INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
PARAÍBA
Campus Cajazeiras

**Engenharia de Controle e Automação**
**Programação Orientada a Objetos – 2021.1**

**Professor:** Leandro Luttiane da Silva Linhares
**E-mail**: leandro.luttiane@ifpb.edu.br

# Documento Texto
# Polimorfismo

Na semana anterior da disciplina de **Programação Orientada a Objetos (POO)** do **Curso de Engenharia de Controle e Automação**, foi dado início ao aprendizado do conceito de **Herança de Classes**, um dos pilares fundamentais do paradigma de programação orientada a objetos. Em nossos estudos, foi evidenciado que características comuns existentes entre um conjunto de classes podem ser exploradas por meio da herança. Neste sentido, uma classe chamada derivada, ou classe filha, herda estas características comuns de uma classe básica, ou classe mãe.

Entre outras coisas, a herança favorece o desenvolvimento de código eficiente, sem a necessidade de replicação de trechos idênticos de código em diferentes classes, fazendo com que as classes e programas possuam uma quantidade menor de linhas de instrução. Além disso, a herança contribui em muito com a manutenção de código, pois uma alteração/correção feita em uma classe básica irá refletir de forma imediata nas classes derivadas que fazem uso das propriedades e funcionalidades dessa classe mãe.

Nesta semana, continuaremos nossos estudos de POO abordando o uso do polimorfismo, conceito relacionado com o princípio de hierarquias de herança. Neste documento, veremos que o polimorfismo permite o desenvolvimento de programas que lidam com objetos de classes que fazem parte da mesma hierarquia de classes como se todos fossem objetos da classe básica dessa hierarquia.

Antes de iniciarmos, é importante comentar que o conteúdo apresentado neste documento tem como base o ***Capítulo 13 do livro C++: Como Programar***, **de Deitel e Deitel (2006)**. Em conjunto com este texto, também estão sendo disponibilizados exemplos extraídos do livro mencionado. Sugere-se fortemente que estes arquivos sejam abertos quando as suas linhas de código forem ser analisadas por este material didático.

## 1. Introdução

Para que vocês possam compreender melhor o princípio do polimorfismo, antes mesmo de partirmos para a análise de código, vamos considerar um exemplo simples. Suponham que um programa foi criado para simular o movimento de diferentes tipos de animais. Neste sentido, as seguintes classes foram criadas: `Peixe`, `Anfibio` e `Passaro`, os três tipos de animais em análise pelo programa.

Por meio de herança, estas classes são derivadas da classe básica `Animal`, que possui um método `mover` (função membro) e contém o registro da localização atual (atributo ou membro de dados) de um animal. Toda classe derivada implementa a sua versão do método `mover`. O programa desenvolvido possui um **vetor de ponteiros** para objetos das várias classes derivadas da classe básica `Animal`.

Para simular os movimentos dos animais, o programa envia a mesma mensagem/comando `mover` a cada objeto uma vez por segundo. Entretanto, cada tipo de `Animal` irá responder a essa mensagem de uma forma diferente, pois são animais diferentes. Por exemplo, um `Peixe` poderia nadar dois metros, um `Sapo` poderia pular três metros e um `Passaro` voar dez metros.

Observem que uma mensagem/comando (`mover`) é emitida para cada objeto `Animal` de uma forma genérica, mas cada objeto irá modificar a sua posição de acordo com as características específicas de sua espécie. A mesma chamada do método `mover` aplicada a uma variedade de objetos de uma hierarquia de classes possui **muitas formas** de execução/resultados, uma vez que cada classe derivada executa o método `mover` de uma **forma diferente**. Esta é a razão do uso do termo **polimorfismo**.

## 2. Um exemplo de polimorfismo

Pelo que já vimos de forma introdutória, podemos afirmar que o polimorfismo permite que a chamada de um determinado método ou função-membro dê origem à execução de diferentes ações, dependendo do tipo do objeto que deu origem a esta chamada.

Por exemplo, se a classe `Retangulo` é derivada da classe `Quadrilatero`, então um objeto `Retangulo` é uma versão mais específica de um objeto `Quadrilatero`. Desta forma, qualquer operação que possa ser realizada em um objeto da classe básica `Quadrilatero` também pode ser realizada por um objeto da classe derivada `Retangulo`, como calcular o perímetro ou a área. Estas mesmas operações também poderão ser realizadas em outros tipos de `Quadrilateros`, tais como `Quadrados`, `Paralelogramos` e `Trapezoides`. Entretanto, a execução destas operações pode ser realizada de forma diferente, dependendo da classe derivada que o objeto pertença.

