



BCC221

# Programação Orientada a Objetos

Prof. Marco Antonio M. Carvalho

2014/2



UFOP

# Endereços Importantes

- Site da disciplina:  
<http://www.decom.ufop.br/marco/>
- Moodle:  
[www.decom.ufop.br/moodle](http://www.decom.ufop.br/moodle)
- Lista de e-mails:  
[bcc221-decom@googlegroups.com](mailto:bcc221-decom@googlegroups.com)
- Para solicitar acesso:  
<http://groups.google.com/group/bcc221-decom>



UFOP



# Avisos



# Avisos

UFOP



# Na aula passada

UFOP

- Polimorfismo
- Exceções
  - *try, throw e catch*
  - Modelo de Terminação
  - Erros comuns
  - Quando Utilizar Exceções?
  - Classes de Exceções da Biblioteca Padrão



# Na aula de hoje

UFOP

- Genéricos (*Templates*)
  - Programação Genérica
  - Funções Genéricas
  - Classes Genéricas
    - Outros Parâmetros e Parâmetros Padronizados
  - Observações Sobre Genéricos e Herança
  - Observações Sobre Genéricos e Funções Amigas
  - Observações Sobre Genéricos e Membros *static*
- *Standard Templates Library*
  - Contêineres
  - Iteradores
  - Algoritmos
  - Exemplos



# Programação Genérica

UFOP

- 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 (sobre carregadas);
  - Esta técnica é chamada *programação genérica*.



# Programação Genérica

UFOP

- 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

UFOP



# Funções Genéricas



# Funções Genéricas

- Funções sobre carregadas 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
  - Baseado nos parâmetros explicitamente enviados ou inferidos a partir da chamada da função, 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 parâmetros serão de qualquer tipo primitivo.

**template< typename T >**

**template< class TipoElemento >**

**template< typename TipoBorda, typename TipoPreenchimento >**



# Exemplo

UFOP

```
#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;
} // fim do template de função printArray
```



# Exemplo

UFOP

```
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 null

    cout << "O vetor a contém:" << endl;
    // chama a especialização da template de função do tipo inteiro
    printArray( a, ACOUNT );

    cout << " O vetor b contém:" << endl;
    // chama a especialização da template de função do tipo double
    printArray( b, BCOUNT );

    cout << "O vetor c contém:" << endl;
    // chama a especialização da template de função do tipo caractere
    printArray( c, CCOUNT );
    return 0;
}
```



# Saída

UFOP

O vetor a contém:

1 2 3 4 5

O vetor b contém:

1.1 2.2 3.3 4.4 5.5 6.6 7.7

O vetor c contém:

H E L L O



# Funções Genéricas

- Quando o compilador detecta a chamada a *printArray()*, ele procura a definição da função
  - Neste caso, a função genérica;
  - Ao comparar os tipos dos parâmetros, nota que há um tipo genérico;
  - Então deduz qual deverá ser a substituição a ser feita
    - *T* por *int*.
  - O compilador cria três especializações
    - **void printArray( const int \*, int );**
    - **void printArray( const double \*, int );**
    - **void printArray( const char \*, int );**



# Funções Genéricas

UFOP

- 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 sobre carregados
  - e.g., `==`, `+`, `<=`, etc;
  - Caso contrário, haverá erro de compilação.



# Funções Genéricas

UFOP

- Uma função genérica pode ser sobre carregada
  - 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 sobre carregar uma função genérica criando uma função não genérica.

# Classes Genéricas



# Classes Genéricas

UFOP

- 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.
- Esta é uma das grandes oportunidade de reuso de *software*
  - O ideal é descrever uma pilha genericamente, assim como a entendemos;
  - Instanciar versões específicas desta **classe genérica** fica por conta do compilador.



# Classes Genéricas

UFOP

- *Classes Genéricas* (ou *Tipos Parametrizados*) requerem um ou mais parâmetros de tipo que especifiquem como customizá-las
  - Gerando assim **especializações de classes genéricas**.
- O programador codifica apenas a definição da classe genérica
  - A cada vez que uma especialização diferente for necessária, o compilador codifica a especialização;
  - Uma classe genérica Pilha vira uma coleção de classes especializadas
    - Pilha de int, float, string, frações, restaurantes, etc.



