Recursos Avançados de C++

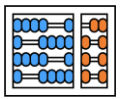
Módulo 3

Prof. Dr. Bruno B. P. Cafeo

Instituto de Computação
Universidade Estadual de Campinas

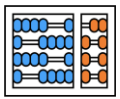
Agenda

- STL Containers
- STL Iterators
- STL Algorithms
- STL Functors

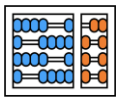


Introdução

- A Standard Template Library (STL) é um conjunto de ferramentas poderosas e reutilizáveis em C++.
- Oferece estruturas de dados e algoritmos prontos para uso, simplificando o desenvolvimento de software.
- Três pilares fundamentais da STL: containeres, algoritmos e iteradores.
- A STL é uma ferramenta indispensável que aumenta a eficiência e a produtividade do programador em C++.



STL Containers

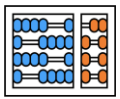


UNICAMP

Containers STL

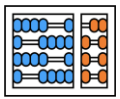
- Os Containers STL (Standard Template Library) são classes de templates que implementam estruturas de dados úteis, como arrays dinâmicos, mapas hash, listas encadeadas, árvores, etc.

- Esses containers permitem que os programadores armazenem e manipulem dados.



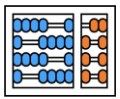
Tipos

- **Containers Sequenciais:** Implementam estruturas de dados com acesso sequencial.
- **Container Adapters:** Implementam estruturas como filas e pilhas, fornecendo interfaces diferentes para containers sequenciais.
- **Containers Associativos:** Usados para armazenar dados ordenados que podem ser pesquisados rapidamente usando a ideia de chave-valor.
- **Containers Não Ordenados:** Semelhantes aos containers associativos, exceto que eles não armazenam dados classificados, mas ainda fornecem um tempo de pesquisa rápido usando pares chave-valor.



Containers Sequenciais

- `Vector (std::vector)`: Um array dinâmico.
- `List (std::list)`: Implementa uma lista duplamente encadeada.
- `Deque (std::deque)`: Implementa uma fila duplamente terminada.
- `Array (std::array)`: Representa um array de tamanho fixo.
- `Forward List (std::forward_list)`: Implementa uma lista encadeada simples.

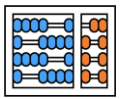


STL Vector

```
std::vector<data_type> vector_name;
```

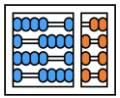
Função	Descrição	Complexidade Temporal
`begin()`	Retorna um iterador para o primeiro elemento.	O(1)
`end()`	Retorna um iterador para o elemento teoricamente após o último elemento.	O(1)
`size()`	Retorna o número de elementos presentes.	O(1)
`empty()`	Retorna true se o vetor estiver vazio, senão false.	O(1)
`at()`	Retorna o elemento em uma posição específica.	O(1)
`assign()`	Atribui um novo valor aos elementos do vetor.	O(n)
`push_back()`	Adiciona um elemento ao final do vetor.	O(1)
`pop_back()`	Remove um elemento do final.	O(1)
`insert()`	Insere um elemento em uma posição específica.	O(n)
`erase()`	Apaga os elementos em uma posição ou intervalo específico.	O(n)
`clear()`	Remove todos os elementos.	O(n)

Referência: https://cplusplus.com/reference/vector/vector/



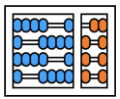
STL Vector

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    // Vetor 1D com inicialização
    vector<int> v1 = {1, 2, 3, 4, 5};
    // Vetor 2D com tamanho e inicialização de elementos
    vector<vector<int>> v2(3, vector<int>(3, 5));

    // Adicionando valores usando push_back()
    v1.push_back(6);
    // Imprimindo v1 usando size()
    cout << "v1: ";
    for (int i = 0; i < v1.size(); i++) {
        cout << v1[i] << " ";
    }
    cout << endl;

    v1.erase(v1.begin() + 4);
    // Imprimindo v1 usando iteradores
    cout << "v1: ";
    for (auto i = v1.begin(); i != v1.end(); i++) {
        cout << *i << " ";
    }

