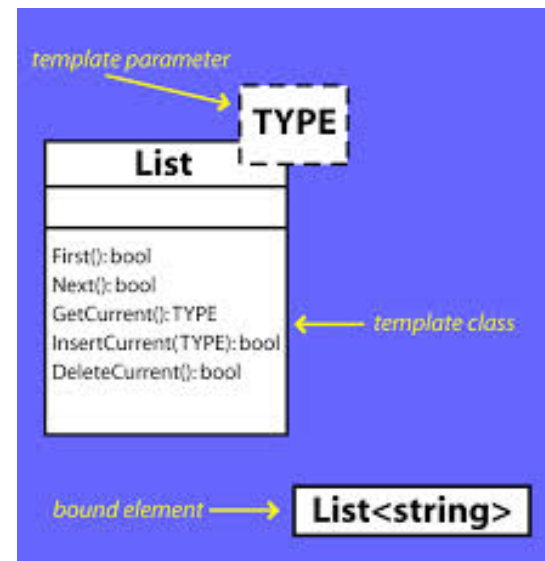
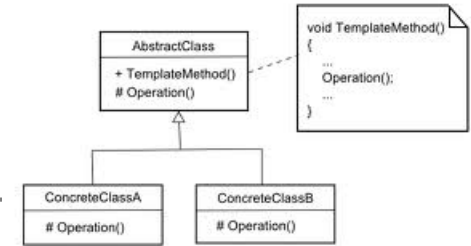


Programmation orientée Objet et C++ pour Eai

Les modèles (template ou patron) et la généricité

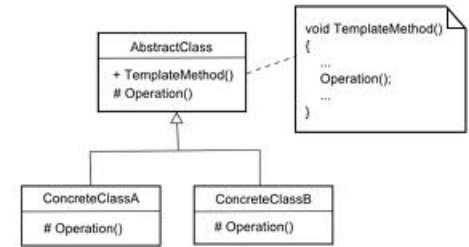


Fonctions génériques: "Template"



- Comment **éviter de répéter** le code pour **chaque type** ?
- Possibilités de « **surcharge** », mais **oblige la réécriture du code** pour tous les types différents
- Possibilités avec les **macros** (dangereux, à éviter)
- Les « **template** » permettent de créer des **fonctions génériques** (aussi appelé **modèle ou patron**)

Fonctions génériques - Syntaxe



- La définition d'une fonction générique est précédée du mot réservé « **template** »
- suivi d'un/plusieurs **type de données génériques** placé entre "<>"
- précédés du mot réservé "**class**" ou avec la nouvelle norme "**typename**" (mieux adapté, car définit un paramètre de type)

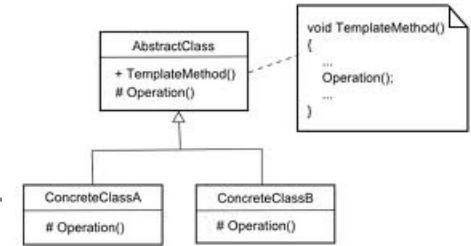
```

template <class Ttype>
ret-type func-name(parameter list)
OU
template <typename Ttype>
ret-type func-name(parameter list)
{
    // body of function
}
  
```

Attention: ici, le mot réservé **class** n'est pas utilisé en tant que classe similaire à structure



Fonctions génériques – exemple 1/2



fonction *maximum* de deux quantités.

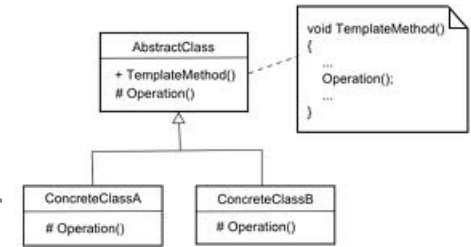
Par exemple: int, double, char, objets, etc.

```
template <typename T>           // déclare le modèle et le paramètre
T max(const T& gauche, const T& droite) // La Fonction qui est paramétrée
{
    return ((gauche > droite) ? gauche : droite);
}
```

Dans la fonction ci-dessus, *T* représente le type et le compilateur remplacera chaque *T* par le type choisi (int, double, char, etc.)



