

Programação Orientada a Objetos em C++

Smart Pointers

Agostinho Brito

2021

- 1 **Ponteiros brutos e suas limitações na gestão de memória**
- 2 **Ponteiros inteligentes**
- 3 **Smart Pointers em C++**

Ponteiros brutos e suas limitações na gestão de memória

Ponteiros brutos e suas limitações gestão de memória

- A alocação dinâmica de memória em C++ feita com `new` e `new []` requer, normalmente, um olho bem treinado do programador.

```
1  class Vetor2d{  
2      float x, y;  
3  };  
4  
5  Vetor2d *v1;  
6  v1 = new Vetor2d();
```

- Há uma regra de ouro do utilizador de ponteiros brutos para alocação dinâmica que deve ser respeitada: **para cada alocação deve haver uma liberação correspondente.**

```
1  v1 = new Vetor2d();  
2  ...  
3  delete v1;
```

- Caso a regra seja quebrada, ocorre o chamado "vazamento de memória".

Ponteiros brutos e suas limitações na gestão de memória

- Ponteiros brutos são rápidos, criados para eficiência.
- Entretanto, quando o processo de gestão de memória é algo crucial no programa, eles podem ser o calcanhar de aquiles do desenvolvedor.
- Exemplos:

!

Programas que lidam com grandes quantidades de dados alocados dinamicamente.

!

Estratégias que repassam dados alocados entre funções que, em virtude de algum evento, decidirão a hora adequada de por fim ao ciclo de vida deste dado.

!

Gestão de memória não é o objetivo do projeto, mas atrapalha o foco do programador

Ponteiros inteligentes

Um pequeno truque...

```
class PonteiroSabido {  
    int *p;  
public:  
    explicit PonteiroSabido(int *p_ = nullptr) { p = p_; }  
    ~PonteiroSabido() {  
        delete p;  
    }  
    int& operator*() { return *p; }  
};  
  
int main(void) {  
    PonteiroSabido pont(new int);  
    *pont = 32;  
    std::cout << "*pont = " << *pont << "\n";  
}
```

- A alocação é feita, mas o destrutor se responsabiliza pela liberação!

Smart Pointers em C++

A biblioteca padrão do C++ provê três classes para uso de *smart pointers*, que funcionam usando gabaritos, podendo ser adaptadas para qualquer tipo de dados.

- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`

Essas três classes funcionam como verdadeiros coletores de lixo, algo padrão em outras linguagens, mas não em C++.

!

Smart Pointers normalmente devem ser preferidos, pois poupam aborrecimentos de vazamento de memória.

- `std::unique_ptr` provê um coletor de lixo simples com baixa sobrecarga para uma variável que é fornecida.
- Ele tem a capacidade de se apropriar da variável, ficando responsável por liberá-la através do seu destrutor e nenhum outro recurso deve cuidar do processo desse processo de liberação.
- Apenas o objeto criado pode apontar para o dado alocado, não sendo permitido a outros objetos semelhantes compartilharem por **cópia** o endereço repassado no construtor.
- A liberação da memória pode se dar também durante uma operação de atribuição ou pelo chamado de `unique_ptr::reset`.

```
std::unique_ptr<int>      x(new int);  
std::unique_ptr<int[]>   y(new int[20]);  
std::unique_ptr<Foo>     z(new Foo());
```

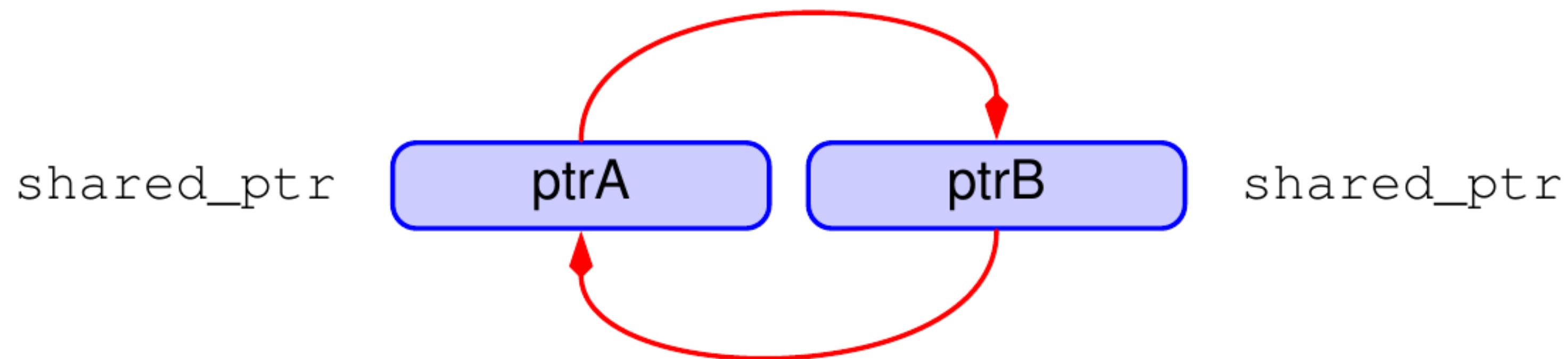
Praticando std::unique_ptr...

