

# Linguagem C++ III

11/12/2023

# Ficheiro ZIP

- Está disponível no Moodle um **ficheiro ZIP** de suporte aos tópicos de hoje
- **Exemplos** (simples) de utilização de estruturas de dados (**containers**): **array**, **vector**, **deque**, **stack**, **queue**, **set** e **map**

# Sumário

- STL – The Standard Template Library
- **Sequence Containers**: Array, Vector, Deque
- Container **Adaptors**: Stack, Queue
- **Associative Containers**: Map, Set

# STL

– The Standard Template Library

# STL – The Standard Template Library

- Objetivo : facilitar o desenvolvimento de código eficiente
- Conjunto de **classes genéricas (template classes)** que disponibilizam estruturas de dados (**containers**) e funcionalidades (**algoritmos e iteradores**)
- Containers : **array, vector, deque, map, set, ...**
- Algoritmos : **searching, sorting, ...**
- Eficiência : containers e funcionalidades são baseados em **algoritmos otimizados**

# STL – Vantagens

- **Algoritmos eficientes** : funcionalidades e estruturas de dados implementadas usando algoritmos rápidos e otimizados
- Legibilidade e manutenção : **API consistente** torna o código fácil de compreender, desenvolver e manter
- Reutilização : escrita de **código genérico** e reutilizável, aplicável a diferentes classes e tipos de dados

# Sequence Containers

# Sequence Containers

- Implementam estruturas de dados que podem ser acedidas de modo sequencial
- **array** : elementos contíguos e **tamanho fixo**
- **vector** : elementos contíguos e **tamanho variável**
- **deque** : double-ended queue
- **forward\_list** : lista simplesmente ligada
- **list** : lista duplamente ligada



# Array

# std::array

```
#include <array>
std::array<some_type, array_size>
std::array<int, 4> elems = {1, 2, 3, 4};
```

- **Tamanho fixo**
- Obter o tamanho do array : **.size()**
- Elementos contíguos e com **acesso aleatório**
- Aceder ao elemento numa dada posição : **.at( )** ou **[ ]**
- Aceder ao primeiro elemento (índice = 0) : **.front()**
- Aceder ao último elemento (índice = size – 1) : **.back()**

# std::array

```
array<unsigned int, 4> a = {0, 1, 2, 3};
```

```
cout << a.size() << endl;  
cout << a.front() << endl;  
cout << a.back() << endl;
```

```
// Replacing !!
```

```
a.front() = 9;  
a.back() = 10;
```

# Iterar sobre todos os elementos

```
for(const auto& e : a) {  
    cout << " " << e;  
}  
cout << endl;
```

// What is the output?

# Vector

# std::vector

```
#include <vector>
std::vector<some_type>
std::vector<int> elems = {1, 2, 3, 4};
```

- **Generalização** do array container
- Elementos contíguos + **Acesso aleatório** + **Tamanho variável**
  - Aumenta ou diminui de tamanho, quando se inserem ou apagam elementos
- Métodos eficientes para realizar **operações** na **cauda** de um vector
- A escolha preferencial, na maioria dos casos. **When in doubt, use a vector !**

# std::vector

```
vector<unsigned int> v = {0, 1, 2, 3};
```

```
cout << v.size() << endl;  
cout << v.front() << endl;  
cout << v.back() << endl;
```

```
// Replacing !!  
v.front() = 9;  
v.back() = 10;
```

# std::vector – Operações habituais

- `v.push_back(6);`
  - Inserir elemento na cauda do vector
- `v.pop_back();`
  - Remover elemento da cauda do vector
- `v.insert(v.begin(), 99);`
  - Inserir elemento **antes** do primeiro elemento do vector
- `v.erase(v.begin());`
  - Remover o primeiro elemento do vector
- `v.begin()` é um **iterador**



# std::vector

```
v.erase(v.begin());  
v.erase(v.begin());  
v.insert(v.begin(), 0);  
v.insert(v.begin(), 1);  
v.pop_back();  
v.push_back(0);  
v.push_back(1);  
v.at(2) = 9;
```

// What are the contents of the vector?

# Deque – Double-ended Queue

# Deque – Double-ended Queue



[java2novice.com]

# std::deque

```
#include <deque>
std::deque<some_type>
std::deque<int> elems = {1, 2, 3, 4};
```

- Outra **generalização**
- Elementos contíguos + **Acesso aleatório** + **Tamanho variável**
  - Aumenta ou diminui de tamanho, quando se inserem ou apagam elementos
- Métodos eficientes para realizar **operações** na **frente** e na **cauda**
- Usar quando são frequentes as operações nas duas extremidades !

