

## PARTIE IV

### Généricité

Bruno Bachelet

Loïc Yon

- Concept apparu dès les années 70
- Ce n'est pas un concept objet
  - Les principes objets ne sont pas nécessaires
- ... mais proposé par les langages objets !
  - ADA
  - C++
  - Java, C#

- Définir des entités abstraites du type de données
  - Structures de données: vecteur, pile, file, ensemble...
  - Algorithmes: chercher, trier, insérer, extraire...

⇒ Abstraction de données

- Autre manière de factoriser le code
  - Dans une fonction, les paramètres sont des valeurs
    - Dans sa définition, des valeurs sont inconnues
    - Au moment de l'appel (à l'exécution), ces valeurs sont fixées
  - Dans un générique, les paramètres sont des types
    - Dans sa définition, des types sont inconnus
    - Au moment d'utiliser le générique (à la compilation), ces types sont fixés

- Un générique est un modèle
  - ❑ Instanciation = création d'un élément à partir d'un modèle
  - ❑ Instancier un générique  $\Rightarrow$  fixer le type de ses paramètres
  
- Spécificités en C++
  - ❑ Génériques appelés «*templates*»
  - ❑ Des constantes peuvent aussi être des paramètres
  - ❑ Peuvent être génériques: fonctions, classes ou méthodes
    - Depuis C++14: variables globales ou attributs de classe
  - ❑ Possibilité de «spécialisation statique»
    - Une nouvelle forme de polymorphisme
    - Permet la spécialisation pour certains types de données

## ■ Algorithme de tri

- ❑ Fonctionne de la même manière sur tout type de données
  - Entiers, flottants, chaînes, instances d'une classe A...
- ❑ Suppose des fonctionnalités sur le type manipulé
  - Une relation d'ordre
    - ❑ Opérateur «<»
    - ❑ Une fonction ou un objet tiers (e.g. foncteur)
  - Un mécanisme de copie
    - ❑ Opérateur «=»

## ■ Type pile

- ❑ Fonctionne de la même manière sur tout type de données
- ❑ Suppose un mécanisme de copie

# Héritage vs. généricité

---

- La généricité est complémentaire de l'héritage
- Tous les deux fournissent une forme de polymorphisme
  - La généricité agit à la compilation
  - L'héritage agit à l'exécution
- Contribuent tous les deux à développer du code générique
  - Tous les deux font abstraction du type
  - L'un par un processus de généralisation
  - L'autre par un mécanisme de paramètre
- Avec l'héritage
  - Plus de flexibilité, mais moins de sûreté (e.g. `vector<ObjetGraphique *>`)
  - Contrôles de type effectués à l'exécution
  - Peut entraîner des ralentissements significatifs
- Avec la généricité
  - Moins de flexibilité, mais plus de sûreté (e.g. `vector<T>`)
  - Contrôles de type effectués à la compilation
  - Moins de ralentissement (voire aucun) à l'exécution

- Mot-clé «**template**»
- Précède un composant générique
  - Fonction, classe ou méthode
  - Ou variable (depuis C++14)
- **template** *<liste\_paramètres> composant*
- Définit une liste de paramètres de différentes natures
  - Point commun: leur valeur sera fixée à la compilation  $\Rightarrow$  instantiation
  - Les plus courants
    - Un type: **typename T** (ou **class T**)
    - Une constante: **int N**
  - Les autres
    - Un *template*: **template <typename> class C**
    - Un «pack» de paramètres: **typename... P** (depuis C++11, cf. *variadic templates*)

- Définition d'une fonction générique

```
template <typename T>  
const T & min(const T & a, const T & b)  
{ return (a < b ? a : b ); }
```

- Suppose l'opérateur de comparaison sur le type paramétré «**T**»

- Appel à une fonction générique (instanciation + exécution)

```
int i, j;  
...  
int k = min<int>(i, j);
```