De uma maneira mais técnica, o polimorfismo acontece quando um determinado programa efetua a chamada de uma **função/método virtual** por meio de um **ponteiro ou referência** de **classe básica** (por exemplo, `Quadrilatero`) e o C++ em tempo de execução determina a chamada correta do método para a classe a partir da qual o objeto foi instanciado. A princípio, tudo isso pode parecer um pouco confuso, mas os exemplos que serão apresentados neste documento deixarão o conceito de polimorfismo bastante claro.

## 3. Interação entre objetos em uma hierarquia de herança

Na semana anterior, analisamos uma hierarquia simples de classes de empregados. Trabalhamos com a ideia de um tipo de empregado que tem seus rendimentos calculados a partir

de um salário base e sua comissão de vendas e outro tipo cujos rendimentos têm origem apenas em sua comissão de vendas. Estes dois tipos de empregados deram origem, respectivamente, à classe derivada `BasePlusCommissionEmployee` que herda características da classe básica `CommissionEmployee`.

Nos exemplos da semana anterior, utilizamos os nomes dos próprios objetos para efetuar as chamadas de seus métodos ou funções-membro. Neste momento de nosso estudo, iremos analisar mais a fundo o relacionamento entre classes em uma hierarquia com o auxílio do uso de **ponteiros**. Para isso, os seguintes exemplos serão analisados:

1. No primeiro exemplo, será atribuído o endereço de um objeto de classe derivada a um ponteiro de classe básica. A partir disso, será demonstrado que chamar um método por meio de um ponteiro de classe básica invoca as funcionalidades da classe básica. Veremos, que neste caso é o `handle` que determina qual método será efetivamente chamado;

2. No segundo exemplo, será atribuído o endereço de um objeto de classe básica a um ponteiro de classe derivada. Veremos que isto irá ocasionar um erro de compilação, pois um objeto da classe básica **não é um** objeto da classe derivada;

3. No terceiro exemplo, será atribuído o endereço de um objeto de classe derivada a um ponteiro de classe básica. Veremos que o ponteiro de classe básica pode ser utilizado para executar apenas as funcionalidades da classe básica. A tentativa de chamar métodos da classe derivada pelo ponteiro de classe básica irá gerar erros de compilação;

4. Finalmente, veremos o que são as **funções/métodos virtuais** e implementaremos o **polimorfismo** declarando métodos da classe básica como `virtual`. Para exemplificar, será atribuído o endereço de um objeto de classe derivada ao ponteiro de classe básica e utilizaremos esse ponteiro para chamar as funcionalidades de classe derivada.

Um ponto fundamental nesses exemplos é demonstrar que um objeto de uma classe derivada pode ser tratado como um objeto de sua classe básica. Isso oferece MUITA flexibilidade no desenvolvimento de nossos programas. É possível, por exemplo, criar um **vetor de ponteiros** de uma classe básica que aponta para objetos de muitos tipos de classes derivadas. Apesar de os objetos de classes derivadas serem de tipos diferentes, o compilador permite isso porque cada objeto de classe derivada **é um** objeto de sua classe básica. Entretanto, deve-se ter em mente que não é possível tratar um objeto de classe básica como um objeto de qualquer uma de suas classes derivadas.

Um objeto da classe `CommissionEmployee` **não é um** `BasePlusCommissionEmployee`. Ele não possui um atributo `baseSalary` e nem os métodos `setBaseSalary` e `getBaseSalary`. O relacionamento **é um** somente se aplica de uma classe derivada para suas classes básicas. Portanto, um objeto `BasePlusCommissionEmployee` **é um** `CommissionEmployee`.

### 3.1 Primeiro exemplo

O primeiro exemplo apresenta três maneiras distintas de apontar ponteiros de classe básica e ponteiros de classe derivada em objetos de classe básica e de classe derivada. As duas primeiras são simples e diretas: utilizamos um ponteiro de classe básica que aponta para um objeto de classe básica e fazemos uso de funcionalidades disponibilizadas pela classe básica; e um ponteiro de classe derivada que aponta para um objeto de classe derivada e fazemos uso de funcionalidades disponibilizadas pela classe derivada. Logo após, o relacionamento **é um** existente entre as classes derivada e básica é demonstrado por meio de um ponteiro de classe básica que aponta para um objeto de classe derivada. A partir disso, fazemos uso das funcionalidades da classe básica que estão, de fato, também disponíveis no objeto de classe derivada.

