Programmation orientée objet

Polymorphisme – Concepts de base

Qu'est-ce que le polymorphisme?

- Le polymorphisme est un concept fondamental de la programmation orientée objet
- Il permet de prendre en compte le type réel d'un pointeur ou d'une référence lorsqu'il est manipulé au travers de l'interface d'une classe de base
- Il s'applique donc à l'exécution du programme

Méthode virtuelle

- En C++, le polymorsphisme fonctionne en déclarant des méthodes virtuelles:

```
class Clock {
public:
    Clock(bool useMilitary) {}
    string getLocation() const { return "Local"; }
    virtual int getHours() const;
    int getMinutes() const;
    bool isMilitary() const;
private:
    bool military_;
};
```

Méthode virtuelle (suite)

- Le type d'une copie d'un objet est déterminé à la compilation, **le polymorphisme ne s'applique donc pas dans le cas de copies**
- Une méthode virtuelle prend effet seulement au travers de **pointeur et de référence** d'objet puisque leur type est déterminé à l'exécution du programme

Méthode virtuelle (suite)

- Considérons le programme suivant:

```
int main() {
    Clock c1(true);
    TravelClock c2(true, "Paris", 6);
    TravelClock c3(true, "Vancouver", -3);

    vector<Clock*> clocks = {&c1, &c2, &c3};

    for (Clock* c : clocks) {
        cout << c->getHours() << ":" << c->getMinutes() << endl;
    }
}
```

Nous avons un vecteur de pointeurs de Clock, donc les méthodes déclarées virtuelles seront choisies à l'exécution

Méthode déclarée virtuelle, le choix de la méthode qui sera appelée est base sur la classe réel de l'objet pointé

Méthode standard, la méthode applelée est déterminée à la compilation et base sur le type du pointeur

Méthode virtuelle (suite)

- Lors de l'exécution du programme, les méthodes appelées seront:

1. Clock::getHours()
2. Clock::getMinutes()
3. TravelClock::getHours()
4. Clock::getMinutes
5. TravelClock::getHours()
6. Clock::getMinutes()

Pointeur vers c1 de type Clock

Pointeur vers c2 de type TravelClock

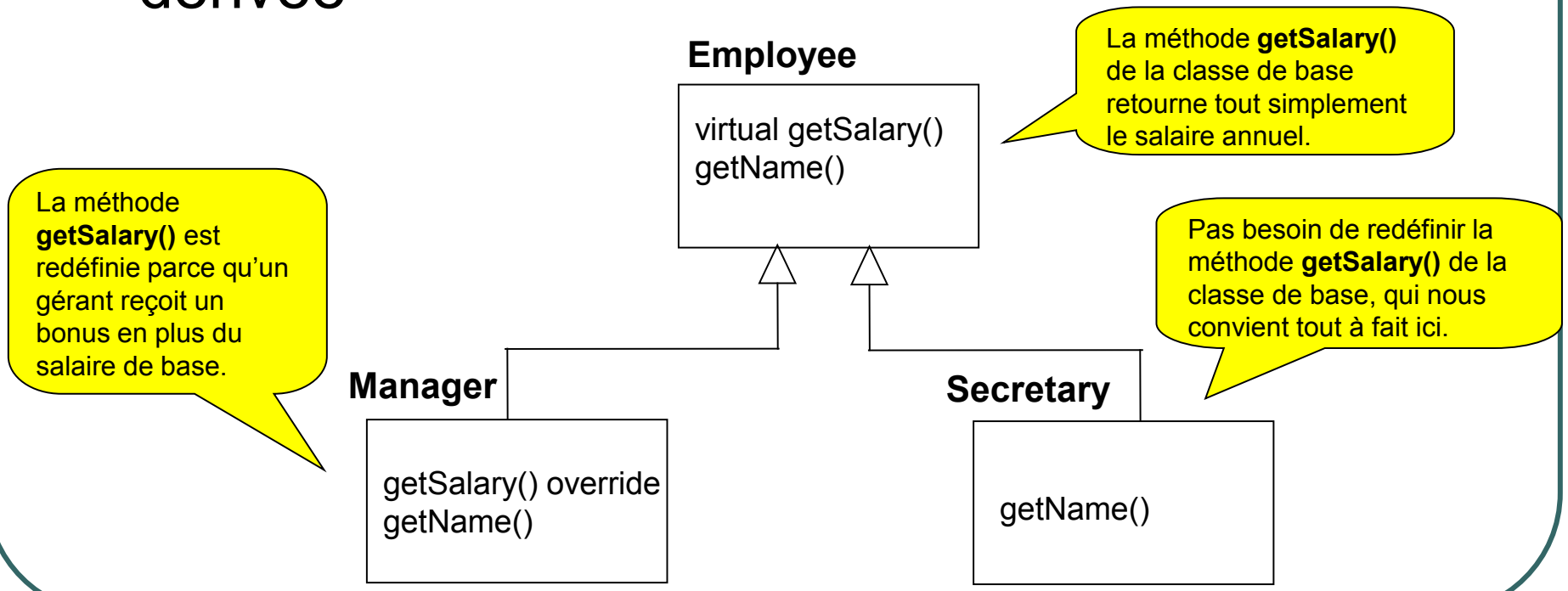
Pointeur vers c3 de type TravelClock

Méthode virtuelle (suite)

