Standard Template Library (STL)

BCC 221 - Programação Orientada a Objectos(POO)

Guillermo Cámara-Chávez

Departamento de Computação - UFOP

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 de **contêineres**, **iteradores** e **algoritmos**

Introdução (cont.)

- **Contêineres** são templates de estruturas de dados

Introdução (cont.)

- **Contêineres** são templates de estruturas de dados
  - Possuem métodos associados a eles

Introdução (cont.)

- **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

Introdução (cont.)

- **Algoritmos** são as **funções** que realizam operações tais como **buscar, ordenar e comparar** elementos ou contêineres inteiros

Introdução (cont.)

- **Algoritmos** são as **funções** que realizam operações tais como **buscar, ordenar e comparar** elementos ou contêineres inteiros
  - Existem aproximadamente 85 algoritmos implementados na STL;

Introdução (cont.)

- **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

- Os padrões mais recentes da linguagem C++ sao: C++11, C++14 e C++17
- 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

- Os contêineres são divididos em três categorias principais:

Contêineres

- Os contêineres são divididos em três categorias principais:
  - Contêineres Sequenciais: estruturas de dados lineares.

Contêineres

- Os contêineres são divididos em três categorias principais:
  - Contêineres Sequenciais: estruturas de dados lineares.
  - Contêineres Associativos:

Contêineres

- Os contêineres são divididos em três categorias principais:
  - Contêineres Sequenciais: estruturas de dados lineares.
  - Contêineres Associativos:
    - Estruturas de dados não lineares

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

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.

Funções Membro Comuns STL

- Funções membro para todos os contêineres
  - Construtor padrão, construtor de cópia, destrutor
  - *empty*
  - *size*
  - `= < <= > >= == !=`
  - *swap*

Funções Membro Comuns STL (cont.)

- Funções para contêineres de primeira classe
  - *begin*, *end*
  - *rbegin*, *rend*
  - *erase*, *clear*
  - *max_size*

Iteradores

- Funcionalidade similar a dos ponteiros
  - Apontam para elementos em contêineres de primeira classe
  - Certas operações com iteradores são as mesmas para todos os contêineres
    - $*$ desreferencia
    - ++ aponta para o próximo elemento
    - *begin()* retorna o iterador do primeiro elementos
    - *end()* retorna iterador do elemento depois do último

Tipos de Iteradores Suportados

- Contêineres sequencias
  - vector: acesso aleatório
  - deque: acesso aleatório
  - list: bidireccional
- Contêineres associativos (todos bidireccionais)
  - set
  - multiset
  - map
  - multimap

Tipos de Iteradores Suportados (cont.)

- Contêineres adaptados (não há suporte a iteradores)
  - stack
  - queue
  - priority_queue

Contêineres Sequenciais

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

Vector

- Definido na biblioteca #include<vector>
- Estrutura de dados com alocação de memória sequencial
- Mais eficientes se inserções forem feitas apenas no final
- Usado quando os dados devem ser ordenados e facilmente acessível

Vector (cont.)

- Quando memória estiver esgotada
  - Aloca maior área sequencial de memória
  - Copia ele mesmo lá
  - Desaloca memória antiga
- Tem iteradores de acesso aleatório

Vector (cont.)

- Declaração
  - std :: vector<type> v;
- Iteradores
  - std :: vector<type>:: const_iterator iterVar ;
    - const_iterator não pode modificar elementos
  - std :: vector<type>:: iterator iterVar;
    - Visita elementos do inicio ao fim
    - Usa *begin* para receber ponto de início
    - Usa *end* para receber ponto final

Vector (cont.)

- std :: vector<type>:: reverse_iterator iterVar;
  - Visita elementos na ordem reversa (fim para o início)
  - Usa *rbegin* para receber ponto de início na ordem reversa
  - Usa *rend* para receber ponto final na ordem reversa

Vector (cont.)

- Funções
  - v.push_back(value)
    - Adiciona elemento ao final (encontrado em todos os contêineres sequenciais)
  - v.size()
    - Número de elementos no vector
  - v.capacity()
    - Quanto o vector pode inserir antes de realocar memória

Vector (cont.)

