ISIMA

PARTIE II

Rappels de C++

Bruno Bachelet

Loïc Yon

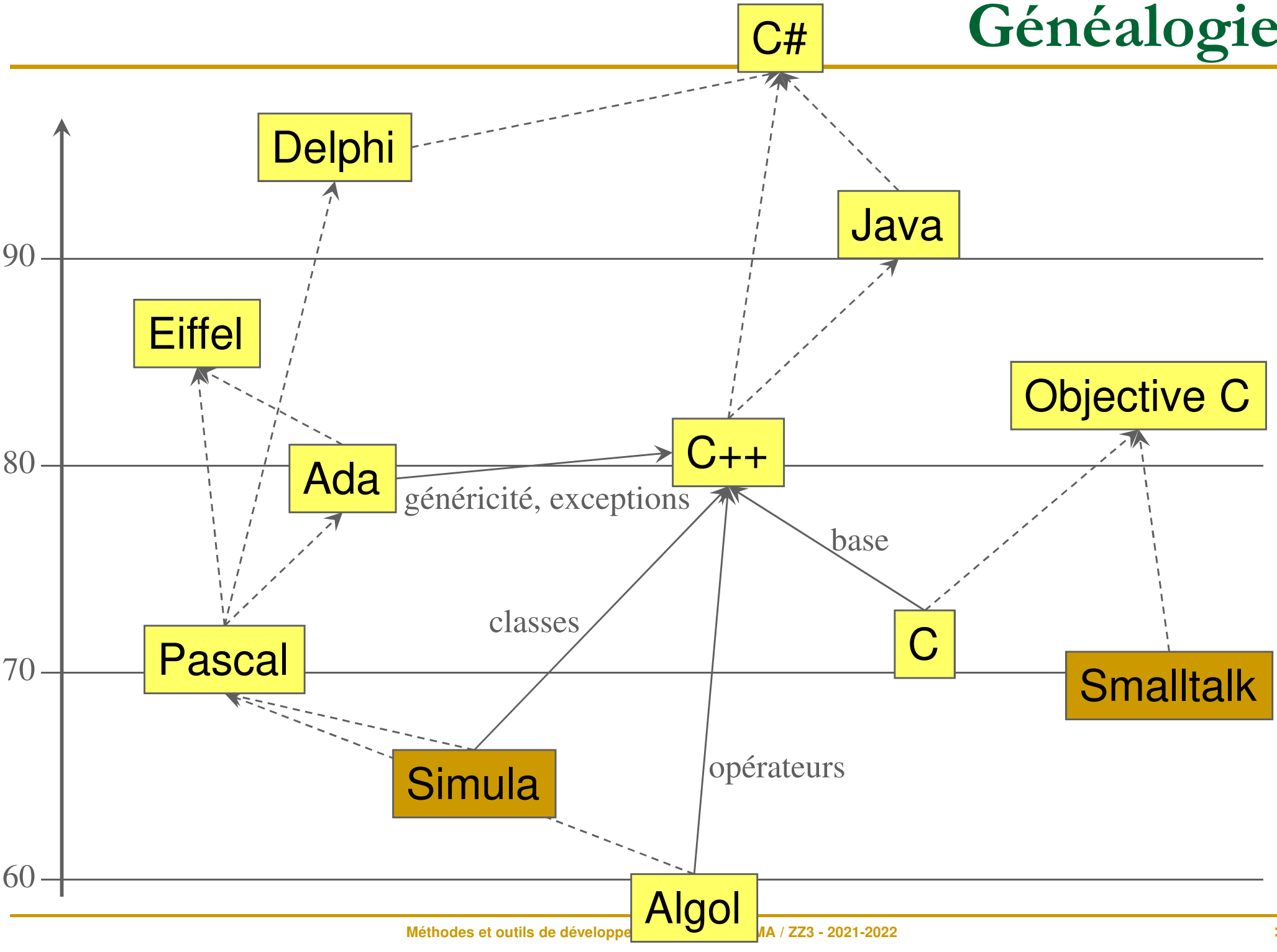
Langage C++

- 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, C++20…
- 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)

