

PARTIE IV

Généricité

Bruno Bachelet

Loïc Yon

- Concept apparu dès les années 70
- Ce n'est pas un concept objet
 - Les principes objet 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, 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, 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
 - ❑ 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
 - ❑ Contrôles de type effectués à l'exécution
 - ❑ Peut entraîner des ralentissements significatifs
- Avec la généricité
 - ❑ Moins de flexibilité, mais plus de sûreté
 - ❑ 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
- Définit des paramètres
 - Soit des types: `typename T`
 - Soit des constantes: `int N`

- 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)

```
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

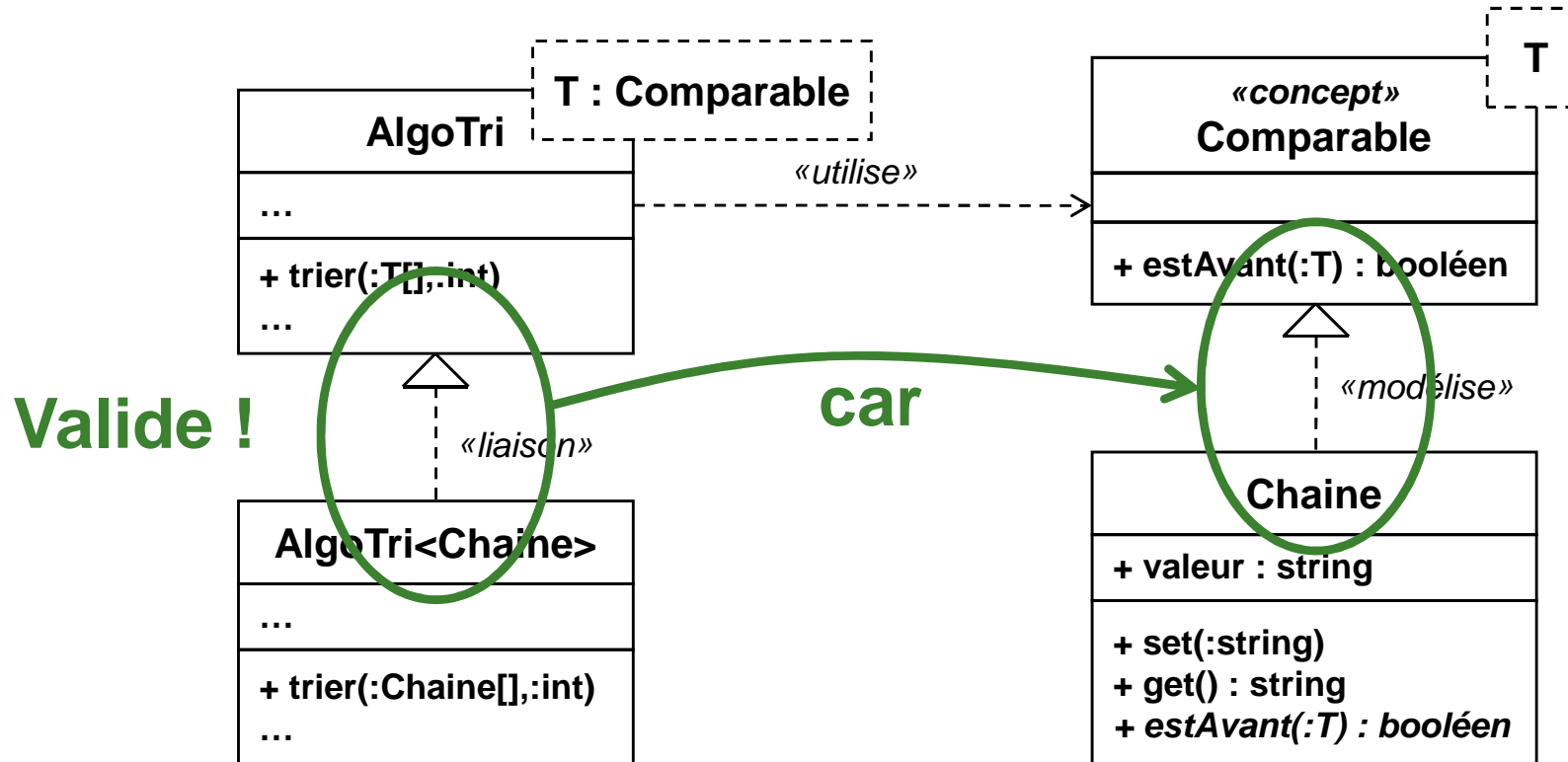
- Algorithme de tri

