PCC104 - Projeto e Análise de Algoritmos

Marco Antonio M. Carvalho

Departamento de Computação
Instituto de Ciências Exatas e Biológicas
Universidade Federal de Ouro Preto

Conteúdo

Fonte

Este material é baseado nos livros

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
- S. Halim. *Competitive Programming*. 3rd Edition, 2013.
- Ian Parberry and William Gasarch. *Problems on Algorithms*. Second Edition, 2002.
- Ian Parberry *Lecture Notes on Algorithm Analysis and Complexity Theory*. Fourth Edition, 2001.

Licença

Introdução

Os **genéricos** (ou templates) são uma das mais poderosas maneiras de reuso de software.

Funções genéricas e classes genéricas permitem que o programador especifique com apenas um segmento de código uma família de funções ou classes relacionadas (sobrecarregadas).

Esta técnica é chamada **programação genérica**.

Introdução

Por exemplo, podemos criar uma função genérica que ordene um vetor e a linguagem se encarrega de criar especializações que tratarão vetores do tipo int, float, string e etc.

Podemos também criar uma classe genérica para a estrutura de dados Pilha, e a linguagem se encarrega de criar as especializações pilha de int, float, string, etc..

O genérico é um estêncil (define o formato), a especialização é conteúdo.

Funções Genéricas

Funções sobrecarregadas normalmente realizam operações similares ou idênticas em diferentes tipos de dados: soma de int, float, e frações.

Se as operações são idênticas para diferentes tipos, elas podem ser expressas mais compacta e convenientemente através de **funções genéricas**.

O programador escreve a definição da função genérica e, baseado nos parâmetros explicitamente enviados ou inferidos a partir da chamada da função, o compilador gera as especializações para cada tipo de chamada.

Classes Genéricas

Para compreendermos o funcionamento da estrutura de dados Pilha, não importa o tipo dos dados empilhados/desempilhados.

No entanto, para implementarmos uma pilha, é necessário associá-la a um tipo.

O ideal é descrever uma pilha genericamente, assim como a entendemos e instanciar versões específicas desta classe genérica fica por conta do compilador.

Desta forma, uma classe genérica Pilha vira uma coleção de classes especializadas: pilha de int, float, string, frações, restaurantes, etc.

Para instanciarmos um objeto de uma classe genérica, precisamos informar qual tipo deve ser associado à classe.

Exemplo

```
#include <iostream>
using namespace std;
#include "Stack.h"
int main(){
    Stack< double > doubleStack( 5 );
    double doubleValue = 1.1;
    while ( doubleStack.push( doubleValue ) ) {
        cout << doubleValue << ' ';
        doubleValue += 1.1;
    }
   Stack< int > intStack;
    int intValue = 1;
    while ( intStack.push( intValue ) ) {
        cout << intValue << ' ';
        intValue++;
    }
    while ( intStack.pop( intValue ) )
        cout << intValue << ' ';
    return 0;
}
```

Introdução

Considerando a utilidade do reuso de software e também a utilidade das estruturas de dados e algoritmos utilizados por programadores a *Standard Template Library* (STL) foi adicionada à biblioteca padrão C++.

A STL define componentes genéricos reutilizáveis poderosos que implementam várias estruturas de dados e algoritmos que processam estas estruturas.

Basicamente, a STL é composta por contêineres, iteradores e algoritmos.

Introdução

Contêineres são genéricos de estruturas de dados e possuem métodos associados a eles. Tornam as estruturas independentes dos tipos de dados.

Iteradores são semelhantes a ponteiros, utilizados para percorrer e manipular os elementos de um contêiner. Tornam os algoritmos independentes dos contêineres.

Algoritmos são os métodos que realizam operações tais como buscar, ordenar e comparar elementos ou contêineres inteiros, utilizando iteradores.

Existem aproximadamente 85 algoritmos padrão implementados na STL, mais operadores e operações de contextos específicos.

Contêineres

Os contêineres são divididos em três categorias principais:

Contêineres Sequenciais: Estruturas de dados lineares;

Contêineres Associativos: Estruturas de dados não lineares, pares chave/valor.

Adaptadores de Contêineres: São contêineres sequenciais, porém, em versões restringidas.

Contêineres Sequenciais

vector	Inserções e remoções no final, acesso direto a qualquer elemento. Elementos contíguos.
deque	Fila de frente dupla, inserções e remoções no início ou no final, acesso direto a qualquer elemento. Elementos não contíguos.
list	Lista de frente dupla, inserção e remoção em qualquer ponto.

