Universidade Federal de Minas Gerais - UFMG
Departamento de Ciência da Computação - DCC
Programação e Desenvolvimento de Sistemas 2
Prof. Camila Laranjeira

Exercícios POO

1. **Sobrecarga**: Podemos especificar mais de uma função com o mesmo nome, contanto que haja uma distinção entre a quantidade de parâmetros e/ou seus tipos.

1.1 Escreva uma classe Lâmpada com os atributos do número de Watts (int) e seu preço (float). Crie 3 construtores para essa classe:

a) Um construtor vazio
b) Um construtor que recebe apenas o número de Watts da lâmpada
c) Um construtor que recebe ambos os atributos.

Para as declarações a seguir, informe qual construtor será chamado (a, b, c ou nenhum):

`Lampada lp;`		`Lampada lp(13.50)`	
`Lampada *lp;`		`Lampada lp(30, 13.50)`	
`Lampada lp();`		`int x = 60;` `Lampada lp(x);`	
`Lampada lp(30)`			

1.1.1 Na classe Lâmpada, sobrecarregue o operador + para que somar duas lâmpadas retorne a soma de seus preços. O método possui a seguinte assinatura:

```
float operator+(Lampada const &lp)
```

Na main, construa duas lâmpadas com preços definidos e imprima a sua soma.

1.2 Implemente a classe ASCIIArt que imprime diferentes artes a partir de caracteres da tabela ASCII. A arte impressa varia de acordo com o tipo do dado recebido por parâmetro. Para tal, você criará as sobrecargas da função print listadas a seguir como **métodos estáticos da classe**, ou seja, não será necessário instanciar um objeto tipo ASCIIArt para usufruir das funcionalidades.

a) static void print(std::string s): imprime a string dentro de um balão de hífens, sendo a quantidade de hífens equivalente a n+4 caracteres da string. Note que a parte inferior do balão tem a substituição do hífen pelas barras para compor a arte do balão de texto. Exemplo para a palavra “código”

```
----------
- código -
----\/----
```

b) static void print(char c) Cria o balão com uma repetição de 13x do caractere recebido e no meio escrito a palavra “ASCII ART”.

```
mmmmmmmmmmmmm
m ASCII ART m
mmmmm\/mmmmmm
```

c) static void print(int c): O inteiro recebido vai definir a quantidade de bonecos palito que devem ser desenhados lado a lado na tela (assuma que todos cabem em uma única linha). No exemplo a seguir, para c=3, os espaços em branco foram substituídos por hífens para facilitar a visualização.

```
-O---O---O
/|\-/|\-/|\
/-\-/-\-/-\
```

Na main, chame a função print 3x com os possíveis tipos de parâmetros.

2. Composição: O mecanismo mais simples de reaproveitamento de classes é a composição, onde uma classe tem relação estrutural com outra, incluindo-a como um de seus campos/atributos.

2.1 Dadas as classes Data e Hora a seguir, você deve criar a classe Calendário que se relaciona com as outras duas através da **composição**. Calendário deve ter atributos de Data, Hora, um construtor que recebe ambos, e uma função imprime que usa as funções imprime de ambas as classes.

```
class Data{
    int dia, mes, ano;
    public:
        Data(int d, int m, int a):
        dia(d), mes(m), ano(a){}
        std::string imprime(){
            return
             std::to_string(dia)+"/"+
             std::to_string(mes)+"/"+
             std::to_string(ano);
        }
};
```

```
class Hora{
    int hora, min, seg;
    public:
        Hora(int h, int m, int s):
        hora(h), min(m), seg(s){}
        std::string imprime(){
            return
            std::to_string(hora)+"/"+
            std::to_string(min) +"/"+
            std::to_string(seg);
        }
};
```

Universidade Federal de Minas Gerais - UFMG
Departamento de Ciência da Computação - DCC
Programação e Desenvolvimento de Sistemas 2
Prof. Camila Laranjeira

No contexto do reaproveitamento de classes via composição, vale ressaltar:

⚠️ Definições implícitas de TADs ⚠️

Quando criamos um TAD (class ou struct por ex.) são definidos implicitamente:

- Operador de atribuição `ClassName& operator= ( const ClassName &obj )` utilizado para copiar objetos já inicializados [documentação]

```
ClassName a, b;
a = b;
```

- Construtor de cópia `ClassName ( const ClassName& obj )` para inicialização de objetos nos seguintes contextos [documentação]:

```
ClassName a = b; ou ClassName a(b) // Declaração de variável
void func(ClassName a) // Passagem de parâmetro
ClassName func(){... return a} // retorno de função
```

⚠️ Construtores: Inicialização vs Atribuição ⚠️

Nós aprendemos duas maneiras de implementar o construtor de uma classe

- Inicialização: a inicialização dos atributos acontece no momento da construção do objeto, quando os atributos são definidos.

```
Class1(Class2 val): _val(val) {};
```

- Atribuição: Os atributos são definidos e inicializados com valores padrão ou lixo de memória, e em seguida é chamado o corpo da função, onde está a atribuição.

```
Class1(Class2 val) { _val = val };
```

O segundo caso equivale a: `Foo(ClassName val): _val()  { _val = val };`

Isso significa que se a classe `Class1` possui como atributo uma instância de outra classe `Class2`, é interessante inicializar este atributo da primeira maneira, para não haver dupla inicialização do atributo pois **garante-se a invocação do construtor de cópia implicitamente definido**. O segundo caso só funciona se a classe `Class2` possuir um construtor padrão (vazio).