    // Imprimindo v2 usando loop baseado em intervalo
    cout << "v2:-" << endl;
    for (auto i : v2) {
        for (auto j : i) {
            cout << j << " ";
        }
        cout << endl;
    }
    return 0;
}
```



Container Adapters

- `Stack (std::stack)`: Implementa uma pilha (LIFO).
- `Queue (std::queue)`: Implementa uma fila (FIFO).
- `Priority Queue (std::priority_queue)`: Implementa uma fila de prioridade.

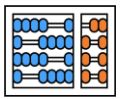


STL Stack

```
std::stack<data_type> stack_name;
```

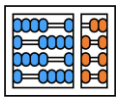
Função	Descrição	Complexidade Temporal
`empty()`	Retorna true se a pilha estiver vazia, senão false.	O(1)
`size()`	Retorna o número de elementos na pilha.	O(1)
`top()`	Retorna o elemento do topo.	O(1)
`push(g)`	Adiciona um elemento na pilha.	O(1)
`pop()`	Remove um elemento da pilha.	O(1)

Referência: https://cplusplus.com/reference/stack/stack/



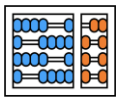
STL Stack

```cpp
using namespace std;

int main() {
    stack<int> s;

    for (int i = 1; i <= 5; i++) {
        s.push(i);
    }

    s.push(6);
    // Verificando o elemento do topo
    cout << "s.top() = " << s.top() << endl;

    // Obtendo todos os elementos
    cout << "s: ";
    while (!s.empty()) {
        cout << s.top() << " ";
        s.pop();
    }

    // Tamanho após remover todos os elementos
    cout << "Tamanho Final: " << s.size();

    return 0;
}
```



Containers Associativos

- `Set (std::set)`: Armazena valores únicos em ordem crescente.
- `Multiset (std::multiset)`: Armazena valores duplicados em ordem crescente.
- `Map (std::map)`: Armazena pares chave-valor em ordem crescente das chaves.
- `Multimap (std::multimap)`: Armazena múltiplos pares chave-valor com chaves duplicadas.

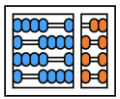


STL Map

```
map<key_type, value_type> map_name;
```

Função	Descrição	Complexidade Temporal
`begin()`	Retorna um iterador para o primeiro elemento.	O(1)
`end()`	Retorna um iterador para o elemento teoricamente após o último elemento.	O(1)
`size()`	Retorna o número de elementos no mapa.	O(1)
`insert()`	Adiciona um novo elemento ao mapa.	O(log n)
`erase(iterator)`	Remove o elemento na posição apontada pelo iterador.	O(log n)
`erase(key)`	Remove a chave e seu valor do mapa.	O(log n)
`clear()`	Remove todos os elementos do mapa.	O(n)
`find()`	Retorna o ponteiro para o elemento fornecido se presente, senão um ponteiro para o final.	O(log n)

Referência: https://cplusplus.com/reference/map/map/

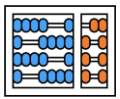


STL Map

```cpp
// Programa C++ para ilustrar o mapa
#include <iostream>
#include <map>
using namespace std;

int main() {
    // Criando um objeto std::map
    map<int, string> m;

    // Adicionando pares chave-valor
    m[1] = "ONE";
    m[2] = "TWO";
    m[3] = "THREE";

    // Verificando o tamanho
    cout << "Tamanho do mapa m: " << m.size() << endl;

    // Inserindo usando inserção de pares
    m.insert({4, "FOUR"});

    // Procurando uma chave
    map<int, string>::iterator it = m.find(2);
    if (it != m.end()) {
        cout << "Chave 2 encontrada, Valor = " << it->second <<
        endl;
    } else {
        cout << "Chave 2 não encontrada" << endl;
    }

