

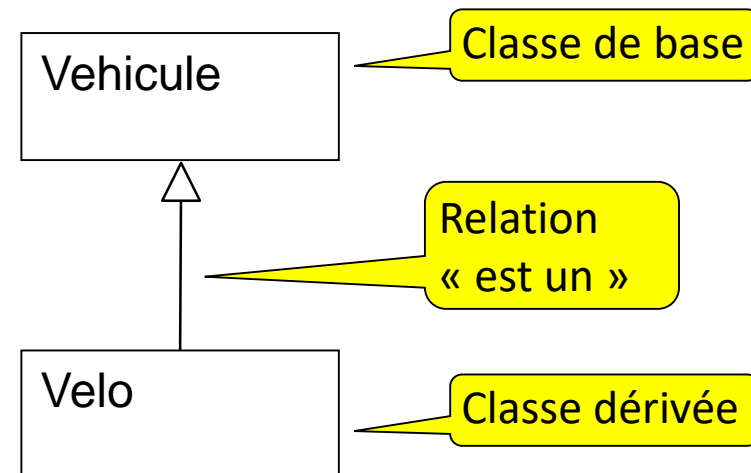
Méthodes virtuelles et classes abstraites

Héritage, redéfinition de méthodes,
polymorphisme, virtual, « slicing »,
classes abstraites et pures, typeid

Héritage (rappel)

- Une classe enfant/dérivée hérite des attributs et méthodes d'une classe parent/base.
 - Elle peut ajouter des méthodes et attributs.
 - Elle peut redéfinir des méthodes pour en changer le comportement par rapport à son parent.
- En C++
 - `class Vehicule { ... };`
 - `class Velo : public Vehicule { ... };`

Nous utiliserons toujours l'héritage public; l'héritage privé dit presque toujours « est implémenté en terme de » qui devrait être fait par composition si possible.



Signification de l'héritage

- Attention, héritage = « est un »
- Ne pas confondre avec « est implémenté en terme de » ou « est composé de » = composition
- Le résultat paraît parfois équivalent, mais les concepts sont différents et la protection par encapsulation est différente
- Ex: Point et Cercle; Cercle: point (centre) et rayon
 - Cercle hérite de Point et ajoute un attribut rayon?
 - Question: un cercle est-il un point?
 - Ou Point hérite de Cercle et force le rayon à zéro?

Signification de l'héritage (suite)

- Soit une classe Triangle: 3 points (sommets)
- On veut définir une classe Fleche, constituée d'un triangle et d'une droite perpendiculaire à un des côtés du triangle
- Fleche hérite de Triangle, et ajoute un attribut pour la droite?
- Est-ce raisonnable? Une flèche est-elle un triangle?
- → il faut décider du lien entre classes: héritage, composition ou agrégation

Exemple de l'horloge

- Soit une classe Clock, qui permet d'obtenir l'heure locale, de deux façons: am/pm ou 24h (norme « militaire »)

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

On n'a pas de constructeur par défaut

Unique attribut, de valeur spécifiée à la construction de l'objet; détermine le format de l'affichage de l'heure.

Utilisation:

```
Clock horloge1(true);
```

Ex: « 23:45 »

```
Clock horloge2(false);
```

Ex: « 11:45 »

Exemple de l'horloge (suite)

- On aimerait une horloge pour l'heure d'une zone autre que locale

```
class TravelClock : public Clock
{
public:
    TravelClock(bool useMilitary, string location,
                int timeDifference);
    string getLocation() const { return location_; }
    int getHours() const;
private:
    string location_;
    int timeDifference_;
};
```

Toujours pas de constructeur par défaut

Implémentation de méthode complètement différente de la classe de base. Dit redéfinition (« override »)

Ajoute deux attributs

Méthode étendue de la classe de base pour ajouter le décalage horaire

Constructeur de la classe dérivée

- Attention: la classe de base est construite avant le corps du constructeur de la classe dérivée

```
TravelClock::TravelClock(bool useMilitary, string location,  
                           int timeDifference) :
```

```
    Clock(useMilitary),  
    location_(location),  
    timeDifference_(timeDifference)
```

```
{}
```

Nécessaire pour spécifier comment construire la classe de base (sauf si construction par défaut)