Fonctions génériques – exemple 2/2



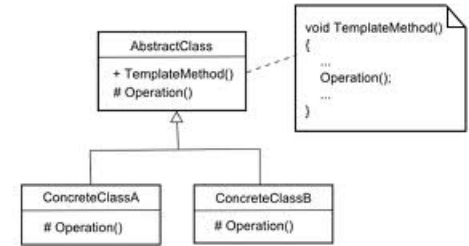
```
int main(void)
{
    int    i1 = 77777;
    int    i2 = 776;
    double d1 = 9999.99;
    double d2 = 33333.3;
    cout << max(i1, i2) << endl;
    cout << max(d1, d2) << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Pour les fonctions,
**l'instanciation se fera
automatiquement** par un
simple appel

The screenshot shows a Windows command prompt window with the title bar "D:\Enseignement\ProgOO\Cours\...". The window contains the following text:

```
77777
33333.3
Appuyez sur une touche pour continuer...
```

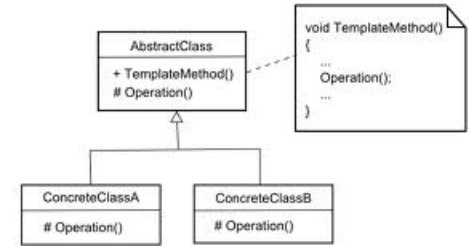
Fonctions génériques - description



- Chaque fois que le compilateur rencontrera une **fonction** template **avec un autre type**, il « **fabriquera** » (on dit aussi « instanciera ») automatiquement la fonction (e.g. *max*) avec des arguments ce type.
- Le **modèle *max*** peut **être utilisé** pour des arguments de **n'importe quel type prédéfini** (short, char, double, int *, char *, int * *, etc.) ou d'un type défini par l'utilisateur (structure ou classe, **si l'opérateur > existe** dans cette classe).
- *Si n et p sont de type int, un appel tel que **max (&n, &p)** conduit le compilateur à instancier une fonction **int * max (int *, int *)**.*

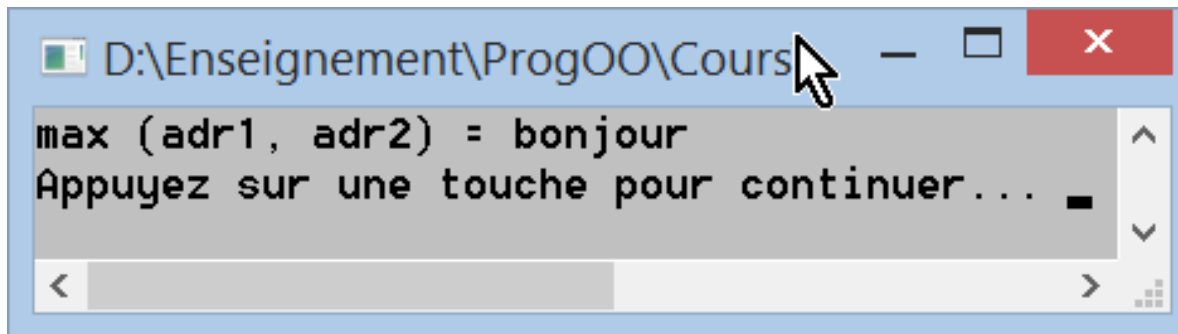


Fonctions génériques - exemple



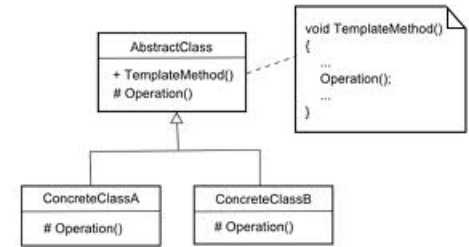
Utilisons le modèle *max* pour fabriquer une fonction avec des chaînes de style C

```
int main(void)
{
    char * adr1 = "monsieur", *adr2 = "bonjour";
    cout << "max (adr1, adr2) = " << max(adr1, adr2) << endl;
    return EXIT_SUCCESS;
}
```



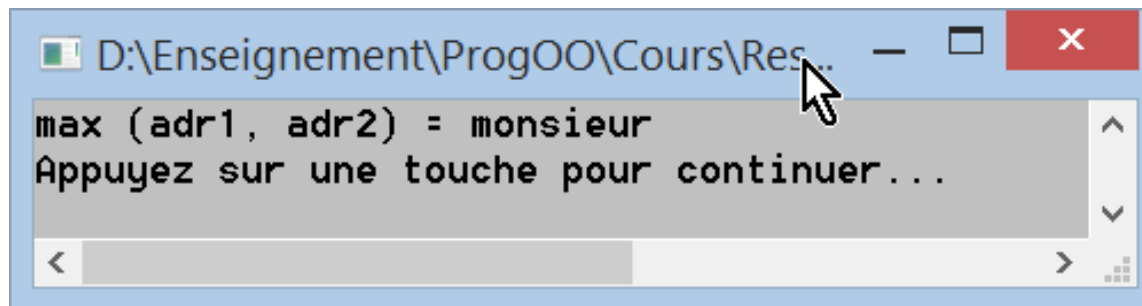
Ici, on va comparer des *adresses* et non des "string" → NOK

Fonctions génériques - exemple



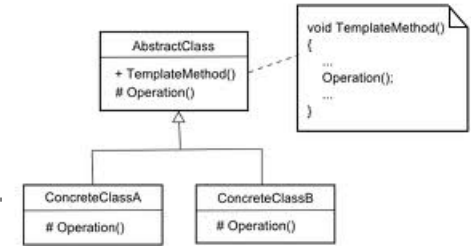
- Utilisons le modèle *max* pour fabriquer une fonction avec des *string*

