

# ***Programmation orientée objet***

Polymorphisme

# Motivation

---

- Supposons que l'on déclare trois horloges:

```
Clock c1(true);
```

```
TravelClock c2(true, "Paris", 6);
```

```
TravelClock c3(true, "Vancouver", -3);
```

Cette horloge devrait nous donner l'heure locale + 6 heures.

... heure locale - 3 heures.

- On aimerait mettre tous ces objets dans un même vecteur
- *Problème:* ils ne sont pas de la même classe
- Pourtant, tous ces objets sont ou dérivent de la classe Clock!

## Motivation (suite)

---

- Voyons ce qui se passe si on met ces trois objets dans un même vecteur:

```
vector< Clock > clocks;  
clocks.push_back(c1);  
clocks.push_back(c2);  
clocks.push_back(c3);  
...  
for (size_t i = 0; i < clocks.size(); ++i) {  
    cout << clocks[i].get_hours() << ":" <<  
        << clocks[i].get_minutes() << endl;  
}
```

## Motivation (suite)

---

- Voici ce qui sera affiché (en supposant que l'heure locale est 14h30):

14 : 30

14 : 30

14 : 30

- Ce n'est sûrement pas le résultat espéré
- Que s'est-il passé?

# Conversion d'objet

---

- Le vecteur s'attend à recevoir un objet de la classe Clock
- On lui passe un objet de la classe dérivée TravelClock, qui contient plus d'attributs
- Cet objet sera donc converti en un objet de la classe Clock, en « oubliant » tout ce qui fait de lui un objet de la classe TravelClock
- Ainsi, lorsqu'on appelle la méthode `get_hours()`, c'est celle de la classe Clock qui est appelée

# Conversion d'objet

- Le même phénomène peut se produire lorsqu'on passe un objet à une fonction:

```
void afficher(Clock clock)
{
    cout << clock.get_hours() << " : "
         << clock.get_minutes();
}

int main()
{
    Clock c;
    TravelClock t;
    afficher(c);
    afficher(t);
    ...
}
```

Remarquez que le constructeur de copie qui sera appelé est celui de la classe **Clock**. Tout objet reçu sera donc converti en objet de la classe **Clock**.

On peut passer en paramètre un objet d'une classe dérivée.

...mais on appellera toujours les méthodes de la classe **Clock**.

# Une autre approche

---

- Voyons ce qu'on peut faire en utilisant un vecteur de pointeurs:

```
Clock* c1 = new Clock(true);  
TravelClock* c2 = new TravelClock(true, "Paris", 6);  
TravelClock* c3 = new TravelClock(true, "Vancouver", -3);  
vector< Clock* > clocks;  
clocks.push_back(c1);  
clocks.push_back(c2);  
clocks.push_back(c3);
```

- Puisque le vecteur ne contient que des pointeurs, aucune conversion d'objet n'est réalisée ici

## Une autre approche (suite)

---

- Par contre, le résultat est toujours le même:

```
for (size_t i = 0; i < clocks.size(); ++i) {  
    cout << clocks[i]->get_hours() << ":"  
        << clocks[i]->get_minutes() << endl;  
}
```

14:30

14:30

14:30

- Comme le type du pointeur est Clock\*, c'est encore une fois les méthodes de cette classe qui sont appelées



## Une autre approche (suite)

---

- En fait, nous voudrions faire comprendre ceci au compilateur:

*Je déclare un pointeur de type **Clock\***, mais il se pourrait que tu reçoives un pointeur sur un objet d'une classe dérivée. J'aimerais que dans ce cas tu appelles la méthode de cette classe dérivée plutôt que celle de la classe **Clock**.*

*ou encore*

*Je déclare une référence de type **Clock**, mais il se pourrait que tu reçoives un objet d'une classe dérivée. J'aimerais que dans ce cas tu appelles la méthode de cette classe dérivée plutôt que celle de la classe **Clock**.*

# Fonction virtuelle

---

- Oui, on peut le faire en C++
- Il suffit de déclarer une méthode virtuelle:

```
class Clock
{
    public:
        ...
        virtual int get_hours() const;
        ...
}
```

## Fonction virtuelle (suite)

---

- Lorsqu'une méthode est déclarée virtuelle, absolument rien ne change si l'objet est copié
- Par contre, si on reçoit un pointeur ou que l'on fait un passage par référence, c'est la méthode de la **classe réelle** de l'objet qui sera appelée

## Fonction virtuelle (suite)

---

- Supposons maintenant que nous avons la déclaration suivante:  
`vector< Clock* > clocks;`
- Supposons aussi que la méthode `get_hours()` est virtuelle
- On ne sait pas a priori quelle méthode sera appelée dans l'instruction suivante:  
`clocks[i]->get_hours()`
- Cela dépend de la classe réelle de l'objet pointé par `clocks[i]`

