decom
departamento
de computação

UFOP

Estruturas de Dados

Prof. Túlio A. M. Toffolo
Prof. Marco Antonio M. Carvalho
http://www.toffolo.com.br

BCC402 – Aula 02
Algoritmos e Programação Avançada

Como resolver o problema?

- Antes de mais nada, **dicas fundamentais**

- Leia o problema atentamente
- Não assuma nada: leia a especificação
- Eficiência nem sempre é algo tão importante a não ser que você esteja lidando com algoritmos exponenciais para problemas polinomiais

Como resolver o problema?

- Qual a melhor forma de representar uma mão de cartas?
  - Você vai embaralhar as cartas?
  - Comparar os seus valores?
  - Procurar por padrões na sua mão?

- Qual a melhor forma de representar cada carta?
  - Operações com cartas?
  - O naipe importa?

Testando e debugando

- Debugar pode ser muito **frustante** com o robô de avaliação (*robot judge*)
- Muito importante testar antes da submissão.

- **Teste a entrada**
- **Teste entradas incorretas**
- **Teste situações limites (entrada vazia, um único item, dois items e valores igual a zero)**

Testando e debugando

- **Teste instâncias em que você sabe a resposta correta**
- **Teste instâncias grandes (muito grandes)**

- Conheça bem seu debugador
- Exiba sua estruturas de dados
- Faça testes invariantes
- Verifique seu código
- Faça seus "prints" significarem alguma coisa

Testando e debugando

- Crie seus arranjos um pouco maiores do que o necessário
- Tenha certeza de que seus bugs são bugs de verdade

ESTRUTURAS DE DADOS

Estruturas de dados elementares

- Arranjos (arrays)
- Pilhas (stacks)
- Filas (queues)
- Listas (lists)
- Dicionários (dictionaries)
- Listas de Prioridade (priority queues)
- Conjuntos (sets)

Pilha (Stack)

- Tipo Abstrato de dados com a seguinte característica:
- O **último** elemento a ser inserido é o **primeiro** a ser retirado/ removido

(**LIFO – Last in First Out)**

- Analogia: pilha de pratos, livros, etc.
- Usos: Chamada de subprogramas, avaliação de expressões aritméticas, etc.

Pilha (Stack)

- Operações primárias:
  - PilhaPush(s, x) – Insere item **x** no **topo** da pilha **s**.
  - PilhaPop(s) – Retorna (e remove) o **topo** da pilha **s**.
  - PilhaInicializa(s) – Cria uma pilha vazia em **s**.
  - PilhaCheia(s), PilhaVazia(s) – Testa se a pilha **s** está cheia ou vazia, respectivamente.
- **Existem opções de estruturas de dados que podem ser usadas para representar pilhas.**
- As duas representações mais utilizadas são as implementações por meio de arranjos e de ponteiros

Fila (Queue)

- Tipo Abstrato de dados com a seguinte característica:
- O **primeiro** elemento a ser inserido é o **primeiro** a ser retirado/ removido

(**FIFO – First in First Out)**

- Analogia: fila em de pessoas, fila de espera, etc.
- Usos: Sistemas operacionais, filas de impressão, processamento, etc.

Fila (Queue)

- Operações primárias:
  - FilaPush(q, x) – Insere o item **x** no **final** da fila **q**.
  - FilaPop(q) – Retorna (e remove) o item na **frente** da fila **q**.
  - FilaInicializa(q) – Cria uma fila vazia em **q**.
  - FilaCheia(q), FilaVazia(q) – Testa se a fila **q** está cheia ou vazia, respectivamente.
- **Existem opções de estruturas de dados que podem ser usadas para representar listas.**
- As duas representações mais utilizadas são as implementações por meio de arranjos e de ponteiros

Dicas de implementação

- Problemas de utilizar arranjos:
  - Possibilidade de ***overflow***.
  - Em caso de **fila**: dificuldade de manutenção.

- Assim, se utilizar array:
  - Use dois índices para implementar filas: **inicio** (head) e **fim** (tail), de forma a evitar movimentações em toda a fila.
  - Use um contador para implementar as funções FilaCheia e FilaVazia.

Dicionários

- Permite **retornar um item a partir de sua chave**, ao invés de utilizar a posição do elemento
- Operações primárias:
  - DicInsere(d, x) – Insere item **x** no dicionário **d**.
  - DicRemove(d, x) – Remove item **x** (ou o item para o qual **x** aponta) do dicionário **d**.
  - DicBusca(d, k) – Retorna o item com chave **k** do dictionário **d** (se o item existir).
  - DicInicializa(d) – Inicializa o dicionário **d**.
  - DicCheio(d), DicVazio(d) – Testa se o dicionário **d** está cheio ou vazio, respectivamente.