# std::deque – Operações habituais

- `d.push_back(6);`
  - Inserir elemento na cauda do deque
- `d.pop_back();`
  - Remover elemento da cauda do deque
- `d.push_front(99);`
  - Inserir elemento **antes** do primeiro elemento do deque
- `d.pop_front();`
  - Remover o primeiro elemento do deque
- `d.empty();`
  - Devolve **true**, se o deque não tiver elementos; **false**, no caso contrário

# Exemplo – Capicua ?

```
// Getting each char and pushing it into the back of a deque
```

```
deque<char> d;
```

```
for (char c : original_string) {
```

```
    d.push_back(c);
```

```
}
```

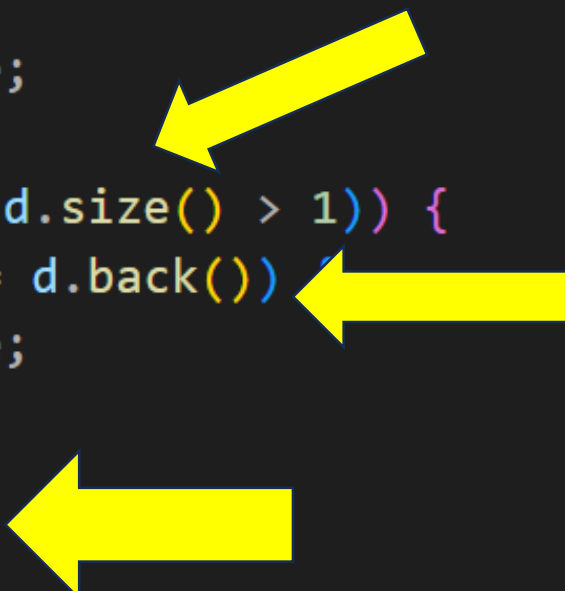
# Exemplo – Capicua ?

```
// Is it a palindrome ?
// Read from both ends of the deque

bool answer = true;

while (answer && (d.size() > 1)) {
    if (d.front() != d.back())
        answer = false;
    else {
        d.pop_front();
        d.pop_back();
    }
}

cout << "The string \"" << original_string << "\"
      << (answer ? " is " : " is NOT ") << "a palindrome !!" << endl;
```