## Fonction virtuelle (suite)

---

- Considérons le programme suivant (n'oublions pas que `get_hours()` a été déclarée virtuelle):

```
vector< Clock* > clocks;  
clocks.push_back(new Clock(true));  
clocks.push_back(new TravelClock(true, "Paris", 6));  
clocks.push_back(new TravelClock(true, "Vancouver", -3));  
for (size_t i = 0; i < clocks.size(); ++i) {  
    cout << clocks[i]->get_hours() << ":"  
        << clocks[i]->get_minutes() << endl;  
}
```

- Nous aurons finalement le résultat espéré:

```
14:30  
20:30  
11:30
```

## Fonction virtuelle (suite)

---

- *Attention:* si une méthode est déclarée virtuelle dans une classe, elle le sera automatiquement dans toutes les classes qui en dérivent
- Pour éviter toute confusion, on redéclare la méthode virtuelle dans les classes dérivées
- L'avantage de cela est qu'on n'aura pas besoin d'aller consulter la classe de base pour savoir si une méthode est virtuelle

# Polymorphisme et méthode héritée

---

- Supposons une fonction virtuelle **f1 ()** définie dans une classe de base **A**
- Supposons maintenant une classe **B** dérivée de **A** qui ne redéfinit pas la fonction **f1 ()**
- Selon le principe de l'héritage, la fonction **f1 ()** dans **B** est héritée de la classe **A**
- Alors, comment se comporte le polymorphisme dans ce cas?

# Polymorphisme et méthode héritée (suite)

---

```
class A
{
    public:
        ...
        virtual void f1();
        ...
};

class B : public A
{
    public:
        ... // f1() non définie ici
};
```



# Polymorphisme et méthode héritée (suite)

- Soient les instructions suivantes:

```
int main()
{
    vector< A* > v;
    v.push_back(new A());
    v.push_back(new B());
    ...

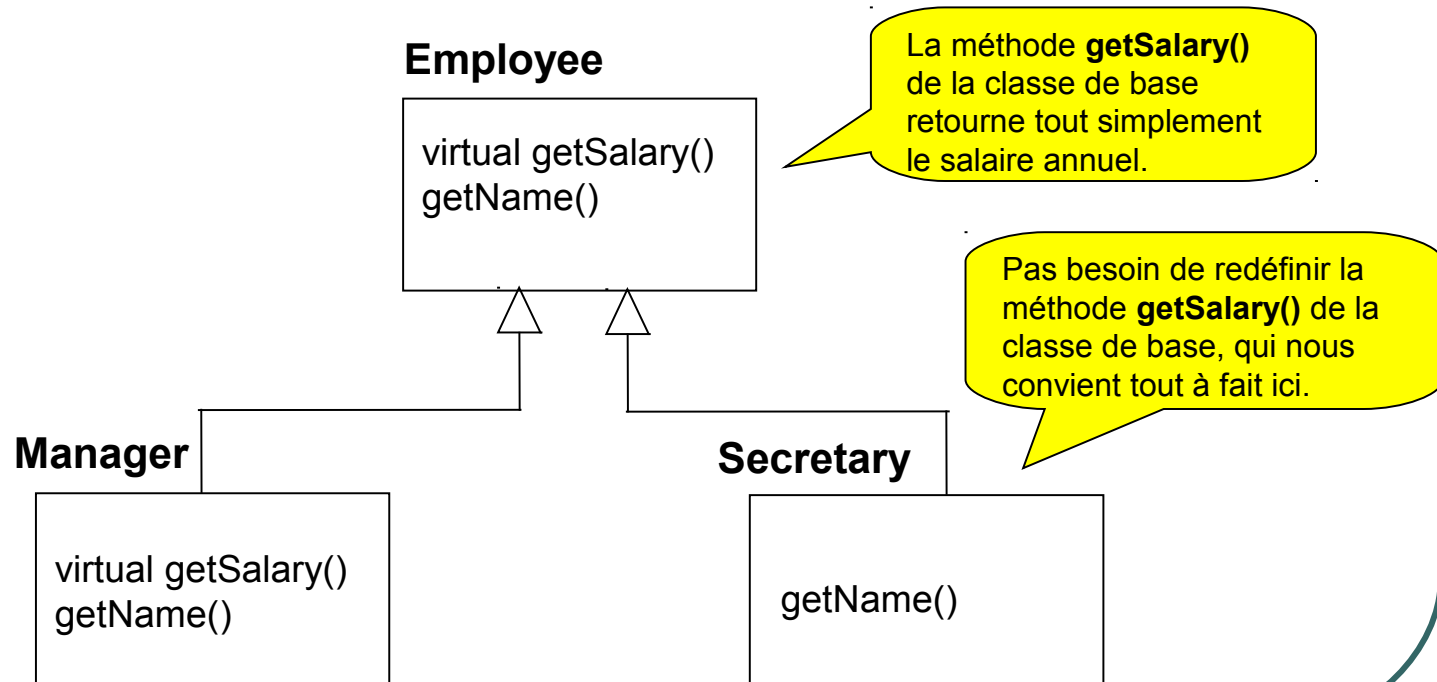
    v[1]->f1();
}
```

On ajoute deux pointeurs dans le vecteur, dont un qui pointe vers un objet de la classe dérivée.

f1() étant virtuelle, on doit appeler la méthode de la classe **B**. Mais comme la méthode n'est pas définie dans **B**, on appelle la méthode f1() de la classe **A** dont elle hérite.

