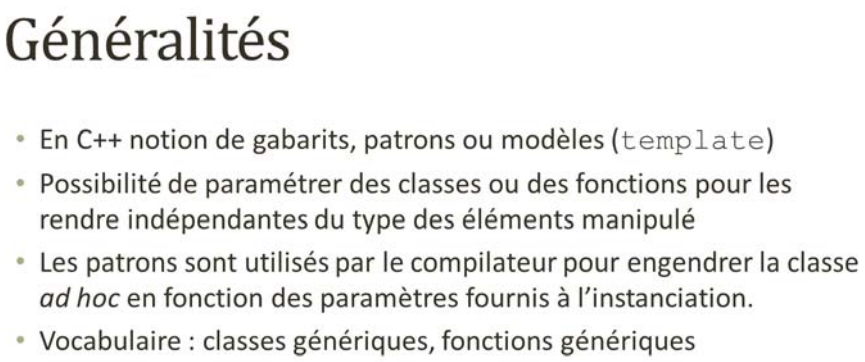
Chapitre 8. La généricité

Généralités

La généricité correspond au fait, pour un objet, de pouvoir être utilisé tel quel dans différents contextes.

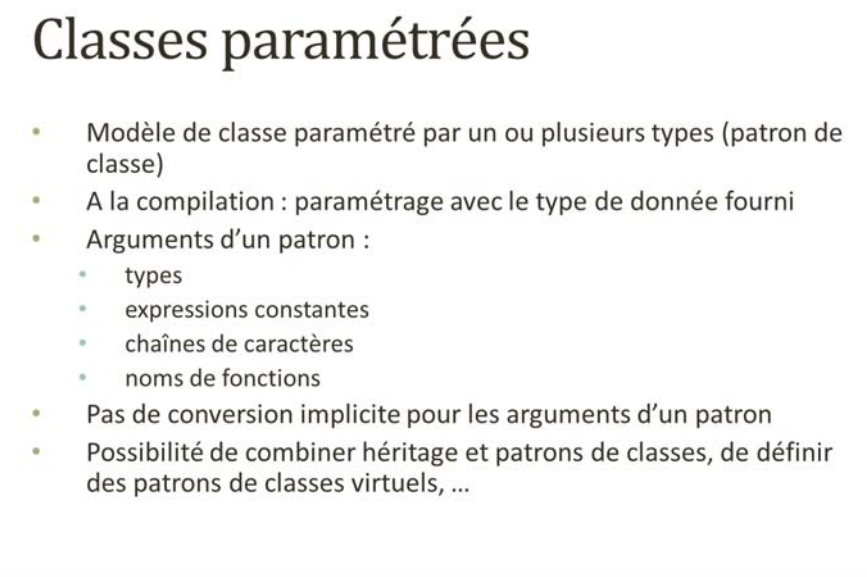


Slide 2

Les templates sont utilisés par le compilateur pour engendrer la classe *ad hoc* en fonction des paramètres effectifs fournis à l'instanciation.

On parle de classes ou de fonctions génériques. Le principe du patron de classe permet de mettre en œuvre la généricité.

Notion de classe paramétrée

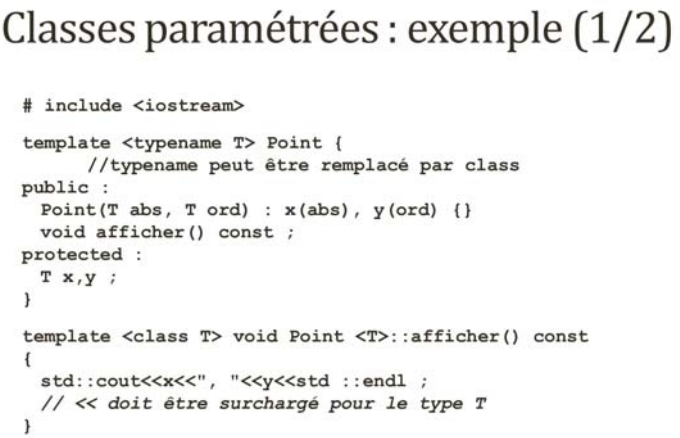


Slide 3

Après la compilation, il ne reste aucune trace du patron dans le programme exécutable, il ressemble exactement à un programme "écrit à la main" sur-mesure.
Les arguments d'un patron ne sont pas forcément des types, il peut s'agir d'expressions, de fonctions, de chaînes de caractères, etc…

Exemple de classe paramétrable

La classe Point que nous avons développée dans les chapitres antérieurs peut être paramétrée par le type des coordonnées du point.
Le type T, qui matérialise le type des coordonnées est ensuite utilisé dans la classe comme n'importe quel autre type (attention toutefois au fait qu'on suppose que certains opérateurs sont définis pour T).



