UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

Programação e Desenvolvimento de Software 2

Biblioteca padrão C++ (STL)

Prof. Douglas G. Macharet
douglas.macharet@dcc.ufmg.br

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Introdução

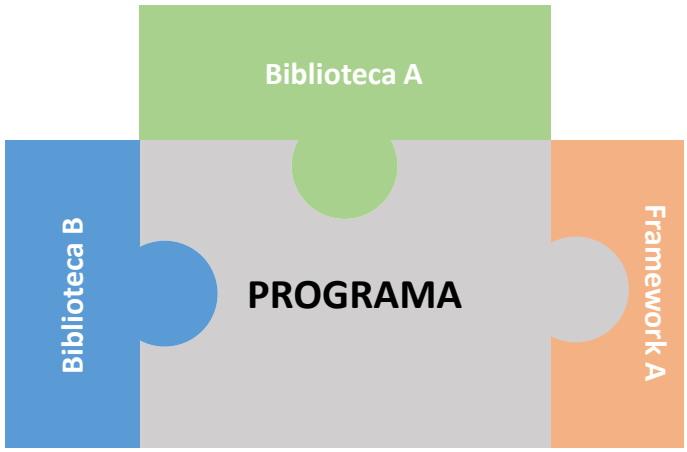
- Quais exemplos de TADs já vimos?
  - Listas encadeadas, Árvores binárias de pesquisa
  - Ponto3D, Aluno, Circunferencia
- Será que existem quantos outros TADs?
  - Infinitos! Você pode definir os seus próprios!
- Sempre vou ter que implementá-los?
  - Não!
  - Bibliotecas *to the rescue*!

Bibliotecas

- Não precisamos escrever tudo "do zero"!
- Bibliotecas
  - Conjunto de implementações de uso geral (TADs, funções)
  - Interface e comportamento bem definidos → Documentação
  - Reutilização de código em diferentes partes e programas

https://en.wikipedia.org/wiki/Library_(computing)

Bibliotecas



Biblioteca padrão C++

- Componentes escritos na linguagem
  - Strings (expressões regulares)
  - Ponteiros inteligentes (unique_ptr, shared_ptr)
  - Entrada/Saída (streams)
  - Funcionalidades numéricas
  - Containers (STL)
  - ...

https://en.wikipedia.org/wiki/C%2B%2B_Standard_Library

Biblioteca padrão C++

- Funcionalidades através de headers

```
#include <string>
#include <iostream>
```

- Espaço de nomes (namespace)
  - std
- Headers da biblioteca padrão de C
  - <stdlib.h> ➔ <cstdlib>

Biblioteca padrão C++

Namespace

```
#include <iostream>

int main() {
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello world!" << endl;
    return 0;
}
```

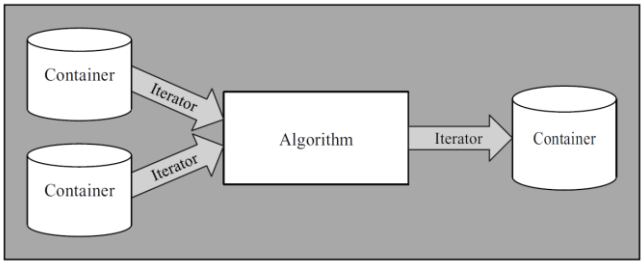
Não é boa prática, mas vamos usar em situações mais simples.

Standard Template Library

- Parte da ISO C++ Standard Library
  - Soluções para problemas que envolvem estruturas de dados
  - Componentes bem escritos e bem testados
- Componentes

Containers | **Iteradores** | **Algoritmos**

Standard Template Library



- **Container**: representa como os dados são armazenados
- **Iterador**: auxilia no acesso e manipulação ("ponteiros")
- **Algoritmo**: procedimento que pode ser aplicado aos dados

Fonte: *The C++ Standard Library: A Tutorial and Reference*.

