



## Programmation C++11/17

Dr. Eng. Chiheb Ameer ABID

[/in/chiheb-ameur-abid](#)  
chiheb.abid@gmail.com

## Programmation générique

### Programmation générique : introduction

- ↳ L'idée de base est de passer les types de données comme paramètres pour décrire des traitements très généraux
  - ☞ Il s'agit d'un niveau d'abstraction supplémentaire
- ↳ De tels modèles de classes/fonctions s'appellent aussi classes/fonctions génériques ou patrons, ou encore templates
- ↳ Les templates permettent d'écrire du code générique
  - ☞ Écrire une famille de fonctions ou de classes qui ne diffèrent que par la valeur de leurs paramètres
  - ☞ Donc, l'utilisation des templates permet de paramétrer le type de données manipulées

## Plan

- 1 Les templates avec C++11/C++17
  - Instanciation implicite vs explicite
  - Surchage des fonctions templates
  - Spécialisation partielle/totale des templates
  - Modèles variadiques
  - Les types locaux et non nommés comme arguments template
- 2 Nouveautés dans la STL

## Programmation générique

### Déclaration d'un modèle

- ↳ La déclaration d'un modèle

```
template <class|typename nom[=type] [, class|typename nom[=type][...]>
```

- ☞ nom est le nom que l'on donne au type générique dans la déclaration
- ☞ class a ici exactement la signification de "type". Il peut être remplacé par le mot clé typename

- ↳ Après la déclaration d'un ou de plusieurs paramètres template suit en général la déclaration ou la définition d'une fonction ou d'une classe template

```
1 // Fonction template
2 template<class T> T mini(T a, T b);
3
4 // Classe template
5 template<class T> class Array {
6     ...
7     T a ;
8     ...
9 };
```

## Programmation générique

### Exemple : fonction template

```

1 #include <iostream>
2
3 template<class T> T mini(T a, T b) {
4     if(a < b) return a;
5     else return b;
6 }
7
8 template<class T> T mini(T a, T b, T c) {
9     return mini(mini(a, b), c);
10 }
11
12 int main() {
13     int n=12, p=15, q=2;
14     float x=3.5, y=4.25, z=0.25;
15     std::cout << "mini(n, p) -> " << mini(n, p) << std::endl; // implicite
16     std::cout << "mini(n, p, q) -> " << mini(n, p, q) << std::endl;
17     std::cout << "mini(x, y) -> " << mini(x, y) << std::endl;
18     std::cout << "mini(x, y, z) -> " << mini(x, y, z) << std::endl;
19     std::cout << "mini(n, p) -> " << mini<float>(n, q) << std::endl; // explicite
20     return 0;
21 }
    
```



## Déduction de la valeur de retour d'une fonction modèle

## Déduction de la valeur de retour d'une fonction modèle

### Déduction de la valeur de retour d'une fonction modèle

La valeur de retour d'une fonction modèle peut dépendre de types des paramètres

```

1 template <typename T1, typename T2>
2 ??? larger(T1 a, T2 b) {
3     return a > b ? a : b;
4 }
    
```

Depuis C++14, il est possible de déduire le type de retour d'une fonction

- ① Utilisation de `auto`
- ② Utilisation de `decltype`



## Déduction de la valeur de retour d'une fonction modèle

### Déduction de la valeur de retour d'une fonction modèle

- ① Déduction par le compilateur en utilisant le mot clé `auto`
- ☒ Supprime les qualificateurs de référence et de `const`

```

1 template <typename T1, typename T2>
2 auto larger(T1 a, T2 b) {
3     return a > b ? a : b;
4 }
5
6 int small_int {10};
7 std::cout << "Larger of " << small_int << " and 9.6 is " << larger(small_int, 9.6)
8     << std::endl; // deduced return type: double
9 std::string a_string {"A"};
10 std::cout << R("Larger of ") << a_string << R(" and \"Z\" is ") << larger(a_string,
11     "Z") << " " << std::endl; // deduced return type: std::string
    
```



### Déduction de la valeur de retour d'une fonction modèle

- ② Introduits en C++14, les mots-clés `decltype` et les types de retour de fin (trailing return types) sont utilisés

```

1 template <typename T1, typename T2>
2 auto larger(T1 a, T2 b) -> decltype(a > b ? a : b) {
3     return a > b ? a : b;
4 }
    
```

Une autre écriture équivalente

```

1 template <typename T1, typename T2>
2 decltype(auto) larger(T1 a, T2 b) {
3     return a > b ? a : b;
4 }
    
```



## Plan

### 1 Les templates avec C++11/C++17

- Instanciation implicite vs explicite
- Surcharge des fonctions templates
- Spécialisation partielle/totale des templates
- Modèles variadiques
- Les types locaux et non nommés comme arguments template

### 2 Nouveautés dans la STL

- `std::optional`, `std::variant` et `std::any`
- Itérateurs
- Nouveaux conteneurs

## Instanciation implicite vs explicite

### Instanciation des classes templates

- ↳ Lorsqu'une classe template est utilisée avec un type spécifique, le compilateur génère une instanciation de la classe pour ce type
  - ▀ Cette instanciation est un code concret qui peut être utilisé par le programme.
- ↳ L'instanciation d'une classe template s'effectue de deux manières
  - 1 Instanciation implicite, appelée aussi sélective
  - 2 Instanciation explicite