Dicionários

- Arrays ordenados e não-ordenados
- Listas ordenadas e não-ordenadas
- Árvores AVL
- Tabelas Hash

- Qual deles utilizar? O que analisar?
  - **Performance?**
  - **Dados serão modificados?**

Dicionários

- Dicionários estáticos
  - Dados não mudam
  - Utilizar **arrays** (ordenados ou não-ordenados)
  - Fazer busca binária apenas se **n >> 1000**
- Dicionários semi-estáticos
  - Apenas inserções e buscas (pouquíssimas remoções)
  - **Tabelas Hash** (com endereçamento aberto)
- Dicionários totalmente dinâmicos
  - Dados são modificados o tempo todo
  - **Tabelas Hash** (caprichar na função de Hash)

Filas de Prioridade

- Estruturas de dados que suportam as operações:
  - FPInsere(q, x) – Insere item **x** na fila de prioridade **q**.
  - FPRemove(q) – Remove e retorna o maior item da fila de prioridade **q**.
  - Maximo(q) – Retorna o maior item da fila de prioridade **q** (se o item existir).
  - FPInicializa(q) – Inicializa a fila de prioridade **d**.
  - FPCheio(q), FPVazio(q) – Testa se a fila de prioridade **q** está cheia ou vazia, respectivamente.

Filas de Prioridade

- Como implementar?
- Quando usar?

- Um heap binário é uma forma eficiente.
- Se houver poucas inserções, que tal um array ordenado?
- Utilizar para manter cronogramas e calendários, determinando próximas tarefas.

Conjuntos

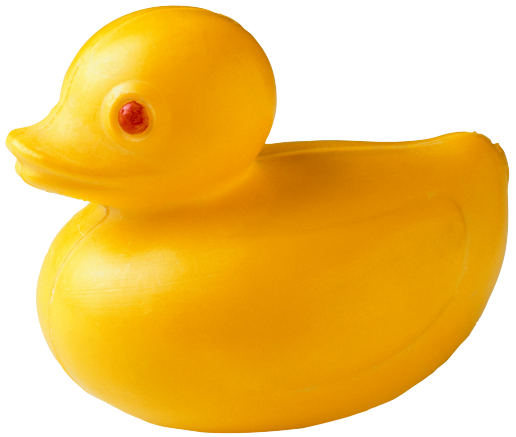
- Coleção de elementos.
- Deve implementar as seguintes operações básicas:
  - Membro(S, x) – Retorna *true* se o item **x** for membro do conjunto **S**.
  - Uniao(A, B) – Retorna um novo conjunto A ∪ B.
  - Intersecao(A, B) – Retorna um novo conjunto A ∩ B.
  - SetInsere(S, x), SetRemove(S, x) – Insere e remove, respectivamente, o item **x** do conjunto **S**.
  - SetInicializa(S) – Inicializa um conjunto **S** vazio.

Conjuntos

- Dicionário é uma boa opção para conjuntos maiores.
- Dicionários ordenados facilitam a operação de interseção.

- Para conjuntos menores, que tal um **array de bits**, com uma posição para cada um dos possíveis elementos?
- Uso de array para representar um conjunto **U**
  - Eficiente até para valores maiores de |**U**|
  - Exemplo: array de 1.000 inteiros é suficiente para representar um conjunto de até 32.000 elementos !!!

Que tal implementar um heap binário ou uma tabela hash durante uma competição?



Bibliotecas de objetos

- **Desvantagem** de utilizar a linguagem **C**
  - Não existe uma biblioteca de estruturas de dados de uso geral
  - Impossível criar uma pilha genérica.
  - Seriam necessárias múltiplas definições:
    - push_char()
    - push_int()
    - push_estudante()
    - push_outro_tipo_dados()

Bibliotecas de Objetos

- **C++ STL** (Standard Template Library) !!!

Fácil de usar

Código eficiente

Agiliza a implementação

```
#include <stdio.h>
#include <stack>

using namespace std;

int main() {
    queue<int> s1;
    queue<char> s2;

    s1.push(10);
    s1.push(20);
    ...
    printf("%d\n", s1.top()); s1.pop();
    printf("%d\n", s1.top()); s1.pop();

    return 0;
}
```

Programação Genérica

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

Programação Genérica

- Por exemplo, podemos criar uma função genérica que ordene um vetor
  - 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
  - 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.

