HLIN603 : Programmation Objet Avancée

# Généricité paramétrique

# Sommaire

## Définition (Généricité paramétrique)

*Possibilité d'énoncer des descriptions de classes ou de fonctions dans lesquelles certains éléments (les paramètres) restent formels.*

- « modèle de classe » *versus* classe
- « modèle de fonction » *versus* fonction
- Dialecte C++ $\rightarrow$ « template » de classe ou de fonction.
- Paramètres = types, classes, fonctions, valeurs.

## Exemple (Modèle de Pile)

*`Pile` pour `int`, `String`, `Voiture` ....*
*⇒ `modèle de pile` paramétré par le type $T$ des éléments stockés*

## Exemple (Modèle de fonction de recherche)

*`fonction de recherche` d'un élément dans un tableau*
*⇒ `modèle de fonction de recherche` d'un élément d'un type formel $T$ dans un tableau d'éléments de type $T$*

## Exemple (Modèle d'association)

*Concept d'association, des bibliothèques proposant des structures de données (des collections), représente un couple d'éléments dont le premier est une clef d'accès au second*
*⇒ `modèle d'association`, paramétré par le type de la clef et le type de la valeur*

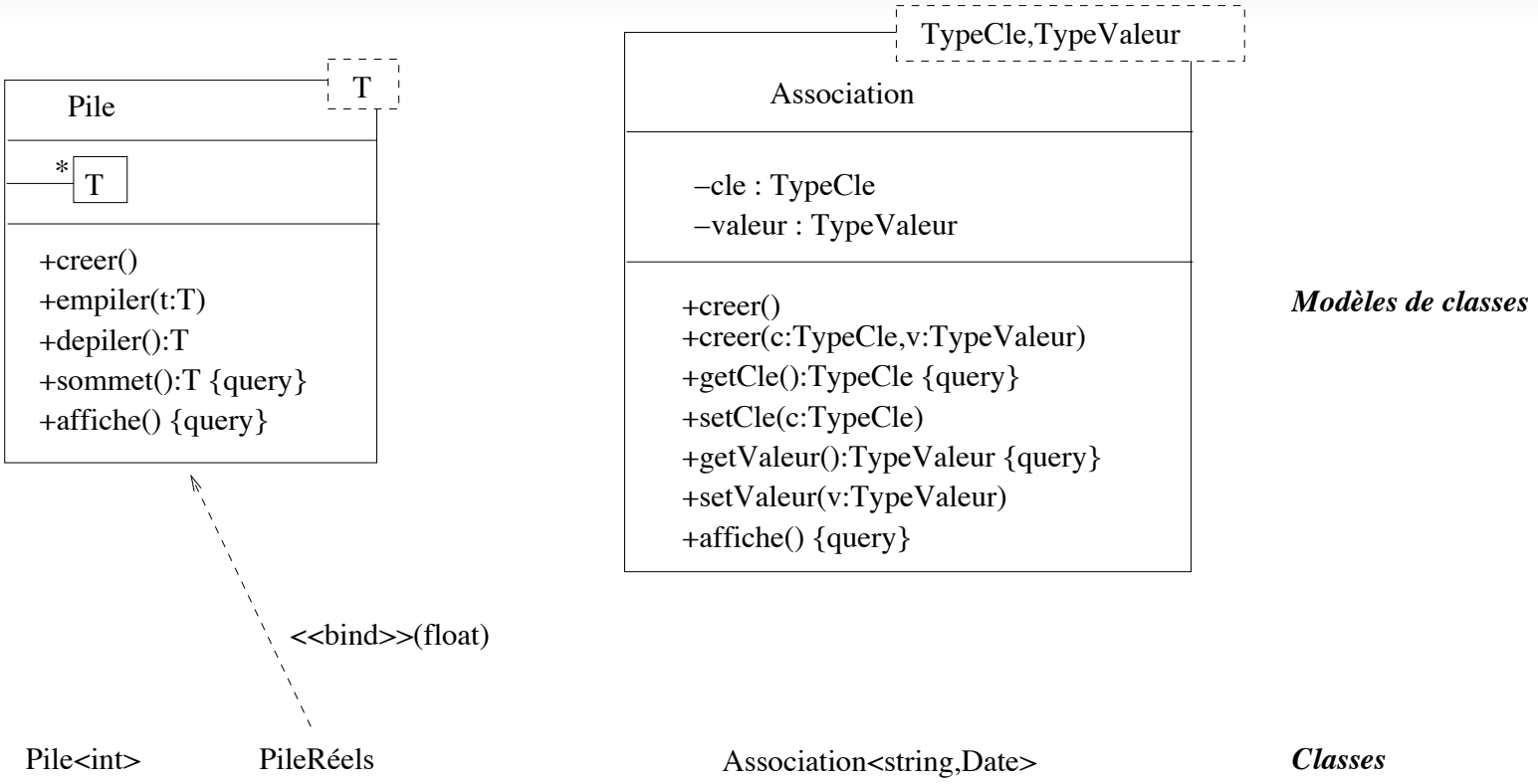


Figure : Modèles de classes et leurs instanciations notés en UML

*instanciation*, *spécialisation*, *invocation* = procédé qui consiste à lier les paramètres formels à des paramètres réels
Exemple : lier $T$ à `int` pour obtenir une pile d'entiers

# La généricité dans les langages de programmation

- Existe notamment en Ada, Eiffel, C++, et Java 5 (tiger)
- Sans généricité paramétrique, des erreurs potentielles à l'exécution dues à :
  - des contrôles de types à l'exécution **plus nombreux**
  - des coercitions de type (`dynamic_cast`) **plus nombreux**
- En C++, la bibliothèque des collections de données (*Standard Template Library*) est implémentée à l'aide de modèles de classes

# Sommaire

# Cas d'étude

Afficher sur un flot de sortie standard les éléments d'un tableau entre deux accolades

```
void affiche(ostream& os, int t[], int taillet)