- Instanciation  $\Rightarrow$  fixer les types paramétrés



# Polymorphisme statique

---

- Pas obligatoire de préciser les types paramétrés à l'instanciation
- Si le compilateur a suffisamment d'informations, il déduit les types
  - Comme avec la surcharge de nom
  - Forme de polymorphisme statique
  - `int i, j; ... min(i, j);`  $\Rightarrow$  instanciation de `min<int>`
  - `double a, b; ... min(a, b);`  $\Rightarrow$  instanciation de `min<double>`
- Le compilateur peut effectuer des conversions implicites si les types ne correspondent pas tout à fait

- Algorithme de tri générique
  - Classe générique «**AlgoTri**» avec paramètre «**T**»
  - **T** = type des éléments à trier
  - Éléments comparés à l'aide de la méthode «**estAvant**»
    - Permet un tri décroissant par exemple

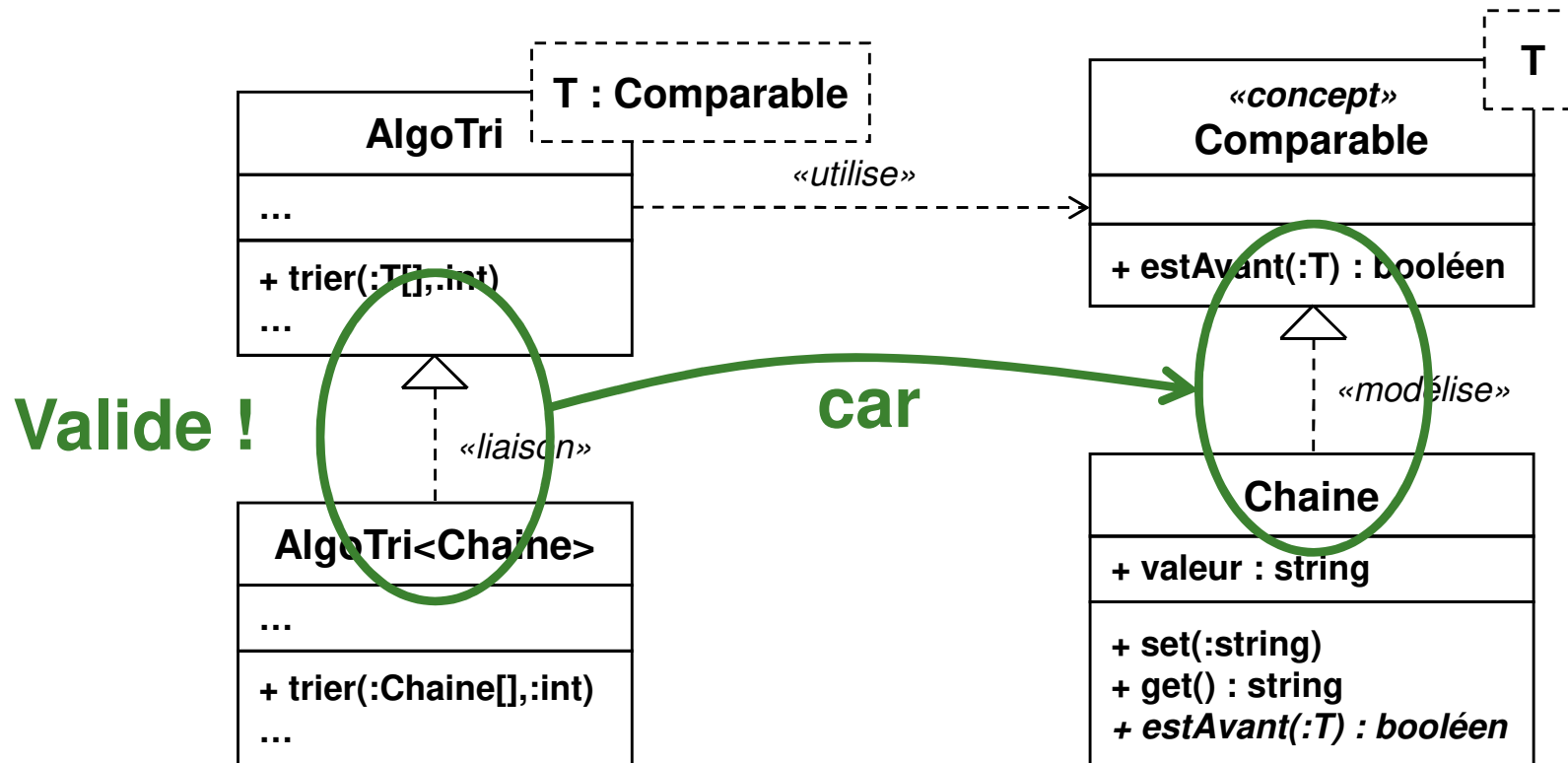
- Exemple

```
template <typename T>
void AlgoTri<T>::trier(T t[],int n) {
    for (int i = 0; i < n-1; i++)
        for (int j = i+1; j < n; j++)
            if (t[j].estAvant(t[i]))
                { T x = t[i]; t[i] = t[j]; t[j] = x; }
}
```

