Programmation multi-paradigmes en C++

Session #10 : Introduction STL

Julien Deantoni, Thomas Soucheyre,
Stephane Janel, Ken-Patrick Lehrmann,
Ludovic Tessier, Leandro Fontoura
Cupertino, Kevin Yeung

Agenda

Template

- Un template est un moyen de faire du **code générique** et réutilisable, qui peut travailler avec n'importe quel type de données.

- Un template peut être une fonction ou une classe.

```
template<typename T>
T max(T a, T b) {
  return (a > b) ? a : b;
}
```

T est un paramètre de type. Lorsque vous appelez la fonction max, vous pouvez utiliser n'importe quel type pour T tant que cet type supporte l'opérateur >.

- De même, vous pouvez avoir une classe template, comme std::vector dans la bibliothèque standard C++, qui peut stocker des éléments de n'importe quel type.

Standard Template Library

STL (Standard Template Library) et std (standard) sont souvent utilisés de manière interchangeable en C++, mais ils se réfèrent à des choses différentes:

- **STL** est une partie de la bibliothèque standard C++ qui fournit plusieurs classes et fonctions génériques. Il s'agissait d'une bibliothèque autonome avant d'être incorporée dans la bibliothèque standard avec la standardisation de C++.

- **STD** est un espace de noms (std::) en C++ où toutes les fonctions, classes et objets standard sont définis. Il comprend plus que simplement la STL. Par exemple, il comprend des fonctionnalités comme l'entrée/sortie (cin, cout), les fonctions de manipulation de chaînes, les fonctions mathématiques, le temps, la date, etc.

Standard Template Library

La STL est une bibliothèque C++ mise en œuvre à l'aide des templates.

Les templates autorisent l'écriture d'un code sans considération envers le type des données avec lesquelles il sera utilisé. Les templates introduisent le concept de **programmation générique**.

La STL fournit:

- abstraction des pointeurs: les **itérateurs**
- classe **string** permettant de gérer efficacement et de manière sûre les chaînes de caractères
- classes **conteneurs**: vecteurs (vector), les tableaux associatifs (map), les listes chaînées (list)
- **algorithmes génériques** tels que des algorithmes d'insertion/suppression, recherche et tri (il existe 105 algorithmes en total)

Standard Template Library

Les avantages d'utiliser la bibliothèque STL

- Efficacité: hautement optimisée pour la performance.
- Productivité : fournit une bibliothèque qui permet d'économiser beaucoup de temps de développement.
- Portabilité: fait partie de la bibliothèque standard C++, prise en charge par tous les compilateurs.
- Fiabilité: les composants STL sont rigoureusement testés.
- Maintenabilité: offre un haut niveau d'abstraction, rend le code plus facile à comprendre.
- Extensibilité: de nouvelles structures de données et algorithmes peuvent être intégrés avec les composants existants.
- Programmation générique : permet d'écrire du code qui fonctionne avec différents types de données sans avoir à réécrire l'ensemble du code pour chaque type.

Basics of STL

Iterators

- Les itérateurs sont une **généralisation de pointeurs** qui permettent de travailler avec différentes structures de données de manière uniforme.

- Les itérateurs sont généralement considérés comme **plus sûrs que les pointeurs bruts**. Ils sont utilisés pour la **vérification des limites** des conteneurs et certaines implémentations fournissent des versions de débogage qui peuvent **détecter des erreurs courantes**, comme le déréférencement d'un itérateur qui est passé à la fin, ou qui a été invalidé en raison d'une modification du conteneur.

- Abstraction: Permet d'écrire **une seule implémentation d'un algorithme** qui fonctionnera pour les données contenues dans un array, une liste ou un autre conteneur.

Basics of STL

Iterators: Categories

_Input iterator

```
std::ifstream file("example.txt");
// Here, 'it' is a LegacyInputIterator
std::istream_iterator<std::string> it(file);
std::istream_iterator<std::string> end;
while (it != end) {
  std::cout << *it << ' ';
  ++it;
}
```

_Bidirectional iterator

```
std::list<int> mylist = {1, 2, 3, 4, 5};
// Here, 'it' is a LegacyBidirectionalIterator
std::list<int>::iterator it = mylist.end();
--it; // Move iterator to the last element
```

Chaque algo a des contraintes sur les itérateur.
Ex. sort(): random access or contiguous iterator

_Forward iterator

```
std::forward_list<int> mylist = {1, 2, 3, 4, 5};
// Here, 'it' is a LegacyForwardIterator
for (std::forward_list<int>::iterator it = mylist.begin(); it !=
  mylist.end(); ++it) {
  std::cout << *it << ' ';
}
```