Généalogie


C#
Delphi
Java
90
Eiffel
Objective C
80
Ada
C++
généricité, exceptions
base
classes
C
Pascal
70
Smalltalk
Simula
opérateurs
60
Algol

Type référence *(1/4)*

- 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
  - Passage par référence simulé par pointeur masqué
  - Sauf pour les méthodes *inline*

Type référence *(2/4)*

- A la mode C

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

int main() {
 int i = 5;
 int j = 6;

 swap(&i,&j);
}
```

- A la mode C++

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

int main() {
 int i = 5;
 int j = 6;

 swap(i,j);
}
```

Type référence *(3/4)*

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

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

Type référence *(4/4)*

- 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	`T & arg`	`C & arg`

Argument constant	`T arg`
	ou
	`const T & arg`
		`const C & arg`

Retour de variable

	Type primitif T	Classe C

Retour (mode lecture) d'un attribut	`T m(…) const;`
	ou
	`const T & m(…) const;`
		`const C & m(…) const;`

Retour (mode lecture/écriture) d'un attribut	`T & m(…);`	`C & m(…);`
Retour d'un résultat produit par une méthode *	`T m(…) const;`	`C m(…) const;`

* Retour d'une variable locale toujours par copie

Allocation dynamique

- En C++, utilisation de «**new**» et «**delete**»
  - Plus d'appel explicite à «**malloc**» et «**free**»

- Pour allouer / libérer une donnée

```
int * iptr = new int;
…
delete iptr;
```

- Pour allouer / libérer un tableau

```
int * iptr = new int[10];
…
delete[] iptr;
```

- Réalisent aussi la construction / destruction
  - **new** = allocation mémoire + appel constructeur
  - **delete** = appel destructeur + libération mémoire

- Mot-clé «**nullptr**» pour le pointeur nul (depuis C++11)

Entrées / sorties

- En C: **`printf`** / **`scanf`** (et consorts)

- En C++: mécanisme de flux
  - Bibliothèque standard ⇒ *namespace* «**`std`**»
  - Flux standards: **`std::cin`** / **`std::cout`** / **`std::cerr`**
  - Inclusion de **`<iostream>`** (ou **`<fstream>`** pour les fichiers)

- Pour lire depuis un flux (classe de base «**`istream`**»)

```
double x; int j;
…
flux >> x >> j;
```

- Pour écrire dans un flux (classe de base «**`ostream`**»)

```
double x; int j;
…
flux << x << " + " << j << " = " << (x + j) << std::endl;
```

La programmation objet en C++

- 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)*

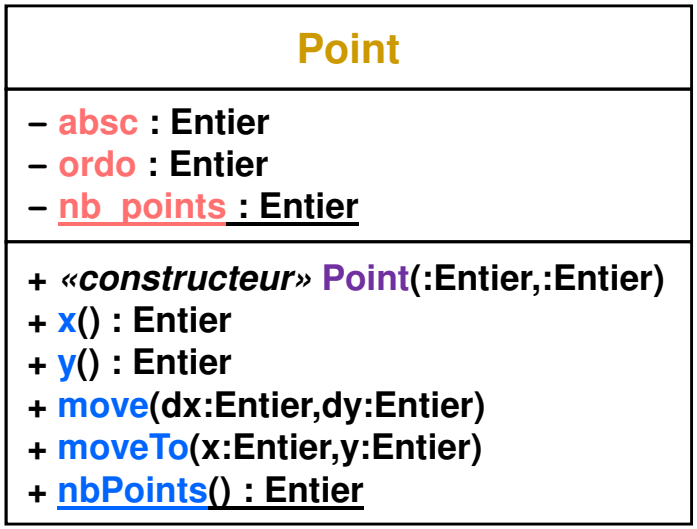
- Mot-clé «`class`» ou «`struct`»

- 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)*




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

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

};
```

Attention !

Définition d'une classe

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

int Point::x() const { return absc; } // Accesseur

void Point::move(int dx, int dy) { // Méthode d'instance
 absc += dx;
 ordo += dy;
}

int Point::nbPoints() // Méthode de classe
{ return nb_points; }

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

Méthodes «*inline*» (1/2)

- 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
- On ne force pas une méthode à être «inline»,
  on demande et le compilateur décide
- Avantage
  - Rapidité d'exécution (coût appel + optimisation supplémentaire)

Méthodes «*inline*» (2/2)

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

- Implémentation <u>toujours dans l'entête</u>
  - Définition avec la déclaration ⇒ méthode souhaitée «*inline*»