- Hypothèse: le type «**T**» possède la méthode «**estAvant**»
  - Vérification faite à la compilation, au moment de l'instanciation

- L'interface supposée de «**T**» fait partie d'un «concept»
- Concept = ensemble «nommé» de spécifications/contraintes
  - ❑ Concerne l'interface: existence d'une méthode
  - ❑ Concerne l'implémentation: sémantique d'une méthode
  - ❑ Mais aussi toute contrainte pertinente liée à l'utilisation du type
- Proposition de définition du concept «**Comparable**»
  - ❑ La méthode d'instance «**estAvant**» doit exister
    - Signature exacte (UML): **booléen estAvant(:T)**
  - ❑ Elle doit définir une relation d'ordre (partielle) entre deux objets
    - Pseudo-code: **si a.estAvant(b) = vrai alors b.estAvant(a) = faux**
- Le type «**T**» de l'algorithme peut être contraint par ce concept
  - ❑ «**T**» respecte le concept «**Comparable**»
  - ❑ On dit aussi: «**T**» modélise le concept «**Comparable**»

# Les «concepts» (3/3)



- En Java: concepts limités aux interfaces
  - Un paramètre générique peut être contraint par des interfaces
- En C++: concepts intégrés au langage depuis C++20
  - Avant: seule une documentation permettait de les identifier (cf. doc STL)
  - Maintenant: un paramètre générique peut être contraint par des concepts

- Définition d'une classe générique

```
template <typename T, int N>
class Pile {
    private:
        T elements[N];
        int sommet;
    public:
        Pile();
        void ajouter(const T &);
        T retirer();
};
```

- Instanciation d'une classe générique

- ❑ `Pile<int, 256> p;`
- ❑ `using pile_double_t = Pile<double, 100>;`

# Paramètres par défaut (1/2)

---

- Possibilité d'une valeur par défaut pour un paramètre
- Constante par défaut
  - `template <typename T, int N = 256>`  
    `class Pile;`
  - `Pile<int>`  $\Rightarrow$  instantiation de `Pile<int, 256>`
- Type par défaut
  - `template <typename T, typename C = int>`  
    `class TableHachage;`
  - `TableHachage<string>`  
     $\Rightarrow$  instantiation de `TableHachage<string, int>`

# Paramètres par défaut (2/2)

- Exemple plus subtil

- Le type de structure utilisée pour modéliser une pile devient un paramètre

- Définition de la classe

```
template <typename T, typename C>
class Pile {
    private:
        C elements;
    ...
};
```