_Random access iterator (deque)

```
std::deque<int> numbers = {1, 2, 3, 4, 5};
// Accessing an element directly using random access
std::deque<int>::iterator it = numbers.begin() + 3;
```

_Contiguous iterator

```
std::vector<int> myvector = {1, 2, 3, 4, 5};
// Contiguous iterator: sequence of elements in memory is continuous
std::vector<int>::iterator it = myvector.begin() + 2;
```

Basics of STL

Iterators

_Categories

Iterator category	Operations and storage requirement						
	write	read	increment		decrement	random access	contiguous storage
			without multiple passes	with multiple passes			
LegacyIterator			Required				
LegacyOutputIterator	Required		Required				
LegacyInputIterator (mutable if supports write operation)		Required	Required				
LegacyForwardIterator (also satisfies *LegacyInputIterator*)		Required	Required	Required			
LegacyBidirectionalIterator (also satisfies *LegacyForwardIterator*)		Required	Required	Required	Required		
LegacyRandomAccessIterator (also satisfies *LegacyBidirectionalIterator*)		Required	Required	Required	Required	Required	
LegacyContiguousIterator[1] (also satisfies *LegacyRandomAccessIterator*)		Required	Required	Required	Required	Required	Required

"Legacy Iterators" est un terme utilisé pour désigner les concepts d'itérateurs originaux qui faisaient partie de la bibliothèque avant l'introduction des plages (ranges) et des nouveaux concepts d'itérateurs en C++20.

Basics of STL

Iterators

_Les itérateurs les plus couramment utilisés (ces itérateurs sont utilisés avec des strings et des conteneurs de la bibliothèque standard C++, tels que std::vector, std::list, std::set, etc):

```
std::vector<int> vec = {1, 2, 3, 4, 5};

auto it = vec.begin();
int x = *it; // x = 1
++it; // advance by 1
auto it2 = it + 2; // advance 2
int y = *it2; // y = 4
auto e = vec.end(); // one past the
                    last element

*it = 7; // change the value of the element at position 3
```

Ne pas accéder **end** avec *
Position mémoire ne correspond pas au vecteur.

```
std::vector<int> vec = {1, 2, 3, 4, 5};

auto it = vec.rbegin();                    //last element
int x = *it; // x = 5
++it; // retreat by 1
auto it2 = it + 2;                  // retreat by 2
int y = *it2; // y = 2
auto e = vec.rend();    // one before the first element
*it = 6; // change the value of the element at position 1
```

Ne pas accéder **rend** avec *
Position mémoire ne correspond pas au vecteur.

Basics of STL

Adaptors

_ Un adaptateur d'insertion est un type d'adaptateur d'itérateur qui est utilisé pour insérer des éléments dans un conteneur.

_ Il existe trois types d'adaptateurs d'insertion: back_inserter, front_inserter et inserter.

```
std::vector<int> vec = {1, 2, 3, 4, 5};
std::list<int> list1 = {9, 9, 9, 9};

std::copy(vec.begin(), vec.end(), std::back_inserter(list1));          // list1: 9 9 9 9 1 2 3 4 5
std::copy(vec.begin(), vec.end(), std::front_inserter(list1));         // list1: 1 2 3 4 5 9 9 9 9
std::copy(vec.begin(), vec.end(), std::inserter(list1, std::next(list1.begin(), 2)));   // list1: 9 9 1 2 3 4 5 9 9
```

Basics of STL

String

- String

  Généralise la façon dont les séquences de caractères sont manipulées et stockées. La création, la manipulation et la destruction de chaînes sont toutes gérées par un ensemble pratique de méthodes de classe et de fonctions associées. La mémoire est gérée par la classe.

```c
#include <cstring>

char *s = strdup("Hello");
char *s2 = (char *)malloc(strlen(s) + 8);
// if !null
strcpy(s2, s);
strcat(s2, ", World");
free(s);
s = s2;
```

```cpp
#include <string>

string s = "Hello";
s += ", World";
```

Basics of STL

String

_operator+=() / append()

```
string str = "What is the meaning of life";
str += ", the universe and everything?";
```

_size() / empty()

```
size_t size = str.size();
bool isEmpty = str.empty();
```

_operator==() / compare()

```
if (str == "What is the meaning of life, the universe and
everything?") {
  cout << "Douglas Adam's quote!" << endl;
}
```

_string_view

```
constexpr string_view lifestr = "life";
```

_find() / substr() / replace()

```
size_t pos = str.find(lifestr);
if (pos == string::npos) {
  cout << "Not found!" << endl;
} else {
  string partial = str.substr(pos);
  str.replace(pos, lifestr.size(), "file");
  cout << str << '\n' << partial << endl;
}
```

_to_string()

```
int answer = 42;
str += " " + to_string(answer);
```

Basics of STL

string_view

_**string_view** fournit un objet léger qui offre un accès en lecture à une chaîne en utilisant une interface similaire à celle de std::basic_string.