# Container Adaptors

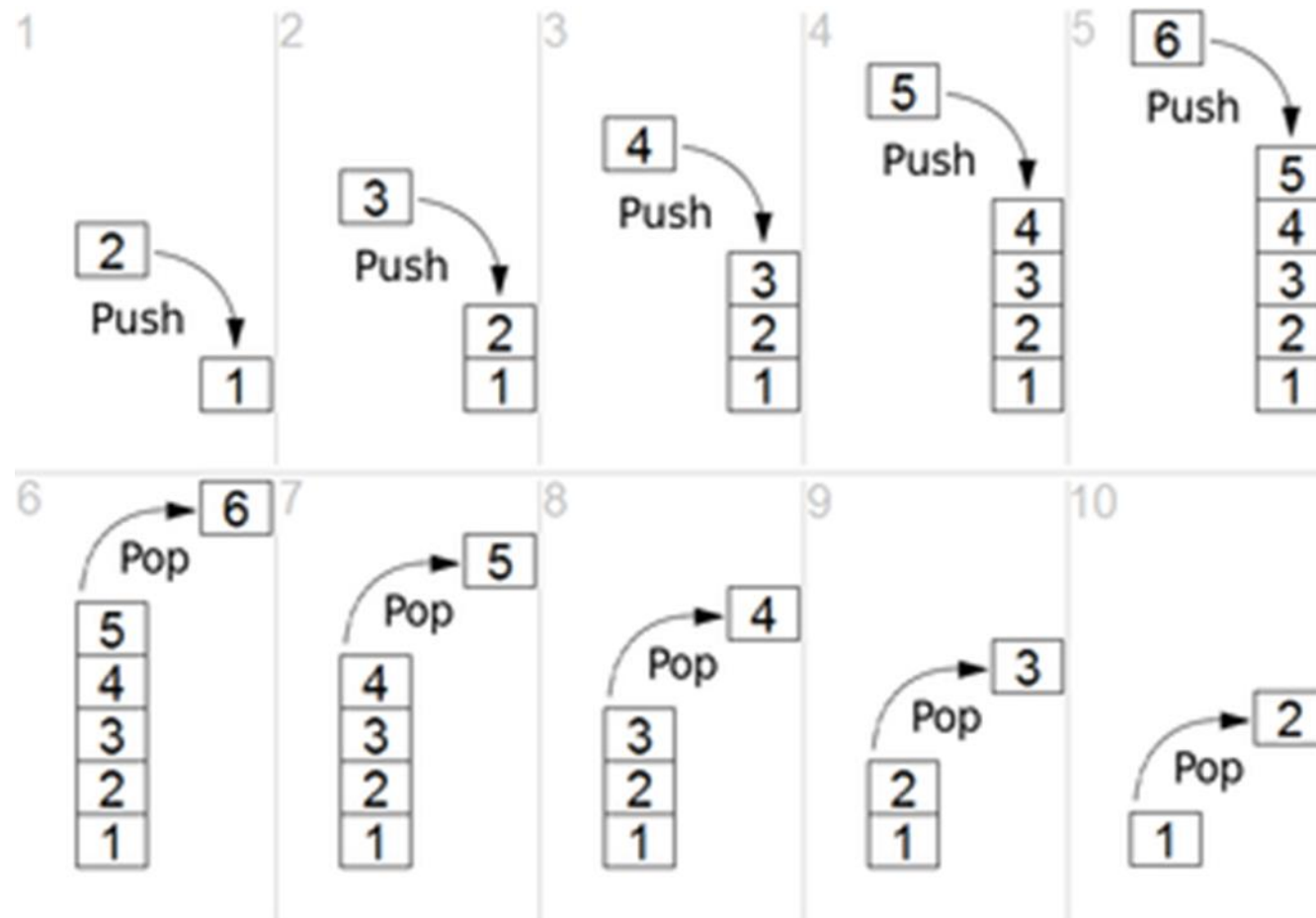


# Container **Adaptors**

- Fornecem uma **API mais restrita**, para um **sequence container**, apenas com algumas funcionalidades
- O sequence container de base está instanciado por omissão (default)
- **stack**
- **queue**
- **priority\_queue**

# Stack – Pilha

# Stack – LIFO: Last-in First-Out



[Wikipedia]

# std::stack

```
#include <stack>
std::stack<some_type>
std::stack<unsigned> s;
```

- **Acesso limitado** ao último element inserido na pilha
- Tamanho variável + **SEM** acesso aleatório