# Polymorphisme et méthode héritée (suite)

- Ainsi, même si une fonction est virtuelle, on est libre de la redéfinir ou non dans une classe dérivée



# Polymorphisme et méthode héritée (exemple)

---

```
class Employee
{
public:
    Employee(string name = "unknown",
              double salary = 0);
    virtual double getSalary() const;
    string getName() const;
    /* ... */
private:
    string name_;
    double salary_;
};

Employee::Employee(string name,
                   double salary)
    : name_(name), salary_(salary)
{
}

string Employee::getName() const
{
    return name_;
}

double Employee::getSalary() const
{
    return salary_;
}
```

# Polymorphisme et méthode héritée (exemple)

```
class Manager : public Employee
{
public:
    Manager(string name = "unknown",
            double salary = 0,
            double bonus = 0);
    virtual double getSalary() const;
    string getName() const;
    /* ... */
private:
    double bonus_;
    /* ... */
};
```

```
Manager::Manager(string name,
                  double salary,
                  double bonus)
    : Employee(name, salary),
      bonus_(bonus)
{
}
string Manager::getName() const
{
    return Employee::getName() +
           " (Manager)";
}
double Manager::getSalary() const
{
    return Employee::getSalary() *
           (1 + bonus_ / 100);
}
```

# Polymorphisme et méthode héritée (exemple)

---

```
class Secretary : public Employee
{
public:
    Secretary(string name = "unknown", {
        double salary = 0);
    string getName() const;
    /* ... */
};

Secretary::Secretary(string name,
                    double salary)
    : Employee(name, salary)
{
    string Secretary::getName() const
    {
        return Employee::getName() +
            " (Secretary)";
    }
}
```

# Polymorphisme et méthode héritée (exemple)

```
bool bien_paye(const Employee& employee)
{
    return (employee.getSalary() > 75000.0);
}

int main()
{
    vector<Manager> v;
    Manager m1;

    /* ... */
    m1 = Manager("Jenny", 12000.0, 10);

    if (bien_paye(m1)) {
        v.push_back(m1);
    }
    /* ... */
}
```

Passage par référence: on exécutera donc méthode **getSalary()** de la classe réelle, puisque cette méthode est déclarée virtuelle.

Lorsqu'on traitera un objet de la classe **Gerant**, c'est la méthode de la classe **Gerant** qui sera appelée.

# Polymorphisme - terminologie

- Soient les instructions suivantes:

```
int main()  
{  
    vector<Employee*> v;  
    v.push_back(new Secretary("Mark", 16000.0));  
    v.push_back(new Manager("Jenny", 12000.0, 10));  
    /* ... */  
  
    v[1]->getSalary();  
}
```

Il s'agit ici du type **statique** du pointeur.

Le type réel de l'objet pointé par ce pointeur n'est pas de la classe **Employee**, mais plutôt de la classe **Manager**, qui est le type **dynamique**.

# Polymorphisme - terminologie

- Le type statique est évidemment déterminé lors de la compilation
- Le type dynamique, lui, est déterminé seulement lors de l'exécution; c'est ce qu'on appelle la **liaison dynamique**
- Par exemple, dans la boucle suivante, on ne peut pas savoir avant l'exécution le type réel de l'objet pointé par **v[i]**:

```
vector<Employee*> v;  
/* ... */  
for (size_t i = 0; i < v.size(); ++i) {  
    cout << v[i]->getSalary() << endl;  
}
```

Liaison dynamique



# Polymorphisme - terminologie

- *Attention:* ce que nous venons de dire ne vaut que pour les méthodes virtuelles
- Nous savons, par exemple, que la méthode **getName()** de la classe **Employee** n'est pas virtuelle
- Dans la boucle suivante, l'objet pointé par **v[i]** sera toujours considéré comme un **Employee** (c'est donc toujours la méthode de cette classe qui sera exécutée):

```
vector<Employee*> v;  
/* ... */  
for (size_t i = 0; i < v.size(); ++i) {  
    cout << v[i]->getName() << endl;  
}
```

Liaison statique

## En résumé

---

- Par défaut, la liaison est statique, c'est-à-dire qu'on appelle toujours la méthode de la classe indiquée dans la déclaration de l'objet (cette méthode pouvant bien sûr être héritée)
- Dans le cas d'un pointeur ou d'une référence, cela signifie qu'on appelle la méthode de la classe qui correspond au type du pointeur ou de la référence
- Le seul cas où on effectue une liaison dynamique, c'est lorsqu'on a un pointeur (ou une référence) sur un objet d'une classe dérivée, alors que ce pointeur (ou référence) a été déclaré du type de la classe de base
- Dans ce cas, **si la méthode a été déclarée virtuelle**, on appellera la méthode de la classe dérivée et non pas celle du pointeur ou de la référence

# Méthode virtuelle appelée dans une autre méthode de la classe

- On sait qu'une méthode peut appeler une autre méthode de la même classe
- Que se passe-t-il si cette autre méthode est virtuelle?
- Supposons que la classe **Employee** a une méthode **print()** définie de la manière suivante:

```
void Employee::print(ostream& out) const
{
    ...
    out << getName() << endl;
    out << "Salary: " << getSalary() << endl;
}
```

Méthode non virtuelle.