```
class Point {
 …
 int x() const { return absc; } // "inline" implicite
 …
};
```

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

```
inline int Point::x() 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
  // Prototypes méthodes
  // Méthodes inline
 };
#endif
```

- Fichier implémentation
  - Définition variables de classe
  - Définitions méthodes

```
#include "classe.hpp"

// Initialisation des
// variables de classe

// Définition des méthodes
// externalisées
```

Cycle de vie des objets

1. Construction
   - Réservation mémoire
   - Appel d'un constructeur
2. Vie
   - Appel des méthodes
3. Destruction
   - Appel du destructeur
   - Libération mémoire

Constructeurs

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

- Initialisation plus complexe
  - Ajout attribut «`dist`» dans «`Point`»
  - Distance du point à l'origine

- Dans le corps du constructeur

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

- Ou dans la liste d'initialisation

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

Appel au constructeur

- Implicite
  - `A a;` ⇒ constructeur par défaut «`A()`»
  - `A a = 4;` ⇒ constructeur «`A(int)`»
- Explicite
  - `A a(4);` ⇒ constructeur «`A(int)`»
  - `f(A(4));` ⇒ constructeur «`A(int)`» (objet créé à la volée)
  - `A * a = new A(4);` ⇒ constructeur «`A(int)`»
- Attention à la syntaxe d'une construction explicite par défaut
  - `A a();` ⇒ déclaration fonction «`a()`»
  - Pas de souci avec «`new`» et «`throw`»
- Cas des types primitifs
  - Paramètre *template* «`T`»: `T x = T();`
  - Si `T` = `A` ⇒ appel constructeur par défaut «`A()`»
  - Si `T` = `int` ⇒ initialise «`x`» à zéro
  - Rappel: `int x` ⇒ aucune garantie que «`x`» vaut 0

Construction par liste de valeurs *(1/2)*

- C++03: possibilité d'initialiser des «agrégats» par liste
  - Agrégat = tableau ou classe avec restrictions
    - `int t[] = {7,8,9};`
  - Classe «agrégat» = attributs publics, pas de constructeur
    - `class Paire { public: int x; double y; };`
    - `Paire p = {3,7.0};`

- C++11: généralisation à n'importe quelle classe
  - ```
    class Paire {
     private: int x; double y;
     public: Paire(int a, double b) : x(a), y(b) {}
    };
    ```
  - `Paire p = {3,7.0};` ⇒ appel constructeur
    ```

Construction par liste de valeurs *(2/2)*

- Autres syntaxes possibles
  - `Paire p{3,7.0};`
  - `return {3,7.0};` ⇒ déduction du type à l'aide de la signature
- Permet d'éviter certaines ambiguïtés de syntaxe
  - `Paire p();` ⇒ interprété comme une fonction
  - `Paire p{};` ⇒ interprété comme une variable
- Objectif: uniformiser l'initialisation des variables
- Mais problème: des subtilités subsistent !
  - `std::vector v1(10,20);` ⇒ vecteur de 10 éléments
  - `std::vector v2{10,20};` ⇒ vecteur de 2 éléments
  - La raison: l'initialisation par liste (*initializer list*)

Parenthèses ou accolades ?

- Conseils
  - Utiliser « `{}` » quand la volonté est d'initialiser par une liste de valeurs
    - `std::vector v{10,20}` ⇒ vecteur contient les valeurs 10 et 20
  - Utiliser « `()` » quand la volonté est d'initialiser par une «fonction»
    - `std::vector v(10,20)` ⇒ vecteur contient 10 fois la valeur 20
  - Utiliser « `{}` » pour la construction par défaut

```
class A {
 private: std::vector v;
 public: A() : v{} {};
};
```

  - Pour éviter des complications, privilégier l'usage de « `()` » avec…
    - Le mot-clé «`auto`» pour la déduction automatique de type
    - Un paramètre *template*

