Não existe almoço grátis!

Funções Virtuais

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR

Pergunta

Podemos fazer isso?

Usar um ponteiro para `Pessoa`, e atribuir a ele um (o endereço de) um objeto do tipo `ProfessorAdjunto`?

```cpp
#include <iostream>

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

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

```cpp
#include <iostream>

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

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

   return 0;
}
```

Pergunta

Existe um problema **grave**.

A construção do exemplo pode levar a problemas de consistência, ou até corromper a memória.

Você consegue encontrar o problema?

```
#include <iostream>

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

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

   return 0;
}
```

Pergunta

Existe um problema **grave**.

A construção do exemplo pode levar a problemas de consistência, ou até corromper a memória.

Você consegue encontrar o problema?

Dica: O problema está no main.

```cpp
#include <iostream>

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

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

   return 0;
}
```

Pergunta

Existe um problema **grave**.

A construção do exemplo pode levar a problemas de consistência, ou até corromper a memória.

Você consegue encontrar o problema?

Dica: O problema está no main.

Dica: Qual destrutor será chamado?

```cpp
#include <iostream>

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

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

   return 0;
}
```

Delete

- O `delete` de `Pessoa` será chamado.
  - 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`.
  - 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() << "\n";
   delete p;

   return 0;
}
```

Faça você mesmo

Coloque `couts` nos destrutores de `Pessoa`, `Professor` e `ProfessorAdjunto`.

Compile, execute, e veja que somente o destrutor de `Pessoa` é invocado.

Faça você mesmo

Modifique o tipo de ponteiro de `Pessoa` para `ProfessorAdjunto`

Compile, execute, e note que agora os destrutores são chamados corretamente.

Virtual

Para termos funções com comportamento realmente polimórfico, precisamos de **funções virtuais**.

Para tornar uma função virtual:

Adicione o modificador `virtual` na declaração da função (no .hpp).

O .cpp não é alterado.

Virtual

Para termos funções com comportamento realmente polimórfico, precisamos de **funções virtuais**.

Para tornar uma função virtual:

Adicione o modificador `virtual` na declaração da função (no .hpp).

O .cpp não é alterado.

Quando uma função é declarada `virtual`, os ponteiros são capazes de **inferir em tempo de execução** qual a função correta a chamar.

Mesmo se o ponteiro declarado for de uma classe base.

Exemplo

Antes de corrigir os construtores, vamos aplicar o conceito de virtual para corrigir a chamada do salário.

Note que por enquanto o exemplo a seguir não chama o `getSalario` corretamente via ponteiro.

```cpp
#include <iostream>

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

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

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

   return 0;
}
```

Exemplo

```
#ifndef PROFESSOR_HPP
#define PROFESSOR_HPP

#include "Pessoa.hpp"
class Professor : public Pessoa{
   public:

       //...

       virtual unsigned int getSalario() const;

   private:
       unsigned int valorHora;
       unsigned short cargaHoraria;
};
#endif
```

Não é obrigatório repetir nas classes derivadas, mas é uma **boa prática**.

```
#ifndef PROFESSOR_ADJUNTO_HPP
#define PROFESSOR_ADJUNTO_HPP

#include "Professor.hpp"

class ProfessorAdjunto : public Professor{
   public:

       //...

       virtual unsigned int getSalario() const;
   private:
       std::string linhaPesquisa;
};
#endif
```

Exemplo

Mesmo exemplo da aula passada.

Note que agora as chamadas funcionam como o esperado.

```cpp
#include <iostream>

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

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

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

   return 0;
}
```

Virtual

Quando uma função é declarada `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.

Boa prática.

Override

A partir do C++11 você pode opcionalmente adicionar `override` no final do protótipo de uma função que sobrecarrega o comportamento de uma função virtual.

Boa prática.

Deixa claro que se trata de uma sobrecarga.

Informa ao compilador a sua intenção.

Se a função na classe base não é `virtual`, ou se você cometer outro erro, o compilador pode te avisar.

```
class Professor : public Pessoa{
   public:

       //...

       virtual unsigned int getSalario() const;
   private:
       unsigned int valorHora;
       unsigned short cargaHoraria;
};
```

```
class ProfessorAdjunto : public Professor{
   public:

       //...

       virtual unsigned int getSalario() const override;
   private:
       std::string linhaPesquisa;
};
```

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), a especificação do C++ diz 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 não fazer isso (eu nunca consegui).