{
  os << "{ ";
  for (int i=0; i<taillet; i++) os << t[i] << " ";
  os << "}" << endl;
}
```

Pour l'adapter à un tableau contenant des éléments d'un autre type `TypeElt`, il suffirait de changer, dans la signature de la fonction, `int t[]` par `TypeElt t[]` et de définir `TypeElt` :
`typedef ....(type qui nous intéresse).... TypeElt`

▷ On va généraliser ce procédé.

# Déclaration d'une fonction générique

```
template<typename TypeElt>
void afficheT(ostream& os, TypeElt t[], int taillet)
{
  os << "{ ";
  for (int i=0; i<taillet; i++) os << t[i] << " ";
  os << "}" << endl;
}
```

Dans l'entête ajoutée à la fonction

- `template` signale une déclaration de modèle
- `typename` introduit un paramètre formel de type, ici `TypeElt`

# Instanciation

En explicitant le paramètre

```
int tab[]={2,4,6,8};
afficheT<int>(cout,tab,4);

string *tab2=new string[2]; tab2[0]="lundi"; tab2[1]="mardi";
afficheT<string>(cout,tab2,2);
```

En laissant le compilateur inférer la valeur

```
afficheT(cout,tab,4);
afficheT(cout,tab2,2);
```

Notons que l'on peut instancier avec un type primitif (int) ou une classe (string)

# Problème de déduction

Modèle de fonction `saisit`

- paramètres du modèle : le type des éléments et la taille du tableau
- paramètre de la fonction : le tableau

```
template<typename TypeElt, int taillet>
void saisit(TypeElt t[])
{
  for (int i=0; i<taillet; i++) cin >> t[i];
}
```

L'appel `saisit(tab);` ne permet pas de déduire la taille du tableau
$\Rightarrow$ utiliser l'appel explicite `saisit<int,4>(tab);`

# Expression de contraintes

Contrairement à d'autres langages, notamment UML, Eiffel, Java 5 (et versions suivantes), C++ ne permet pas d'exprimer de contraintes sur les paramètres de modèles.

Dans les modèles de fonction précédents, les opérateurs d'insertion `«(ostream&,const TypeElt&)` et d'extraction `»(istream&,TypeElt&)` doivent exister pour que les instanciations soient compilables, mais il n'y a pas de moyen d'écrire cette contrainte dans le code source.