```
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**» est appelée un «concept»
- Concept = ensemble de spécifications
 - 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

Les «concepts» (2/2)

- Dans l'algo, «T» doit respecter le concept «Comparable»
 - On dit: «T» modélise le concept «Comparable»



- En C++, les concepts sont pour l'instant implicites
 - Seule une documentation permet de les identifier
 - Voir la documentation de la STL par exemple
 - En Java: les types contraints + interface
 - A venir: «concepts lite» (C++20)

- Définition d'une classe générique

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

- Instanciation d'une classe générique

- `Pile<int,256> p;`
- `typedef Pile<double,100> pile_double_t;`

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
 - Paramétrage du type de structure utilisé pour modéliser une pile
- Définition de la classe

```
template <typename T,typename C> class Pile {  
    ...  
    protected: C elements;  
    ...  
};
```
- Instanciation: `Pile<int,vector<int> >`
- Proposons une structure par défaut
 - `template <typename T,typename C = vector<T> >`
 `class Pile;`
 - `Pile<int>` \Rightarrow instanciation de `Pile<int,vector<int> >`
- Remarque de syntaxe
 - Il arrive d'avoir `>>` dans la définition d'un template
 - Avant C++11: mettre un espace pour éviter la confusion avec l'opérateur de flux

- Exemple de déclaration

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

- Utilisation

```
Pile<double> p1;  
Pile<int> p2;  
...  
p1.copier(p2);
```

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

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 de lisibilité: 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);  
        void ajouter(const T &);  
        T retirer(void);  
};
```

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

Mot-clé «*typename*»

- Indique que ce qui suit est un type
- Utilisé dans la déclaration d'un paramètre
 - `template <typename T>`
 - `template <class T>` \Rightarrow déconseillé, à éviter
- 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: typedef T type_t;
};
```
    - ```
template <typename T> class B {  
    typedef typename A<T>::type_t type_A;  
};
```
 - On ne sait pas ce qu'est `A<T>::type_t`
 - Type ou attribut ? \Rightarrow `typename`

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 2 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
 - Même sur les paramètres constants
 - `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
- 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
- Mot-clé «**friend**»
 - ❑

```
class A {  
    friend class B;  
    friend void f(void);  
    friend void C::g(void);  
    ...  
};
```
 - ❑ La classe B voit les membres cachés de A
 - ❑ La fonction f voit les membres cachés de A
 - ❑ La méthode g de la classe C voit les membres cachés de A

Relation d'amitié (2/2)

- L'amitié n'est pas réciproque (ni transitive)
 - `class B { friend class A; ... };`
- Une fonction peut être amie
 - `class B { friend void f(void); ... };`
- Déclaration préalable pas nécessaire pour établir une amitié
 - Sauf cas particuliers avec généricité (voir plus loin)
- L'amitié ne remplace pas une déclaration
 - `f` ou `A` ne peuvent pas être utilisées par `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 \Rightarrow déclaration anticipée («*forward declaration*»)
 - ❑ Car une classe doit forcément être déclarée avant l'autre
- Exemple de dépendance réciproque
 - ❑ `class B; // Déclaration anticipée`
...
`class A { ... void m1(B & 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
- Utilisation autorisée avec «`typedef`» et «`friend`»
 - `typedef A mon_ami;`

■ Exemples

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

■ Amitié avec toutes les instances

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

■ 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>(void); ... };`

Amitié et généricité (2/2)

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