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;
```

Não precisa adicionar no .cpp.

Faça você mesmo

Modifique os destrutores da hierarquia de classes de `Pessoa` para `virtuais`.

Teste novamente o `main` (inclua `couts` nos destrutores).

```cpp
#include <iostream>

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

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

   return 0;
}
```

Construtores

Construtores **não** podem ser virtuais.

Virtual

Parece ser uma excelente ideia que toda função seja virtual.

De fato a ideia é tão boa, que em Java toda função é virtual por definição.

O programador não tem controle sobre isso.

Por que esse não é o comportamento padrão do C++?

Virtual

Funções virtuais são custosas.

- Adicionam indireções extras.
- **Overhead.**

Overhead

Funções virtuais são implementadas internamente via três níveis de indireção (ponteiros).

Os níveis são **ocultos e implementados pelo compilador.**

Mas você precisa conhecê-los para saber dos custos envolvidos.

Primeiro nível

Para **toda classe que possui ao menos uma função virtual**.

O compilador monta na memória uma tabela **virtual function table** – *vtable*.

Contém os endereços na memória das funções virtuais implementadas nessa classe.

Esse pode ser considerado o primeiro nível de ponteiros (indireção).

Segundo nível

Quando um objeto de uma classe que possui ao menos uma função virtual é **instanciado**.

O compilador adiciona internamente a esse objeto um **ponteiro que aponta para a vtable correta.**

Esse é considerado o segundo nível de ponteiros (indireção).

Terceiro nível

O terceiro nível é o próprio handle (ponteiro) do objeto.

Exemplo: `Professor* p{new ProfessorAdjunto};`

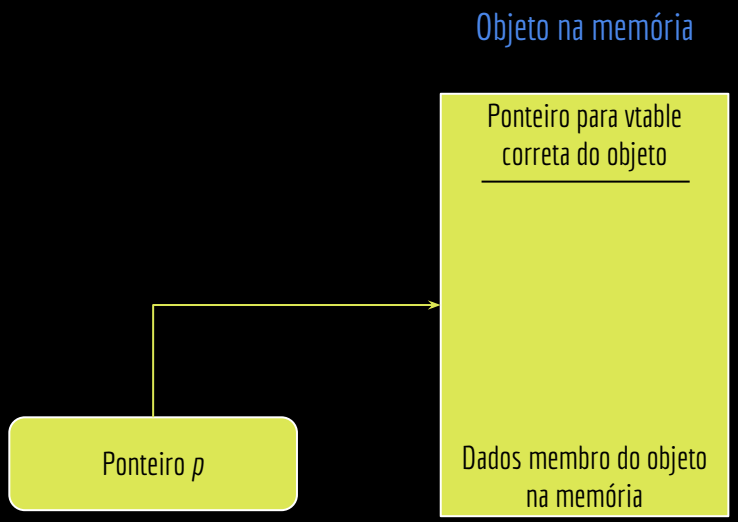
Exemplo de execução

```
int main(){
    Professor* p{new ProfessorAdjunto{"Maria", 11111111111, 100, 40}};
    std::cout << p->getSalario() << std::endl;
    delete p;
    return 0;
}
```

Ponteiro *p*

Exemplo de execução

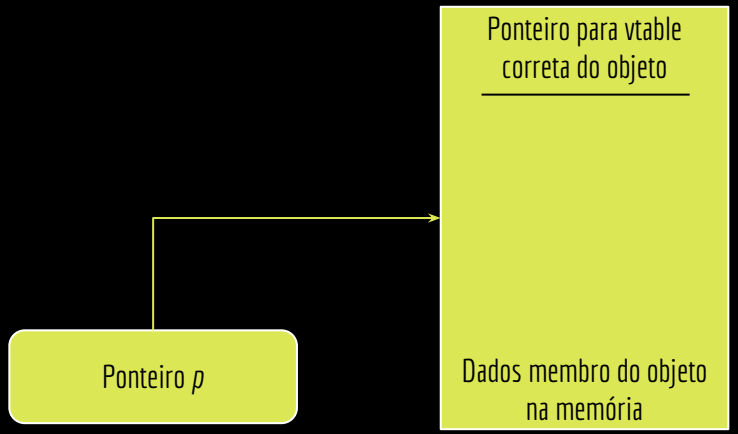
```
int main(){
    Professor* p{new ProfessorAdjunto{"Maria", 11111111111, 100, 40}};
    std::cout << p->getSalario() << std::endl;
    delete p;
    return 0;
}
```



Exemplo de execução

