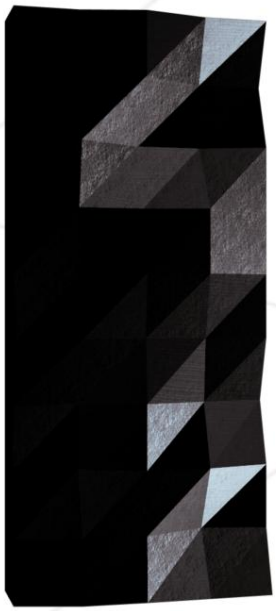

C++ - Módulo 04

Polimorfismo de subtipos, classes abstratas, interfaces

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

Versão: 10

Conteúdo

Capítulo I

Introdução

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

O objetivo desses módulos é apresentá-lo à **Programação Orientada a Objetos.** Este será o ponto de partida de sua jornada C++. Muitos idiomas são recomendados para aprender OOP. Decidimos escolher C++, pois é derivado de seu velho amigo C.
Por se tratar de uma linguagem complexa e para manter as coisas simples, seu código estará em conformidade com o padrão C++98.

Sabemos que o C++ moderno é muito diferente em muitos aspectos. Portanto, se você deseja se tornar um desenvolvedor C++ proficiente, cabe a você ir além do 42 Common Core!

Capítulo II

Regras gerais

Compilando

- Compile seu código com c++ e os sinalizadores -Wall -Wextra -Werror
- Seu código ainda deve compilar se você adicionar o sinalizador -std=c++98

Convenções de formatação e nomenclatura

- Os diretórios de exercícios serão nomeados desta forma: ex00, ex01, ... , exn
- Nomeie seus arquivos, classes, funções, funções de membro e atributos conforme exigido em As diretrizes.
- Escreva os nomes das classes no formato **UpperCamelCase .** Arquivos contendo código de classe serão sempre ser nomeado de acordo com o nome da classe. Por exemplo: ClassName.hpp/ClassName.h, ClassName.cpp ou ClassName.tpp. Então, se você tiver um arquivo de cabeçalho contendo a definição de uma classe "BrickWall" que representa uma parede de tijolos, seu nome será BrickWall.hpp.
- A menos que especificado de outra forma, todas as mensagens de saída devem ser encerradas com uma nova linha caractere e exibido na saída padrão.
- *Adeus Norminette!* Nenhum estilo de codificação é aplicado nos módulos C++. Você pode seguir o seu favorito. Mas lembre-se de que um código que seus pares avaliadores não conseguem entender é um código que eles não podem avaliar. Faça o seu melhor para escrever um código limpo e legível.

Permitido/Proibido

Você não está mais codificando em C. Hora de C++! Portanto:

- Você tem permissão para usar quase tudo da biblioteca padrão. Portanto, em vez de se ater ao que você já sabe, seria inteligente usar o máximo possível as versões em C++ das funções C às quais você está acostumado.
- No entanto, você não pode usar nenhuma outra biblioteca externa. Isso significa que as bibliotecas C++11 (e formas derivadas) e Boost são proibidas. As seguintes funções também são proibidas: *printf(), *alloc() e free(). Se você usá-los, sua nota será 0 e pronto.

• Observe que, a menos que explicitamente declarado de outra forma, o namespace using <ns_name> e palavras-chave de amigos são proibidas. Caso contrário, sua nota será -42.

• **Você tem permissão para usar o STL apenas no Módulo 08 e 09.** Isso significa: sem **contêineres** (vetor/lista/mapa/e assim por diante) e sem **algoritmos** (qualquer coisa que requeira incluir o cabeçalho <algorithm>) até então. Caso contrário, sua nota será -42.

Alguns requisitos de projeto

• O vazamento de memória também ocorre em C++. Quando você aloca memória (usando o novo palavra-chave), você deve evitar **vazamentos de memória.**

• Do Módulo 02 ao Módulo 09, suas aulas devem ser elaboradas no **Ortodoxo Forma Canônica, exceto quando explicitamente declarado de outra forma.**

• Qualquer implementação de função colocada em um arquivo de cabeçalho (exceto para modelos de função) significa 0 para o exercício.

• Você deve ser capaz de usar cada um de seus cabeçalhos independentemente dos outros. Assim, eles devem incluir todas as dependências de que precisam. No entanto, você deve evitar o problema de inclusão dupla adicionando **guardas de inclusão.** Caso contrário, sua nota será 0.

