Funções Virtuais

Paulo Ricardo Lisboa de Almeida

UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

Pergunta

- Podemos fazer isso?
  - Usar um ponteiro para *Pessoa*, e atribuir a ele um (o endereço de) um objeto do tipo *ProfessorAdjunto*?

```
//...

int main(){
        Pessoa* p{new ProfessorAdjunto{"Maria", 11111111111, 100, 40}};
        std::cout << p->getNome() << std::endl;
        delete p;

        return 0;
}
```

Pergunta

- Podemos fazer isso?
  - Usar um ponteiro para *Pessoa*, e atribuir a ele um (o endereço de) um objeto do tipo *ProfessorAdjunto?*
    - **Sim podemos**
      - ProfessorAdjunto é um **tipo de** Pessoa

```
//...

int main(){
        Pessoa* p{new ProfessorAdjunto{"Maria", 11111111111, 100, 40}};
        std::cout << p->getNome() << std::endl;
        delete p;

        return 0;
}
```

UDESC

Pergunta

- No entanto existe um problema
  - A construção do exemplo pode levar a problemas de consistência no sistema, ou até a memory leaks
  - **Você consegue encontrar o problema?**

```
//...

int main(){
        Pessoa* p{new ProfessorAdjunto{"Maria", 11111111111, 100, 40}};
        std::cout << p->getNome() << std::endl;
        delete p;

        return 0;
}
```

Pergunta

- No entanto existe um problema
  - A construção do exemplo pode levar a problemas de consistência no sistema, ou até a memory leaks
  - **Você consegue encontrar o problema?**
    - Dica 1: o problema está no main, não precisa abrir as demais classes para encontrá-lo

```
//...

int main(){
        Pessoa* p{new ProfessorAdjunto{"Maria", 11111111111, 100, 40}};
        std::cout << p->getNome() << std::endl;
        delete p;

        return 0;
}
```

Pergunta

- No entanto existe um problema
  - A construção do exemplo pode levar a problemas de consistência no sistema, ou até a memory leaks
  - **Você consegue encontrar o problema?**
    - Dica 1: o problema está no main, não precisa abrir as demais classes para encontrá-lo
    - Dica 2: Qual destrutor será chamado?

```
//...

int main(){
        Pessoa* p{new ProfessorAdjunto{"Maria", 11111111111, 100, 40}};
        std::cout << p->getNome() << std::endl;
        delete p;

        return 0;
}
```

Pergunta

- No exemplo, o delete de Pessoa será chamado
  - Mas os destrutores de Professor, e de ProfessorAdjunto não serão invocados
    - Geramos os mais diversos problemas
  - Isso ocorre pelo mesmo motivo discutido na aula passada
    - O ponteiro é para Pessoa, e o ponteiro não é capaz de identificar em tempo de execução que o objeto é do tipo ProfessorAdjunto
    - Lembre-se que o destrutor **também é uma função membro**

```
//...

int main(){
        Pessoa* p{new ProfessorAdjunto{"Maria", 11111111111, 100, 40}};
        std::cout << p->getNome() << std::endl;
        delete p;

        return 0;
}
```

Teste você mesmo

- Coloque “couts” nos destrutores de Pessoa, Professor e ProfessorAdjunto
  - Compile, execute, e veja que somente o destrutor de Pessoa é invocado

Teste você mesmo

- **Modifique** o tipo de ponteiro de Pessoa para ProfessorAdjunto
  - Compile, execute, e note que agora os destrutores são chamados corretamente

O Modificador Virtual

- Para que o comportamento polimórfico funcione corretamente, precisamos de **funções virtuais**
- Para tornar uma função virtual
  - Basta adicionar o modificador ***virtual*** na declaração da função (no .hpp)
  - O .cpp **não é alterado**

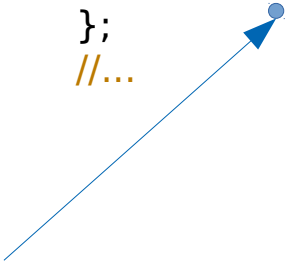
Exemplo

Professor.hpp

```
//...
class Professor : public Pessoa{
	public:
		virtual unsigned int getSalario() const;

		//...
	private:
		unsigned int valorHora;
		unsigned short cargaHoraria;
};
//...
```

