

PARTIE II

Rappels de C++

Christophe Duhamel

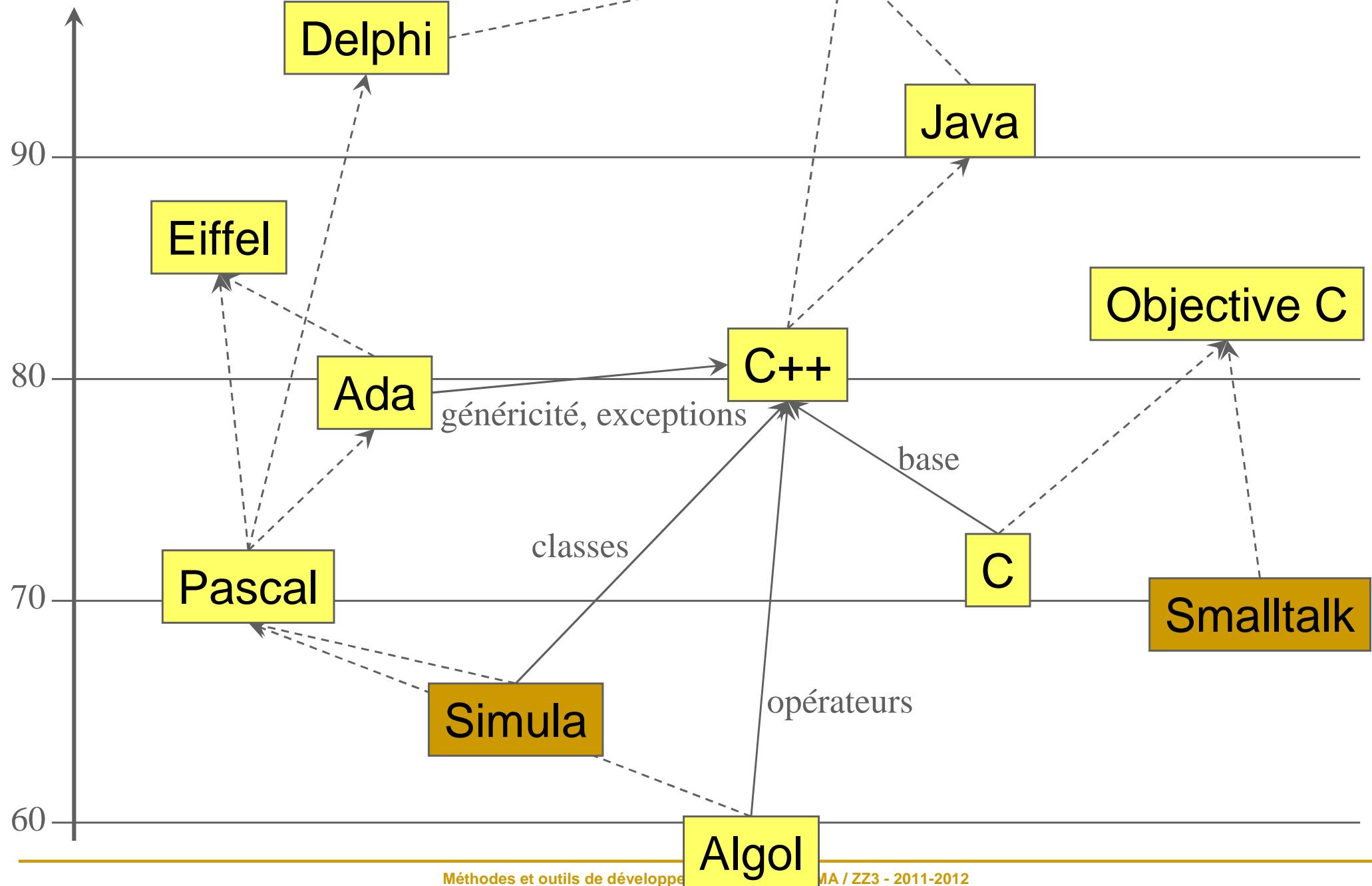
Bruno Bachelet

Luc Touraille

- 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

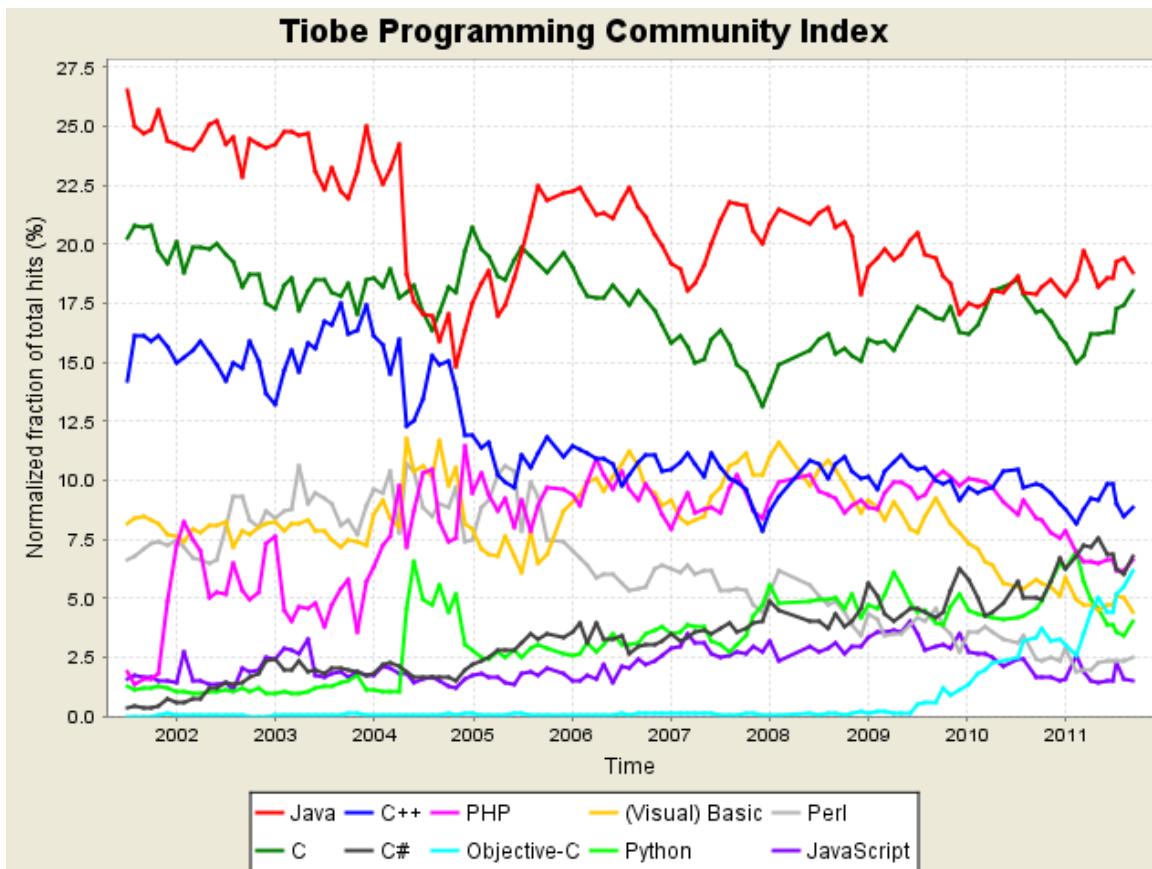
Caractéristiques générales

- Origines
 - Travaux de Bjarne Stroustrup (AT&T Bell)
 - «C *with classes*» (80) → C++ (83)
 - Normalisation en 98 (ISO/IEC 98-14882)
 - Depuis 2003, norme C++ 03
 - Mi-2011, nouvelle norme C++ 1x
- 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)



«Popularité» des langages

- Tiobe Index (Septembre 2011) – www.tiobe.com
 - Basé sur résultats moteurs de recherche ⇒ discutable
 - Mais fournit tout de même une tendance



Programming Language	Position Sep 2011	Position Sep 2006	Position Sep 1996	Position Sep 1986
<u>Java</u>	1	1	5	-
<u>C</u>	2	2	1	1
<u>C++</u>	3	3	2	5
<u>C#</u>	4	8	-	-
<u>PHP</u>	5	5	-	-
<u>Objective-C</u>	6	39	-	-
<u>(Visual) Basic</u>	7	4	3	6
<u>Python</u>	8	7	23	-
<u>Perl</u>	9	6	6	-
<u>JavaScript</u>	10	9	21	-
<u>Lisp</u>	14	14	13	3
<u>Ada</u>	18	19	10	2

- En C, paramètres uniquement passés par valeur
 - Passage en mode *in/out* ⇒ «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
 - Pointeur masqué, simulant le passage par référence
 - Pour les méthodes *inline*, vrai passage par référence

