Ponteiros e alocação dinâmica

Paulo Ricardo Lisboa de Almeida

UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

Diagramas UML

- Faça o download do projeto disponibilizado
  - Entenda a classe Disciplina
  - Compile e execute

Definindo um professor

- Vamos definir que uma Disciplina possui um dado membro que indica o professor da disciplina
  - Utilizando a estrutura que já temos, como fazer?

Definindo um professor

- A forma mais simples é definir um objeto do tipo pessoa dentro de *Disciplina*, que indicará quem é o *professor*

Definindo um professor

Disciplina.hpp

```
#ifndef DISCIPLINA_H
#define DISCIPLINA_H

#include <string>

#include "Pessoa.hpp"

class Disciplina{
	public:
		//...

		Pessoa getProfessor();
		void setProfessor(Pessoa novoProfessor);
	private:
		//...

		Pessoa professor;
};
#endif
```

Disciplina.cpp

```
#include "Disciplina.hpp"

//...

Pessoa Disciplina::getProfessor(){
	return professor;
}

void Disciplina::setProfessor(Pessoa novoProfessor){
	professor = novoProfessor;
}
```

UDESC

No main

```cpp
#include <iostream>
#include <string>

#include "Pessoa.hpp"
#include "Disciplina.hpp"

int main(){
	Pessoa p1{"Joao", 20};
	Disciplina disciplina{"C++"};
	disciplina.setProfessor(p1);

	std::cout << p1.getNome() << std::endl;
	std::cout << disciplina.getProfessor().getNome() << std::endl;



	return 0;
}
```

Pergunta

- Note que passamos *p1* como o objeto *professor* de Disciplina via *set*
- A questão é
  - P1 e professor se referem ao mesmo objeto na memória, ou agora temos “duas cópias” da pessoa João na memória?

```
int main(){
        Pessoa p1{"Joao", 20};
        Disciplina disciplina{"C++"};
        disciplina.setProfessor(p1);


        std::cout << p1.getNome() << std::endl;
        std::cout << disciplina.getProfessor().getNome() << std::endl;



        return 0;
}
```

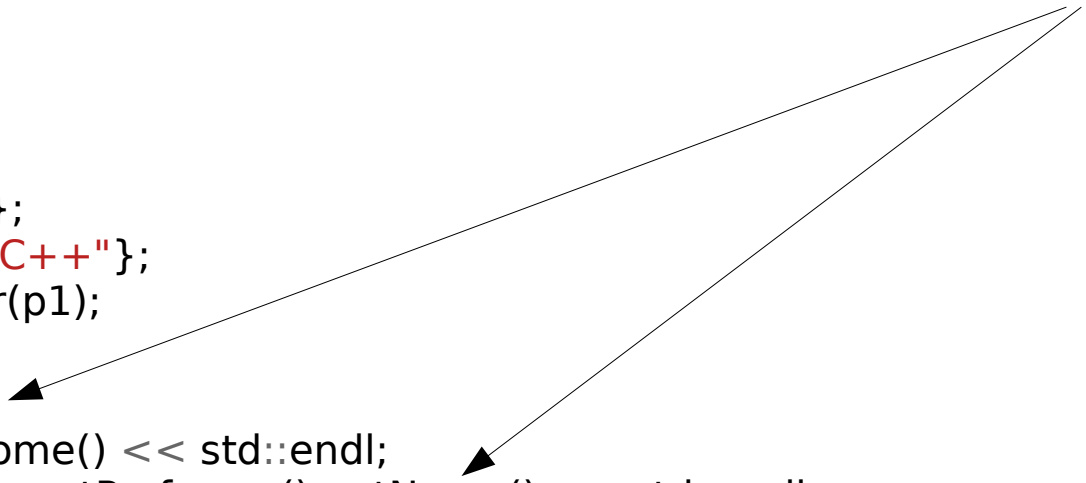
Pergunta

- *p1* e *professor* se referem ao mesmo objeto na memória, ou agora temos “duas cópias” da pessoa João na memória?
  - O conceito é o mesmo que com *structs* em C, e nesse cenário fizemos uma cópia do objeto
    - Temos duas cópias na memória
    - O construtor de cópia padrão foi chamado
      - Veremos mais sobre ele no futuro

Faça o teste você mesmo

```
#include <iostream>
#include <string>

#include "Pessoa.hpp"
#include "Disciplina.hpp"

int main(){
	Pessoa p1{"Joao", 20};
	Disciplina disciplina{"C++"};
	disciplina.setProfessor(p1);

	p1.setNome("Pedro");
	std::cout << p1.getNome() << std::endl;
	std::cout << disciplina.getProfessor().getNome() << std::endl;


	return 0;
}
```

Modificamos o nome de p1, mas o nome do objeto professor se mantém o mesmo



UDESC

Vantagens e desvantagens

- No teste que fizemos uma passagem por cópia
  - Quais as vantagens de se realizar uma cópia do objeto?
  - Quais as desvantagens?