```
template <typename T> class Vecteur;
```

```
template <typename T>  
ostream & operator << (ostream &, const Vecteur<T> &);
```

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

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

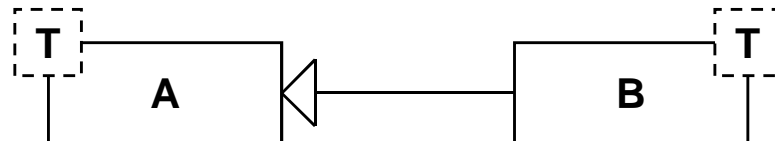
Héritage et généricité (1/4)

- 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> {...};`



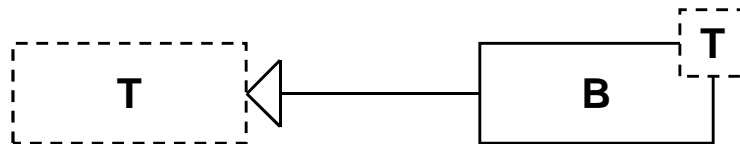
Héritage et généricité (4/4)

■ 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 à l'ordre
 - ❑ Déclarer d'abord la version générique, puis les versions spécifiques
- En C++, 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 {  
    protected: T * elements;  
    protected: int taille;  
    ...  
    public: T operator [] (int i)  
    { return elements[i]; }  
};
```

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

```
template <> class Vecteur<bool> {  
    protected: char * elements;  
    protected: 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 $T = \text{bool}$

Spécialisation partielle (1/2)

- 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)); }  
};
```

Spécialisation partielle (2/2)

- Exemple de recherche d'un élément dans un conteneur

```
template <typename T,typename C>
class Recherche {
public:
    static bool executer(const C & conteneur,
                        const T & element);
};
```

- Spécialisation pour $C = \text{vector}<T>$ (T reste inconnu)

```
template <typename T>
class Recherche< T,vector<T> > {
public:
    static bool executer(const vector<T> & conteneur,
                        const T & element);
};
```

Alternative à l'héritage (1/2)

- Polymorphisme dynamique \Rightarrow coût important à l'exécution

- Exemple: stratégie de recherche dans un conteneur

```
template <typename T> class Vecteur {  
    ...  
    public: virtual bool rechercher(const T & x);  
};
```

```
template <typename T> class VecteurTrie  
: public Vecteur<T> {  
    ...  
    public: bool rechercher(const T & x);  
};
```

- Héritage des conteneurs vraiment nécessaire ?
 - Aspect dynamique sans intérêt \Rightarrow spécialisation statique

Alternative à l'héritage (2/2)

- Version générique avec spécialisation statique

```
template <typename T,typename C> class Recherche {  
    public: static bool executer(const C & c,  
                                const T & e);  
};
```

```
template <typename T>  
class Recherche< T,VecteurTrie<T> > {  
    public:  
        static bool executer(const VecteurTrie<T> & c,  
                              const T & e);  
};
```

- Astuce: fonction d'aide à la déduction automatique des paramètres

```
template <typename T,typename C>  
bool rechercher(const C & c,const T & e)  
{ return Recherche<T,C>::executer(c,e); }
```

Retour sur l'héritage avec généricité

■ Exemple

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

■ Instantiation partielle \Rightarrow doute

■ Toujours utiliser **this->** sur un membre hérité

- ❑ Si une fonction «**m**» existe, elle peut être appelée
- ❑ Donc: **void n(void) { ... this->m(); ...; }**

Paramètre «*template template*»

- Possibilité d'avoir une classe générique comme paramètre d'un générique
 - Mot-clé «**template**» utilisé dans les paramètres du générique
- Exemple
 - ```
template <typename T,template <typename> class C>
class Pile {
 ...
 protected: C<T> elements;
 ...
};
```
  - Utilisation: **Pile<int,Vecteur>**
- Attention: «**C**» n'est pas un type mais bien un modèle !
  - «**C**» est une classe générique
  - C'est «**C<T>**» le type du conteneur
- **Pile<int,Vecteur<int> >** est incorrect !
  - Fonctionne avec: **template <typename T,typename C> class Pile;**