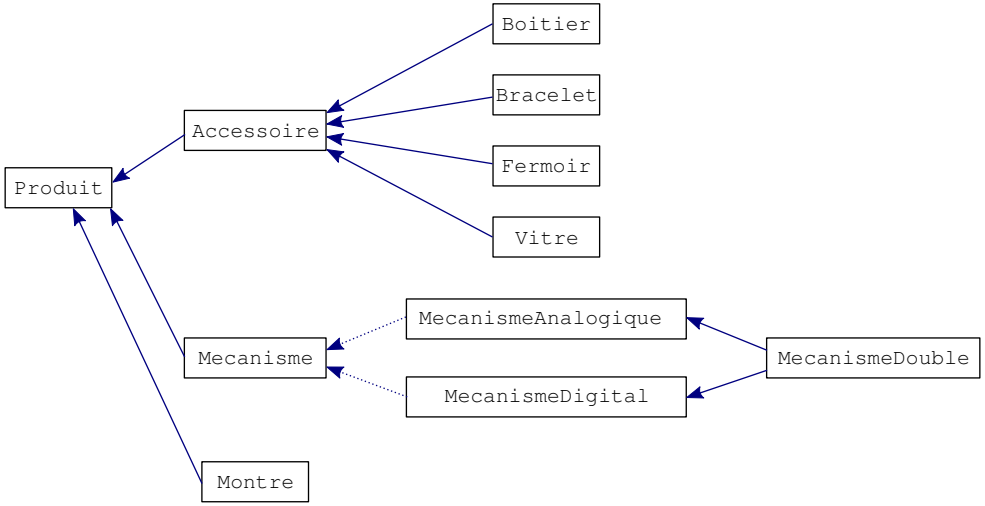
Problématiques abordées :

- conception POO, héritage, classes abstraites (prix, affichage)
- affichage polymorphique
- surcharge d'opérateurs
- héritage multiple
- copie polymorphique

Le problème

- Les *montres* sont des *produits* (que l'on peut vendre : ont un prix)
- Les montres ont un *mécanisme* de base et sont constituées de différents *accessoires* (boîtier, bracelet, ...)
- Les *produits* ont un prix dont le calcul peut varier, à partir d'une valeur de base
- Tous les produits sont « affichables », chacun à sa façon
- Les *mécanismes* et *accessoires* de montre sont aussi des produits (on pourrait en acheter séparément)
- Il existe trois sortes de *mécanismes* : *analogiques* (montre à aiguilles), *digitaux* et *doubles*.
- Pour les *mécanismes doubles*, on supposera ici qu'ils n'indiquent qu'une seule heure, mais se comportent sinon à la fois comme des *mécanismes analogiques* et comme des *mécanismes digitaux*

Hiérarchie de classes



Du problème au premier code

- Les *montres* sont des *produits*

```
class Produit {
};

// ========================================
class Montre : public Produit {
};

// ========================================
int main() {

  return 0;
}
```

Du problème au premier code

- Les montres ont un *mécanisme* et sont constituées de différents *accessoires*

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

// ...

// ========================================
class Accessoire {
};

// ========================================
class Mecanisme {
};

// ========================================
class Montre : public Produit {
private:
  Mecanisme coeur;
  vector<Accessoire> accessoires;
};
```

Du problème au premier code

- Les montres ont un *mécanisme* et sont constituées de différents *accessoires*

```
#include <memory>
#include <vector>
using namespace std;
// ...

// ========================================
class Accessoire {
};

// ========================================
class Mecanisme {
};

// ========================================
class Montre : public Produit {
private:
  unique_ptr<Mecanisme> coeur;
  vector<unique_ptr<Accessoire>> accessoires;
};
```

Du problème au premier code

- Les montres ont un *mécanisme* et sont constituées de différents *accessoires*

```
// ...