# Sommaire

# Cas d'étude : Association (Assoc)

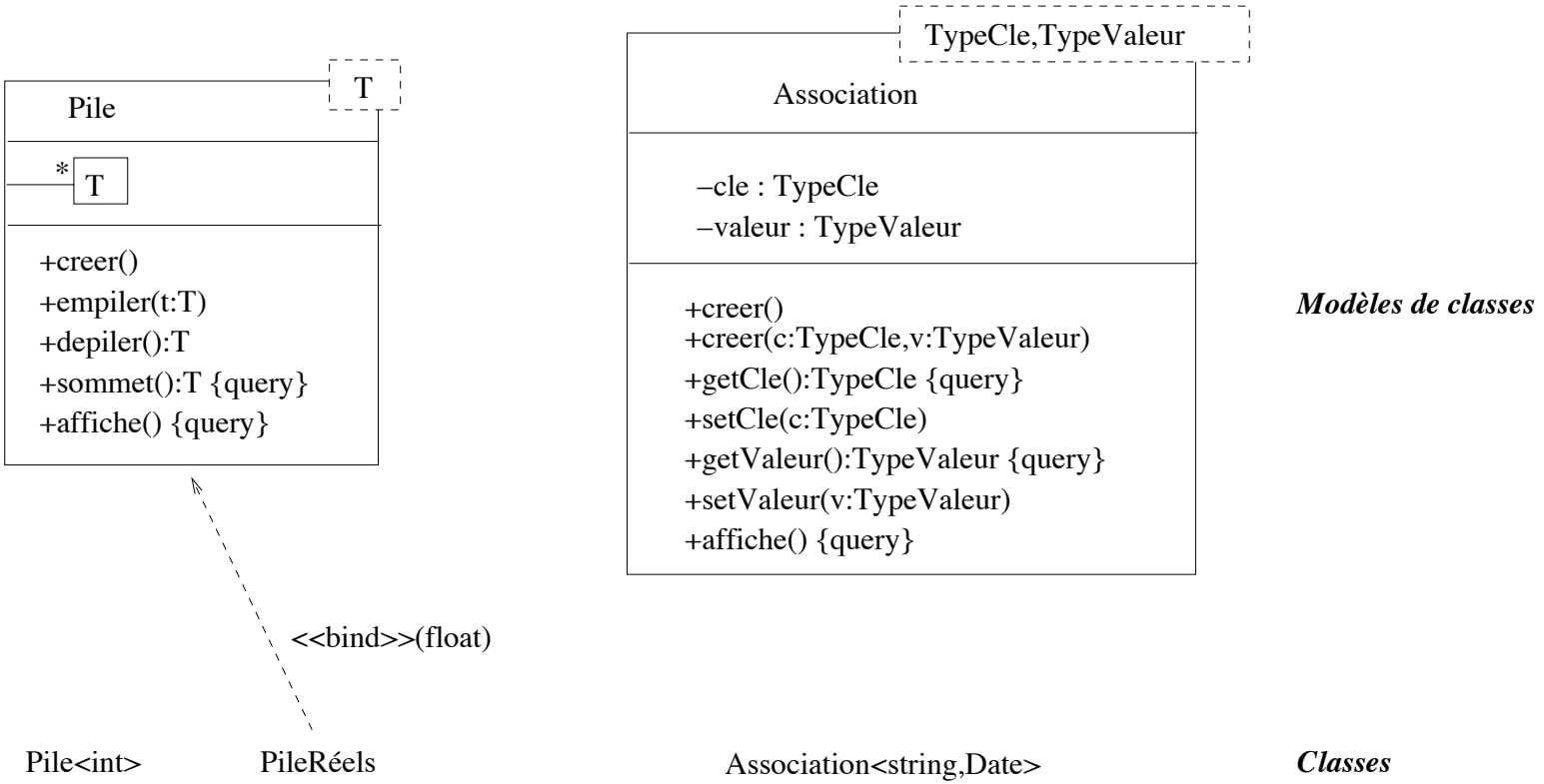


Figure : Modèles de classes et leurs instanciations notés en UML

# Déclaration du modèle de classe Assoc

```
//.............................. Assoc.h .......................

template<typename TypeCle, typename TypeValeur>
class Assoc{
private:
  TypeCle cle; TypeValeur valeur;
public:
  Assoc();
  Assoc(TypeCle, TypeValeur);
  virtual ~Assoc ();
  virtual TypeCle getCle()const;
  virtual void setCle(TypeCle);
  virtual TypeValeur getValeur()const;
  virtual void setValeur(TypeValeur);
  virtual void affiche(ostream&)const;
};

template<typename TypeCle, typename TypeValeur>
ostream& operator<<(ostream&, const Assoc<TypeCle,TypeValeur>&);
```