ProfessorAdjunto.hpp

```
//...
class ProfessorAdjunto : public Professor{
	public:
		using Professor::Professor;

		~ProfessorAdjunto();

		virtual unsigned int getSalario() const;
};
//...
```



Não é obrigatório, mas é uma boa prática repetir o virtual

UDESC

No main

- Teste você mesmo
  - Mesmo exemplo da aula passada
    - Note que agora as chamadas funcionam como o esperado

```
#include <iostream>

#include "ProfessorAdjunto.hpp"
#include "Professor.hpp"

int main(){
	ProfessorAdjunto p{"Joao", 11111111111, 8500, 40};
	Professor p2{"Pedro", 11111111111, 8500, 40};
	Professor* ptr{&p};

	std::cout << p.getNome() << " " << p.getSalario() << std::endl;
	std::cout << ptr->getNome() << " " << ptr->getSalario() << std::endl;

	return 0;
}
```

Atenção

- Quando declaramos uma **função virtual**
  - Ela **permanece virtual por toda a hierarquia** de classes
    - Mesmo que uma das classes que a sobrescrevem não a definam como virtual
- Pelo bem da clareza
  - Se uma função é declarada como virtual na classe pai, declare-a como virtual na classe filha também quando for sobrescrever a função
    - Isso não é o obrigatório
      - Mas deixa a leitura do programa mais simples

Destrutores

- E o que os destrutores têm a ver com isso?

Destrutores

- E o que os destrutores têm a ver com isso?
  - Se chamarmos o destrutor **não virtual** via delete a partir de um ponteiro para a classe base (assumindo que o objeto em questão deriva da classe base), o **padrão do C++** especifica que teremos um **comportamento indefinido**

Destrutores

- **Todas as classes** devem ter um **destrutor virtual** para evitar esses problemas
  - Exceto se você conseguir pensar em um motivo realmente bom para que a classe não tenha um destrutor virtual
  - Declare destrutores virtuais mesmo em classes que possuem destrutor default
    - Nesse caso, o seu destrutor default não vai realizar tarefa alguma
  - No C++11, você pode criar um destrutor default virtual da seguinte forma
    virtual ~NomeClasse() = default;
    - Para evitar que você precise escrever o destrutor default no .cpp também

Construtores

- Construtores **não podem ser virtuais**

Custo computacional

- Funções virtuais são mais custosas
  - Adicionam indireções extras
    - Overhead
  - Estudaremos melhor nas próximas aulas

Virtual em outras linguagens

- Java
  - No Java toda função é “virtual”
    - O programador não tem controle sobre isso, toda função é obrigatoriamente virtual
  - Pró: Programação mais simples e melhor do ponto de vista da Eng. de Software
  - Contra: O custo extra de uma função virtual é mandatório no Java
- C#
  - Conta com mecanismos similares ao C++
    - Inclusive usa a palavra-chave virtual

Exercícios

1. Declare os destrutores virtuais para todas as classes do nosso Projeto

2. Analise o projeto disponibilizado no Moodle, que se trata de uma classe *Trajeto* que calcula a distância de um trajeto através de uma lista de pontos. Existe uma função chamada *calcularDistanciaPontos*, que é usada pela classe. Essa função sempre retorna zero.
   - Você deve criar duas classes que derivam da classe *Trajeto*, sendo que uma delas deve calcular a distância via a distância Euclidiana, e outra deve fazer o cálculo via distância Manhattan.
   - Utilize funções **virtuais sempre onde necessário**.
   - Crie instâncias de objetos de DistanhaManhattan e DistanciaEuclidiana no main para testar.

3. No exercício 2 foi utilizado o princípio da Inversão de Controle, comumente utilizado na criação de frameworks. Estude sobre inversão de controle.
   - Dica: procure por artigos sérios ou em livros de engenharia de software.
     - A maioria dos artigos em português misturam inversão de controle com injeção de dependência em frameworks Java, Python, PHP, … e fazem uma verdadeira salada (vocês são Cientistas da Computação, não programadores Java).

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.