# Classes Genéricas

UFOP

- A definição de uma classe genérica é semelhante à definição de uma classe normal
  - Antes da definição da classe, adiciona-se o cabeçalho **template< typename T >;**
  - Em que o T representa o **tipo** que a classe manipula, passado como um parâmetro
    - Na verdade, qualquer identificador serve, mas T é padrão.
  - Em caso de tipos definidos pelo programador, deve-se tomar cuidados com a sobrecarga de operadores e também garantir a existência de pelo menos um construtor (o *default*).



# Stack.h

UFOP

```
template< typename T >
class Stack
{
public:
    Stack( int = 10 ); // construtor padrão (tamanho de Stack 10)

    // destrutor
    ~Stack()
    {
        delete [] stackPtr; // desaloca o espaço interno para Stack
    }

    bool push( const T& ); // insere (push) um elemento na Stack
    bool pop( T& ); // remove (pop) um elemento da Stack

    // determina se a Stack está vazia
    bool isEmpty() const
    {
        return top == -1;
    }
}
```



# Stack.h

UFOP

```
// determina se Stack está cheia
bool isFull() const
{
    return top == size - 1;
}

private:
int size; // número de elementos na Stack
int top; // localização do elemento superior (-1 significa vazio)
T *stackPtr; // ponteiro para a representação interna da Stack
};

// template construtora
template< typename T >
Stack< T >::Stack( int s )
    : size( s > 0 ? s : 10 ), // valida o tamanho
    top( -1 ), // Stack inicialmente vazia
    stackPtr( new T[ size ] ) // aloca memória para elementos
{
    // corpo vazio
}
```



# Stack.h

```
// insere elemento na Stack;
template< typename T >
bool Stack< T >::push( const T &pushValue )
{
    if ( !isFull() )
    {
        stackPtr[ ++top ] = pushValue; // insere item em Stack
        return true; // inserção bem-sucedido
    }
    return false; // inserção mal-sucedido
}
// remove elemento da Stack;
template< typename T >
bool Stack< T >::pop( T &popValue )
{
    if ( !isEmpty() )
    {
        popValue = stackPtr[ top-- ]; // remove item da Stack
        return true; // remoção bem-sucedida
    }
    return false; // remoção mal-sucedida
}
```



# Classes Genéricas

UFOP

- Os membros de uma classe genérica são funções genéricas
  - As definições que aparecem fora da classe devem ser precedidas pelo cabeçalho  
**template< typename T >**
  - O operador de escopo utiliza o nome da classe genérica Stack <T>;
  - Na implementação, os elementos da pilha aparecem genericamente como sendo do tipo T;



# driverStack.cpp

UFOP

```
#include <iostream>
using namespace std;
#include "Stack.h" // Definição de template de classe Stack

int main()
{
    Stack< double > doubleStack( 5 ); // tamanho 5
    double doubleValue = 1.1;

    // insere 5 doubles em doubleStack
    while ( doubleStack.push( doubleValue ) )
    {
        cout << doubleValue << ' ';
        doubleValue += 1.1;
    }

    while ( doubleStack.pop( doubleValue ) )
        cout << doubleValue << ' ';
```



# driverStack.cpp

UFOP

```
Stack< int > intStack; // tamanho padrão de 10
int intValue = 1;

// insere 10 inteiros em intStack
while ( intStack.push( intValue ) )
{
    cout << intValue << ' ';
    intValue++;
}

// remove elementos de intStack
while ( intStack.pop( intValue ) )
    cout << intValue << ' ';

return 0;
}
```



# Classes Genéricas

UFOP

- Para instanciarmos um objeto de uma classe genérica, precisamos informar qual tipo deve ser associado à classe
  - No exemplo, declaramos dois objetos, associando os tipos **double** e **int**;
  - O compilador substituirá o tipo do **T** da definição da classe pelo tipo informado, e criará uma nova codificação da classe;
  - Note que o código do *driver* é idêntico para as duas pilhas criadas.

# Outros Parâmetros e Parâmetros Padronizados



# Outros Parâmetros

- Nossa classe genérica do exemplo anterior recebia apenas um parâmetro, que representava um tipo
  - No entanto, uma classe pode receber mais que um parâmetro;
  - O parâmetro não precisa necessariamente representar um tipo;
  - Por exemplo, nossa classe genérica poderia receber também um inteiro que representasse o tamanho da pilha.

```
template< typename T, int elementos >
class Stack
```



# Parâmetros Padronizados

UFOP

- Outra possibilidade é definir valores padrão para os parâmetros de uma classe genérica:  
**template< typename T = int >**  
**class** Stack
- No exemplo acima, caso um tipo não seja especificado, será criada uma pilha de inteiros pela chamada abaixo:  
    Stack<> intStack( 5 );
- Parâmetros padronizados devem ser definidos mais à direita na lista de parâmetros
  - Quando instanciamos uma classe com um ou mais parâmetros padronizados, se um parâmetro omitido não é o mais à direita, então todos os parâmetros depois dele também devem ser omitidos.