# Définition du modèle de classe Assoc

```
//..................... Assoc.cc ..................................
#include "Assoc.h"

template<typename TypeCle, typename TypeValeur>
Assoc<TypeCle,TypeValeur>::Assoc()   {}

template<typename TypeCle, typename TypeValeur>
Assoc<TypeCle,TypeValeur>::Assoc(TypeCle c, TypeValeur v)
:cle(c), valeur(v)    {}

template<typename TypeCle, typename TypeValeur>
Assoc<TypeCle,TypeValeur>::~Assoc ()   {}
```

```
template<typename TypeCle, typename TypeValeur>
TypeCle Assoc<TypeCle,TypeValeur>::getCle()const   {return cle;}

template<typename TypeCle, typename TypeValeur>
void Assoc<TypeCle,TypeValeur>::setCle(TypeCle c)   {cle=c;}

template<typename TypeCle, typename TypeValeur>
TypeValeur Assoc<TypeCle,TypeValeur>::getValeur()const {return valeur;}

template<typename TypeCle, typename TypeValeur>
void Assoc<TypeCle,TypeValeur>::setValeur(TypeValeur v)   {valeur=v;}

template<typename TypeCle, typename TypeValeur>
void Assoc<TypeCle,TypeValeur>::affiche(ostream &os) const
  {os <<getCle() << ", " <<getValeur();}

template<typename TypeCle, typename TypeValeur>
ostream& operator<<(ostream& os, const Assoc<TypeCle,TypeValeur>& a)
  {a.affiche(os); return os;}
```

# Instanciation

## Cas de la création d'un objet

```
// ..................... mainAssoc1.cc ..........................
#include <iostream>
#include <string>
#include "Assoc.h"

int main()
{
  Assoc<char, int*> a;
  Assoc<string,string>* pass
                  = new Assoc<string,string>("citron","jaune");
  cout << *pass;
}
```

# Instanciation

Lors de la définition d'un type

```
typedef Assoc<string,Date> CarnetAnniversaires;
```

# Instanciation

Avec directive d'instanciation explicite

- vérification de l'instanciation indépendamment de tout programme utilisateur
- optimisation de la compilation

```
//.............. AssocStringFloat.cc ................
#include <iostream>
#include <string>
#include "Assoc.cc"
template class Assoc<string, float>;
template ostream& operator<<(ostream&, const Assoc<string, float>&);
```

# Mise en œuvre

```
// ..................... mainAssoc2.cc ..........................
#include <iostream>
#include <string>
#include "Assoc.h"

int main()
{ Assoc<string, float> a;
  Assoc<string,float>* pass= new Assoc<string,float>("citron",12);
  cout << *pass;
}
```