```
int main(void)
{
    string str1 = "monsieur", str2 = "bonjour";
    cout << "max (adr1, adr2) = " << max(str1, str2) << endl;
    return EXIT_SUCCESS;
}
```



- Ici, on compare des *string* donc résultat OK

Fonctions génériques – multi-générique



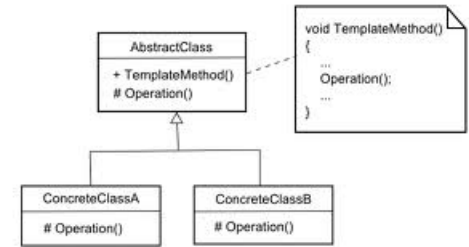
Si une fonction comporte plusieurs paramètres de généricité

```
template <class T1, class T2>
    void f (T1 val1, T2 val2)
{
    ...
}
```

- **Chaque type** donné comme paramètre générique, doit **apparaître au moins une fois** dans les **paramètres formels**
- **Chaque type** doit **être précédé** par le mot réservé **class** ou **typename** (et séparé par une virgule)



Fonctions génériques – multi-générique

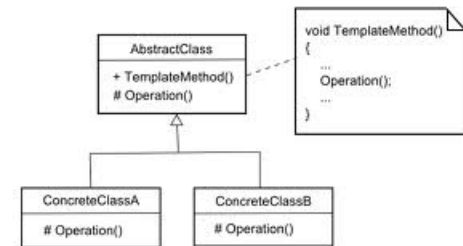


Les **paramètres de types** peuvent se trouver à **n'importe quel endroit** :

- dans les paramètres formels
- dans des déclarations de variables locales
- dans les instructions exécutables

```
template <class T, class U> fct(T a, T * b, U c)
{
    T x; // variable locale x de type T
    U * adr; // variable locale adr de type U *
    ...
    adr = new T[10]; // allocation tableau de 10 éléments de type T
    ...
    n = sizeof(T);
    ...
}
```

Fonctions génériques – avec types différents

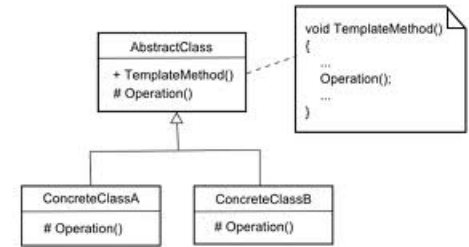


- Il doit y avoir correspondance absolue des types
- On peut utiliser le modèle *max* que pour des appels dans lesquels les deux arguments ont le même type
- Mais on peut forcer un type:

```
int    i1 = 77777;
double d2 = 33333.3;
int n; char c; unsigned int q;
const int ci1 = 10, ci2 = 12;
```

```
max<int>(c, n); // force la conversion de c en int
max<char>(q, n); // force la conversion de q et de n en char
max<int>(i1, d2); // force la conversion de d2 en int
```

Fonctions génériques - initialisation

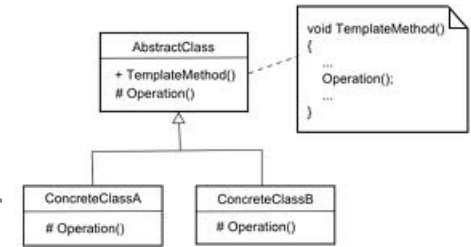


- Un **paramètre** de **type** est susceptible de correspondre tantôt à un **type standard**, tantôt à un **type classe**.
- Problème si l'on doit transmettre un ou plusieurs arguments à son constructeur !
Voici la solution:

```
template <class T> fct(T a)
{
    T x(3); // x est un objet local de type T qu'on construit
           // en transmettant la valeur 3 à son constructeur
           // ...
}
```

- Si x est un *int*, le compilateur C++ l'interprète comme ceci:
int x(3); // est interprété comme: int x = 3;

Fonctions génériques – surcharge



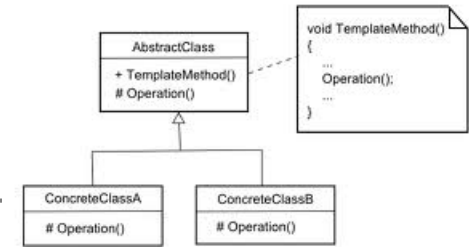
Une **fonction générique** peut être **surchargée**!