```
int main(){
    Professor* p{new ProfessorAdjunto{"Maria", 11111111111, 100, 40}};
    std::cout << p->getSalario() << std::endl;
    delete p;
    return 0;
}
```

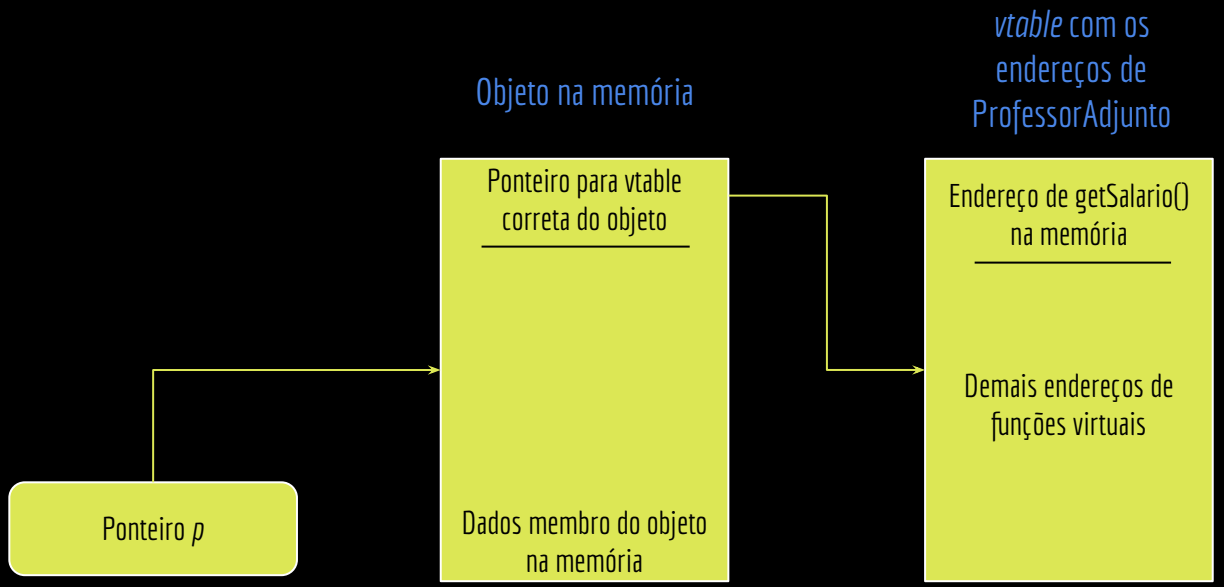
Objeto na memória



Até aqui o custo é o similar para uma função “comum”.

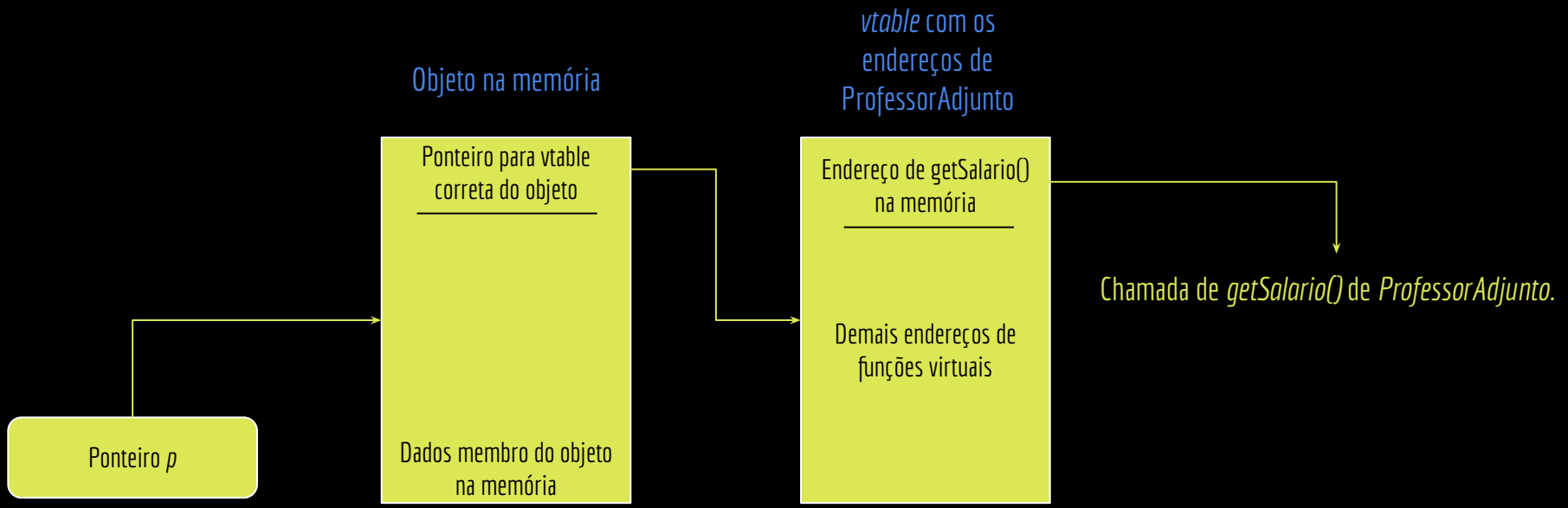
Exemplo de execução

