

## PARTIE II

# Rappels de C++

Bruno Bachelet

Loïc Yon

- Caractéristiques générales
  - ❑ Historique
  - ❑ Héritage des autres langages
  
- POO en C++
  - ❑ Définition d'une classe
  - ❑ Cycle de vie des objets
  - ❑ Relations entre classes
  
- Autres concepts
  - ❑ Généricité
  - ❑ Exceptions
  - ❑ Surcharge d'opérateurs

# Caractéristiques générales

---

## ■ Origines

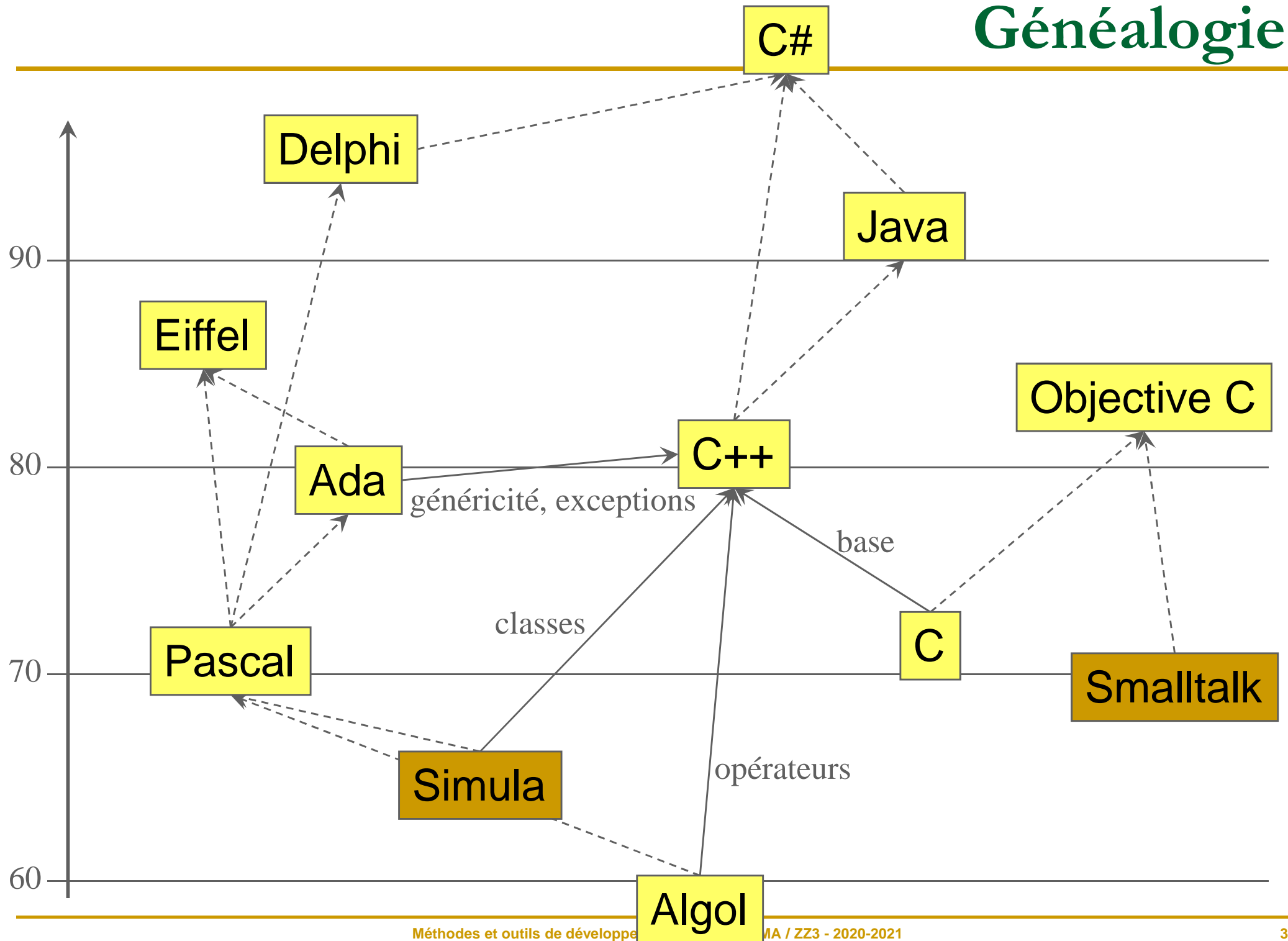
- ❑ Travaux de Bjarne Stroustrup (AT&T Bell)
- ❑ «*C with classes*» (80) → C++ (83)
- ❑ Normalisation en 98 (ISO/IEC 98-14882), norme C++ 98
- ❑ Depuis 2003, norme C++ 03
- ❑ A partir de 2011, cycle de 3 ans
- ❑ C++ 11, C++14, C++17...