    // Removendo uma chave
    m.erase(3);

    // Percorrendo o mapa e imprimindo todos os pares chave-valor
    for (const auto &pair : m) {
        cout << pair.first << ": " << pair.second << endl;
    }

    return 0;
}
```

Containers Não Ordenados

- `Unordered Set (std::unordered_set)`: Armazena valores únicos sem ordenação.
- `Unordered Multiset (std::unordered_multiset)`: Armazena valores duplicados sem ordenação.
- `Unordered Map (std::unordered_map)`: Armazena pares chave-valor sem ordenação.
- `Unordered Multimap (std::unordered_multimap)`: Armazena múltiplos pares chave-valor com chaves duplicadas sem ordenação.

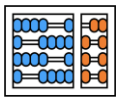


STL Unordered set

```
unordered_set<data_type> unordered_set_name;
```

Função	Descrição	Complexidade Temporal
`begin()`	Retorna um iterador para o primeiro elemento.	O(1)
`end()`	Retorna um iterador para o elemento teoricamente após o último elemento.	O(1)
`size()`	Retorna o número de elementos.	O(1)
`empty()`	Verifica se o container está vazio.	O(1)
`insert()`	Insere um único elemento.	O(1)
`erase()`	Remove o elemento fornecido.	O(1)
`clear()`	Remove todos os elementos.	O(n)
`find()`	Retorna o ponteiro para o elemento fornecido se presente, senão um ponteiro para o final.	O(1)

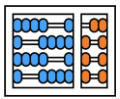
Referência: https://cplusplus.com/reference/unordered_set/unordered_set/



STL Unordered set

```cpp
// Programa C++ para ilustrar o mapa
#include <iostream>
#include <map>
using namespace std;

int main() {
    // Criando um objeto std::map
    map<int, string> m;

    // Adicionando pares chave-valor
    m[1] = "ONE";
    m[2] = "TWO";
    m[3] = "THREE";

    // Verificando o tamanho
    cout << "Tamanho do mapa m: " << m.size() << endl;

    // Inserindo usando inserção de pares
    m.insert({4, "FOUR"});

    // Procurando uma chave
    map<int, string>::iterator it = m.find(2);
    if (it != m.end()) {
        cout << "Chave 2 encontrada, Valor = " << it->second <<
        endl;
    } else {
        cout << "Chave 2 não encontrada" << endl;
    }

    // Removendo uma chave
    m.erase(3);

    // Percorrendo o mapa e imprimindo todos os pares chave-valor
    for (const auto &pair : m) {
        cout << pair.first << ": " << pair.second << endl;
    }