## Instanciation implicite vs explicite

## Instanciation implicite vs explicite

### 1 Instanciation implicite (sélective)

↳ Lors de l'instanciation implicite, le compilateur génère le code pour :

- ▀ Toutes les fonctions membres virtuelles, même si elles ne sont pas utilisées explicitement
- ▀ Les fonctions membres non-virtuelles qui sont réellement appelées dans le code

```
nom_classe<arguments> objet;
```

- ✓ Réduction de la taille du code généré en ne générant que le code nécessaire
- ✓ Le compilateur peut mieux optimiser le code généré lorsqu'il sait quelles fonctions sont réellement utilisées
- ✗ Masquer des erreurs de syntaxe dans les fonctions membres inutilisées

### Exemple d'instanciation implicite (sélective) (1/2)

```
1 template <typename T>
2 class Grid {
3     std::vector<T> m_cells;
4     std::size_t m_width { 0 }, m_height { 0 };
5     public:
6     explicit Grid(std::size_t width = 5 , std::size_t height = 5);
7     virtual Grid() = default;
8     // Explicitly default a copy constructor and copy assignment operator.
9     Grid(const Grid& src) = default;
10    Grid& operator=(const Grid& rhs) = default;
11    // Explicitly default a move constructor and move assignment operator.
12    Grid(Grid&& src) = default;
13    Grid& operator=(Grid&& rhs) = default;
14    std::size_t getHeight() const { return m_height; }
15    std::size_t getWidth() const { return m_width; }
16    T at(size_t x, size_t &y);
17 };
```

## Instanciation implicite vs explicite

### Exemple d'instanciation implicite (sélective) (2/2)

↳ Soit l'instanciation implicite suivante :

```
1 Grid<int> myIntGrid;  
2 myIntGrid.at(0, 0) = 10;
```

↳ Le compilateur génère le code pour uniquement les méthodes suivantes :

- ↳ Constructeur avec 0 argument
- ↳ Destructeur
- ↳ La méthode `at()`

↳ Le code pour les autres méthodes n'est pas généré

## Instanciation implicite vs explicite

### 2 Instanciation explicite

↳ L'instanciation explicite d'une classe template force la génération du tout le code

```
template class nom_classe<arguments>;
```

- ✓ Détection proactive des erreurs de syntaxe avant leur apparition en production
- ✓ Amélioration de la qualité et de la robustesse du code
- ✓ Gain de temps lors du débogage en évitant les surprises liées aux erreurs cachées
- ✗ Code plus verbeux
- ✗ Augmentation potentielle de la taille du code final si de nombreuses instanciations sont nécessaires

## Instanciation implicite vs explicite

### Exemple d'instanciation explicite

↳ Instanciation explicite de la classe `Grid<T>`

```
template class Grid<string>;
```

↳ Le compilateur génère le code pour toutes les méthodes, même ceux qui ne sont pas utilisées

- ↳ Permet d'identifier les erreurs

## Plan

### 1 Les templates avec C++11/C++17

- Instanciation implicite vs explicite
- Surcharge des fonctions templates
- Spécialisation partielle/totale des templates
- Modèles variadiques
- Les types locaux et non nommés comme arguments template

### 2 Nouveautés dans la STL

- `std::optional`, `std::variant` et `std::any`
- Itérateurs
- Nouveaux conteneurs

## Surcharge de fonctions templates

### Surcharge de fonctions templates

La surcharge des fonctions templates permet de définir plusieurs versions d'une même fonction template avec des signatures différentes

☛ Gérer différents types de données et cas d'utilisation

```
1 template <typename T>
2 void fonction(T param1, T param2) {
3     // Implémentation pour le type T
4 }
5
6 template <typename U>
7 void fonction(U param1, U param2) {
8     // Implémentation pour le type U
9 }
```

- ☛ Chaque version de la fonction template doit avoir un nom identique et une signature unique (différence par les types d'arguments)
- ☛ Il est possible de surcharger des fonctions templates avec des fonctions non-templates
- ☛ Le compilateur sélectionne la version appropriée de la fonction en fonction des types d'arguments utilisés lors de l'appel



## Paramètres templates par défaut

### Paramètres templates par défaut

Les paramètres de templates peuvent avoir des valeurs par défaut

```
1 template <class T = int, unsigned int size = 2>
2 struct MyArray {
3     T data[size];
4 };
5
6 MyArray<> a; // MyArray<int, 2>
7 MyArray<double> b; // MyArray<double, 2>
8 MyArray<double, 10> b; // MyArray<double, 10>
```