## ■ Langage orienté objet (← SIMULA 67)

- ❑ Typage fort
- ❑ Maintien des types primitifs et des fonctions

## ■ Support de la généricité et des exceptions (← ADA 79)

## ■ Surcharge des opérateurs (← ALGOL 68)



- En C, paramètres uniquement passés par valeur
  - ❑ Passage en mode *in/out*  $\Rightarrow$  «passage par adresse»
    - Passe (par valeur) l'adresse de la variable
  - ❑ Conséquences
    - Code peu lisible, passage de pointeurs, source d'erreurs
  
- En C++, utilisation de références (&)
  - ❑ Référence = nouvel alias d'une variable
  - ❑ Utilisation identique à une variable
  - ❑ Passage par référence simulé par pointeur masqué
  - ❑ Sauf pour les méthodes *inline*

## ■ A la mode C

```
void swap (int * a, int * b)
{
    int c = *b;
    *b = *a;
    *a = c;
}

int main (int, char **)
{
    int i = 5, j = 6;
    swap (&i, &j);
    return 0;
}
```

## ■ A la mode C++

```
void swap (int & a, int & b)
{
    int c = b;
    b = a;
    a = c;
}

int main (int, char **)
{
    int i = 5, j = 6;
    swap (i, j);
    return 0;
}
```

## ■ Avantages

- ❑ Code plus lisible
- ❑ Appel plus simple
- ❑ Moins d'erreurs
- ❑ Efficace

## ■ Inconvénients

- ❑ Syntaxe ambiguë à cause de «&»
- ❑ Peu évident à comprendre au départ

## ■ Déclaration d'une référence

- ❑ Se déclare «comme» un pointeur
- ❑ Se comporte comme un alias sur l'objet
- ❑ Nécessite un objet référencé à la déclaration
- ❑ Ne peut changer d'objet par la suite

```
int i = 5;  
int & j = i;  
j = 4; // maintenant i=4 !
```

## ■ Référencer quoi ?

- ❑ Une référence non constante est toujours liée à une variable
- ❑ Une référence constante peut être liée à une constante
  - `const int & j = 4;`
- ❑ La référence nulle n'existe pas !



# Règles d'usage des types: *const*, référence ?

## Passage d'arguments

	Type primitif T	Classe C
Argument variable	<code>T &amp; arg</code>	<code>C &amp; arg</code>
Argument constant	<code>T arg</code> ou <code>const T &amp; arg</code>	<code>const C &amp; arg</code>

## Retour de variable

	Type primitif T	Classe C
Retour (mode lecture) d'un attribut	<code>T m(...) const;</code> ou <code>const T &amp; m(...) const;</code>	<code>const C &amp; m(...) const;</code>
Retour (mode lecture/écriture) d'un attribut	<code>T &amp; m(...);</code>	<code>C &amp; m(...);</code>
Retour d'un résultat produit par une méthode *	<code>T m(...) const;</code>	<code>C m(...) const;</code>

\* Retour d'une variable locale toujours par copie

# Allocation dynamique

---

- En C, couple `malloc` / `free`
- En C++, couple `new` / `delete`
- Pour allouer une donnée  
`int * iptr = new int;`  
...  
`delete iptr;`
- Pour allouer un tableau  
`int * iptr = new int[10];`  
...  
`delete[] iptr;`
- Réalisent aussi la construction / destruction
  - `new` = allocation mémoire + appel constructeur
  - `delete` = appel destructeur + libération mémoire
- Plus de `malloc` / `free` !