    return 0;
}
```

STL Iterators



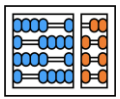
UNICAMP

Containers Iterator

- Iteradores são objetos utilizados para navegar através dos elementos armazenados em contêineres ou sequências.

- Os iteradores fornecem uma interface padronizada para percorrer e acessar os elementos sem a necessidade de conhecer os detalhes internos da estrutura de dados subjacente.

Referência: https://en.cppreference.com/w/cpp/iterator



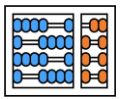
Containers Iterator

- Iteradores de Início (begin) e Fim (end): São utilizados para percorrer todo o conteúdo de um contêiner. Um iterador de início aponta para o primeiro elemento, enquanto um iterador de fim aponta para a posição após o último elemento. Eles são usados principalmente em loops para percorrer o contêiner inteiro.

- Iteradores de Inserção (inserter): São utilizados com contêineres que suportam inserções eficientes, como listas. Eles permitem inserir elementos em posições específicas dentro do contêiner.

- Iteradores de Remoção (eraser): São usados para remover elementos de contêineres, como conjuntos e mapas. Eles ajudam a apagar elementos com base em critérios específicos.

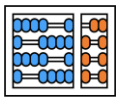
- Iteradores Constantes (const_iterator): São usados para percorrer contêineres de maneira "só leitura". Eles não permitem a modificação dos elementos, o que é útil para garantir que os dados permaneçam inalterados durante a iteração.



Iterador de Entrada (Input Iterator)

- Usado para operações de entrada de única passagem.
- Apenas para acessar operações de leitura, não para atribuição.
- Não pode ser decrementado.
- Um elemento só pode ser acessado uma vez.
- Possuem capacidade limitada e estão na parte mais baixa da hierarquia de iteradores.

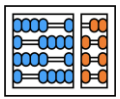
- Exemplo: `istream_iterator`



Iterador de Saída (Output Iterator)

- Usado para operações de saída de única passagem.
- Apenas para fins de atribuição (operações de escrita).
- Um elemento só pode ser acessado uma vez.
- Não pode ser decrementado.
- Também estão na parte mais baixa da hierarquia de iteradores.

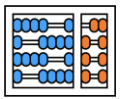
- Exemplo: `ostream_iterator`



Iterador Avançado (Forward Iterator)

- Combinam características de iteradores de entrada e saída.
- Usado para operações de leitura e escrita.
- Não pode ser decrementado, movendo-se apenas em uma direção.
- Movimenta-se sequencialmente, avançando um passo de cada vez.
- Estão em uma hierarquia superior em relação aos iteradores de entrada e saída.

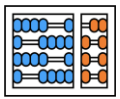
- Exemplo: `forward_list::iterator`



Iterador Bidirecional (Bi-Directional Iterator)

- Possuem todas as características dos iteradores avançados, com algumas adições
- Movem-se tanto em direção à frente quanto para trás.
- Usados para operações de leitura e escrita.

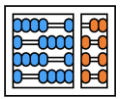
- Exemplos de iteradores bidirecionais incluem `map::iterator`, `set::iterator`, `multiset::iterator` e `multimap::iterator`



Iterador de Acesso Aleatório (Random Access Iterator)

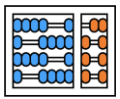
- Os iteradores de acesso aleatório são os mais poderosos.
- Possuem todas as características dos outros tipos de iteradores.
- Podem se mover tanto em direção à frente quanto para trás.
- Permitem operações de leitura e escrita.
- Podem acessar qualquer ponto no contêiner de forma aleatória.

- Exemplos incluem `vector::iterator e array::iterator`

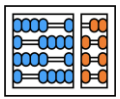


Iteradores

CONTAINER	TYPES OF ITERATOR SUPPORTED
Vector	Random-Access
List	Bidirectional
Deque	Random-Access
Map	Bidirectional
Multimap	Bidirectional
Set	Bidirectional
Multiset	Bidirectional
Stack	No iterator Supported
Queue	No iterator Supported
Priority-Queue	No iterator Supported



STL Algorithms



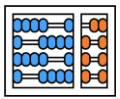
UNICAMP

STL Algorithms

- Ao utilizar os algoritmos da STL, os programadores podem economizar tempo e reduzir a probabilidade de erros, pois muitas operações comuns já foram implementadas de forma otimizada.

- Esses algoritmos são encontrados no cabeçalho `<algorithm>`

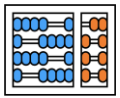
Referência: https://www.geeksforgeeks.org/algorithms-library-c-stl/



STL sort

```
sort (beginIterator, endIterator);
```

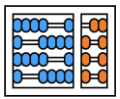
```
1  int main() {
2      int arr[] = {1, 5, 8, 9, 6, 7, 3, 4, 2, 0};
3      int n = sizeof(arr) / sizeof(arr[0]);
4
5      sort(arr, arr + n);
6
7      cout << "\nArray after sorting using default sort is: \n";
8      for (int i = 0; i < n; ++i)
9          cout << arr[i] << " ";
10
11     return 0;
12 }
13
```



STL sort

```
sort (beginIterator, endIterator, comparator);
```

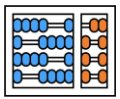
```cpp
#include <iostream>
#include <vector>
#include <algorithm>