Programação Genérica

TEMPLATES

FUNÇÕES

Funções Genéricas

- Funções sobrecarregadas realizam operações em diferentes tipos de dados
  - Soma de int, float, e frações;
  - Se as operações são idênticas para diferentes tipos, podem ser expressas por meio de funções genéricas.
- O programador escreve a definição da função genérica
  - O **compilador** gera as especializações para cada tipo de chamada.

Funções Genéricas

- Uma definição de função genérica começa com a palavra **template** seguida de uma lista de **parâmetros genéricos** entre **<** e **>**
  - Cada parâmetro deve ser precedido por **class** ou **typename**.
    - Especificam que os parâmetro serão de qualquer tipo primitivo.

```
template< typename T >
template< class TipoElemento >
template< typename TipoBorda, typename TipoPreenchimento >
```

Funções Genéricas

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

// Definição da template de função printArray
template< typename T >
void printArray (const T *array, int count) {
   for (int i = 0; i < count; i++)
      cout << array[i] << " ";
   cout << endl;
}
```

Funções Genéricas

```
int main() {
   const int ACOUNT = 5; // tamanho do array a
   const int BCOUNT = 7; // tamanho do array b
   const int CCOUNT = 6; // tamanho do array c

   int a[ ACOUNT ] = { 1, 2, 3, 4, 5 };
   double b[ BCOUNT ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
   char c[ CCOUNT ] = "HELLO"; // posição 6 para ‘/0’

   cout << "O vetor a contém:" << endl;
   printArray( a, ACOUNT );

   cout << "O vetor b contém:" << endl;
   printArray( b, BCOUNT );

   cout << "O vetor c contém:" << endl;
   printArray( c, CCOUNT );

   return 0;
}
```

Funções Genéricas

- Quando o compilador detecta a chamada a *printArray*(), ele procura a definição da função e **cria três especializações**
  - **void** printArray**(const int *, int);**
  - **void** printArray**(const double *, int);**
  - **void** printArray**(const char *, int);**

Funções Genéricas

- Todas as ocorrências de ***T*** serão substituídas pelo tipo adequado;
- Note que, se um genérico é invocado com parâmetros que são tipos definidos pelo programador e o genérico usa operadores, estes devem ser sobrecarregados
  - e.g., ==, +, <=, etc;
  - Caso contrário, haverá erro de **compilação**.

Funções Genéricas

- Uma função genérica pode ser sobrecarregada
  - Podemos definir diferentes funções genéricas com o mesmo nome, porém, com parâmetros diferentes.
- Por exemplo, poderíamos adicionar um terceiro parâmetro à função *printArray*() que indicasse se o vetor deve ser impresso do início para o final ou do final para o início