## Surcharge de fonctions templates

### Surcharge de fonctions templates

```
1 // [1] Fonction template
2 template <typename T>
3 T minimum(T a, T b) {
4     return a <= b ? a : b;
5 }
6
7 // [2] Surcharge avec une fonction template
8 template <typename T>
9 std::basic_string<T> minimum(std::basic_string<T> a, std::basic_string<T> b) {
10     return a.length() <= b.length() ? a : b;
11 }
12
13 // [3] Surcharge avec une fonction non-template
14 std::string minimum(std::string a, std::string b) {
15     return a.length() <= b.length() ? a : b;
16 }
17
18 /*****
19 minimum(3, 4); // Appel de [1]
20 minimum(3.99, 4.01); // Appel de [1]
21 minimum(std::wstring(L"def"), std::wstring(L"acxyz")); // Appel de [2]
22 minimum(std::string("def"), std::string("acxyz")); // Appel de [3]
23 *****/
```



## Plan

- 1 Les templates avec C++11/C++17
  - Instanciation implicite vs explicite
  - Surcharge des fonctions templates
  - Spécialisation partielle/totale des templates
  - Modèles variadiques
  - Les types locaux et non nommés comme arguments template
- 2 Nouveautés dans la STL
  - std::optional, std::variant et std::any
  - Itérateurs
  - Nouveaux conteneurs



## Spécialisation partielle ou totale des templates

## Présentation

- Une classe template peut ne pas fonctionner avec tous les types d'arguments imaginables
  - ☛ Il faut définir des versions spécifiques pour certains types de données
    - ☛ Spécialisation des templates
- Deux types de spécialisation existent
  - ☛ Spécialisation complète : définit une version indépendante du template pour un type spécifique.
  - ☛ Spécialisation partielle : définit une version du template pour un type spécifique tout en laissant les autres types génériques
- La spécialisation doit être placée après la définition générale du template

## Spécialisation partielle ou totale des templates

## Spécialisation totale des templates

- Définit une version indépendante du template pour des types spécifiques de tous les arguments du template

```

1  template <typename T> class A {
2  // Définition générale
3  };
4
5  template <> class A<int> {
6  // Définition spécialisée pour le type int
7  };

```

## Spécialisation partielle ou totale des templates

## Spécialisation partielle des templates

- Définit une version du template pour un type spécifique tout en laissant les autres types génériques

```

1  template <typename T, typename U> class B {
2  // Définition générale
3  };
4
5  template <typename T> class B<T, int> {
6  // Définition spécialisée pour U=int
7  };

```

## Programmation générique

## Exemple : fonction template

```

1  #include <iostream>
2  using namespace std;
3  template<class T> class Array {
4  private:
5  enum { size = 100 };
6  T A[size];
7  public:
8  // ...
9  T& operator[](int index)
10 {
11 if(index >= 0 && index < size) return A[index];
12 else std::cerr << "Index out of range" << std::endl;
13 }
14 };
15
16 int main()
17 {
18 Array<int> ia; // un "tableau" d'entiers (ici T = int)
19 Array<float> fa; // un "tableau" de réels (ici T = float)
20 for(int i = 0; i < 20; i++) {
21 ia[i] = i * i;
22 fa[i] = float(i) * 1.414;
23 }
24 for(int j = 0; j < 20; j++)
25 std::cout << j << ": " << ia[j] << ", " << fa[j] << std::endl;
26 }

```

## Programmation générique

## Variables templates

➤ C++14 a introduit les variables templatisées

- ☞ Définir des valeurs qui peuvent prendre différents types en fonction du contexte
- ☞ Au lieu de créer plusieurs variables pour différents types, cette approche permet de définir une seule variable capable de s'adapter à diverses situations
- ☞ Les variables templatisées sont souvent utilisées dans les fonctions et classes templates.

```
1 template <typename T> T pi = T(3.14159265359);
2
3 template <typename T> T circle_perim(T rad) {
4     return 2*rad*pi<T>;
5 }
```



## Plan

## 1 Les templates avec C++11/C++17

- Instanciation implicite vs explicite
- Surcharge des fonctions templates
- Spécialisation partielle/totale des templates
- Modèles variadiques
- Les types locaux et non nommés comme arguments template

## 2 Nouveautés dans la STL

- `std::optional`, `std::variant` et `std::any`
- Itérateurs
- Nouveaux conteneurs



## Alias

## Alias

➤ Un alias (de type) permet de déclarer un nom à utiliser comme synonyme à type

➤ Syntaxe

```
1 using identifieur = type;
```

➤ Exemple

```
1 // C++11
2 using func = void (*)(int);
3
4 // C++03 equivalent:
5 // typedef void (*func)(int);
6
7 // func can be assigned to a function pointer value
8 void actual_function(int arg) { /* some code */ }
9 func fptr = &actual_function;
```

➤ `using` permet de pallier une limitation du `typedef` mécanisme qui ne fonctionne pas avec des modèles

```
1 template<typename T> using ptr = T*; // 'ptr<T>' is a synonym for 'T*'
2 using MyVector=std::vector<int,MyAlloc<int>>; //MyVector is an alias for vector
3 // <int>
4 ptr<int> ptr_int;
```



## Fonctions variadiques

## Présentation

➤ Une fonction variadique est une fonction qui prend un nombre de paramètre variables

- ☞ En C++, une fonction variadique est définie à l'aide d'un modèle (template) et d'une ellipse ...

```
1 template<typename arg, typename... args>
2 return_type function_name(arg var1, args... var2)
```

➤ L'opérateur ... est interprété selon sa position

- ☞ `typename ...P` : regrouper plusieurs arguments de type dans le pack de types P
- ☞ `<P...>` : décompresser P lors de l'instanciation d'une classe ou d'un modèle de fonction
- ☞ `P...p` : regrouper plusieurs arguments de fonction dans le pack de variables p
- ☞ `somme(p...)` : décompresser la variable pack p et appeler `sum` avec plusieurs arguments.