- *Attention*: si une méthode est déclarée virtuelle dans une classe, elle le sera automatiquement dans toutes les classes qui en dérivent (**le polymorphisme est hérité**)
- Pour éviter toute confusion, on ajoute le mot clé « override » à la signature des méthodes virtuelles 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
  - « override » s'assure que la méthode était déjà virtuelle dans la classe de base

Polymorphisme et méthode héritée

- Même si une méthode est virtuelle, on est libre de la redéfinir ou non dans une classe dérivée



Choix du type

- Le *type statique* est déterminé lors de la compilation
- Le *type dynamique* est déterminé seulement lors de l'exécution

```
int main() {
    vector<unique_ptr<Employee>> employees;
    employees.push_back(make_unique<Secretary>("Mark", 16000.0));
    employees.push_back(make_unique<Manager>("Jenny", 12000.0, 10));

    for (const unique_ptr<Employee>& emp : employees) {
        cout << emp->getSalary() << endl;
    }
}
```

Types de liaison

- Il y a *liaison statique* lorsque la méthode appelée est choisie à la compilation et donc basée sur son type statique
- Il y a *liaison dynamique* lorsque la méthode appelée est choisie à l'exécution et donc basée sur son type dynamique (méthodes virtuelles)

```
int main() {
    vector<unique_ptr<Employee>> employees;
    employees.push_back(make_unique<Secretary>("Mark", 16000.0));
    employees.push_back(make_unique<Manager>("Jenny", 12000.0, 10));
    for (const unique_ptr<Employee>& emp : employees) {
        cout << emp->getSalary() << endl;
        cout << emp->getName() << endl;
    }
}
```

Exemple avec Employee et ses dérivées

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

Exemple avec Employee et ses dérivées

```
class Manager : public Employee {
public:
    Manager(string name = "unknown",double salary = 0,
        double bonus = 0) : Employee(name, salary),
        bonus (bonus) {}
    double getSalary() const override {
        return Employee::getSalary() * (1 + bonus_ / 100);
    }
    string getName() const {
        return Employee::getName() + " (Manager)";
    }
private:
        double bonus_;
};
```

Exemple avec Employee et ses dérivées

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

Polymorphisme et méthode héritée (exemple)

```
void afficher_statique(Employee employee) {
    cout << "Name: " << employee.getName() << ", " << "Salary: "
    << employee.getSalary() << endl;
}

void afficher_dynamique(const Employee& employee) {
    cout << "Name: " << employee.getName() << ", " << "Salary: "
    << employee.getSalary() << endl;
}

void afficher_dynamique(Employee* employee) {
    cout << "Name: " << employee->getName() << ", " << "Salary:
    " << employee->getSalary() << endl;
}
```

Polymorphisme et méthode héritée (exemple)

```
int main() {
    vector<unique_ptr<Employee>> employees;
    employees.push_back(make_unique<Secretary>("Mark", 16000.0));
    employees.push_back(make_unique<Manager>("Jenny", 12000.0, 10));

    for (const unique_ptr<Employee>& emp : employees) {
        afficher_statique(*emp);
        afficher_dynamique(*emp);
        afficher_dynamique(emp.get());
    }
}
```

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()** qui n'est pas virtuelle et définie de la manière suivante:

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

Méthode virtuelle appelée dans une autre méthode de la classe

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

```
int main() {
    Manager manager("John", 15000, 15);
    manager.print(cout);
}
```

```
John
Salary: 17250
```

- Pour répondre, considérez le code équivalent suivant:

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

Méthode virtuelle appelée dans une autre méthode de la classe

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

```
int main() {
    Secretary secretaire("John", 15000);
    secretaire.print(cout);
}
```

```
John
Salary: 15000
```

- Pour répondre, considérez le code équivalent suivant:

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

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;
    void print(ostream& out) 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:

```
int main() {
    Employee employe("John", 15000);
    employe.print(cout);
}
```

```
John
Salary: 15000
```

- Pour répondre, considérez le code équivalent suivant:

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

Méthode virtuelle appelée dans une autre méthode de la classe

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

```
int main() {
    Manager manager("John", 15000, 15);
    manager.print(cout);
}
```

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

- Pour répondre, considérez le code équivalent suivant:

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

Méthode virtuelle appelée dans une autre méthode de la classe

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

```
int main() {
    Secretary secretaire("John", 15000);
    secretaire.print(cout);
}
```

```
John (Secretary)
Salary: 15000
```

- Pour répondre, considérez le code équivalent suivant:

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

Programmation orientée objet

Destructeurs virtuels

Destructeur virtuel

- Soient les classes suivantes:

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

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

Destructeur virtuel (suite)

- Considérons maintenant le programme suivant:

```
int main() {
        unique_ptr<StockExchangeEntity> s = make_unique<Company>();
}
```

- À la sortie du programme, quel destructeur sera appelé lors de la désallocation de l'espace mémoire gérée par **s**?