// Função de comparação personalizada para ordenar pares antes de ímpares
bool customComparator(int a, int b) {
    if (a % 2 == 0 && b % 2 == 1) {
        return true;  // a é par e b é ímpar, a deve vir antes de b
    }
    if (a % 2 == 1 && b % 2 == 0) {
        return false; // a é ímpar e b é par, b deve vir antes de a
    }
    return a < b;     // mesmo tipo de número, ordene em ordem crescente
}

int main() {
    std::vector<int> numbers = {1, 4, 2, 7, 6, 9, 8, 3, 5};

    // Usando a função de comparação personalizada para ordenar
    std::sort(numbers.begin(), numbers.end(), customComparator);

    // Exibindo o vetor ordenado
    std::cout << "Vetor após a ordenação personalizada: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }

    return 0;
}
```



STL copy

```
copy(beginIterator, endIterator, destIterator);
```

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

using namespace std;

int main() {
    vector<int> v = {1, 2, 3, 4, 5};
    std::vector<int> novo;

    copy(v.begin(), v.end(), std::inserter(novo, novo.end()));

    // Exibindo os elementos no vetor novo
    for (int num : novo) {
        std::cout << num << " ";
    }
    return 0;
}
```

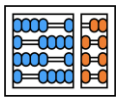


UNICAMP

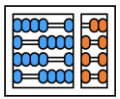
STL max_element

```
max_element(firstIterator, lastIterator);
```

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> v = {10, 88, 2, 9, 45, 82, 546, 42, 221};

    auto max = max_element(begin(v), end(v));

    cout << "Maximum Element: " << *max << "\n";
    return 0;
}
```



STL Functors



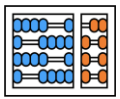
UNICAMP

STL Functors

- Functors, também conhecidos como objetos funções, são objetos que se comportam como funções.

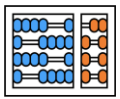
- Em C++, um functor é uma instância de uma classe que sobrecarrega o operador de chamada de função `operator()`.

- Eles são usados para criar objetos que podem ser chamados como funções.



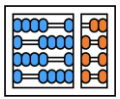
Definindo um Functor

```
1  class MeuFunctor {
2  public:
3      int operator()(int x) {
4          return x * 2;
5      }
6  };
7
```



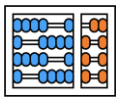
Definindo um Functor

```
1  class MeuFunctor {
2  public:
3      int operator()(int x) {
4          return x * 2;
5      }
6  };
7
```



Usando um Functor

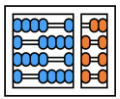
```
1  MeuFunctor meuFunctor;
2  int resultado = meuFunctor(5); // Chama o operador() do functor
3
```



UNICAMP

Functor como parâmetro

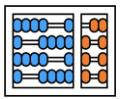
```
1  std::vector<int> numeros = {1, 2, 3, 4, 5};
2
3  MeuFunctor meuFunctor;
4  std::for_each(numeros.begin(), numeros.end(), meuFunctor);
5
```

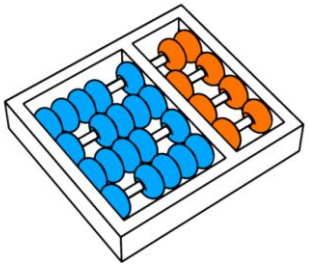


UNICAMP

Estudar!

- Relembrar os conceitos de estruturas de dados e complexidade
- Comparar os containers, algoritmos e iteradores
- Procurar os conceitos passados no site https://cplusplus.com/ e identificar diferenças entre as versões do C++





INSTITUTO DE COMPUTAÇÃO

UNICAMP

Prof. Dr. Bruno B. P. Cafeo

Sala 04
Instituto de Computação - Unicamp
Av. Albert Einstein, 1251
Cidade Universitária
Campinas – SP
13083-852

https://ic.unicamp.br/~cafeo/
cafeo@ic.unicamp.br