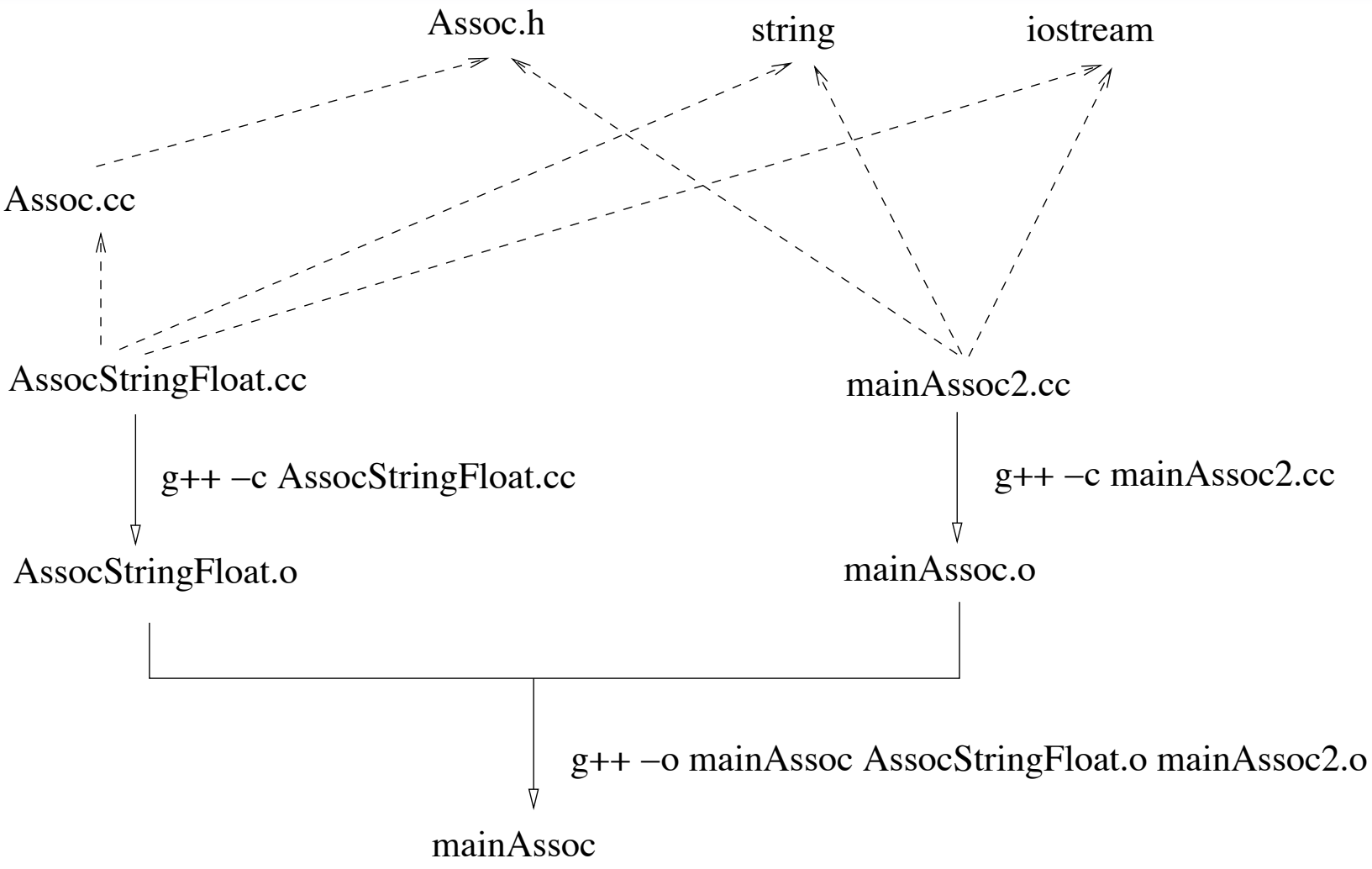


Figure : Un exemple de compilation et d'édition de liens avec instanciation séparée

# Paramétrage par des constantes

## Exemple (Passage de constante : borne d'une pile)

*Piles bornées, paramétré par le type des éléments et par la taille maximale de la pile.*
*Dans la déclaration, le mot-clef `const` est sous-entendu*

```
template<typename T, const int capacite>
class PileBornee
{ ....................};

int main()
{...PileBornee<int,10> p;...}
```

# Paramétrage par des constantes : valeurs par défaut

```
// ............... AssocVD.h ....................................
template<typename TypeCle, typename TypeValeur,
         const TypeCle vdc, const TypeValeur vdv>
class Assoc{
private:
  TypeCle cle; TypeValeur valeur;
public:
  Assoc();
  Assoc(TypeCle, TypeValeur);
  virtual ~Assoc ();
  virtual TypeCle getCle()const;
  virtual void setCle(TypeCle);
  virtual TypeValeur getValeur()const;
  virtual void setValeur(TypeValeur);
  virtual void affiche(ostream&)const;
};
template<typename TypeCle, typename TypeValeur,
         const TypeCle vdc, const TypeValeur vdv>
ostream& operator<<(ostream&,const Assoc<TypeCle,TypeValeur,vdc,vdv>&);
```

# Paramétrage par des constantes : valeurs par défaut

## Réécriture du constructeur

```
// .......... Dans AssocVD.cc ......................
template<typename TypeCle, typename TypeValeur,
         const TypeCle vdc, const TypeValeur vdv>
Assoc<TypeCle,TypeValeur,vdc,vdv>::Assoc()   {cle=vdc; valeur=vdv;}
```