```
void printArray( const T *, int );
void printArray( const T *, int, int);
```

- Podemos também sobrecarregar uma função genérica criando uma função não genérica.

STANDARD TEMPLATES LIBRARY

S T L

STL

- A Standard Template Library (STL) foi adicionada à biblioteca padrão C++;
- A STL define componentes genéricos reutilizáveis que implementam várias estruturas de dados e algoritmos que processam estas estruturas;
- Basicamente, a STL é composta de **contêineres**, **iteradores** e **algoritmos**.

STL

- **Contêineres** são *templates* de estruturas de dados
  - Possuem métodos associados a eles.
- **Iteradores** são ponteiros, utilizados para percorrer e manipular os elementos de um contêiner;
- **Algoritmos** são as funções que realizam operações tais como buscar, ordenar e comparar elementos ou contêineres inteiros
  - Existem aproximadamente **70** algoritmos implementados na STL;
  - A maioria utiliza iteradores para acessar os elementos de contêineres.

STANDARD TEMPLATES LIBRARY

CONTÊINERS

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, e, versões restringidas.

Contêineres

Contêineres Sequenciais	Descrição
vector	Inserções e remoções no final, acesso direto a qualquer elemento.
deque	Fila duplamente ligada, inserções e remoções no início ou no final, acesso direto a qualquer elemento.
list	Lista duplamente ligada, inserção e remoção em qualquer ponto.

Contêineres

Contêineres Associativos	Descrição
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.

Contêineres

Adaptadores de Contêineres	Descrição
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
  - Muitas operações genéricas se aplicam a todos os contêineres
  - Outras operações se aplicam a subconjuntos de contêineres similares

Funções Comuns a Todos Contêineres

Funcionalidade	Descrição
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.

Funções Comuns a Todos Contêineres

Funcionalidade	Descrição
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.

Funções Comuns a Todos Contêineres

- **Atenção!**
  - Os operadores **<**, **<=**, **>**, **>=**, **==** e **!=** não são fornecidos para o contêiner ***priority_queue*** (fila de prioridade).

Funções de Contêineres Sequenciais e Associativos

Funcionalidade	Descrição
max_size	Número máximo de elementos de um contêiner.
begin	As duas versões deste método retornam um iterator para o primeiro elemento do contêiner.
end	As duas versões deste método retornam um iterator para a posição após o final do contêiner.
rbegin	As duas versões deste método retornam um reverse_iterator para o primeiro elemento do contêiner invertido.
rend	As duas versões deste método retornam 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

Biblioteca	Observação
<vector>	Vetor
<list>	Lista
<deque>	Fila duplamente ligada
<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).

Contêineres

- É necessário garantir que os elementos armazenados em um contêiner suportam um conjunto mínimo de operações
  - Quando um elemento é inserido em um contêiner, ele é copiado: o operador de atribuição deve ser sobrecarregado se necessário;
  - Dica: pode ser necessário que os operadores **==** e **<** devem ser sobrecarregados.

STANDARD TEMPLATES LIBRARY

ITERADORES

Iteradores

- **Iteradores** são utilizados para apontar elementos de contêineres sequenciais e associativos
  - Entre outras coisas;
  - Algumas funcionalidades como *begin()* e *end()* retornam iteradores.
- Se um iterador ***i*** aponta para um elemento:
  - ***++i*** aponta para o próximo elemento;
  - ****i*** é o conteúdo do elemento apontado por i. (funciona como um ponteiro !!!)

Iteradores

- Os iteradores são objetos declarados na biblioteca ***<iterator>***;
- Existem basicamente dois tipos de objetos iteradores:
  - *iterator*: aponta para um elemento que pode ser modificado;
  - *const_iterator*: aponta para um elemento que não pode ser modificado.

Categorias de Iteradores

Categoria	Descrição
input	Utilizado para ler um elemento de um contêiner. Só se move do início para o final, um elemento por vez. Não pode ser utilizado para percorrer um contêiner mais que uma vez.
output	Utilizado para escrever um elemento em um contêiner. Só se move do início para o final, um elemento por vez. Não pode ser utilizado para percorrer um contêiner mais que uma vez.
forward	Combina os iteradores input e output e retém a sua posição no contêiner.

Categorias de Iteradores

Categoria	Descrição
bidirectional	Combina o iterador forward com a capacidade de se mover do final para o início. Pode ser utilizado para percorrer um contêiner mais que uma vez.
random access	Combina o iterador bidirectional com a capacidade de acessar diretamente qualquer elemento. Ou seja, pode saltar uma quantidade arbitrária de elementos.

Tipos Predefinidos de Iteradores