- En C, couple `printf` / `scanf` (et consorts)
- En C++, mécanisme de flux
  - ❑ Bibliothèque standard (namespace «`std`»)
  - ❑ Flux standards `std::cin`, `std::cout` et `std::cerr`
  - ❑ Inclusion de `<iostream>`
  - ❑ Pour lire depuis le flux en entrée

```
double x; int j;
std::cin >> x >> j;
```
  - ❑ Pour envoyer dans le flux en sortie

```
double x; int j;
std::cout << x << " + " << j << " = "
<< x + j << std::endl;
```
  - ❑ Pour les fichiers: `<fstream>`

- Les classes en C++
  - ❑ Déclaration / définition
  - ❑ Cycle de vie des objets
- Les relations entre classes
  - ❑ Agrégation
  - ❑ Héritage
  - ❑ Association
- La généricité
  - ❑ Fonctions
  - ❑ Classes
- Les exceptions
- Les opérateurs

# Déclaration d'une classe (1/2)

---

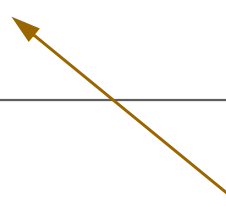
- Mot-clé «**class**» ou «**struct**»
- Contient les attributs et les prototypes des méthodes
  - **class**: membres privés par défaut
  - **struct**: membres publics par défaut
- Modificateurs d'accès
  - **public**: membre accessible par tous
    - Réservé exclusivement aux méthodes de l'interface
  - **private**: membre accessible aux méthodes de la classe
    - Pour les attributs
    - Pour les méthodes non destinées à l'utilisateur
  - **protected**: membre accessible aux méthodes de la classe et de ses sous-classes
    - Assouplit l'accès privé à des fins de redéfinition dans les sous-classes
- Modificateur «**static**»
  - Définit un membre de classe

# Déclaration d'une classe (2/2)

Point
- absc : Entier - ordo : Entier - <u>nb_points</u> : Entier
+ <u>créerPoint(:Entier,:Entier)</u> + x() : Entier + y() : Entier + move(dx:Entier,dy:Entier) + moveTo(x:Entier,y:Entier) + <u>nbPoints()</u> : Entier

```
class Point
{
    private:
        int absc;
        int ordo;
        static int nb_points;

    public:
        Point(int x, int y);
        int x(void) const;
        int y(void) const;
        void move(int,int);
        void moveTo(int,int);
        static int nbPoints(void);
};
```



Attention !

# Définition d'une classe

---

```
Point::Point(int x, int y) {  
    absc = x;  
    ordo = y;  
    nb_points++;  
}
```

```
int Point::x(void) const { return absc; }
```

```
void Point::move(int dx, int dy) {  
    absc += dx;  
    ordo += dy;  
}
```

```
int Point::nbPoints(void) { return nb_points; }
```

```
int Point::nb_points = 0; // Attribut de classe
```

- Appel méthode  $\Rightarrow$  coût d'exécution
- Parfois, dommage d'utiliser un appel de méthode
  - ❑ Pour récupérer la valeur d'un attribut
  - ❑ Pour un traitement simple
- Méthode «*inline*»: développée «comme» une macro
  - ❑ S'applique aussi aux fonctions
- On ne force pas une méthode à être «*inline*», on demande et le compilateur décide
- Avantage
  - ❑ Rapidité d'exécution (coût appel + optimisation supplémentaire)



## ■ Inconvénients

- ❑ Augmentation taille exécutable
  - A utiliser donc sur des méthodes courtes
- ❑ Implémentation avec la déclaration de la classe
  - Dans un fichier d'entête

## ■ Implémentation toujours dans l'entête

- ❑ Définition avec la déclaration  $\Rightarrow$  méthode souhaitée «*inline*»

```
class Point {  
    ...  
    int x(void) const { return absc; }  
    ...  
};
```

- ❑ Utilisation mot-clé «*inline*» (en dehors de la déclaration)

```
inline int Point::x(void) const { return absc; }
```

# Structure du code source

## ■ Fichier entête

- ❑ Déclaration de la classe
- ❑ Définition méthodes «*inline*»

```
#ifndef __CLASSE_HPP__
#define __CLASSE_HPP__

// includes
// déclarations anticipées

class Classe
{
    // attributs
    // proto méthodes
    // méthodes inline
};
#endif
```