■ 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 est toujours liée à une variable
 - Elle ne peut pas être liée à une constante
 - 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 & arg</code>	<code>C & arg</code>
Argument constant	<code>T arg</code> ou <code>const T & arg</code>	<code>const C & 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 & m(...) const;</code>	<code>const C & m(...) const;</code>
Retour (mode lecture/écriture) d'un attribut	<code>T & m(...);</code>	<code>C & m(...);</code>
Retour d'un résultat produit par une méthode	<code>T m(...) const;</code>	<code>C m(...) const;</code>

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 standard **std::cin**, **std::cout** et **std::cerr**
 - Inclusion de **<iostream>**
 - Pour lire depuis le flux en entrée

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

```
double x;int j;
cout << x << " + " << j << " = " << x + j << 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**»
- Contient les attributs et les prototypes des méthodes
- 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 surcharge dans les sous-classes
- Modificateur «**static**»
 - Définit un membre de classe

Déclaration d'une classe (2/2)

Point
-absc_ : entier
-ordo_ : entier
-NbPoints : entier
+x() : entier
+y() : entier
+move(incX : entier, incY : entier)
+moveTo(X : entier, Y : entier)
<u>+NbPoints() : entier</u>

```
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);

}; i
```

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 incX, int incY) {
    absc += incX;
    ordo += incY;
}

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

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

- Appel méthode ⇒ 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
- Avantage
 - Rapidité d'exécution (coût appel + optimisation supplémentaire)

■ Inconvénients

- Augmentation taille exécutable
 - A utiliser donc sur des méthodes courtes
- Implémentation dans la partie déclaration de la classe
 - Ou dans un fichier d'entête

■ Implémentation

- Définition avec la déclaration

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

- Utilisation du mot-clé «*inline*» (indication au compilateur)

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

Structure du code source

- Fichier entête
 - Déclaration de la classe
 - Définition méthodes «*inline*»
- Fichier implémentation
 - Définition variables de classe
 - Définitions méthodes

```
#ifndef __CLASSE_H__
#define __CLASSE_H__


// includes
// forward déclaration

class Classe
{
    // attributs
    // proto méthodes
    // méthodes inline
};

#endif
```

```
#include "classe.h"


// init. des variables
// de classe


// définition des méthodes
// externalisées
```

① Construction

- ❑ Réservation 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 (statique) 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 ⇒ 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)
        {}

};
```

- 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);
}
```

Solution

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

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, 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 programmeur

- Moment de la construction
 - Variables globales: avant l'exécution du «**main**»
 - Variables locales: à l'entrée dans le bloc
 - Variables locales statiques: à la 1^{ère} 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
 - Utilisable 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

- Dériver une classe d'une ou plusieurs autres
- Syntaxe: `class derivee : modificateur mere1, modificateur mere2...`
- Modificateur conditionne la visibilité des membres de la classe mère

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

- 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 ⇒ perte de l'interface
- Raison 1: s'approprier l'implémentation
 - Héritage bizarre
 - L'agrégation peut être utilisée à la place
 - A éviter donc
- Raison 2: proposer une nouvelle interface
 - Modéliser un «wrapper» ⇒ agrégation
 - Héritage privé ⇒ solution sans agrégation
 - Conservation du type original

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 ⇒ 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 ⇒ pas d'instanciation

Héritage et polymorphisme (2/2)

- Hériter l'implémentation de la classe mère
 - `classe_mère::nom_méthode(arguments)`

- Exemple

```
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(string ss) { s=ss; }  
};  
  
class B : public A {  
protected: string t;  
  