Slide 4

Classes paramétrées : exemple (2/2)

```
int main()
{
  Point<int> p1(1,2) ;     // instanciation de Point
                           // avec le type entier
  Point<double> p2(1.5,2.3) ;
  p1.afficher() ; // 2 entiers
  p2.afficher() ; // 2 réels

  return 0 ;
}
```

slide 5

Dans la fonction principale, on instancie la classe Point avec le type entier ou tout autre type pertinent (possible plusieurs fois dans un même programme).
Remarque : on pourrait (ce serait même judicieux) surcharger l'opérateur << pour la classe Point, ce qui éviterait d'utiliser la méthode Affiche().
On pourrait alors écrire : **`cout<<p1<<","<<p2;`**

Pour manipuler les patrons de classe, on peut par exemple utiliser des fonctions paramétrées.

Fonctions paramétrées

- Remplacent les macro-instructions du C (`#define`)
- Naturel pour la manipulation de patrons de classes
- Possibilité de manipuler des conteneurs définis comme des patrons de classes (cf. la STL)

Slide 6

Exemples de fonctions paramétrées

Fonctions paramétrées : exemples 1/2

```
# include <iostream>

template <class T> T max (T a, T b) {return a<b ?b :a ;}
// a et b sont de type identique, T

int main()
{
  std::cout <<max(1,3) <<std ::endl ;
      // instantiation de int max(int, int)
  std::cout <<max('a','c') <<std ::endl ;
      // instantiation de char max(char, char)
  std::cout <<max(1,'c') <<std ::endl ;
      // ERREUR : ambiguïté T = char ou int ?
}
```

Slide 7

La fonction proposée ici est une fonction max paramétrée, comparant deux objets de type inconnu mais identique. On peut donc ensuite l'utiliser avec différents types passés en paramètres.
NB : on suppose que l'opérateur < est défini pour le type T.

Fonctions paramétrées, exemples 2/2

- Implémentez un patron de classe pour une pile générique. il contiendra en attributs :
  - le nombre d'élément maximal accepté pour la pile (un entier)
  - un vecteur stockant les éléments
- Proposez deux méthodes
  - bool *peutAjouter*()
  - void *ajouterElement*(X element)

Slide 8

⇒ exercice pour s'entraîner.

```
#include <iostream>
#include <vector>

using namespace std;

template <typename P> class Pile {
	unsigned int nbElementMaximum;
	vector<P> contenu;
	public:
		Pile(int nbMax, P element){nbElementMaximum = nbMax;
contenu.push_back(element);};
		void ajouterElement(P);
		bool peutAjouter();
};

template <typename P> bool Pile<P>::peutAjouter() {
	return contenu.size() < nbElementMaximum;	}

template <typename P> void Pile<P>::ajouterElement(P element) {
	if (this->peutAjouter()) {
		contenu.push_back(element);
		cout << "La pile contient " << contenu.size() << " elements" << endl;
	}
	else {
		cout << "Impossible d'ajouter un autre élément" << endl;
	}
}
```

Autre exemple avec la classe Point

```
template <typename T> class Point {
	T x,y ;
	public :
		Point(T abs, T ord) : x(abs), y(ord) {}
		void afficher() const ;
};

int main() {
Point<double> point1(3.0,4.0);
Pile<Point<double> > pileDouble(3.0, point1); // pile de points d'une taille max de 3 avec un
premier élément Point1 dedans
	// Danger : ne pas écrire Pile<Point<double>> sinon message d'erreur  error: '>>' should be
'> >' within a nested template argument list
pileEntier.ajouterElement(point1);
return 0;}
```