## ■ Fichier implémentation

- ❑ Définition variables de classe
- ❑ Définitions méthodes

```
#include "classe.hpp"

// init. des variables
// de classe

// définition des méthodes
// externalisées
```

## ① Construction

- ❑ Réserveation mémoire
- ❑ Appel d'un constructeur

## ② Vie

- ❑ Appel des méthodes

## ③ Destruction

- ❑ Appel du destructeur
- ❑ Libération mémoire

- Rôle: initialiser les objets
- Syntaxe
  - ❑ Même nom que la classe
  - ❑ Pas de type de retour
  - ❑ Surcharge à volonté
  - ❑ Une particularité: la liste d'initialisation

- Exemples

```
Point::Point() {...}
```

```
Point::Point(int x, int y) {...}
```

```
Point::Point(const Point & p) {...}
```

# Liste d'initialisation (1/2)

---

## ■ Syntaxe

- ❑ *nom\_classe(...)* : *liste\_initialisation {...}*
- ❑ Liste = *nom\_attribut(valeur)* , *nom\_attribut(valeur)* ...
- ❑ Les valeurs peuvent être des expressions
  - Calcul, appel de fonction...

## ■ Rôle: initialisation des attributs d'un objet

- ❑ Même sans liste, initialisation avant le bloc de code

## ■ Construction de chacun des attributs

- ❑ Dans l'ordre de déclaration
- ❑ Donc, il faut lister les attributs dans l'ordre de déclaration
- ❑ Si un attribut est omis dans la liste  $\Rightarrow$  construction par défaut
- ❑ Les attributs de type référence obligatoirement dans la liste

# Liste d'initialisation (2/2)

## ■ Respecter l'ordre des attributs

```
class Rationnel
{
    private:
        int num;
        int den;

    public:
    Rationnel(int n=0, int d=1)
    : den(d), num(n)
    {}
};
```

## Solution

```
Rationnel::Rationnel(int n=0,
                    int d=1)
: num(n), den(d)
{ }
```

## ■ Initialisation plus complexe

- Ajout d'un attribut *distance*
- Distance du point à l'origine

avec code

```
Point::Point(int x, int y) :
    absc(x), ordo(y)
{
    dist = sqrt(x*x+y*y);
}
```

ou tout dans la liste

```
Point::Point(int x, int y) :
    absc(x), ordo(y),
    dist(sqrt(x*x+y*y))
{ }
```

- Trois types d'allocation (comme en C)
  - ❑ Statique: variable globale, attribut de classe, variable locale statique
  - ❑ Automatique: variable locale sur la pile
  - ❑ Dynamique: variable allouée sur le tas
    - `new` = allocation mémoire + appel constructeur
    - `delete` = appel destructeur + libération mémoire
  
- Gestion mémoire
  - ❑ Statique et automatique: par le système
  - ❑ Dynamique: par le développeur

- Moment de la construction
  - Variables globales et attributs de classe: avant l'exécution du «**main**»
  - Variables locales: à l'entrée dans le bloc
    - Variables locales statiques: à la 1<sup>ère</sup> entrée
  - Variables dynamiques: à l'exécution de «**new**»
  
- Moment de la destruction
  - Variables statiques: après la sortie du «**main**»
    - Même chose pour les variables locales statiques
  - Variables locales sur la pile: à la sortie du bloc
  - Variables dynamiques: à l'exécution de «**delete**»



# Méthodes constantes (1/3)

---

- Utilisation du mot-clé «**const**» en fin de prototype
- Indique les méthodes ne modifiant pas l'objet
  - Qui ne modifient pas les attributs
- Limité aux méthodes d'instance
- Avantages
  - Seules méthodes utilisables sur un objet constant
    - Une méthode «non constante» ne peut pas être exécutée
  - La méthode ne peut pas modifier les attributs
  - Contrôlé à la compilation
- Signification plus subtile
  - «**const**» fait partie de la signature
  - Possibilité de définir deux versions

# Méthodes constantes (2/3)

- Définition d'accesseurs (version 1 – recommandée)

```
class Exemple {  
    protected: string s;  
  
    public:  
        const string & getS(void) const { return s; }  
        void setS(const string & x) { s=x; }  
};
```

