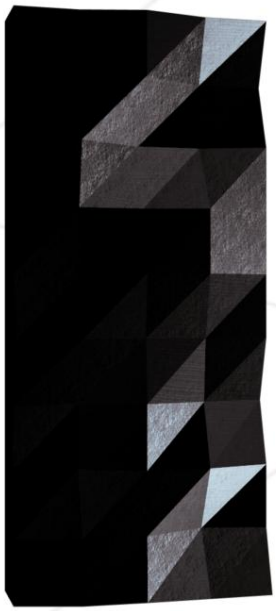

C++ - Módulo 04

Polimorfismo por subtipagem, classes abstratas, interfaces

Resumo:
Este documento contém os exercícios do Módulo 04 dos módulos C++.

Versão: 11

Índice

Capítulo I

Introdução

C++ é uma linguagem de programação de propósito geral criada por Bjarne Stroustrup como uma extensão da linguagem de programação C, ou "C com Classes" (fonte: Wikipedia).

C++ é uma linguagem de programação compilada que permite a programação sob vários paradigmas, incluindo programação processual, programação orientada a objetos e programação genérica. Seu bom desempenho e sua compatibilidade com C fazem dela uma das linguagens de programação mais utilizadas em aplicações onde o desempenho é crítico (fonte: Wikipedia).

Estes módulos têm como objetivo apresentar a **Programação Orientada a Objetos.**
Várias linguagens são recomendadas para aprender OOP. Por ser derivado do seu bom e velho amigo C, escolhemos a linguagem C++. Porém, por ser uma linguagem complexa e para não complicar sua tarefa, você cumprirá o padrão C++98.

Percebemos que o C++ moderno é diferente em muitos aspectos. Se você deseja melhorar seu domínio de C++, cabe a você buscar o núcleo comum de 42!

Capítulo II

Instruções gerais

Compilação

• Compile seu código com c++ e os sinalizadores -Wall -Wextra -Werror • Seu código deverá ser compilado se você adicionar o sinalizador -std=c++98

Convenções de formato e nomenclatura

• Os arquivos de exercícios terão os seguintes nomes: ex00, ex01, ..., ex

• Nomeie seus arquivos, classes, funções, funções-membro e atributos. homenagens conforme especificado nas instruções.

• Escreva os nomes das suas classes no **formato UpperCamelCase.** Os arquivos que contêm o código de uma classe terão o nome desta última. Por exemplo: ClassName.hpp/ClassName.h, ClassName.cpp ou ClassName.tpp.
Portanto, se um arquivo de cabeçalho contiver a definição de uma classe "BrickWall", seu nome será BrickWall.hpp.

• A menos que especificado de outra forma, todas as mensagens devem ser finalizadas com um retorno à linha e ser exibido na saída padrão.

• *Ciao Norminette!* Nenhum padrão é imposto durante os módulos C++. Você pode seguir o estilo de sua escolha. Mas tenha em mente que o código que seus colegas não conseguem entender é um código que seus colegas não conseguem avaliar. Portanto, faça o seu melhor para produzir um código limpo e legível.

O que é permitido e o que não é

A linguagem C acabou por enquanto. É hora de começar com C++! Portanto :

• Você pode usar quase toda a biblioteca padrão. Portanto, em vez de permanecer em terreno familiar, tente usar o máximo possível as versões C++ das funções C com as quais você está familiarizado.

• Entretanto, você não pode usar nenhuma outra biblioteca externa.
O que significa que C++ 11 (e derivados) e o conjunto Boost são proibidos. Além disso, certas funções permanecem proibidas. Usar as seguintes funções resultará

na nota 0: *printf(), *alloc() e free().

- Salvo indicação explícita em contrário, palavras-chave que usam namespace <ns_name> e amigo são proibidos. Seu uso resultará em uma pontuação de -42.
- **Você só tem direito ao STL nos Módulos 08 e 09.** Até então, é proibido o uso de **Containers** (vetor/lista/mapa/etc.) e Algoritmos **(qualquer** coisa que requeira incluir <algoritmo>). Caso contrário, você obterá uma pontuação de -42.

Algumas obrigações de design

- Vazamentos de memória também existem em C++. Ao alocar memória (usando a palavra-chave new), você **não deverá ter vazamentos de memória.**
- Do Módulo 02 ao Módulo 09, suas aulas devem estar em conformidade com a **forma canônica, conhecida como Coplian, a menos que seja explicitamente**

especificado o contrário. • Uma função implementada em um arquivo de cabeçalho (exceto no caso de uma função de modelo) será

equivalente a uma pontuação de 0. • Você deve ser capaz de usar seus arquivos de cabeçalho separadamente u
É por isso que deverão incluir todas as dependências que serão necessárias para eles. No entanto, você deve evitar o problema de dupla inclusão protegendo-os com **protetores de inclusão.** Caso contrário, sua pontuação será 0.

Leia-me

- Se necessário, você pode renderizar arquivos adicionais (por exemplo, para separar seu código em mais arquivos). Como seu trabalho não será avaliado por um programa, faça o que achar melhor, desde que torne os arquivos obrigatórios.