- Instanciation: `Pile<int, std::vector<int>>`

- Possibilité d'une structure par défaut

- `template <typename T, typename C = std::vector<T>>`  
    `class Pile;`
- `Pile<int>`  $\Rightarrow$  instanciation de `Pile<int, std::vector<int>>`

# Paramètre «*template template*»

---

- Possibilité d'avoir une classe générique comme paramètre
  - Mot-clé «**template**» utilisé dans les paramètres du générique
  - Exemple: pile paramétrée par la classe générique de la structure sous-jacente

- Définition de la classe

```
template <typename T, template <typename> class C>
class Pile {
    private:
        C<T> elements;
    ...
};
```

- Instanciation: **Pile<int, std::vector>**
- Attention: «**C**» n'est pas un type mais bien un modèle !
  - **C** = classe générique, **C<T>** = type de la structure sous-jacente
  - **Pile<int, std::vector<int>>** est donc incorrect



# Méthodes génériques

- Exemple: copie de piles de types différents

- `Pile<double> p1;`  
`Pile<int> p2;`  
  
...  
`p1 = p2;` // Incorrect, "p1" et "p2" de types différents

- Solution avec une méthode générique

- ```
template <typename T> class Pile {  
    ...  
    template <typename U> void copier(const Pile<U> &);  
};
```

- Appel à la méthode générique

- Instanciation implicite: `p1.copier(p2);`
  - Instanciation explicite: `p1.copier<int>(p2);`

- Opérateur d'affectation générique

- `template <typename U> Pile<T> & operator=(const Pile<U> &);`

# Implémentation d'un template (1/2)

---

- Normalement, séparation interface et implémentation
  - ❑ Fichier entête
    - Déclaration méthodes + attributs
  - ❑ Fichier implémentation
    - Définition méthodes + attributs statiques
- Pour la suite, méthode «*template*»  
= méthode générique ou méthode d'une classe générique
- Implémentation des méthodes *template* dans un entête
  - ❑ Utilisation des méthodes *inline* similaire aux méthodes *template*
  - ❑ Ce sont des modèles de méthodes
  - ❑ Leur implémentation doit être visible au moment de l'appel
- Conseil: placer les implémentations en dehors de la classe

# Implémentation d'un template (2/2)

---

```
template <typename T, int N>
class Pile {
private:
    T elements[N];
    int sommet;
public:
    Pile();
    void ajouter(const T &);
    T retirer();
};
```

```
template <typename T, int N>
Pile<T,N>::Pile() : sommet(0) {}
```

```
template <typename T, int N>
void Pile<T,N>::ajouter(const T & e)
{ elements[sommet++] = e; }
```

```
template <typename T, int N>
T Pile<T,N>::retirer() { return elements[--sommet]; }
```

# Compilation d'un générique (1/2)

---

- Un code générique n'est pas compilé
  - Analyse «succincte» au niveau syntaxique
- Un code instancié est compilé
  - Analyse «complète» au niveau sémantique
- Instanciation = réécriture
  - Code générique dupliqué
  - Types paramètres remplacés par types concrets
- Equivalent d'un copier-coller-remplacer
  - Permet une efficacité optimale du code

# Compilation d'un générique (2/2)

---

- Attention: une instance par jeu de paramètres
  - Travail du compilateur important  $\Rightarrow$  temps de compilation
  - Duplication de code  $\Rightarrow$  taille de l'exécutable
  
- Attention: aucun lien entre deux instances (en C++)
  - Pas de parenté entre les instances d'une classe générique
  - Pas de passerelle de conversion
    - `Pile<int> p1;`
    - `Pile<double> p2;`
    - `p2 = p1;`  $\Rightarrow$  interdit (bien que la conversion `int`  $\rightarrow$  `double` existe)
  - Même dans le cas de constantes
    - `Pile<int,10> p1;`
    - `Pile<int,20> p2;`
    - `p2 = p1;`  $\Rightarrow$  interdit