- Utilisation d'accesseurs

```
Exemple e1;
```

```
const Exemple e2;
```

```
e1.setS("nawouak"); ⇒ ok
```

```
e2.setS("nawouak"); ⇒ problème
```

```
std::cout << e1.getS() << std::endl; ⇒ ok
```

```
std::cout << e2.getS() << std::endl; ⇒ ok
```

# Méthodes constantes (3/3)

- Définition d'accesseurs (version 2 – non recommandée)

```
class Exemple {  
    protected: string s;  
  
    public:  
        const string & getS(void) const { return s; }  
        string & getS(void) { return s; }  
};
```

- Utilisation d'accesseurs

```
Exemple e1;  
const Exemple e2;
```

```
e1.getS() = "nawouak"; ⇒ ok
```

```
e2.getS() = "nawouak"; ⇒ problème
```

```
std::cout << e1.getS() << std::endl; ⇒ ok
```

```
std::cout << e2.getS() << std::endl; ⇒ ok
```

- Regrouper un ou plusieurs objets dans un autre = les attributs
- Trois manières d'agréger / trois types d'attributs
  - ❑ Attribut objet: construit en même temps que l'objet
  - ❑ Attribut référence: initialisation obligatoire dans le constructeur
    - Pas de changement par la suite
  - ❑ Attribut pointeur: peut être initialisé n'importe quand
    - Attention à la forme normale de Coplien
    - Si la mémoire de l'attribut est gérée par la classe
- Vie de l'objet agrégé
  - ❑ Objet construit par l'agrégeant
    - Attribut objet ou pointeur
  - ❑ Objet en provenance de l'extérieur
    - Recopie: attribut objet
    - Référence: attribut pointeur ou référence

- Une classe peut hériter d'une ou plusieurs autres
  - **class** *derivee* : *modificateur mere1, modificateur mere2...*
  - Modificateur  $\Rightarrow$  limitation de visibilité
- Visibilité de l'héritage : qui voit l'héritage ?
  - **public**  $\Rightarrow$  tout le monde
  - **protected**  $\Rightarrow$  classes filles uniquement
  - **private**  $\Rightarrow$  classe mère uniquement
  - Perte du lien de parenté  $\Rightarrow$  plus de conversion ascendante
- Visibilité des membres de la classe mère

Visibilité dans classe mère	Visibilité dans classe fille		
	Héritage « <b>public</b> »	Héritage « <b>protected</b> »	Héritage « <b>private</b> »
<b>public</b>	<b>public</b>	<b>protected</b>	<b>private</b>
<b>protected</b>	<b>protected</b>	<b>protected</b>	<b>private</b>
<b>private</b>	<b>private</b>	<b>private</b>	<b>private</b>

- Modificateur d'accès **protected**
  - Visible des classes fille mais pas de l'extérieur
- Utilisation classique de l'héritage
  - Attributs **protected** + héritage **public**
- Passer les attributs **private** en **protected** ?
  - Avantage: accessibles directement
  - Inconvénient: violation de l'encapsulation
    - Problèmes de maintenabilité si héritage en cascade
    - Solution: méthodes protégées pour l'accès aux attributs

- Héritage privé  $\Rightarrow$  perte de l'interface
- Utilisation 1: s'approprier l'implémentation
  - Mais l'héritage n'a pas forcément de sens
  - L'agrégation peut être utilisée à la place
  - A éviter donc dans ce but
- Utilisation 2: proposer une nouvelle interface
  - Modéliser un «*wrapper*»
  - Solution possible: l'agrégation
  - Héritage privé  $\Rightarrow$  solution sans agrégation

# Héritage et polymorphisme (1/2)

---

- Rendre une méthode polymorphe (virtuelle): **virtual**
  - ❑ Virtuelle un jour, virtuelle toujours !
    - Mot-clé «**virtual**» pas nécessaire dans les sous-classes
  - ❑ Peut être redéfinie dans les sous-classes
  