Destructeur virtuel (suite)

- Si on regarde en détail la définition de la classe StockExchangeEntity:

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

Destructeur virtuel (suite)

- Étant donné que le destructeur de la classe StockExchangeEntity n'a pas été déclaré virtuel, le destructeur de la classe Company ne sera jamais appelé
- Il est donc important de déclarer virtuel le destructeur de StockExchangeEntity
- Ceci sera vrai pour toutes les situations où on utilise le polymorphisme. **Toute classe de base polymorphe devrait avoir un destructeur virtuel**

Destructeur virtuel (suite)

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

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

Si ce destructeur n'a rien à faire de spécial. (C++11)

La convention actuelle est de mettre "virtual", pas "override", sur le destructeur,

car un destructeur n'"override" pas les autres, au sens anglais du mot, il s'ajoute aux autres.

Programmation orientée objet

Fonctionnement du polymorphisme

Comment le polymorphisme fonctionne-t-il?

Soient les classes suivantes:

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

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

```
class C : public A
{
public:
   void f1() override;
   ...
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*

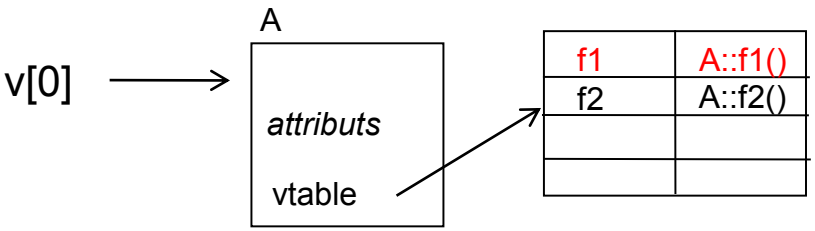
Comment le polymorphisme fonctionne-t-il?

- Soit le programme suivant:

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

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

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



i = 0

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

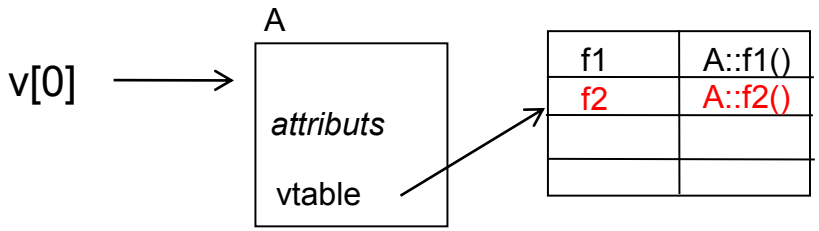
Comment le polymorphisme fonctionne-t-il?

- 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()**

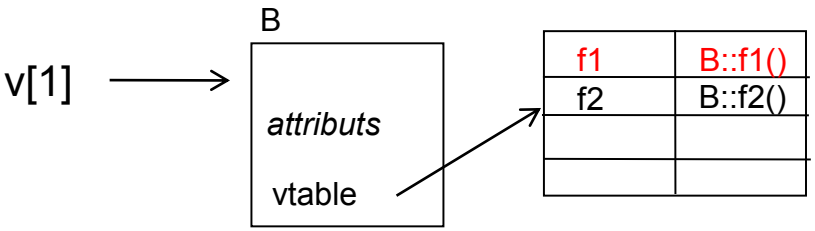
Comment le polymorphisme fonctionne-t-il?

- 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()**

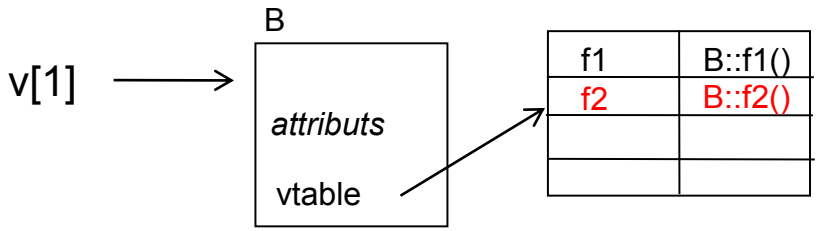
Comment le polymorphisme fonctionne-t-il?

- 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()**

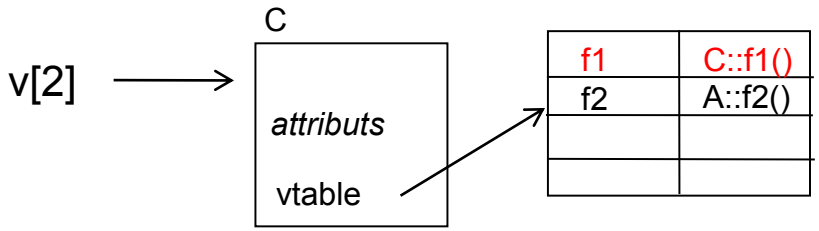
Comment le polymorphisme fonctionne-t-il?

- 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()**

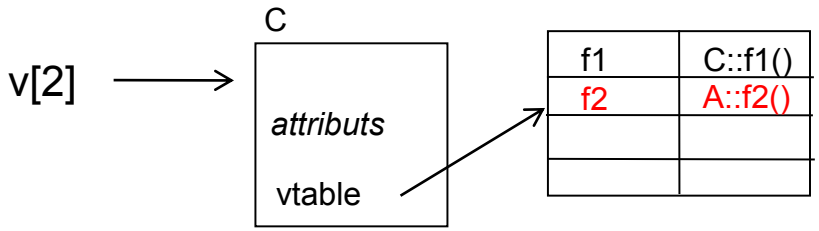
Comment le polymorphisme fonctionne-t-il?

- 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

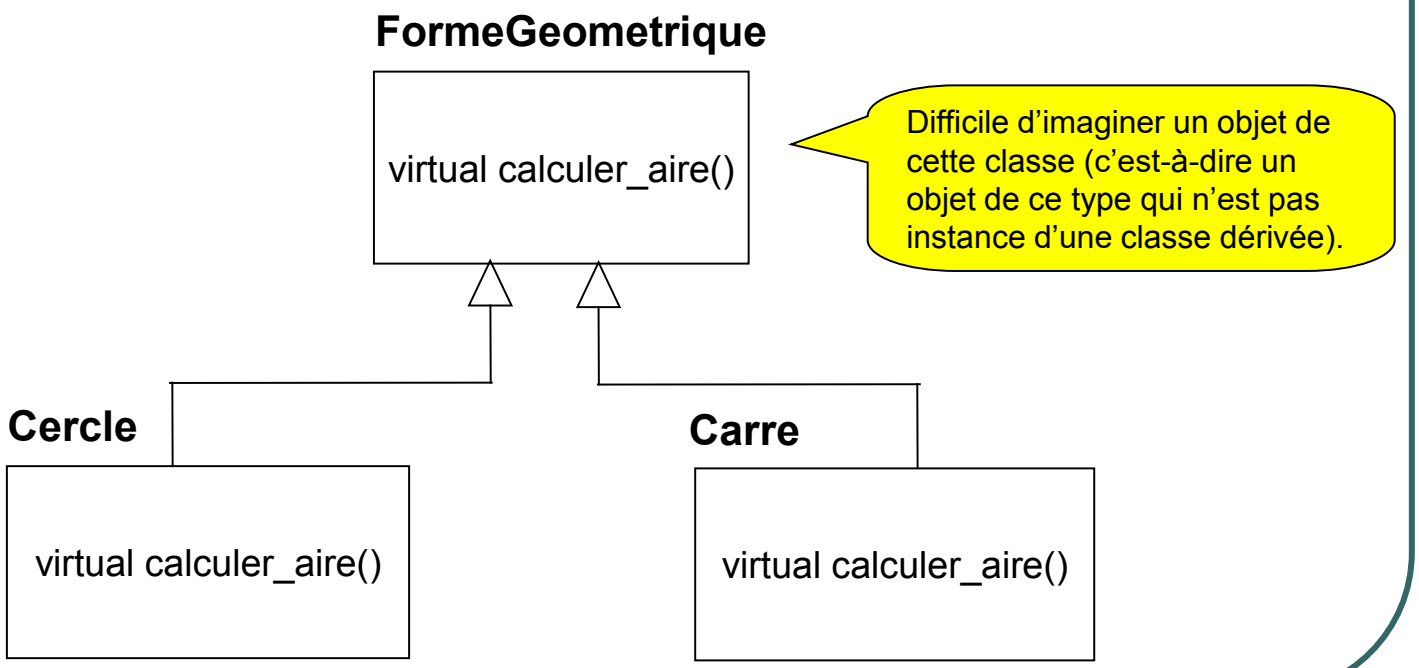
Ici il faut appeler **A::f2()**, puisque la méthode n'est pas redéfinie dans **C**

Programmation orientée objet

Classes abstraites

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**