➤ Exemple de déclaration d'une fonction variadique

```
1 template < typename T , typename ... P >
2 inline T sum ( T t , P ... p ) {}
```



## Fonctions variadiques

### Implémentation d'une fonction variadique

- Principalement, le développement des fonctions variadiques se base sur la récursivité

```
1 #include <iostream>
2
3 template <typename T>
4 T sum (T t) { return t; }
5
6 template <typename T, typename... P>
7 T sum (T t,P ... p)
8 {
9     return t+sum(p...);
10 }
11 int main() {
12     auto s = sum(-7,3.7f,9u,-2.6);
13     std::cout<<"s is "<<s<<" and its type is "<<typeid(s).name()<<"\n";
14     return 0;
15 }
```



## Fonctions variadiques

## Fonctions variadiques

### Pliage des arguments

- La notion de pliage des arguments, ou **fold expressions**, a été introduite avec le standard C++17
- Le pliage d'expression permet d'effectuer des opérations sur tous les éléments d'un pack de paramètres variadiques.
- Il existe deux types de pliages d'expression : le pliage binaire et le pliage unaire.
- ❶ Pliage binaire : Il utilise un opérateur binaire (comme +, -, \*, /, etc.) pour combiner tous les éléments du pack. Il y a deux formes de pliage binaire :
  - Pliage à gauche : (args + ... + init) où init est une valeur initiale et args est le pack de paramètres.
  - Pliage à droite : (init + ... + args) où l'opération commence avec la valeur initiale à droite.
- ❷ Pliage unaire : Il utilise un opérateur unaire et n'a pas besoin de valeur initiale. Il y a aussi deux formes :
  - Pliage à gauche : (... op args) où op est l'opérateur unaire.
  - Pliage à droite : (args op ...)



## Fonctions variadiques

### Exemple de pliage d'arguments

```
1 #include <iostream>
2
3 // Fonction modèle variadique pour imprimer tous les arguments
4 template<typename... Args>
5 void printAll(Args... args) {
6     // Pliage d'expression unaire à gauche pour imprimer tous les arguments
7     (std::cout << ... << args) << '\n';
8 }
9
10 int main() {
11     // Appel de la fonction avec différents arguments
12     printAll(1, " deux ", 3.14, " quatre ", '5');
13     // Affiche : 1 deux 3.14 quatre 5
14     return 0;
15 }
```



### Implémentation d'une fonction variadique

- L'opérateur sizeof... permet de renvoyer le nombre d'éléments d'une variable pack

```
1 #include <iostream>
2
3 template <typename T, typename... P>
4 T sum (T t,P... p) {
5     std::cout << "_PRETTY_FUNCTION_" << "\n";
6     if constexpr (sizeof...(p) == 0)
7         return t;
8     else
9         return t+sum(p...);
10 }
11
12 int main() {
13     auto s = sum(-7,3.7f,9u,-2.6);
14     std::cout<<"s is "<<s<<" and its type is "<<typeid(s).name()<<"\n";
15     return 0;
16 }
```



## Variadic templates ( patrons variadiques)

### Présentation

- ➔ Introduits depuis C++11
- ➔ Un patron variadique possède un nombre d'arguments variable
- ➔ Déclaration

```
1 template <typename... Types>
2 class MyVariadicTemplate { };
```

### ➔ Instanciation

```
1 MyVariadicTemplate<int> instance1;
2 MyVariadicTemplate<string, double, vector<int>> instance2;
3 MyVariadicTemplate<> instance3; // Instanciation sans argument
```

### ➔ Pour interdire l'instanciation sans aucun argument

```
1 template <typename T1, typename... Types>
2 class MyVariadicTemplate { };
```



## Variadic templates ( patrons variadiques)

```
1 template<typename... Args> class Tuple; // Classe variadique 'Tuple'
2
3 // Classe partielle spécialisée pour gérer un élément et le reste du tuple
4 template<typename Head, typename... Tail>
5 class Tuple<Head, Tail...> {
6     private:
7         Head head;
8         Tuple<Tail...> tail;
9
10    public:
11        // Constructeur qui initialise la tête et la queue du tuple
12        Tuple(Head h, Tail... t) : head(h), tail(t...) {}
13        constexpr static int size() { // Obtenir la taille du tuple
14            return 1 + sizeof...(Tail);
15        }
16
17        template<std::size_t idx>
18        decltype(auto) get() const { // Obtenir la valeur à l'index spécifié
19            if constexpr (idx == 0) {
20                return head;
21            } else {
22                return tail.template get<idx - 1>();
23            }
24        }
25    };
```