Nos exemplos deste documento de texto, serão utilizadas as classes estudadas em nossa última semana de aula: `CommissionEmployee` e `BasePlusCommissionEmployee`. Já vimos que todo objeto `BasePlusCommissionEmployee` **é um** `CommissionEmployee` que possui um salário base. Por esta razão, alguns métodos da classe `CommissionEmployee` foram redefinidos em sua classe derivada.

O método `earnings` da classe `BasePlusCommissionEmployee` (linhas 32–25 do arquivo de implementação desta classe, `BasePlusCommissionEmployee.cpp`) redefine o método `earnings` da classe `CommissionEmployee` (linhas 79–82 de seu arquivo de implementação) para incluir o salário-base do objeto. Da mesma maneira, o método `print` da classe `BasePlusCommissionEmployee` (linhas 38–46 de seu arquivo de implementação) redefine o método `print` da classe `CommissionEmployee` (linhas 85–92 de seu arquivo de implementação) para exibir as mesmas informações que o método `print` da classe `CommissionEmployee`, bem como o salário-base do empregado.

Feitas estas considerações, vamos analisar em detalhes o arquivo principal deste exemplo, o arquivo `mainPolimorfismo.cpp`. As linhas 19–20 criam um objeto `CommissionEmployee` e a linha 23 cria um ponteiro para um objeto `CommissionEmployee`. As linhas 26–27 criam um objeto `BasePlusCommissionEmployee` e a linha 30 cria um ponteiro para um objeto `BasePlusCommissionEmployee`. As linhas 37 e 39 utilizam o nome de cada objeto (`commissionEmployee` e `basePlusCommissionEmployee`, respectivamente) para efetuar a chamada do método `print` específico de cada uma das classes a qual o objeto pertence.

A linha 42 atribui o endereço do objeto da classe básica `commissionEmployee` ao ponteiro de classe básica `commissionEmployeePtr`, que a linha 45 utiliza para efetuar a chamada da função-membro `print` nesse objeto `CommissionEmployee`. Isso efetua a chamada da versão de `print` definida na classe básica `CommissionEmployee`. De modo semelhante, a linha 48 atribui o endereço do objeto da classe derivada `basePlusCommissionEmployee` ao ponteiro de classe derivada `basePlusCommissionEmployeePtr`, que a linha 52 utiliza para efetuar a chamada do método `print` nesse objeto `BasePlusCommissionEmployee`. Este procedimento chama a versão de `print` definida na classe derivada `BasePlusCommissionEmployee`.

A linha 55 atribui o endereço de objeto de classe derivada `basePlusCommissionEmployee` ao ponteiro de classe básica `commissionEmployeePtr`, que a linha 59 utiliza para efetuar a chamada do método `print`. O compilador C++ permite que isso ocorra porque um objeto de uma classe derivada **é um** objeto de sua classe básica. Observem pela saída do programa que, apesar de o ponteiro de classe básica `CommissionEmployee` apontar para um objeto `BasePlusCommissionEmployee` de classe derivada, o método `print` da classe básica `CommissionEmployee` é chamado, ao invés do método `print` da classe derivada `BasePlusCommissionEmployee`.

```
Print base-class and derived-class objects:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00


Calling print with base-class pointer to
base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06


Calling print with derived-class pointer to
derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00


Calling print with base-class pointer to derived-class object
invokes base-class print function on that derived-class object:

commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
```

Sugere-se que o código seja compilado e o resultado da saída da execução do arquivo principal `mainPolimorfismo.cpp` seja analisado juntamente com o que foi previamente explicado. A saída de cada chamada da função-membro `print` revela que o método chamado depende do tipo do `handle` (**ponteiro ou referência**) utilizado para chamar o método e não do tipo do objeto para o qual o `handle` aponta.

### 3.2 Segundo exemplo

No primeiro exemplo, atribuímos o endereço de um objeto de classe derivada a um **ponteiro** de classe básica. Isso é permitido pelo compilador C++, porque um objeto de classe derivada **é um** objeto de classe básica.