# Observações Sobre Genéricos e Herança

# Observações Sobre Genéricos e Herança



UFOP

- Uma classe genérica pode ser derivada a partir de uma especialização de classe genérica;
- Uma classe genérica pode ser derivada a partir de uma classe não genérica;
- Uma especialização de classe genérica pode ser derivada a partir de outra especialização;
- Uma classe não genérica pode ser derivada a partir de uma especialização de uma classe genérica.

# Observações Sobre Genéricos e Funções Amigas

# Observações Sobre Genéricos e Funções Amigas



UFOP

- Como vimos, funções e classes podem ser declaradas amigas de classes não genéricas;
- Com genéricos a amizade pode ser estabelecida entre uma classe genérica e:
  1. Uma função global;
  2. Um método de outra classe (possivelmente uma especialização de classe genérica);
  3. Uma classe (possivelmente uma especialização de classe genérica).



# 1- Função Global

UFOP

- Considerando uma classe genérica X, a declaração

**friend void f1();**

- Torna a função f1() amiga de todas as especializações da classe X.

- A declaração

**friend void f2( X< float > & );**

- Torna a função f2() amiga apenas da especialização x< float >.



## 2- Método de Outra Classe

- Considerando uma classe genérica X e uma classe A, a declaração  
**friend void A::f3();**
- Torna o método f3() da classe A amigo de todas as especializações da classe X;
- A declaração  
**A< float >::f4( X< float > & );**
- Torna o método f4() da especialização A < float > amigo apenas da especialização x< float >.



## 3- Outra Classe

UFOP

- Considerando uma classe genérica X e uma classe Y, a declaração  
**friend class Y;**
- Torna a classe Y amiga de todas as especializações da classe X;
- A declaração  
**friend class Y< float >;**
- Torna a especialização Y< float > amiga apenas da especialização x< float >.

# Observações Sobre Genéricos e Membros *static*

# Observações Sobre Genéricos e Membros *static*



UFOP

- Relembrando, em uma classe não genérica, uma única cópia de cada membro ***static*** é compartilhada entre todos os objetos;
- Em programação genérica, cada especialização de classe genérica instanciada possui sua própria cópia do membro ***static***
  - Os seus objetos a compartilham.



UFOP



**Continua na  
próxima aula...**

# *Standard Templates Library*



# STL

UFOP

- 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 de **contêineres, iteradores e algoritmos**.



- **Contêineres** são *templates* de estruturas de dados
  - Possuem métodos associados a eles.
- **Iteradores** são semelhantes a 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 85 algoritmos implementados na STL;
  - A maioria utiliza iteradores para acessar os elementos de contêineres.



# C++11

UFOP

- C++11 é o padrão mais recente da linguagem C++
  - Aprovado pela ISO em 12 de Agosto de 2011.
- O C++11 estendeu a STL
  - Facilidades *multithreading*;
  - Tabelas *hash*;
  - Expressões regulares;
  - Números aleatórios;
  - Novos algoritmos.
- Para compilar códigos que utilizem o C++11 use a *flag* `-std=c++11`.

# Contêineres



# Contêineres

UFOP

- 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 restrinvidas.



# Contêineres

UFOP

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



# Contêineres

UFOP

Contêineres Associativos	Descrição
<i>set</i>	Busca rápida, não permite elementos duplicados.
<i>multiset</i>	Busca rápida, permite elementos duplicados.
<i>map</i>	Mapeamento um-para-um, não permite elementos duplicados, busca rápida.
<i>multimap</i>	Mapeamento um-para-um, permite elementos duplicados, busca rápida.



# Contêineres

UFOP

Adaptadores de Contêineres	Descrição
<i>stack</i>	Last-in, first out (LIFO)
<i>queue</i>	First –in, first out (FIFO)
<i>priority_queue</i>	O elemento de maior prioridade é sempre o primeiro elemento a sair.