## Variadic templates ( patrons variadiques)

```
1 template<> class Tuple<> { // Cas de base : Tuple vide
2     public:
3         constexpr static int size() {
4             return 0;
5         }
6 };
7 int main() {
8     Tuple<int, double, char> myTuple(1, 3.14, 'a');
9     std::cout << "Taille du tuple : " << myTuple.size() << std::endl;
10    std::cout << "Premier élément : " << myTuple.get<0>() << std::endl;
11    std::cout << "Deuxième élément : " << myTuple.get<1>() << std::endl;
12    std::cout << "Troisième élément : " << myTuple.get<2>() << std::endl;
13    return 0;
14 };
```



## Plan

- 1 Les templates avec C++11/C++17
  - Instanciation implicite vs explicite
  - Surcharge des fonctions templates
  - Spécialisation partielle/totale des templates
  - Modèles variadiques
  - Les types locaux et non nommés comme arguments template
- 2 Nouveautés dans la STL
  - std::optional, std::variant et std::any
  - Itérateurs
  - Nouveaux conteneurs



## Les types locaux et non nommés comme arguments template

## Les types locaux et non nommés comme arguments template

- ▶ C++11 autorise l'utilisation des types non nommés et les types locaux comme arguments pour les templates

```

1  template <typename T>
2  void f(T) { } // function template
3
4  template <typename T>
5  class C { }; // class template
6  struct { } obj; // object obj of unnamed C++ type1
7  void g()
8  {
9      struct S { }; // local type
10     f(S()); // OK in C++11; was error in C++03
11     f(obj); // OK in C++11; was error in C++03
12     C<S> cs; // OK in C++11; was error in C++03
13     C<decltype(obj)> co; // OK in C++11; was error in C++03
14 }

```



## Les conteneurs

## Présentation

- ▶ En C++, les conteneurs sont des classes qui stockent des données. Ils fournissent une interface commune pour accéder et manipuler les données stockées.
- ▶ En C++17, il existe de nombreux types de conteneurs disponibles, chacun avec ses propres caractéristiques. Les conteneurs les plus courants sont les suivants :
  - ☞ Les vecteurs `std::vector<T>` sont des conteneurs dynamiques qui peuvent stocker des données de type quelconque. Ils sont efficaces pour accéder aux éléments par leur indice
  - ☞ Les listes `std::list<T>` sont des conteneurs dynamiques qui peuvent stocker des données de type quelconque. Ils sont efficaces pour ajouter et supprimer des éléments au début ou à la fin de la liste.
  - ☞ Les tableaux associatifs `std::map<T>` sont des conteneurs qui stockent des données sous forme de paires clé-valeur. Ils sont efficaces pour accéder aux éléments par leur clé.
  - ☞ Les ensembles `std::set<T>` sont des conteneurs qui stockent des données uniques. Ils sont efficaces pour rechercher des éléments dans le conteneur.
- ▶ Autres conteneurs
  - ☞ Les conteneurs de séquence, tels que `std::deque` et `std::forward_list`
  - ☞ Les conteneurs de piles et de files, tels que `std::stack` et `std::queue`



## Plan

## 1 Les templates avec C++11/C++17

## 2 Nouveautés dans la STL

- `std::optional`, `std::variant` et `std::any`
- Itérateurs
- Nouveaux conteneurs



## Plan

## 1 Les templates avec C++11/C++17

- Instanciation implicite vs explicite
- Surcharge des fonctions templates
- Spécialisation partielle/totale des templates
- Modèles variadiques
- Les types locaux et non nommés comme arguments template

## 2 Nouveautés dans la STL

- `std::optional`, `std::variant` et `std::any`
- Itérateurs
- Nouveaux conteneurs



## std::optional

### Le conteneur std::optional

- Le type `std::optional` a été introduit avec le standard C++17
- `std::optional` est un type de données qui représente une valeur optionnelle
  - ☞ Une valeur qui peut être présente ou absente
  - ☞ Défini dans `<optional>`
- Avantages
  - ☞ Expliciter clairement qu'une valeur peut être absente
  - ☞ Facilite la gestion des erreurs en permettant de vérifier si une valeur est présent
- Méthodes utiles
  - ☞ `has_value()` : Renvoie true si `std::optional` contient une valeur, false sinon.
  - ☞ `value()` : Renvoie la valeur contenue dans `std::optional`, ou lance une exception si elle est vide.
  - ☞ `std::optional` peut être utilisé comme condition dans un if : il sera évalué à true s'il contient une valeur, false sinon.