Iremos agora seguir um caminho diferente do que foi realizado no exemplo anterior: um ponteiro de classe derivada irá apontar para um objeto de classe básica. As classes `CommissionEmployee` e `BasePlusCommissionEmployee` serão novamente utilizadas, mas o arquivo principal que iremos analisar neste exemplo é o `mainPolimorfismoErro1.cpp`.

As linhas 8–9 do arquivo `mainPolimorfismoErro1.cpp` criam um objeto `CommissionEmployee` e a linha 10 cria um ponteiro `BasePlusCommissionEmployee`. A linha 14 tenta atribuir o endereço de objeto de classe básica `commissionEmployee` ao ponteiro de classe derivada `basePlusCommissionEmployeePtr`, entretanto, isso fará com que o compilador C++ gere um erro durante a etapa de compilação.

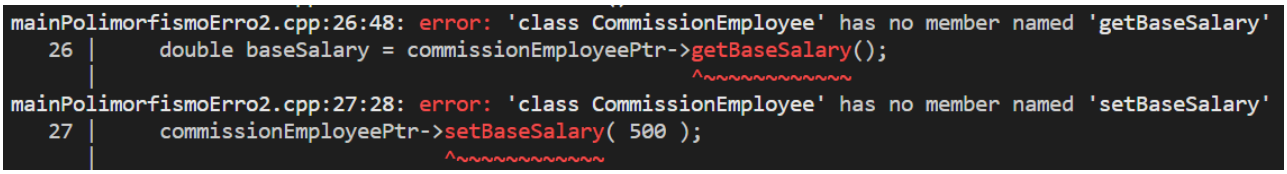
```
mainPolimorfismoErro1.cpp: In function 'int main()':
mainPolimorfismoErro1.cpp:14:37: error: invalid conversion from 'CommissionEmployee*' to 'BasePlusCommissionEmployee*' [-fpermissive]
   14 |     basePlusCommissionEmployeePtr = &commissionEmployee;
      |                                     ^~~~~~~~~~~~~~~~~~~
      |                                     |
      |                                     CommissionEmployee*
```

O compilador impede essa atribuição, porque um `CommissionEmployee` **não é um** `BasePlusCommissionEmployee`. Vamos analisar um possível problema que poderia ser gerado caso o compilador permitisse que essa atribuição fosse realizada.

Por meio de um ponteiro `BasePlusCommissionEmployee`, podemos chamar cada método da classe `BasePlusCommissionEmployee`, inclusive `setBaseSalary`, do objeto para o qual o ponteiro aponta (isto é, o objeto de classe básica `commissionEmployee`). Entretanto, o objeto `CommissionEmployee` não fornece um método `setBaseSalary` e nem um atributo `baseSalary` para configurar. Isso poderia resultar em problemas, porque o método `setBaseSalary` iria supor que há um membro de dados `baseSalary` a configurar em sua 'localização normal' em um objeto `BasePlusCommissionEmployee`. Essa memória não pertence ao objeto `CommissionEmployee`, então a função-membro `setBaseSalary` poderia sobrescrever outros dados importantes na memória, possivelmente dados que pertencem a um objeto diferente.

### 3.3 Terceiro exemplo

É importante termos em mente que a partir de um ponteiro de classe básica, o compilador C++ permite que sejam chamadas apenas métodos de classe básica. Assim, se um ponteiro de classe básica apontar para um objeto de classe derivada e for realizada uma tentativa de acessar um método exclusivo da classe derivada, ocorrerá um erro de compilação. É isso que comprovaremos a partir deste exemplo.




Novamente iremos utilizar a classe básica `CommissionEmployee` e sua classe derivada `BasePlusCommissionEmployee`, sem realizar nenhuma alteração em seus códigos. A modificação ocorrerá no arquivo principal de nosso exemplo, que agora chamaremos de `mainPolimorfismoErro2.cpp`.

A linha 9 do arquivo `mainPolimorfismoErro2.cpp` cria `commissionEmployeePtr` — um ponteiro para um objeto `CommissionEmployee` — e as linhas 10–11 criam um objeto `BasePlusCommissionEmployee`. A linha 14 aponta `commissionEmployeePtr` para o objeto de classe derivada `basePlusCommissionEmployee`. Lembrem-se que isso é permitido pelo compilador C++, pois um `BasePlusCommissionEmployee` **é um** `CommissionEmployee`.

