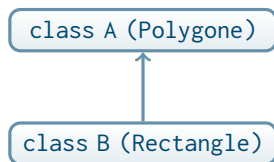


Cours C++

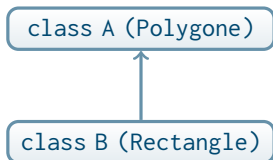
Héritage

Introduction

- L'héritage est une relation hiérarchique entre classes. C'est un mécanisme permettant de construire une classe à partir d'une autre classe existante.

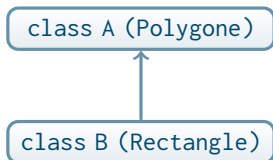


Une classe B héritant d'une classe A



- ▶ La classe B hérite de tous les membres (attributs et méthodes) de la classe A
- ▶ Les méthodes (et fonctions amies) de B ont accès aux membres publics et protégés de A
- ▶ Ainsi, la classe B est basée sur la classe A mais peut également ajouter de nouveaux membres ainsi que de nouvelles méthodes propres à sa fonction :
 - ▶ la classe A que l'on appellera classe mère ou de base est une généralisation/abstraction de B,
 - ▶ la classe B que l'on dit dérivée ou classe fille de A est donc une spécialisation de A.

Une classe B héritant d'une classe A



- ▶ La classe B hérite de tous les membres (attributs et méthodes) de la classe A
- ▶ Les méthodes (et fonctions amies) de B ont accès aux membres publics et protégés de A
- ▶ Ainsi, la classe B est basée sur la classe A mais peut également ajouter de nouveaux membres ainsi que de nouvelles méthodes propres à sa fonction :
 - ▶ la classe A que l'on appellera **classe mère ou de base** est une **généralisation/abstraction** de B,
 - ▶ la classe B que l'on dit **dérivée ou classe fille** de A est donc une **spécialisation** de A.

Définition : héritage simple/multiple, composition

On distingue les deux liens logiques que sont l'héritage et la composition:

Composition relation de type **a un (verbe avoir)**; la classe Polygone a un ensemble d'objets de type point,

Héritage[†] relation de type **est un (verbe être)**; un Rectangle est un Polygone, une Voiture est un Véhicule.

Héritage simple : une classe n'hérite que d'une seule classe,

Héritage multiple : une classe hérite de plus d'une classe (plus délicat à mettre en oeuvre)

[†] On devrait parler de **sous-typage**; l'héritage n'étant qu'une façon de faire du sous-typage.

Notions d'héritage simple

```
class Polygone {  
public:  
    Polygone (const unsigned int nbr_segment);  
protected:  
    unsigned int m_nombre_segment;  
};  
  
class Rectangle : public Polygone {  
public:  
    Rectangle (const double longueur, const double largeur);  
private:  
    double m_longueur;  
    double m_largeur;  
};
```

- ▶ La relation d'héritage est matérialisée par l'usage de la directive `public Polygone`
- ▶ Le mot-clé `protected` autorise la classe dérivée (ici `Rectangle`) à accéder aux membres de la classe de base (ici `Polygone`)

Notions d'héritage simple

```
class Polygone {  
public:  
    Polygone (const unsigned int nbr_segment);  
protected:  
    unsigned int m_nombre_segment;  
};  
  
class Rectangle : public Polygone {  
public:  
    Rectangle (const double longueur, const double largeur);  
private:  
    double m_longueur;  
    double m_largeur;  
};
```

- ▶ La relation d'héritage est matérialisée par l'usage de la directive `public Polygone`
- ▶ Le mot-clé `protected` autorise la classe dérivée (ici `Rectangle`) à accéder aux membres de la classe de base (ici `Polygone`)

Notions d'héritage simple

```
class Polygone {  
public:  
    Polygone (const unsigned int nbr_segment);  
protected:  
    unsigned int m_nombre_segment;  
};  
  
class Rectangle : public Polygone {  
public:  
    Rectangle (const double longueur, const double largeur);  
private:  
    double m_longueur;  
    double m_largeur;  
};
```