Standard Template Library

- Programação Genérica
  - A mesma definição de função atua sobre diferentes tipos
    - Provê maior liberdade e flexibilidade para a implementação
  - Polimorfismo universal – Paramétrico
    - Os tipos são passados como parâmetros
- Linguagem
  - Templates (C++), Generics (Java)

Standard Template Library

```
struct NodeI {

  int   data;
  NodeI* next;

};
```

```
struct NodeA {

  Aluno data;
  NodeA* next;

};
```

```
template <typename T>
class NodeG {

  T data;
  NodeG* next;

};
```

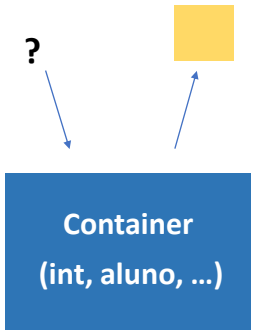
Veremos com mais detalhes na parte de polimorfismo!

Standard Template Library
Containers

- Estruturas de dados que armazenam coleções
  - Todos os elementos são do mesmo tipo
  - Primitivos/definidos pelo programador (TADs)
- Praticamente todos os TADs são containers
  - Tipos específicos (estrutura rígida)
  - Tipos genéricos (podem ser reutilizados)

Standard Template Library
Containers

- Acesso uniforme aos dados (contrato)
  - Independente do tipo do elemento
  - Independente de como está armazenado
- Recuperação dos dados
  - Índice (N-ésimo elemento)
  - Valor (Elemento com valor "João")
  - Propriedades (Elemento com "idade > 18")



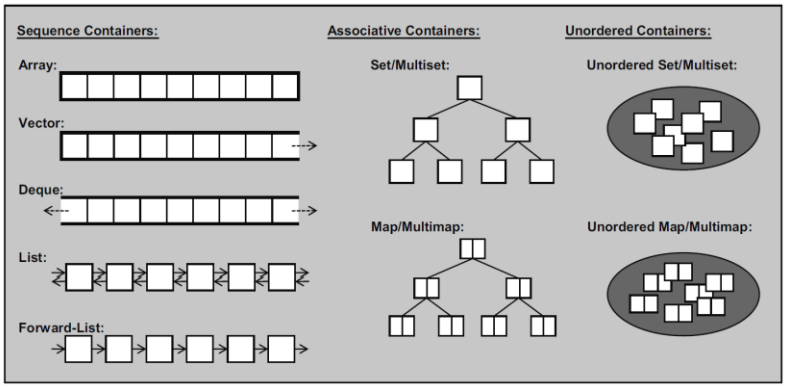
Standard Template Library
Containers

- Organização de dados
  - Acesso rápido, exibição, ...
- Operações (padronização)
  - Adicionar, remover, ordenar, buscar, ...
- Implementação
  - Correta, eficiente, documentação, ...

Standard Template Library
Containers

Sequenciais	Associativos	Adaptadores
• Vector • Deque • List	• Set • Map • Multiset • Multimap	• Stack • Queue • Priority queue

Standard Template Library
Containers



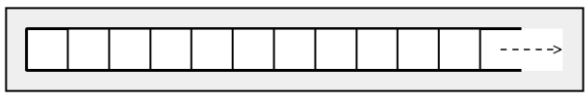
Fonte: *The C++ Standard Library: A Tutorial and Reference*.

Standard Template Library
Containers – Sequenciais

- Elementos estão em uma **ordem linear**
  - Precedido por um elemento específico e seguido por outro
- Gerenciamento automático de memória, o que permite que o tamanho (alocação) possa variar dinamicamente
- Exemplos de funções
  - `front(), back()`
  - `push_back(), pop_back()`
  - `size(), empty()`

Standard Template Library
Vector

- Array genérico e dinamicamente redimensionável



- Vantagens
  - Acesso individual aos elementos (índice)
  - Adição/remoção de elementos no final*
  - Percorrer em uma ordem específica

https://en.cppreference.com/w/cpp/container/vector

Standard Template Library
Vector – Exemplo 1

Include da especificação

Informamos o container e o tipo que queremos armazenar

```
#include <iostream>
#include <vector>

int main() {

  std::vector<int> v = {7, 5, 16, 8};

  v.push_back(25);
  v.push_back(13);

  for(int n : v) {
    std::cout << n << std::endl;
  }

  return 0;
}
```

Adicionando elementos ao final da sequência

```
for (int i = 0; i < v.size(); i++)
  int n = v[i];
```

Wandbox

Standard Template Library
Vector – Exemplo 2

- E se eu quisesse guardar "Pessoas"?