Méthode virtuelle.

# Méthode virtuelle appelée dans une autre méthode de la classe

---

- Quel sera le résultat de l'affichage dans le programme suivant:

```
void Employee::print(ostream& out) const
{
    ...
    out << getName() << endl;
    out << "Salary: " << getSalary() << endl;
}
int main() {
    Employee e1("John", 15000);
    e1.print(cout);

    return 0;
}
```

# Méthode virtuelle appelée dans une autre méthode de la classe

- Pour répondre, considérez le code équivalent suivant:

```
void Employee::print(ostream& out) const
{
    ...
    out << this->getName() << endl;
    out << "Salary: " << this->getSalary() << endl;
}
int main() {
    Employee e1("John", 15000);
    e1.print(cout);

    return 0;
}
```

Le type de **this** est **Employee\***, c'est donc la méthode de cette classe qui sera appelée, puisque **getName()** n'est pas virtuelle.

Ici, c'est la méthode de la classe dérivée qui sera appelée, puisque **getSalary()** est virtuelle.

# Méthode virtuelle appelée dans une autre méthode de la classe

- Quel sera le résultat de l'affichage dans le programme suivant:

```
void Employee::print(ostream& out) const
{
    ...
    out << getName() << endl;
    out << "Salary: " << getSalary() << endl;
}
int main() {
    Employee e1("John", 15000);
    e1.print(cout);

    return 0;
}
```

C'est la méthode de la classe **Employee** qui est appelée.

C'est la méthode de la classe **Employee** qui est appelée.

```
John
Salary: 15000
```

# Méthode virtuelle appelée dans une autre méthode de la classe

---

- Quel sera le résultat de l'affichage dans le programme suivant:

```
void Employee::print(ostream& out) const
{
    ...
    out << getName() << endl;
    out << "Salary: " << getSalary() << endl;
}
int main() {
    Manager e1("John", 15000);
    e1.print(cout);

    return 0;
}
```

# Méthode virtuelle appelée dans une autre méthode de la classe

- Quel sera le résultat de l'affichage dans le programme suivant:

```
void Employee::print(ostream& out) const
{
    ...
    out << getName() << endl;
    out << "Salary: " << getSalary() << endl;
}
int main() {
    Manager e1("John", 15000);
    e1.print(cout);

    return 0;
}
```

C'est la méthode de la classe **Employee** qui est appelée.

C'est la méthode de la classe **Manager** qui est appelée.

```
John
Salary: 17250
```



# Méthode virtuelle appelée dans une autre méthode de la classe

---

- Quel sera le résultat de l'affichage dans le programme suivant:

```
void Employee::print(ostream& out) const
{
    ...
    out << getName() << endl;
    out << "Salary: " << getSalary() << endl;
}
int main() {
    Secretary e1("John", 15000);
    e1.print(cout);

    return 0;
}
```

# Méthode virtuelle appelée dans une autre méthode de la classe

- Quel sera le résultat de l'affichage dans le programme suivant:

```
void Employee::print(ostream& out) const
{
    ...
    out << getName() << endl;
    out << "Salary: " << getSalary() << endl;
}
int main() {
    Secretary e1("John", 15000);
    e1.print(cout);

    return 0;
}
```

C'est la méthode de la classe **Employee** qui est appelée.

C'est la méthode de la classe **Employee** qui est appelée, puisqu'elle est héritée par la classe **Secretary**.  
Si la méthode virtuelle avait été redéfinie dans la classe **Secretary**, c'est celle-ci que l'on aurait utilisée.

```
John
Salary: 15000
```

# Méthode virtuelle appelée dans une autre méthode de la classe

---

- Pour avoir le résultat désiré, il faut que **getName()** soit elle aussi déclarée virtuelle:

```
class Employee
{
public:
    Employee(string name = "unknown", double salary = 0);
    virtual double getSalary() const;
    virtual string getName() const;
    /* ... */
private:
    string name_;
    double salary_;
};
```

# Méthode virtuelle appelée dans une autre méthode de la classe

- Quel sera le résultat de l'affichage dans le programme suivant:

```
void Employee::print(ostream& out) const
{
    /* ... */
    out << this->getName() << endl;
    out << "Salary: " << this->getSalary() << endl;
}
int main() {
    Employee e1("John", 15000);
    e1.print(cout);

    return 0;
}
```

Le type de **this** est **Employee\***, mais comme la méthode **getName()** est virtuelle, c'est la méthode de la classe dérivée qui sera appelée.