- Arguments de fonctions: permet à la fonction d'accepter **n'importe quel objet de type chaîne**.(paramètres passé par copie)

```
bool is_palindrome(string_view text) {
  size_t length = text.size();
  for (size_t i = 0; i < length / 2; i++) {
    if (text[i] != text[length - i - 1]) {
      return false;
    }
  }
  return true;
}
```

```
char radar_chararr[] = "radar";
cout << is_palindrome(radar_chararr) << endl;

const char *radar_charptr = "radar";
cout << is_palindrome(radar_charptr) << endl;

string radar_str = radar_charptr;
cout << is_palindrome(radar_str) << endl;

string_view radar_view = radar_str;
cout << is_palindrome(radar_view) << endl;
```

- Performance: peut aider à **réduire les allocations de mémoire** (copies).

```
bool is_palindrome(string text);
```

- /!\ Une string view n'est probablement pas terminée par un caractère nul. Il faut utiliser .size().
- C'est une vue, la chaine de caractère sous-jacente doit exister au moins aussi longtemps que la view

Basics of STL

span

_ **span** est une vue sur une séquence contiguë d'éléments dans un tableau ou un autre type de conteneur de données. Il fournit un moyen d'accéder à plusieurs éléments consécutifs sans avoir à copier les données.

- Arguments de fonctions: paramètres sont passe par copie

```cpp
double norm(std::span<int> vec) {
  double sum_squares = 0.0;
  for (const auto &element : vec) {
    sum_squares += element * element;
  }
  return std::sqrt(sum_squares);

}

int main() {
  std::vector<int> vec = {1, 2, 3, 4, 5};
  std::cout << "Norm of the vector: " << norm(vec) << std::endl;

  std::array<int, 5> arr = {1, 2, 3, 4, 5};
  std::cout << "Norm of the array: " << norm(arr) << std::endl;

}
```

```cpp
int dot_product(std::span<int> a, std::span<int> b) {
  if (a.size() != b.size()) {
    throw std::invalid_argument("Spans must be the same size");
  }
  int product = 0;
  for (std::size_t i = 0; i < a.size(); ++i) {
    product += a[i] * b[i];
  }
  return product;

}

int main() {
  std::vector<int> vec = {1, 2, 3};
  std::array<int, 3> arr = {4, 5, 6};
  std::cout << "Dot product: " << dot_product(vec, arr) << std::endl;

}
```

- Performance: peut aider à **réduire les allocations de mémoire** (copies ou variables temporaires).

```cpp
int* arr = static_cast<int*>(std::malloc(5 * sizeof(int)));

std::span<int> s(arr, 5);
```

- /!\ C'est une référence, la séquence sous-jacente doit exister au moins aussi longtemps que le span.

Containers

Basics of STL

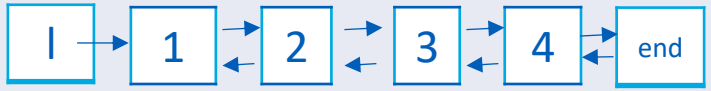
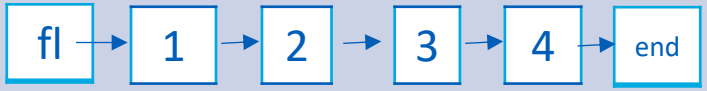
- La bibliothèque Containers est une **collection générique** de classes et d'algorithmes qui permettent aux programmeurs **d'implémenter facilement des structures de données courantes** telles que des files, listes et piles.

- Le conteneur **gère l'espace de stockage** alloué à ses éléments et fournit des fonctions membres pour y accéder, soit directement, soit via des itérateurs (objets avec des propriétés similaires à des pointeurs).

- La plupart des conteneurs ont au moins **plusieurs fonctions membres en commun et partagent des fonctionnalités**. Le choix du conteneur le mieux adapté à une application particulière dépend non seulement des fonctionnalités offertes, mais également de son efficacité pour différentes charges de travail.

