Université de Nice-Sophia Antipolis
Polytech ELEC4

Mercredi 13 novembre 2019
Durée: 2h

C++ Templates
Travaux Dirigés – Séance n. 6

---

1 Objectif

Dans ce TD, nous verrons comment écrire des *méthodes* et des *classes génériques*, c'est-à-dire paramétrées sur le type des paramètres ou de leurs données. En C++, on parle de *templates*.

2 Méthodes génériques

Tout d'abord, nous souhaitons écrire une fonction qui renvoie le minimum de deux entiers. La programmation de cette fonction est très simple :

```
int minimum(int x, int y) {
  return x < y ? x : y;
}
```

Maintenant, nous voulons faire la même chose mais pour des chaînes de caractères :

```
string minimum(string x, string y) {
  return x < y ? x : y;
}
```

Si nous devons renvoyer le minimum d'objets d'autres types, il faudra écrire d'autres fonctions similaires, ce qui est fastidieux et alourdit considérablement le code. Ce qu'il faut faire, c'est écrire une *seule* fonction `minimum` paramétrée sur le type de ses paramètres et de son résultat, afin qu'elle puisse s'appliquer à des objets de types différents. La notion de *template* permet de faire cela en C++.

À l'aide de cette notion, on récrira la fonction `minimum` comme suit :

```
template <typename T>
const T& minimum(const T& x, const T& y) {
  return x < y ? x : y;
}
```

Dans cette déclaration, le type `T` représente n'importe quel type d'objet.

exercice 1) Dans le fichier `fctgen.cpp`, écrivez un programme qui déclare la fonction précédente, et qui, dans la fonction `main` écrit sur la sortie standard le minimum de deux entiers, de deux réels et de deux chaînes de caractères.

exercice 2) Affichez maintenant le minimum d'un entier et d'un réel. Que se passe-t-il ? Expliquez.

exercice 3) Une façon de régler le problème précédent est de spécialiser la fonction en spécifiant le type `T`. Testez les appels suivants : `minimum<double>(5.7, 13)` et `minimum<int>(5.7, 13)`.

---

exercice 4) Dans le fichier `rectangle.hpp`, déclarez la classe `rectangle` avec deux variables membres privées `largeur` et `longueur` de type **double**.

exercice 5) Ajoutez un constructeur pour initialiser ces deux variables membres.

exercice 6) Ajoutez la méthode `surface` qui renvoie la surface du rectangle courant. Le corps de cette méthode sera défini dans le fichier `rectangle.cpp`.

exercice 7) Dans la fonction `main` de votre fichier `fctgen.cpp`, déclarez deux variables `r1` et `r2` de type `rectangle`, initialisées, par exemple, à $(2, 3.1)$ et $(12.1, 0.43)$ et à l'aide de votre fonction générique `minimum`, affichez le minimum de ces deux rectangles :

```
std::cout << minimum(r1,r2) << std::endl;
```

Que se passe-t-il ? Expliquez.

exercice 8) Dans la classe `rectangle`, ajoutez la surcharge de l'opérateur inférieur `<` dont l'en-tête est **bool operator<(const** `rectangle &c`**) const**. Que se passe-t-il ? Expliquez.

exercice 9) Dans la classe `Rectangle` ajoutez la surcharge de l'opérateur `<<`. Testez votre programme.

3 Classes génériques

De même que nous avons défini des méthodes génériques, il est possible de déclarer des *classes génériques*.

La classe `PileChainee` que nous avons écrite dans le TD précédent ne permet de manipuler *que* des entiers puisque les valeurs des éléments de la pile sont de type **int**. Nous allons utiliser la notion de **template** pour paramétrer la classe sur le type des éléments qu'elle manipule. Dans la suite, nous implémenterons la pile à l'aide d'un tableau (et non pas avec une structure chaînée, comme dans le TD précédent) à l'aide de la classe `PileTableau`. Soit la déclaration suivante :

```
#pragma once

template <typename T>
class PileTableau {
private:
  int sp;           // l'indice du sommet de pile
  int max;          // nombre maximun d'éléments
  T *lesElements;   // tableau générique d'éléments de type T
};
```

exercice 10) Complétez cette classe avec un constructeur qui alloue *dynamiquement* un tableau de taille n (le paramètre qui a pour valeur par défaut 100) pour les éléments de la pile.

exercice 11) Ajoutez le destructeur `~PileTableau` pour supprimer les éléments du tableau.

exercice 12) Ajoutez le constructeur de copie et la surcharge de l'opérateur d'affectation ;

exercice 13) Puis, ajoutez les méthodes (classiques) génériques de manipulation de la pile suivantes :

```
bool estVide() const
void empiler(const T &x)
void depiler()
const T &sommet() const
```

Il est important de noter que les méthodes génériques précédentes ne peuvent être écrites dans un fichier `.cpp` séparé du fichier `.hpp`, car la généricité ne peut être traitée par la compilation séparée. L'implémentation des méthodes génériques devra donc être faite dans le fichier `.hpp`. En fait, à chaque instanciation, une nouvelle classe est créée avec le type effectif choisi.

exercice 14) Ajoutez à votre classe `PileTableau` la surcharge de l'opérateur `<<` de façon à pouvoir écrire tous les éléments d'une pile sur un `ostream`.

exercice 15) Dans le fichier `testpile.cpp`, écrivez la méthode `main` dans laquelle vous déclarerez trois piles, une de **double**, une de `string`, et la dernière de `rectangle`. Testez vos méthodes.

```
...
int main() {
    PileTableau<int> pi;
    PileTableau<string> ps;
    PileTableau<rectangle> pr;
    ....
    ....
    std::cout << pi << std::endl;
    std::cout << ps << std::endl;
    std::cout << pr << std::endl;

    return EXIT_SUCCESS;
}
```

exercice 16) Dans la fonction `main` précédente ajoutez la déclaration d'une variable `ppi` qui est une pile de piles d'entiers. Empilez sur `ppi` la pile `pi`, et affichez sur la sortie standard la pile `ppi`.

---

exercice 17) Reprenez votre classe `PileChainee` qui implémente une pile à l'aide d'une structure chaînée, et rendez-la générique.

exercice 18) Dans la fonction `main`, construisez et affichez :

1. une `PileChainee` de `PileTableau` de **int** ;
2. une `PileTableau` de `PileChainee` de `string`.