Contêineres Associativos

set	Busca rápida, não permite elementos duplicados.
multiset	Busca rápida, permite elementos duplicados.
map	Mapeamento um-para-um, não permite elementos duplicados, busca rápida.
multimap	Mapeamento um-para-um, permite elementos duplicados, busca rápida.

Adaptadores de Contêineres

stack	Last-in, first out (LIFO);
queue	First –in, first out (FIFO);
priority_queue	O elemento de maior prioridade é sempre o primeiro elemento a sair.

Contêineres

Todos os contêineres da STL fornecem funcionalidades similares, e muitas operações genéricas se aplicam a todos os contêineres.

Outras operações se aplicam somente a subconjuntos de contêineres similares.

Funções Comuns a Todos Contêineres

Construtor *default* Fornece a inicialização padrão do contêiner.

Construtor Cópia Construtor que inicializa um contêiner para ser a cópia de outro do mesmo tipo;

Destrutor Simplesmente destrói o contêiner quando não for mais necessário.

empty Retorna *true* se não houver elementos no contêiner e *false* caso contrário.

size Retorna o número de elementos no contêiner.

operator= Atribui um contêiner a outro.

operator< Retorna *true* se o primeiro contêiner for menor que o segundo e *false* caso contrário.

Operadores

operator<=	Retorna *true* se o primeiro contêiner for menor ou igual ao segundo e *false* caso contrário;
operator>	Retorna *true* se o primeiro contêiner for maior que o segundo e *false* caso contrário;
operator>=	Retorna *true* se o primeiro contêiner for maior ou igual ao segundo e *false* caso contrário;
operator==	Retorna *true* se o primeiro contêiner for igual ao segundo e *false* caso contrário;
operator!=	Retorna *true* se o primeiro contêiner for diferente do segundo e *false* caso contrário;
swap	Troca os elementos de dois contêineres

Atenção! Os operadores <, <=, >, >=, == e != não são fornecidos para o contêiner **priority_queue**.

Funções Comuns a Contêineres Sequenciais e Associativos

max_size	Retorna o número máximo de elementos de um contêiner;
begin	Retorna um iterator para o primeiro elemento do contêiner;
end	Retorna um iterator para a posição após o final do contêiner;
rbegin	Retorna um reverse_iterator para o primeiro elemento do contêiner invertido;
rend	Retorna um reverse_iterator para a posição após o final do contêiner invertido;
erase	Apaga um ou mais elementos do contêiner;
clear	Apaga todos os elementos do contêiner.

Bibliotecas de Contêineres

<vector>	Vetor;
<list>	Lista;
<deque>	Fila de frente dupla;
<queue>	Contém queue e priority_queue;
<stack>	Pilha;
<map>	Contém map e multimap;
<set>	Contém set e multiset;
<bitset>	Conjunto de bits (vetor em que cada elemento é um bit – 0 ou 1).

Iteradores

Iteradores são utilizados para apontar elementos de contêineres sequenciais e associativos.

