

C/C++

Estruturas de Dados STL

Prof. Edson Alves - UnB/FGA

2019

std::vector

Declaração do Vector

```
std::vector<tipo_dado> nome;
```

Um vector traz a ideia de um array dinâmico, um armazenamento contínuo de elementos na memória que se realoca dinamicamente quando necessário.

Propriedades do Container:

- Sequencial: elementos ordenados de maneira linear. Acesso individual baseado em suas posições na sequência.
- Vetor dinamico: permite acesso direto ao elemento por aritmética de ponteiros e provê adição/remoção eficiente de elementos no fim da sequência.
- Alocação dinamica: o container utiliza um objeto alocador para lidar dinamicamente com as mudanças de necessidade de alocamento.

- Inserção: possui complexidade constante ou linear, vai depender da necessidade de realocar o vetor ou não.
- Acesso: o acesso é constante pois é um simples acesso direto à uma localização da memória.
- Deleção: caso tente apagar um número ao final do vetor o tempo será constante, porém para apagar qualquer elemento que não seja o último o vetor terá de ser realocado.
- Pesquisa: achar um valor no vetor tem complexidade de tempo linear com o tamanho, pois precisa percorrer todo ele no pior caso, porém se estiver ordenado a pesquisa será logarítmica utilizando busca binária.

- Inserção:

```
vector<int> a { 1, 2 };  
a.push_back(1);  
// a = { 1, 2, 1 }
```

- Acesso:

```
vector<int> a { 0, 1 };  
a[0]; // retorna 0  
a.at(1); // retorna 1  
a.front() // retorna 0  
a.back() // retorna 1
```

- Deleção:

```
vector<int> a { 0, 1, 2 };  
a.erase(a.begin() + 1); // a se torna { 0, 2 }  
a.pop_back(); // a se torna { 0 }  
a.clear(); // limpa a, a se torna { }
```

- Pesquisa:

```
vector<int> a { 1, 2, 3, 4 };  
auto it = find(a.begin(), a.end(), 2);  
// it será um iterador para o elemento  
auto bit = binary_search(a.begin(), a.end(), 2);  
// bit é um iterador para o elemento encontrado com complexidade  
//  $O(\log n)$  pois é resultado de uma busca binária
```


- Informações:

```
vector<int> a { 1, 2, 3 };  
a.size(); // retorna 3  
a.empty(); // retorna false
```

Vector

```
1 #include <vector>
2
3 using namespace std;
4
5 int main () {
6     vector<int> v0;           // empty vector
7     vector<int> v1(5);        // v1 = 0 0 0 0 0
8     vector<int> v2(5, 1);     // v2 = 1 1 1 1 1
9     vector<int> v3(v2);       // v3 = v2
10    vector<int> v4 {1, 3};     // v4 = 1 3
11
12    return 0;
13 }
```

Vector

```
1 #include <vector>
2 #include <iostream>
3 #include <algorithm>
4
5 using namespace std;
6
7 int main () {
8     vector<int> v {0, 1, 2};
9
10    v.push_back(3);
11    // v : 0 1 2 3
12
13    cout << v[0] << endl;
14    // prints: 0
15
16    cout << v.back() << endl;
17    // prints: 3
18
19    v.pop_back();
20    cout << v.back() << endl;
21    // prints: 2
```

Vector

```
23 // v : 0 1 2
24 auto it = v.begin() + 1;
25 v.erase(it);
26 // v : 0 2
27
28 auto found = find(v.begin(), v.end(), 2);
29 if(found != v.end())
30     cout << *found << endl;
31 // prints: 2
32
33 for(auto e : v)
34     cout << e << " ";
35 cout << endl;
36 // prints: 0 2
37
38 return 0;
39 }
```

std::queue

Declaração da Queue

```
std::queue<tipo_dado> nome;
```

No STL, a queue é um container associativo. Ela traz uma estrutura FIFO (first in, first out). É possível inserir elementos no fim da fila e retirar apenas da frente.

Propriedades do container adaptado ('std::deque'):

- Sequencial: elementos ordenados de maneira linear e acessados individualmente de acordo com suas posição na sequência.
- Vetor dinamico: permite acesso direto ao elemento na sequência com adição e remoção eficientes no início ou final.
- Alocação dinamica: utiliza de um objeto de alocação dinamica para lidar com diferentes necessidades de espaço.

- Inserção: a inserção será sempre constante em relação à sua complexidade de tempo.
- Acesso: o acesso ao primeiro e ao último elemento é constante.
- Deleção: a interface permite apenas deletar o elemento da frente da fila, complexidade de tempo constante.

- Inserção:

```
queue<int> q;  
q.push(1); // q = { 1 }
```

- Acesso:

```
queue<int> q { 1, 2, 3 };  
q.front(); // 1  
q.back(); // 3
```

- Deleção:

```
queue<int> q { 1, 2, 3 };  
q.pop(); // q = { 2, 3 }
```

- Informações:

```
queue<int> q { 1, 2, 3 };  
q.size(); // retorna 3  
q.empty(); // retorna false
```

Queue

```
1 #include <queue>
2 #include <iostream>
3
4 using namespace std;
5
6 int main () {
7     queue<int> q;
8     int f;
9
10    q.push(1);
11    q.push(10);
12
13    while(!q.empty()) {
14        f = q.front();
15        foo(f);
16        q.pop();
17    }
18
19    return 0;
20 }
```

std::stack

Declaração da Stack

```
std::stack<tipo_dado> nome;
```

No STL, o stack é um container associativo. Ele traz uma estrutura LIFO (last in, first out). É possível inserir elementos no topo da pilha e retirar apenas do topo.