- Classe abstraite en C++
  - ❑ Pas de mot-clé
  - ❑ Classe abstraite  $\Rightarrow$  au moins une méthode abstraite
  - ❑ Méthode abstraite = méthode virtuelle pure
    - Pas de code
    - **virtual** *type\_retour nom\_méthode(arguments) = 0;*
  - ❑ Redéfinir impérativement dans les sous-classes
    - Car tant qu'une méthode est abstraite  $\Rightarrow$  pas d'instanciation



# Héritage et polymorphisme (2/2)

---

- Appeler l'implémentation de la classe mère
  - `classe_mère::nom_méthode(arguments)`
- Exemple : compléter l'implémentation de la classe mère

```
class Personne {  
    ...  
    virtual void afficher(void) const  
    { cout << nom << " " << prenom; }  
    ...  
};
```

```
class Etudiant : public Personne {  
    ...  
    void afficher(void) const {  
        Personne::afficher();  
        cout << " " << ecole;  
    }  
    ...  
};
```

# Héritage et constructeur

- Les constructeurs ne peuvent pas être virtuels
  - Pas d'héritage des constructeurs
  - Mais séquence de construction prédéfinie
- Exemple: B hérite de A
  - Construction B = Construction A, puis construction attributs de B
  - ```
class A {  
    protected: string s;  
  
    public:  
    A() { s=...; } ⇔ A() : s() { s=...; }  
    A(const string & ss) : s(ss) {}  
};  
  
class B : public A {  
    protected: string t;  
  
    public:  
    B() { s=...; t=...; } ⇔ B() : A(),t() { s=...; t=...; }  
  
    B(const string & ss, const string & tt)  
    : A(ss),t(tt) {}  
};
```

- Méthode virtuelle  $\Rightarrow$  destructeur virtuel

- Destruction impérativement polymorphe
- Exemple

```
vector<Point *> v;
```

```
...
```

```
for (int i=0; i<v.size(); ++i) delete v[i];
```

- Si destructeur polymorphe

- Appel destructeur sous-classe
- Puis appel destructeur super-classe

- Si destructeur non-polymorphe

- Appel destructeur super-classe  $\Rightarrow$  incohérent !

# Héritage virtuel (1/2)

- Héritage en diamant

- Duplication des attributs

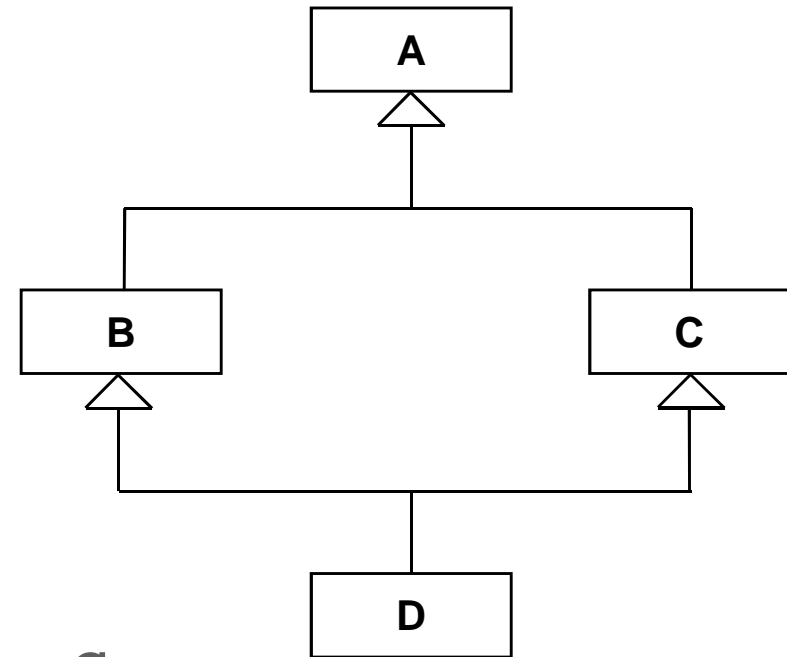
- ❑ `class A`  
`{ A(...) {} };`

- `class B : public A`  
`{ B(...) : A(...) {} };`

- `class C : public A`  
`{ C(...) : A(...) {} };`

- `class D : public B, public C`  
`{ D(...) : B(...), C(...) {} };`

- ❑ 2 appels au constructeur de A dans D  
⇒ attributs de A dupliqués dans D



- Collision des noms de méthode (ou attribut)

- ❑ Exemple: méthode `A::x()`

- ❑ `D::x()` signifie appel sur l'objet A issu de B ou de C ?