As linhas 18–22 efetuam as chamadas das funções-membro de classe básica `getFirstName`, `getLastName`, `getSocialSecurityNumber`, `getGrossSales` e `getCommissionRate` a partir do ponteiro de classe básica. Todas essas chamadas são legítimas, porque `BasePlusCommissionEmployee` herda esses métodos de `CommissionEmployee`.

Sabemos que `commissionEmployeePtr` está apontando para um objeto `BasePlusCommissionEmployee`, então nas linhas 26–27 tentamos chamar os métodos `getBaseSalary` e `setBaseSalary` da classe `BasePlusCommissionEmployee`. Neste caso, o compilador C++ irá gerar erros no ato da compilação de nosso programa, porque estas linhas de código efetuam a chamada de métodos que não fazem parte da classe básica `CommissionEmployee`. O `handle` pode chamar somente aqueles métodos que são membros do tipo de classe associada a esse `handle`. Ou seja, a partir do ponteiro `CommissionEmployee *`, é possível chamar somente os métodos definidos na classe `CommissionEmployee`.

O compilador C++ permite acesso a membros exclusivos de classe derivada a partir de um ponteiro de classe básica que aponta para um objeto de classe derivada se explicitamente fizermos a coerção (`cast`) do ponteiro de classe básica para um ponteiro de classe derivada — uma técnica conhecida como *downcasting*.

É possível apontar um ponteiro de classe básica para um objeto de classe derivada. Entretanto, um ponteiro de classe básica pode ser utilizado para chamar apenas os métodos declarados na classe básica. O *downcasting* permite a um programa realizar uma operação específica de classe derivada em um objeto de classe derivada apontado por um ponteiro de classe básica. Depois de um *downcast*, o programa pode chamar métodos da classe derivada que não estão na classe básica.

## 4. Funções virtuais (métodos virtuais)

Em nosso primeiro exemplo apontamos um ponteiro da classe básica `CommissionEmployee` para um objeto da classe derivada `BasePlusCommissionEmployee` e efetuamos a chamada do método `print` por meio desse ponteiro. Nesse caso, o tipo do `handle` é que determina de qual classe o método será utilizado. Portanto, o ponteiro `CommissionEmployee` efetua a chamada da função-membro `print` da classe `CommissionEmployee` sobre o objeto `BasePlusCommissionEmployee`, embora este ponteiro aponte para um objeto `BasePlusCommissionEmployee` que tem sua própria função `print` personalizada. Com o uso das funções `virtual`, o **tipo do objeto apontado** e não o tipo do `handle` é quem determina qual versão de uma **função virtual** será chamada.

Vamos utilizar um exemplo para compreender as razões pelas quais as funções virtuais são úteis. Suponham quem um conjunto de classes de formas geométricas como `Circulo`, `Triangulo`, `Retangulo` e `Quadrado` são todas derivadas da classe básica `Forma`. A capacidade de desenhar a si própria por meio de um método `desenhar` poderia ser fornecida a cada uma dessas classes. Embora cada classe tenha o seu próprio método `desenhar`, o método para desenhar cada forma é bem diferente.

Em um programa que desenha um conjunto de formas, seria útil poder tratar todas as formas genericamente como objetos da classe básica `Forma`. Então, para desenhar qualquer forma, poderíamos simplesmente utilizar um ponteiro da classe básica `Forma` para chamar o método `desenhar` e deixar o programa determinar dinamicamente (em tempo de execução) qual método `desenhar` de classe derivada utilizar, com base no tipo do objeto para o qual o ponteiro da classe básica `Forma` aponta em um determinado momento. Para que tudo isso seja possível, é preciso que a função-membro `desenhar` seja declarada como uma função virtual na classe básica e que ela seja sobrescrita em cada uma das classes derivadas para desenhar a forma apropriada.

Da perspectiva da implementação, **sobrescrever** uma função-membro, ou método, não é diferente de **redefinir** um método. Um método sobrescrito em uma classe derivada tem o mesmo protótipo (assinatura e tipo de retorno) que o método que ele sobrescreve de sua classe básica. Se o método não for declarado na classe básica como uma função virtual, seremos capazes de **redefinir** este método/função-membro. Em contrapartida, se o método for declarado como uma função virtual, poderemos sobrescrevê-lo com o objetivo de ativar o comportamento polimórfico.