```
template <class t>
    bool fct(t val1)...
```

```
template <class t>
    bool fct(t val1, t val2)...
```

```
template <class t>
    bool fct(t val1, t val2, t val3)...
```

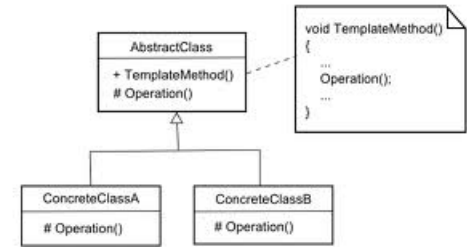
Fonctions génériques – spécialisation



- Nous avons vu que le template *max*, pour fabriquer une fonction avec des chaînes de style C ne fonctionne PAS !
- La notion de spécialisation offre une solution à ce problème
- Il existe deux types de spécialisation :
 1. Les spécialisations totales n'ont aucun paramètre *template* (ils ont tous une valeur bien déterminée)
 2. Les spécialisations partielles, pour lesquelles seuls quelques paramètres *template* ont une valeur fixée
- Le préfixe *template<>* indique au compilateur que ce qui suit est une spécialisation totale
- La diapo suivante montre la spécialisation totale de la fonction *max* pour les chaînes de caractères



Fonctions génériques – spécialisation totale



```

template <class T> T max(T a, T b) // modèle max
{
    if (a > b) return a; else return b;
}

```

```

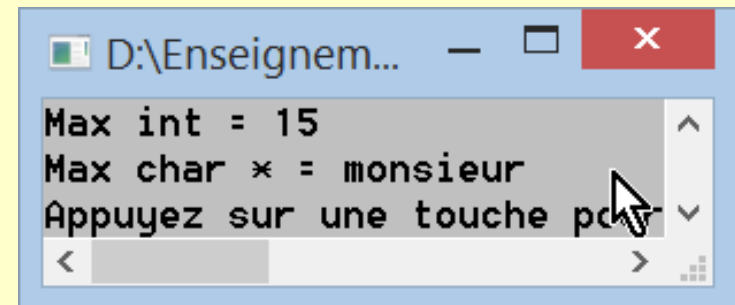
template <> char * max(char * cha, char * chb) // spécialisation p. chaînes
{
    if (strcmp(cha, chb) > 0) return cha;
    else return chb;
}

```

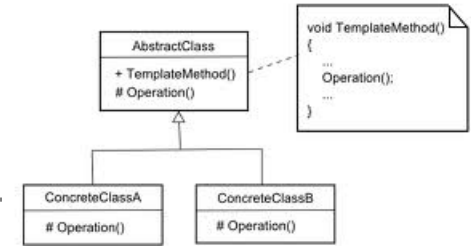
```

void main(void)
{
    int n = 12, p = 15;
    char * adr1 = "bonjour", *adr2 = "monsieur";
    cout << "Max int = " << max(n, p) << endl; // modèle int max (int, int)
    // fonction spécialisée char* max (char*, char*)
    cout << "Max char * = " << max(adr1, adr2) << endl;
}

```

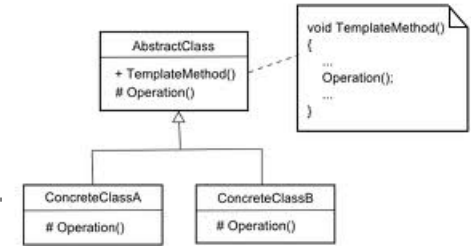


Fonctions génériques – spécialisation partielle



- Les spécialisations partielles permettent de **définir certains paramètres** et de **garder d'autres paramètres indéfinis**.
- Comme pour les spécialisations totales, il est nécessaire de déclarer la liste des paramètres template utilisés par la spécialisation
- Cependant, à la différence des spécialisations totales, cette liste **ne peut plus être vide**

Fonctions génériques – spécialisation partielle



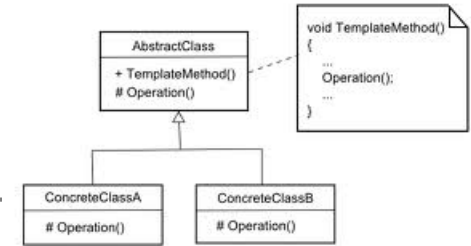
- Ces possibilités de spécialisation partielle s'avèrent très utiles dans les situations suivantes :
- Traitement particulier pour un pointeur, en spécialisant partiellement T en T *