## std::optional

### Exemple avec le conteneur std::optional

```

1 #include <iostream>
2 #include <optional>
3 int main() {
4     // We might not know the hour. But if we know it, it's an integer
5     std::optional<int> currentHour;
6     if (!currentHour.has_value()) {
7         std::cout << "We don't know the time" << std::endl;
8     }
9     currentHour = 18;
10    if (currentHour) {
11        std::cout << "Current hour is: " << currentHour.value() << std::endl;
12    }
13 }

```



## std::variant

### Le conteneur std::variant

- Le type `std::variant` a été introduit avec le standard C++17
- `std::variant` est un type de données sécurisé pour les types polymorphiques
  - ☞ Il permet de stocker une valeur d'un type parmi un ensemble prédéfini de types
  - ☞ Le type stocké est appelé le **type actif** de la variante
  - ☞ Seul un type de valeur est actif à la fois
- L'accès à l'élément actif se fait par la fonction modèle `std::get<>()`
  - ☞ En spécifiant l'index
 

```
std::variant<int,string> v;
v=15;
std::get<0>(v); // Accès
```
  - ☞ En spécifiant le type
 

```
std::variant<int,string> v;
v="Hello variant!";
std::get<string>(v); // Accès
```
- En cas d'erreur d'accès, une exception de type `std::bad_variant_access` est levée



## std::any

### Le conteneur std::any

- `std::any` permet de stocker une valeur de n'importe quel type constructible par copie
  - ☞ Un objet `std::any` peut contenir une instance de n'importe quel type ou être vide.
  - ☞ La valeur contenue est appelée l'objet contenu.
- La fonction modèle non-membre `std::any_cast<>` fournit un accès sûr de type à l'objet contenu
- L'exception `std::bad_any_cast` est lancée par `std::any_cast<>` en cas de non-concordance de type



## std::any

## Principales méthodes de std::any

## ↳ Les observateurs

```
1 bool has_value() const noexcept; // Vérifie si l'objet contient une valeur
2 const std::type_info& type() const noexcept; // Retourne le typeid de l'objet contenu
```

## ↳ Les modificateurs

```
1 template< class ValueType, class... Args >
2 std::decay_t<ValueType>& emplace( Args&&... args ); // Remplace l'objet contenu par
   un nouveau construit
3
4 void reset() noexcept; // Si n'est pas vide, détruit l'objet courant
```



## std::any

## Utilisation de std::any

```
1 #include <iostream>
2 #include <string>
3 #include <cassert>
4 #include <any>
5 int main() {
6     std::any x{std::string("Hello")};
7     assert(x.has_value() && x.type() == typeid(std::string));
8     std::any y;
9     assert(!y.has_value());
10    x.swap(y);
11    assert(!x.has_value() && y.has_value());
12    x = y;
13    std::cout << std::any_cast<std::string>(x) << '\n';
14    y.reset();
15    assert(!y.has_value());
16    try {
17        std::any_cast<int>(x);
18    }
19    catch (const std::bad_any_cast&) {
20        std::cout << "any_cast failed\n";
21    }
22 }
```



## Plan

## 1 Les templates avec C++11/C++17

- Instanciation implicite vs explicite
- Surcharge des fonctions templates
- Spécialisation partielle/totale des templates
- Modèles variadiques
- Les types locaux et non nommés comme arguments template

## 2 Nouveautés dans la STL

- std::optional, std::variant et std::any
- Itérateurs
- Nouveaux conteneurs



## Itérateurs

## Hiérarchie des itérateurs (1/2)

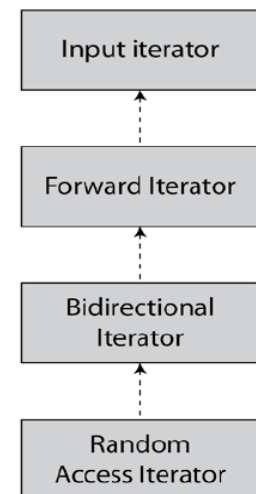
## ↳ C++ offre quatre catégories d'itérateurs

## 1 L'itérateur d'entrée

- Peut avancer et permet de lire l'élément pointé.
- Peut être copié, mais l'incrément ou la déréférenciation invalide les autres copies.
- Les éléments accessibles via un itérateur d'entrée ne peuvent être lus qu'une seule fois au maximum.
- Utilisé typiquement pour accéder aux éléments d'un flux, un par un.

## 2 L'itérateur avant

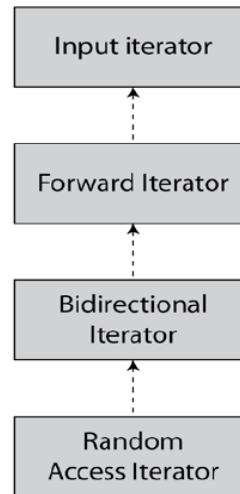
- Peut être déréférencé plusieurs fois.
- Les copies restent valides après incrément ou déréférenciation.
- Deux itérateurs pointant le même élément sont garantis égaux.



## Itérateurs

## Hiérarchie des itérateurs (2/2)

- ③ L'itérateur bidirectionnel
  - ☞ La capacité d'itérer en arrière.
  - ☞ Utilise l'opérateur -- pour décrémenter la position.
- ④ L'itérateur a accès aléatoire
  - ☞ Peut être déréférencé plusieurs fois.
  - ☞ La capacité d'accéder directement à n'importe quelle position en temps constant
  - ☞ Fourni par la fonction membre `operator[]` pour accéder aux éléments à des indices génériques.
  - ☞ Les opérateurs binaires + et - permettent d'avancer ou de reculer de n'importe quelle quantité.