# std::stack – Operações habituais

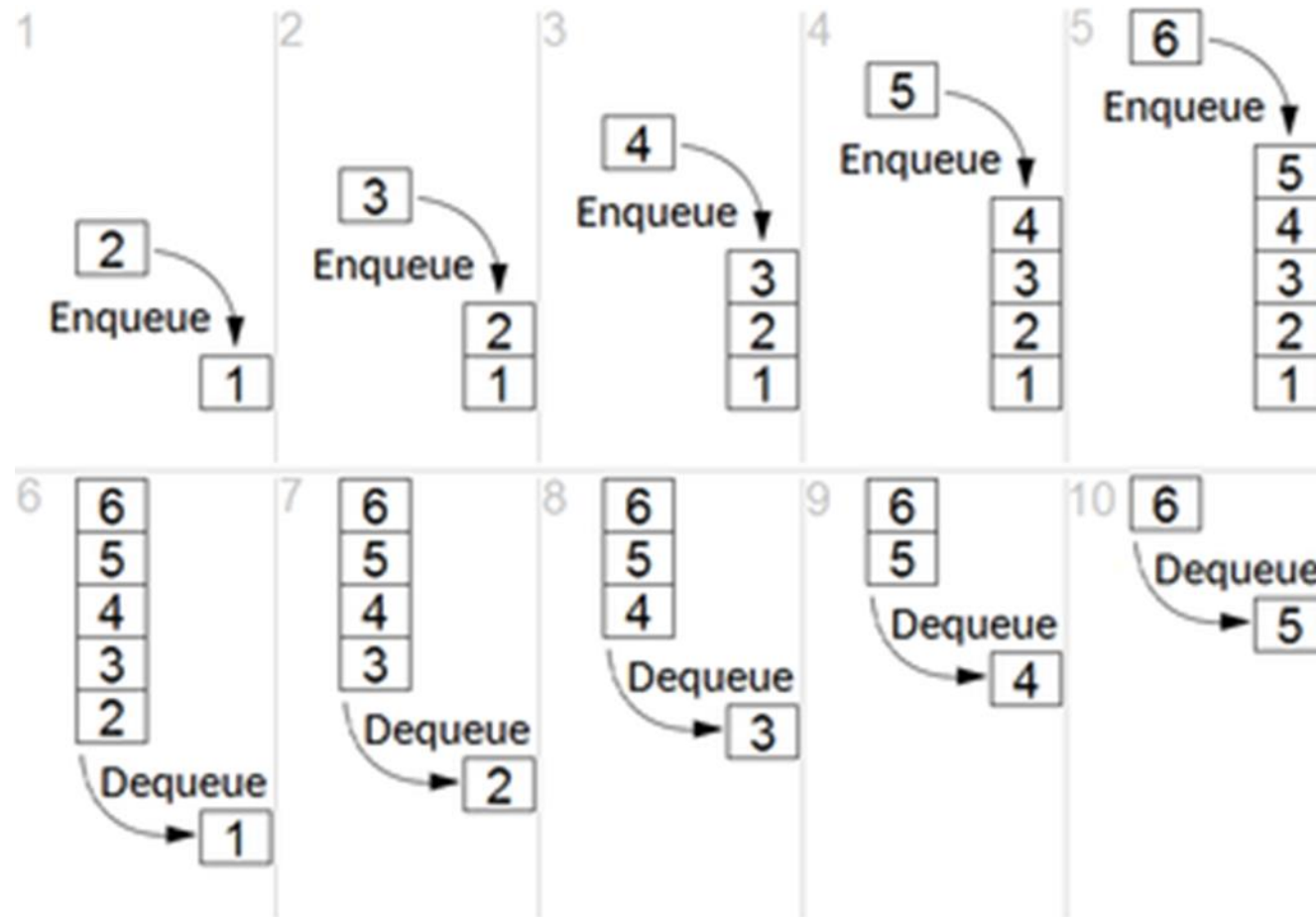
- `s.push(6);`
  - Adicionar elemento no topo da pilha
- `s.top();`
  - Devolve uma referência para o elemento no topo da pilha
- `s.pop();`
  - Remover o elemento do topo da pilha
- `s.size();`
  - Devolver o número de elementos da pilha
- `s.empty();`
  - Devolve **true**, se a pilha não tiver elementos; **false**, no caso contrário

# std::stack

```
stack<size_t> uint_stack;
for (size_t i = 0; i < 10; ++i) {
    uint_stack.push(i);
}
while (!uint_stack.empty()) {
    cout << uint_stack.top() << endl;
    uint_stack.pop();
}
// What is the output ?
```

# Queue – Fila

# Queue – FIFO: First-in First-Out



[Wikipedia]



# std::queue

```
#include <queue>
std::queue<some_type>
std::queue<unsigned> q;
```

- Acesso limitado à frente e à cauda da fila
- Tamanho variável + SEM acesso aleatório

# std::queue – Operações habituais

- `q.front()` ;
  - Devolve uma referência para o primeiro elemento, na frente da fila
- `q.back()` ;
  - Devolve uma referência para o último elemento, na cauda da fila
- `q.pop()` ;
  - Remover o elemento na frente da file
- `q.push(10)` ;
  - Juntar um elemento à cauda da fila
- `q.size()` ;
- `q.empty()` ;

# std::queue

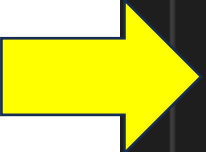
```
queue<size_t> uint_queue;
for (size_t i = 0; i < 10; ++i) {
    uint_queue.push(i);
}
while (!uint_queue.empty()) {
    cout << uint_queue.front() << endl;
    uint_queue.pop();
}
// What is the output ?
```

# Exemplo – Algoritmos pela ordem inversa

```
// Getting each digit and pushing it into a queue

queue<unsigned int> q;
unsigned int digit;

while (original_number > 0) {
    digit = original_number % 10;
    q.push(digit);
    original_number /= 10;
}
```



# Exemplo – Algoritmos pela ordem inversa

```
// New number with digits in reverse order
```

```
unsigned int new_number = 0;
```

```
while (!q.empty()) {
```

```
    digit = q.front();
```

```
    q.pop();
```

```
    new_number = new_number * 10 + digit;
```

```
}
```

```
cout << "In reverse order : " << new_number << endl;
```