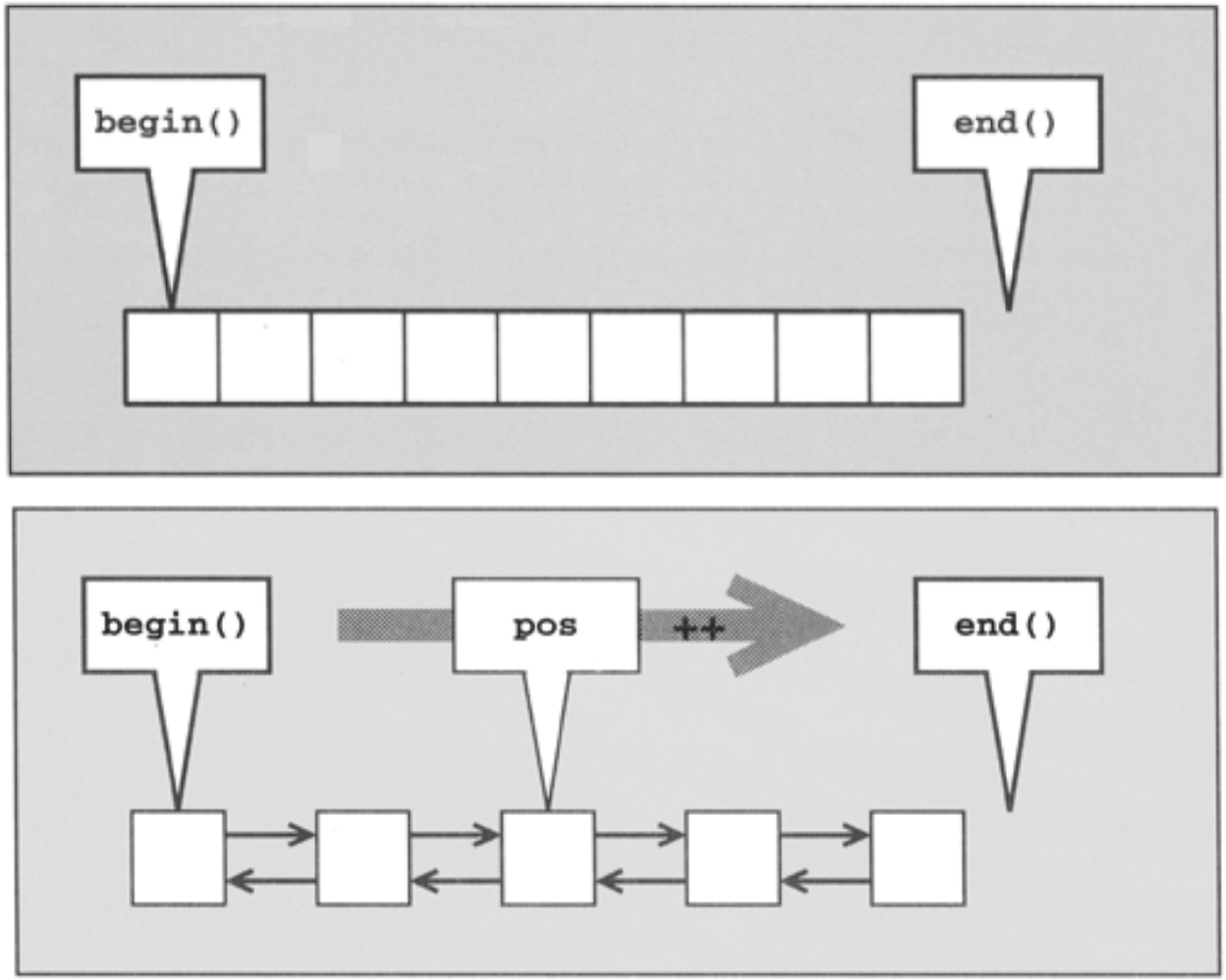
São objetos declarados na biblioteca <**iterator**>.

Algumas operações e algoritmos retornam iteradores como resultado.

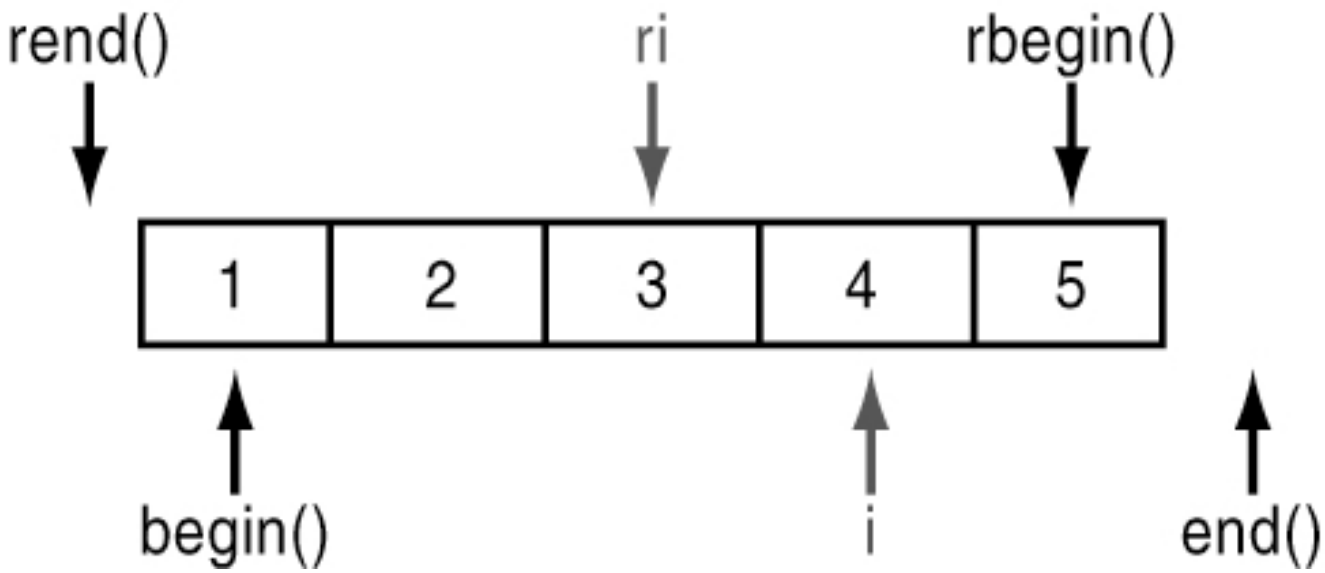
Se um iterador i aponta para um elemento:

- $++i$ aponta para o próximo elemento (há diferença para o reverse_iterator);
- $*i$ se refere ao conteúdo do elemento apontado por i.

Iteradores



Iteradores



Operações em Iteradores

Atenção! As operações variam de acordo com o tipo de iterador declarado.

++p	Incremento prefixado;
p++	Incremento pós-fixado;
*p	Referencia o conteúdo apontado;
p = p1	Atribui um iterador a outro;
p == p1	Compara dois iteradores quanto a igualdade;
p != p1	Compara dois iteradores quanto a desigualdade;
--p	Decremento prefixado;
p--	Decremento pós-fixado;
p+= i	Incrementa o iterador em i posições;
p-= i	Decrementa o iterador em i posições;
p + i	Resulta em um iterador posicionado em p+i elementos;

Operações em Iteradores

Atenção! As operações variam de acordo com o tipo de iterador declarado.

- p – i: Resulta em um iterador posicionado em p-i elementos;
- p[i]: Retorna uma referência para o elemento a i posições a partir de p;
- p < p1: Retorna *true* se o primeiro iterador estiver antes do segundo no contêiner;
- p < =p1: Retorna *true* se o primeiro iterador estiver antes ou na mesma posição do segundo no contêiner;
- p > p1: Retorna *true* se o primeiro iterador estiver após o segundo no contêiner;
- p >= p1: Retorna *true* se o primeiro iterador estiver após ou na mesma posição do segundo no contêiner.

Algoritmos

A STL inclui aproximadamente 85 algoritmos padrão:

- Podem ser utilizados genericamente, em vários tipos de contêineres;
- Operam indiretamente sobre os elementos de um contêiner usando iteradores;
- Frequentemente, também retornam iteradores como resultado.

Este desacoplamento dos contêineres permite que os algoritmos sejam genéricos, assim como as estruturas de dados.

vector

A classe vector implementa a estrutura de dados sequencial e contígua **arranjo**, possuindo acesso indexado aos elementos.

Este contêiner é dinâmico, ou seja, a cada inserção o contêiner se redimensiona automaticamente.

Adicionalmente, arranjos estáticos podem ser encontrados na classe *array*.

As classes que implementam contêineres são genéricas, logo, deve ser definido o tipo na declaração de um objeto.

vector

Alguns métodos do contêiner vector incluem:

- **push_back:** : adiciona um elemento ao final do vector.
- **front:** determina o primeiro elemento;
- **back:** determina o último elemento;
- **at:** determina o elemento em uma determinada posição, mas antes verifica se é uma posição válida;
- **insert:** insere um elemento em uma posição especificada por um iterador.

Exemplo

```
#include <iostream>
#include <vector>
using namespace std;
int main (){

  int i;

  vector<int> first;
  vector<int> second (4,100);

  second.push_back(1);
  vector<int>::iterator it = second.begin();
  second.insert(it, 200);

  cout << "Primeiro elemento:" <<second.front()<<endl;
  cout << "Ultimo elemento:" <<second.back()<<endl;
  cout << "Elemento 3:" <<second.at(2)<<endl;
```

Exemplo

```
    cout << "O tamanho:" <<second.size()<<endl;

    first.push_back(3);
    first.pop_back();

    for (it = first.begin(); it != first.end(); ++it)
        cout << *it << ' ';

    first.clear();
    it = first.begin();
    first.erase(it);

    return 0;
}
```

list

A classe list implementa a estrutura de dados sequencial **lista duplamente encadeada**, que pode ser percorrida em ambas as direções.

Adicionalmente, listas simplesmente encadeadas podem ser encontradas na classe *forward_list*.