L'initialisation des autres membres pourrait être dans le corps

Appel des méthodes de la classe de base

- On peut appeler une méthode de la classe de base même si la classe dérivée la redéfinit

Redéfinition qui utilise la méthode de la classe de base dite étendue

```
int TravelClock::getHours() const
{
    int h = Clock::getHours() + timeDifference_;
    return modulo_positif(h, isMilitary() ? 24 : 12);
}
```

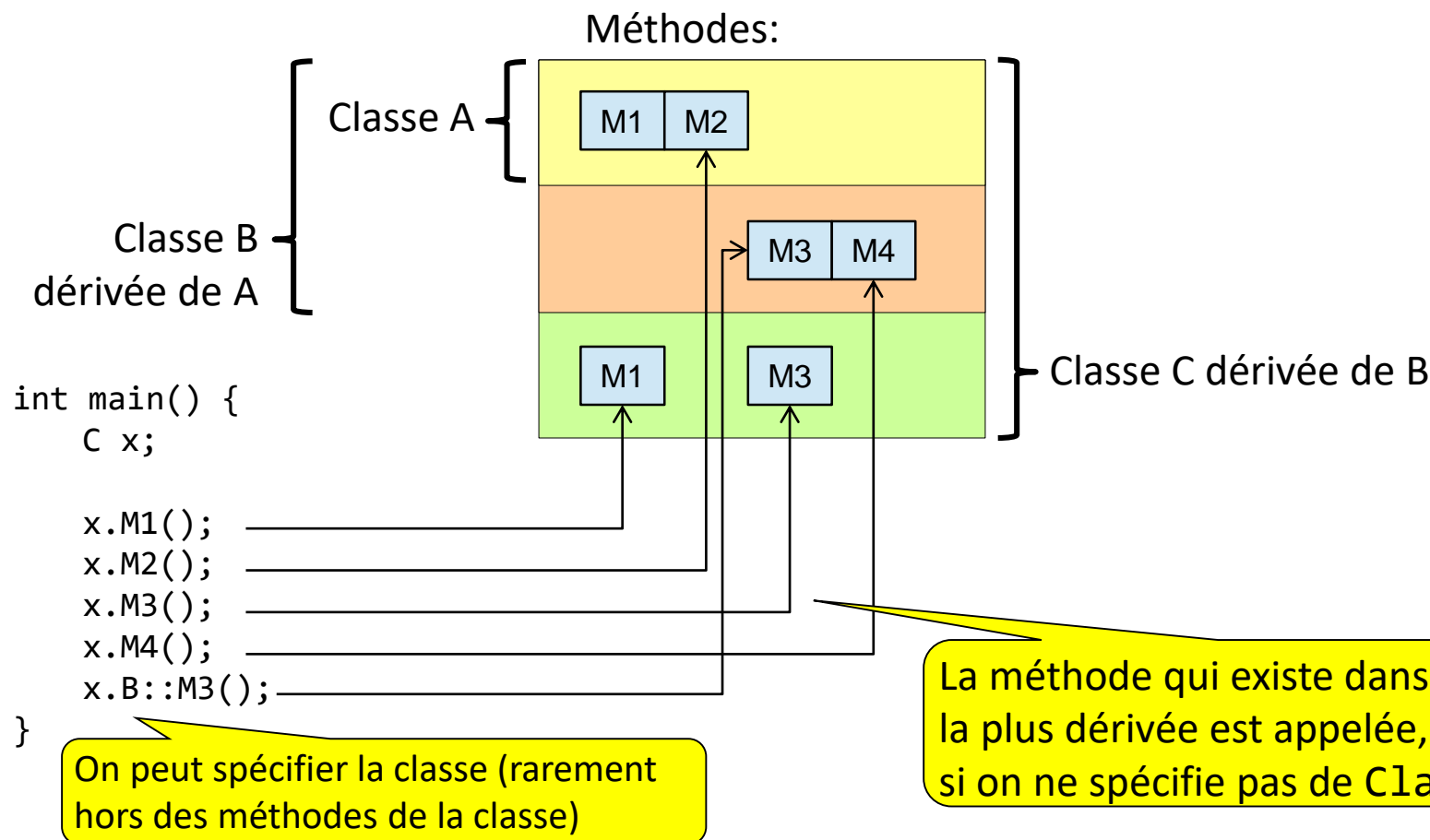
Spécifier la classe de la méthode pour appeler celle de la classe de base

```
int modulo_positif(int x, int n) {
    return (x % n) + (x < 0 ? n : 0);
}
```