# Relation d'amitié (1/2)

- Amitié = rompre l'encapsulation avec un composant bien identifié
  - Mot-clé «**friend**»
- A éviter, mais parfois nécessaire
  - Entre composants d'un même module
  - Evite des méthodes publiques inutiles hors module
  - En C++, pas d'amitié inter-module comme en Java
- Autoriser la classe «**B**» à voir les membres cachés de la classe «**A**»
  - ```
class A {  
    friend class B;  
    ...  
};
```
  - Membre caché = «**protected**» ou «**private**»
- L'amitié n'est pas réciproque (ni transitive)
  - Réciprocité ⇒ 

```
class B { friend class A; ... };
```

- Une fonction peut être amie
  - `class A { friend void f(int); ... };`
  - La fonction «`f(int)`» voit les membres cachés de «`A`»
  - Ce n'est pas le cas des autres surcharges de «`f`»
- Une méthode peut être amie
  - `class A { friend void B::g(void); ... }`
  - La méthode «`g`» de la classe «`B`» voit les membres cachés de «`A`»
- Déclaration préalable pas nécessaire pour établir une amitié
  - Sauf cas particuliers avec généricité (voir plus loin)
- L'amitié ne remplace pas une déclaration
  - «`A`» ne peut pas utiliser «`f`» ou «`B`» sans déclaration préalable

# Déclaration anticipée (1/3)

- Pour utiliser une classe ou une fonction, celle-ci doit être connue
  - Elle doit être déclarée
  - Pour une fonction  $\Rightarrow$  prototype
  - Pour une classe  $\Rightarrow$  déclaration complète ou «anticipée»
- Dépendance réciproque entre classes  
 $\Rightarrow$  déclaration anticipée («*forward declaration*»)
- Avant chaque classe, déclaration anticipée de l'autre classe
  - Entête classe A

```
class B; // Déclaration anticipée
...
class A { ... void m1(B & b); ... };
```
  - Entête classe B

```
class A; // Déclaration anticipée
...
class B { ... void m2(A & a); ... };
```



# Déclaration anticipée (2/3)

---

- Déclaration anticipée = déclaration partielle d'un type

- Seul le nom est indiqué
- Rien n'est précisé sur la structure du type

⇒ Restrictions tant qu'il n'est pas complètement déclaré

- Aucune méthode ou attribut ne peut être appelé

- `A a;` ⇒ interdit
- `A::x;` ⇒ interdit
- `A::m();` ⇒ interdit

- Le type peut être utilisé sans restriction dans les déclarations

- `void m(A *);` ⇒ ok
- `void m(A &);` ⇒ ok
- `void m(A);` ⇒ ok

# Déclaration anticipée (3/3)

---

- Seuls les pointeurs et références peuvent être utilisés dans les définitions
  - Variables
    - `A * a;`  $\Rightarrow$  ok
    - `A & a;`  $\Rightarrow$  ok
    - `a->m();`  $\Rightarrow$  interdit
  - Arguments
    - `void m(A *) {...}`  $\Rightarrow$  ok
    - `void m(A &) {...}`  $\Rightarrow$  ok
    - `void m(A) {...}`  $\Rightarrow$  interdit
- Possibilité de faire des alias d'une déclaration anticipée
  - `using mon_ami = A;`

## ■ Exemples

- ❑ `template <typename T> class B;`
- ❑ `template <typename T> void f();`

## ■ Amitié avec toutes les instances

- ❑ `class A { template <typename T> friend class B; ... };`
- ❑ `class A { template <typename T> friend void f(); ... };`

## ■ Amitié avec une instance particulière

- ❑ Attention: une déclaration préalable (de «**B**» et «**f**») est nécessaire
- ❑ `class A { friend class B<int>; ... };`
- ❑ `class A { friend void f<int>(); ... };`