# Associative Containers

# Associative Containers

- Implementam estruturas de dados **ordenadas**, de **tamanho variável**, e que suportam uma **procura eficiente** –  **$O(\log n)$**
- **map** : coleção (**dicionário**) de **pares (chave, valor)**, **sem** chaves **repetidas**, e ordenados de acordo com as chaves
- **multimap** : possíveis **chaves repetidas**
- **set** : coleção (**conjunto**) de **elementos ordenados**, e **sem repetições**
- **multiset** : possíveis **elementos repetidos**

# Associative Containers

- O **tipo das chaves** deve suportar o **operador de comparação <**, para permitir a **procura** e assegurar a **ordem** dos itens do **dicionário**
- O mesmo se passa para o **tipo dos elementos** do **conjunto**
- Esse operador é implícito para os tipos pré-definidos: int, double, etc.
- Sempre que as chaves ou elementos sejam instâncias de uma classe, essa classe deve suportar o **operador de comparação <**
- Assegurar implementando, se necessário, o **overloaded operator <** para essa classe
- A igualdade (**==**) é habitualmente inferida usando o operador <
  - P.ex., **!(a < b) && !(b < a)**



# Map – Dicionário Ordenado

# std::map – Dicionário Ordenado

```
#include<map>
std::map<key_type, value_type>
std::map<int, char> m = { {1, 'a'}, {2, 'b'} };
```

- O **valor da chave é único** e usado para **procurar** e **ordenar** os itens
- O **valor** de um item **pode ser diretamente alterado**
- O **valor da chave é constante** e não pode ser alterado

# std::map – Operações habituais

- `m.insert({key, value})`
  - Adicionar o elemento (key, value) ao dicionário
- `m.erase(key)`
  - Remover do dicionário o elemento com a chave dada, caso exista
- `m.count(key)`
  - Devolve o número de itens cuja chave é igual à chave dada (0 ou 1)
- `m.at(key)`
  - **Consulta** o elemento com a chave dada, caso exista
- `m[key]`
  - Consultar / adicionar um elemento com a chave dada

# std::map

```
map<string, unsigned int> m;  
m["February"] = 2;  
m["April"] = 4;  
m.insert({"May", 5});  
m.insert({"January", 1});  
m.insert({"March", 3});  
m["December"] = 12;  
cout << "key = January value = " << m.at("January") << endl;  
cout << "key = February value = " << m["February"] << endl;
```

# Iterar sobre todos os pares (chave, valor)


```
for (const auto& e : m) {  
    // Pair with attributes (first, second)  
    cout << "key = " << e.first  
        << " value = " << e.second << endl;  
}
```

// What is the output?

# Exemplo – Contar o número de ocorrências

```
map<Fraction, unsigned int> fraction_counter;

for (unsigned int i = 0; i < n; ++i) {
    Fraction frac = Fraction(1 + random(RANGE), 1 + random(RANGE));
    ++fraction_counter[frac]; // Great way of doing it !!
                             // If not in the map, insert with value 0
                             // and then update
}
```



# Set – Conjunto Ordenado

# std::set – Conjunto Ordenado

```
#include <set>
std::set<some_type>
std::set<int> elems = {1, 2, 3, 4};
```

- **Não** são permitidos **elementos repetidos!**
- Os elementos são **ordenados** de acordo com o seu valor
- O valor de um elemento não pode ser alterado !
- **Apagar** elementos desnecessários e **inserir** novos valores



# std::set – Operações habituais

- `s.insert(element)`
  - Adicionar um elemento ao conjunto
- `s.erase(element)`
  - Remover um elemento do conjunto
- `s.count(element)`
  - Devolve o número de elementos iguais ao element procurado (0 ou 1)

# std::set – O que acontece ?

```
set<string> s;
```

```
s.insert("February");
```

```
s.insert("April");
```

```
s.insert("May");
```

```
s.insert("January");
```

```
s.insert("March");
```

```
s.insert("December");
```

```
s.insert("March");
```

```
s.erase("March");
```

# Iterar sobre todos os elementos

```
for (const auto& e : s) {  
    cout << e << endl;  
}
```

// What is the output?

# Exemplo – Registrar, sem repetições

```
set<Fraction> fractions_set;

for (unsigned int i = 0; i < n; ++i) {
    Fraction frac = Fraction(1 + random(RANGE), 1 + random(RANGE));
    auto result = fractions_set.insert(frac);
    if (result.second == false) { // Tried to insert a repeated fraction value
        ++repeated_values;
    }
}
```