Containers

Sequence Containers

Container	Description	Example
array<T, size>	Tableau de taille fixe stocké de manière contiguë en mémoire. La taille doit être connue à la compilation et ne peut pas être modifiée par la suite.	
vector<T>	Tableau de taille variable stocké de manière contiguë en mémoire.	
deque<T>	Un deque (double-ended queue) est un conteneur qui permet des insertions et suppressions rapides aux deux extrémités.	
list<T>	Une liste est un conteneur qui permet des insertions et suppressions rapides à n'importe quelle position.	
forward_list<T>	Une forward_list est une liste simplement chaînée qui permet des insertions et suppressions efficaces à n'importe quelle position.	

Array

Sequence containers

- Corrige des limitations des tableaux C. Avec des arrays "assignables" avec taille, itérateur, etc.
- Accès rapide : Les éléments d'un tableau peuvent être accédés rapidement par leur indice.
- Utilisation efficace de la mémoire : Les tableaux stockent les données de manière contiguë en mémoire, ce qui peut améliorer la performance en raison de la localité des références.
- Taille fixe : La taille d'un tableau est fixe et ne peut pas être modifiée une fois qu'elle est définie. Cela peut entraîner un gaspillage de mémoire si la taille du tableau est surévaluée.
- La taille ne doit pas etre trop grande

```cpp
#include <array>
#include <iostream>
int main() {
std::array<int, 5> arr = {4, 5, 1, 3, 9};
  std::cout << arr.size() << std::endl;  // 5
  std::cout << arr[0] << std::endl;      // 4
  std::cout << arr[2] << std::endl;      // 1
  std::cout << arr.front() << std::endl; // 4
  std::cout << arr.back() << std::endl;  // 9
  return 0;
}
```

```cpp
std::array<int, 5> arr1 = {4, 5, 1, 3, 9};
std::array<int, 6> arr2 = arr1;
```

Erreur de compilation. Meme si **`arr2`** est plus grand, la copie ne se fait pas si les tailles des arrays diferent.

Vector

Sequence containers

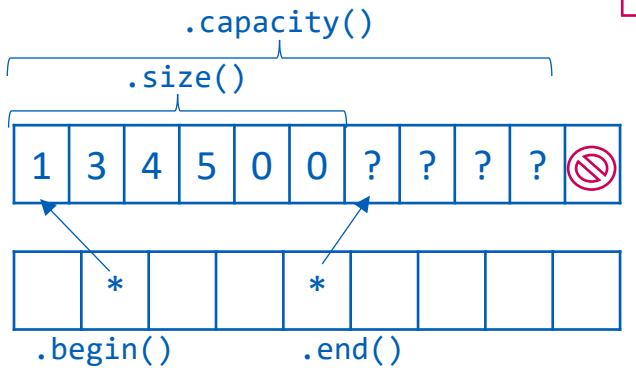
- Taille dynamique: les vecteurs sont dynamiques, leur taille peut changer pendant l'exécution.
- Accès aléatoire: les vecteurs permettent un accès aléatoire aux éléments, vous pouvez accéder à n'importe quel élément directement par son indice.

- La réallocation peut être coûteuse.
- Pas adapté pour l'insertion/suppression au milieu: L'insertion ou la suppression d'éléments au milieu d'un vecteur est une opération coûteuse car elle nécessite le déplacement des éléments suivants.
- Les itérateurs et références sont invalides en cas d'insertion et suppression.

```cpp
#include <iostream>
#include <vector>
int main() {
  vector<int> v = {1, 2, 3, 4, 5};
  v.push_back(6);          // 1 2 3 4 5 6
  v.pop_back();            // 1 2 3 4 5
  v.erase(v.begin() + 2); // 1 2 4 5
  v[1] = 3;                // 1 3 4 5
  v.reserve(8);            // allocate space for 8 elements
  v.resize(6, 0);          // 1 3 4 5 0 0
  std::cout << v.capacity() << std::endl; // 8 (or more)
  std::cout << v.size() << std::endl;     // 6
  return 0;
}
```

Attention a l'utilisation de erase dans des boucles

Ne pas utiliser reserve/resize dans des boucles



Double Ended Queue

Sequence containers

- Accès aléatoire: permettent un accès aléatoire direct aux éléments.
- Insertions et suppressions efficaces aux deux extrémités.
- Taille dynamique: la taille peut changer pendant l'exécution.