```

template <class T> void f(T t) // modèle I
{
    ....
}
template <class T> void f(T * t) // modèle II
{
    ....
}
....
int n; int *adc;
f(n);    // f(int) en utilisant modèle I avec T = int
f(adc);  // f(int *) en utilisant modèle II avec T = int* car il est
          // plus spécialisé que modèle I (avec T = int)
  
```

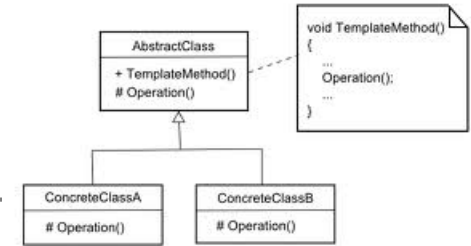


Stratégie de développement de template



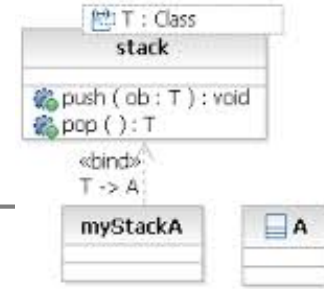
1. Développer normalement la fonction
 - Utiliser des types prédéfinis
2. Debugger la fonction
3. Puis convertir la fonction en modèle
 - Remplacez les noms des types avec le nom des types paramètres
4. Avantages:
 - Plus facile de résoudre un algorithme avec un cas concret
 - Permet de traiter l'algorithme, sans se soucier de la syntaxe template
5. Cette méthode s'applique aussi pour les classes template

Types inapproprié dans les "template"



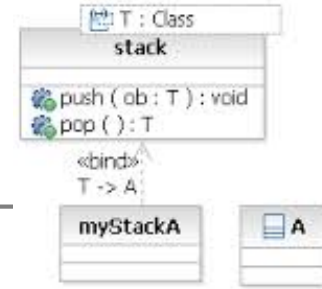
1. On peut utiliser n'importe quel type qui fait sens dans les "template"
 - Le code doit se comporter de manière appropriée
2. Par exemple, **max()** fonction template
 - Vous ne pouvez **pas utiliser un type** pour lequel **l'opérateur d'affectation n'est pas défini**
 - Exemple:
 - Un tableau: `int a [10], B [10];`
 - `max (a, b);`
 - **Les tableaux ne peuvent pas être "assignés"!**

Classes génériques "Template"



- Les **classes génériques** encapsulent des **opérations** qui ne sont **PAS spécifiques** à un **type de données** particulier. Par exemple:
 - les listes liées
 - les piles
 - les files d'attente
 - les arborescences
 - etc.

Classes génériques - Syntaxe

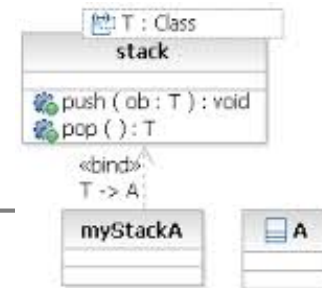


```
template <class Ttype>
class class-name
OU
template <typename Ttype>
class class-name
{
    ...
};
```

- Une classe générique peut comporter **plusieurs paramètres**, séparés par des virgules
- Contrairement aux fonctions génériques, les **paramètres** génériques d'une classe générique peuvent avoir une **valeur par défaut**

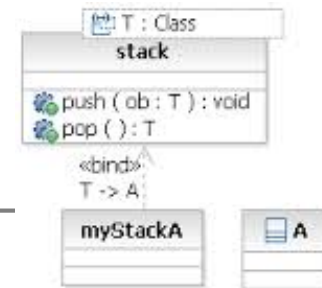
```
template < typename T=int, int i=10>
class class-name ...
```

Classes génériques – Définition - exemple



```
template<class T>
class Pair
{
public:
    Pair();
    Pair(T firstVal, T secondVal);
    void setFirst(T newVal);
    void setSecond(T newVal);
    T getFirst() const;
    T getSecond() const;
private:
    T first;
    T second;
};
```

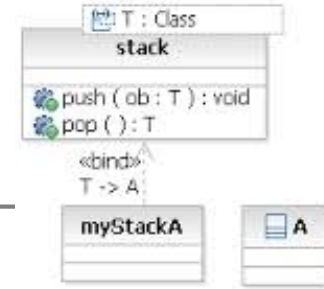
Classes génériques – Déclaration - exemple