Leia-me

• Você pode adicionar alguns arquivos adicionais se precisar (ou seja, para dividir seu código). Como essas atribuições não são verificadas por um programa, sinta-se à vontade para fazê-lo, desde que entregue os arquivos obrigatórios.

• Às vezes, as diretrizes de um exercício parecem curtas, mas os exemplos podem mostrar requisitos que não estão explicitamente escritos nas instruções.

• Leia cada módulo completamente antes de começar! Realmente, faça isso.

• Por Odin, por Thor! Use seu cérebro!!!

Você terá que implementar muitas classes. Isso pode parecer tedioso,
a menos que você seja capaz de criar o script de seu editor de texto favorito.

Você tem uma certa liberdade para completar os exercícios.
No entanto, siga as regras obrigatórias e não seja preguiçoso. Você poderia
perca muita informação útil! Não hesite em ler sobre
conceitos teóricos.

Capítulo III

Exercício 00: Polimorfismo

	Exercício: 00
Polimorfismo	
Diretório de entrega: ex00/	
Arquivos a serem entregues: Makefile, main.cpp, *.cpp, *.{h, hpp}	
Funções proibidas: Nenhuma	

Para cada exercício, você deve fornecer os **testes mais completos** que puder.
Construtores e destruidores de cada classe devem exibir mensagens específicas. Não use a mesma mensagem para todas as classes.

Comece implementando uma classe base simples chamada **Animal.** tem um protegido atributo:

• tipo std::string;

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

Essas duas classes derivadas devem definir seu campo de tipo dependendo de seu nome. Então, o tipo do Dog será inicializado como "Dog" e o tipo do Cat será inicializado como "Cat".
O tipo da classe Animal pode ser deixado em branco ou definido com o valor de sua escolha.

Todo animal deve ser capaz de usar a função de membro:
makeSound()
Ele imprimirá um som apropriado (gatos não latem).

A execução desse código deve imprimir os sons específicos das classes Dog e Cat, não os de Animal.

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

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


        ...

        retorna 0;
}
```

Para garantir que você entendeu como funciona, implemente uma classe **WrongCat** que herda de uma classe **WrongAnimal .** Se você substituir o Animal e o Gato pelos errados no código acima, o WrongCat deve produzir o som WrongAnimal.

Implemente e entregue mais testes do que os dados acima.

Capítulo IV

Exercício 01: Não quero botar fogo no mundo

	Exercício: 01
Eu não quero incendiar o mundo	
Diretório de entrega : ex01/	
Arquivos a entregar : Arquivos do exercício anterior + *.cpp, *.{h, hpp}	
Funções proibidas: Nenhuma	

Construtores e destruidores de cada classe devem exibir mensagens específicas.

Implemente uma classe **Brain .** Ele contém um array de 100 std::string chamado ideas.
Dessa forma, Cão e Gato terão um atributo Cérebro* privado.
Após a construção, Cão e Gato criarão seu Cérebro usando new Brain(); Após a destruição,
Cão e Gato irão deletar seu Cérebro.

Em sua função principal, crie e preencha uma matriz de objetos **Animal .** Metade serão objetos **Dog** e a outra metade serão objetos **Cat .** No final da execução do programa, faça um loop sobre esse array e exclua todos os animais. Você deve excluir diretamente cães e gatos como Animais. Os destruidores apropriados devem ser chamados na ordem esperada.

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

Uma cópia de um cão ou gato não deve ser superficial. Portanto, você deve testar se suas cópias são cópias profundas!

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


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

        ...

        retornar 0;
}
```

Implemente e entregue mais testes do que os dados acima.

Capítulo V

Exercício 02: Aula abstrata

	Exercício: 02
classe abstrata	
Diretório de entrega : ex02/	
Arquivos a entregar : Arquivos do exercício anterior + *.cpp, *.{h, hpp}	
Funções proibidas: Nenhuma	

Afinal, criar objetos Animal não faz sentido. É verdade, eles não fazem barulho!

Para evitar possíveis erros, a classe padrão Animal não deve ser instanciável.
Corrija a classe Animal para que ninguém possa instanciá-la. Tudo deve funcionar como antes.

Se desejar, você pode atualizar o nome da classe adicionando um prefixo A a Animal.