- As instruções para um exercício podem parecer simples, mas os exemplos por vezes contêm instruções adicionais que não são explicitamente solicitadas.
- Leia cada módulo completamente antes de começar! Realmente. • Por Odin, por

Thor! Use seu cérebro! ! !

Você terá que implementar um bom número de aulas, o que pode ser difícil... ou não!
Pode haver uma maneira de facilitar sua vida usando seu editor de texto favorito.

Você tem bastante liberdade para resolver os exercícios.
No entanto, siga as instruções e não se limite ao mínimo, pois você pode perder conceitos interessantes.
Sinta-se à vontade para ler alguma teoria.

Capítulo III

Exercício 00: Polimorfismo

	Exercício: 00
Polimorfismo	
Pasta de renderização:	
ex00/ Arquivos para renderizar: Makefile, main.cpp, *.cpp, *.{h, hpp}	
Funções proibidas: Nenhuma	

Para cada exercício, forneça os testes **mais completos** possíveis.
Os construtores e destruidores de cada classe devem exibir mensagens exclusivas para eles. Não use a mesma mensagem para todas as aulas.

Comece implementando uma **classe Animal básica simples.** Possui um atributo protegido:

- tipo std::string;

Implemente uma classe **Dog** que herda de *Animal* .
Implemente uma classe **Cat** que herda de *Animal* .

Essas duas classes derivadas devem inicializar seu tipo com base em seu nome. Portanto, o tipo de Cachorro será “Cachorro” e o tipo de Gato será “Gato”. O tipo da classe Animal pode ser deixado vazio ou inicializado com o valor de sua escolha.

Cada animal deve ser capaz de usar a função membro: makeSound()

Ele exibirá um som consistente (gatos não latem).

A execução deste código deve exibir os sons específicos das classes Dog e Cat, não os de a classe Animal.

```
int principal()
{
        const Animal* meta = novo Animal();
        const Animal* j = novo Cachorro();
        const Animal* i = novo Gato();

        std::cout << j->getType() << << std::endl;
        std::cout << i->getType() << << std::endl;
        i->makeSound(); //irá emitir o som do gato!
        j->makeSound();
        meta->makeSound();
        ...

        retornar 0;
}
```

Para ter certeza de que você entendeu, implemente uma herança de classe **WrongCat** de uma **classe WrongAnimal.** No código acima, se você substituir Animal e Gato pelo WrongAnimal e pelo WrongCat, o WrongCat deve exibir o som do Wron-gAnimal.

Escreva e entregue mais testes do que os fornecidos acima.

Capítulo IV

Exercício 01: Não quero queimar o mundo

	Exercício: 01
Eu não quero queimar o mundo	
Pasta de renderização: ex01/	
Arquivos para renderizar: Arquivos do exercício anterior + *.cpp, *.{h, hpp}	
Funções proibidas: Nenhuma	

Construtores e destruidores de cada classe devem exibir mensagens que são específicos para eles.

Implemente uma classe Brain **contendo** *um* array de 100 std::string chamado ideas .

Portanto as classes Cão e Gato terão um atributo privado Cérebro*. Na construção, as classes Cão e Gato criarão seu Cérebro com o novo Cérebro(); Após a destruição, as classes Cão e Gato terão que deletar seus Cérebros.

Em sua função principal, crie e preencha uma matriz de objetos **Animal ,** metade dos quais são objetos **Dog** e a outra metade são objetos Cat . Ao final da execução do programa, percorra esta tabela para deletar cada Animal. Você deve excluir diretamente cães e gatos como Animais. Os destruidores correspondentes devem ser chamados na ordem correta.

Não se esqueça de verificar se não há **vazamentos de memória.**

Copiar um objeto Dog ou um objeto Cat não deve ser superficial. Portanto- importante, você precisa ter certeza de que suas cópias são realmente cópias profundas.

```
int principal()
{
        const Animal* j = novo Cachorro(); const
        Animal* i = novo Gato();


        delete j;//não deve criar um vazamento delete i;

        ...

        retornar 0;
}
```

Escreva e entregue mais testes do que os fornecidos acima.

Capítulo V

Exercício 02: Aula abstrata

	Exercício: 02
Classe abstrata	
Pasta de renderização: ex02/	
Arquivos para renderizar: Arquivos do exercício anterior + *.cpp, *.{h, hpp}	
Funções proibidas: Nenhuma	

Criar objetos Animal não adianta muito no final. Eles não fazem barulho!

Para evitar possíveis erros, a classe base Animal não deve ser instanciável. Modifique-o para que ninguém possa instanciá-lo. Seu código deve funcionar como antes.

Se desejar, você pode adicionar a letra A para prefixar o nome Animal.

Capítulo VI

Exercício 03: Interface e recapitulação

	Exercício: 03
Interface e recapitulação	
Pasta de renderização: ex03/	
Arquivos para renderizar: Makefile, main.cpp, *.cpp, *.{h, hpp}	
Funções proibidas: Nenhuma	

Não há interfaces em C++98 (nem em C++20). No entanto, classes puramente abstratas são comumente chamadas de interfaces. Portanto, neste último exercício, para garantir que este módulo seja dominado, você implementará interfaces.