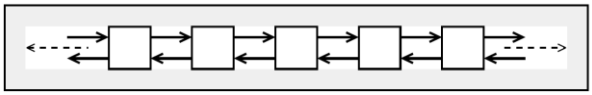
```
struct Pessoa {
  string nome;
  int idade;
};
```

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  Pessoa p1;
  p1.nome = "Ana";
  p1.idade = 25;

  Pessoa p2;
  p2.nome = "Pedro";
  p2.idade = 18;

  Pessoa p3;
  p3.nome = "Douglas";
  p3.idade = 30;

  vector<Pessoa> pessoas;
  pessoas.push_back(p1);
  pessoas.push_back(p2);
  pessoas.push_back(p3);

  // Primeira forma de acesso
  cout << pessoas[0].nome << endl;
  cout << pessoas[1].nome << endl;

  // Segunda forma, com at
  cout << pessoas.at(2).nome << endl;

  return 0;
}
```

Wandbox

Standard Template Library
List

- Lista duplamente encadeada



- Não possui acesso via índice: **Iterador**
- Adição/remoção é mais eficiente
  - Considerando-se que já se sabe a posição
  - Não é necessário mover outros elementos

https://en.cppreference.com/w/cpp/container/list

Standard Template Library
List – Exemplo 1

O iterator é um *ponteiro* que ajuda no acesso aos elementos

```
#include <iostream>
#include <list>

int main() {
  std::list<int> l = {7, 5, 16, 8};

  // Adicionar um número inteiro no início (frente) da lista
  l.push_front(25);
  // Adicionar um número inteiro no final da lista
  l.push_back(13);

  for (std::list<int>::iterator it=l.begin(); it != l.end(); ++it) {
    std::cout << *it << std::endl;
  }

  return 0;
}
```

Pythontutor

Standard Template Library
List – Exemplo 2

```
struct TAD {
  int x;
};
```

```
for(TAD *t : lista_ptr)
  delete t;
```

```
#include <iostream>
#include <list>

int main() {

  std::list<TAD> lista_ref;
  std::list<TAD*> lista_ptr;

  TAD t1;
  t1.x = 10;
  lista_ref.push_back(t1);

  TAD* t2 = new TAD();
  t2->x = 20;
  lista_ptr.push_back(t2);

  t1.x = 99;
  t2->x = 99;

  std::cout << lista_ref.front().x << std::endl;
  std::cout << lista_ptr.front()->x << std::endl;

  return 0;
}
```

10
99

Pythontutor

Standard Template Library
Containers

Containers armazenam os dados por **valor**, não por referência.

- Quando você insere um objeto, o container faz uma cópia
- Se o container precisar reorganizar objetos, ele faz cópias

Standard Template Library
Containers

- Ponteiros em containers: usar ou não usar?

Usar	Não usar
Manipulação será feita corretamente.	Você consegue criar os objetos da sua estrutura sem a necessidade de realocar memória.
Elementos tem custo computacional alto para serem criados.	Para criar um objeto da sua estrutura você não realiza operações custosas.
Você deseja usar polimorfismo.	Não existem ponteiros para outras estruturas.

* Smart pointers podem ajudar nessas questões: https://en.cppreference.com/book/intro/smart_pointers

Standard Template Library
Containers

```
struct Ponto {
  float x;
  float y;
};

void doSomethingWithList(list<Ponto> lista) {
  //Faz alguma coisa
}
```

Existe algum problema se essa lista for muito grande?

Standard Template Library
Containers