Uma função-membro virtual (ou método virtual) é declarada precedendo o protótipo da função com a palavra-chave `virtual` na classe básica. Por exemplo,

```
virtual void desenhar() const;
```

apareceria na classe básica `Forma`. O protótipo apresentado declara que a função `desenhar` é um método virtual que não aceita argumentos e não possui valor de retorno. Este método é declarado `const` porque, em geral, a função-membro `desenhar` não faria alterações no objeto `Forma` que

efetua a chamada deste método. É importante comentar que as funções virtuais não têm necessariamente de ser funções `const`.

Se o programa efetuar a chamada de uma função virtual por meio de um ponteiro de classe básica para um objeto de classe derivada, por exemplo, `formaPtr->desenhar()`, o programa escolherá dinamicamente, em tempo de execução, a função `desenhar` da classe derivada correta com base no tipo de objeto apontado e não no tipo do ponteiro. Escolher o método apropriado para a chamada em tempo de execução, em vez de em tempo de compilação, é conhecido como **vinculação dinâmica** ou **vinculação tardia**.

Quando uma função virtual é chamada referenciando um objeto específico por nome e utilizando o operador de seleção de membro ponto, por exemplo, `objetoQuadrado.desenhar()`, a chamada de função é convertida em tempo de compilação (isso é denominado **vinculação estática**) e a função virtual que é chamada é aquela definida para (ou herdada pela) a classe desse objeto particular — esse não é um comportamento polimórfico. Portanto, a vinculação dinâmica com funções virtual ocorre somente a partir de `handles` de **ponteiro ou de referência**.

Podemos fazer algumas observações em relação ao uso das funções virtuais. Uma vez que uma função-membro (ou método) de uma classe é declarada virtual, ela permanece virtual por toda a hierarquia de herança a partir desse ponto, mesmo que essa função não seja declarada explicitamente como virtual quando uma classe a sobrescreve.

Mesmo que uma função seja implicitamente virtual por causa de uma declaração feita em um ponto mais alto da hierarquia de classes, é uma boa prática de programação declarar explicitamente essa função-membro virtual em cada nível da hierarquia para que o programa se torne mais claro e de fácil compreensão. Quando uma classe derivada não sobrescreve um método virtual de sua classe básica, a classe derivada simplesmente herda a implementação do método virtual de sua classe básica.

Vejamos agora como o comportamento polimórfico pode ser implementado na nossa hierarquia de classes de empregados a partir do uso das funções/métodos virtuais. No interior da pasta `FuncoesVirtuais`, podem ser encontrados os códigos-fontes (arquivos de cabeçalho e de implementações) das classes `CommissionEmployee` e `BasePlusCommissionEmployee`. Mais uma vez, sugere-se fortemente que os arquivos sejam abertos e compilados. Da mesma maneira, recomenda-se que os arquivos de código sejam utilizados para acompanhar as explicações a seguir e que o programa executável gerado após a compilação seja executado para verificar as saídas produzidas pelas linhas de código de seu arquivo principal (`mainPolimorfismo.cpp`).

As diferenças entre os arquivos de código destas classes e os respectivos arquivos apresentados nos exemplos iniciais são encontradas nos arquivos de cabeçalho `CommissionEmployee.h` e `BasePlusCommissionEmployee.h`. Observem que os métodos `earnings` e `print` de cada classe são especificadas como virtual (linhas 30–31 do arquivo `CommissionEmployee.h` e linhas 21–22 do arquivo `BasePlusCommissionEmployee.h`). Como as funções-membro `earnings` e `print` são virtuais na classe `CommissionEmployee`, as funções

`earnings` e `print` da classe `BasePlusCommissionEmployee` **sobrescrevem** as da `CommissionEmployee`.

Com estas pequenas mudanças, se apontarmos um ponteiro da classe básica `CommissionEmployee` para um objeto da classe derivada `BasePlusCommissionEmployee` e o programa usar esse ponteiro para chamar o método `earnings` ou `print`, o método correspondente do objeto `BasePlusCommissionEmployee` será chamado. **Nenhuma alteração** foi realizada nos arquivos de implementações das classes `CommissionEmployee` e `BasePlusCommissionEmployee`.