Vantagens e desvantagens

- No teste que fizemos uma passagem por cópia
  - Quais as vantagens de se realizar uma cópia do objeto?
    - Podemos alterar o objeto original sem alterar o copiado, e vice-versa
    - A passagem é simples e fácil de entender
  - Quais as desvantagens?
    - Uma passagem por cópia custa caro
      - Memória e CPU → Overhead
    - Muitas vezes desejamos que uma alteração em “qualquer versão” do objeto gere alterações em “todos os objetos”
      - Exemplo: desejamos modificar o nome do professor tanto através de *p1*, quanto através do objeto *professor* da disciplina
        - Geralmente esse é o caso
          - Ex.: Não desejamos que uma mesma pessoa (de mesmo cpf) possua diferentes nomes na memória

Pergunta

- Como passar o objeto “original”

Ponteiros

- Podemos utilizar ponteiros da mesma forma que em C

Teste você mesmo

- Podemos utilizar ponteiros da mesma forma que em C
- Modifique o dado membro “professor” da classe Disciplina para que ele seja um ponteiro para pessoa
  - Modifique os *gets* e *sets* também
- Faça novamente o teste no main
  - Passe agora o ponteiro para o objeto pessoa criado
  - Verifique se mudanças no “objeto original” alteram também o objeto professor que está dentro de Disciplina

Exemplo

Disciplina.hpp

```
        //...
        Pessoa* getProfessor();
        void setProfessor(Pessoa* novoProfessor);
private:
        std::string nome;
        unsigned short int cargaHoraria;
        Pessoa* professor;
        //...
```

Disciplina.cpp

```
#include "Disciplina.hpp"


//...
Pessoa* Disciplina::getProfessor(){
        return professor;
}

void Disciplina::setProfessor(Pessoa* novoProfessor){
        professor = novoProfessor;
}
```

main.cpp

Professor é um ponteiro, então use o perador seta (->) como em C

```
#include <iostream>
#include <string>


#include "Pessoa.hpp"
#include "Disciplina.hpp"


int main(){
        Pessoa p1{"Joao", 20};
        Disciplina disciplina{"C++"};
        disciplina.setProfessor(&p1);

        p1.setNome("Pedro");
        std::cout << p1.getNome() << std::endl;
        std::cout << disciplina.getProfessor()->getNome() << std::endl;



        return 0;
}
```

UDESC

Operadores

- Os operadores de ponteiros de C *****, **&** e **->** continuam válidos em C++

Alocação dinâmica de memória

- Podemos por exemplo, declarar um ponteiro, que recebe o endereço de um objeto já alocado
  - Exemplo

```
Pessoa p1{"Joao", 20};
Pessoa* ponteiro = &p1; //recebe o endereço de p1
```

Alocação dinâmica de memória

- Podemos por exemplo, declarar um ponteiro, que recebe o endereço de um objeto já alocado
  - Exemplo

```
Pessoa p1{"Joao", 20};
Pessoa* ponteiro = &p1; //recebe o endereço de p1
```

- Mas e se desejarmos criar um novo objeto dinamicamente
  - Como faríamos em C?

Alocação dinâmica de memória

- Mas e se desejarmos criar um novo objeto dinamicamente
  - Como fazer em C?
    - Em C utilizamos uma combinação de *malloc* e *sizeof*
    - *malloc* ainda é válido em C++
- Mas **nunca use** *malloc* em C++, a não ser que você tenha certeza do que está fazendo
  - O *malloc* vai gerar problemas principalmente com os construtores
    - *malloc* aloca memória, mas não chama os construtores
  - Um problema similar acontece com o *free*

Operador new

- O operador ***new***
  - Aloca memória para o objeto
    - O mesmo que um *malloc* com *sizeof*
  - Chama **automaticamente o construtor** da classe
- Para acessar os dados e funções membro de um objeto alocado dinamicamente, utilize o operador **->**
  - O mesmo operador utilizado em C para *structs* alocadas dinamicamente

Operador new

- O operador ***new*** lança uma exceção **bad_alloc** caso não consiga criar o objeto (ex.: devido a falta de memória)
  - Para conseguir tratar isso, mais uma vez precisaremos esperar pelas aulas relacionadas a exceções
- O objeto alocado fica na **heap** da memória
  - Assim como as variáveis alocadas dinamicamente em C
  - Veja detalhes sobre o mapa de memória nas disciplinas de
    - Arquitetura e Organização de Computadores
    - Sistemas Operacionais