Création d'objets *(1/2)*

- 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

Création d'objets *(2/2)*

- 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 1ère 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 {
 private:
  string s;

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

- Utilisation d'accesseurs

```
Exemple e1;
const Exemple e2;

e1.setS("nawouak");
e2.setS("nawouak");

std::cout << e1.getS() << std::endl;
std::cout << e2.getS() << std::endl;
```

$\Rightarrow$ ok

$\Rightarrow$ problème

$\Rightarrow$ ok

$\Rightarrow$ ok

Méthodes constantes *(3/3)*

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

```
class Exemple {
 private:
  string s;

 public:
  const string & getS() const { return s; }
  string & getS() { 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
```

Agrégation

- 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

Héritage *(1/2)*

- Une classe peut hériter d'une ou plusieurs autres
  - `class` *derivee* `:` *modificateur mere1*`,` *modificateur mere2*…
  - Modificateur ⇒ limitation de visibilité

- Visibilité de l'héritage : qui voit l'héritage ?
  - `public` ⇒ tout le monde
  - `protected` ⇒ classes filles uniquement
  - `private` ⇒ classe uniquement
  - Perte du lien de parenté ⇒ plus de conversion ascendante

- Visibilité des membres de la classe mère

Visibilité dans classe mère	Visibilité dans classe fille		
	Héritage «`public`»	Héritage «`protected`»	Héritage «`private`»
`public`	`public`	`protected`	`private`
`protected`	`protected`	`protected`	`private`
`private`	`private`	`private`	`private`

Héritage *(2/2)*

- Rappel sur le modificateur «`protected`»
  - Permet de rendre un membre visible par les classes filles
  - Tout en restant caché vis-à-vis de l'extérieur

- Utilisation classique de l'héritage
  - Attributs «`private`» ⇒ encapsulation
  - Et héritage «`public`»

- Passer les attributs «`private`» en «`protected`» ?
  - Avantage: accessibles directement par les classes filles
  - Inconvénient: violation partielle de l'encapsulation
    - Problèmes de maintenabilité si héritage en cascade
    - Solution: méthodes protégées pour l'accès aux attributs privés

Héritage privé ?

- Héritage privé ⇒ 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 ⇒ 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 ⇒ 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)*

- Contrôle de la redéfinition d'une méthode (depuis C++11)
  - Lors de la redéfinition d'une méthode, une erreur est vite arrivée
  - Exemple
    - `class A { public: virtual void f(int); };`
    - `class B : public A { public: void f(double); };`
    - Compile, mais «`B::f`» ne redéfinit pas «`A::f`»
  - Mot-clé «`override`» ⇒ intention de redéfinition
    - ```
      class B : public A
      { public: void f(double) override; };
      ```
    - Contrôle à la compilation ⇒ erreur
  - Mot-clé «`final`» ⇒ pas de redéfinition
    - `void f(void) final` ⇒ pas de redéfinition possible de «`f`»
    - `class A final` ⇒ héritage de «`A`» impossible
      ```

Redéfinition «par complément»

- Appel à une méthode de la classe mère (sans polymorphisme)
  - *classe_mère*`::`*nom_méthode*`(`*arguments*`)`

- Exemple: compléter l'implémentation de la classe mère

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

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

Héritage et constructeur

- Les constructeurs ne peuvent pas être virtuels
  - Pas d'héritage (au sens classique) des constructeurs
  - Mais séquence de construction prédéfinie

- Exemple: B hérite de A
  - Construction B ⇒ construction partie A, puis construction attributs de B
  - ```
    class A {
     private: string s;

     public:
      A() { s="X"; } ⇔ A() : s() { s="X"; }
      A(const string & ss) : s(ss) {}
    };

    class B : public A {
     private: string t;

     public:
      B() { s="Y"; t="Z"; } ⇔ B() : A(), t() { s="Y"; t="Z"; }

      B(const string & ss, const string & tt)
      : A(ss), t(tt) {}
    };
    ```
    ```

Héritage et destructeur

- Méthode virtuelle ⇒ destructeur virtuel
  - Destruction impérativement polymorphe
- Exemple