Complete a definição da seguinte classe **AMateria** e implemente as funções membros necessários.

```
classe AMateria
{
        protegido: [...]


        público:
                AMateria(std::string const & tipo); [...]


                    std::string const & getType() const; //Retorna o tipo de matéria

                    virtual AMateria* clone() const = 0; virtual void
                    use(ICharacter& target);
};
```

Implemente as **Ice** and Cure **Materias** *como* classes concretas.
Use seus nomes em letras minúsculas ("ice" para Ice, "cure" para Cure) como tipos. Claro, sua função membro clone() retornará uma nova instância do mesmo tipo (ao clonar uma Ice Materia, obtemos outra Ice Materia).

Quanto à função membro use(ICharacter&), ela exibirá:

- Gelo: "* atira um raio de gelo em <nome> *"
- Cura: "* cura as feridas de <nome> *"

<nome> é o nome do *personagem* **passado** como parâmetro. Não exiba colchetes angulares (< e >).

Quando você atribui uma Matéria a outra, copiar seu tipo tem pouco interesse.

Crie a classe concreta **Character** que implementará a seguinte interface:

```
classe ICharacter {

    público:
        virtual ~ICharacter() {} virtual std::string const
            & getName() const = 0; virtual void equipar(AMateria* m) = 0; virtual void
            desequipar(int idx) = 0; virtual void usar(int idx, ICharacter&
            target) = 0;

};
```

O **Personagem** possui um inventário de 4 itens, ou no máximo 4 Matérias. Na construção, o estoque está vazio. As Matérias são equipadas no primeiro slot vazio encontrado, ou seja, na seguinte ordem: do slot 0 ao 3. No caso de tentar adicionar uma Matéria a um inventário cheio, ou usar/remover uma Matéria que não existe, faça nada (isso não permite bugs). A função membro unequip() NÃO deve excluir a Matéria!

Cuide das Matérias deixadas no chão pelo seu personagem como achar melhor. Você pode salvar o endereço antes de chamar unequip(), ou qualquer outra coisa, desde que não vaze nenhum
memória.

A função membro use(int, ICharacter&) usará a Matéria de location[idx] e passará o alvo como parâmetro para a função AMateria::use.

O inventário do seu personagem deve conter qualquer tipo de item AMateria.

Seu **personagem** deve incluir um construtor tomando seu nome como parâmetro. Qualquer cópia (com o construtor de cópia ou operador de atribuição) de um personagem deve ser **profunda.** Assim, durante uma cópia, as Matérias do Personagem devem ser deletadas antes que as novas as substituam no inventário. Obviamente, as Matérias também devem ser excluídas quando um Personagem é destruído.

Crie a classe concreta **MateriaSource** que implementará a seguinte interface:

```
classe IMateriaSource
{
        público:
                virtual ~IMateriaSource() {} virtual void
                  learnMateria(AMateria*) = 0; virtual AMateria*
                  createMateria(std::string const & type) = 0;
};
```

- aprenderMateria(AMateria*)
  Copia a Matéria passada como parâmetro e a armazena na memória para cloná-la posteriormente. Assim como o Personagem, a **MateriaSource** pode conter no máximo 4 Matérias. Estes não são necessariamente únicos.
- criarMateria(std::string const &)
  Retorna uma nova Matéria. Esta é uma cópia daquela aprendida anteriormente pelo **MateriaSource** e cujo tipo é igual ao passado como parâmetro. Retorna 0 se o tipo for desconhecido.

Resumindo, seu **MateriaSource** deve ser capaz de aprender “padrões” de Materias para recriá-los à vontade. Assim, você poderá gerar uma nova Matéria a partir do seu tipo na forma de uma sequência de caracteres.

Execute este código:

```
int principal()
{
        IMaterialSource* src = new MaterialSource(); src-
        >learnMatter(new Ice()); src-
        >learnMaterial(new Cure());

        ICharacter* eu = novo Personagem("eu");

        AMateria* tmp; tmp
        = src->createMateria("gelo"); me->equipar(tmp);
        tmp = src-
        >createMateria("cura"); me->equipar(tmp);


        ICharacter* bob = novo Personagem("bob");

        eu->usar(0, *bob); eu-
        >usar(1, *bob);

        apagar bob;
        apagar -me;
        apagar src;

        retornar 0;
}
```

Deve exibir:

```
$> clang++ -W -Wall -Werror *.cpp $> ./a.out | cat
-e * atira um raio de gelo em
bob *$ * cura os ferimentos de bob *$
```

Como sempre, escreva e entregue mais testes do que os fornecidos acima.

Você pode concluir este módulo sem o exercício 03.

Capítulo VII

Submissão e avaliação por pares

Envie seu trabalho para o repositório Git normalmente. Somente o trabalho presente em seu depósito será avaliado na defesa. Verifique novamente os nomes de suas pastas e arquivos para que atendam às solicitações do sujeito.

??????????? XXXXXXXXXX = $3$$6b616b915363639715 73e58914295d42