Universidade Federal de Minas Gerais - UFMG
Departamento de Ciência da Computação - DCC
Programação e Desenvolvimento de Sistemas 2
Prof. Camila Laranjeira

3. Herança: Um dos pilares da Orientação a Objetos, a herança permite o reaproveitamento de classes através da criação de uma classe base que pode ser estendida por outras. Define-se portanto uma hierarquia de classes, onde superclasses compartilham seus atributos e métodos com suas subclasses.

3.1 Dadas as classes Produto e Eletronico sublinhe as instruções **inválidas** no main.

```
class Produto{
    int id;
    std::string nome;
    float preco;
public:
    Produto() = default;
    Produto(int i, string s, float p):
    id(i), nome(s), preco(p){}
    void imprime(){
    std::cout << id << " "
              << nome << " "
              << preco << "\n";
    }
};
```

```
class Eletronico: public Produto{
    int garantia;
public:
    Eletronico() = default;
    Eletronico(int i, string s, float
p, int g=0):
    Produto(i, s, p), garantia(g){}
    void comprar_garantia(int g){
         garantia = g;
    }
};
```

```
int main(){
    Produto p1;
    Produto p2(3418, "Controle", 399.19);
    Produto p3(1301, "Celular", 1280.99, 60);
    p3.imprime();

    Produto *pel = new Eletronico();
    pel->imprime();
    pel->comprar_garantia(120);

    Eletronico el1;
    Eletronico el2(1092, "Action Figure", 92.99);
    Eletronico el3(1093, "Chaveiro", 12.99, 60);
    el3.imprime();
    el3.comprar_garantia(120);
}
```

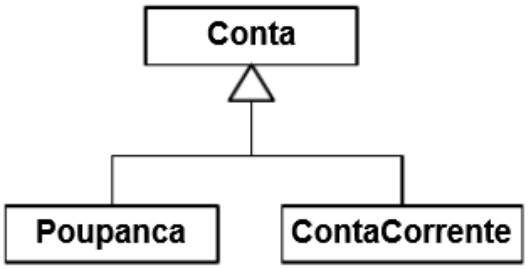
Universidade Federal de Minas Gerais - UFMG
Departamento de Ciência da Computação - DCC
Programação e Desenvolvimento de Sistemas 2
Prof. Camila Laranjeira

3.2 Crie uma hierarquia de herança que um banco possa utilizar para representar dois tipos de conta: poupança e conta corrente, apresentado no diagrama abaixo.

A classe *Conta* deve possuir um atributo de saldo da conta. Devem ser criados métodos:

- mostrar o saldo,
- sacar
- depositar na conta
- getter e setter do saldo

Se o valor de saque for maior que o saldo, deve-se imprimir uma mensagem de erro.



A classe *Poupanca* deve possuir um atributo relacionado ao rendimento, com métodos getter e setter. Crie também um método *CalculaRendimento*, que informa o valor do saldo multiplicado pela taxa de rendimento.

A classe *ContaCorrente* deve incluir um atributo que represente a taxa cobrada por cada transação de saque ou depósito, com getter e setter. Redefina (sobrescreva) os métodos de saque e depósito para aplicar a taxa a transações bem sucedidas.

Refaça o diagrama de classes apresentado acima, incluindo todos os métodos e atributos implementados por você.

> ⚠️ **Sobrescrita != Sobrecarga** ⚠️
>
> Note que os métodos de saque e depósito serão sobrescritos na *ContaCorrente*, ou seja, **redefinidos** com assinatura idêntica aos métodos da classe mãe *Conta*. A sobrescrita é uma característica da herança que permite às subclasses redefinir métodos da superclasse, de modo que apenas o método sobrescrito está acessível para objetos da subclasse.
>
> Já a sobrecarga pode existir em qualquer escopo, e apesar das funções terem o mesmo nome, suas assinaturas divergem no tipo e/ou quantidade de parâmetros, de modo que todas as funções sobrecarregadas são acessíveis dentro de seu escopo.

Universidade Federal de Minas Gerais - UFMG
Departamento de Ciência da Computação - DCC
Programação e Desenvolvimento de Sistemas 2
Prof. Camila Laranjeira

Referências:

- Santos, Rafael. *Introdução à programação orientada a objetos usando Java*. Elsevier, 2003.
    - Livro: https://ramonrdm.files.wordpress.com/2011/09/java-orientado-a-objetos.pdf
    - Livro de exercícios: http://www.lac.inpe.br/~rafael.santos/Docs/IntroPOOJava/POO-EXER.pdf

- Santache, André. MC302 – Programação Orientada a Objetos. Unicamp, 2011.
    - https://www.ic.unicamp.br/~santanch/teaching/oop/