# Amitié et généricité (2/2)

- Cas d'une classe générique: exemple d'amitié avec une instance

```
template <typename T> class Vecteur; // Déclaration anticipée
```

```
template <typename T> // Prototype nécessite déclaration  
ostream & operator << (ostream &, const Vecteur<T> &);
```

```
template <typename T> class Vecteur {  
    // Amitié nécessite prototype  
    friend ostream & operator<< <T> (ostream &,  
                                       const Vecteur<T> &);  
  
    private  
        T * elements;  
        int nb;  
  
    ...  
};
```

```
template <typename T> // Définition nécessite amitié  
ostream & operator<<(ostream & f, const Vecteur<T> & v) {  
    for (int i=0; i<v.nb; ++i) f << v.elements[i] << " ";  
    return f;  
}
```

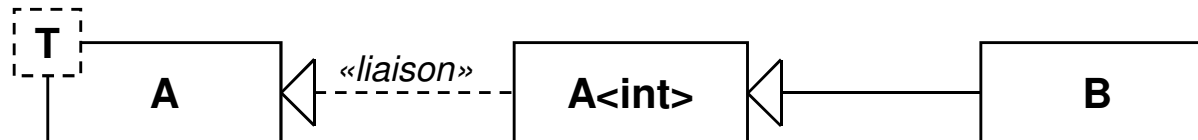
# Héritage et généricité (1/4)

## ■ Héritage «simple»

- Héritage d'une instance d'une classe générique
- Exemple: «**NuagePoint**» hérite de «**Vecteur<Point>**»

## ■ Illustration

- **template <typename T>**  
  **class A {...};**
- **class B : public A<int> {...};**



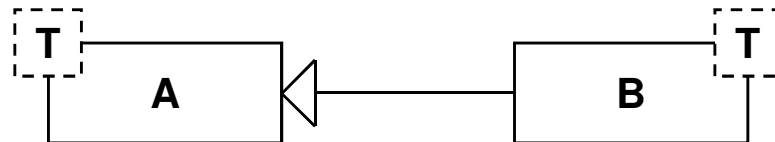
# Héritage et généricité (2/4)

## ■ Héritage «classique»

- Héritage entre deux classes génériques
- Exemple: «**FileAttente<T>**» hérite de «**Vecteur<T>**»

## ■ Illustration

- **template <typename T>**  
**class A {...};**
- **template <typename T>**  
**class B : public A<T> {...};**



# Héritage et généricité (3/4)

- Héritage avec «extension»
  - Héritage entre classes génériques avec ajout d'un paramètre
  - Exemple: «**FilePriorite**<T, C>» hérite de «**Vecteur**<T>»
    - C = objet comparateur qui indique la relation d'ordre

- Illustration

- `template <typename T>`  
`class A {...};`
- `template <typename T, typename U>`  
`class B : public A<T> {...};`

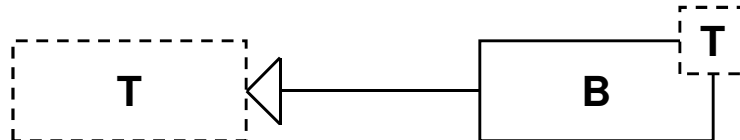


## ■ Héritage «générique»

- Héritage d'une classe qui est un paramètre
  - Extension potentielle de toutes les classes
- Exemple: «**Comparable**<T>» hérite de «**T**»
  - Toute classe peut devenir un «comparable»

## ■ Illustration

- `template <typename T> class B : public T {...};`





- Composant générique = modèle indépendant des types
- Mais cela peut être pénalisant
  - Exemple: recherche d'un élément dans une structure
  - Approches différentes suivant que la structure soit triée ou non
- ⇒ Mécanisme de spécialisation «statique»
  - Spécialisation du modèle générique pour un jeu de paramètres
  - Jeu de paramètres partiel ou complet
    - On parle aussi d'«instanciation» partielle ou complète
- Associé au polymorphisme statique de l'instanciation
  - «Meilleure» instanciation choisie en fonction du jeu de paramètres