Ici, c'est la méthode de la classe dérivée qui sera appelée, puisque **getSalary()** est virtuelle.

# Méthode virtuelle appelée dans une autre méthode de la classe

- Quel sera le résultat de l'affichage dans le programme suivant:

```
void Employee::print(ostream& out) const
{
    /* ... */
    out << getName() << endl;
    out << "Salary: " << getSalary() << endl;
}
int main() {
    Employee e1("John", 15000);
    e1.print(cout);

    return 0;
}
```

C'est la méthode de la classe **Employee** qui est appelée.

C'est la méthode de la classe **Employee** qui est appelée.

```
John
Salary: 15000
```

# Méthode virtuelle appelée dans une autre méthode de la classe

---

- Quel sera le résultat de l'affichage dans le programme suivant:

```
void Employee::print(ostream& out) const
{
    /* ... */
    out << getName() << endl;
    out << "Salary: " << getSalary() << endl;
}
int main() {
    Manager e1("John", 15000);
    e1.print(cout);

    return 0;
}
```

# Méthode virtuelle appelée dans une autre méthode de la classe

- Quel sera le résultat de l'affichage dans le programme suivant:

```
void Employee::print(ostream& out) const
{
    /* ... */
    out << getName() << endl;
    out << "Salary: " << getSalary() << endl;
}
int main() {
    Manager e1("John", 15000);
    e1.print(cout);

    return 0;
}
```

C'est maintenant la méthode de la classe **Manager** qui est appelée.

C'est la méthode de la classe **Manager** qui est appelée.

```
John (Manager)
Salary: 17250
```

# Méthode virtuelle appelée dans une autre méthode de la classe

---

- Quel sera le résultat de l'affichage dans le programme suivant:

```
void Employee::print(ostream& out) const
{
    /* ... */
    out << getName() << endl;
    out << "Salary: " << getSalary() << endl;
}
int main() {
    Secretary e1("John", 15000);
    e1.print(cout);

    return 0;
}
```



# Méthode virtuelle appelée dans une autre méthode de la classe

- Quel sera le résultat de l'affichage dans le programme suivant:

```
void Employee::print(ostream& out) const
{
    /* ... */
    out << getName() << endl;
    out << "Salary: " << getSalary() << endl;
}
int main() {
    Secretary e1("John", 15000);
    e1.print(cout);

    return 0;
}
```

C'est maintenant la méthode de la classe **Secretary** qui est appelée.

C'est la méthode de la classe **Employee** qui est appelée, puisqu'elle est héritée par la classe **Secretary**.

```
John (Secretary)
Salary: 17250
```

# Destructeur virtuel

---

- Soient les classes suivantes:

```
class StockExchangeEntity
{
public:
    StockExchangeEntity();
    ~StockExchangeEntity();
    virtual void addShare();
};
```

```
class Company
    : public StockExchangeEntity
{
public:
    Company();
    ~Company();
    virtual void addShare();
private:
    Employee *president_;
};
```

# Destructeur virtuel

- Soient les classes suivantes:

```
Company::Company()  
    : /* ... */  
{  
    president_ = new Employee(/* ... */);  
    /* ... */  
}
```

```
Company::~~Company()  
{  
    delete president_;  
    /* ... */  
}
```

Cette classe doit avoir un destructeur puisqu'il faut désallouer le pointeur qui a été alloué par le constructeur.

## Destructeur virtuel (suite)

---

- Considérons maintenant le programme suivant:

```
int main ()
{
    StockExchangeEntity *s = new Company();
    /* ... */
    delete s;
    /* ... */
}
```

- Quel destructeur sera appelé lors de la désallocation de **s**?

## Destructeur virtuel (suite)

---

Le destructeur de la classe `StockExchangeEntity` n'a pas été déclaré virtuel

Ce n'est donc pas le destructeur de la classe `Company` qui sera appelé, même si l'objet appartient en fait à cette classe

- Le pointeur `president_` ne sera donc pas désalloué et on aura une fuite de mémoire

Il est donc important de déclarer virtuel le destructeur de `StockExchangeEntity`