- Overhead de mémoire: surcoût de mémoire pour permettre le redimensionnement dynamique et les insertions aux deux extrémités.
- Pas adapté pour l'insertion/suppression au milieu : L'insertion ou la suppression d'éléments au milieu d'un deque est une opération coûteuse car elle nécessite le déplacement des éléments.

```cpp
#include <deque>
#include <vector>
int main() {
  std::deque<int> deq = {1, 2, 3, 4, 5};
  deq.push_back(6);                          // 1 2 3 4 5 6
  deq.push_front(0);                         // 0 1 2 3 4 5 6
  std::vector<int> vec{3, 4, 5};
  deq.insert(begin(deq),
             begin(vec), end(vec));          // 3 4 5 0 1 2 3 4 5 6
  deq.pop_front();                           // 4 5 0 1 2 3 4 5 6
  deq.pop_back();                            // 4 5 0 1 2 3 4 5
  deq.erase(begin(deq) + 2, begin(deq) + 5); // 4 5 3 4 5
}
```

Doubly-linked List

Sequence containers

- Insertions et suppressions efficaces à n'importe quelle position: ces opérations ne nécessitent que la modification des pointeurs des éléments voisins.
- Pas de déplacement d'éléments: l'insertion ou la suppression d'éléments dans une liste ne nécessite pas le déplacement d'autres éléments.
- Les itérateurs et références restent valides: tous les itérateurs et références restent valides, sauf ceux qui pointent vers les éléments insérés ou supprimés.

- Pas d'accès aléatoire : ne permettent pas un accès aléatoire aux éléments. Pour accéder à un élément, vous devez traverser la liste depuis le début.
- Utilisation inefficace de la mémoire : Chaque élément d'une liste doit stocker des pointeurs vers les éléments précédent et suivant, ce qui nécessite plus de mémoire que les vecteurs et les deque.
- Moins performant pour la traversée : Les listes sont moins performantes pour la traversée que les vecteurs et les deque en raison de la non-contiguïté des données en mémoire.

```cpp
#include <iterator>
#include <list>

int main() {
  std::list<int> lst = {1, 2, 3};
  lst.push_back(1);                       // 1 2 3 1
  lst.push_front(0);                      // 0 1 2 3 1
  lst.splice(std::next(lst.begin(), 2),
             std::list<int>{9, 8});       // 0 1 9 8 2 3 1
  lst.reverse();                          // 1 3 2 8 9 1 0
  lst.sort();                             // 0 1 1 2 3 8 9
  lst.unique();                           // 0 1 2 3 8 9
  return 0;
}
```

Simply-linked List

Sequence containers

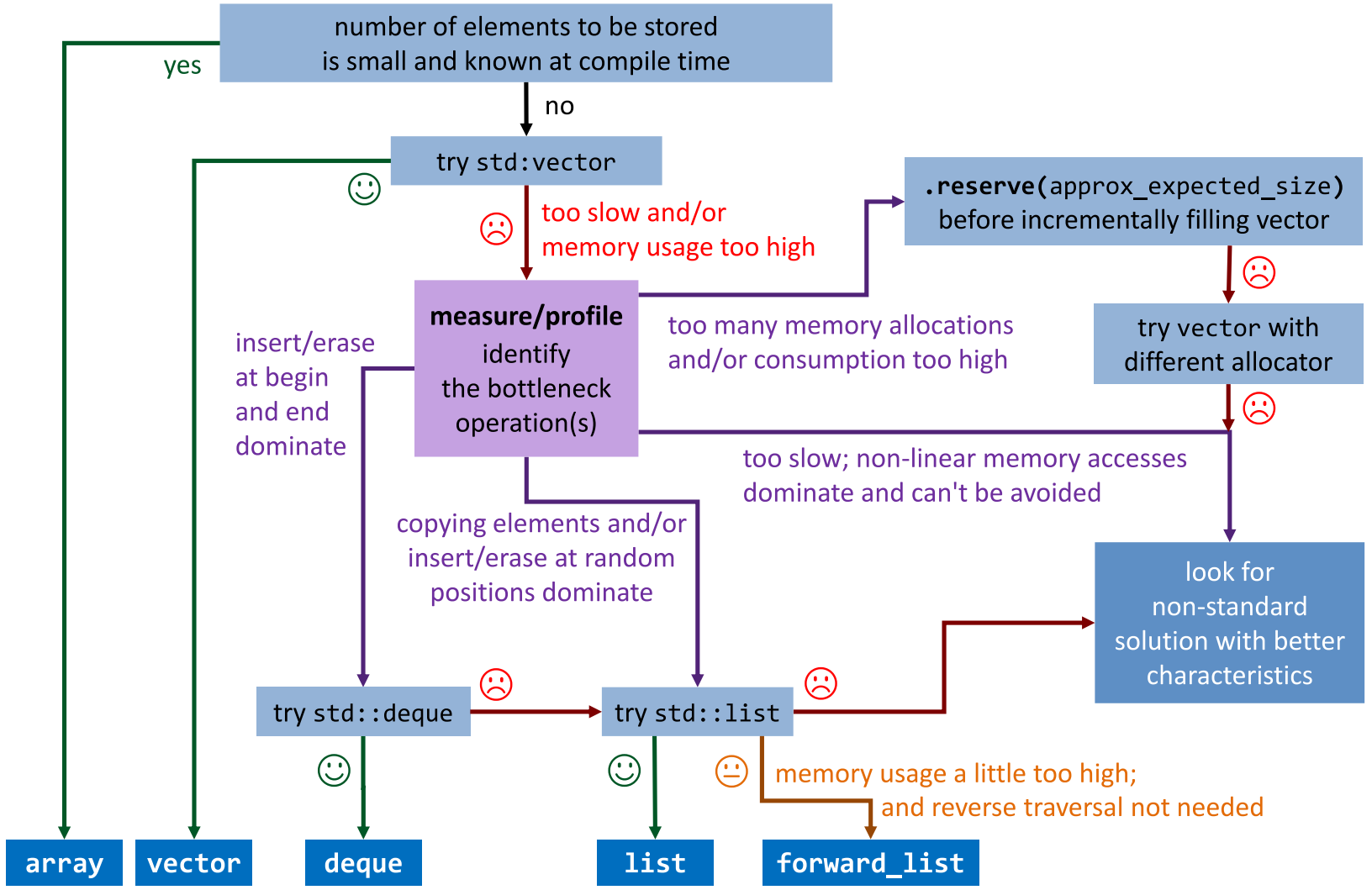
- Insertions et suppressions efficaces à n'importe quelle position: ces opérations ne nécessitent que la modification des pointeurs des éléments voisins.
- Pas de déplacement d'éléments: l'insertion ou la suppression d'éléments dans une liste ne nécessite pas le déplacement d'autres éléments.
- Les itérateurs et références restent valides: tous les itérateurs et références restent valides, sauf ceux qui pointent vers les éléments insérés ou supprimés.