- ▶ La relation d'héritage est matérialisée par l'usage de la directive `public Polygone`
- ▶ Le mot-clé `protected` autorise la classe dérivée (ici Rectangle) à accéder aux membres de la classe de base (ici Polygone)

Héritage simple

```
class Polygone {  
public:  
    Polygone (const unsigned int nbr_segment);  
protected:  
    unsigned int m_nombre_segment;  
};
```

Encapsulation des données
pour les classes dérivées



```
class Rectangle : public Polygone {  
public:  
    Rectangle (const double longueur, const double largeur);  
private:  
    double m_longueur;  
    double m_largeur;  
};
```

Relation d'héritage



Héritage simple

```
class Polygone {  
public:  
    Polygone (const unsigned int nbr_segment);  
protected:  
    unsigned int m_nombre_segment;  
};
```

Encapsulation des données
pour les classes dérivées



```
class Rectangle : public Polygone {  
public:  
    Rectangle (const double longueur, const double largeur);  
private:  
    double m_longueur;  
    double m_largeur;  
};
```

Relation d'héritage



Héritage simple

```
class Polygone {  
public:  
    Polygone (const unsigned int nbr_segment);  
protected:  
    unsigned int m_nombre_segment;  
};
```

← Encapsulation des données
pour les classes dérivées

```
class Rectangle : public Polygone {  
public:  
    Rectangle (const double longueur, const double largeur);  
private:  
    double m_longueur;  
    double m_largeur;  
};
```

← Relation d'héritage

Statut des membres et méthodes de classe

- `private` les membres ne sont accessibles qu'aux méthodes et aux fonctions amies de la classe
- `protected` les membres sont accessibles aux méthodes de la classe de base ainsi qu'aux classes dérivées. Ils demeurent toutefois inaccessibles à l'utilisateur de la classe contrairement au statut `public`
- `public` les membres sont accessibles non seulement aux méthodes mais également à l'utilisateur de la classe

Appels des constructeurs et destructeurs

```
class Polygone {  
public:  
    Polygone (const unsigned int nbr_segment);  
protected:  
    unsigned int m_nombre_segment;  
};  
  
class Rectangle : public Polygone {  
public:  
    Rectangle (const double longueur, const double largeur);  
private:  
    double m_longueur;  
    double m_largeur;  
};
```

Appels des constructeurs et destructeurs

```
Polygone::Polygone(const unsigned int nbr_segment) :  
    m_nombre_segment(nbr_segment)  
{}
```

```
Rectangle::Rectangle(const double longueur, const double largeur) :  
    Polygone(4), ←  
    m_longueur(longueur), m_largeur(largeur)  
{}
```

Appel au constructeur
de la classe de base

Appels des constructeurs et destructeurs

```
Polygone::Polygone(const unsigned int nbr_segment) :  
    m_nombre_segment(nbr_segment)  
{}
```

```
Rectangle::Rectangle(const double longueur, const double largeur) :  
    Polygone(4),  
    m_longueur(longueur), m_largeur(largeur)  
{}
```

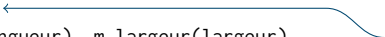
Appel au constructeur
de la classe de base

Appels des constructeurs et destructeurs

```
Polygone::Polygone(const unsigned int nbr_segment) :  
    m_nombre_segment(nbr_segment)  
{}
```

```
Rectangle::Rectangle(const double longueur, const double largeur) :  
    Polygone(4),  
    m_longueur(longueur), m_largeur(largeur)  
{}
```

Appel au constructeur
de la classe de base



Appels des constructeurs et destructeurs

- ▶ À la construction d'une classe fille, le constructeur de la classe mère est appelé **avant** toutes autres opérations
- ▶ Lors de la destruction d'une classe fille, le destructeur de la classe de base est appelé automatiquement **après** le destructeur de la classe fille

Redéfinition de méthodes

```
class Polygone {  
public:  
    void affiche() const {  
        cout << "Polygone !" << endl;  
    }  
    ...  
};
```

```
class Rectangle : public Polygone {  
public:  
    void affiche() const {  
        cout << "Rectangle !" << endl;  
    }  
    ...  
};
```

La classe Rectangle **redéfinit** (override) la méthode `affiche()` pour adapter son comportement à la finalité de la classe, au lieu d'hériter celle de la classe mère. C'est tout l'intérêt de l'héritage !