- **Ceci sera vrai pour toutes les situations où on utilise le polymorphisme**

# Destructeur virtuel (suite)

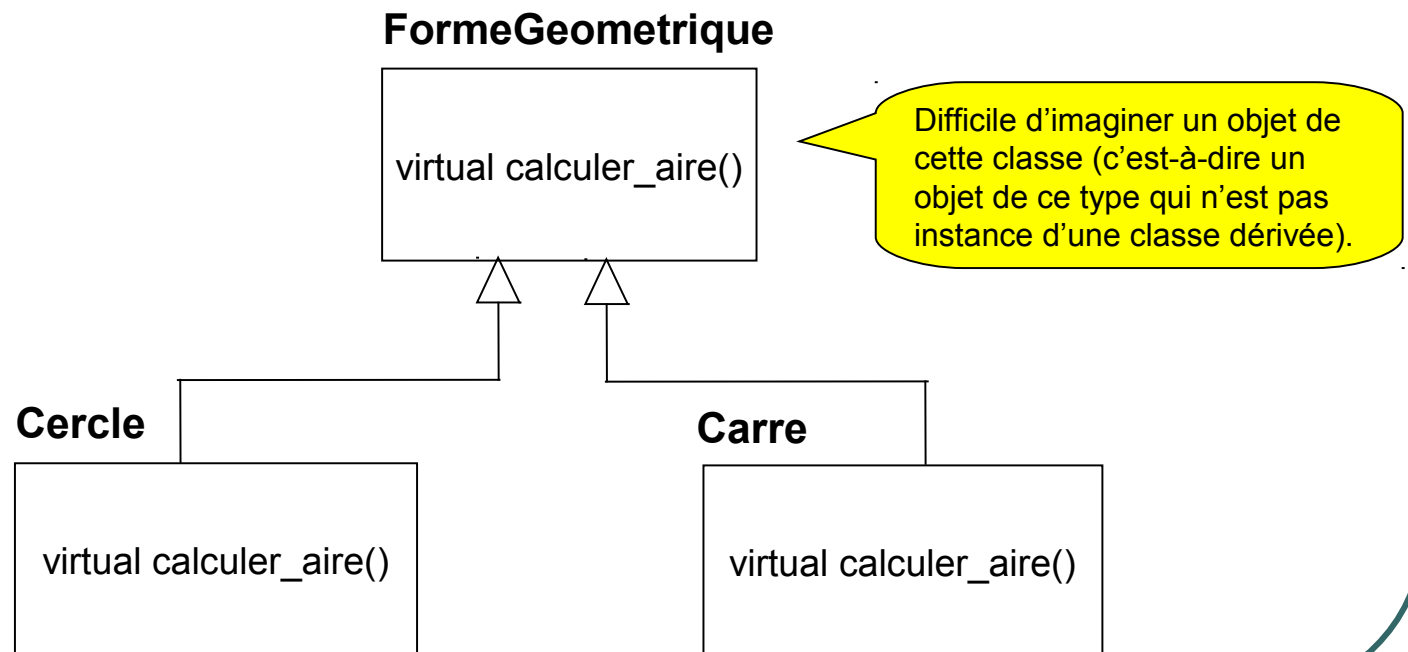
---

```
class StockExchangeEntity
{
public:
    StockExchangeEntity();
    virtual ~StockExchangeEntity();
    virtual void addShare();
};

class Company : public StockExchangeEntity
{
public:
    Company();
    virtual ~Company();
    virtual void addShare();
private:
    Employee *president_;
};
```

# Classes abstraites

- Soit la hiérarchie de classe suivante:



## Classes abstraites (suite)

---

- La classe **FormeGeometrique** est une classe abstraite
- Aucun objet ne peut appartenir directement à cette classe
- Un objet ne peut appartenir qu'à une classe dérivée
- En fait, la classe **FormeGeometrique** ne sert qu'à établir les méthodes qui seront héritées et certains attributs qui seront partagés par toutes les classes dérivées



# Classes abstraites (suite)

---

- En C++, pour définir une classe abstraite, il suffit d'y déclarer **au moins une** fonction virtuelle pure
  - Pour déclarer une fonction virtuelle pure, il suffit d'ajouter « = 0 » après la déclaration d'une fonction virtuelle
- Voici par exemple comment déclarer virtuelle pure la fonction **addShare()** de la classe `StockExchangeEntity`:

```
virtual void addShare() = 0;
```

- Si une classe contient une fonction virtuelle pure, il sera interdit de déclarer un objet de cette classe

## Obtention du type à l'exécution

---

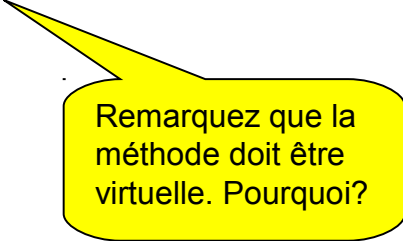
- Supposons que nous avons un vecteur de pointeurs d'employés
- Nous voudrions compter le nombre de secrétaires parmi ces employés
- Une façon de le faire serait d'ajouter une méthode getType() qui retournerait le type de l'objet

# Obtention du type à l'exécution (suite)

---

```
class Employee
{
public:
    Employee(string name = "unknown", double salary = 0);
    virtual string getType() const;
    /* ... */
private:
    /* ... */
};

string Employee::getType() const
{
    return "Employee";
}
```



Remarquez que la méthode doit être virtuelle. Pourquoi?

