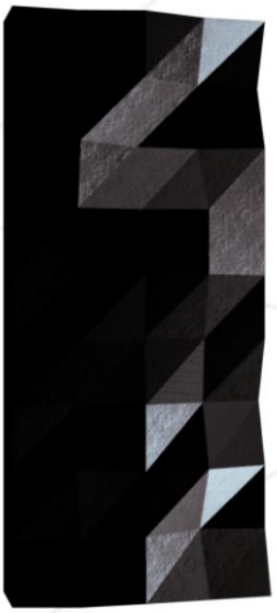

C++ - Módulo 04

Polimorfismo de subtipo, classes abstratas, interfaces

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

Versão: 11

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 com Classes" (fonte: Wikipedia).

O objetivo desses módulos é apresentar a Programação Orientada a Objetos.
Este será o ponto de partida de sua jornada em C++. Muitas linguagens são recomendadas para aprender OOP. Decidimos escolher C++ porque ele é derivado do 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. Então se você quer se tornar um desenvolvedor C++ proficiente, cabe a você ir além após o 42 Common Core!

Capítulo II

Regras gerais

Compilando

- Compile seu código com c++ e as tags -Wall -Wextra -Werror
- Seu código ainda deverá ser compilado se você adicionar a tag -std=c++98

Convenções de formatação e nomenclatura

- Os diretórios dos exercícios serão nomeados desta forma: ex00, ex01, ... , ex
- Nomeie seus arquivos, classes, funções, funções de membro e atributos conforme exigido em As diretrizes.
- Escreva nomes de 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 finalizadas com uma nova linha caractere e exibido na saída padrão.
- Adeus Norminette! Nenhum estilo de codificação é imposto nos módulos C++. Você pode seguir o seu favorito. Mas tenha em mente 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. É hora de C++! Portanto:

- Você tem permissão para usar quase tudo da biblioteca padrão. Assim, em vez de se ater ao que você já sabe, seria inteligente usar o máximo possível as versões C++ das funções C com as quais você está acostumado.
- Entretanto, você não pode usar nenhuma outra biblioteca externa. Isso significa que C++ 11 (e formas derivadas) e bibliotecas 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, salvo indicação explícita em contrário, o namespace using <ns_name> e palavras-chave de amigos são proibidas. Caso contrário, sua nota será -42.
- É permitido utilizar o STL somente nos Módulos 08 e 09. Isso significa: nenhum contêiner (vetor/lista/mapa/e assim por diante) e nenhum algoritmo (qualquer coisa que exija a inclusão do cabeçalho <algorithm>) até então. Caso contrário, sua nota será -42.

Alguns requisitos de design

- 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 planejadas no estilo Ortodoxo Forma Canônica, exceto quando explicitamente indicado de outra forma.
- Qualquer implementação de função colocada em um arquivo de cabeçalho (exceto modelos de função) significa 0 para o exercício.
- Você deve ser capaz de usar cada um dos seus cabeçalhos independentemente dos outros. Assim, eles devem incluir todas as dependências de que necessitam. No entanto, você deve evitar o problema da 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 tarefas não são verificadas por um programa, fique à vontade para fazê-lo, desde que entregue os arquivos obrigatórios.
- Às vezes, as orientações 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 consiga criar um script em seu editor de texto favorito.

Você tem uma certa liberdade para completar os exercícios.
Porém, siga as regras obrigatórias e não seja preguiçoso. Você poderia
perca muitas informações úteis! Não hesite em ler sobre
conceitos teóricos.

Capítulo III

Exercício 00: Polimorfismo

	Exercício: 00
Polimorfismo	
Diretório de entrega: ex00ÿ	
Arquivos para entregar: Makefile, main.cpp, *.cpp, *.{h, hpp}	
Funções proibidas: Nenhuma	

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

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 que herda 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 membro: makeSound()

Irá imprimir um som apropriado (gatos não latem).