```
template<class T>
Pair<T>::Pair(T firstVal, T secondVal)
{
    first = firstVal;
    second = secondVal;
}

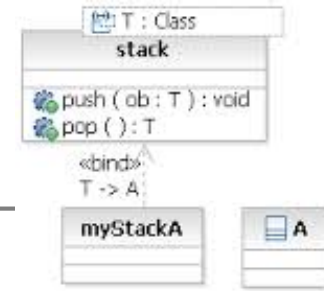
template<class T>
void Pair<T>::setFirst(T newVal)
{
    first = newVal;
}
```

Classes génériques – déclaration – méthodes



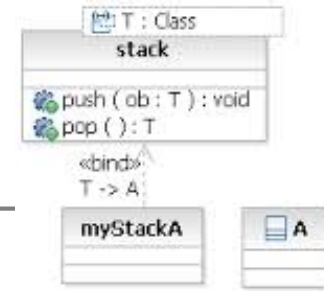
- Dans les déclarations des méthodes
 - Chaque déclaration est elle-même un template
 - Il faut préfixer chaque méthode avec le template e.g.:
template<class T>
 - Puis encore mettre la classe et le type, e.g.:
void Pair<T>::

Classes génériques – Instanciation -exemple



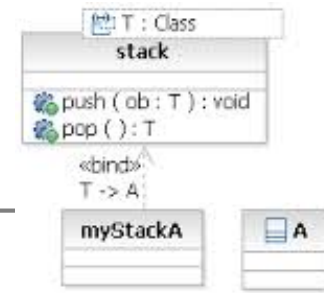
- Les objets de la classe "Pair" ont des paramètres de type T
- On peut déclarer des objets de type:
Pair<int> score;
Pair<char> siege;
 - Ces objets s'utilisent comme n'importe quel autre objet
- Exemples :
score.setFirst(3);
score.setSecond(0);

Classes génériques - informations



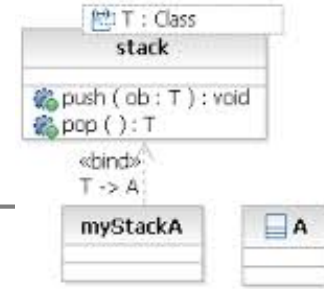
- Les **paramètres de type** peuvent correspondre à n'importe quel type **simple** ou **structuré**, voire même une **classe générique**
- Contrairement aux fonctions génériques, il n'y a **PAS d'instanciation implicite**
- Sauf en cas d'utilisation des valeurs par défaut, il faudra toujours **passer en paramètres** effectifs les **types ou valeurs** souhaitées

Classes génériques - Déclaration



- La **définition** d'une méthode générique (son corps) ne peut **pas être compilée** en tant que tel et donc donner du code objet
- En d'autres termes, la **définition** d'une **méthode générique** doit être vue en fait **comme une déclaration**
- Corrolaire:
La **définition** d'une **méthode générique** se **place** généralement **dans le même fichier ".h"** que la **déclaration de la classe**
Il est également possible de mettre la **définition** d'une **méthode générique** dans un **autre fichier ".h"**

Classes "Template" - Exemple



- Nous allons voir un exemple classique de classe générique: celui de la **classe Pile**

Classes génériques – Exemple "Pile" (fichier *.h) 1/2

```
// Déclaration de la classe
// template
#ifndef __PILE_GEN__
#define __PILE_GEN__

template <typename T>
class Pile
{
public:
    Pile(int taille = 128);
    ~Pile(void);
    bool empty(void);
    void push(const T& val);
    void pop();
    const T& top(void) const;
    T& top(void);

private:
    int nbElements;
    // adresse tableau dynamique
    T *tab;
    // adresse position courante
    T *pos;
};
```

```
// Définitions des méthodes de la classe
// (inclus dans fichier *.h !!!)
template <typename T>
Pile<T>::Pile(int taille = 128)
{
    tab = new T[taille];
    pos = tab - 1;
    nbElements = 0;
}

