Programação Orientada a Objetos

Biblioteca STL (Standard Template Library) de C++

1. Introdução à STL

A **STL (Standard Template Library)** é uma das bibliotecas fundamentais do C++ que fornece **contêineres genéricos, iteradores e algoritmos reutilizáveis**. Ela foi projetada para ser eficiente, modular e extensível, promovendo a reutilização de código de forma segura e eficiente.

---

2. Motivação e Racional por Trás da STL

Antes da STL, os desenvolvedores precisavam implementar manualmente estruturas de dados e algoritmos, como listas, pilhas e buscas binárias. A STL foi criada para:

1. **Aumentar a produtividade**: Fornecendo implementações prontas e testadas.
2. **Promover código reutilizável**: Usando templates, a mesma estrutura funciona para diferentes tipos.
3. **Otimizar a eficiência**: Implementações eficientes em termos de tempo e memória.
4. **Uniformizar o acesso a dados**: Usando **iteradores** que se comportam de maneira uniforme em diferentes contêineres.

---

3. Componentes da STL

A STL é composta por três partes principais:

1. **Contêineres**: Estruturas de dados genéricas para armazenar objetos.
2. **Iteradores**: Abstrações que permitem percorrer os elementos dos contêineres.
3. **Algoritmos**: Funções genéricas que operam sobre contêineres, como `sort` e `find`.

3.1 Contêineres

Os contêineres da STL são **templates de classe** que armazenam objetos. Eles podem ser divididos em:

- **Sequenciais**: `vector`, `list`, `deque`
- **Associativos**: `set`, `map`
- **Associativos não ordenados**: `unordered_set`, `unordered_map`

3.2 Iteradores

Os **iteradores** são como **ponteiros** que percorrem elementos dos contêineres. Eles fornecem uma interface comum para diferentes tipos de contêineres, permitindo que algoritmos funcionem de forma genérica.

3.3 Algoritmos

Os algoritmos da STL são funções genéricas que realizam operações comuns, como:

- **Ordenação**: `sort`, `stable_sort`
- **Busca**: `find`, `binary_search`
- **Manipulação**: `transform`, `reverse`

---

4. Principais Contêineres da STL

4.1 `vector`

O `vector` é um contêiner dinâmico que armazena elementos contíguos na memória, como arrays, mas com a vantagem de redimensionamento automático.

Exemplo:

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

int main() {
    vector<int> numbers = {1, 2, 3, 4, 5};
    numbers.push_back(6);  // Adiciona um novo elemento

    for (int n : numbers) {
        cout << n << " ";
    }
    return 0;
}
```

Vantagens: Acesso rápido por índice.
Desvantagens: Custo alto ao inserir elementos no início.

4.2 `list`

A `list` é uma lista duplamente encadeada, ideal para inserções e remoções frequentes em qualquer posição.

Exemplo:

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

int main() {
    list<int> numbers = {1, 2, 3};
    numbers.push_front(0);  // Adiciona no início

    for (int n : numbers) {
        cout << n << " ";
```

```
    }
    return 0;
}
```

4.3 `deque`

O `deque` é um contêiner sequencial que permite inserções e remoções eficientes em ambos os extremos.

4.4 `set` e `unordered_set`

O `set` armazena elementos únicos e ordenados. Já o `unordered_set` usa uma tabela hash para armazenar elementos sem ordem definida.

Exemplo:

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

int main() {
    set<int> unique_numbers = {3, 1, 4};
    unique_numbers.insert(2);

    for (int n : unique_numbers) {
        cout << n << " ";
    }
    return 0;
}
```

4.5 `map` e `unordered_map`

O `map` armazena pares chave-valor em ordem crescente. O `unordered_map` é mais rápido para busca devido ao uso de hash.

Exemplo:

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

int main() {
    map<string, int> age = {{"Alice", 25}, {"Bob", 30}};
    age["Eve"] = 22;

    for (const auto& [name, value] : age) {
```

```
        cout << name << ": " << value << endl;
    }
    return 0;
}
```

---

5. Iteradores e Suas Aplicações

Os **iteradores** são uma abstração que permite acessar e manipular os elementos de um contêiner de forma uniforme, **independente da implementação interna** do contêiner. Eles funcionam como **ponteiros inteligentes**, fornecendo uma interface comum para percorrer e manipular coleções de dados.