```
struct Ponto {
  float x;
  float y;
};

void doSomethingWithList(list<Ponto> const& lista) {
  //Faz alguma coisa
}
```

Para evitar criar uma cópia, passar por referência.

Quando a função não deve ser capaz de alterar o argumento usar const.

Standard Template Library
Vector vs. List

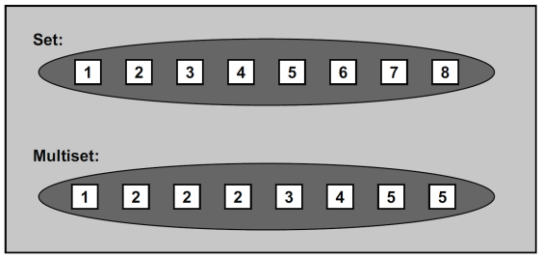
- Quando utilizar Vector ou List?
  - Vector
    - Tipo de sequência que deve ser usado por padrão
    - Muitos acessos em posições aleatórias da sequência
  - List
    - Muitas inserções e remoções que não serão no final
- Diferenças relacionadas a custo computacional
  - Assunto para outra disciplina!

Standard Template Library
Containers – Associativos

- Elementos não possuem ordem específica (inserção)
- Projetados para suportar o acesso direto aos elementos usando **chaves** pré-determinadas (informadas)
  - Dados armazenados (ordenados) em uma BST
- Exemplos de funções
  - `insert(), erase()`
  - `find(), count()`

Standard Template Library
Set

- Guarda uma coleção de elementos **distintos**
  - Semelhante ao conceito matemático de conjunto
  - Comparáveis de acordo com algum critério



https://en.cppreference.com/w/cpp/container/set

Standard Template Library
Set – Exemplo 1

```
#include <iostream>
#include <set>

int main() {

  std::set<int> s;

  for(int i = 10; i >= 1; i--) {
    s.insert(i);
  }

  s.insert(7);

  for(int i = 2; i <= 10; i += 2) {
    s.erase(i);
  }

  std::cout << "(" << s.size() << ")" << std::endl;
  for (int e : s) {
     std::cout << e << std::endl;
  }

  return 0;
}
```

Insere inteiros de 10 a 1

Não faz nada! O elemento já está no conjunto

Remove números pares

1 3 5 7 9

Wandbox

Standard Template Library
Set – Exemplo 2

```
#include <iostream>
#include <set>

int main() {

  std::set<int> s = {1, 3, 5, 7, 9};

  if(s.find(6) != s.end()) {
    // Element in set
  } else {
    // Element not presents in set
    s.insert(6);
  }

  if(s.insert(6).second) {
    // Element INSERTED in set
  } else {
    // Element NOT INSERTED in set
  }

  for (int e : s) {
     std::cout << e << std::endl;
  }

  return 0;
}
```

Podemos verificar antes de inserir se o elemento já está no conjunto

O tipo de retorno é uma estrutura *pair* (dois valores), e o segundo item informa se a inserção foi mesmo feita

Standard Template Library
Set – Exemplo 3

- E se eu quiser um conjunto de "Pessoas"?

```
struct Pessoa {
  string nome;
  int idade;
};
```

```
// Comparador verifica apenas < (outros baseados nele)
struct compara_pessoa_f {
  bool operator()(const Pessoa& p1, const Pessoa& p2) {
    return p1.idade < p2.idade;
  }
};
```

Criamos uma função para comparar nossos elementos.
Essa função é um tipo especial chamado *functor* (veja link).

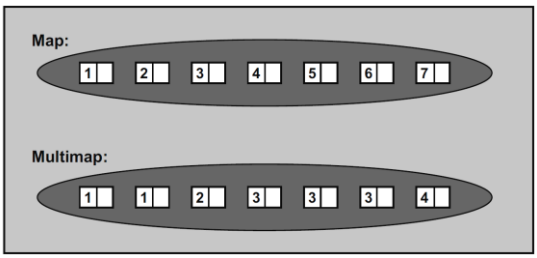
```
#include <iostream>
#include <set>