- vector<type> v(a, a + SIZE)
  - Cria vector *v* com elementos do array *a* até o (não incluindo) *a* + *SIZE*
- v. insert (iterator , value)
  - Insere *value* antes do posicionamento do *iterator*
- v. insert (iterator , array , array + SIZE)
  - Insere elementos do *array* (até, mas não incluindo *array* + *SIZE*) antes do posicionamento do *iterator*

Vector (cont.)

- v.erase(iterator)
  - Remove elemento do contêiner
- v.erase(iter1 , iter2)
  - Remove elementos começando do *iter1* e até (não incluindo) *iter2*
- v.clear ()
  - Apaga todo o contêiner

Vector (cont.)

- Operações de funções
  - v.front(), v.back()
    - Retorna uma referência para o primeiro e último elemento no contêiner, respectivamente
  - v[elementNumber] = value;
    - Atribui *value* a um elemento
  - v.at(elementNumber)= value;
    - Como acima, com checagem de intervalo

Vector (cont.)

```
#include <vector>
...
vector<int> vec;  // vec.size() == 0
vec.push_back(4);
vec.push_back(1);
vec.push_back(8);  // vec:4,1,8 vec.size() == 3

//operaçoes com vector
cout << vec[2];  // 8 (não verifica se pos. válida)
cout << vec.at(2);  // 8 (lança exceção se pos. inválida)
...
```

Vector (cont.)

```
// mostrando o vector
for (int i = 0; i < vec.size(); i++)
    cout << vec[i] << " ";

for (vector<int>::iterator itr = vec.begin(); itr !=
    vec.end(); ++itr)
    cout << *itr << " ";

for (auto itr = vec.begin(); itr != vec.end(); ++itr)
    cout << *itr << " ";

for (int it : vec)
    cout << it << " ";
...
```

Vector (cont.)

```
...
vector<double> V;
double value = 1.1;
for (int i = 0; i < 5; i++){
    V.push_back(value);
    value += 1.1;
}
cout << "\nBackward: ";
for (vector<double>::reverse_iterator itr = V.rbegin();
     itr != V.rend(); ++itr)
    cout << *itr << " ";

cout << "\nAuto Backward: ";
for (auto itr = V.rbegin(); itr != V.rend(); ++itr)
    cout << *itr << " ";

...
```

Vector (cont.)

```
// Vector é um arranjo sequencial alocado dinamicamente
int *p = &vec[0];

// funções membros comuns entre os contêineres
if ( vec.empty() )
    cout << "Vazio";
cout << vec.size(); // 3
vector<int> vec2(vec);
vec.clear(); // remove todos os elementos; vec.size() == 0
vec2.swap(vec); // troca os dados entre vec e vec2
```

Vector (cont.)

Propriedades da classe vector

- Rápida inserção/remoção de dados no final do vetor: O(1)
- Lenta inserção/remoção de dados no inicio ou no meio: O(n)
- Pesquisa lenta: O(n)

Contêiner Sequencial *list*

- Cabeçalho: #include<list>
- Inserção/remoção eficiente em qualquer lugar no contêiner
- Lista duplamente encadeada (dois ponteiros por nó)
  - Um para elemento anterior outro para elemento posterior
- Iteradores bidirecionais
- Declaração: std :: list <type> name;

Contêiner Sequencial *list* (cont.)

- Funções *list* para o objeto *L*
  - L. sort ()
    - Ordena em ordem crescente
  - L. splice (iterator , otherObject);
    - Insere valores de *otherObject* em *L* antes do iterador e remove os objetos de *otherObject*
  - L.merge(otherObject)
    - Remove *otherObject* e o insere em L, ordenado
  - L.unique()
    - Remove elementos duplicados

Contêiner Sequencial *list* (cont.)