## STL : Itérateurs

## Itérateurs prédéfinis

- Types d'itérateurs prédéfinis trouvés dans les définitions de classes de conteneurs de la bibliothèque standard
  - ☞ Chaque type d'itérateur n'est pas défini pour chaque conteneur

Predefined iterator type name	Direction of ++	Capability
<code>iterator</code>	forward	read/write
<code>const_iterator</code>	forward	read
<code>reverse_iterator</code>	backward	read/write
<code>const_reverse_iterator</code>	backward	read

## STL : Itérateurs

## STL : Itérateurs

## Conteneurs et itérateurs

- Chaque conteneur de la STL fournit au moins :
  - ☞ Un type local d'itérateur permettant d'accéder à ses éléments, et d'en faire un parcours exhaustif
 

```

1 std::list<int> ll;
2 std::list<int>::iterator it;
```
  - ☞ Une fonction membre `begin()` renvoyant un itérateur du type local au conteneur pointant sur le premier élément
  - ☞ Une fonction membre `end()` renvoyant un itérateur du type local au conteneur pointant juste après le dernier élément du conteneur
- Pour avoir des itérateurs constants interdisant la modification des éléments pointés
  - ☞ `cbegin()` renvoie un itérateur `const_iterator`
  - ☞ `cend()` renvoie un itérateur `const_iterator`

## Utiliser les itérateurs

```

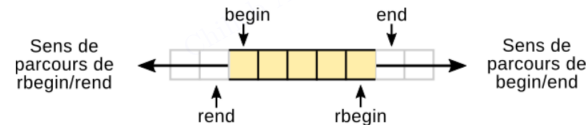
1 // Création d'un std::vector<int> avec quelques éléments
2 std::vector<int> vect = {10, 20, 30, 40, 50};
3
4 // Utilisation de iterator pour parcourir et modifier les éléments
5 std::cout << "Utilisation de iterator:" << std::endl;
6 for(std::vector<int>::iterator it {vect.begin()}; it != vect.end(); ++it) {
7     std::cout << *it << " "; // Affiche l'élément
8     *it += 5; // Modifie l'élément en ajoutant 5
9 }
10 std::cout << std::endl;
11
12 // Utilisation de const_iterator pour parcourir sans modifier les éléments
13 std::cout << "Utilisation de const_iterator:" << std::endl;
14 for(std::vector<int>::const_iterator cit {vect.cbegin()}; cit != vect.cend(); ++cit) {
15     std::cout << *cit << " "; // Affiche l'élément
16 }
```

## STL : Itérateurs

## Itérateurs inverses

➤ Parcourir une collection de la fin vers le début

- ▣ `rbegin()` (version constante `crbegin()`) retourne un itérateur inversé pointant vers le dernier élément du conteneur
- ▣ `rend()` (version constante `crend()`) retourne un itérateur inversé pointant vers l'élément théorique juste avant le premier élément du conteneur



```
1 std::vector<int> v {1, 2, 3, 4, 5};
2
3 std::cout << "Les éléments du vecteur dans l'ordre inverse sont :\n";
4 for (auto it = v.rbegin(); it != v.rend(); ++it) {
5     std::cout << *it << ' ';
6 }
}
```



## Tableaux statiques

## Tableaux statiques

➤ Les tableaux statiques sont des tableaux dont la taille est connue à la compilation et qui ne peut varier

- ▣ Plus rapide à créer qu'un `std::vector` puisqu'il n'y a pas d'opération d'ajout ou de retrait d'éléments
- ▣ Conteneur défini dans le fichier d'en-tête `<array>`

```
template<class T, std::size_t N> struct array;
```

▣ Accès aux éléments

- `operator[]` // Accès à un élément
- `at()` // Accès à un élément avec vérification des bornes
- `front()` // Référence sur le premier élément
- `back()` // Référence sur le dernier élément

▣ Itérateurs : `begin()`, `cbegin`, `rbegin`, `end`, `cend` et `rend`



## Plan

## 1 Les templates avec C++11/C++17

- Instanciation implicite vs explicite
- Surcharge des fonctions templates
- Spécialisation partielle/totale des templates
- Modèles variadiques
- Les types locaux et non nommés comme arguments template

## 2 Nouveautés dans la STL

- `std::optional`, `std::variant` et `std::any`
- Itérateurs
- Nouveaux conteneurs

## Tableaux statiques

## Exemple : tableaux statiques

```
1 #include <algorithm>
2 #include <array>
3 #include <iostream>
4 #include <iterator>
5 #include <string>
6
7 int main() {
8     std::array<int, 3> a = {1, 2, 3}; // Double braces never required after =
9
10    // Ranged for loop is supported
11    std::array<std::string, 2> a3{"E", "\u0018E"};
12    for (const auto& s : a3)
13        std::cout << s << ' ';
14    std::cout << '\n';
15
16    // Deduction guide for array creation (since C++17)
17    [[maybe_unused]] std::array a4{3.0, 1.0, 4.0}; // std::array<double, 3>
18
19    // Behavior of unspecified elements is the same as with built-in arrays
20    [[maybe_unused]] std::array<int, 2> a5; // No list init, a5[0] and a5[1]
21    // are default initialized
22    [[maybe_unused]] std::array<int, 2> a6{}; // List init, both elements are value
23    // initialized, a6[0] = a6[1] = 0
24    [[maybe_unused]] std::array<int, 2> a7{1}; // List init, unspecified element is value
25    // initialized, a7[0] = 1, a7[1] = 0
26 }
```



