Introdução a Orientação a Objetos[1][2]

Parte IV: Polimorfismo – I/O

A orientação a objetos (OO) é um paradigma de programação que se baseia em 03 (três) pilares. Neste roteiro, você completará seu estudo prático sobre o segundo pilar – o polimorfismo. Para isto, utilizaremos a biblioteca padrão do `C++`, a `STL` (*Standand Template Library*). Especificamente, neste roteiro utilizaremos as classes relacionadas a I/O em arquivos e strings, através de exemplos, exercícios de implementação e da referência fornecida pelo site `cplusplus.com`.

1 Introdução conceitual

O polimorfismo é uma propriedade de linguagens de programação que permite que uma mesma entidade apresente diferentes comportamentos. Em `C++`, há quatro diferentes tipos de polimorfismo, dentre os quais dois já foram estudados em roteiros anteriores. Neste roteiro, você estudará os dois tipos restantes de polimorfismo em `C++`:

Coerção – corresponde ao polimorfismo que permite a conversão entre tipos pré-definidos. Em `C++`, são exemplos de coerção a conversão de `float` pra `int` e de `array` para pointeiro. No entanto, o exemplo mais importante é a coerção usando o tipo genérico `void *`, que permite conversões para qualquer tipo de ponteiro.

Subtipo – corresponde ao polimorfismo característico da orientação a objetos, apresentando três aspectos principais:

1. *Coerção*: classes são consideradas tipos e o mecanismo da herança define relações entre classes base e derivadas. Efetivamente, classes derivadas se tornam subtipos do tipo definido pela classe base. Neste contexto, `C++` permite operações de coerção entre um tipo e seus subtipos, isto é, um objeto da classe tipo pode se tornar um objeto do subtipo e vice-versa.
2. *Sobreposição de métodos:* classes derivadas podem redefinir o comportamento da classe base, sobrepondo seus métodos (*method overriding*). Especificamente, a classe derivada deve prover uma nova implementação do método da classe base que deseja redefinir (sobrepor). Em `C++`, é possível fazer distinção entre as diferentes definições de um método usando o operador de escopo. No entanto, um objeto instanciado a partir da classe derivada que tenha sofrido coerção para a classe base só consegue usar seus métodos sobrepostos caso os métodos originais da classe base sejam definidos como virtuais, usando a palavra reservada `virtual` antes do tipo de retorno do método.
3. *Classes abstratas:* um conceito da orientação a objetos que expande a noção de sobreposição de métodos é o conceito das classes abstratas. Diz-se que uma classe é abstrata quando ela apresenta pelo menos um método definido como puramente virtual, isto é, um método que não apresenta definição base e que obriga as classes derivadas a o definirem de forma personalizada. No `C++`, um método

[1] Roteiro de estudo fornecido na disciplina de Linguagem de Programação 1 (LP1) da Universidade Federal do Rio Grande do Norte (UFRN). Disponível em `https://leobezerra.github.io/LP1-2017-1-T05`.

[2] Autor: Leonardo Bezerra (`leobezerra@imd.ufrn.br`).

puramente virtual é definido atribuindo o valor 0 ao protótipo do método na classe base. Note que, por não apresentar definição para os métodos puramente virtuais, não é possível instanciar um objeto de uma classe abstrata. No entanto, é possível usar ponteiros e referências para classes abstratas e apontar para objetos de classes derivadas por meio de coerção. A principal aplicação de classes abstratas é a definição de interfaces, isto é, um conjunto de métodos que classes derivadas devam apresentar caso queiram pertencer a determinado tipo (classe base).

2 Prática STL: arquivos e strings

As classes de entrada/saída (input/output, ou I/O) do `C++` formam uma estrutura ampla e flexível, baseada predominantemente em herança. Para aumentar sua compreensão destas classes, neste roteiro você irá estudar mais a fundo os objetos de entrada e saída usados para representar streams de arquivos e strings do `C++`. Para isto, acesse a referência disponibilizada no GitHub Pages da disciplina e faça o exercício sugeridos abaixo. Note que você deverá criar dois novos projetos no GitLab para este exercício (`lab10` e `lab11`), sendo o segundo projeto extra.

(***lab10***) **Matrizes** – no roteiro anterior, você implementou uma classe para encapsular matrizes e efetuar leitura e escrita de objetos desta classe de maneira formatada usando os operadores de inserção e extração. Neste roteiro, você deverá reimplementar os operadores de inserção e extração, utilizando pra isso as classes de stream de strings definidas na biblioteca `sstream`. Em seguida, você deverá definir parâmetros de linha de comando para que o usuário escolha entre trabalhar com a entrada/saída padrão ou com arquivos. Especificamente, a flag `--std` ativa o uso da entrada e saída padrão, sendo usada como ilustrado abaixo:

```
bin/lab10 --std
```

Por sua vez, a opção `--file nome` define a entrada e saída usando arquivos de nome `nome.in` e `nome.out`, respectivamente. Seu uso é ilustrado abaixo:

```
bin/lab10 --file teste
```

Note que, para evitar redundâncias em seu código, você deverá usar a coerção entre classe base (`istream` e `ostream`) e derivada (`ifstream` e `ofstream`).

(***lab11***) **DataFrames** – usando como base o projeto de matrizes, você deverá implementar uma classe derivada denominada DataFrame. Mais precisamente, um data frame é um tipo abstrato de dados bastante usado no contexto da ciência de dados, onde colunas representam diferentes características dos dados e linhas representam os dados em si. Em particular, cada coluna é tradicionalmente nomeada para identificar o seu conteúdo.

Neste roteiro extra, você deverá implementar apenas as operações de entrada e saída formatadas de DataFrames, isto é, a leitura de uma matriz precedida por um conjunto de nomes de colunas. Além disso, esta classe deverá fazer leitura e impressão considerando como separador o caracter ponto-e-vírgula (`;`). Isto é particularmente importante na prática de uso de data frames, uma vez que este tipo de conjunto de dados é tradicionalmente armazenado em arquivos **CSV** (valores separados por ponto-e-vírgula, do inglês *colon separated values*).

Para testar e expandir seu aprendizado de polimorfismo, sua aplicação deverá permitir que o usuário especifique um parâmetro dizendo se o arquivo de entrada conterá uma matriz (com separação de valores por espaços) ou um dataframe (em formato CSV). Os exemplos abaixo ilustram o uso da sua aplicação.

- Para testar sua aplicação com um dos testes de aceitação disponíveis na pasta `matrizes`, use a flag `--matriz` na linha de comando:

```
bin/lab11 --matriz < aceitacao/matrizes/entrada1.in
```

- Para testar sua aplicação com um dos testes de aceitação disponíveis na pasta `dataframes`, use a flag `--dataframe` na linha de comando:

```
bin/lab11 --dataframe < aceitacao/matrizes/IMD0030.in
```