- *L.swap(otherObject);*
  - Troca conteúdo de l com o de otherObject
- *L.assign(iterator1, iterator2);*
  - Substitui conteúdo com elementos no intervalo dos iteradores
- *L.remove(value);*
  - Apaga todas as instâncias de *value*

Contêiner Sequencial *list* (cont.)

```
#include<list>
#include<iterator>

using namespace std;

template<typename T>
void printList(list<T> & L){
    for (auto elem : L)
        cout << elem << " ";
    cout << endl;
}

template<typename T>
void reverse_printList(const list<T> & L){
    for (list<T>::reverse_iterator it = L.rbegin(); it
        != L.rend(); it++)
        cout << *it << " ";
    cout << endl;
}
```

Contêiner Sequencial *list* (cont.)

```
int main()
{
   const int SIZE = 4;
   int array[ SIZE ] = { 2, 6, 4, 8 };
   list< int > values; // cria lista de ints
   list< int > otherValues; // cria lista de ints

   // insere itens em values
   values.push_front( 1 );
   values.push_front( 2 );
   values.push_back( 4 );
   values.push_back( 3 );
   cout << "values contains: ";
   printList( values ); // 2, 1, 4, 3
```

Contêiner Sequencial *list* (cont.)

```
values.sort(); // ordena values
cout << "\nvalues after sorting contains: ";
printList( values ); // 1,2,3,4

// insere elementos do array em otherValues
otherValues.insert( otherValues.begin(), array,
    array + SIZE );
cout << "\nAfter insert, otherValues contains: ";
printList( otherValues ); //2,6,4,8

// remove elementos otherValues e insere no fim de values
values.splice( values.end(), otherValues );
cout << "\nAfter splice, values contains: ";
printList( values ); //1,2,3,4,2,6,4,8

values.sort(); // classifica values
cout << "\nAfter sort, values contains: ";
printList( values ); // 1,2,2,3,4,4,6,8
```

Contêiner Sequencial *list* (cont.)

```
// insere elementos do array em otherValues
otherValues.insert( otherValues.begin(), array,
    array + SIZE );
otherValues.sort();
cout << "\nAfter insert, otherValues contains: ";
printList( otherValues ); // 2,4,6,8

// remove elementos otherValues e insere em values na ordem classificada
values.merge( otherValues );
cout << "\nAfter merge:\n   values contains: ";
printList( values ); //1,2,2,2,3,4,4,4,6,6,8,8
cout << "\n   otherValues contains: ";
printList( otherValues ); // Lista vazia

values.pop_front(); // remove elemento da parte da frente
values.pop_back(); // remove elemento da parte de trás
cout << "\nAfter pop_front and pop_back:\n   values
    contains: ";
printList( values ); // 2,2,2,3,4,4,4,6,6,8
```

Contêiner Sequencial *list* (cont.)

```
values.unique(); // remove elementos duplicados
cout << "\nAfter unique, values contains: ";
printList( values ); //2,3,4,6,8

// permuta elementos de values e otherValues
values.swap( otherValues );
cout << "\nAfter swap:\n   values contains: ";
printList( values ); // Lista vazia
cout << "\n   otherValues contains: ";
printList( otherValues ); // 2,3,4,6,8

// substitui conteúdo de values por elementos otherValues
values.assign( otherValues.begin(), otherValues.end
    () );
cout << "\nAfter assign, values contains: ";
printList( values ); //2,3,4,6,8
```

Contêiner Sequencial *list* (cont.)

```
// remove elementos otherValues e insere em values na ordem classificada
values.merge( otherValues );
cout << "\nAfter merge, values contains: ";
printList( values ); //2,2,3,3,4,4,6,6,8,8

values.remove( 4 ); // remove todos os 4s
cout << "\nAfter remove( 4 ), values contains: ";
printList( values ); //2,2,3,3,6,6,8,8
cout << endl;
return 0;
```

Contêiner Sequencial *deque*

- **deque** ("deek"): fila com final duplo (*double-ended queue*)
  - Header #include<deque>
  - Acesso indexado usando []
  - Inserção/remoção eficiente na frente e no final

Contêiner Sequencial *deque* (cont.)