// ========================================
class Accessoire {
};

// ========================================
class Mecanisme {
};

// ========================================
class Montre : public Produit {
private:
  unique_ptr<Mecanisme> coeur;
  vector<unique_ptr<Accessoire>> accessoires;

  Montre(const Montre&)     = delete;
  Montre& operator=(Montre) = delete;
};
```

Du problème au premier code

- Les *produits* ont un prix

```
#include <memory>
#include <vector>
using namespace std;

// ========================================
class Produit {
private:
  double prix;
};

// ...
```

Du problème au premier code

- Les *produits* ont un prix dont le calcul peut varier, à partir d'une valeur de base

```
#include <memory>
#include <vector>
using namespace std;

// ========================================
class Produit {
public:
  virtual double prix() const
  { return valeur; }
private:
  double valeur;
};

// ...
```

Du problème au premier code

- Tous les produits sont « affichables » chacun à sa façon

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

// ========================================
class Produit {
public:
  virtual double prix() const
  { return valeur; }
private:
  double valeur;
};

// On y revient plus tard
ostream& operator<<(ostream& sortie,
                    Produit const&);

// ...
```

Du problème au premier code

- Les *mécanismes* et *accessoires* sont aussi des produits

```
// ...

// ========================================
class Accessoire : public Produit {
};

// ========================================
class Mecanisme : public Produit {
};

// ========================================
class Montre : public Produit {
private:
  unique_ptr<Mecanisme> coeur;
  vector<unique_ptr<Accessoire>> accessoires;

  Montre(const Montre&)     = delete;
  Montre& operator=(Montre) = delete;
};
```

Du problème au premier code

- Il existe trois sortes de *mécanismes* : *analogiques*, *digitaux* et *doubles*

```
// ...

// ========================================
class Mecanisme : public Produit {
};

// ========================================
class MecanismeAnalogique : public Mecanisme
{};

// ========================================
class MecanismeDigital : public Mecanisme {
};

// ========================================
class MecanismeDouble : public Mecanisme {
};

// ...
```

Du problème au premier code

- ...
- Tous les produits sont « affichables » chacun à sa façon
- Les *mécanismes* et *accessoires* sont aussi des produits
- Il existe trois sortes de *mécanismes* : *analogiques*, *digitaux* et *doubles*
- Pour les *mécanismes doubles*, on supposera ici qu'ils n'indiquent qu'une seule heure, mais se comportent sinon à la fois comme des *mécanismes analogiques* et comme des *mécanismes digitaux*

```
// ...

// ========================================
class Mecanisme : public Produit {
};

// ========================================
class MecanismeAnalogique : public Mecanisme
{};

// ========================================
class MecanismeDigital : public Mecanisme {
};

// ========================================
class MecanismeDouble : public Mecanisme {
};

// ...
```

Affichage polymorphique

- Tous les produits sont « affichables », chacun à sa façon

☞ affichage polymorphique ?

Le plus simple est de faire appel à une méthode polymorphique définie au niveau de la super-classe :

```
class Produit {
public:
  virtual ~Produit() {}
  virtual void afficher(ostream& sortie) const ;
};

// ------------------------------
ostream& operator<<(ostream& sortie, Produit const& machin) {
  machin.afficher(sortie);
  return sortie;
}
```

Affichage par défaut

Précisons encore le comportement par défaut :

```
virtual void Produit::afficher(ostream& sortie) const {
  sortie << prix();
}
```

Finalisation de la classe `Produit`

Supposons enfin que :

- l'on fixe la valeur de base d'un produit au départ et que l'on ne puisse pas en changer

```
class Produit {
public:
  Produit(double une_valeur)
  : valeur(une_valeur)
  {}

  // ...

private:
  const double valeur;
};
```

Finalisation de la classe `Produit`

Supposons enfin que :

- un produit ait par défaut une valeur de base nulle

```
class Produit {
public:
  Produit(double une_valeur = 0.0)
  : valeur(une_valeur)
  {}

  // ...

};
```

Finalisation de la classe `Produit`

Supposons enfin que :

- un produit n'existe pas en tant que tel : c'est une *abstraction*

```
class Produit {
public:
  Produit(double une_valeur)
  : valeur(une_valeur)
  {}

  virtual ~Produit() = 0;

  // ...

};

Produit::~Produit() {}
```