Méthode héritée, pas besoin d'indiquer la classe

Opérateur ternaire, comme Python:
n if x < 0 else 0

Appel d'une méthode



Accès aux membres d'une classe de base

- **private** : accessible seulement dans la classe (et friend)
- **protected** : accessible à la classe et ses classes dérivées (et friend)
- En général, un attribut est toujours privé
- → pour qu'une classe dérivée puisse accéder à un attribut de la classe de base, on lui fournira une méthode d'accès protégée, s'il n'en existe pas déjà une publique

Exemple accès protégé / privé

```
class Base
```

```
{
```

```
public: — Accessible à tous
```

```
    int getA() const;
```

```
    void setA(int x);
```

```
    int getB() const;
```

Attribut accessible à tous en lecture seulement

```
protected: — Accessible aux classes dérivées et friend
```

```
    void setB(int x);
```

```
    int getC() const;
```

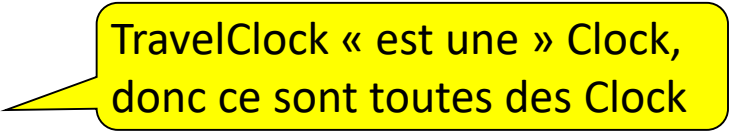
```
    int setC(int x);
```

```
private:
```

```
    int a_, b_, c_; — Accessible juste dans cette classe
```

```
};
```

Polymorphisme par héritage (rappel)

- Un objet peut avoir « plusieurs formes »
 - Avec une classe de base commune
 - Principe de substitution de Liskov dit en gros:
 - on peut utiliser un objet de classe dérivée comme un objet de la classe de base
- Supposons qu'on déclare trois horloges:
 - `Clock c1(true);`
 - `TravelClock c2(true, "Paris", +6);`
 - `TravelClock c3(true, "Vancouver", -3);`
 - Peut-on appeler une fonction `f(Clock c)` ?
 - Peut-on les mettre dans un `vector<Clock>` ?

Polymorphisme, premier test

- Fonction d'affichage comme on a l'habitude

Comme on passe les objets d'habitude

```
ostream& operator<< (ostream& os, const Clock& clock)
{
    return os << clock.getLocation() << ": " <<
        clock.getHours() << ":" << clock.getMinutes();
}
```

- On l'utilise

```
Clock c1(true);
TravelClock c2(true, "Paris", +6);
TravelClock c3(true, "Vancouver", -3);
cout << c1 << ", " << c2 << ", " << c3 << endl;
```

Compile mais affiche 3 fois « Local: 23:45 »
??

Polymorphisme, deuxième test : template

- Fonction d'affichage template (comme vu avant)

Concept C++20 dans <concepts>

```
template <derived_from<Clock> T>  
ostream& operator<< (ostream& os, const T& clock)  
{ ... même corps ... }
```

- On l'utilise

```
Clock c1(true);  
TravelClock c2(true, "Paris", +6);  
TravelClock c3(true, "Vancouver", -3);  
cout << c1 << ", " << c2 << ", " << c3 << endl;
```

Ok: « Local: 23:45, Paris: 5:45, Vancouver: 20:45 »

Mais ça compile 2 versions, c'est du polymorphisme ad-hoc, pas par héritage

Polymorphisme ad-hoc ou par héritage

● La version template fonctionne?

Pointeurs vers les horloges qui existent déjà, pour être certain que ce n'est pas un problème de constructeur de copie ; la conversion de pointeur ne modifie pas l'objet

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

```
for (auto&& horloge : horloges)
```

```
    cout << *horloge << ", ";
```

```
cout << endl;
```

auto pour être certain qu'on a le bon type

Affiche 3 fois « Local: 23:45 »

● Pourquoi?

Type déterminé et fixé à la compilation

- Le type statique du pointeur est Clock*

- << est appelé avec type statique Clock&

Pourquoi & au lieu * ?

- Dans <<, la version de getHours est choisie selon le type statique

Dit aiguillage statique (ou liaison statique) « static dispatch »

Polymorphisme par héritage, méthodes virtuelles

- On aimerait dire

- Le pointeur est Clock* mais j'aimerais appeler la méthode selon le « type réel » de l'objet pointé d'une classe possiblement dérivée de Clock

Son type dynamique, pouvant changer à l'exécution.