Tipo Predefinido	Direção do ++	Capacidade
Iterator	Início para o Final	Leitura/Escrita
const_iterator	Início para o Final	Leitura
reverse_iterator	Final para o Início	Leitura/Escrita
const_reverse_iterator	Final para o Início	Leitura

Operações em Iteradores

Todos Iteradores	Descrição
++p	Incremento prefixado.
p++	Incremento pós-fixado.

Input	Descrição
*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.

Output	Descrição
*p	Referencia o conteúdo apontado.
p = p1	Atribui um iterador a outro.

Iteradores Bidirecionais

Bidirectional	Descrição
--p	Decremento prefixado.
p--	Decremento pós-fixado.

- Iteradores da categoria *forward* posssuem todas as funcionalidades dos iteradores das categorias *input* e *output*.

Operações em Iteradores

Random Access	Descrição
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.
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	Retonar se o primeiro iterador está antes do segundo no contêiner.
p < =p1	Retonar se o primeiro iterador está antes ou na mesma posição do segundo no contêiner.
p > p1	Retonar se o primeiro iterador está após o segundo no contêiner.
p >= p1	Retonar se o primeiro iterador está após ou na mesma posição do segundo no contêiner.

STANDARD TEMPLATES LIBRARY

ALGORITMOS

Algoritmos

- A STL inclui aproximadamente 70 algoritmos
- Os algoritmos operam indiretamente sobre os elementos de um contêiner usando iteradores
  - Vários utilizam **pares de iteradores**, um apontando para o início e outro apontando para o final;
  - Frequentemente os algoritmos também **retornam iteradores** como resultado;
  - Este desacoplamento dos contêineres permite que os algoritmos sejam **genéricos**.

Algoritmos

copy	remove	reverse_copy
copy_backward	remove_copy	rotate
fill	remove_copy_if	rotate_copy
fill_n	remove_if	stable_partition
generate	replace	swap
generate_n	replace_copy	swap_ranges
iter_swap	replace_copy_if	transform
partition	replace_if	unique
random_shuffle	reverse	unique_copy

Altera sequência

Não Altera sequência

<numeric>

adjacent_find	*find*	*find_if*
count	*find_each*	*mismatch*
count_if	*find_end*	*search*
equal	*find_first_of*	*search_n*

accumulate	*partial_sum*
inner_product	*adjacent_difference*

Ordenação

Algoritmo	Descrição
sort	Ordena os elementos do contêiner
stable_sort	Ordena os elementos do contêiner preservando a ordem relativa dos equivalentes.
partial_sort	Ordena parcialmente o contêiner.
partial_sort_copy	Copia os menores elementos e os ordena no contêiner de destino.
nth_element	Ordena o n-ésimo elemento.

Busca Binária

Algoritmo	Descrição
lower_bound	Retorna um iterador para o limite esquerdo.
upper_bound	Retorna um iterador para o limite direito.
equal_range	Retorna os limites que incluem um conjunto de elementos com um determinado valor.
binary_search	Testa se um valor existe em um intervalo.

Intercalação

Algoritmo	Descrição
merge	Intercala os elementos de dois intervalos e coloca o resultado em outro intervalo.
inplace_merge	Intercala os elementos de dois intervalos e coloca o resultado no mesmo intervalo.
includes	Testa se um intervalo ordenado inclui outro intervalo ordenado, se cada elemento de um intervalo é equivalente a outro do segundo intervalo.
set_union	Calcula a união entre dois intervalos de valores.
set_intersection	Calcula a interseção entre dois intervalos de valores.
set_difference	Calcula a diferença entre dois intervalos de valores.
set_symmetric_difference	Calcula a diferença simétrica entre dois intervalos de valores.

Heap

Algoritmo	Descrição
push_heap	Adiciona um elemento a um heap.
pop_heap	Remove um elemento de um heap.
make_heap	Cria um heap a partir de um intervalo de valores.
sort_heap	Ordena os elementos de um heap.

Min/Max

Algoritmo	Descrição
min	Retorna o menor de dois argumentos.
max	Retorna o maior de dois argumentos.
min_element	Retorna o menor elemento de uma sequência.
max_element	Retorna o maior elemento de uma sequência.
lexicographical_compare	Comparação lexicográfica (menor que).
next_permutation	Transforma uma sequência na próxima permutação (ordem lexicográfica).
prev_permutation	Transforma uma sequência na permutação anterior (ordem lexicográfica).

STANDARD TEMPLATES LIBRARY

EXEMPLOS

Vector

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

int main () {
    unsigned int i;

    vector<int> first;// vector de ints vazio
    vector<int> second (4,100);// quatro inteiros de valor 100
    vector<int> third (second.begin(),second.end());
    vector<int> fourth (third);// copia o terceiro vector

    // o construtor iterador também pode ser utilizado com arrays
    int myints[] = {16,2,77,29};
    vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );
    cout << "O tamanho do quinto é:" << fifth.size() << endl;

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

    ...
```