La classe `Produit`

```
class Produit {
public:
  Produit(double une_valeur = 0.0) : valeur(une_valeur) {}

  virtual ~Produit() = 0;

  virtual double prix()                    const  { return valeur;    }
  virtual void   afficher(ostream& sortie) const  { sortie << prix(); }

private:
  const double valeur;
};

Produit::~Produit() {}

ostream& operator<<(ostream& sortie, Produit const& machin) {
  machin.afficher(sortie);
  return sortie;
}
```

Ajout d'accessoires (aux `Montre`s)

Rappel :

```
class Montre : public Produit {
private:
  unique_ptr<Mecanisme> coeur;
  vector<unique_ptr<Accessoire>> accessoires;

  Montre(const Montre&)     = delete;
  Montre& operator=(Montre) = delete;
};
```

On souhaite ajouter des accessoires à une montre au moyen de l'opérateur `+=`.

Par exemple :

```
montre += new Bracelet(...);
```

On a donc ici comme prototype (surcharge interne) :

```
void Montre::operator+=(Accessoire* ajout);
```

Ajout d'accessoires (aux `Montre`s)

```
class Montre : public Produit {
public:
  void operator+=(Accessoire* p_accessoire) {
    accessoires.push_back(unique_ptr<Accessoire>(p_accessoire));
  }

private:
  unique_ptr<Mecanisme> coeur;
  vector<unique_ptr<Accessoire>> accessoires;

  Montre(const Montre&)     = delete;
  Montre& operator=(Montre) = delete;
};
```

Finalisation d'une première version

Essayons maintenant d'avoir une première version opérationnelle de notre code, pour le moment :

- *sans* tous les mécanismes
- *sans* copie des montres

Pour cela, il nous faut encore :

- quelques *accessoires*
- terminer la classe `Montre`
- un exemple d'utilisation dans le `main()`

Quelques accessoires

Décidons par exemple que les accessoires :

- ont un nom et une valeur de base fixés au départ (sans valeur par défaut)

```
class Accessoire : public Produit {
public:
  Accessoire(string const& un_nom,
             double prix_de_base)
  : Produit(prix_de_base), nom(un_nom)
  {}

private:
  const string nom;
};
```

Quelques accessoires

Décidons par exemple que
les accessoires :

- s'affichent en indiquant leur nom et leur prix

```
class Accessoire : public Produit {
public:
  Accessoire(string const& un_nom,
             double prix_de_base)
  : Produit(prix_de_base), nom(un_nom)
  {}

  virtual ~Accessoire() {}

  virtual void afficher(ostream& sortie)
                              const override {
    sortie << nom << " coûtant ";
    Produit::afficher(sortie);
  }

private:
  const string nom;
};
```

Quelques accessoires

Décidons par exemple que
les accessoires :

- que leur prix est celui d'un produit usuel

Quelques accessoires

```
class Bracelet : public Accessoire {
public:
  Bracelet(string const& un_nom, double prix_de_base)
    : Accessoire("bracelet " + un_nom, prix_de_base)
  {}
  virtual ~Bracelet() {}
};

//-------------------------------------------------------------
class Fermoir : public Accessoire {
public:
  Fermoir(string const& un_nom, double prix_de_base)
    : Accessoire("fermoir " + un_nom, prix_de_base)
  {}
  virtual ~Fermoir() {}
};

// ...
```

Finalisation des `Montre`s

Pour finaliser la classe `Montre`
(mais sans mécanisme) :

- ajoutons un constructeur (au moins remettre le constructeur par défaut)

```
class Montre : public Produit {
public:
  Montre() = default;
  virtual ~Montre() {}

  // ...

private:
  // unique_ptr<Mecanisme> coeur;

  // ...

};
```

Finalisation des Montres

Pour finaliser la classe Montre
(mais sans mécanisme) :

- décidons d'un calcul de prix :
  somme des prix des accessoires

```
  // ...
  virtual double prix() const override {

  // Au départ, le prix est la valeur de base
    double prix_final(Produit::prix());

    for (auto const& p_acc : accessoires) {
      prix_final += p_acc->prix();
    }

    return prix_final;
  }
  // ...
private:
  // unique_ptr<Mecanisme> coeur;
  vector<unique_ptr<Accessoire>> accessoires;
  // ...
};
```

Finalisation des Montres

Pour finaliser la classe Montre
(mais sans mécanisme) :