A execução deste código deve imprimir os sons específicos das classes Dog e Cat, não dos 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(); //irá emitir o som do gato! j-
        >makeSound(); meta->makeSound();


        ...

        retornar 0;
}
```

Para garantir que você entendeu como funciona, implemente uma classe WrongCat que herda de uma classe WrongAnimal. Se você substituir Animal e Cat pelos errados no código acima, WrongCat deverá emitir o som WrongAnimal.

Implemente e entregue mais testes do que os fornecidos acima.

Capítulo IV

Exercício 01: Não quero colocar fogo no mundo

	Exercício: 01
Eu não quero colocar fogo no mundo	
Diretório de entrega: ex01ÿ	
Arquivos para 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 uma matriz de 100 std::string chamadas ideias.
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 excluirão seu Cérebro.

Na sua função principal, crie e preencha um array de objetos Animal. Metade serão objetos Cachorro e a outra metade serão objetos Gato. 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 de um 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
        exclua eu;
        ...

        retornar 0;
}
```

Implemente e entregue mais testes do que os fornecidos acima.

Capítulo V

Exercício 02: Aula abstrata

	Exercício: 02
Classe abstrata	
Diretório de entrega: ex02ÿ	
Arquivos para entregar: Arquivos do exercício anterior + *.cpp, *.{h, hpp}	
Funções proibidas: Nenhuma	

Afinal, criar objetos Animal não faz sentido. É verdade, eles não emitem nenhum som!

Para evitar possíveis erros, a classe Animal padrão 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: ex03ÿ	
Arquivos para entregar: 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 ter certeza de que você obteve este módulo.

Complete a deÿnição da seguinte classe AMateria e implemente as medidas necessárias funções de membro.

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


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


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

                    virtual AMateria* clone() const = 0; uso de vazio virtual
                    (ICharacter& target);
};
```

Implementar as classes concretas Materias Ice e Cure. Use o nome deles em letras minúsculas ("ice" para Ice, "cure" para Cure) para definir seus tipos. Claro, sua função membro clone() retornará uma nova instância do mesmo tipo (ou seja, se você clonar uma Ice Materia, você obterá uma nova Ice Materia).

A função membro use(ICharacter&) 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 imprima os colchetes angulares (< e >).

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

Escreva a classe concreta Character que implementará a seguinte interface:

```
classe ICharacter
{
        público:
                virtual ~ICharacter() {} virtual
                   std::string const & getName() const = 0; equipamento de vazio
                   virtual(AMateria* m) = 0; virtual void desequipado(int
                   idx) = 0; uso de vazio virtual (int idx, ICharacter&
                   target) = 0;
};
```

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

Manuseie as Matérias 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 membro use(int, ICharacter&) terá que usar a Materia no slot[idx] e passe o parâmetro target para a função AMateria::use.

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

Seu personagem deve ter um construtor tomando 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 Matérias devem ser deletadas quando um Personagem é destruído.

Escreva a classe concreta MateriaSource que implementará a seguinte interface:

```
class 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 que possa ser clonada posteriormente. Assim como o Personagem, a MateriaSource pode conhecer no máximo 4 Matérias. Eles não são necessariamente únicos.

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

Resumindo, seu MateriaSource deve ser capaz de aprender “templates” de Materias para criá-los quando necessário. Assim, você poderá gerar uma nova Matéria utilizando apenas uma string que identifique seu tipo.

Executando este código:

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


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

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


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

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

        excluir bob;
        exclua -me;
        excluir fonte;

        retornar 0;
}
```

Deve produzir:

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

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

Você pode passar neste módulo sem fazer o exercício 03.

Capítulo VII

Envio e avaliação por pares

Entregue sua tarefa em seu repositório Git normalmente. Somente o trabalho dentro do seu repositório será avaliado durante a defesa. Não hesite em verificar os nomes de suas pastas e arquivos para garantir que estejam corretos.

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