O arquivo principal `mainPolimorfismo.cpp` possui linhas de código que deixam claro as potencialidades do uso das funções virtuais para implementar o polimorfismo em nossos programas. As linhas 46–57 demonstram, mais uma vez, que um ponteiro `CommissionEmployee` apontado para um objeto `CommissionEmployee` pode ser utilizado para chamar a funcionalidade da classe `CommissionEmployee`, e um ponteiro `BasePlusCommissionEmployee` apontado para um objeto `BasePlusCommissionEmployee` pode ser utilizado para chamar funcionalidades da classe `BasePlusCommissionEmployee`.

A linha 60 aponta o ponteiro de classe básica `commissionEmployeePtr` para objeto de classe derivada `basePlusCommissionEmployee`. Observem que, quando a linha 67 efetua a chamada do método `print` a partir do ponteiro de classe básica, o método `print` da classe derivada `BasePlusCommissionEmployee` é chamado. Isso comprova que declarar o método como virtual faz com que o programa determine dinamicamente que método chamar com base no tipo de objeto para o qual o `handle` aponta, em vez do tipo do `handle`. Esta tomada de decisão sobre qual método chamar é um exemplo de **polimorfismo**.

Verifiquem que quando `commissionEmployeePtr` aponta para um objeto `CommissionEmployee` (linha 46 do arquivo `mainPolimorfismo.cpp`), o método `print` da classe `CommissionEmployee` é chamado. Por outro lado, quando `CommissionEmployeePtr` aponta para um objeto `BasePlusCommissionEmployee`, o método `print` da classe `BasePlusCommissionEmployee` é chamado. Portanto, é possível notar um comportamento polimórfico na chamada do método `print`, dependendo do tipo do objeto apontado pelo ponteiro (`handle`).

## 5. Resumo

Apesar de um objeto de classe derivada também ser um objeto da classe básica, os dois objetos são, apesar disso, diferentes. Os objetos de classe derivada podem ser tratados como se fossem objetos da classe básica. Afinal, a classe derivada contém todos os membros da classe básica. Entretanto, os objetos da classe básica não podem ser tratados como se fossem de classe derivada. A classe derivada pode ter membros exclusivos de classes derivadas adicionais. Por essa razão, não é permitido apontar um ponteiro de classe derivada para um objeto de classe básica sem uma coerção (`cast`) explícita. A coerção alivia o compilador da responsabilidade de emitir uma

mensagem de erro. De certo modo, utilizando a coerção você está dizendo: “sei que o que estou fazendo é perigoso e assumo toda a responsabilidade por minhas ações”.

Podemos fazer um resumo do que foi visto nos exemplos apresentados neste documento. Neles, discutimos quatro maneiras de apontar ponteiros de classe básica e ponteiros de classe derivada para objetos de classe básica e objetos de classe derivada:

1. Apontar um ponteiro de classe básica para um objeto de classe básica é simples e direto — as chamadas feitas a partir do ponteiro de classe básica simplesmente invocam as funcionalidades da classe básica;

2. Apontar um ponteiro de classe derivada para um objeto de classe derivada é simples e direto — as chamadas feitas a partir do ponteiro de classe derivada simplesmente invocam as funcionalidades de classe derivada;

3. É seguro apontar um ponteiro de classe básica para um objeto de classe derivada, porque o objeto de classe derivada é um objeto de sua classe básica. Entretanto, esse ponteiro pode ser utilizado para chamar apenas os métodos da classe básica. Se o programador tentar referenciar um membro exclusivo da classe derivada por meio do ponteiro de classe básica, o compilador irá gerar uma mensagem de erro. Para evitar esse erro, o programador deve pode realizar a coerção do ponteiro de classe básica para um ponteiro de classe derivada, em uma técnica que pode ser perigosa e é conhecida como *downcasting*.

4. Apontar um ponteiro de classe derivada para um objeto de classe básica gera um erro de compilação. O relacionamento **é um** se aplica apenas de uma classe derivada para suas classes básicas, e não vice-versa. Um objeto de classe básica não contém membros exclusivos de classe derivada que podem ser chamados a partir de um ponteiro de classe derivada.

**REFERÊNCIAS**

DEITEL, H. M.; DEITEL, P. J. **C++: como programar**. 5 ed. São Paulo: Pearson Prentice Hall, 2006.