BENEFÍCIO E APLICAÇÃO DE *SMART POINTERS*

Matheus Percário Bruder, Mariana Ramos dos Santos, José Gabriel Alves, Gabriel Gomes Gonçalves

RESUMO

A pesquisa foi executada a fim de demonstrar de maneira lógica e estruturada os *Smart Pointers* relacionado ao padrão C++11 da linguagem de programação C++. O modelo será baseado nos ponteiros inteligentes *unique_ptr* e *shared_ptr*, por meio de códigos fonte e explicação conceitual. Os benefícios resultantes da utilização dos *Smart Pointers* estão relacionados aos aspectos práticos da programação, como por exemplo, a redução das responsabilidades do programador em relação a liberação de memória após o uso.

INTRODUÇAO

No âmbito da programação, uma variável pode conter um valor ou um endereço de memória, assim sendo, uma variável que contém um endereço é denominada ponteiro [1]. Logo, se uma variável possui o endereço de outra variável, diz-se que a primeira variável aponta para a segunda [1].

A utilização dos ponteiros é imprescindível para um bom desenvolvedor de *software* e o emprego desse artificio traz consigo diversos benefícios. A primeira vantagem é que os ponteiros fornecem meios pelos quais as funções podem modificar seus argumentos, além disso, podem aumentar a eficiência de rotinas e, por fim, fornecerão suporte para alocação dinâmica [2], portanto, o uso dos ponteiros torna possível alocar memória para o programa a tempo de execução. Existem diversos benefícios na utilização dos ponteiros, também há problemas relacionados a sua má utilização. Toda memória alocada dinamicamente, a tempo de execução, deve ser liberada após o uso para que não ocorra o risco do estouro de pilha, devido ao excesso de lixo produzido. Dessa forma, o programador é responsável por alocar e desalocar a memória, porém, por um lapso em algum momento o programador não liberará essa memória e certamente ocorrerá problemas no *software*.

A partir desse cenário é que surgem os *Smart Pointers* ou ponteiro inteligentes, que são responsáveis por desalocar a memória reservada para o ponteiro imediatamente após o termino de sua utilização, ou seja, são uma espécie de coletor de lixo [3]. Dessa maneira, os ponteiros inteligentes evitam os diversos problemas relacionados à não liberação da memória [3]. Para a pesquisa, embora existam vários tipos, serão utilizados dois ponteiros inteligentes, *unique_ptr* e *shared_ptr*, os quais exemplificarão a pesquisa sobre *Smart Pointers* de forma prática e conceitual.

O *unique_ptr* é o ponteiro inteligente usado com a maior frequência, visto que, esse ponteiro permite a atribuição de somente um objeto por vez, ou seja, possui apenas um proprietário. Dessa forma, o ponteiro *unique_ptr* pode apenas ser movido para um outro ponteiro do mesmo tipo, contudo, jamais pode ser copiado para outro ponteiro, sendo do mesmo tipo ou não [3]. Após ser movido, o ponteiro original é redefinido e recebe *NULL*. O *shared_ptr* é o ponteiro inteligente utilizado com uma frequência menor, dado que, sua complexidade é maior quando comparado ao *unique_ptr*. Esse ponteiro tem uma complexidade um pouco maior que o anterior, pois, permite o compartilhamento de objeto entre os ponteiros. Diversos ponteiros *shared_ptr* podem apontar para um mesmo endereço de memória, logo, a complexidade aumenta porque também é necessário o controle de quantos ponteiros são proprietários do objeto em questão e então somente quando nenhum ponteiro for proprietário daquele objeto é possível que seja deletado [3]. Além disso, é permitida a cópia do ponteiro inteligente *shared_ptr* para outro do mesmo tipo, dado que esse ponteiro é compartilhado.

O objetivo da pesquisa foi demonstrar de forma prática que o uso de *Smart Pointers* mitiga qualquer probabilidade de ocorrerem problemas relacionados ao lapso da liberação de memória por parte do desenvolvedor.

MATERIAL E MÉTODO

Os códigos fontes que serviram como exemplo de *Smart Pointers* foram desenvolvidos no programa DEV C++, versão 5.11 na plataforma *Microsoft Windows*. Os ponteiros inteligentes, *unique_ptr* e *shared_ptr*, são responsáveis por exemplificar a parte conceitual, como também, a parte prática da pesquisa sobre os *Smart Pointers*.