Liaison statique

```
class Polygone {  
public:  
    void affiche() const {  
        cout << "Polygone !" << endl;  
    }  
    ...  
};
```

```
class Rectangle : public Polygone {  
public:  
    void affiche() const {  
        cout << "Rectangle !" << endl;  
    }  
    ...  
};
```

```
Polygone my_polygone;  
Rectangle my_rectangle;  
my_polygone.affiche(); // -> Polygone !  
my_rectangle.affiche(); // -> Rectangle !
```

Le compilateur connaît, lors de la compilation, le type d'objet instancié → comportement souhaité

Liaison statique

```
class Polygone {  
public:  
    void affiche() const {  
        cout << "Polygone !" << endl;  
    }  
    ...  
};
```

```
class Rectangle : public Polygone {  
public:  
    void affiche() const {  
        cout << "Rectangle !" << endl;  
    }  
    ...  
};
```

```
Polygone my_polygone;  
Rectangle my_rectangle;  
my_polygone.affiche(); // -> Polygone !  
my_rectangle.affiche(); // -> Rectangle !
```

Le compilateur connaît, lors de la compilation, le type d'objet instancié → comportement souhaité

Liaison statique

```
class Polygone {  
public:  
    void affiche() const {  
        cout << "Polygone !" << endl;  
    }  
    ...  
};
```

```
class Rectangle : public Polygone {  
public:  
    void affiche() const {  
        cout << "Rectangle !" << endl;  
    }  
    ...  
};
```

```
Polygone my_polygone;  
Rectangle my_rectangle;  
my_polygone.affiche(); // -> Polygone !  
my_rectangle.Polygone::affiche(); // -> Polygone !
```

Liaison statique

```
class Polygone {  
public:  
    void affiche() const {  
        cout << "Polygone !" << endl;  
    }  
    ...  
};
```

```
class Rectangle : public Polygone {  
public:  
    void affiche() const {  
        cout << "Rectangle !" << endl;  
    }  
    ...  
};
```

```
Polygone* ptr_polygone1 = new Polygone;  
Polygone* ptr_polygone2 = new Rectangle;  
ptr_polygone1->affiche(); // -> Polygone !  
ptr_polygone2->affiche(); // -> Polygone !
```

Comme on manipule des `Polygone*`, le compilateur n'a aucun moyen de savoir le type d'objet réellement instancié → **liaison statique** aux méthodes de `Polygone` → comportement non souhaité

Liaison statique

```
class Polygone {  
public:  
    void affiche() const {  
        cout << "Polygone !" << endl;  
    }  
    ...  
};
```

```
class Rectangle : public Polygone {  
public:  
    void affiche() const {  
        cout << "Rectangle !" << endl;  
    }  
    ...  
};
```

```
Polygone* ptr_polygone1 = new Polygone;  
Polygone* ptr_polygone2 = new Rectangle;  
ptr_polygone1->affiche(); // -> Polygone !  
ptr_polygone2->affiche(); // -> Polygone !
```

Comme on manipule des `Polygone*`, le compilateur n'a aucun moyen de savoir le type d'objet réellement instancié → **liaison statique** aux méthodes de `Polygone` → comportement non souhaité