Exemplo

No main.cpp

```
Pessoa* ptr1{new Pessoa};//Utilizando construtor default
Pessoa* ptr2{new Pessoa{"Joana", 22}};//Utilizando construtor com parâmetros
int* ptrInt{new int};//inteiro alocado e com lixo de memória
int* ptrIntIniciado{new int{2}};//inteiro alocado e inicializado com 2

ptr1->setNome("Maria");
*ptrInt = 20;

std::cout << ptr1->getNome() << std::endl;
std::cout << ptr2->getNome() << std::endl;
std::cout << *ptrInt << std::endl;
std::cout << *ptrIntIniciado << std::endl;
```

new e vetores

- Para alocar um vetor dinamicamente utilizando *new*, basta indicar entre colchetes o tamanho do vetor
- Exemplo

```
int* array{new int[10]};//vetor de 10 posições

for(int i=0; i < 10; i++){
	array[i] = i;
	std::cout << array[i] << std::endl;
}
```

new e matrizes

- Uma das formas de se alocar uma matriz dinamicamente em C++ segue os mesmos princípios clássicos do C
  - Como podemos alocar uma matriz dinamicamente em C/C++?

new e matrizes

- Uma das formas de se alocar uma matriz dinamicamente em C++ segue os mesmos princípios clássicos do C
  - Como podemos alocar uma matriz dinamicamente em C/C++?
    - Aloque um vetor de ponteiros
      - Cada ponteiro deve apontar para um vetor de itens (ex.: inteiros)

new antes do C++11

- Antes do C++11, a maneira de se alocar dinamicamente objetos era usando uma atribuição seguida do new
- Exemplo

```
Pessoa* ptr1 = new Pessoa();
Pessoa* ptr2 = new Pessoa("Joana", 22);
```

- Essas construções ainda são válidas, mas são **desencorajadas**
  - Obs.: notou semelhança com o Java? De onde você acha que o Java tirou essa ideia?

new

- Memória alocada dinamicamente **precisa** ser manualmente desalocada
- Não liberar a memória causa os mesmos problemas que já conhecemos em C
  - Vazamentos de memória (*memory leaks*)

delete

- O operador ***delete*** libera a memória alocada por um ***new***
- Utilize
  - Para variáveis e objetos simples
    - *delete*
    -
  - *Para vetores*
    - *delete[]*
      - Delete seguido de abre e fecha colchetes

Exemplo

```
Pessoa* ptr1{new Pessoa};
Pessoa* ptr2{new Pessoa{"Joana", 22}};
int* ptrInt{new int};//inteiro alocado e com lixo de memória
int* ptrIntIniciado{new int{2}};//inteiro alocado e inicializado com 2
int* array{new int[10]};
ptr1->setNome("Maria");
*ptrInt = 20;


std::cout << ptr1->getNome() << std::endl;
std::cout << ptr2->getNome() << std::endl;
std::cout << *ptrInt << std::endl;
std::cout << *ptrIntIniciado << std::endl;
for(int i=0; i < 10; i++){
        array[i] = i;
        std::cout << array[i] << std::endl;
}


delete ptr1;//sem o asterisco!!!
delete ptr2;
delete ptrInt;
delete ptrIntIniciado;
delete[] array;
```

UDESC

Importante

- Da mesma forma que em C, um ponteiro não inicializado aponta para uma região aleatória da memória
  - Um ponteiro selvagem (*wild pointer*)

- A constante ***nullptr*** indica um **ponteiro nulo**
  - Você pode inicializar um ponteiro com ***nullptr***, atribuir ***nullptr*** a um ponteiro, ou fazer comparações com ***nullptr***
    - O conceito de ***nullptr*** se tornou oficial no C++11
  - Exemplo

```
Pessoa* ptr3{nullptr};
if(ptr3 == nullptr)
        std::cout << "Ptr3 eh nulo" << std::endl;
```

Exercícios

1. Na classe **Disciplina**, adicione uma função membro chamada getNomeProfessor, que retorna uma string com o nome do professor

2. Veja nessa matéria que memory leaks eram responsáveis pelo baixo desempenho do jogo Just Cause 3
   - www.gamesear.com/news/just-cause-3-pc-patch-is-now-live-aims-to-fix-loading-times-memory-leaks-and-more

3. Adicione dados membro para representar alunos da Disciplina
   - Adicione um **vetor fixo** de tamanho 50 em Disciplina
   - O vetor vai conter os **alunos** da disciplina
   - Adicione pelo menos os seguintes métodos em Disciplina
     - bool adicionarAluno(Pessoa* aluno);
     - Pessoa* getVetorAlunos();
       - Opcional: crie funções membro
         - bool removerAluno(Pessoa* aluno)
         - bool removerAluno(unsigned long cpfAluno);
   - Aloque dinamicamente algumas pessoas no main, e adicione-as como alunos da disciplina
   - Os métodos bool retornam true se tudo ocorreu corretamente, ou false em caso de erro

Referências

- DEITEL, P.; DEITEL, H. **C++ how to Program**. [S.l.]: Pearson, 2017. ISBN 9780134448237
- STROUSTRUP, B. **The C++ Programming Language**. Pearson Education, 2013. ISBN 9780133522853.