template <typename T>
Pile<T>::~~Pile(void)
{
    delete[] tab;
}

template <typename T>
bool Pile<T>::empty(void)
{
    return (nbElements == 0);
}
...
```

Attention à la syntaxe des méthodes

1. Mettre template + typename
2. Mettre type de retour
3. Mettre le nom de la classe et type générique



Classes génériques – Exemple "Pile" (fichier *.h) 2/2

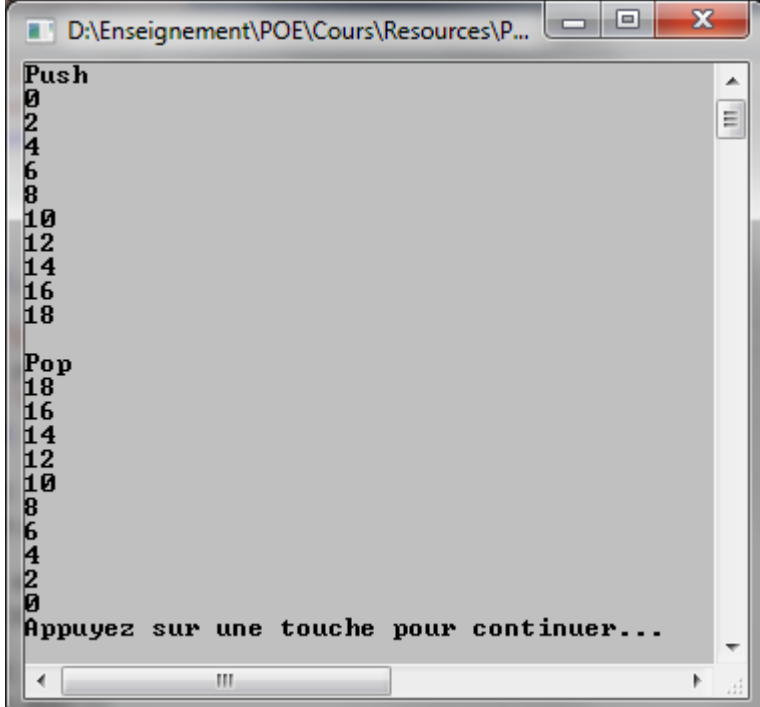
```
...
template <typename T>
void Pile<T>::push(const T& val)
{
    ++pos;
    *pos = val;
    ++nbElements;
}
template <typename T>
void Pile<T>::pop()
{
    --pos;
    --nbElements;
}
template <typename T>
const T& Pile<T>::top(void) const
{
    return *pos;
}
template <typename T> T& Pile<T>::top(void)
{
    return *pos;
}
#endif
```



Classes génériques – Exemple "Pile" (fichier *.cpp)

```
#include "pile_gen.h"
#include <iostream>
using namespace std;

typedef Pile<int> PileInt;
int main(void)
{
    // Pile<int> pile; // ou...
    PileInt pile;
    cout << "Push" << endl;
    for (int i = 0; i < 10; i++)
    {
        cout << 2 * i << endl;
        pile.push(2 * i);
    }
    cout << endl << "Pop" << endl;
    while (!pile.empty())
    {
        cout << pile.top() << endl;
        pile.pop();
    }
    system("PAUSE");
    return 0;
};
```



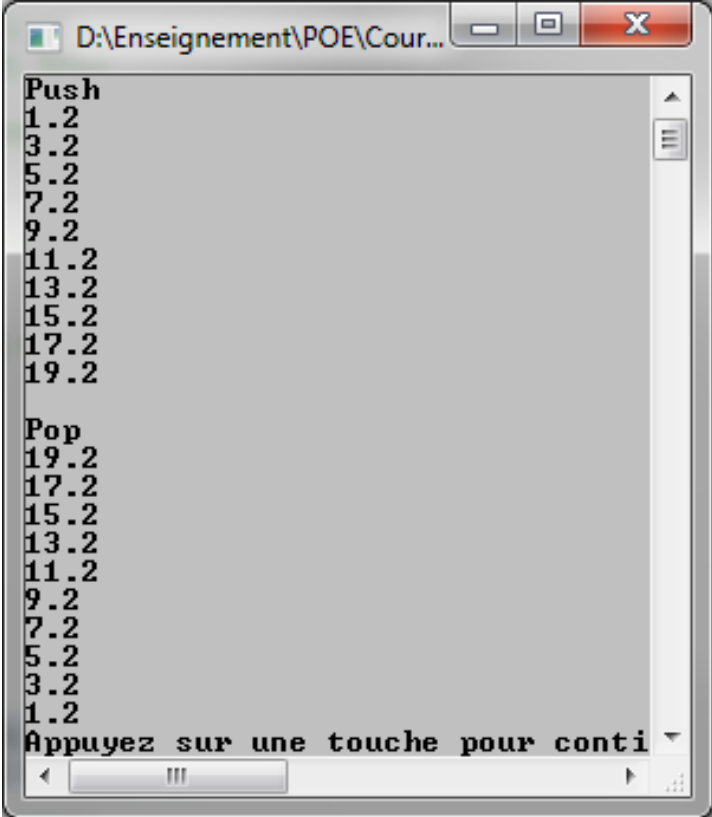
```
Push
0
2
4
6
8
10
12
14
16
18
Pop
18
16
14
12
10
8
6
4
2
0
Appuyez sur une touche pour continuer...
```

Classes génériques – Exemple "Pile" (fichier *.cpp)

