IMD0030 – LINGUAGEM DE PROGRAMAÇÃO I

Aula 14 – Ponteiros Inteligentes.

Conteúdo baseado nos slides do Prof. Dr. Ivan Luiz Marques Ricarte DCA - FEEC – UNICAMP.

IMD INSTITUTO METRÓPOLE DIGITAL
UFRN UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

Objetivo

- Introduzir os conceitos de ponteiros inteligentes em **C++11/C++14**
- Para isso, estudaremos:
  - unique_ptr
  - shared_ptr
  - weak_ptr
- Ao final da aula espera-se que o aluno seja capaz de:
  - Distinguir a utilização de ponteiros traicionais e ponteiros inteligentes
  - Desenvolver programas com o uso de ponteiros inteligentes

Temos que conversar...

C++ Smart Pointers

Ponteiros inteligentes em C++

In brief, smart pointers are C++ objects that simulate simple pointers by implementing operator-> and the unary operator*. In addition to sporting pointer syntax and semantics, smart pointers often perform useful tasks—such as memory management or locking—under the covers, thus freeing the application from carefully managing the lifetime of pointed-to objects.

Andrei Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, 2001.

Estrutura mínima de um ponteiro inteligente

```
template <typename T>
class PonteiroInteligente {
public:
    PonteiroInteligente (T* _ponteiro): ponteiro(_ponteiro);
    ~PonteiroInteligente () {
        delete ponteiro;
        std::cout << "Ponteiro liberado." << std::endl;
    };
    T* operator->() const { return ponteiro; };
    T& operator*() const { return *ponteiro; };
private:
    T* ponteiro;
}
```

Usando o PonteiroInteligente

```
#include "ponteirointeligente.h"

int main(int argc, char const *argv[])
{
    PonteiroInteligente<int> ptr(new int(80));
    std::cout << (*ptr) << std::endl;
    return 0;
}
```

Aloca memória e cria o ponteiro.

Faz uso do ponteiro e da área de memória alocada.

Libera a memória apontada.

Ponteiros Inteligentes no C++

- Recurso incorporado no C++11
- Classes parametrizadas (definidas na biblioteca **memory**) para ponteiros inteligentes:
  - unique_ptr
  - shared_ptr
  - weak_ptr
- Seleção da classe de ponteiro reflete a intenção do programador em seu uso
  - Ao contrário do que ocorre com ponteiro tradicional
- Tornam programação mais simples e código mais robusto

O ponteiro **unique_ptr**

- Existe uma única referência para o objeto apontado
- Ponteiro não pode ser copiado
  - Não pode ser atribuído a outro ponteiro diretamente, armazenado em um container ou passado como argumento para uma função
- Posse do ponteiro pode ser transferida
  - Conteúdo é movido e área anterior passa a ser inválida
    - Mover é sempre mais eficiente que copiar
- Utiliza novos recursos introduzidos na linguagem C++
  - Referência rvalor e a semântica de mover
- Substitui **auto_ptr**

Referência rvalor

- lvalor é algo do qual se pode obter o endereço
- rvalor não se pode obter o endereço
  - Tipicamente, variável temporária numa expressão ou um valor de retorno de um método ou função
  - Sendo temporária, não precisa ser mantida após utilizada – mover seu conteúdo é mais eficiente que copiá-lo
- Sintaxe para referências em C++11:
  - T&: referência lvalor
  - T&&: referência rvalor

Cópia e atribuição de **unique_ptr**

- Na especificação da classe, define construtor para rvalor

  public:
  - MeuPtr (const MeuPtr&&);
  - MeuPtr& operator=(MeuPtr&&);

- Desabilita o acesso público dos mecanismos de construção e cópia com lvalor

  private:
  - MeuPtr (const MeuPtr&) = delete;
  - MeuPtr const operator= (const MeuPtr&) = delete;

- Para transferir a posse do ponteiro explicitamente, introduz função **move()**
- **std::make_unique()** permite criar um ponteiro inteligente **unique_ptr** (C++14)