```
int main(){
    Professor* p{new ProfessorAdjunto{"Maria", 11111111111, 100, 40}};
    std::cout << p->getSalario() << std::endl;
    delete p;
    return 0;
}
```



Exemplo de execução

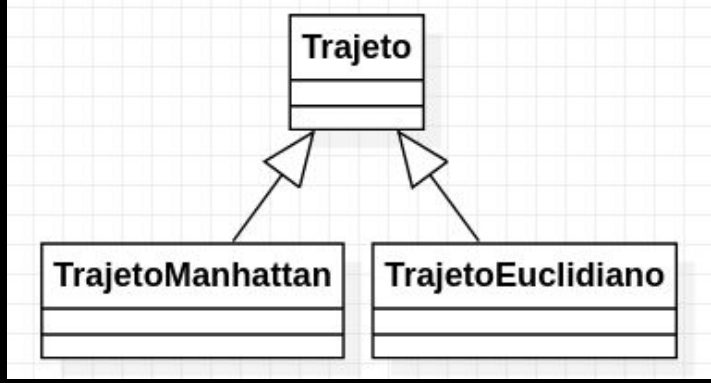
```
int main(){
    Professor* p{new ProfessorAdjunto{"Maria", 11111111111, 100, 40}};
    std::cout << p->getSalario() << std::endl;
    delete p;
    return 0;
}
```



Veja o vídeo

Assista o vídeo de demonstração: https://youtu.be/0Dv5gLIFT0w

Funções Puramente Virtuais



Muitas vezes declaramos **classes que não devem ser instanciadas.**

Classes incompletas.

Servem como base para outras classes, mas não deveriam ser usadas para gerar objetos.

Exemplo: a classe Trajeto do projeto disponibilizado.

Serve como base para classes como `TrajetoEuclidiano` e `TrajetoManhattan`.

Mas tentar calcular a distância de um trajeto com essa classe (Trajeto) é um erro.

Funções Puramente Virtuais

Podemos declarar uma **função puramente virtual** em uma classe base.

- Não damos implementação alguma para a função.

Uma classe que possui **uma ou mais funções puramente virtuais** é chamada de **classe abstrata**.

Não podemos criar instâncias de classes abstratas.

- Classes abstratas são consideradas classes incompletas.
- **Exigem que outras classes herdem** delas para que as implementações necessárias sejam concluídas.

Funções Puramente Virtuais

Para declarar uma função como puramente virtual, basta adicionar `= 0` no final da sua declaração no .hpp.

Não implementar no .cpp.

Isso inicializa o ponteiro de função interno para NULL.

```
class Trajeto{
   public:
       Trajeto();
       virtual ~Trajeto();

       void adicionarPonto(const double coordX, const double coordY);
       void imprimirTrajeto() const;
       double getDistanciaPercorrida() const;
   protected:
       virtual double calcularDistanciaPontos(const Ponto* const p1, const Ponto* const p2) const = 0;
   private:
       std::list<const Ponto*>* pontos;
};
```

Faça você mesmo

O que está errado?

Teste você mesmo.