- Mesmas operações básicas como vector
  - Entretanto, também possui como nas listas
    - *push_front* (insere na frente do deque)
    - *pop_front* (remove da frente)

Contêiner Sequencial *deque* (cont.)

```
deque<int> dew = {4, 6, 7};
deq.push_front(2);  // deq: 2,4,6,7
deq.push_back(3);  // deq: 2,4,6,7,3

cout << deq[1];   // 4
```

Contêiner Sequencial *deque* (cont.)

```
#include <iostream>
#include <deque>
#include <algorithm>
#include <iterator>

int main()
{
    std::deque< double > values; // cria deque de doubles
    std::ostream_iterator< double > output( cout, " " );

    // insere elementos em values
    values.push_front( 2.2 );
    values.push_front( 3.5 );
    values.push_back( 1.1 );
```

Contêiner Sequencial *deque* (cont.)

```
   cout << "values contains: ";

   // utiliza o operador de subscrito para obter elementos de values
   for ( unsigned int i = 0; i < values.size(); i++ )
      cout << values[ i ] << ' ';

   values.pop_front(); // remove o primeiro elemento
   cout << "\nAfter pop_front, values contains: ";
   std::copy( values.begin(), values.end(), output );

   // utiliza o operador de subscrito para modificar elemento na localização 1
   values[ 1 ] = 5.4;
   cout << "\nAfter values[ 1 ] = 5.4, values contains:
        ";
   std::copy( values.begin(), values.end(), output );
   cout << endl;
   return 0;
} // fim de main
```

Contêineres Adaptados

- *stack*, *queue* e *priority_queue*
- Não são contêineres de primeira classe
- Não suportam iteradores
- Programador pode selecionar implementação
- Funções membro *push* e *pop*

Adaptador *Stack*

- Cabeçalho #include <stack>
- Inserções e remoções em uma extremidade
- Estrutura de dados last-in, *first-out* (LIFO)
- Pode usar *vector*, *list*, ou *deque* (padrão)

Adaptador *Stack* (cont.)

- Declarações

```
stack<type, vector<type> > myStack;
stack<type, list<type> > myOtherStack;
stack<type> anotherStack; // padrão deque
```

- *vector*, *list*
  - Implementação de stack (padrão deque)
  - Não muda comportamento, apenas desempenho (*deque* e *vector* mais rápidos)

Adaptador *Stack* (cont.)

```
#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;

// pushElements function-template prototype
template< typename T > void pushElements(T &stackRef);
// protótipo de template de função popElements
template< typename T > void popElements(T &stackRef);

int main()
{
    // pilha com deque subjacente padrão
    std::stack< int > intDequeStack;
    // pilha com vetor subjacente
    std::stack< int, vector< int > > intVectorStack;
    // pilha com lista subjacente
    std::stack< int, list< int > > intListStack;
```

Adaptador *Stack* (cont.)

```
    // insere os valores 0-9 em cada pilha
    cout << "Pushing onto intDequeStack: ";
    pushElements(intDequeStack);
    cout << "\nPushing onto intVectorStack: ";
    pushElements(intVectorStack);
    cout << "\nPushing onto intListStack: ";
    pushElements(intListStack);
    cout << endl << endl;

    // exibe e remove elementos de cada pilha
    cout << "Popping from intDequeStack: ";
    popElements(intDequeStack);
    cout << "\nPopping from intVectorStack: ";
    popElements(intVectorStack);
    cout << "\nPopping from intListStack: ";
    popElements(intListStack);
    cout << endl;
    return 0;
} // fim de main
```

Adaptador *Stack* (cont.)

```
// insere elementos no objeto stack que stackRef referencia
template< typename T > void pushElements(T &stackRef)
{
    for (int i = 0; i < 10; i++)
    {
        stackRef.push(i); // insere o elemento na pilha
        cout << stackRef.top() << ' '; // visualiza (e exibe)
            elemento superior
    } // fim do for
} // fim da função pushElements
// remove elementos do objeto stack que stackRef referencia
template< typename T > void popElements(T &stackRef)
{
    while (!stackRef.empty())
    {
        cout << stackRef.top() << ' '; // visualiza (e exibe)
            elemento superior
        stackRef.pop(); // remove elemento superior
    } // fim do while
} // fim da função popElements
```

Adaptador *Stack* (cont.)

```
Pushing onto intDequeStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto intVectorStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto intListStack: 0 1 2 3 4 5 6 7 8 9

Popping from intDequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from intVectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from intListStack: 9 8 7 6 5 4 3 2 1 0
```

Adaptador *queue*

- Cabeçalho #include<queue>
- Inserções no final, remoções na frente
- Estrutura de dados *first-in-first-out* (FIFO)
- Implementada com list ou deque (padrão)

```
std::queue<double> values;
```

Adaptador *queue* (cont.)