Liaison dynamique & Méthodes virtuelles

```
class Polygone {  
public:  
    virtual void affiche() const {  
        cout << "Polygone !" << endl;  
    }  
    ...  
};
```

```
class Rectangle : public Polygone {  
public:  
    virtual void affiche() const {  
        cout << "Rectangle !" << endl;  
    }  
    ...  
};
```

```
Polygone* ptr_polygone1 = new Polygone;  
Polygone* ptr_polygone2 = new Rectangle;  
ptr_polygone1->affiche(); // -> Polygone !  
ptr_polygone2->affiche(); // -> Rectangle !
```

En déclarant les méthodes **virtuelles** (via le mot-clé `virtual`), on indique au compilateur que le choix de la méthode ne s'effectuera qu'à l'*exécution du code* : **liaison dynamique** / dynamic dispatch

Liaison dynamique & Méthodes virtuelles

→ Oublier le type à la compilation, le déduire à l'exécution.

Comment le compilateur fait-il ?

- ▶ Dans chaque instance, le type et les adresse des méthodes sont stockée en mémoire dans une *vtable*
- ▶ Attention, coûte plus de temps et de mémoire qu'une liaison statique. La majorité des méthodes de la bibliothèque standard ne sont *pas* virtuelles (exception : `iostream`)

Polymorphisme

Pourquoi vouloir oublier le type de l'objet à la compilation ?

- ▶ Manipuler des objets de la même façon, avec les méthodes de la classe de base, quelque soit le type instancié → code **générique**[†]
- ▶ Stocker un ensemble hétérogène d'objets[‡]

Le fait que l'appel `ptr_polygone->affiche()` dépende du type instancié : **polymorphisme**[§].

[†] L'autre façon, préférable lorsque l'on manipule un seul objet, est la métaprogrammation (templates)

[‡] Note : il est nécessaire d'utiliser des pointeurs (possiblement automatiques) ou références en C++ car une variable du type de base ne peut stocker quoi que ce soit d'autre qu'un objet du type de base !

[§] Dans les langages dynamiquement typés comme le Python, on appelle ça le **duck typing**.

Polymorphisme

Pourquoi vouloir oublier le type de l'objet à la compilation ?

- ▶ Manipuler des objets de la même façon, avec les méthodes de la classe de base, quelque soit le type instancié → code **générique**[†]
- ▶ Stocker un ensemble hétérogène d'objets[‡]

Le fait que l'appel `ptr_polygone->affiche()` dépende du type instancié : **polymorphisme**[§].

[†] L'autre façon, préférable lorsque l'on manipule un seul objet, est la métaprogrammation (templates)

[‡] Note : il est nécessaire d'utiliser des pointeurs (possiblement automatiques) ou références en C++ car une variable du type de base ne peut stocker quoi que ce soit d'autre qu'un objet du type de base !

[§] Dans les langages dynamiquement typés comme le Python, on appelle ça le **duck typing**.

Polymorphisme

Pourquoi vouloir oublier le type de l'objet à la compilation ?

- ▶ Manipuler des objets de la même façon, avec les méthodes de la classe de base, quelque soit le type instancié → code **générique**[†]
- ▶ Stocker un ensemble hétérogène d'objets[‡]

Le fait que l'appel `ptr_polygone->affiche()` dépende du type instancié : **polymorphisme**[§].

[†] L'autre façon, préférable lorsque l'on manipule un seul objet, est la métaprogrammation (templates)

[‡] Note : il est nécessaire d'utiliser des pointeurs (possiblement automatiques) ou références en C++ car une variable du type de base ne peut stocker quoi que ce soit d'autre qu'un objet du type de base !

[§] Dans les langages dynamiquement typés comme le Python, on appelle ça le **duck typing**.