- décidons d'un affichage :

```
Une montre composée de :
  * bracelet cuir coûtant 54
  * fermoir acier coûtant 12.5
  * etc.
==> Prix total : 147.9
```

```
// ...
virtual void afficher(ostream& sortie)
                              const override {

  sortie << "Une montre composée de :"
         << endl;

  for (auto const& p_acc : accessoires) {
    sortie << "  * " << *p_acc << endl;
  }

  sortie << "==> Prix total : " << prix()
         << endl;
}
// ...
```

Un exemple de main()

```
int main() {

  Montre m;

  m += new Bracelet("cuir", 54.0);
  m += new Fermoir("acier", 12.5);
  m += new Boitier("acier", 36.60);
  m += new Vitre("quartz", 44.80);

  cout << "Montre m :" << endl;
  cout << m << endl;

  return 0;
}
```

```
Montre m :
Une montre composée de :
  * bracelet cuir coutant 54
  * fermoir acier coutant 12.5
  * boitier acier coutant 36.6
  * vitre quartz coutant 44.8
==> Prix total : 147.9
```

Le code complet à ce stade (134 lignes) peut être téléchargé sur le site du cours sous la rubrique « Compléments » :

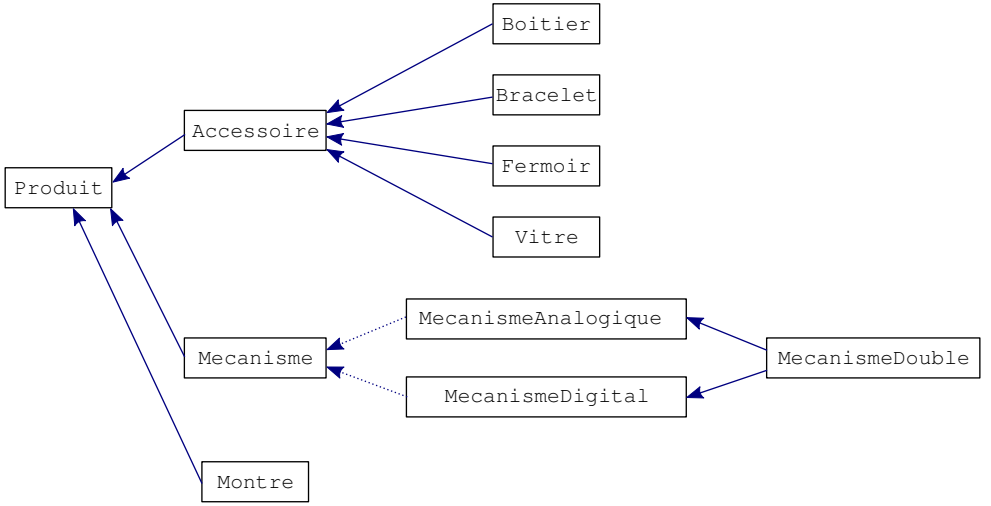
`final01.cc`

Rappel du problème

- Les montres ont un *mécanisme* de base [...]
- Les *mécanismes* [...] sont aussi des produits
- Il existe trois sortes de *mécanismes* : *analogiques*, *digitaux* et *doubles*.
- Pour les *mécanismes doubles*, on supposera ici qu'ils n'indiquent qu'une seule heure, mais se comportent sinon à la fois comme des *mécanismes analogiques* et comme des *mécanismes digitaux*

```
class Mecanisme : public Produit {
};

class MecanismeAnalogique : public Mecanisme
{};

class MecanismeDigital : public Mecanisme {
};

class MecanismeDouble : public Mecanisme {
};

// ========================================
class Montre : public Produit {
  // ...
private:
  unique_ptr<Mecanisme> coeur;
  // ...
};
```

Hiérarchie de classes



Révision de la hiérarchie

```
class Mecanisme : public Produit {
private:
  string heure;
};

class MecanismeAnalogique : virtual public Mecanisme {
private:
  int date;
};

class MecanismeDigital : virtual public Mecanisme {
private:
  string reveil;
};

class MecanismeDouble : public MecanismeAnalogique
                      , public MecanismeDigital
{
};
```

Conséquences

En raison de la classe *virtuelle* `Mecanisme`,
toutes les sous-classes **doivent** appeler le constructeur de cette classe !