- Distinction possible via `B::x()` ou `C::x()` ou conversion vers `B &` ou `C &`

# Héritage virtuel (2/2)

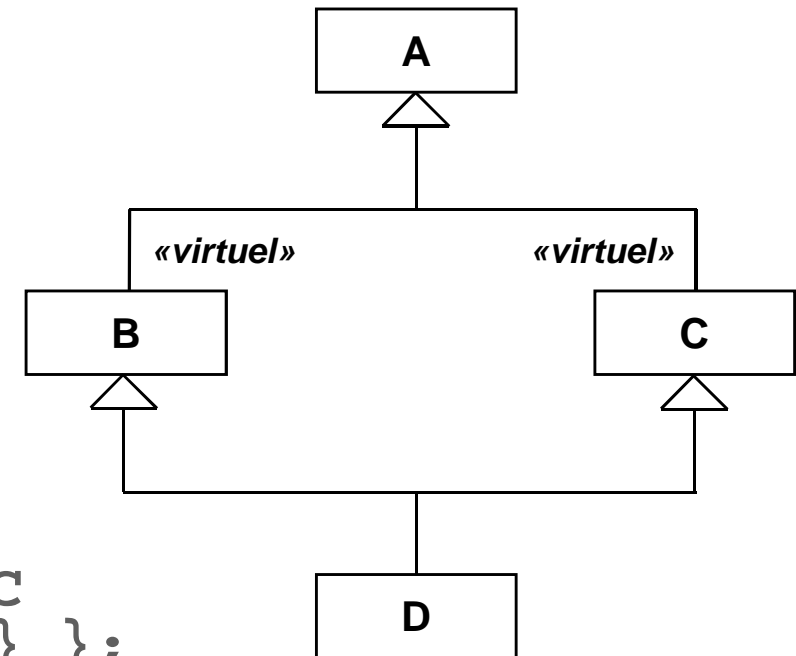
## ■ Solution: héritage «virtuel»

```
class A  
{ A(...) {} };
```

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

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

```
class D : public B, public C  
{ D(...) : A(...), B(...), C(...) {} };
```



- ❑ Une seule copie de A
- ❑ Appel explicite au constructeur de A dans D
- ❑ Paramètres destinés à A ignorés dans les constructeurs de B et C

## ■ Autres solutions: héritage d'interfaces (ou classes abstraites en C++) ou délégation

# Surcharge opérateurs (1/3)

- Constructeurs
  - Constructeur par défaut
    - `A(void);`
  - Constructeur de copie
    - `A(const A &);`
- Affectation (méthode)
  - `A & operator = (const A & x) {  
... // Recopie de x dans «this»  
return (*this);  
}`
  - Retour de l'objet pour chaînage: `a = b = c;`
- Opérations arithmétiques / logiques binaires (fonctions)
  - `A operator + (const A & x, const A & y) {  
A resultat;  
... // Calcul de x+y  
return resultat;  
}`
  - `bool operator == (const A & x, const A & y);`

## ■ Opérations arithmétiques unaires (méthodes)

### □ Préfixé

- `A & operator ++ () {  
... // Incrémentation de «this»  
return *this;  
}`

- Retour de l'objet: `a = ++b;`

### □ Postfixé

- `A operator ++ (int) {  
A copie = *this;  
... // Incrémentation de «this»  
return copie;  
}`

- Retour d'une copie avant incrément: `a = b++;`

# Surcharge opérateurs (3/3)

## ■ Opérateurs de flux (fonctions)

### □ Ecriture

- `ostream & operator << (ostream & flux, const A & x)`  
{  
  ... // Ecriture de x dans le flux  
  return flux;  
}

- Retour du flux: `f << a << b;`

### □ Lecture

- `istream & operator >> (istream & flux, A & x)` {  
  ... // Lecture du flux dans x  
  return flux;  
}

- Ne jamais passer un flux par copie !

## ■ Autres symboles

- `()`, `[]`, `*`, `,` ...