Syntaxe

Le mot clé `virtual` est utilisé seulement dans la **déclaration** des méthodes (.h), on ne le mentionne pas dans la **définition** des méthodes (.cpp). Si une méthode est déclarée virtuelle dans une classe de base, le mot clé `virtual` est facultatif lors de la re-déclaration dans une classe fille.

```
class Polygone {  
public:  
    virtual void affiche() const;  
    ...  
};
```

```
void Polygone::affiche() const {  
    cout << m_nombre_segment  
        << " segments" << endl;  
}
```

```
class Rectangle : public Polygone {  
public:  
    virtual void affiche() const;  
    ...  
};
```

facultatif

appel méthode parent

```
void Rectangle::affiche() const {  
    Polygone::affiche();  
    cout << "Rectangle de " << m_longueur  
        << " x " << m_largeur << endl;  
}
```

Syntaxe

Le mot clé `virtual` est utilisé seulement dans la **déclaration** des méthodes (.h), on ne le mentionne pas dans la **définition** des méthodes (.cpp). Si une méthode est déclarée virtuelle dans une classe de base, le mot clé `virtual` est facultatif lors de la re-déclaration dans une classe fille.

```
class Polygone {  
public:  
    virtual void affiche() const;  
    ...  
};  
  
void Polygone::affiche() const {  
    cout << m_nombre_segment  
        << " segments" << endl;  
}
```

```
class Rectangle : public Polygone {  
public:  
    virtual void affiche() const;  
    ...  
};  
  
void Rectangle::affiche() const {  
    Polygone::affiche();  
    cout << "Rectangle de " << m_longueur  
        << " x " << m_largeur << endl;  
}
```

... ← facultatif

appel méthode parent

Syntaxe

Le mot clé `virtual` est utilisé seulement dans la **déclaration** des méthodes (.h), on ne le mentionne pas dans la **définition** des méthodes (.cpp). Si une méthode est déclarée virtuelle dans une classe de base, le mot clé `virtual` est facultatif lors de la re-déclaration dans une classe fille.

```
class Polygone {  
public:  
    virtual void affiche() const;  
    ...  
};  
  
void Polygone::affiche() const {  
    cout << m_nombre_segment  
        << " segments" << endl;  
}
```

```
class Rectangle : public Polygone {  
public:  
    virtual void affiche() const;  
    ...  
};  
  
void Rectangle::affiche() const {  
    Polygone::affiche();  
    cout << "Rectangle de " << m_longueur  
        << " x " << m_largeur << endl;  
}
```

Diagram annotations:

- An arrow points from the text "facultatif" to the `virtual` keyword in the `Rectangle` class declaration.
- An arrow points from the text "appel méthode parent" to the `Polygone::affiche()` call inside the `Rectangle::affiche()` definition.

Liaisons dynamiques et constructeurs/destructeurs

- ▶ Un constructeur ne peut pas être virtuel (par définition, un constructeur n'est appelé que pour construire un objet de type bien défini).
- ▶ Un destructeur peut être virtuel et doit l'être si la classe déclare au moins une méthode virtuelle. Par ex, on considère le programme précédent où la méthode `affiche()` est virtuelle et les destructeurs de `polygone` et `rectangle` sont non-virtuels

```
Polygone* ptr_polygone = new Rectangle (...);  
ptr_polygone->affiche(); // affiche "Rectangle !"  
                        // car liaison dynamique  
delete ptr_polygone;    // on veut détruire le rectangle mais c'est  
                        // le destructeur de polygone qui est appelé
```

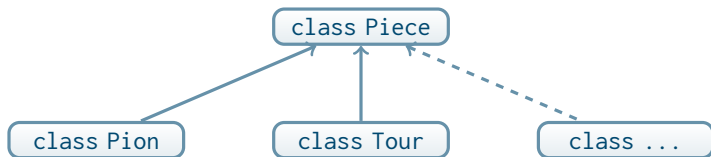
Il faut une liaison dynamique pour la destruction →
destructeur virtuel

Classes abstraites

- ▶ Parfois, la classe de base n'est là que pour définir une **interface** commune, et ça n'a pas de sens d'implémenter la méthode. Par exemple dessiner un Polygone n'aurait pas de sens.
- ▶ C++ permet la déclaration de **méthodes virtuelles pures** c'est-à-dire des méthodes dont **la définition est nulle**. Syntaxe de déclaration :
`virtual void affiche() const = 0;`
- ▶ Si il y a au moins une méthode virtuelle pure, la classe est alors dite **abstraite**. Elle n'est **pas instanciable**.