# Obtention du type à l'exécution (suite)

---

```
class Manager : public Employee
{
public:
    /* ... */
    virtual string getType() const;
    /* ... */
private:
    /* ... */
};

string Manager::getType() const
{
    return "Manager";
}
```

# Obtention du type à l'exécution (suite)

---

```
class Secretary : public Employee
{
public:
    /* ... */
    virtual string getType() const;
    /* ... */
private:
    /* ... */
};
string Secretary::getType() const
{
    return "Secretary";
}
```

# Obtention du type à l'exécution (suite)

---

```
int main()  
{  
    vector<Employee*> v;  
    /* ... */  
  
    int nbSecretary = 0;  
    for (size_t i = 0; i < v.size(); ++i) {  
        if (v[i]->getType() == "Secretary")  
            ++nbSecretary;  
    }  
  
    /* ... */  
    return 0;  
}
```

# Obtention du type à l'exécution (suite)

---

- Le problème avec cette approche est qu'elle exige de programmer une méthode pour chaque classe
- En plus, il faut gérer nous-même un ensemble de descripteurs de types (remarquez que l'utilisation de string n'est pas très efficace)
- Tout ça alors qu'on sait à l'exécution à quelle classe on a affaire
- Meilleure approche: utiliser l'opérateur ***typeid*** de C++
- *Attention:* il faut éviter le plus possible d'utiliser cet opérateur

# Opérateur typeid

---

```
#include <typeinfo>
/* ... */
int main()
{
    vector<Employee*> v;
    /* ... */
    int nbSecretary = 0;
    for (size_t i = 0; i < v.size(); ++i) {
        if (typeid(*v[i]) == typeid(Secretary))
            ++nbSecretary;
    }
    /* ... */
    return 0;
}
```



# Comment le polymorphisme fonctionne-t-il?

---

Soit les classes suivantes:

```
class A
{
public:
    A();
    virtual void f1();
    virtual void f2();
private:
    int attA_;
};
```

```
class B : public A
{
public:
    virtual void f1();
    virtual void f2();
    ...
private:
    ...
};
```

```
class C : public A
{
public:
    virtual void f1();
    ...
private:
    ...
};
```

# Comment le polymorphisme fonctionne-t-il?

---

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i) {
        v[i]->f1();
        v[i]->f2();
    }
}
```

Comment fait-on pour  
savoir quelle méthode  
appeler?

# Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i) {
        v[i]->f1();
        v[i]->f2();
    }
}
```

i = 0

Ici il faut appeler A::f1()

# Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i) {
        v[i]->f1();
        v[i]->f2();
    }
}
```

i = 0

Ici il faut appeler A::f2()

# Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i) {
        v[i]->f1();
        v[i]->f2();
    }
}
```

i = 1

Ici il faut appeler **B::f1()**

# Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i) {
        v[i]->f1();
        v[i]->f2();
    }
}
```

i = 1

Ici il faut appeler B::f2()

# Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i) {
        v[i]->f1();
        v[i]->f2();
    }
}
```

i = 2

Ici il faut appeler C::f1()

# Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());

    for (size_t i = 0; i < v.size(); ++i) {
        v[i]->f1();
        v[i]->f2();
    }
}
```

*i = 2*

Ici il faut appeler **A::f2()**,  
puisque la méthode n'est  
pas redéfinie dans **C**.



# Comment le polymorphisme fonctionne-t-il?

---

- Normalement, avec l'héritage, on peut savoir dès la compilation quelle méthode doit être appelée
- Ici, ce n'est pas possible, puisqu'on ne peut pas toujours savoir quel est le type d'objet réellement pointé par le pointeur
- On a vu, dans l'exemple précédent, que le type peut changer lors de l'exécution

# Comment le polymorphisme fonctionne-t-il?

---

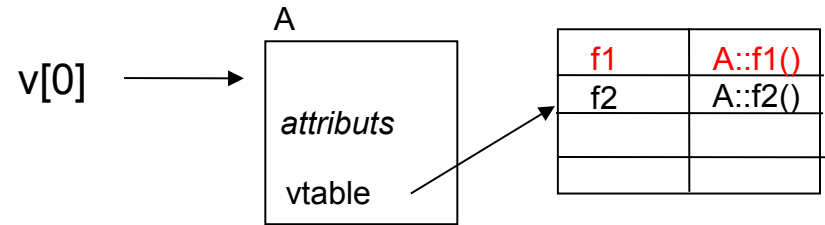
- Quand un objet est une instance d'une classe contenant des méthodes virtuelles, cet objet, en plus d'avoir de l'espace alloué pour ses attributs, aura un pointeur à une table appelée *vtable*
- La table *vtable* indique, pour chaque fonction virtuelle, quelle fonction doit être appelée
- Ainsi, dans l'appel `objet1->f1()`, on n'a qu'à rechercher « f1 » dans la *vtable*, et exécuter la fonction spécifiée
- Tous les objets d'une même classe pointent vers la même *vtable*

# Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());
```