Quando comparados a arrays, vectors e deques, lists possuem melhor performance em operações de inserção, remoção e movimentação de elementos.

list

No exemplo a seguir, temos os seguintes métodos da classe list:

- sort: ordena a lista em ordem crescente;
- unique: remove elementos duplicados;
- remove: apaga todas as ocorrências de um determinado valor da lista.
- push_front: insere um elemento no início da lista.
- push_back: insere um elemento no final da lista.
- pop_front: remove o elemento no início da lista.
- pop_back: remove elemento no final da lista.

Exemplo

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

int main (){
  list<int> first;
  list<int> second (4,100);

  first.push_front(1);
  first.push_front(2);
```

Exemplo

```
    first.push_back(4);
    first.push_back(1);

    first.remove(4);

    first.unique();

    first.pop_front();
    first.pop_back();

    first.sort();
    list<int>::iterator it;
    for (it = first.begin(); it != first.end(); it++)
        cout << *it << " ";

    return 0;
}
```

deque

Um **deque** (*double-ended queue*) é um contêiner dinâmico a partir do qual podem ser removidos e inseridos itens em ambas as extremidades.

Para inserção no início, o deque também possui o método push_front.

Na STL, deques não são implementados como **arranjos** dinâmicos. Porém, o operador [] permite acesso direto aos elementos do deque, como em vectors.

Em geral, um deque possui um desempenho levemente inferior em relação a um vector, porém, é mais eficiente para fazer inserções e remoções no início.

Exemplo

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

int main (){
  int i;

  deque<int> first;
  deque<int> second (4,100);
```

Exemplo

```
  first.push_front(2);
  first.push_front(3);

  first.push_back(1);

  first[1] = 5;

  first.pop_front();
  first.pop_back();

  for (i=0; i < first.size(); i++)
    cout << "␣" << first[i];

  return 0;
}
```

set e multiset

Estruturas de conjuntos e multiconjuntos são implementadas nas classes set e multiset, respectivamente.

Ambos são implementados como **árvores binárias de busca** ou **árvores vermelho-e-preto**.

Internamente, os elementos estão sempre ordenados de acordo com o **comparador** fornecido.

Sets e multisets não possuem acesso direto aos elementos, e estes não podem ter seus valores alterados.

Um multiset possui todas as características de um set, porém, permite elementos repetidos.

set e multiset

Adicionalmente, conjuntos e multiconjuntos desordenados (implementados por tabelas hash com encadeamento) podem ser encontrados respectivamente nas classes **unordered_set** e **unordered_multiset**.

Exemplo

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

int main (){

  set<int> first(10);
  set<int> second;

  for (int i = 0; i < 15; ++i)
    second.insert(i);

  first.insert(10);
```

Exemplo

```
  first.erase(40);

  set<int>::iterator it = first.find(40);

  first.swap(second);

  for (it=first.begin(); it!=first.end(); it++)
    cout << "␣" << *it;
  cout << endl;

  return 0;
}
```

Exemplo

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

int main (){
  multiset<int> first;
  multiset<int> second;

  for (int i = 0; i < 15; ++i)
    second.insert(15);
```

Exemplo

```
    second.insert(10);

    cout<<second.count(15)<<endl;

    multiset<int>::iterator result = first.find(15);

    if(result == first.end())
            cout<<"nao encontrou";

    pair<multiset<int>::iterator, multiset<int>::iterator> ret;
    ret = second.equal_range(15);

    second.erase(ret.first,ret.second);

    return 0;
}
```

map e multimap

Na STL, mapas e multimapas são implementados como árvores binárias de busca, portanto, ordenados.

Adicionalmente, mapas e multimapas desordenados (implementados por tabelas hash com encadeamento) podem ser encontrados respectivamente nas classes **unordered_map** e **unordered_multimap**.

As chaves são únicas no map e podem se repetir no multimap.

Chaves e valores podem possuir tipos diferentes.

map e multimap

Como os mapas armazenam pares de elementos, os iteradores possuem uma característica extra:

- it->first ou (*it).first acessa a chave do elemento referenciado pelo iterador it;
- it->second ou (*it).second acessa o valor do elemento referenciado pelo iterador it.

map e multimap

As classes implementam o operador **[]**, entretanto, não permitem acesso direto aos elementos.

Acessar uma chave através deste operador insere esta chave no mapa, caso ela não esteja presente.

Assim, o operador não é adequado para verificar se uma chave está presente no mapa, é preciso utilizar o método **find**.