```
vector<ObjetGraphique *> v;
…
for (int i=0; i<v.size(); ++i) delete v[i];
```

- Si destructeur polymorphe
  - Appel destructeur sous-classe (e.g. `~Rectangle`)
  - Puis appel destructeur super-classe (`~ObjetGraphique`)
- Si destructeur non-polymorphe
  - Appel destructeur super-classe ⇒ incohérent !
- Important si ressource allouée dans le constructeur
  - Elle doit être libérée par le destructeur
  - Destructeur non-polymorphe ⇒ pas de libération

Héritage virtuel *(1/2)*

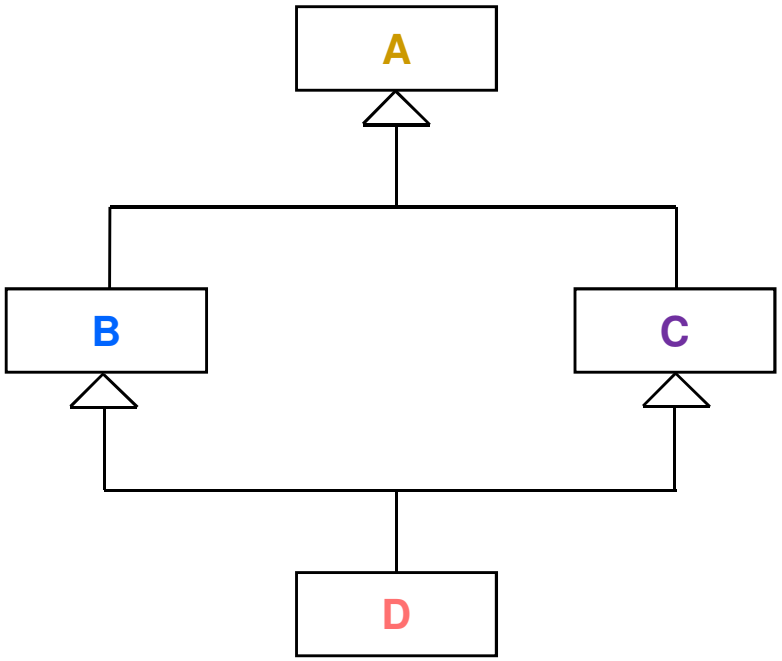
- Duplication des attributs
  - ```
    class A
    { A(…) {} };

    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

Héritage en diamant



- Collision des noms de membres hérités
  - Si classe A définie la méthode `x()`
  - `D::x()` signifie appel sur la partie A issue de l'héritage avec B ou C ?
  - Distinction possible via `B::x()` ou `C::x()` ou conversion vers `B&` ou `C&`
    ```

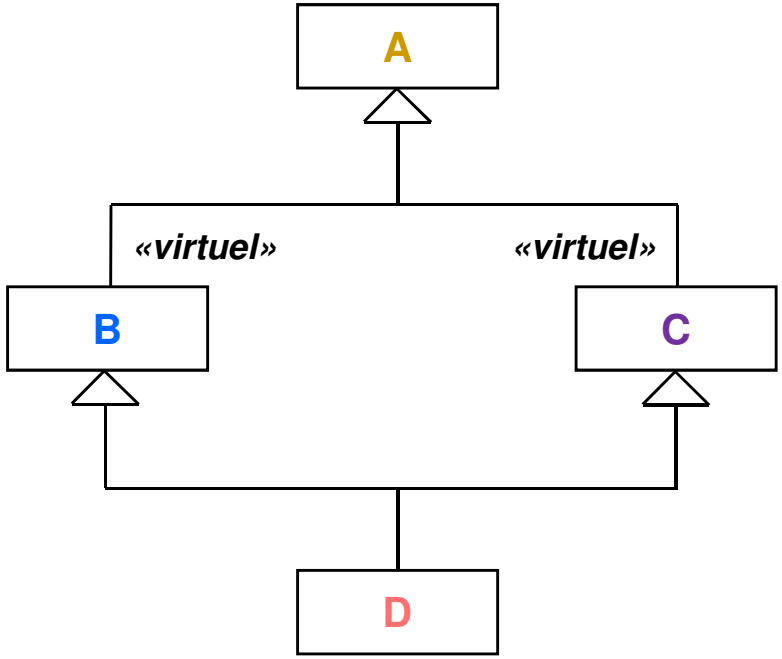
Héritage virtuel *(2/2)*

- 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
  - Dans les constructeurs de B et C, les arguments destinés à A sont ignorés