Capítulo VI

Exercício 03: Interface e recapitulação

	Exercício: 03
Interface e recapitulação	
Diretório de entrega: ex03/	
Arquivos a serem entregues: Makefile, main.cpp, *.cpp, *.{h, hpp}	
Funções proibidas: Nenhuma	

Interfaces não existem em C++98 (nem mesmo em C++20). No entanto, classes abstratas puras são comumente chamadas de interfaces. Assim, neste último exercício, vamos tentar implementar interfaces para garantir que você tenha este módulo.

Conclua a definição da seguinte classe **AMateria** e implemente as funções de membro necessárias.

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


        public:
                AMateria(std::string const & type); [...]


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

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

Implemente as classes concretas de Matérias **Gelo** e **Cura.** Use seu nome em letras minúsculas ("ice" para Ice, "cure" para Cure) para definir seus tipos. Obviamente, a função de membro clone() retornará uma nova instância do mesmo tipo (ou seja, se você clonar uma Ice Materia, obterá uma nova Ice Materia).

A função de membro use(ICharacter&) exibirá:

• Gelo: "* atira uma flecha de gelo em <nome> *"

• Cura: "* cura as feridas de <nome> *"

<name> é o nome do Character passado como parâmetro. Não imprima os colchetes angulares (< e >).

Ao atribuir uma Matéria a outra, copiar o tipo não faz senso.

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

```
classe ICaractere {

    public:
        virtual ~ICharacter() {} virtual
            std::string const & getName() const = 0; virtual void equip(AMateria*
            m) = 0; virtual void unequip(int idx) = 0; virtual void
            use(int idx, ICharacter& target) = 0;

};
```

O **Personagem** possui um inventário de 4 slots, o que significa no máximo 4 Materias. O inventário está vazio na construção. Eles equipam as Materias no primeiro espaço vazio que encontram. Isso significa, nesta ordem: do slot 0 ao slot 3. Caso eles tentem adicionar uma Materia a um inventário completo, ou usar/desequipar uma Materia inexistente, não faça nada (mas ainda assim, bugs são proibidos). A função de membro unequip() NÃO deve excluir o Materia!

Manuseie as Materias que seu personagem deixou no chão como quiser.
Salve os endereços antes de chamar unequip(), ou qualquer outra coisa, mas não esqueça que você deve evitar vazamentos de memória.

A função de membro use(int, ICharacter&) terá que usar a Materia no slot[idx] e passe o parâmetro de destino para a função AMateria::use.

O inventário do seu personagem será capaz de suportar qualquer tipo de AMatéria.

Seu **Personagem** deve ter um construtor levando seu nome como parâmetro. Qualquer cópia (usando construtor de cópia ou operador de atribuição de cópia) de um personagem deve ser **profunda.**
Durante a cópia, as Matérias de um Personagem devem ser excluídas antes que as novas sejam adicionadas ao seu inventário. Claro, as Materias devem ser excluídas quando um Personagem é destruído.

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

```
classe IMateriaSource
{
        public:
                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 que possa ser clonada posteriormente. Assim como o Personagem, a **MateriaSource** pode conhecer no máximo 4 Materias. Eles não são necessariamente únicos.

• createMateria(std::string const &)
Retorna uma nova Matéria. Este último é uma cópia da Materia previamente aprendida pela **MateriaSource** cujo tipo é igual ao passado como parâmetro. Retorna 0 se o tipo for desconhecido.

Resumindo, sua **MateriaSource** deve ser capaz de aprender "modelos" de Materias para criá-los quando necessário. Então, você poderá gerar uma nova Matéria usando apenas uma string que identifica seu tipo.

Executando este código:

```
int principal()
{
        IMateriaSource* src = new MateriaSource(); src-
        >learnMateria(new Ice()); src-
        >learnMateria(new Cure());

        ICaractere* eu = new Personagem("eu");

        AMatéria* tmp; tmp
        = src->createMateria("gelo"); me->equip(tmp);
        tmp = src-
        >createMateria("cura"); me->equip(tmp);


        ICaractere* bob = new Personagem("bob");

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

        excluir bob;
        exclua- me;
        excluir src;

        retorna 0;
}
```

Deve produzir:

```
$> clang++ -W -Wall -Werror *.cpp $> ./a.out |
cat -e * atira uma flecha
de gelo em bob *$ * cura as feridas de
bob *$
```

Como de costume, implemente e entregue mais testes do que os dados acima.

Você pode passar neste módulo sem fazer o exercício 03.