- Funções
  - push(elemento)
    - Mesmo que *push_back*, adicionar no final
  - pop(element)
    - Implementado com *pop_front*, remove da frente
  - *empty()* e *size()*

Adaptador *queue* (cont.)

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

int main(){
    std::queue< double > values; // fila com doubles
    // insere elementos nos valores de fila
    values.push(3.2);
    values.push(9.8);
    values.push(5.4);
    cout << "Popping from values: ";
    // remove elementos da fila
    while (!values.empty()){
        cout << values.front() << ' '; // visualiza elemento
            da frente da fila
        values.pop(); // remove o elemento
    } // fim do while
    return 0;
} // fim de main
```

Adaptador *priority_queue*

- Cabecalho #include<queue>
- Inserções acontecem ordenadas, remoções da frente
- Implementada com *vector* (padrão) ou *deque*
- Elemento de prioridade mais alta é sempre removido primeiro
  - Algoritmo heapsort coloca elementos maiores na frente
  - less <T> padrão, programador especifica outro comparador

Adaptador *priority_queue* (cont.)

- Funções
  - *push(value)*, *pop(value)*
  - *top()*: vê elemento do topo
  - *size()* e *empty()*

Adaptador *priority_queue* (cont.)

- Diferente dos anteriores, a classe *template priority_queue* possui três parâmetros:
  - Tipo dos elementos
  - Contêiner
  - Classe de comparação: Pode ser uma classe implementando uma função ou um ponteiro para uma função

Adaptador *priority_queue* (cont.)

```
#include <iostream>
#include <queue> // definição do adaptador priority_queue
using namespace std;
int main() {
        std::priority_queue< double > priorities; // cria
            priority_queue

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

        cout << "Popping from priorities: ";
        // remove elemento de priority_queue
        while (!priorities.empty()) {
                cout << priorities.top() << ' '; //
                    visualiza elemento superior
                priorities.pop(); // remove elemento superior
        } // fim do while
        return 0;
} // fim de main
```

Exemplo com função de comparação

```
#include <iostream>
#include <queue>
#include <cstdlib> // rand()
#include <functional> // function
using namespace std;

template<typename T>
void pushElements(T& fila){
    cout << endl;
    int num;
    for (int i = 0; i < 10; i++){
        num = rand();
        fila.push(num);
        cout << num << " ";
    }
}
```

Exemplo com função de comparação (cont.)

```
template<typename T>
void popElements(T& fila){
    cout << endl;
    while (!fila.empty()){
        cout << fila.top() << " ";
        fila.pop();
    }
}

class compare{
public:
    bool operator()(int a, int b){
        return a < b;
    };
};

bool compareFunc(int a, int b){
    return a < b;
};
```

Exemplo com função de comparação (cont.)

```
int main()
{
    // forma 1
    priority_queue<int, deque<int>, compare>
        p_queueDeque1;

    // forma 2
    priority_queue<int, deque<int>, std::function<bool(
        int,int)>> p_queueDeque2(compareFunc);

    // forma 3
    auto cmp = [](int a, int b){return a < b;};
    priority_queue<int, deque<int>, decltype(cmp)>
        p_queueDeque3(cmp);

    // forma 4
    priority_queue<int, deque<int>, function<bool(int a
        , int b)>> p_queueDeque4([](int a, int b){
        return a < b;});
```

Exemplo com função de comparação (cont.)