Exemplo

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

int main (){
  map<char,int> first;

  first['a']=10;
  first['b']=30;
  first['c']=50;
  first['d']=70;

  first.insert('e', 80);
```

Exemplo

```
  map<char,int>::iterator it = first.find('a');

  cout << it->first << '\t' << it->second << '\n';

  map<char, int>::iterator it;
  for (it = first.begin(); it != first.end(); ++it)
        cout << it->first << '\t' << it->second << '\n';

    return 0;
}
```

Exemplo

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

int main (){
  multimap<char,int> first;

  first.insert(pair<char,int>('a',10));
  first.insert(pair<char,int>('b',15));
  first.insert(pair<char,int>('b',20));
  first.insert(pair<char,int>('c',25));

  first['c']=26;

  multimap<char,int>::iterator it = first.find('a');

  first.erase(it);
```

Exemplo

```
  multimap<char, int>::iterator it;

  for (it = first.begin(); it != first.end(); ++it)
        cout << it->first << '\t' << it->second << '\n';

   pair <multimap<char,int>::iterator,
         multimap<char,int>::iterator> ret;
   ret = first.equal_range('b');

   for (it=ret.first; it!=ret.second; ++it)
      cout << ' ' << it->second;
   cout << '\n';

  return 0;
}
```

Adaptadores de Contêineres

Adaptadores de contêineres são classes que usam um contêiner encapsulado como estrutura subjacente:

- Uma pilha pode ser implementada sobre um vector, deque ou list.
- Uma fila pode ser implementada sobre um deque ou list.
- Uma fila de prioridades pode ser implementada sobre um vector ou um deque.

Além de definir o tipo dos elementos de um adaptador de contêiner na declaração do objeto, também é possível definir a estrutura de dados subjacente.

Adaptadores de Contêineres

Os adaptadores de contêineres (pilha, fila e fila de prioridades) contém praticamente os mesmos métodos:

- empty: testa se o contêiner está vazio;
- size: retorna a quantidade de elementos do contêiner;
- top (exceto fila): acessa o elemento do topo;
- push: insere um elemento;
- pop: remove um elemento;
- front (somente fila): acessa o próximo elemento;
- back (somente fila): acessa o último elemento.

stack

O adaptador de contêiner stack implementa a estrutura de dados pilha.

Por padrão, utiliza um deque como estrutura subjacente.

Os elementos são inseridos e removidos no final da estrutura.

Exemplo

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

int main(){
    stack< int > intDequeStack;
    stack< int, vector< int > > intVectorStack;
    stack< int, list< int > > intListStack;
```

Exemplo

```
    intDequeStack.push(1);
    intVectorStack.push(1);
    intListStack.push(1);

    cout << intDequeStack.size() << ' ' <<intDequeStack.top();

    intDequeStack.pop();
    intVectorStack.pop();
    intListStack.pop();

    return 0;
}
```

queue

O adaptador de contêiner queue implementa a estrutura de dados fila.

Por padrão, utiliza um deque como estrutura subjacente.

Os elementos são inseridos no final da estrutura e removidos no início.

Exemplo

```
#include <iostream>
#include <queue>
namespace std;

int main(){
    queue< int, list<int> > intListQueue;
    queue< double > values;

    values.push(3.2);
    values.push(9.8);
    values.push(5.4);

    while (!values.empty()){
        cout << values.front() << ' ';
        values.pop();
    }
    cout << endl;

    return 0;
}
```

priority_queue

O adaptador de contêiner priority_queue implementa a estrutura de dados fila de prioridades.

Por padrão, utiliza um vector como estrutura subjacente, utilizado para organizar um heap.

Os elementos são inseridos e removidos no final da estrutura.

Por padrão, tem-se um **MaxHeap**, ou seja, quanto maior o valor do elemento, maior sua prioridade.

Exemplo

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

int main(){
    priority_queue< double > priorities;
    priority_queue< double, deque<double> > dequePriorityQueue;

    priorities.push(3.2);
    dequePriorityQueue.push(9.8);

    while (!priorities.empty() && !dequePriorityQueue.empty()) {
        cout << priorities.top() << ' ';
        priorities.pop();

        cout << dequePriorityQueue.top() << ' ';
        dequePriorityQueue.pop();
    }
    cout << endl;

    return 0;
}
```

Muito além...

Ainda há na STL:

- Classe bitset (manipulação de conjuntos de bits);
- Pair;
- Objetos Função;
- Vários algoritmos.

Dúvidas?