- Um objeto da classe `std::shared_ptr` é capaz de compartilhar a propriedade de um objeto alocado.
- Diversos `std::shared_ptr` podem apontar para o mesmo objeto.
- É mantido um contador no compartilhamento e quando o último objeto `std::shared_ptr` perde seu escopo, a memória do objeto gerenciado é liberada.
- Podem ser resetados, se necessário.


```
std::shared_ptr<Foo> x(new Foo());  
std::shared_ptr<Foo> y;  
y = x;
```

Praticando std::shared_ptr...

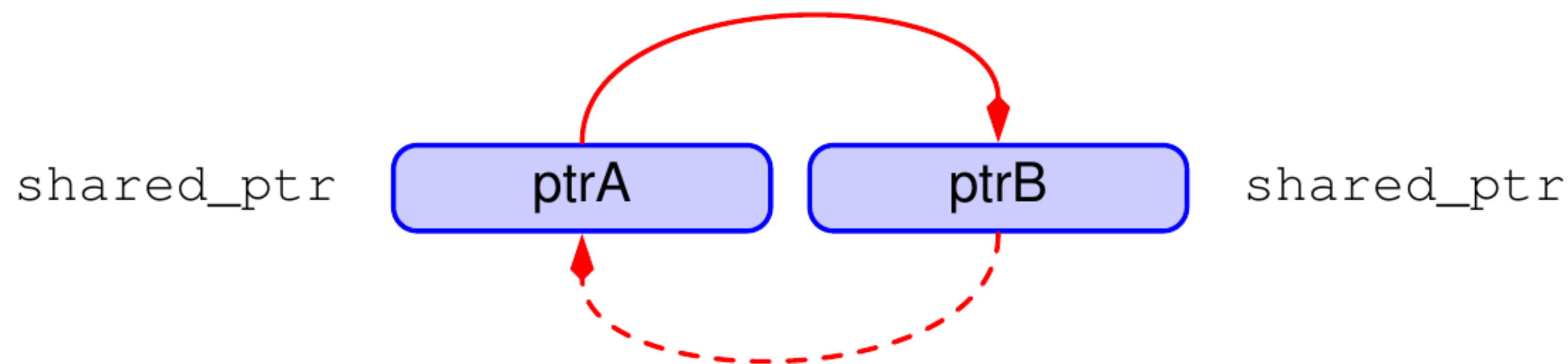
- A dependência circular ocorre quando dois ponteiros apontam um para o outro.



- Quando o destrutor de ptrA procura realizar a limpeza, ele descobre que ptrB aponta para ptrA e para.
- O mesmo ocorre para o destrutor de ptrB, de sorte que `use_count` nunca chega a zero e eles nunca são deletados.

 Explorando a dependência
cíclica...

- `std::unique_ptr` possibilita único proprietário.
- `std::shared_ptr` possibilita vários proprietários.
- `std::weak_ptr` não possui proprietários. Trata-se apenas de uma referência para um objeto gerenciado por `std::shared_ptr`, não contribuindo para o contador de referências.
- É criado como uma cópia de `std::shared_ptr` para evitar problemas de dependência cíclica entre objetos `std::shared_ptr`.



- Se o objeto precisar ser deletado, o `std::weak_ptr` não se incomoda com isso, nem interfere no processo.

 Explorando `weak_ptr`...



Obrigado