- Pas d'accès aléatoire : ne permettent pas un accès aléatoire aux éléments. Pour accéder à un élément, vous devez traverser la liste depuis le début.
- Utilisation inefficace de la mémoire : Chaque élément d'une liste doit stocker des pointeurs vers les éléments précédent et suivant, ce qui nécessite plus de mémoire que les vecteurs et les deque.
- Moins performant pour la traversée : Les listes sont moins performantes pour la traversée que les vecteurs et les deque en raison de la non-contiguïté des données en mémoire.

```cpp
#include <forward_list>

int main() {
  std::forward_list<int> lst = {1, 2, 2};     // 1 2 2
  lst.push_front(0);                          // 0 1 2 2
  lst.reverse();                              // 2 2 1 0
  lst.sort();                                 // 0 1 2 2
  lst.unique();                               // 0 1 2
  lst.push_front(0);                          // 0 0 1 2
  lst.remove(1);                              // 0 0 2
  std::forward_list<int>::iterator it = lst.begin();
  lst.insert_after(it, 5);                    // 0 5 0 2
  lst.insert_after(++it, 6);                  // 0 5 6 0 2

  return 0;

}
```

Guidelines

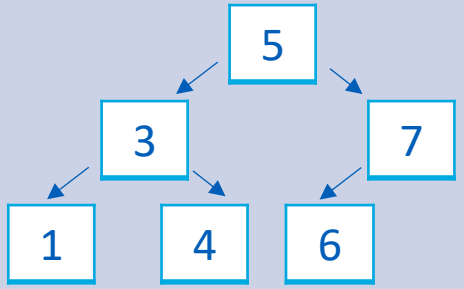
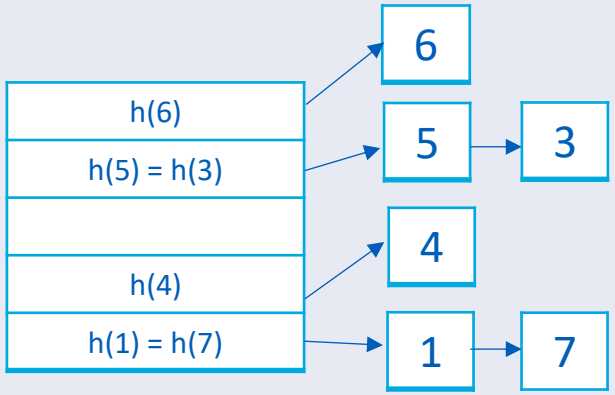
Sequence containers



