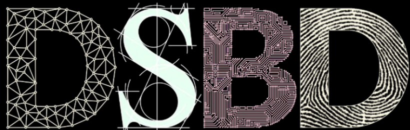
"Todos os problemas na ciência da computação podem ser resolvidos adicionando um nível extra de indireção, exceto pelo problema de se ter muitos níveis de indireção" (David Wheeler).

Ponteiros e alocação dinâmica

Paulo Ricardo Lisboa de Almeida



Antes de começar

Faça o download do projeto disponibilizado.

Entenda a classe Disciplina.

Compile e execute.

Professor

Vamos definir que uma Disciplina possui um dado membro que indica o professor da disciplina.

Utilizando a estrutura que já temos, como fazer?

Professor

Vamos definir que uma Disciplina possui um dado membro que indica o professor da disciplina.

Podemos definir um membro do tipo Pessoa dentro de Disciplina, que indicará quem é o professor.

```cpp
#ifndef DISCIPLINA_H
#define DISCIPLINA_H

#include <string>

#include "Pessoa.hpp"

class Disciplina{
   public:
       Disciplina(std::string nomeDisciplina);

       std::string getNome();
       void setNome(std::string novoNome);

       int getCargaHoraria();
       void setCargaHoraria(unsigned int novaCarga);

       Pessoa getProfessor();
       void setProfessor(Pessoa prof);

   private:
       std::string nome;
       unsigned short int cargaHoraria;
       Pessoa professor;
};
#endif
```

```cpp
#include "Disciplina.hpp"

//...

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

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

No main

```cpp
#include<iostream>

#include<string>

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

int main(){
   Pessoa p1{"Joao", 11111111111, 20};

   Disciplina d1{"Orientacao a Objetos"};
   d1.setProfessor(p1);

   std::cout << p1.getNome() << '\t' << p1.getIdade() << '\t' <<  p1.getCpf() << std::endl;

   std::cout << "Disciplina: " << d1.getNome() << std::endl;
   std::cout << "Professor: " << d1.getProfessor().getNome() << std::endl;

   return 0;
}
```

Pergunta

Passamos p1 como o objeto professor de Disciplina via set.

P1 e professor se referem ao mesmo objeto na memória, ou agora temos duas cópias da “mesma pessoa” na memória?

Pergunta

Passamos p1 como o objeto professor de Disciplina via set.

P1 e professor se referem ao mesmo objeto na memória, ou agora temos duas cópias da “mesma pessoa” 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 invocado.
- Você pode estudar sobre construtores de cópia nos livros indicados na bibliografia.

Teste você mesmo

Alterar o nome de p1 não altera o nome do Professor da disciplina.

```
#include<iostream>

#include<string>

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

int main(){
   Pessoa p1{"Joao", 11111111111, 20};

   Disciplina d1{"Orientacao a Objetos"};
   d1.setProfessor(p1);

   p1.setNome("Joao Silva");

   std::cout << p1.getNome() << '\t' << p1.getIdade() << '\t' <<  p1.getCpf() << std::endl;

   std::cout << d1.getNome() << std::endl;
   std::cout << d1.getProfessor().getNome() << std::endl;

   return 0;
}
```

Passagem por cópia

No exemplo foi feita uma **passagem por cópia.**

- \+ Vantagens de realizar uma cópia do objeto?
- \- Quais as desvantagens?

Passagem por cópia

No exemplo foi feita uma **passagem por cópia.**

+ Vantagens:
  + Podemos alterar o objeto original sem alterar o copiado, e vice-versa.
    Muitas vezes é esse o comportamento esperado.
  + A passagem é simples e fácil de entender.
- Quais as desvantagens?

Passagem por cópia

No exemplo foi feita uma **passagem por cópia.**

- Desvantagens:
  - Uma passagem por cópia custa caro.
    - Memória e CPU ⟶ Overhead.
  - Muitas vezes desejamos que uma alteração em "qualquer" objeto gere alterações em "todos os objetos".
    - Exemplo: modificar o nome do professor tanto através de p1, quanto através do objeto professor de d1.
      - 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.

Disciplina.hpp

```cpp
#ifndef DISCIPLINA_H
#define DISCIPLINA_H

#include <string>

#include "Pessoa.hpp"

class Disciplina{
   public:
      //...

      Pessoa* getProfessor();
      void setProfessor(Pessoa* prof);

   private:
      std::string nome;
      unsigned short int cargaHoraria;
      Pessoa* professor;
};
#endif
```

main.cpp

```cpp
int main(){
   Pessoa p1{"Joao", 11111111111, 20};

   Disciplina d1{"Orientacao a Objetos"};
   d1.setProfessor(&p1);

   p1.setNome("Joao Silva");
   std::cout << p1.getNome() << '\t' << p1.getIdade()
       << '\t' <<  p1.getCpf() << '\n';

   std::cout << d1.getNome() << '\n';
   std::cout << d1.getProfessor()->getNome() << '\n';

   return 0;
}
```

Disciplina.cpp

```cpp
#include "Disciplina.hpp"

//...

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

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

Operadores

Os operadores de ponteiros de C continuam válidos em C++.

```
*

&

->
```

Atribuição

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

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

Alocação Dinâmica

Mas e se desejarmos criar um novo objeto dinamicamente.

Como faríamos em C?

Alocação Dinâmica

Mas e se desejarmos criar um novo objeto dinamicamente.

Em C utilizamos uma combinação de `malloc` e `sizeof`.

`malloc` ainda é válido em C++.

Alocação Dinâmica

Mas e se desejarmos criar um novo objeto dinamicamente.

Em C utilizamos uma combinação de `malloc` e `sizeof`.