using namespace std;

int main() {

  std::set<Pessoa, compara_pessoa_f> pessoas;

  pessoas.insert({"Douglas", 30});
  pessoas.insert({"Pedro", 18});
  pessoas.insert({"Ana", 25});

  for (Pessoa p : pessoas)
    std::cout << p.nome << std::endl;

  return 0;
}
```

Código online

https://docs.microsoft.com/pt-br/cpp/standard-library/function-objects-in-the-stl?view=msvc-160

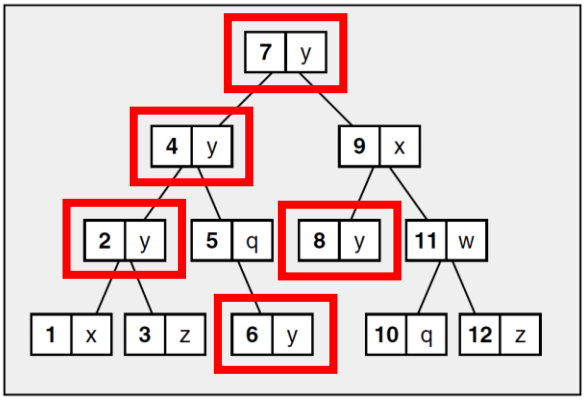
Standard Template Library
Map

- Array associativo ou dicionário
  - Conjunto de **pares**: <chave, valor>
  - As chaves devem ser distintas (valor pode ser igual)



https://en.cppreference.com/w/cpp/container/map

Standard Template Library
Map



Standard Template Library
Map

Informar os tipos da **chave** e **valor**

Duas formas de inserção

Prefira utilizar find/insert!

Iterator considera o tipo combinado do elemento

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

int main() {

  std::map<int,std::string> m;

  m.insert(std::pair<int,std::string>(2017123456, "Joao"));

  m[2016123456] = "Maria";
  m[2018123456] = "Carlos";
  m[2015123456] = "Jose";
  m[2014123456] = "Joana";

  std::map<int,std::string>::iterator it;
  for (it = m.begin(); it != m.end(); it++) {
    std::cout << it->first << ": " << it->second << std::endl;
  }

  return 0;
}
```

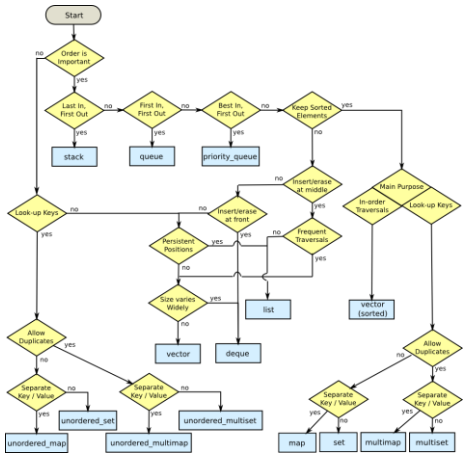
2014123456: Joana
2015123456: Jose
2016123456: Maria
2017123456: Joao
2018123456: Carlos

Wandbox

Standard Template Library
Set vs. Map

- Quando utilizar Set ou Map?
  - Set
    - Coleção de dados de chaves exclusivas, sem nenhum valor associado
    - Elemento não pode ser modificado (const) → Remover/Inserir
  - Map
    - Coleção de dados de chaves exclusivas e algum valor associado
    - Não podemos modificar a chave, mas podemos alterar o valor

Standard Template Library
Containers



https://stackoverflow.com/questions/471432/in-which-scenario-do-i-use-a-particular-stl-container

Standard Template Library
Iterators

- Objetos usados para acessar os elementos armazenados
  - Iteradores **apontam** para itens que estão em um container
  - Podemos "iterar" sobre os elementos (percorrer)
- Exemplos de funções
  - `begin(), end()`
- Exemplos de operações
  - `it++`
  - `*it` (acesso ao elemento)