```
#include <iostream>

#include "Trajeto.hpp"

int main(){
   Trajeto* t{new Trajeto};

   //..

   delete t;
   return 0;
}
```

Faça você mesmo

O que está errado?

Teste você mesmo.

Erro de compilação. Impossível instanciar uma classe abstrata.


```cpp
#include <iostream>

#include "Trajeto.hpp"

int main(){
    Trajeto* t{new Trajeto};

    //..

    delete t;
    return 0;
}
```

Classes abstratas

Ao declarar uma função puramente virtual.

- Você força as classes que herdam da classe base a implementarem a função.
  - Se a classe que herdar não implementar a função, ela também será abstrata.
- A sobrescrita não é opcional como em uma função virtual comum.
  - Em algum momento, alguma classe na hierarquia vai precisar implementar.

As classes que herdam da classe abstrata e implementam as funções puramente virtuais são **classes concretas**.

- **Você só pode criar instâncias de classes concretas.**

Exemplo

TrajetoEuclidiano.cpp

```cpp
#include "TrajetoEuclidiano.hpp"

#include <cmath>

double TrajetoEuclidiano::calcularDistanciaPontos(
            const Ponto* const p1, const Ponto* const p2) const{
   double dx {p1->getCoordX() - p2->getCoordX()};
   double dy {p1->getCoordY() - p2->getCoordY()};

   return std::sqrt(dx*dx + dy*dy);
}
```

main.cpp

```cpp
int main(){
   Trajeto* te{new TrajetoEuclidiano};
   te->adicionarPonto(1.0, 1.0);
   te->adicionarPonto(2.0, 2.0);
   te->adicionarPonto(3.0, 3.0);
   std::cout << "Distancia Euclidiana: "
            << te->getDistanciaPercorrida() << "\n";
   delete te;
   return 0;
}
```

TrajetoEuclidiano.hpp

```cpp
#ifndef TRAJETO_EUCLIDIANO_HPP
#define TRAJETO_EUCLIDIANO_HPP

#include "Trajeto.hpp"
#include "Ponto.hpp"

class TrajetoEuclidiano : public Trajeto{
   public:
      virtual ~TrajetoEuclidiano() = default;
   protected:
      virtual double calcularDistanciaPontos(const Ponto* const p1, const Ponto* const p2) const;
};
#endif
```

Classes abstratas e inversão de controle

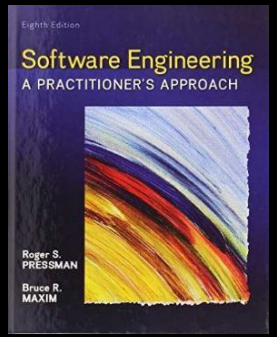
As classes do exemplo são um caso clássico de inversão de controle.

- Utilizado amplamente em frameworks.
- Relacionado a Inversão de dependência.

Leia sobre inversão de dependência, inversão de controle e injeção de dependência em livros de Engenharia de Software.

- Entenda como o uso de classes abstratas pode ser usado com esses conceitos.

Maxim, Pressman. Software Engineering: A Practitioner's Approach. McGraw-Hill. 2014.



Classes finais

- A partir do C++11:
  - Uma função pode ser declarada como `final` em seu protótipo;
    - Indica que a função não pode ser sobrescrita nas classes derivadas.
  - Exemplo:

```
someFunction(parameters) final;
```

- Uma classe pode ser declarada como final:
  - Indica que a classe não pode ser derivada.
  - Exemplo:

```
class MyClass final {
    //…
};
```

- Tentar herdar de uma classe final, ou sobrescrever uma função final, resulta em um erro de compilação.

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 geralmente melhor do ponto de vista da Eng. de Software

- Contra: O custo extra de uma função virtual é mandatório no Java.

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 geralmente 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++.

Também usa a palavra-chave virtual.

Curiosidades

Muitas classes da STL, como array e vector, são implementadas sem usar funções virtuais.

Não pagam o overhead.

Geram problemas caso você precise herdar dessas classes para modificar algum comportamento.

Engenharia de Software e Performance

- Do ponto de vista de engenharia de software, idealmente todas as funções deveriam ser virtuais.
  - Leia sobre o princípio **Open Closed**.

Engenharia de Software e Performance

- Do ponto de vista de engenharia de software, idealmente todas as funções deveriam ser virtuais.
  - Leia sobre o princípio **Open Closed**.
- Quanto ao custo computacional.
  - O seu compilador faz o possível para tentar resolver tudo em tempo de compilação e evitar indireções extras.
  - Mesmo com as indireções, **o custo é relativamente baixo**.
  - Ex.: um processador x86-64 possui mecanismos internos para tratar indireções.
    - Caches grandes (especialmente cache de instruções).
    - Mecanismos de predição de desvio sofisticados.

Engenharia de Software e Performance

- Do ponto de vista de engenharia de software, idealmente todas as funções deveriam ser virtuais.
  - Leia sobre o princípio **Open Closed**.
- Quanto ao custo computacional.
  - O seu compilador faz o possível para tentar resolver tudo em tempo de compilação e evitar indireções extras.
  - Mesmo com as indireções, **o custo é relativamente baixo**.
  - Ex.: um processador x86-64 possui mecanismos internos para tratar indireções.
    - Caches grandes (especialmente cache de instruções).
    - Mecanismos de predição de desvio sofisticados.
- **Remova as declarações virtuais em último caso.**
  - Geralmente você pode otimizar o desempenho de diversas outras formas antes de apelar para as funções não virtuais.
  - **Remova as chamadas virtuais apenas quando você não tiver mais opções e o requisito de desempenho for extremo.**

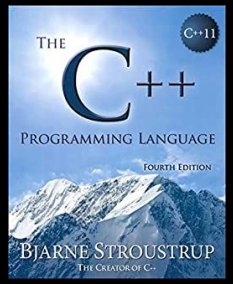
```
setApelationModeOn(true);
```

Exercícios

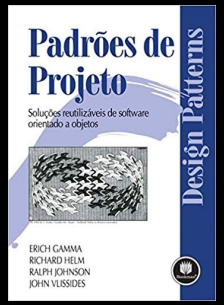
1. Declare os destrutores virtuais para todas as classes do Projeto.
2. Leia sobre inversão de dependência, inversão de controle e injeção de dependência. Dica: procure em artigos sérios ou livros. Muitos artigos da internet misturam inversão de controle com injeção de dependência, reflexão, ... em frameworks Java, Python, PHP, ... e fazem uma salada (vocês são Cientistas da Computação, não programadores Java, PHP, C++, ...).
3. 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 puramente virtual `calcularDistanciaPontos`, que é usada pela classe.
    a. 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.
    b. Utilize funções virtuais onde necessário.
    c. Crie instâncias de objetos de DistanhaManhattan e DistanciaEuclidiana no main para testar. Chame a função calcularDistanciaPontos a partir dessas instâncias.
    d. Crie uma classe de Console que imprime os dados de um trajeto. A classe deve ter uma função para imprimir os pontos por onde a pessoa passou, e a distância total percorrida.
    e. Os conceitos aplicados se referem a uma inversão de controle ou injeção de dependência?
4. Pesquise sobre o princípio Open Closed.

Referências

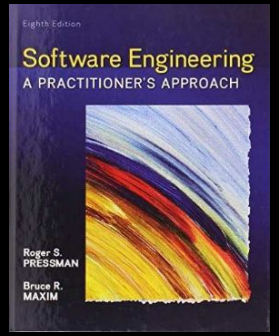
Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2013.



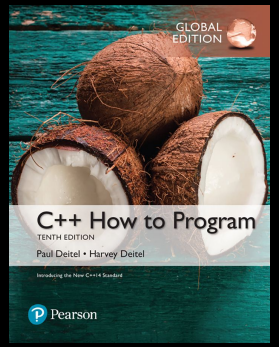
Gamma, E. Padrões de Projetos: Soluções Reutilizáveis. Bookman. 2009.



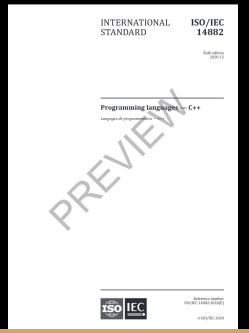
Maxim, Pressman. Software Engineering: A Practitioner's Approach. McGraw-Hill. 2014.



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



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



Licença