# Spécialisation d'une fonction générique

- Modèle générique d'une fonction de calcul de moyenne

```
template <int N> double moyenne(int * tab) {  
    double somme = 0.0;  
    for (int i = 0; i < N; ++i) somme += tab[i];  
    return (somme/N);  
}
```

- Spécialisation du modèle pour  $N = 2$  et  $N = 1$

```
template <> double moyenne<2>(int * tab)  
{ return (double(tab[0] + tab[1])/2); }
```

```
template <> double moyenne<1>(int * tab)  
{ return double(tab[0]); }
```

- Attention

- ❑ Déclarer d'abord la version générique, puis les versions spécifiques
- ❑ Spécialisation «partielle» d'une fonction (ou méthode) interdite

# Spécialisation d'une classe générique

- Modèle générique d'un vecteur d'éléments

```
template <typename T> class Vecteur {  
    private:  
        T * elements;  
        int taille;  
        ...  
    public: T operator[](int i) { return elements[i]; }  
};
```

- Spécialisation du modèle pour  $T = \text{bool}$

```
template <> class Vecteur<bool> {  
    private:  
        char * elements;  
        int taille;  
        ...  
    public: bool operator[](int i)  
    { return ((elements[i/8] >> (i%8)) & 1); }  
};
```

- Mécanisme statique lors de l'instanciation d'un modèle
  - Sélection de la version la plus spécialisée
  - En fonction du jeu de paramètres

⇒ Génération du code le plus dédié possible

- Exemples d'instanciations
  - `moyenne<10>(tab);` ⇒ version générique
  - `moyenne<2>(tab);` ⇒ version spécialisée pour  $N = 2$
  - `Vecteur<int> v;` ⇒ version générique
  - `Vecteur<bool> v;` ⇒ version spécialisée pour  $T = bool$

# Spécialisation partielle

- Spécialisation partielle  
= spécialisation avec un jeu de paramètres incomplet

- Retour sur l'exemple de calcul de moyenne

```
template <typename T, int N> class Moyenne {  
    public: static T calculer(T * tab) {  
        T somme = T();  
        for (int i = 0; i < N; ++i) somme += tab[i];  
        return (somme/T(N));  
    }  
};
```

- Spécialisation pour  $N = 2$  ( $T$  reste inconnu)

```
template <typename T> class Moyenne<T, 2> {  
    public: static T calculer(T * tab)  
    { return ((tab[0] + tab[1])/T(2)); }  
};
```

# Retour sur l'héritage avec généricité

---

## ■ Exemple

- ❑ `template <typename T> class A`  
    `{ ... public: void m(); ... };`
- ❑ `template <typename T> class B : public A<T>`  
    `{ ... public: void n() { ... m(); ... } ... };`

## ■ Instanciation partielle $\Rightarrow$ doute

- ❑ La version générique de A peut être remise en question par une spécialisation
- ❑ La méthode «`m`» peut ne pas exister dans cette spécialisation  
 $\Rightarrow$  appel à la fonction «`m`» si elle existe au lieu de la méthode

## ■ Conseil: toujours utiliser «`this->`» sur un membre hérité

- ❑ Cela provoquera une erreur si une spécialisation ne possède pas ce membre
- ❑ Toujours mieux qu'un comportement implicite (certainement non voulu)
- ❑ Solution: `void n() { ... this->m(); ... }`

- Deux syntaxes possibles
  - ❑ C++03: `typedef pair<int, double> paire2_t;`
  - ❑ C++11: `using paire2_t = pair<int, double>;`
  - ❑ Strictement équivalentes  $\Rightarrow$  privilégier la seconde
- Alias de type *template* possible avec «**using**»
  - ❑ `template <typename T> using paire_t = pair<int, T>;`
  - ❑ Forme d'instanciation «partielle»
- Type interne à une classe
  - ❑ Possibilité de déclarer une classe dans une autre
    - `class vector { ... class iterator { ... }; ... };`
  - ❑ Et de déclarer des alias de type
    - `class vector { ... using iterator = ...; ... };`