Nem todos os containers suportam iteradores, e nem todo iterador suporta todas as operações disponíveis!

http://www.cplusplus.com/reference/iterator/

Standard Template Library
Iterators – Exemplo 1

Declaração do iterator a partir do tipo do container

```
#include <iostream>
#include <vector>

int main() {

  std::vector<int> v;

  v.push_back(5);
  v.push_back(2);
  v.push_back(9);

  std::vector<int>::iterator it;
  for (it = v.begin(); it != v.end(); ++it)
      std::cout << *it << std::endl;

  return 0;
}
```

Limites

Avança o iterator para o próximo elemento

Elemento atual apontado pelo iterator

Standard Template Library
Iterators – Exemplo 2

A partir de C++11 o compilador é capaz de inferir tipos na inicialização. Isso pode nos ajudar, mas devemos usar com moderação!

Últil quando temos tipos complicados e com baixa legibilidade. Usamos a keyword **auto**.

```
std::map<int,std::string>::iterator it;
```

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

int main() {

  std::map<int,std::string> m;

  m[2017123456] = "Joao";
  m[2016123456] = "Maria";
  m[2018123456] = "Carlos";
  m[2015123456] = "Jose";
  m[2014123456] = "Joana";

  for (auto it = m.begin(); it != m.end(); it++) {
    std::cout << it->first << ": " << it->second << std::endl;
  }

  return 0;
}
```

https://en.cppreference.com/w/cpp/language/auto

Standard Template Library
Iterators – Exemplo 3

O *for each* faz uma cópia do elemento, o que pode ser um problema

Solução é fazer um laço por referência

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {

  std::vector<int> v;

  v.push_back(5);
  v.push_back(2);
  v.push_back(9);

  for (int &i : v)
      std::cout << i << std::endl;

  return 0;
}
```

https://en.cppreference.com/w/cpp/language/range-for

Standard Template Library
Algorithms

- Algoritmos a serem usados em intervalos de elementos
  - **Range**: qualquer sequência que pode ser acessada por meio de iteradores ou ponteiros, como matrizes ou alguns containers

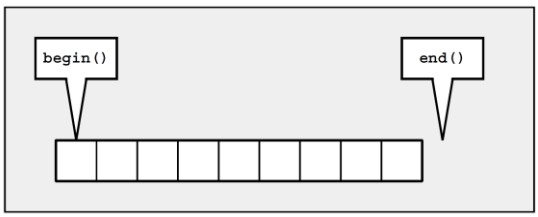
```
algorithm(begin, end, ...);
```

- Exemplos de algoritmos
  - `find(), count(), max_element()`
  - `sort(), swap(), reverse()`

http://www.cplusplus.com/reference/algorithm/

Standard Template Library
Algorithms

- O range considera o intervalo semi-aberto: [first, last)
  - *Last* é o elemento exatamente depois do range desejado
- **Atenção**: em containers, o `begin()` aponta para o primeiro elemento, e o `end()` para a posição **após** o último elemento



Fonte: *The C++ Standard Library: A Tutorial and Reference*.

Standard Template Library
Algorithms – Exemplo 1

Encontra o maior elemento

Ordena os elementos

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {

  std::vector<int> v = {23, 7, 5, 16, 8, 1, 12, 20, 10};

  for(int &n : v)
    std::cout << n << std::endl;

  int max = *std::max_element(v.begin(), v.end());
  std::cout << "Max: " << max << std::endl;

  std::cout << "Sort:" << std::endl;
  std::sort(v.begin(), v.end());
  for(int &n : v)
    std::cout << n << std::endl;

  return 0;
}
```

Wandbox

Considerações finais

- Breve introdução dos componentes da STL
  - Existem outros tipos de containers, algoritmos, funções, …
- Verifique a documentação para entender a utilização
- Foco principal na utilização de código existente
- Faça exercícios e pratique todos os conceitos vistos!