```
#include "pile_gen.h"
#include <iostream>
using namespace std;

typedef Pile<double> PileDouble;

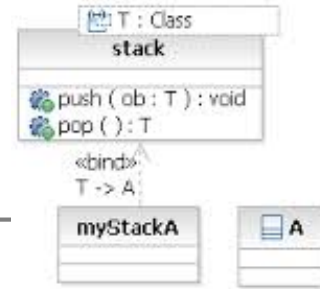
int main(void)
{
    PileDouble pile;
    cout << "Push" << endl;
    for (int i = 0.6; i < 10; i++)
    {
        cout << 2 * i << endl;
        pile.push(2 * i);
    }
    cout << endl << "Pop" << endl;
    while (!pile.empty())
    {
        cout << pile.top() << endl;
        pile.pop();
    }
    system("PAUSE");
    return 0;
};
```



```
Push
1.2
3.2
5.2
7.2
9.2
11.2
13.2
15.2
17.2
19.2

Pop
19.2
17.2
15.2
13.2
11.2
9.2
7.2
5.2
3.2
1.2
Appuyez sur une touche pour conti...
```


Fonctions membres template

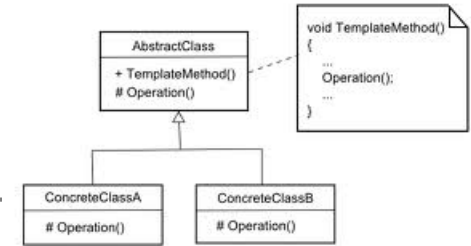


- Les **méthodes virtuelles** ne peuvent **pas être template**
- Les **méthodes** d'une classe **peuvent être template**, même si la **classe n'est pas template** (sauf le destructeur)

```
class A // Classe NON template
{
    int i; // Valeur de la classe.
public:
    template <class T>
    void add(T valeur);
};

template <class T> // Fonction template
void A::add(T valeur)
{
    i=i+((int) valeur); // Ajoute valeur à A::i.
    return ;
}
```

Classes génériques – spécialisation



- Nous avons vu qu'il était possible de « **spécialiser** » certaines fonctions d'un **modèle de fonctions**.
- Si la même possibilité existe pour les **modèles de classes**, elle prend toutefois un aspect légèrement différent, comme nous le verrons dans un exemple

Classes génériques – spécialisation –exemple (1/2)

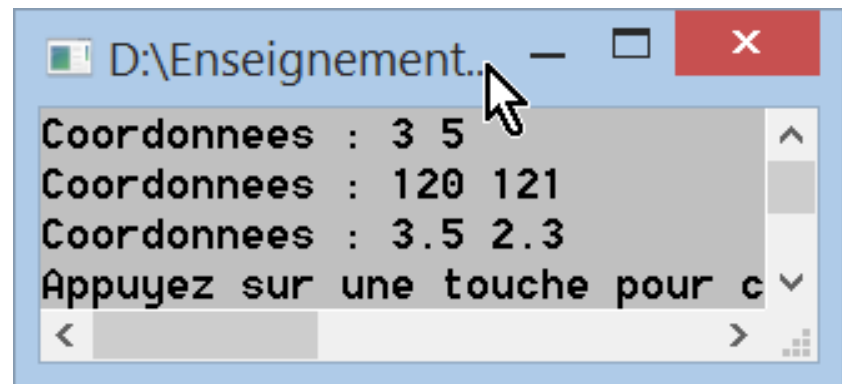
```
template <class T> class point
{
    T x; T y;
public:
    point(T abs = 0, T ord = 0)
    {
        x = abs; y = ord;
    }
    void affiche();
};
// définition de la fonction affiche
template <class T> void point<T>::affiche()
{
    cout << "Coordonnees : " << x << " " << y << "\n";
}

// ajout d'une fonction affiche spécialisée pour les caractères
void point<char>::affiche()
{
    cout << "Coordonnees : " << (int)x << " " << (int)y << "\n";
}
```



Classes génériques – spécialisation – exemple (2/2)

```
int main(void)
{
    system("COLOR 70");
    point <int> ai(3, 5); ai.affiche();
    point <char> ac('x', 'y'); ac.affiche();
    point <double> ad(3.5, 2.3); ad.affiche();
    system("PAUSE");
}
```



```
D:\Enseignement...
Coordonnees : 3 5
Coordonnees : 120 121
Coordonnees : 3.5 2.3
Appuyez sur une touche pour c
```

Vos questions