- Indique que ce qui suit est un type
- Utilisé dans la déclaration d'un paramètre
  - `template <typename T>`
  - Equivalent à: `template <class T>`
- Utilisé pour lever une ambiguïté
  - A cause de l'instanciation partielle
  - Tant que l'instanciation n'est pas effective  $\Rightarrow$  doute
  - Exemple
    - `template <typename T> class A`  
    `{ public: using type_t = T; // type interne };`
    - `template <typename T> class B`  
    `{ public: using type_A = typename A<T>::type_t; };`
  - Avec l'instanciation partielle, doute sur la nature de «`A<T>::type_t`»
    - Type ou attribut ?  $\Rightarrow$  `typename`



# Variadic template (1/2)

---

- *Variadic template* = générique à paramètres variables
  - Depuis C++11
  - Liste des paramètres *templates* non fixée
  - A l'instar des arguments variables d'une fonction
- Permet de modéliser des collections hétérogènes
  - `template <typename... TYPES> class Tuple;`
- Syntaxe simple pour instancier le générique
  - `Tuple<int, double, std::string> t;`
- Mais syntaxe pas toujours intuitive pour écrire le générique
  - Mécanisme d'«expansion»
  - Ou approche récursive (métaprogrammation)

- Permet de renforcer le contrôle de types
  - `fprintf(const char * format, ...);`
    - Impossible d'identifier les types des arguments variables
  - `template <typename... TYPES>`  
`void fprintf(const char * format, TYPES... args);`
    - Possibilité d'identifier le type de chaque argument variable
- Paramètres variables = «pack» de paramètres
  - Pack représenté par le symbole «...»
  - Pack = 0 à  $n$  paramètres
- Pack de valeurs: `template <int... VALEURS>`
- Pack de types: `template <typename... TYPES>`

# Expansion de pack (1/3)

---

- `template <typename... TYPES>`
  - ⇒ `template <typename T1, ... ,typename Tn>`
    - Il s'agit d'une illustration:  $T1...Tn$  n'existent pas explicitement
- Accès direct à un paramètre d'un pack impossible
  - Un paramètre n'a pas d'identifiant
    - Aucun moyen d'obtenir le nom d'un paramètre
  - Un paramètre n'a pas de numéro
    - Aucun moyen direct d'obtenir le  $n^{\text{ième}}$  paramètre
- Parcours et identification possibles par récursivité
  - Ecriture de *templates* récursifs ⇒ voir métaprogrammation
- Nombre d'éléments d'un pack: opérateur `sizeof... (PACK)`

# Expansion de pack (2/3)

---

- Pour utiliser ces paramètres  $\Rightarrow$  mécanisme d'expansion
- On décrit un schéma d'expansion
  - Expression contenant l'identifiant d'un pack
  - Et terminée par « . . . »
- Expansion  $\Rightarrow$  réplication du schéma
  - Pour chaque paramètre du pack
  - Séparation par une virgule
- Depuis C++17: introduction des «*fold expressions*»
  - Expansion possible avec les opérateurs binaires

# Expansion de pack (3/3)

- Supposons un pack
  - `template <typename... TYPES>`
- Exemples d'expansions possibles
  - `TYPES...`  
⇒ `T1, ..., Tn`
  - `X<TYPES>...`  
⇒ `X<T1>, ..., X<Tn>`
  - `X<TYPES>::a...`  
⇒ `X<T1>::a, ..., X<Tn>::a`
  - `X<TYPES>::m(u,v)...`  
⇒ `X<T1>::m(u,v), ..., X<Tn>::m(u,v)`
  - `const TYPES &... x`  
⇒ `const T1 & x1, ..., const Tn & xn`