Dit aiguillage dynamique (ou liaison dynamique) « dynamic dispatch »

- Méthode virtuelle


```
class Clock
{ ...
    virtual string getLocation() const;
    virtual int getHours() const;
    ...
};
```

Un objet avec **au moins une méthode virtuelle** est dit polymorphe

On veut l'aiguillage dynamique

La boucle précédente donne maintenant ce qu'on aimerait et on n'a pas besoin de << template

Méthodes virtuelles (suite)

- *Attention:* si une méthode est virtuelle dans une classe, elle le sera automatiquement dans toutes ses classes dérivées
- Pour éviter toute confusion C++11
 - Déclarer la méthode **override** dans la classe dérivée
 - S'assure que la méthode est virtuelle dans la classe de base
 - Vieux C++: déclarer **virtual** aussi dans les classes dérivées
- On n'a alors pas besoin d'aller consulter la classe de base pour savoir si une méthode est virtuelle

Pourquoi pas tout « virtual »?

Similaire à Python

- Il y a un coût:

- Ajout d'un pointeur vers vtable
- Appel indirect par table de pointeurs de fonctions
- Le compilateur ne sais pas quelle fonction est utilisée
 - → ne peut pas optimiser l'appel avec « inlining »

Vu en INF1600

Enlever l'appel et le remplacer par ce que fait la fonction, ou directement son résultat

- « In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for » - Bjarne Stroustrup

(créateur du C++) La conception du langage est beaucoup basée sur ce principe

Destructeur virtuel

- On a un tableau qui possède des horloges

```
{
    array<unique_ptr<Clock>, 2> horloges = {
        make_unique<Clock>(true),
        make_unique<TravelClock>(true, "Paris, capitale de la France", +6)
    };
    ...
}
```

Pointeurs intelligents, aucun « new » dans notre programme

Fuite de mémoire (et *undefined behavior*); la string n'est pas désallouée

- Le destructeur de TravelClock n'est pas appelé

- Le destructeur n'est pas virtuel: choisi par le type statique Clock

On n'a pas déclaré de destructeur mais un est créé automatiquement si un attribut en a un

- **Toute classe de base polymorphe devrait avoir un destructeur virtuel**

(ou protégé, dans des cas pas matière au cours)

Exemple de l'horloge, résumé

```
class Clock
{
public:
    Clock(bool useMilitary);
    virtual ~Clock() = default;
    virtual string getLocation() const;
    virtual int getHours() const;
    int getMinutes() const;
    bool isMilitary() const;
private:
    bool isMilitary_;
};
```

Destructeur par défaut ici suffisant, mais doit spécifier **virtual** car la classe est polymorphe

Méthodes virtuelles pour polymorphisme (aiguillage dynamique)

```
class TravelClock : public Clock
{
public:
    TravelClock(bool useMilitary, string location,
                int timeDifference);
    string getLocation() const override;
    int getHours() const override;
private:
    string location_;
    int timeDifference_;
};
```

Redéfinition des méthodes virtuelles pour la classe dérivée; dire **override** pour lisibilité et s'assurer qu'il y a bien le virtual sur les bonnes méthodes de la classe de base

« Object slicing »

- Peut-on mettre les objets dans le vector?

```
vector<Clock> horloges = { c1, c2, c3 };  
for (auto&& horloge : horloges)  
    cout << horloge << ", ";  
cout << endl;
```

Affiche 3 fois « Local: 23:45 »