A Figura 1 retrata o código fonte que utiliza o ponteiro *unique_ptr*. Nesse código há uma classe denominada “ExemploUNIQUE” e a função principal.

```
3   #include<iostream>
4   #include<memory>
5   using namespace std;
6
7   class ExemploUNIQUE
8   {
9   public:
10      void mostrar()
11          {
12              cout << "A::mostrar()\n{\tbla bla bla\n\tbla bla bla\n\t...\n}" << endl << endl;
13          }
14  };
15
16
17  int main(int argc, char*argv[])
18  {
19      // p1 é um "unique_ptr" da classe [tipo] Exemplo
20      unique_ptr<ExemploUNIQUE> p1 (new ExemploUNIQUE);
21
22
23      p1 -> mostrar();
24      // Retorna o endereço de memória de p1
25      cout << "Endereco de memoria de p1: " << p1.get() << endl;
26      cout << "_____" << endl;
27
28      // Move-se p1 para p2, então p1 fica vazio
29      // Nao pode copiar p1 para p2, pois, "unique_ptr" permite um ponteiro por vez
30      unique_ptr<ExemploUNIQUE> p2 = move(p1);
31      p2 -> mostrar();
32      cout <<"//Endereco de p1 eh zero, pois foi movido para p2!\n" << "Endereco de memoria de p1: " << p1.get() << endl;
33      cout << "Endereco de memoria de p2: " << p2.get() << endl;
34      cout << "_____" << endl;
35
36      // Move-se p2 para p3, então p2 fica vazio
37      unique_ptr<ExemploUNIQUE> p3 = move(p2);
38      p3 -> mostrar();
39      cout <<"//Endereco de p1 eh zero, pois foi movido para p2!\n" << "Endereco de memoria de p1: " << p1.get() << endl;
40      cout <<"//Endereco de p2 eh zero, pois foi movido para p3!\n" << "Endereco de memoria de p2: " << p2.get() << endl;
41      cout << "Endereco de memoria de p3: " << p3.get() << endl;
42      cout << "_____" << endl << endl;
43
44      return 0;
45  }
```

Figura 1 – *Código fonte utilizando o ponteiro inteligente unique_ptr*

Na Figura 1, no início da função principal, na linha de código 20 cria-se o objeto “*p1*” apontado pelo ponteiro inteligente *unique_ptr* do tipo “ExemploUNIQUE”. A partir dessa criação são geradas algumas situações para demonstrar as principais características do *unique_ptr*. Na linha 23 ocorre a chamada do método “*mostrar()*” pertencente a classe ‘ExemploUNIQUE’, esse método é encarregado de exibir na tela uma mensagem simples.

Posteriormente, na linha de código 25, há o exemplo de uma funcionalidade mais interessante, que é dada pela função “*p1.get()”*. Essa função deve retornar ao usuário o endereço de memória do objeto em questão. Por fim, é possível notar nas linhas de código 30 e 37, a restrição do ponteiro *unique_ptr*, a qual é dada pela permissão de apenas um ponteiro por objeto, assim sendo, para alterar o ponteiro é necessário mover o objeto para outro ponteiro do mesmo tipo usando a função *“move()”*, a qual também é responsável por redefinir o *unique_ptr* antigo.

Na Figura 2, percebe-se que o ponteiro inteligente *shared_ptr* se difere do *unique_ptr* a partir da linha de código número 34, em que ocorre a chamada da função *“use_count()*”, a qual é responsável por retornar a quantidade de proprietários que o objeto possui no momento em que a chamada foi realizada. Posteriormente, na linha de código 39 ocorre chamada da função *“reset()”*, a qual é complementar a anterior, pois, é responsável por redefinir o ponteiro, atribuir *NULL* a ele e então decrementar o *“use_count()”*.

A fim de conseguir uma homogeneidade dos resultados, os códigos fonte foram compilados e executados utilizando a mesma plataforma, *Microsoft Windows*, mesmo programa, DEV C++ 5.11 e a mesma máquina.