```
    cout << "\n Pushing \n";
    pushElements(p_queueDeque1);
    pushElements(p_queueDeque2);
    pushElements(p_queueDeque3);
    pushElements(p_queueDeque4);

    cout << "\n Poping \n";
    popElements(p_queueDeque1);
    popElements(p_queueDeque2);
    popElements(p_queueDeque3);
    popElements(p_queueDeque4);
    return 0;
}
```

Smart Pointers - Ponteiros Inteligentes

- Em C++, temos operadores *new* e *delete*
- Criação manual e liberação manual, risco de vazamento
- Uso padrão dos ponteiros

```
MyClass *ptr = new MyClass();
ptr->Function();
delete ptr;
```

Smart Pointers - Ponteiros Inteligentes (cont.)

- Os ponteiros inteligêntes podem manipular três aspectos sobre o comportamento dos ponteiros
  - Construção
  - Desreferênciação
  - Destruição
- O vazamento da memoria pode acontecer em diferentes casos

```
MyClass *ptr = new MyClass();
ptr->Function(); // pode lancar uma excecao
delete ptr; // esquecer de liberar memoria
```

Smart Pointers - Ponteiros Inteligentes (cont.)

- Implementos nossa propria classe de ponteiros inteligentes

```
template<typename T>
class SmartPointer{
    T* ptr;
public:
    explicit SmartPointer(T* ptr = nullptr):ptr(ptr){}
    virtual ~SmartPointer(){delete ptr;}
    T* operator->(){return ptr;}
    T& operator*(){return *ptr;}
};
```

Smart Pointers - Ponteiros Inteligentes (cont.)

```
class MyClass{
public:
    MyClass(){cout << "\nConstrutor";}
    ~MyClass(){cout << "\nDestrutor";}
};

int main()
{
    {
        SmartPointer<MyClass> p(new MyClass);
    }
    return 0;
}
```

Smart Pointers C++ - unique_ptr

```
#include<memory>
int main()
{
    {
        unique_ptr<MyClass> ptr(new MyClass);
        unique_ptr<MyClass> ptr2;
        ptr2 = make_unique<MyClass>();
        unique_ptr<MyClass> ptr3 = make_unique<MyClass
            >();
        ptr->print();
        ptr2->print();
        ptr3->print();
    }
    return 0;
}
```

Smart Pointers C++ - unique_ptr (cont.)

```
class MyClass{
      int n;
public:
     MyClass(int n = 0) : n(n){cout << "\nConstrutor";}
     ~MyClass(){cout << "\nDestrutor";}
     void set(int n){this->n = n;}
     void print(){
          cout << endl << n << "executando ...";
     }
};

void foo(unique_ptr<MyClass> ptr){
     cout << "\n Funcao: ";
     ptr->set(10);
     ptr->print();
     // ptr eh destruido
}
```

Smart Pointers C++ - unique_ptr (cont.)

```
int main()
{
    {
        //SmartPointer¡MyClass¿ p(new MyClass);
        unique_ptr<MyClass> ptr(new MyClass);
        foo(move(ptr));
        ptr->print(); // gera erro
    }
    return 0;
}
```

Smart Pointers C++ - shared_ptr

```
#include<memory>
class MyClass;
void someFn(shared_ptr<MyClass> arg){
    // someFn also has ownership of
    // the MyClass (references = 2)

    //at the end of someFn scope
    //references to Foo -= 1
}
int main(){
    shared_ptr<MyClass> f = make_shared<MyClass>();

    // references to that MyClass = 1

    someFn(f);

    //only at the end of main's scope
    return 0;
    // is the MyClass destroyed (refs = 0)
}
```

Smart Pointers C++ - Polimorfismo

```
class Base{
public:
    Base(){}
    virtual ~Base(){}
    virtual void print(){cout << "\n Base";}
};

class Derivada : public Base{
public:
    Derivada(){}
    virtual ~Derivada(){}
    virtual void print(){cout << "\n Derivada";}
    void privado(){cout << "\n Privado";}
};
```

Smart Pointers C++ - Polimorfismo (cont.)

```
int main()
{
    shared_ptr<Base> ptr1(new Base);
    shared_ptr<Base> ptr2(new Derivada);

    ptr1->print();
    ptr2->print();

    shared_ptr<Derivada> ptr3 = dynamic_pointer_cast<
        Derivada>(ptr2);
    if (ptr3 != nullptr)
        ptr3->privado();
    return 0;
}
```

FIM