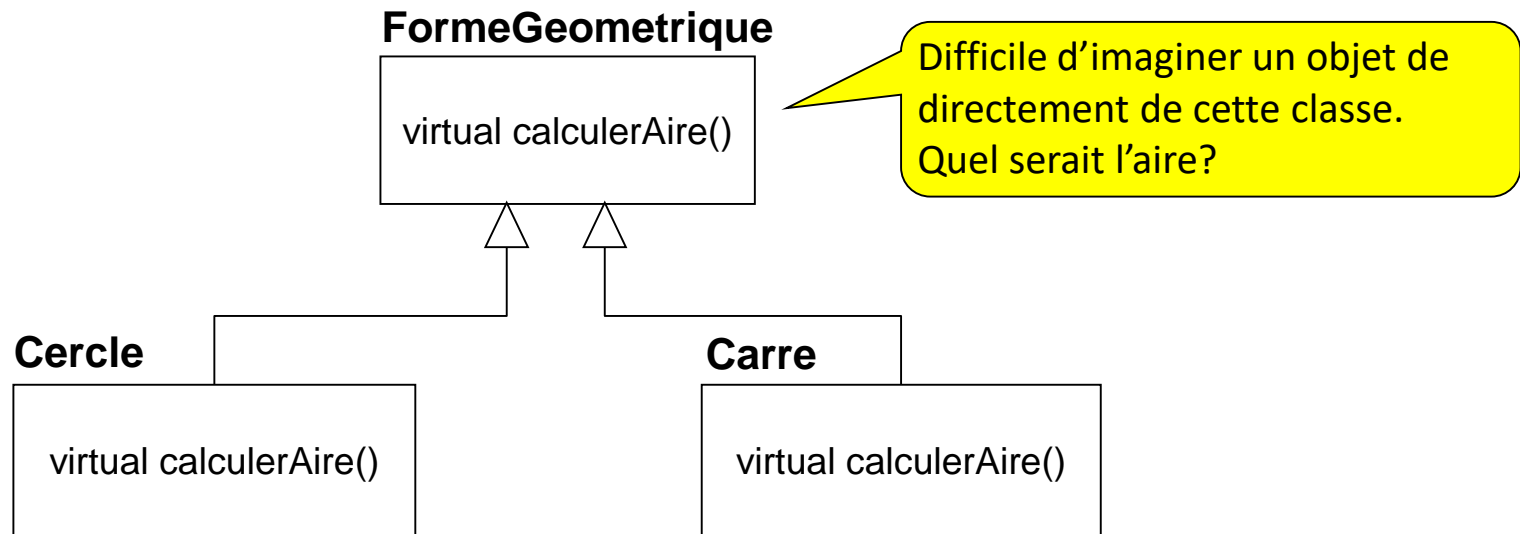
- Pourquoi?

- Copie dans des nouveaux objets de type Clock
 - Constructeur de copie de Clock
- Il « oublie » le reste qui n'entre pas dans Clock
- → ne peut pas contenir directement un objet de type dynamique différent du type statique
 - **Doit être pointeur** (intelligent) ou **référence**

Noter que **this** est un pointeur

Sinon pas de type dynamique différent, pas de liaison dynamique

Classes abstraites (rappel)



- Une classe abstraite **ne peut pas être instanciée**
- Sert à établir des méthodes à **définir** dans les classes dérivées, et parfois des attributs communs

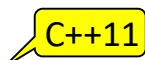
```

class FormeGeometrique {
public:
    virtual double calculerAire() = 0;
};
  
```

Est-ce que tout est correct?

« =0 » sans définition, dit méthode virtuelle pure. Rend la classe abstraite.

Classe abstraite pure

- Classe abstraite: contient *au moins une* méthode virtuelle pure
 - Soit elle la déclare (avec « =0 »)
 - Soit elle en hérite et ne lui donne pas de définition
- Aucun objet de cette classe ne peut être créé
- Classe abstraite pure: contient *uniquement* des méthodes virtuelles pures
 - Et un destructeur virtuel **=default** ou vide 
 - Aucun attribut
- Sert à définir une interface pour manipuler l'objet par polymorphisme (par référence/pointeur de ce type)

Destructeur ne peut pas être « =0 »

La déclaration de `FormeGeometrique` précédente répond-t-elle à ces critères?

Obtention du type dynamique

● Peut-on compter les cercles?

```
span<unique_ptr<FormeGeometrique>> formes = ... ;
int compte = 0;
for (auto&& forme : formes)
    if ( ...?... ) compte++;
```

- FormeGeometrique peut avoir une méthode virtuelle string obtenirType()
- Mais s'il sait quelle méthode appeler, il sait certainement le type?

● typeid(*forme) == typeid(Cercle) Dans <typeinfo>

- Possible mais ne vérifie pas pour les classes dérivées de Cercle qui « sont des » cercles

● Mieux: dynamic_cast<Cercle*>(forme.get())

 Vérifie l'héritage; voir prochain chapitre