A Figura 2 retrata o código fonte utilizando o ponteiro inteligente *shared_ptr*, no código há uma classe denominada “ExemploSHARED” e a função principal.

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

class ExemploSHARED
{
public:
    void mostrar()
    {
        cout << "A::mostrar()\n{\n\t...\n\t...\n\t...\n}" << endl;
    }
};

int main(int argc, char*argv[])
{
    shared_ptr<ExemploSHARED> p1(new ExemploSHARED);


    p1 -> mostrar();
    cout << "Endereco de memoria de p1: " << p1.get() << endl;
    cout << "________________________________________________________________________________" << endl;

    shared_ptr<ExemploSHARED> p2(p1);
    p2 -> mostrar();
    cout << "Endereco de memoria de p1: " << p1.get() << endl;
    cout << "Endereco de memoria de p2: " << p2.get() << endl;
    cout << "________________________________________________________________________________" << endl;

    // Retorna o número de objetos "shared_ptr"
    //(compartilhados) referentes ao mesmo
    //objeto gerenciado.
    cout << "Ponteiros acessando objeto: " << p1.use_count() << endl;
    cout << "Ponteiros acessando objeto: " << p2.use_count() << endl;

    // P1 é resetado, deixa de ter propriedade
    //sobre o objeto e ponteiro se torna NULL
    p1.reset();
    cout << "\n//Endereco de p1 agora eh zero, pois foi resetado\n" << "Endereco de memoria de p1: " << p1.get() << endl;
    cout << "Ponteiros acessando objeto: "  << p2.use_count()   << endl;
    cout << "Endereco de memoria de p2: "   << p2.get()        << endl;
    cout << "________________________________________________________________________________" << endl;

    return 0;
}
```

Figura 2 – *Código fonte utilizando o ponteiro inteligente shared_ptr*

RESULTADOS

Os testes realizados em ambos códigos fonte compilaram perfeitamente, isto é, após a utilização, os ponteiros foram desalocados corretamente sem a necessidade de utilizar os comandos *free* ou *delete* em nenhum momento.

As Figuras 3 e 4 retratam, respectivamente, a tela de execução dos programas que utilizam os ponteiros inteligentes *unique_ptr* e *shared_ptr*.

```
Endereco de memoria de p1: 0x821590
A::mostrar()
{       bla bla bla
        bla bla bla
        ...
}

//Endereco de p1 eh zero, pois foi movido para p2!
Endereco de memoria de p1: 0
Endereco de memoria de p2: 0x821590
A::mostrar()
{       bla bla bla
        bla bla bla
        ...
}

//Endereco de p1 eh zero, pois foi movido para p2!
Endereco de memoria de p1: 0
//Endereco de p2 eh zero, pois foi movido para p3!
Endereco de memoria de p2: 0
Endereco de memoria de p3: 0x821590


--------------------------------
Process exited after 0.1175 seconds with return value 0
Pressione qualquer tecla para continuar. . .
```

Figura 3 – Resultado da execução do programa utilizando o ponteiro unique_ptr

```
A::mostrar()
{
        ...
        ...
        ...
}
Endereco de memoria de p1: 0xbd1590
A::mostrar()
{
        ...
        ...
        ...
}
Endereco de memoria de p1: 0xbd1590
Endereco de memoria de p2: 0xbd1590
Ponteiros acessando objeto: 2
Ponteiros acessando objeto: 2

//Endereco de p1 agora eh zero, pois foi resetado
Endereco de memoria de p1: 0
Ponteiros acessando objeto: 1
Endereco de memoria de p2: 0xbd1590


