# Chapitre 5 : programmation générique, bibliothèque standard (STL)

Fondamentaux du C++

Master 2 Ingénierie mathématique
Sorbonne Université

Automne 2019

# Plan

# Modèles ("templates") de fonctions

- Permettent de programmer plusieurs fonctions d'un seul coup ("automatisation" de la surcharge de fonction) ;
- On crée une famille de fonction indexée par un ou plusieurs paramètres (entier ou type) ;
- Avec le mot-clé `template` ;
- Les paramètres doivent être connus à la compilation.

# Modèles ("templates") de fonctions

## Exemple

```
template <typename mon_type>
mon_type max(mon_type x, mon_type y)
{
   return x>y ? x : y;
}
```

```
max<int>(2,4);
max<double>(4.2,2.1);
max<rationnel>(rationnel(4,3),1);
```

# Modèles ("templates") de classes

## Exemple

```
template<int n, typename coeff>
class matrice
{
   coeff det(void);
   ...
};

template <int n, typename T>
T matrice<n,T>::det (void) // Définition de méthode
{
...
}
```

```
matrice<3,int> A;
matrice<6,double> B;
```

# Plan

# La bibliothèque standard (STL)

Les classes issues de la bibliothèque standard (STL, pour “standard template library”) sont signalées par le préfixe `std::`.

# Vecteurs

Les tableaux dynamiques/statiques sont avantageusement remplacés en C++ par les vecteurs.

## Les vecteurs

- Type `std::vector<typename>` ;
- Méthodes `push_back`, `pop_back`, `operator[]`, `size`... ;
- Efficacité :
  - Efficaces pour accéder à un élément, ajouter/enlever un élément à la fin ;
  - Inefficaces pour ajouter/enlever un élément à un endroit quelconque du tableau.

# Vecteurs

Les tableaux dynamiques/statiques sont avantageusement remplacés en C++ par les vecteurs.

## Les vecteurs

- Type `std::vector<typename>` ;
- Méthodes `push_back`, `pop_back`, `operator[]`, `size`... ;
- Efficacité :
  - Efficaces pour accéder à un élément, ajouter/enlever un élément à la fin ;
  - Inefficaces pour ajouter/enlever un élément à un endroit quelconque du tableau.

## Variante : deque

- Type `std::deque<typename>` ;
- Permet également d'ajouter efficacement un élément au début.

# Vecteurs

Les tableaux dynamiques/statiques sont avantageusement remplacés en C++ par les vecteurs.

## Les vecteurs

- Type `std::vector<typename>` ;
- Méthodes `push_back`, `pop_back`, `operator[]`, `size`... ;
- Efficacité :
  - Efficaces pour accéder à un élément, ajouter/enlever un élément à la fin ;
  - Inefficaces pour ajouter/enlever un élément à un endroit quelconque du tableau.

## Variante : `array` (en C++ 2011)

- Type `std::array<typename, int>` ;
- La taille est un paramètre template : mieux optimisé pour les tableaux de taille fixée.

# Listes

## Les listes

- Type `std::list<typename>` ;
- Méthodes `insert`, `erase`, `remove`... ;
- Efficacité :
  - Efficaces pour ajouter/enlever un élément, pour accéder au premier/dernier élément ;
  - Inefficaces pour accéder à un élément en une position quelconque.

# Ensembles

## Les ensembles

- Type `std::set<typename>` ;
- Le type `set` correspond à une structure de données où les éléments sont systématiquement triés.
- Méthodes `insert`, `erase`, `find`... ;
- Efficace pour chercher un élément ;

# Chaînes de caractères

## En C

- Chaînes de caractères = tableaux de caractères (`char*`) ;
- Exemple : `char *texte="Bonjour";`, tableau à 8 cases contenant `'B'`, `'o'`, `'n'`, `'j'`, `'o'`, `'u'`, `'r'` et `'\0'` ;
- `'\0'` (caractère de code ASCII 0) ne s'affiche pas mais marque la fin de la chaîne ;
- Peu pratique.

## En C++

- Classe `std::string` ;
- Méthodes `size`, `find`, `operator+`, `operator[]`,... ;
- Syntaxe plus naturelle ;
- Méthode `data` pour la conversion en `char*`.

# Itérateurs

Les conteneurs `vector`, `list`, etc. peuvent tous être parcourus à l'aide d'itérateurs.

```
for (std::list<int>::iterator I=L.begin() ; I!=L.end() ;
I++)
{
    *I;
    ...
}
```

# Aléatoire

La bibliothèque `random` (en C++ 2011) définit des classes permettant de simuler des variables aléatoires de loi donnée.

```
std::mt19937 gen(98765);// Générateur avec sa graine

// On déclare deux variables aléatoires
std::normal_distribution<double> X(0,1);
std::poisson_distribution<int> Y(1);

double a = X(gen); // Une réalisation de X
a += Y(gen); // Une réalisation de Y
```

# Aléatoire

Lois discrètes :

- `std::uniform_distribution<int>`
- `std::bernoulli_distribution<int>`
- `std::geometric_distribution<int>`
- `std::poisson_distribution<int>`
- `std::binomial_distribution<int>`

Lois à densité :

- `std::uniform_real_distribution<double>`
- `std::normal_distribution<double>`
- `std::exponential_distribution<double>`

Changer le paramètre `double` ou `int` permet de changer la précision (`long long`, `long double`).

# Aléatoire

On peut également simuler un aléa qui diffère à chaque exécution :

```
std::random_device rd;// Aléa issu de sources ‘‘physiques’’
std::mt19937 gen(rd());// Générateur avec une graine
aléatoire

std::normal_distribution<double> X(0,1);
X(gen); // Une réalisation de X
X(rd);// Marche aussi, mais plus lent
```

# Flux

Les flux d'entrée/sortie sont aussi des classes définies dans la bibliothèque standard :

```
std::ofstream flux("fichier1.dat"); // Constructeur
flux << "Bonjour\n"; // Surcharge de l'opérateur <<
flux.close(); // Appel de la méthode "close"
flux.open("fichier2.dat") // Appel de la méthode "open"
```