Pour le moment, il n'y a qu'un constructeur par défaut,
mais fixons la construction des `Mecanisme` :

- initialisation de la valeur de base (`Produit`)
- initialisation de l'heure

```
class Mecanisme : public Produit {
public:
  Mecanisme(double valeur_de_base, string une_heure = "12:00")
  : Produit(valeur_de_base), heure(une_heure)
  {}

private:
  string heure;
};
```

Constructeurs des sous-classes

```
class MecanismeAnalogique : virtual public Mecanisme {
public:
  MecanismeAnalogique(double valeur_de_base, string une_heure, int une_date)
 : Mecanisme(valeur_de_base, une_heure), date(une_date)
  {}
private:
  int date;
};
// ...
class MecanismeDouble : public MecanismeAnalogique , public MecanismeDigital {
public:
  MecanismeDouble(double valeur_de_base, string une_heure, int une_date,
                  string heure_reveil)
  : Mecanisme(valeur_de_base, une_heure)
  , MecanismeAnalogique(valeur_de_base, une_heure, une_date)
  , MecanismeDigital(valeur_de_base, une_heure, heure_reveil)
  {}
};
```

Gestion de la valeur par défaut de la super-classe

```
class MecanismeAnalogique : virtual public Mecanisme {
public:
  MecanismeAnalogique(double valeur_de_base, string une_heure, int une_date)
 : Mecanisme(valeur_de_base, une_heure), date(une_date)
  {}

  MecanismeAnalogique(double valeur_de_base, int une_date)
  : Mecanisme(valeur_de_base), date(une_date)
  {}

private:
  int date;
};

// ...
```

Gestion de la valeur par défaut de la super-classe

```
class MecanismeDouble : public MecanismeAnalogique , public MecanismeDigital {
public:
  MecanismeDouble(double valeur_de_base, string une_heure, int une_date,
                  string heure_reveil)
  : Mecanisme(valeur_de_base, une_heure)
  , MecanismeAnalogique(valeur_de_base, une_heure, une_date)
  , MecanismeDigital(valeur_de_base, une_heure, heure_reveil)
  {}

  MecanismeDouble(double valeur_de_base, int une_date, string heure_reveil)
  : Mecanisme(valeur_de_base)
  , MecanismeAnalogique(valeur_de_base, une_date)
  , MecanismeDigital(valeur_de_base, heure_reveil)
  {}
};
```

Affichage des `Mecanisme`s

Supposons que l'on veuille :

- que tous les mécanismes s'affichent suivant le **même** schéma, **imposé** et non modifiable

  Par exemple :
  affichage du type de mécanisme, suivi d'un affichage du cadran (heure, date, heure de réveil, ...), suivi du prix
- mais que chaque partie de ce schéma soit adaptable
- offrir une version par défaut, utilisable dans les sous-classes, de l'affichage du cadran (par exemple affichage de l'heure)
- imposer la redéfinition de l'affichage du type de mécanisme

☞ Comment faire ?

Affichage des `Mecanisme`s : niveau général

```
class Mecanisme : public Produit {
public:
  //...
  // Tous les mécanismes DOIVENT s'afficher comme ceci
  virtual void afficher(ostream& sortie) const override final {
    sortie << "mécanisme "      ;     afficher_type(sortie);
    sortie << " (affichage : " ;   afficher_cadran(sortie);
    sortie << "), prix : "      ; Produit::afficher(sortie);
  }

protected: // On veut offrir la version par défaut aux sous-classes
  // Par défaut, on affiche juste l'heure.
  virtual void afficher_cadran(ostream& sortie) const {
    sortie << heure;
  }

private:
  virtual void afficher_type(ostream& sortie) const = 0;
};
```

Affichage des `Mecanisme`s : sous-classes

```
class MecanismeAnalogique: virtual public Mecanisme {
  // ...
protected:
  virtual void afficher_type(ostream& sortie) const override {
    sortie << "analogique";
  }

  virtual void afficher_cadran(ostream& sortie) const override {
    // On affiche l'heure (façon de base)...
    Mecanisme::afficher_cadran(sortie);
    // ...et en plus la date.
    sortie << ", date " << date;
  }
  // ...
};
```