Vector

```
    ...

    //iterador inicio para o final, somente leitura
    vector< int >::const_iterator it;

    // exibe elementos vector utilizando const_iterator
    for ( it = first.begin(); it != first.end(); ++it )
        cout << *it << ' ';

    return 0;
}
```

Vector

- A classe *vector* é genérica, logo, deve ser definido o tipo na declaração de um objeto;
- Este contêiner é dinâmico
  - A cada inserção o contêiner se redimensiona automaticamente.
- O método ***push_back*** adiciona um elemento ao final do *vector*.

Vector

- Outros possíveis métodos incluem:
  - *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.

Deque

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

int main ()
{
    unsigned int i;

    deque<int> first;                // deque vazio deo tipo int
    deque<int> second (4,100);       // quatro inteiros com o valor 100
    deque<int> third (second.begin(),second.end());  // itera pelo segundo
    deque<int> fourth (third);       // construtor cópia

    // o construtor iterador também pode ser utilizado com arrays
    int myints[] = {16,2,77,29};
    deque<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

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

    ...
```

Deque

```
    ...

    first.push_back( 1 );

    // modifica elemento na localização 1
    first[ 1 ] = 5;

    first.pop_front(); // remove o primeiro elemento

    first.pop_back(); // remove o primeiro elemento

    cout << "O conteúdo é:";
    for (i=0; i < first.size(); i++)
        cout << " " << first[i];

    return 0;
}
```

Deque

- O método *push_front* está disponível apenas para *list* e *deque*;
- O operador **[]** permite acesso direto aos elementos do *deque*
  - Também pode ser utilizado em um *vector*.
- Em geral, um *deque* possui um desempenho levemente inferior em relação a um *vector*
  - No entanto, é mais eficiente para fazer inserções e remoções no início.

Set

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

int main () {
    set<int> first; //conjunto vazio de inteiros
    int myints[]= {10,20,30,40,50};
    set<int> second (myints, myints+5); // ponteiros como iteradores
    set<int> third (second);  //construtor cópia

    first.insert(10);
    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;
}
```

map

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

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

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

    map<char,int> second (first.begin(),first.end());//construtor iterador
    map<char,int> third (second);//construtor cópia

    ...
```

map

```
    ...

    //localiza a ocorrência do elemento
    multimap<char,int>::iterator it = first.find('a');

    //imprime o par
    cout << it->first << '\t' << it->second << '\n';

    //imprime os pares do mapa
    for (map<char, int>::const_iterator it = first.begin();
                                        it != first.end(); ++it)
        cout << it->first << '\t' << it->second << '\n';

    return 0;
}
```

map

- O contêiner *map* possui os mesmos métodos *find*(), *count*(), *swap*() e *clear*().

multimap

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

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

    //insere os pares
    first.insert(pair<char,int>('a',10));
    first.insert(pair<char,int>('b',15));

    //localiza a ocorrência do elemento
    multimap<char,int>::iterator it = first.find('a');
    first.erase(it);

    for (multimap<char, int>::const_iterator it = first.begin();
                                              it != first.end(); ++it)
        cout << it->first << '\t' << it->second << '\n';
    return 0;
}
```

multimap

- O contêiner *multimap* possui os mesmos métodos *find*(), *count*(), *swap*() e *clear*().

Contêineres Adaptativos

- Os contêineres adaptativos (pilha, fila e fila de prioridades) contém praticamente os mesmos métodos:
  - *empty*: testa se o contêiner está vazio;
  - *size*: retonar 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

```
#include <iostream>
#include <stack> // definição de stack adapter
#include <vector> // definição da template de classe vector
#include <list> // definição da template de classe list
namespace std;

int main()
{
    // pilha com deque subjacente padrão
    stack< int > intDequeStack;

    // pilha com vetor subjacente
    stack< int, std::vector< int > > intVectorStack;

    // pilha com lista subjacente
    stack< int, std::list< int > > intListStack;

    ...
```

stack

```
    ...

    // insere os valores em cada pilha
    intDequeStack.push(1);
    intVectorStack.push(1);
    intListStack.push(1);

    // remove elementos de cada pilha
    intDequeStack.pop();
    intVectorStack.pop();
    intListStack.pop();

    return 0;
}
```

queue

```
#include <iostream>
#include <queue> // definição da classe queue adaptadora
namespace std;

int main() {
   queue< double > values; // fila com doubles

   // insere elementos nos valores de fila
   values.push(3.2);
   values.push(9.8);
   values.push(5.4);

   // remove elementos da fila
   while (!values.empty()) {
      cout << values.front() << ' '; // visualiza elemento da frente
      values.pop(); // remove o elemento
   }

   cout << endl;
   return 0;
}
```

priority_queue

```
#include <iostream>
#include <queue> // definição do adaptador priority_queue
namespace std;

int main() {
   priority_queue< double > priorities; // cria priority_queue

   // insere elementos em prioridades
   priorities.push(3.2);
   priorities.push(9.8);
   priorities.push(5.4);

   // remove elemento de priority_queue
   while (!priorities.empty())  {
      cout << priorities.top() << ' '; // visualiza elemento superior
      priorities.pop(); // remove elemento superior
   }

   cout << endl;
   return 0;
}
```

Comentários Finais

- Ainda há na STL:
  - Classe *bitset* (manipulação de conjuntos de *bits*);
  - Objetos Função.

Perguntas?