public:  
    B() { s=...; t=...; } ⇔ B() : A(),t() { s=...; t=...; }  
    B(string ss,string tt) : A(ss),t(tt) {}  
};
```

- Méthode virtuelle ⇒ 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 ⇒ incohérent !

Héritage virtuel (1/2)

- Héritage en diamant
- Duplication des attributs

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

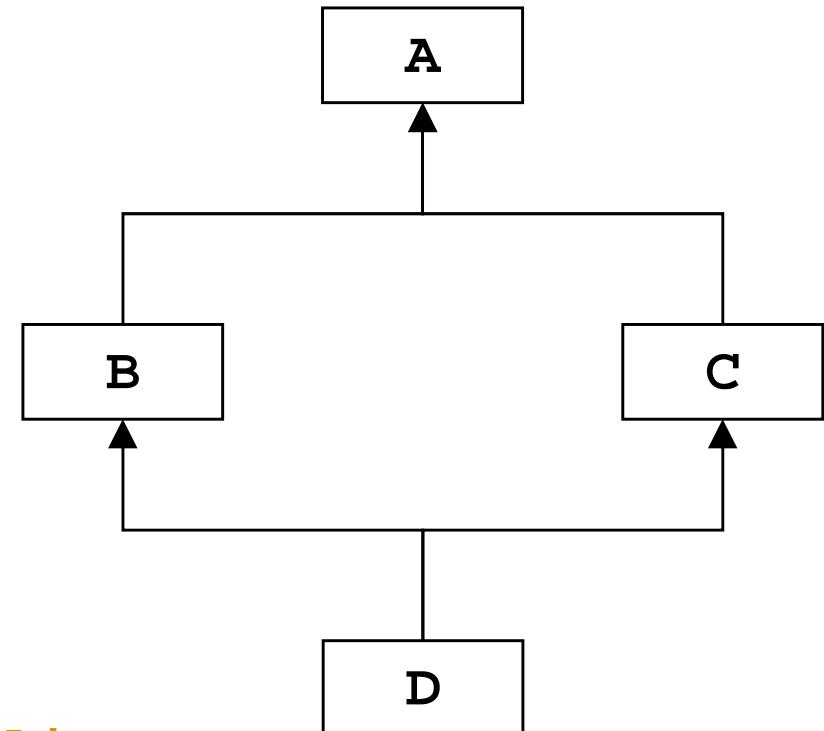
- `class B : public A
{ B(...) : A(...) {} };`

- `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 ?



Héritage virtuel (2/2)

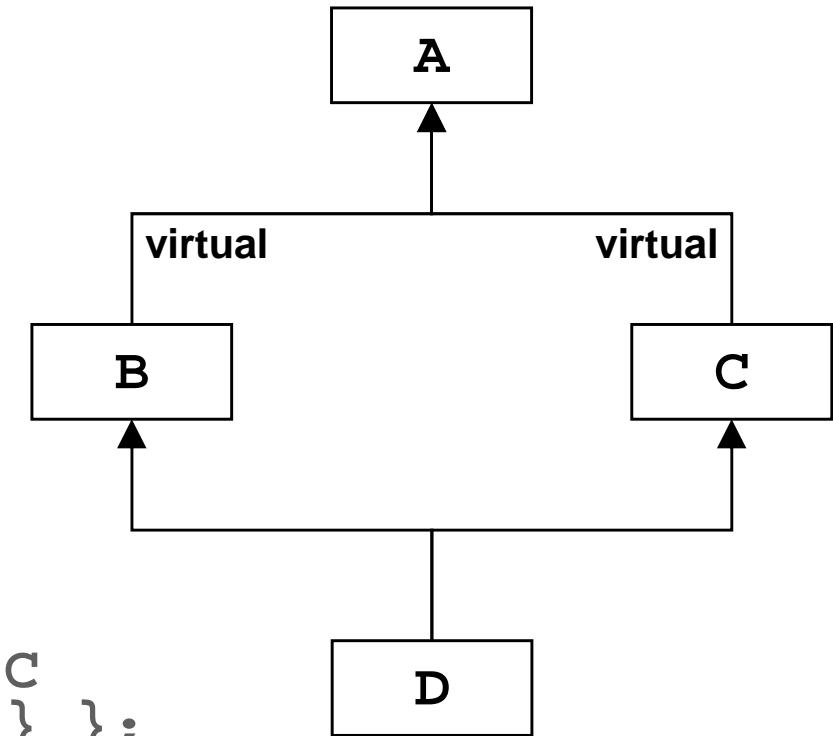
■ Solution: héritage «virtuel»

- ```
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 délégation

# Surcharge opérateurs (1/3)

---

- Constructeurs
  - Constructeur par défaut
    - `A(void);`
  - Constructeur de copie
    - `A(const A &);`
- Affectation
  - `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
  - `A operator + (const A & x, const A & y) {  
 A resultat;  
 ... // Calcul de x+y  
 return resultat;  
}`
  - `bool operator == (const A & x, const A & y);`

# Surcharge opérateurs (2/3)

---

## ■ Opérations arithmétiques unaires

### □ 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

### □ 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

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

# Opérateurs de conversion

---

- Par un constructeur
  - `class A { ... A(const B &); ... };`
  - Attention: conversion automatique !
    - `void f(const A &);`
    - `B b;`
    - `f(b);` ⇒ Construction d'un objet intermédiaire A
  - Possibilité de forcer l'explicitation
    - `explicit A(const B &);`
    - `f(b);` ⇒ erreur de compilation
    - `f(A(b));` ⇒ obligation d'expliciter la conversion

- Opérateur de conversion (⇒ conversion implicite)

```
class B {
 ...
public: operator A(void) {
 A a;
 ... // Conversion de «this» dans a
 return a;
}
};
```