`i = 0`

```
for (size_t i = 0; i < v.size(); ++i) {
    v[i]->f1();
    v[i]->f2();
}
```

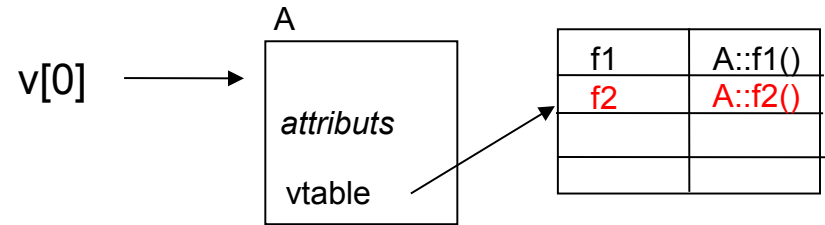
Ici il faut appeler `A::f1()`

# Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());
```



`i = 0`

```
for (size_t i = 0; i < v.size(); ++i) {
    v[i]->f1();
    v[i]->f2();
}
```

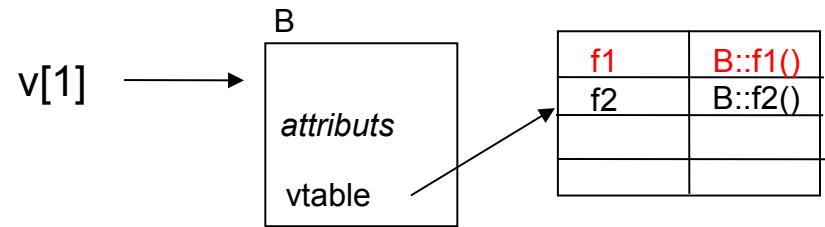
Ici il faut appeler `A::f2()`

# Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());
```



$i = 1$

```
for (size_t i = 0; i < v.size(); ++i) {
    v[i]->f1();
    v[i]->f2();
}
```

Ici il faut appeler **B::f1()**

# Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

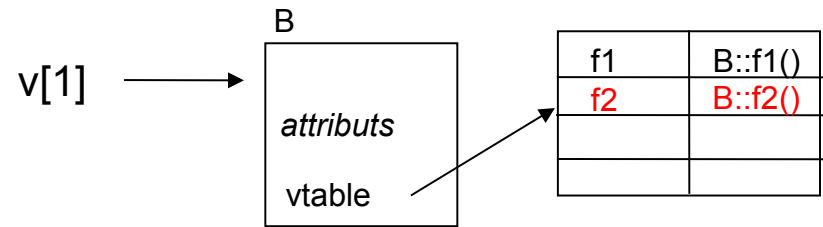
```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());
```

`i = 1`

```
    for (size_t i = 0; i < v.size(); ++i) {
        v[i]->f1();
        v[i]->f2();
    }
}
```

Ici il faut appeler **B::f2()**



# Comment le polymorphisme fonctionne-t-il?

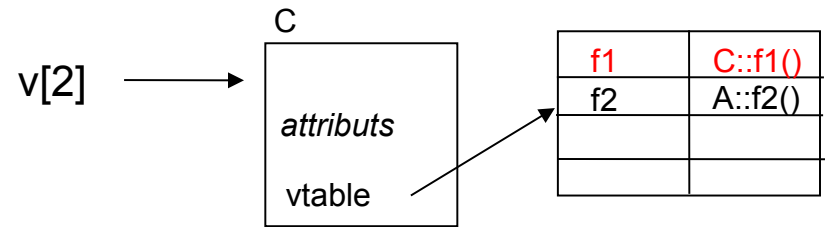
- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());
```

*i = 2*

```
    for (size_t i = 0; i < v.size(); ++i) {
        v[i]->f1();
        v[i]->f2();
    }
}
```



Ici il faut appeler `C::f1()`

# Comment le polymorphisme fonctionne-t-il?

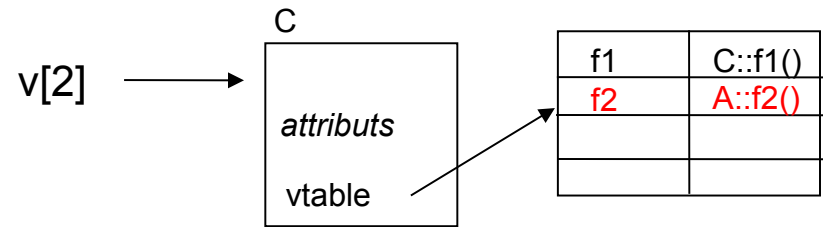
- Soit le programme suivant:

```
int main()
{
    vector< A* > v;

    v.push_back(new A());
    v.push_back(new B());
    v.push_back(new C());
```

`i = 2`

```
    for (size_t i = 0; i < v.size(); ++i) {
        v[i]->f1();
        v[i]->f2();
    }
}
```



Ici il faut appeler **A::f2()**,  
puisque la méthode n'est  
pas redéfinie dans **C**