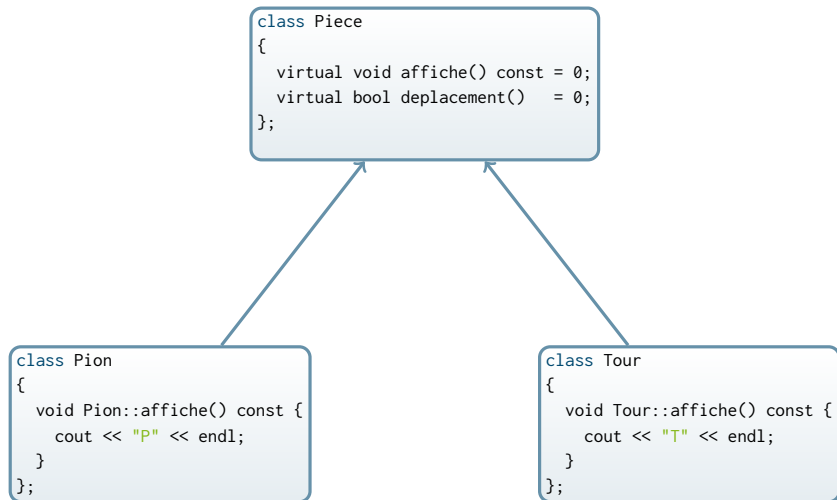
Classes abstraites

Exemple du jeu d'échec :



- ▶ La classe `Piece` est par nature **une classe abstraite** : elle déclare des méthodes virtuelles pures : `affiche()`, `deplacement()`
- ▶ La définition n'intervient que dans les classes dérivées qui spécialisent les méthodes en fonction de leur besoin

Classes abstraites



On résume tout

```
#include "Pion.h"
#include "Tour.h"

int main()
{
    std::vector< Piece* > pieces; // conteneur d'objets hétérogènes

    // instantiation d'objets non-abstraits
    pieces.push_back( new Pion );
    pieces.push_back( new Tour );

    for (size_t i = 0; i < pieces.size(); ++i)
        pieces[i]->affiche(); // polymorphisme à l'œuvre
}
```

Étant donné le canevas / l'interface fournie par la déclaration de la classe Piece, libre à chaque classe fille d'implémenter, de façon indépendante, ces fonctionnalités.

On résume tout

```
#include "Pion.h"
#include "Tour.h"

int main()
{
    std::vector< Piece* > pieces; // conteneur d'objets hétérogènes

    // instantiation d'objets non-abstraits
    pieces.push_back( new Pion );
    pieces.push_back( new Tour );

    for (size_t i = 0; i < pieces.size(); ++i)
        pieces[i]->affiche(); // polymorphisme à l'œuvre
}
```

Étant donné le canevas / l'interface fournie par la déclaration de la classe Piece, libre à chaque classe fille d'implémenter, de façon indépendante, ces fonctionnalités.

Utilisations de l'héritage

- ▶ Extension (ajout de fonctionnalités...) d'une classe existante, même tierce.
- ▶ Structuration : Découpage d'un programme en sous-programmes, en composants non-indépendants
→ découplage de concepts distincts, évolutivité, modularité
→ TD 4
- ▶ Sous-typage et création d'une famille d'objets (relation de type **est un**). Utile pour des systèmes de simulation, pour la représentations d'objets physiques ou logiciels (interfaces utilisateurs...). Polymorphisme. → TD 5
- ▶ Spécifications d'interfaces, d'APIs (classes virtuelles pures)

Et dans d'autres langages ?

L'héritage diffère notablement entre les langages orientés objets.

En Python :

```
class Polygone:

    def affiche(self):
        # ...

class Rectangle (Polygone):

    def __init__(self):
        super.__init__()

    def affiche(self):
        Polygon.affiche()
        # ...
```