# Paramétrage par des constantes : valeurs par défaut

## Instanciation

```
#include<iostream>
#include<string>
#include"externS.cc"        // contient ...   string s="truc";
#include"AssocVD.cc"

extern string s;const int i=0;

//spécialisation de l'opérateur d'insertion dans un flot
ostream& operator<<(ostream& os, const Assoc<string*,int,&s,0>& a)
 {a.affiche(os); return os;}

int main()
 {Assoc<string*, int, &s, i> a; cout << a;}
```

# Paramétrage par des fonctions

Ensemble ordonné paramétré par la fonction de comparaison

```
// ......... déclaration et définition partielle .............
template<typename T, bool compare(T,T)>
class EnsembleOrdonne
{
public:
  virtual bool comparables(T t1, T t2)const;
};

template<typename T, bool compare(T,T)>
bool EnsembleOrdonne<T,compare>::comparables(T t1, T t2)const
 {return (compare(t1,t2) || compare(t2,t1));}
```

# Ensembles ordonnés

```
// ......... instanciation ..............................

bool inf(int i1, int i2){return i1<i2;}
bool divise(int i1, int i2){return (i2%i1==0);}

int main()
{ EnsembleOrdonne<int,inf> E1; // ORDRE NATUREL
  cout << E1.comparables(3,4);

  EnsembleOrdonne<int,divise> E2; // ORDRE PARTIEL DE DIVISIBILITE
  cout << E2.comparables(8,4);
  cout << E2.comparables(4,8);
  cout << E2.comparables(5,4);
}
```

# Variables de classe

Autant de variables de classe que de spécialisations du modèle les déclarant !

```
// --------- Dans Assoc.h --------------------
template<typename TypeCle, typename TypeValeur>
class Assoc{
private:
............
  static int nbAssoc;
............
};
// ---------- Dans Assoc.cc -----------------
template<typename TypeCle, typename TypeValeur>
int Assoc<TypeCle,TypeValeur>::nbAssoc=0;
```

# Variables de classe

Si on crée deux spécialisations, par exemple grâce aux expressions :

```
typedef Assoc<string, Date> CarnetAnniversaire;
typedef Assoc<string, string> Ass;
```

On dispose alors de deux variables, en l'occurrence `CarnetAnniversaire::nbAssoc` et `Ass::nbAssoc`

# Modèles et héritage

## Exemple (un modèle dérive d'un autre modèle)

```
template <typename T>
class Conteneur{};

template <typename T>
class Liste : public virtual Conteneur<T>{};
```

# Modèles et héritage

## Exemple (un modèle dérive d'une classe)

```
class Graphe{};

template <typename TypeEtiquette>
class GrapheEtiquete : public virtual Graphe{};
```

# Modèles et héritage

## Exemple (une classe dérive d'un modèle)

```
class Point : virtual public Assoc<int,int>{};
```

# Spécialisation d'une fonction ou d'une méthode

Redéfinir un modèle de fonction ou une méthode d'un modèle de classe pour un cas particulier, par exemple :

- on veut redéfinir le modèle de recherche dans un tableau pour le cas particulier des tableaux de caractères où l'on désire ne pas tenir compte des majuscules et des minuscules ;
- on veut redéfinir la méthode `affiche` d'association pour le cas où le type des clés est un pointeur sur une chaîne et le type des valeurs est un réel.

Solution : faire cohabiter le modèle de fonction ou de méthode avec la forme spécialisée.

# Spécialisation d'une fonction ou d'une méthode

```
//............ Assoc.cc contient toujours .........
template<typename TypeCle, typename TypeValeur>
void Assoc<TypeCle,TypeValeur>::affiche(ostream &os) const
  {os <<getCle() << ", " <<getValeur();}

//............ AssocPtrStringFloat.cc contiendrait les spécialisations
template class Assoc<string*, float>;
template ostream& operator<<(ostream&, const Assoc<string*, float>&);

void Assoc<string*, float>::affiche(ostream &os) const
  {os << "clef : " << *getCle() << "-- valeur :" << getValeur();}
```

La définition d'un modèle peut même être incomplète lorsqu'il n'est possible d'écrire une méthode que dans le cadre d'une spécialisation.