# Contêineres

UFOP

- Todos os contêineres da STL fornecem funcionalidades similares
  - Muitas operações genéricas se aplicam a todos os contêineres
    - Como determinar sua quantidade de elementos.
  - Outras operações se aplicam a subconjuntos de contêineres similares

# Funções Comuns a Todos Contêineres



UFOP

Funcionalidade	Descrição
<b>Construtor default</b>	Fornece a inicialização padrão do contêiner.
<b>Construtor Cópia</b>	Construtor que inicializa um contêiner para ser a cópia de outro do mesmo tipo.
<b>Destrutor</b>	Simplesmente destrói o contêiner quando não for mais necessário.
<b><i>empty</i></b>	Retorna <i>true</i> se não houver elementos no contêiner e <i>false</i> caso contrário.
<b><i>size</i></b>	Retorna o número de elementos no contêiner.
<b><i>operator=</i></b>	Atribui um contêiner a outro.
<b><i>operator&lt;</i></b>	Retorna <i>true</i> se o primeiro contêiner for menor que o segundo e <i>false</i> caso contrário.

# Funções Comuns a Todos Contêineres



UFOP

Funcionalidade	Descrição
<code>operator&lt;=</code>	Retorna <i>true</i> se o primeiro contêiner for menor ou igual ao segundo e <i>false</i> caso contrário.
<code>operator&gt;</code>	Retorna <i>true</i> se o primeiro contêiner for maior que o segundo e <i>false</i> caso contrário.
<code>operator&gt;=</code>	Retorna <i>true</i> se o primeiro contêiner for maior ou igual ao segundo e <i>false</i> caso contrário.
<code>operator==</code>	Retorna <i>true</i> se o primeiro contêiner for igual ao segundo e <i>false</i> caso contrário.
<code>operator!=</code>	Retorna <i>true</i> se o primeiro contêiner for diferente do segundo e <i>false</i> caso contrário.
<code>swap</code>	Troca os elementos de dois contêineres.

# Funções Comuns a Todos Contêineres



UFOP

- **Atenção!**
  - Os operadores `<`, `<=`, `>`, `>=`, `==` e `!=` não são fornecidos para o contêiner *priority\_queue*.



# Funções Comuns a Contêineres Sequenciais e Associativos

UFOP

Funcionalidade	Descrição
<b><i>max_size</i></b>	Retorna o número máximo de elementos de um contêiner.
<b><i>begin</i></b>	As duas versões deste método retornam um <i>iterator</i> ou um <i>const_iterator</i> para o primeiro elemento do contêiner.
<b><i>end</i></b>	As duas versões deste método retornam um <i>iterator</i> ou um <i>const_iterator</i> para a posição após o final do contêiner.
<b><i>rbegin</i></b>	As duas versões deste método retornam um <i>reverse_iterator</i> ou um <i>const_reverse_iterator</i> para o primeiro elemento do contêiner invertido.
<b><i>rend</i></b>	As duas versões deste método retornam um <i>reverse_iterator</i> ou um <i>const_reverse_iterator</i> para a posição após o final do contêiner invertido.
<b><i>erase</i></b>	Apaga um ou mais elementos do contêiner.
<b><i>clear</i></b>	Apaga todos os elementos do contêiner.



# Bibliotecas de Contêineres

UFOP

Biblioteca	Observação
<code>&lt;vector&gt;</code>	Vetor
<code>&lt;list&gt;</code>	Lista
<code>&lt;deque&gt;</code>	Fila duplamente ligada
<code>&lt;queue&gt;</code>	Contém <i>queue</i> e <i>priority_queue</i>
<code>&lt;stack&gt;</code>	Pilha
<code>&lt;map&gt;</code>	Contém <i>map</i> e <i>multimap</i>
<code>&lt;set&gt;</code>	Contém <i>set</i> e <i>multiset</i>
<code>&lt;bitset&gt;</code>	Conjunto de <i>bits</i> (vetor em que cada elemento é um <i>bit</i> – 0 ou 1).



# Contêineres

UFOP

- 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
    - Logo, o operador de atribuição deve ser sobre carregado se necessário;
    - Também deve haver um construtor cópia.
  - Contêineres associativos e alguns algoritmos requerem que os elementos sejam comparados
    - Pelo menos os operadores == e < devem ser sobre carregados.

# Iteradores



# Iteradores

UFOP

- **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;
  - $*i$  se refere ao conteúdo do elemento apontado por *i*.



# Iteradores

UFOP

- 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.



# Iteradores

UFOP

- Por exemplo, é possível criar iteradores para o *cin* e o *cout*
  - São sequências de dados, assim como os contêineres
  - É possível percorrer os dados captados por um *cin* e os dados a serem escritos por um *cout*.