Source:.https://hackingcpp.com/

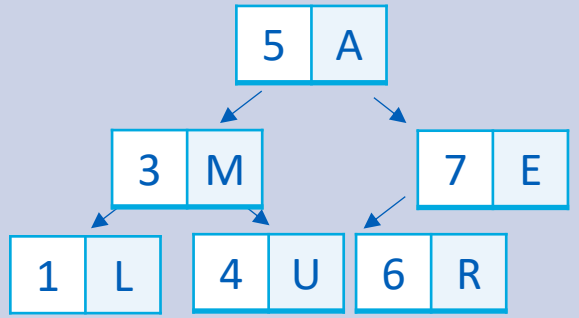
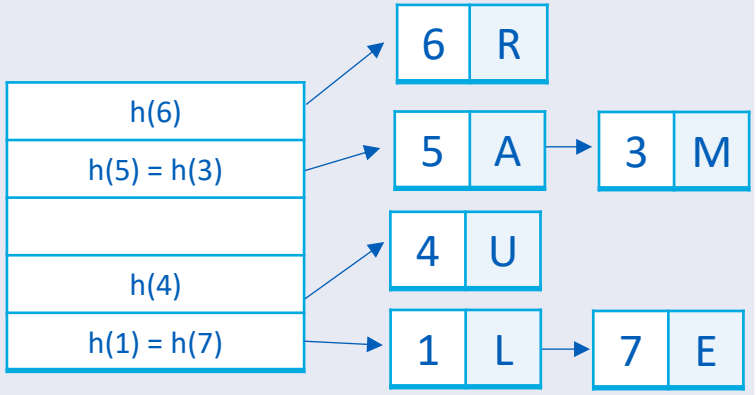
Containers

Associative Containers

Container	Description	Example
set<Key>	Stocke des éléments uniques dans un ordre trié. Généralement implémenté sous la forme d'un arbre binaire équilibré.	
unordered_set<Key>	Stocke des éléments uniques sans ordre particulier. Table de hachage pour la recherche de nœuds liste chainé pour le stockage.	

Containers

Associative Containers

Container	Description	Example
map<Key, Value>	Stocke des paires clé-valeur avec des clés uniques dans un ordre trié. Généralement implémenté sous la forme d'un arbre binaire équilibré.	
unordered_set<Key, Value>	Stocke des paires clé-valeur avec des clés uniques sans ordre particulier. Table de hachage pour la recherche de nœuds liste chainé pour le stockage.	

Set

Associative containers

- Unicité des éléments : ne contient que des éléments uniques, ce qui est utile pour éliminer les doublons.
- Ordre des éléments : les éléments sont automatiquement triés, ce qui facilite les recherches et itérations dans un ordre spécifique.
- Opérations ensemblistes : supportent des opérations comme l'union, l'intersection et la différence via des algorithmes STL
- Recherche efficace : Les opérations de recherche, d'insertion et de suppression sont en temps logarithmique, O(log n)
- Performance : opérations sont plus lentes que sur un std::unordered_set en raison de la nécessité de maintenir l'ordre des éléments
- Mémoire : peuvent utiliser plus de mémoire que les autres conteneurs comme les vecteurs ou les listes en raison de la surcharge des structures d'arbres
- Complexité : La gestion des arbres binaires équilibrés ajoute une complexité supplémentaire par rapport à des conteneurs plus simples comme les vecteurs.
- Pas d'accès direct

```cpp
#include <iostream>
#include <set>
#include <algorithm>
#include <iterator>

int main() {
  std::set<int> set1 = {1, 2, 3, 4, 5};

  set1.insert(42);
  // {1, 2, 3, 4, 5, 42}


  set1.insert(3);
  // {1, 2, 3, 4, 5, 42}

  set1.erase(42);
  // {1, 2, 3, 4, 5}

  std::cout << "Erase all odd numbers:\n";
  for (auto it = c.begin(); it != c.end();)
  {
     if (*it % 2 != 0)
        it = c.erase(it);
     else
        ++it;
  }

  return 0;
}
```