Tipos de Iteradores

1. **Input Iterator**
    - **Leitura** sequencial dos elementos de um contêiner.
    - Exemplo: `std::istream_iterator` para ler de entrada padrão.
2. **Output Iterator**
    - **Escrita** de valores em contêineres.
    - Exemplo: `std::ostream_iterator` para imprimir valores na saída padrão.
3. **Forward Iterator**
    - **Avança** apenas na direção direta, usado em contêineres simples como `forward_list`.
4. **Bidirectional Iterator**
    - Pode **avançar e retroceder** nos contêineres.
    - Usado em contêineres como `list` e `set`.
5. **Random Access Iterator**
    - Permite **acesso direto a qualquer elemento** com tempo constante.
    - Usado em `vector` e `deque`.

Exemplo: Uso de Iteradores

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

int main() {
    vector<int> numbers = {1, 2, 3, 4, 5};
    vector<int>::iterator it;

    for (it = numbers.begin(); it != numbers.end(); ++it) {
        cout << *it << " ";
```

```
    }
    return 0;
}
```

Explicação:

- `begin()` e `end()` retornam iteradores para o início e fim do contêiner.
- `*it` desreferencia o iterador para acessar o valor.

Iteradores Constantes e Reverse Iterators

1. **Const Iterators**: Garantem que os elementos não serão modificados.

```
vector<int>::const_iterator it = numbers.begin();
```

2. **Reverse Iterators**: Permitem percorrer o contêiner de trás para frente.

```
for (auto it = numbers.rbegin(); it != numbers.rend(); ++it) {
    cout << *it << " ";
}
```

---

6. Algoritmos na STL (Standard Template Library)

Os algoritmos da STL facilitam operações sobre contêineres e conjuntos de dados. Eles operam de forma eficiente e **genérica**, pois trabalham com **iteradores**, que permitem a aplicação dos algoritmos a qualquer tipo de contêiner.

Nesta subseção, vamos explorar alguns dos algoritmos mais usados:

- **Ordenação:** `sort`, `stable_sort`
- **Busca:** `find`, `binary_search`
- **Manipulação:** `transform`, `reverse`

---

1. Ordenação: `sort`, `stable_sort`

1.1 `std::sort`

O algoritmo `std::sort` é usado para **ordenar elementos em ordem crescente** por padrão. Ele é eficiente e tem complexidade **O(n log n)**. O `sort` pode aceitar uma **função de comparação** para definir uma ordem personalizada.

Exemplo Simples de `std::sort`

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

int main() {
    vector<int> numbers = {5, 1, 4, 2, 3};
    sort(numbers.begin(), numbers.end());  // Ordenação crescente

    for (int n : numbers) {
        cout << n << " ";
    }
    return 0;
}
```

Saída:

```
1 2 3 4 5
```

Ordenação Personalizada com Funções Lambda

```cpp
sort(numbers.begin(), numbers.end(), [](int a, int b) { return a > b; });  // Decrescente
```

Aqui, a função lambda `[](int a, int b) { return a > b; }` define a ordem decrescente.

1.2 `std::stable_sort`

O `std::stable_sort` é similar ao `sort`, mas **preserva a ordem relativa dos elementos iguais**. Ele é útil em situações onde essa preservação é importante.

Exemplo de `stable_sort`

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

struct Person {
    string name;
    int age;
};
```

```cpp
int main() {
    vector<Person> people = {{"Alice", 25}, {"Bob", 30}, {"Charlie", 25}};
    stable_sort(people.begin(), people.end(), [](const Person& a, const Person& b)
{
        return a.age < b.age;
    });

    for (const auto& p : people) {
        cout << p.name << " (" << p.age << ")" << endl;
    }
    return 0;
}
```

Saída:

```
Alice (25)
Charlie (25)
Bob (30)
```

Aqui, `Alice` e `Charlie`, ambos com 25 anos, permanecem na mesma ordem relativa após a ordenação.

---

2. Busca: `find`, `binary_search`

2.1 `std::find`

O `std::find` é um **algoritmo linear** que percorre o contêiner para **encontrar o primeiro elemento que corresponda ao valor** fornecido. Ele retorna um **iterador** para o elemento encontrado ou para o fim do contêiner se o elemento não existir.

Exemplo de `std::find`

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

int main() {
    vector<int> numbers = {1, 2, 3, 4, 5};
    auto it = find(numbers.begin(), numbers.end(), 3);

    if (it != numbers.end()) {
        cout << "Elemento encontrado: " << *it << endl;
    } else {
        cout << "Elemento não encontrado" << endl;
    }
```

```
    return 0;
}
```

Saída:

```
Elemento encontrado: 3
```

2.2 `std::binary_search`

O `binary_search` verifica se um elemento está presente em um contêiner **ordenado**. Ele é mais rápido que `find`, com **complexidade O(log n)**.