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

Réutilisation de constructeurs *(1/2)*

- Définition d'un constructeur à partir d'un autre (depuis C++11)
  ⇒ évite la duplication de code

```
class Personne {
 private:
  string nom;
  string prenom;

 public:
  Personne(const string & n,
           const string & p) : nom(n), prenom(p) {}

  Personne() : Personne("Doe","John") {}
};
```

Réutilisation de constructeurs *(2/2)*

- «Héritage» des constructeurs de la classe mère (depuis C++11)

```
class Etudiant : public Personne {
 using Personne::Personne;

 private:
  string ecole;
 …
};
```

- Tous les constructeurs sont «hérités»
  - Chaque constructeur est «recréé» dans la classe fille
    - Avec la même signature que la classe mère
    - Appelle le constructeur de la classe mère
    - Et construit par défaut les attributs de la classe fille
  - Exemple
    - `Etudiant e("Doe","Jane");`
  - Appelle constructeur équivalent à
    - `Etudiant(const string & n, const string & p)`
      `: Personne(n,p), ecole() {}`

Choix explicite des opérateurs par défaut

- Par défaut, une classe possède
  - Un constructeur par défaut
    - Si aucun autre constructeur n'est manuellement défini
  - Un constructeur de copie
  - Un opérateur d'affectation

- C++03: pour empêcher l'utilisation ⇒ déclarer privé

- C++11: choix explicite ⇒ «`default`» ou «`delete`»

- Exemple

```
class NonCopiable {
 NonCopiable(void) = default;
 NonCopiable(const NonCopiable &) = delete;
 NonCopiable & operator = (const NonCopiable &) = delete;
};
```

Surcharge opérateurs *(1/4)*

- Constructeurs
  - Constructeur par défaut
    - `A(void);`
  - Constructeur de copie
    - `A(const A &);`

- Affectation ⇒ méthode
  - `a = b` ⇔ `a.operator=(b)`
  - ```
    A & operator=(const A & x) {
     … // Recopie de x dans "this"
     return *this;
    }
    ```
  - Retour de l'objet pour le chaînage
    `a = b = c` ⇔ `a.operator=(b.operator=(c))`
    ```

Surcharge opérateurs *(2/4)*

- Opérations arithmétiques binaires ⇒ fonctions
  - `c = a + b` ⇔ `c = operator+(a,b)`
  - ```
    A operator+(const A & x, const A & y) {
     A resultat;
     … // Calcul x + y
     return resultat;
    }
    ```
  - Retour par copie car objet local

- Opérations logiques binaires ⇒ fonctions
  - `if (a < b)` ⇔ `if (operator<(a,b))`
  - ```
    bool operator<(const A & x, const A & y) {
     bool resultat;
     … // Comparaison x et y
     return resultat;
    }
    ```

Surcharge opérateurs *(3/4)*

- Opérations arithmétiques unaires ⇒ méthodes

- Incrément préfixé ⇒ retour d'une référence
  - `b = ++a ⇔ b = a.operator++()`
  - ```
    A & operator++() {
      … // Incrémentation de "this"
      return *this;
    }
    ```

- Incrément postfixé ⇒ retour d'une copie avant incrément
  - `b = a++ ⇔ b = a.operator++(0)`
  - ```
    A operator++(int) {
      A copie = *this;
      … // Incrémentation de "this"
      return copie;
    }
    ```

Surcharge opérateurs *(4/4)*

- Opérateurs (binaires) de flux ⇒ fonctions
  - `f << a` ⇔ `operator<<(f,a)`
  -
```
std::ostream & operator<<(std::ostream & flux,
                          const A & x) {
 … // Ecriture de x dans le flux
 return flux;
}
```
  - Retour du flux pour le chaînage
    - `f << a << b` ⇔ `operator<<(operator<<(f,a),b)`
- Ne jamais passer un flux par copie
- Autres symboles surchargeables
  - `()`, `[]`, `*`, `,` …