Map

Associative containers

- Unicité des éléments : ne contient que des éléments uniques, ce qui est utile pour éliminer les doublons.
- Ordre des éléments : les éléments sont automatiquement triés, ce qui facilite les recherches et itérations dans un ordre spécifique.
- Opérations ensemblistes : supportent des opérations comme l'union, l'intersection et la différence via des algorithmes STL
- Recherche efficace : Les opérations de recherche, d'insertion et de suppression sont en temps logarithmique, O(log n)
- Performance : opérations sont plus lentes que sur un std::unordered_set en raison de la nécessité de maintenir l'ordre des éléments
- Mémoire : peuvent utiliser plus de mémoire que les autres conteneurs comme les vecteurs ou les listes en raison de la surcharge des structures d'arbres
- Complexité : La gestion des arbres binaires équilibrés ajoute une complexité supplémentaire par rapport à des conteneurs plus simples comme les vecteurs.
- Pas d'accès direct

```cpp
#include <iostream>
#include <map>
#include <string>


struct Person {
  std::string name;
  int age;
  bool operator==(const Person& other) const {
    return name == other.name && age == other.age;
  }
};

// Custom hash function for Person
struct PersonHash {
  std::size_t operator()(const Person& person) const {
    return std::hash<std::string>()(person.name) ^
std::hash<int>()(person.age);
  }
};

int main() {
  std::map<Person, int, PersonHash> personMap;
  personMap[{"Alice", 30}] = 1;
  personMap[{"Bob", 25}] = 2;
  personMap[{"Charlie", 35}] = 3;

  return 0;
}
```

Containers

Unordered associative containers

- Les conteneurs associatifs non ordonnés implémentent des structures de données non triées (hachées) qui peuvent être recherchées rapidement (complexité moyenne de (O(1)), complexité dans le pire des cas de (O(n))).

  - unordered_set
  - unordered_map

  - unordered_multiset
  - unordered_multimap

Containers

Container adaptors

_ Les adaptateurs de conteneurs offrent un moyen d'utiliser des **conteneurs existants** avec une **interface différente** adaptée à des besoins spécifiques tels que le traitement LIFO, FIFO ou basé sur la priorité. Ils sont utiles pour simplifier l'implémentation de structures de données et d'algorithmes courants.

_ stack
std::dequeue

_ queue
std::dequeue

_ priority_queue
std::vector

_ flat_set
std::vector

_ flat_map
std::vector

_ flat_multiset / flat_multimap
std::vector

Containers

Container adaptors

- **stack**: pile (LIFO - Last In, First Out)
  - Conteneur: std::deque (par défaut), std::vector ou std::list
  - Utile pour les scénarios où vous devez stocker des données de manière à ce que le dernier élément ajouté soit le premier à être retiré, comme dans la gestion des appels de fonctions, l'évaluation des expressions, etc.
- **queue**: file d'attente (FIFO - First In, First Out)
  - Conteneur: std::deque (par défaut) ou std::list
  - Utile pour les scénarios où vous devez traiter les éléments dans l'ordre où ils ont été ajoutés, comme dans la planification des tâches, les algorithmes de recherche en largeur, etc.
- **priority_queue**: file de priorité, où les éléments sont ordonnés par priorité
  - Conteneur sous-jacent: std::vector (par défaut) et une fonction de comparaison pour maintenir l'ordre
  - Utile pour les scénarios où vous devez traiter les éléments en fonction de leur priorité, comme dans l'algorithme de Dijkstra, les systèmes de simulation d'événements, etc.

```cpp
#include <iostream>
#include <stack>
#include <queue>

int main() {
  // Exemple de std::stack
  std::stack<int> stack;
  stack.push(1); // 1
  stack.push(3); // 1 3
  stack.push(2); // 1 3 2
  std::cout << stack.top() << std::endl; // Affiche 2
  stack.pop();   // 1 3
  std::cout << stack.top() << std::endl; // Affiche 3

  // Exemple de std::queue
  std::queue<int> queue;
  queue.push(1); // 1
  queue.push(3); // 1 3
  queue.push(2); // 1 3 2
  std::cout << queue.front() << std::endl; // Affiche 1
  queue.pop();   // 3 2
  std::cout << queue.front() << std::endl; // Affiche 3

  // Exemple de std::priority_queue
  std::priority_queue<int> priorityQueue;
  priorityQueue.push(1); // 1
  priorityQueue.push(3); // 1 3
  priorityQueue.push(2); // 1 2 3
  std::cout << priorityQueue.top() << std::endl; // Affiche 3
  priorityQueue.pop();
  std::cout << priorityQueue.top() << std::endl; // Affiche 2

  return 0;
}
```

Ne pas utiliser pop() avec des containers vides

References

1. C++ Standard Library. https://en.cppreference.com/w/cpp/
2. 105 STL Algorithms in Less Than an Hour - Jonathan Boccara [ACCU 2018]. https://www.youtube.com/watch?v=bXkWuUe9V2I
3. C++ reference. Standard Containers. https://cplusplus.com/reference/stl/?kw=stl
4. Cpp Core Guidelines. The Standard Library. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-stdlib
5. Cpp Core Guidelines. Templates and generic programming. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-templates
6. A. Müller. Hacking C++. https://hackingcpp.com/index.html

POLYTECH NICE-SOPHIA

amadeus