Propriedades do container adaptado ('std::deque'):

- Sequencial: elementos ordenados de maneira linear e acessados individualmente de acordo com suas posição na sequência.
- Vetor dinamico: permite acesso direto ao elemento na sequência com adição e remoção eficientes no início ou final.
- Alocação dinamica: utiliza de um objeto de alocação dinamica para lidar com diferentes necessidades de espaço.

- Inserção: a inserção será sempre constante em relação à sua complexidade de tempo.
- Acesso: o acesso ao primeiro e ao último elemento é constante.
- Deleção: a interface permite apenas deletar o elemento do topo da fila, complexidade de tempo constante.

- Inserção:

```
stack<int> s;  
s.push(1); // s = { 1 }
```

- Acesso:

```
stack<int> s { 0, 1, 2 };  
s.top(); // 2
```


- Deleção:

```
stack<int> s { 1, 2 };  
s.pop(); // s = { 1 }
```

- Informações:

```
stack<int> s { 1, 2, 3 };  
s.size(); // retorna 3  
s.empty(); // retorna false
```

Stack

```
1 #include <stack>
2 #include <iostream>
3
4 using namespace std;
5
6 int main () {
7     stack<int> s;
8     int t;
9
10    s.push(1);
11    s.push(10);
12
13    while(!s.empty()) {
14        t = s.front();
15        foo(t);
16        t.pop();
17    }
18
19    return 0;
20 }
```

std::set

Declaração do set

```
std::set<tipo_dado> nome;
```

O set é uma estrutura de dado não linear, normalmente implementada com uma árvore binária vermelha-preta. Como ela é mantida balanceada o set possui boa performance para pesquisas.

Propriedades do container:

- Ordenado: elementos no container vão seguir sempre um ordem. Todos elementos quando inseridos são colocados na respectiva posição respeitando essa ordem.
- Set: o elemento também é a própria chave usada para sua própria identificação.
- Chaves únicas: dois elementos não poderão possuir chaves equivalentes.
- Alocação dinâmica: utiliza de um objeto de alocação dinâmica para lidar com as necessidades de espaço em memória.

- Inserção: a inserção vai ter complexidade logarítmica em relação ao tamanho, é necessário achar o melhor lugar para inserir e manter a árvore balanceada.
- Deleção: também possui complexidade logarítmica em relação ao tamanho por causa da necessidade de manter a árvore balanceada.
- Pesquisa: achar um valor no set possui uma performance logarítmica, o que é um dos melhores motivos para se utilizar um set.

- Inserção:

```
set<int> s;  
s.insert(1); // s = { 1 }
```

- Deleção:

```
set<int> s;  
s.insert(1); // s = { 1 }  
s.erase(1); // retorna 1 (num de elementos apagados) e apaga  
s.erase(2); // retorna 0
```

- Pesquisa:

```
set<int> s { 1, 2, 3 };  
s.find(2); // retorna um iterador para o elemento
```

- Informações:

```
set<int> s { 1, 2, 3 };  
s.size(); // retorna 3  
s.empty(); // retorna false  
s.count(1); // retorna se um item está no set
```



```
1 #include <set>
2 #include <iostream>
3
4 using namespace std;
5
6 int main () {
7     set<int> s { 1, 2, 3, 10, 15 };
8
9     s.insert(1); // s = { 1, 2, 3, 10, 15 }
10
11     auto it = s.find(2); // it é o iterador para o elemento 2
12
13     for(auto i = it; i != s.end(); i++)
14         cout << *i << " ";
15     cout << endl;
16     // 3 10 15
17
18     return 0;
19 }
```

std::map

Declaração do map

```
std::map<tipo_chave, tipo_valor> nome;
```

O map é uma estrutura de dado não linear, ela possui pares de chaves e valores. Para cada chave existe um valor.

Propriedades do container:

- Ordenado: a estrutura do map mantém ordenado.
- Chaves únicas: dois elementos (par chave-valor) não possuirão chaves equivalentes.
- Alocação dinâmica: a estrutura vai alterar sua alocação de acordo com a necessidade.

- Inserção:
- Deleção:
- Pesquisa:

- Inserção:

```
map<string, int> m;  
m["hello"] = 1;
```

- Deleção:

```
map<int, double> m;  
m[1] = 2.01;  
m.erase(1);
```

- Pesquisa:

```
map<int, string> m;  
m[100] = "bla";  
m.find(100); // retorna iterador para o elemento
```

Map

```
1 #include <map>
2 #include <iostream>
3
4 using namespace std;
5
6 int main () {
7     map<char, int> m;
8     string s = "Hello";
9
10    for(auto c : s)
11        m[c]++;
12
13    for(auto const& [key, value] : m) {
14        cout << "Letra "
15             << key
16             << ": "
17             << value << endl;
18    }
```



```
20 // output:
21 // Letra H: 1
22 // Letra e: 1
23 // letra l: 2
24 // letra o: 1
25
26 return 0;
27 }
```