Affichage des `Mecanisme`s : sous-classes

```
class MecanismeDouble: public MecanismeAnalogique, public MecanismeDigital {
  // ...
protected:
  virtual void afficher_type(ostream& sortie) const override {
    sortie << "double";
  }

  virtual void afficher_cadran(ostream& sortie) const override {
    // Par exemple...
    sortie << "sur les aiguilles : ";
    MecanismeAnalogique::afficher_cadran(sortie);
    sortie << ", sur l'écran : ";
    MecanismeDigital::afficher_cadran(sortie);
  }
};
```

Test : un exemple de `main()`

```
  // test de l'affichage des mécanismes
  MecanismeAnalogique v1(312.00, 20141212);
  MecanismeDigital    v2( 32.00, "11:45", "7:00");
  MecanismeDouble     v3(543.00, "8:20", 20140328, "6:30");
  cout << v1 << endl << v2 << endl << v3 << endl;

  // Test des montres
  Montre m(new MecanismeDouble(468.00, "9:15", 20140401, "7:00"));
  m += new Bracelet("cuir", 54.0);
  m += new Fermoir("acier", 12.5);
  m += new Boitier("acier", 36.60);
  m += new Vitre("quartz", 44.80);
  cout << endl << "Montre m :" << endl;
  cout << m << endl;
```

Le code complet à ce stade (263 lignes) peut être téléchargé sur le site du cours sous la rubrique « Compléments » :

`final02.cc`

Copie profonde

```
class Montre : public Produit {
  // ...
private:
  unique_ptr<Mecanisme> coeur;
  vector<unique_ptr<Accessoire>> accessoires;
};
```

Si l'on veut faire des copies de `Montre`s, on **doit** ici faire une **copie profonde** :
copie de chaque constituant (mécanisme, accessoires)

Il est alors également normal de redéfinir également l'opérateur d'affectation (`=`)

Constructeur de copie

```
Montre(const Montre& autre)
// Ne pas oublier d'appeler le constructeur DE COPIE de la super-classe
: Produit(autre),
  coeur(???)
  // ...??
{
  // ...??
}
```

Copie polymorphique

```
Montre(const Montre& autre)
// Ne pas oublier d'appeler le constructeur DE COPIE de la super-classe
: Produit(autre),
  coeur(autre.coeur->copie())
{
  for (auto const& p_acc : autre.accessoires) {
    accessoires.push_back(p_acc->copie());
  }
}
```

Copie polymorphique

```
  vector<unique_ptr<Accessoire>> accessoires;
  // ...
  accessoires.push_back(p_acc->copie());
```

```
class Accessoire : public Produit {
public:
  // ...
  // copie polymorphique d'Accessoire
  virtual unique_ptr<Accessoire> copie() const = 0;
  // ...
};
//-------------------------------------------------------
class Bracelet : public Accessoire {
public:
  // ...
  // copie polymorphique de Bracelet
  virtual unique_ptr<Accessoire> copie() const override
  { return unique_ptr<Accessoire>(new Bracelet(*this)); }
};
```

Opérateur d'affectation

```
class Montre : public Produit {
public:
  // ...
  Montre& operator=(Montre source) // Notez le passage par VALEUR
  {
    swap(coeur      , source.coeur      );
    swap(accessoires, source.accessoires);
    return *this;
  }
};
```

Test : un exemple de `main()`

```
// ... (le reste du main() comme avant)
// Nous faisons une copie de la montre m
Montre m2(m);
cout << "Montre m2 :" << endl;
cout << m2 << endl;

// Et testons l'opérateur d'affectation
Montre m3(new MecanismeAnalogique(87.00, 20140415));
cout << "Montre m3 (1) :" << endl;
cout << m3 << endl;

m3 = m2; m2.mettre_a_l_heure("10:10");
cout << "Montre m3 (2) :" << endl;
cout << m3 << endl;
// ...
```

Le code complet à ce stade (367 lignes) peut être téléchargé sur le site du cours sous la rubrique « Compléments » :

`montres.cc`