# Iteradores

UFOP

```
#include <iostream>
using namespace std;
#include <iterator> // ostream_iterator e istream_iterator

int main()
{
    cout << "Informe dois inteiros: ";
    // cria istream_iterator para ler valores de int a partir de cin
    istream_iterator< int > inputInt( cin );
    int number1 = *inputInt; // lê int a partir da entrada padrão
    ++inputInt; // move iterador para o próximo valor de entrada
    int number2 = *inputInt; // lê int a partir da entrada padrão
    // cria ostream_iterator para gravar valores int em cout
    ostream_iterator< int > outputInt( cout );
    cout << "A soma é: ";
    *outputInt = number1 + number2; // gera saída do resultado para cout
    cout << endl;
    return 0;
}
```



# Iteradores

UFOP

Informe dois inteiros

12 25

A soma é: 37



# Categorias de Iteradores

UFOP

Categoria	Descrição
<i>input</i>	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.
<i>output</i>	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.
<i>forward</i>	Combina os iteradores <i>input</i> e <i>output</i> e retém a sua posição no contêiner.



# Categorias de Iteradores

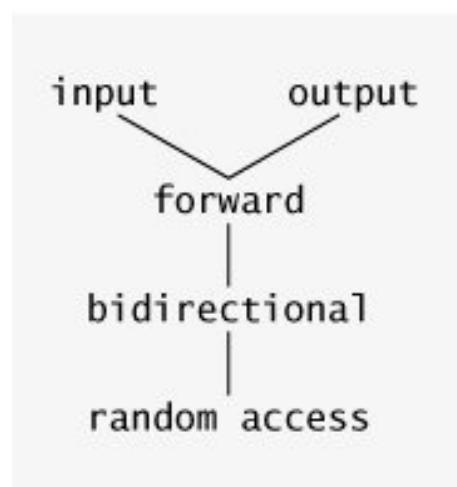
UFOP

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



# Categorias de Iteradores

- Usar o iterador “mais fraco” maximiza a reusabilidade
  - Um algoritmo que usa um iterador *forward* aceita iteradores *forward*, *bidirectional* ou *random access*
  - Um algoritmo que usa um iterador *random access* não aceira outro.





# Tipos Predefinidos de Iteradores

UFOP

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



# Tipos Predefinidos de Iteradores

UFOP

- Nem todos os tipos predefinidos o são para todos os contêineres;
- As versões *const* são utilizadas para percorrer contêineres de somente leitura;
- Iteradores invertidos são utilizados para percorrer contêineres na direção inversa.



# Operações em Iteradores

UFOP

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

Input	Descrição
$*p$	Referencia o conteúdo apontado.
$p = p_1$	Atribui um iterador a outro.
$p == p_1$	Compara dois iteradores quanto a igualdade.
$p != p_1$	Compara dois iteradores quanto a desigualdade.

Output	Descrição
$*p$	Referencia o conteúdo apontado.
$p = p_1$	Atribui um iterador a outro.



# Operações em Iteradores

<i>Bidirectional</i>	Descrição
$--p$	Decremento prefixado.
$p--$	Decremento pós-fixado.

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



# Operações em Iteradores

UFOP

## *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.



# 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 $p$ .
$p < p_1$	Retonar <i>true</i> se o primeiro iterador estiver antes do segundo no contêiner.
$p <= p_1$	Retonar <i>true</i> se o primeiro iterador estiver antes ou na mesma posição do segundo no contêiner.
$p > p_1$	Retonar <i>true</i> se o primeiro iterador estiver após o segundo no contêiner.
$p >= p_1$	Retonar <i>true</i> se o primeiro iterador estiver após ou na mesma posição do segundo no contêiner.

# Algoritmos



# Algoritmos

UFOP

- A STL inclui aproximadamente 85 algoritmos
  - Podem ser utilizados genericamente, em vários tipos de contêineres.
- Os algoritmos operam indiretamente sobre os elementos de um contêiner usando iteradores
  - Vários deles 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

UFOP

- A STL é extensível
  - Ou seja, é possível adicionar novos algoritmos a ela.
- Entre os vários algoritmos disponíveis, temos:
  - Algoritmos alteradores de sequências;
  - Algoritmos não alteradores de sequências;
  - Algoritmos numéricos
    - Definidos em *<numeric>*.