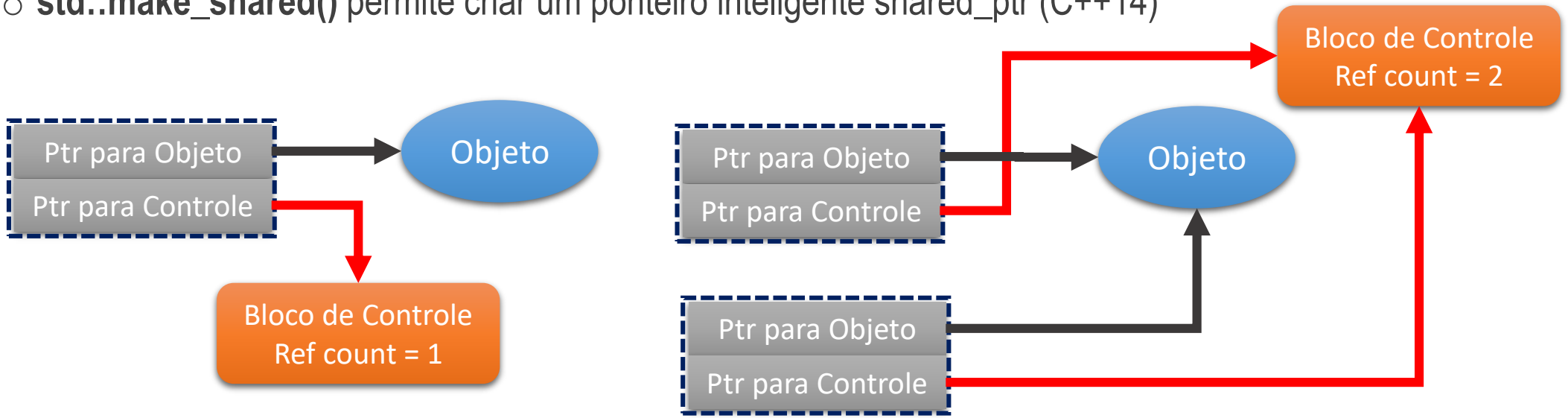
Exemplo: **unique_ptr**

```cpp
#include <iostream>
#include <memory>

int main(int argc, char const *argv[])
{
    std::unique_ptr<int> ptr1(new int);
    std::unique_ptr<int> ptr2(nullptr);
    *ptr1 = 25;
    std::cout << (*ptr1) << std::endl;
    // ptr2 = ptr1; // Causaria erro!
    ptr2 = std::move(ptr1); // É preciso transferir a posse
    //std::cout << (*ptr1) << std::endl; // Causaria erro!
    std::cout << (*ptr2) << std::endl;
    return 0;
}
```

O ponteiro **shared_ptr**

- Ponteiro para um recurso que pode ser compartilhado
  - Com controle do número de referências
  - Somente quando última referência deixa de existir, o recurso é liberado
  - Ocupa o dobro de um ponteiro tradicional
  - **std::make_shared()** permite criar um ponteiro inteligente shared_ptr (C++14)



Exemplo: **shared_ptr**

```cpp
#include <iostream>
#include <memory>

void imprime(std::shared_ptr<int> valor) {
    std::cout << "Valor recebido: " << (*valor) << std::endl;
}

int main(int argc, char const *argv[])
{
    auto p = new int; // Ponteiro tradicional
    *p = 33;
    std::shared_ptr<int> ptr1 (p);
    imprime(ptr1);
    std::cout << (*ptr1) << std::endl;
    return 0;
}
```

Risco em compartilhar referências

- **Referências cíclicas**
  - Um objeto mantém referências circulares entre objetos
- **Solução:** uso do **weak_ptr**
  - Ponteiro inteligente para uso em conjunto com **shared_ptr**
  - Um **weak_ptr** fornece acesso a um objeto que pertence a um ou mais instâncias de **shared_ptr**, mas não participa de contagem de referência
  - Use quando você deseja observar um objeto, mas não precisam permanecer ativo
  - Necessária em alguns casos para quebrar referências circulares entre instâncias **shared_ptr**

Prefira o **std::make_unique** e **std::make_shared**

- Prefira usar o **std::make_unique** e **std::make_shared** como substituto ao uso direto do **new**
- Recurso do C++14
  - **std::make_unique** constrói um **std::unique_ptr** do ponteiro cru que o comando **new** produz
- Recurso do C++11
  - **std::make_shared** constrói um **std::shared_ptr** do ponteiro cru que o comando **new** produz

Exemplo: **std::make_unique**

```cpp
#include <iostream>
#include <memory>

int main(int argc, char const *argv[])
{
    std::unique_ptr<int> ptr1 = std::make_unique<int>(25);
    std::unique_ptr<int> ptr2(nullptr);
    std::cout << (*ptr1) << std::endl;
    // ptr2 = ptr1; // Causaria erro!
    ptr2 = std::move(ptr1); // É preciso transferir a posse
    //std::cout << (*ptr1) << std::endl; // Causaria erro!
    std::cout << (*ptr2) << std::endl;
    return 0;
}
```

Exemplo: **std::make_shared**

```
#include <iostream>
#include <memory>

void imprime(std::shared_ptr<int> valor) {
    std::cout << "Valor recebido: " << (*valor) << std::endl;
}

int main(int argc, char const *argv[])
{
    std::shared_ptr<int> ptr1 = std::make_shared<int>(33);
    imprime(ptr1);
    std::cout << (*ptr1) << std::endl;
    return 0;
}
```

Novo paradigma: ponteiros inteligentes

- Quando usar ponteiros tradicionais em C++?
  - Praticamente **NUNCA**
- Quando usar os ponteiros inteligentes em C++?
  - Apenas quando a semântica de ponteiros for necessária
    - Quando um objeto precisa ser compartilhado
    - Quando é necessário fazer uma referência polimórfica
  - Para as demais situações, usar as classes da biblioteca padrão de C++ (STL)

Alguma Questão?