Exemplo de `binary_search`

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

int main() {
    vector<int> numbers = {1, 2, 3, 4, 5};
    bool found = binary_search(numbers.begin(), numbers.end(), 3);

    if (found) {
        cout << "Elemento encontrado!" << endl;
    } else {
        cout << "Elemento não encontrado" << endl;
    }
    return 0;
}
```

Saída:

```
Elemento encontrado!
```

Nota: O contêiner **precisa estar ordenado** para que `binary_search` funcione corretamente.

---

3. Manipulação: `transform`, `reverse`

3.1 `std::transform`

O `std::transform` aplica uma **função a cada elemento** de um contêiner, podendo armazenar o resultado em um outro contêiner ou sobrescrever o existente.

Exemplo de `std::transform`

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

int main() {
    vector<int> numbers = {1, 2, 3, 4, 5};
    vector<int> squared(numbers.size());

    transform(numbers.begin(), numbers.end(), squared.begin(), [](int n) { return
n * n; });

    for (int n : squared) {
        cout << n << " ";
    }
    return 0;
}
```

Saída:

```
1 4 9 16 25
```

Aqui, a função lambda `[](int n) { return n * n; }` calcula o quadrado de cada número.

3.2 `std::reverse`

O `std::reverse` inverte a **ordem dos elementos** em um contêiner.

Exemplo de `std::reverse`

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

int main() {
    vector<int> numbers = {1, 2, 3, 4, 5};
    reverse(numbers.begin(), numbers.end());
```

```
    for (int n : numbers) {
        cout << n << " ";
    }
    return 0;
}
```

Saída:

```
5 4 3 2 1
```

Resumo dos Algoritmos Abordados

Algoritmo	Função	Complexidade
`sort`	Ordena os elementos (não estável)	O(n log n)
`stable_sort`	Ordena mantendo a ordem relativa	O(n log n)
`find`	Busca linear	O(n)
`binary_search`	Busca binária (contêiner ordenado)	O(log n)
`transform`	Aplica uma função a cada elemento	O(n)
`reverse`	Inverte a ordem dos elementos	O(n)

---

7. As Novidades da STL nas Versões Recentes do C++

C++17

1. `std::optional`: Permite representar valores opcionais.

```
std::optional<int> maybe_value = 42;
if (maybe_value) {
    cout << "Valor presente: " << maybe_value.value() << endl;
}
```

2. `std::any`: Pode armazenar qualquer tipo de valor.

```
std::any value = 10;
cout << std::any_cast<int>(value) << endl;
```

3. `std::string_view`: Uma visualização leve de strings.

```
std::string_view sv = "Hello, World!";
cout << sv << endl;
```

C++20

1. **Ranges**: Facilita a aplicação de algoritmos a contêineres.

```
#include <ranges>
auto v = std::views::iota(1, 10);
for (int n : v) {
    cout << n << " ";
}
```

2. `std::span`: Uma visualização de array não-possuidora.

```
int arr[] = {1, 2, 3};
std::span<int> sp(arr);
```

3. **Conceitos**: Permitem restringir templates.

```
template <std::integral T>
T add(T a, T b) {
    return a + b;
}
```

C++23

1. **Novas funções com ranges**: Como `std::ranges::to` para conversão direta.
2. **Performance aprimorada**: Contêineres mais rápidos para inserção e busca.

---

8. Casos de Uso Práticos

1. **Filtragem e Transformação de Dados**

```
vector<int> numbers = {1, 2, 3, 4, 5};
auto even_numbers = numbers | std::views::filter([](int n) { return n % 2 ==
0; });
```

2. **Ordenação Customizada com Lambda**

```
sort(numbers.begin(), numbers.end(), [](int a, int b) { return a > b; });
```

3. **Uso de `std::unordered_map` para Cache**

```
unordered_map<int, string> cache = {{1, "um"}, {2, "dois"}};
cout << cache[1] << endl;
```