`malloc` ainda é válido em C++.

Nunca use `malloc` em C++, exceto se você tiver certeza do que está fazendo.

Alocação Dinâmica

Nunca use `malloc` em C++, exceto se você tiver certeza do que está fazendo.

- O `malloc` vai gerar problemas principalmente com os construtores.
- O `malloc` aloca memória, mas não chama os construtores.
- Um **problema similar** acontece com o `free`.

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 seta ->.

- O mesmo operador utilizado em C para structs referenciadas por ponteiros.

new

O operador `new` lança uma exceção `bad_alloc` caso não consiga criar o objeto (ex.: devido a falta de memória).

- Para tratar isso, mais uma vez precisaremos esperar pelas aulas relacionadas a exceções.

O objeto alocado fica na **heap**.

- Assim como as variáveis alocadas dinamicamente em C.

Exemplo

```
#include<iostream>

#include "Pessoa.hpp"

int main(){
   Pessoa* ptr1{new Pessoa};//Utilizando construtor default
   Pessoa* ptr2{new Pessoa{"Joana", 11111111111,22}};//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");//operador -> para derreferenciar a função de objeto apontado

   *ptrInt = 20;//mesma coisa que com C

   std::cout << ptr1->getNome() << '\n';
   std::cout << ptr2->getNome() << '\n';
   std::cout << *ptrInt << '\n';
   std::cout << *ptrIntIniciado << '\n';

   return 0;
}
```

new e vetores

Para alocar um vetor dinamicamente utilizando new, basta indicar entre colchetes [ ] o tamanho do vetor

Exemplo:

```cpp
#include<iostream>


int main(){
   int* array{new int[10]};//vetor de 10 posições

   for(int i=0; i < 10; i++){
       array[i] = i;
       std::cout << array[i] << '\n';
   }

   return 0;
}
```

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**.

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? O Java herdou muitas das construções do C++.

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`.

Para variáveis e objetos simples:

```
delete
```

Para vetores:

```
delete[]
```

Delete seguido de abre e fecha colchetes.

Exemplo

```cpp
#include<iostream>

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

int main(){
   Pessoa* ptr1{new Pessoa};
   Pessoa* ptr2{new Pessoa{"Joana", 11111111111, 22}};
   int* ptrInt{new int};//inteiro alocado e com lixo de memória
   int* array{new int[10]};
   ptr1->setNome("Maria");
   *ptrInt = 20;
   std::cout << ptr1->getNome() << '\n' << ptr2->getNome() << '\n';
   std::cout << *ptrInt << '\n';
   for(int i=0; i < 10; i++){
       array[i] = i;
       std::cout << array[i] << '\n';
   }
   delete ptr1;//sem o asterisco!!!
   delete ptr2;
   delete ptrInt;
   delete[] array;

   return 0;
}
```

Cuidado

Da mesma forma que em C, um ponteiro não inicializado aponta para uma região indeterminada da memória.

- Um ponteiro selvagem (wild pointer).

Cuidado

Da mesma forma que em C, um ponteiro não inicializado aponta para uma região indeterminada 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:

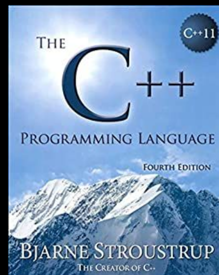
```
Pessoa* ptr3{nullptr};//inicializado com nullptr
if(ptr3 == nullptr)
    std::cout << "Ptr3 eh nulo" << std::endl;
```

Exercícios

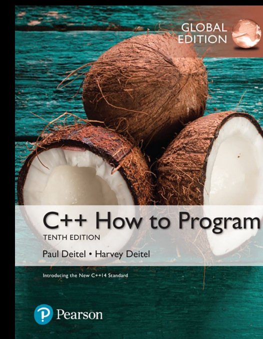
1. Na classe Disciplina, adicione uma função membro chamada getNomeProfessor, que retorna uma string com o nome do professor.
2. Adicione dados membro para representar alunos da Disciplina:
   a. Adicione um vetor fixo de tamanho 50 em Disciplina.
      O vetor vai conter os alunos da disciplina.
   b. Adicione pelo menos as seguintes funções em Disciplina.
      bool adicionarAluno(Pessoa* aluno);
      Pessoa* *getVetorAlunos();
3. Aloque dinamicamente algumas pessoas no main, e adicione-as como alunos da disciplina.
4. Opcional: crie funções membro:
      bool removerAluno(Pessoa* aluno)
      bool removerAluno(unsigned long cpfAluno);
      Obs.: As funções bool retornam true se tudo ocorreu corretamente, ou false em caso de erro.

Referências

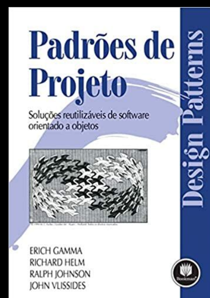
Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2013.



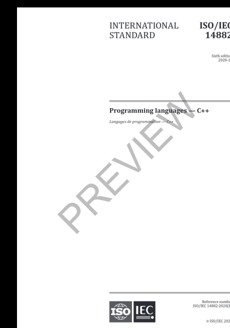
Deitel, H. M., Deitel, P. J. C++: como programar. 10a ed. Pearson Prentice Hall. 2017.



Gamma, E. Padrões de Projetos: Soluções Reutilizáveis. Bookman. 2009.



ISO/IEC 14882:2020 Programming languages - C++: www.iso.org/obp/ui/#iso:std:iso-iec:14882:ed-6:v1:en



Licença

Esta obra está licenciada com uma Licença Creative Commons Atribuição 4.0 Internacional.