--------------------------------
Process exited after 0.1316 seconds with return value 0
Pressione qualquer tecla para continuar. . .
```

Figura 4 – Resultado da execução do programa utilizando o ponteiro shared_ptr

Na Figura 3 é apresentado o funcionamento efetivo da restrição de cópia do *unique_ptr*, pois, ao mover o objeto “*p1*” para o “*p2*” o endereço de memória do primeiro é redefinido, então quando é

solicitado seu endereço de memória, há o retorno do valor zero. O mesmo procedimento ocorre quando "*p2*" é movido para "*p3*".

Na Figura 4, ao utilizar o ponteiro *shared_ptr* é possível copiar "*p1*" para "*p2*" sem problemas. Então, após a cópia, ambos apontam para o mesmo endereço de memória. Logo, existem dois ponteiros acessando aquele objeto simultaneamente. Por fim, quando um ponteiro do tipo *shared_ptr* é usado só pode ser liberado quando não houver mais proprietários e isso ocorrerá somente quando todos os objetos forem "*resetados*" por meio do comando "*reset()*".

DISCUSSÃO E CONCLUSÃO

A utilização dos *Smart Pointers* tem como consequência benefícios significativos relacionados ao aspecto prático e estrutural da programação. Nos exemplos anteriores, ao usar os ponteiros inteligentes foram otimizadas três linhas de código referente a desalocação dos três ponteiros declarados. Portanto, partindo do pressuposto que o programador por um lapso não implementar a desalocação de memória para essas três linhas, nenhum erro será gerado e o programa será compilado. Contudo, ainda assim haveria uma falha estrutural e em algum momento futuro o programa pode estourar a memória e ocasionar falhas.

Existem três benefícios principais na implementação de um código usando ponteiros inteligentes. A primeira vantagem está relacionada ao aspecto prático da programação, em que os problemas relativos à liberação de memória passam a ser responsabilidade do compilar e não mais do desenvolvedor [3][4]. O segundo benefício, assim como o primeiro, também está relacionado a parte prática da programação, isto é, o programador não precisa utilizar os comandos *delete* ou *free* em nenhum momento [3][4]. A utilização de *Smart Pointers* elimina também o risco de "*Deny Pointers*", ou seja, ponteiros que apontam para objetos já deletados. Assim sendo, essa última vantagem diferente das duas primeiras, agora está ligada a um aspecto estrutural da programação.

A partir de toda essa problemática que pode ser gerada devido a um lapso do desenvolvedor, pode-se inferir que a implementação de ponteiros inteligentes deve ser primordial ao trabalhar com alocação de memória dinâmica. A necessidade de usar ponteiros inteligentes aumenta ainda mais quando se trata de programação orientada a objetos, dado que, o ambiente da orientação a objeto será benéfico apenas com determinadas características, sendo elas: equipes grandes (mais de mil pessoas), código gigante (chegando a milhões de linhas), requisitos voláteis e multiplataformas [5].

Dessa maneira, conclui-se que o uso de ponteiros inteligentes é mais do que uma boa prática de programação. É uma ferramenta essencial para a programação, a qual realiza a desalocação automática da memória dinâmica, além disso, é um coletor de lixo, pois, após desalocar a memória, os ponteiros inteligentes *unique_ptr* e *shared_ptr* removem todo o lixo que foi gerado durante a execução. Portanto, os desenvolvedores devem se atentar a essa funcionalidade e utilizá-la sempre que possível a fim de minimizar problemas posteriores.

REFERÊNCIAS

[1] SILVA, Flávio de Oliveira – Ponteiros. 2010. Disponível em: <http://www.facom.ufu.br/~flavio/ed1/files/C++%20ORIENTADO%20A%20OBJETOS%20-%20Ponteiros.pdf >. Acesso em: 09 mai. 2019.

[2] MANSSOUR, I. H. – Linguagem de Programação C – 8. Ponteiros. Disponível em: <http://www.inf.pucrs.br/~manssour/LinguagemC/PoligC-Cap08.pdf>. Acesso em: 08 mai. 2019.

[3] KIERAS, D.E. – Using C++11's Smart Pointers. 2016. Disponível em: < http://umich.edu/~eecs381/handouts/C++11_smart_ptrs.pdf >. Acesso em: 15 mai. 2019.

[4] MOCK, K. – Smart Pointers C++11. 2015. Disponível em: <http://www.math.uaa.alaska.edu/~afkjm/csce331/handouts/SmartPointers.pdf >. Acesso em:15 mai. 2019.

[5] KAMIENSKI, C. A. – Introdução ao paradigma de orientação a objetos. 1996. Disponível em: <http://www.cin.ufpe.br/~rcmg/cefet-al/proo/apostila-poo.pdf>. Acesso em: 16 mai. 2019.