## Les conteneurs non triés

### Présentation

- Les conteneurs non triés `unordered_set` et `unordered_map` ont été introduits depuis C++11
- L'objectif est de disposer des conteneurs permettant des opérations d'insertion, de recherche et de suppression en temps constant
- L'idée de base est de stocker les éléments dans des compartiments (bucket) tel que chaque compartiment est retrouvé grâce à l'utilisation d'une fonction de hachage



## Les conteneurs non triés

### Le conteneur `unordered_set`

- Insérer un élément

```
1 std::pair<iterator,bool> insert( const value_type& value );
2 std::pair<iterator,bool> insert( value_type&& value );
```

- Supprimer un élément

```
size_type erase(const Key& key );
```

- Chercher un élément

```
iterator find(const Key& key )
```



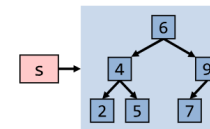
## Les conteneurs non triés

### Le conteneur `unordered_set`

- Le conteneur `unordered_set` modélise un ensemble d'éléments non ordonnés
  - Chaque élément apparaît une seule fois
  - Il n'y a aucun ordre sur les éléments
- `unordered_set` est implémenté à l'aide d'un tableau de compartiments où chaque compartiment contient plusieurs éléments dans une liste chaînée
  - La localisation du compartiment contenant un élément est déterminée à l'aide d'une fonction de hachage en fonction de la valeur de l'élément

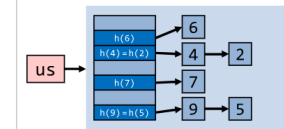
#### Ordered Sets

```
#include <set>
```



#### Hash Sets

```
#include <unordered_set>
```



## Les conteneurs non triés

### Exemple : le conteneur `unordered_set`

```
1 #include <iostream>
2 #include <unordered_set>
3
4 void print(const auto& set) {
5     for (const auto& elem : set)
6         std::cout << elem << ' ';
7     std::cout << '\n';
8 }
9
10 int main()
11 {
12     std::unordered_set<int> mySet{2, 7, 1, 8, 2, 8, 5}; // creates a set of ints
13     print(mySet);
14
15     mySet.insert(5); // puts an element 5 in the set
16     print(mySet);
17
18     if (auto iter = mySet.find(5); iter != mySet.end())
19         mySet.erase(iter); // removes an element pointed to by iter
20     print(mySet);
21
22     mySet.erase(7); // removes an element 7
23     print(mySet);
24 }
```



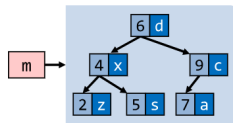
## Les conteneurs non triés

### Le conteneur unordered\_map

- Le conteneur `unordered_map` permet de stocker des paires (clé,valeur) non ordonnées
- `unordered_map` est implémenté à l'aide d'un tableau de compartiments où chaque compartiment contient plusieurs éléments dans une liste chaînée
  - ☞ La localisation du compartiment contenant un élément est déterminée à l'aide d'une fonction de hachage en fonction de la valeur de la clé

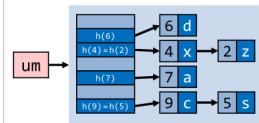
#### Ordered Key→Value Maps

```
#include <map>
```



#### Hashed Key→Value Maps

```
#include <unordered_map>
```



## Les conteneurs non triés

### Le conteneur unordered\_map

- Insérer un élément
- Insérer un élément
 

```
1 std::pair<iterator,bool> insert( const value_type& value );
2 std::pair<iterator,bool> insert( value_type&& value );
```
- Supprimer un élément
 

```
size_type erase(const Key& key );
```
- Chercher un élément
 

```
iterator find(const Key& key )
```



### Exemple : le conteneur unordered\_map

```
1 #include <iostream>
2 #include <string>
3 #include <unordered_map>
4
5 int main() {
6     std::unordered_map<std::string, std::string> u = {
7         {"RED", "#FF0000"},
8         {"GREEN", "#00FF00"},
9         {"BLUE", "#0000FF"}
10    };
11    auto print_key_value = [](const auto& key, const auto& value) {
12        std::cout << "Key:[" << key << "] Value:[" << value << "]\n";
13    };
14    for (const std::pair<const std::string, std::string>& n : u)
15        print_key_value(n.first, n.second);
16    std::cout << "\nIterate and print key-value pairs using C++17 structured binding:\n";
17    for (const auto& [key, value] : u)
18        print_key_value(key, value);
19    u["BLACK"] = "#000000";
20    u["WHITE"] = "#FFFFFF";
21    std::cout << "Use operator[] with non-existent key to insert a new key-value pair:\n";
22    print_key_value("new_key", u["new_key"]);
23    std::cout << "\nIterate and print key-value pairs, using auto;\n";
24    "new_key is now one of the keys in the map:\n";
25    for (const auto& n : u)
26        print_key_value(n.first, n.second);
27 }
```



Merci pour votre attention



Questions ?