# Algoritmos

UFOP

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

	Altera sequência
	Não Altera sequência
	<numeric>

<i>adjacent_find</i>	<i>find</i>	<i>find_if</i>
<i>count</i>	<i>find_each</i>	<i>mismatch</i>
<i>count_if</i>	<i>find_end</i>	<i>search</i>
<i>equal</i>	<i>find_first_of</i>	<i>search_n</i>

<i>accumulate</i>	<i>partial_sum</i>
<i>inner_product</i>	<i>adjacent_difference</i>



# Ordenação

UFOP

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



# Busca Binária

UFOP

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



# Intercalação

UFOP

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



# Heap

UFOP

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



# Min/Max

UFOP

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



UFOP



**Continua na  
próxima aula...**

# Exemplos



# Vector

UFOP

```
#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()); // itera pelo segundo vector
    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) );
```



# Vector

UFOP

```
cout << "O tamanho do quinto é:" <<fifth.size()<<endl;

//adiciona o elemento ao final do vector
first.push_back(3);

//remove o elemento ao final do vector
first.pop_back();

//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

UFOP

- 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

UFOP

- 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.



# List

UFOP

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

int main ()
{
    list<int> first; // lista de inteiros vazia
    list<int> second (4,100); // quatro inteiros de valor 100
    list<int> third (second.begin(),second.end()); // itera pela segunda lista
    list<int> fourth (third); //construtor cópia

    // the iterator constructor can also be used to construct from arrays:
    int myints[] = {16,2,77,29};
    list<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

    first.push_front( 1 ); //insere na frente
    first.push_front( 2 );
```



# List

UFOP

```
first.push_back( 4 ); //insere no final
first.push_back( 1 );

first.remove( 4 ); // remove todos os 4s
first.unique(); // remove elementos duplicados

first.pop_front(); // remove elemento da parte da frente
first.pop_back(); // remove elemento da parte de trás

first.sort(); // ordena values

cout << "O conteúdo é: ";

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

return 0;
}
```



# List

- Neste exemplo, 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.
- Existem outros como:
  - *reverse*: inverte a lista;
  - *merge*: intercala listas;
  - *remove\_if*: remove elementos que atendam um critério.



# Deque

UFOP

```
#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) );
```



# Deque

UFOP

```
first.push_front( 2 );
first.push_front( 3 );
first.push_back( 1 );

// utiliza o operador de subscrito para modificar elemento na localização 1
first[ 1 ] = 5;

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

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

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

return 0;
}
```



# Deque

UFOP

- 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

UFOP

```
#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 utilizados como iteradores

    set<int> third (second); //construtor cópia

    set<int> fourth (second.begin(), second.end()); // construtor iterador.

    //insere o elemento
    first.insert(10);
```



# set

UFOP

```
//remove o elemento
first.erase(40);

//localiza o elemento
set<int>::iterator it = first.find(40);

//troca os elementos dos conjuntos
first.swap(second);

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

return 0;
}
```



# multiset

UFOP

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

int main ()
{
    multiset<int> first; // multiset de inteiros vazio

    int myints[] = {10,20,30,20,20};
    multiset<int> second (myints,myints+5); // ponteiros utilizados como iteradores

    multiset<int> third (second); //construtor cópia

    multiset<int> fourth (second.begin(), second.end()); // construtor iterador.
```



# multiset

```
//insere no conjunto
first.insert(15);

//conta quantos elementos 15 existem no conjunto
cout<<first.count(15)<<endl;

//encontra a ocorrência do valor 15
multiset<int>::iterator result = first.find(15);

//retorna um iterador para o final se não achar
if(result == first.end())
    cout<<"não encontrou";
return 0;
}
```



# map

UFOP

```
#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

UFOP

```
//localiza a ocorrência do elemento
map<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*



UFOP

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



# multimap

UFOP

```
#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));
    first.insert(pair<char,int>('b',20));
    first.insert(pair<char,int>('c',25));

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



# multimap

UFOP

```
//remove o elemento
first.erase(it);

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

multimap<char,int> third (second); //construtor cópia

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

return 0;
}
```



# *multimap*

UFOP

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



# Contêineres Adaptativos

UFOP

- Os contêineres adaptativos (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

UFOP

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

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

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

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



# stack

UFOP

```
// 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

UFOP

```
#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 da fila
        values.pop(); // remove o elemento
    }

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



# priority\_queue

```
#include <iostream>
#include <queue> // definição do adaptador priority_queue
using 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

UFOP

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



UFOP



# Perguntas?



# Na próxima aula

- Processamento de Arquivos



UFOP



**FIM**