



C++ - Módulo 04

Polimorfismo de subtipo, classes abstratas, interfaces

Resumo:

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

Versão: 10

Conteúdo

I	Introdução	2
II	Regras gerais	3
III	Exercício 00: Polimorfismo	5
IV	Exercício 01: Não quero colocar fogo no mundo	7
V	Exercício 02: Classe abstrata	9
VI	Exercício 03: Interface e recapitulação	10

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 apresentá-lo à **programação orientada a objetos**. Esse será o ponto de partida de sua jornada em C++. Muitas linguagens são recomendadas para aprender OOP. Decidimos escolher o C++, pois ele é derivado do seu velho amigo C. Como essa é uma linguagem complexa, e para manter as coisas simples, seu código obedecerá ao padrão C++98.

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

Capítulo II Regras

gerais

Compilação

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

Convenções de formatação e nomenclatura

- Os diretórios de exercícios serão nomeados da seguinte forma: ex00, ex01, ..., exn
- Nomeie seus arquivos, classes, funções, funções de membro e atributos conforme exigido nas diretrizes.
- Escreva os nomes das classes no formato **UpperCamelCase**. Os arquivos que contêm código de classe sempre serão nomeados de acordo com o nome da classe. Por exemplo: ClassName.hpp/ClassName.h, ClassName.cpp ou ClassName.hpp. Então, se você tiver um arquivo de cabeçalho que contenha 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 um caractere de nova linha e exibidas na saída padrão.
- *Adeus Norminette!* Nenhum estilo de codificação é imposto nos módulos C++. Você pode seguir seu estilo preferido. Mas lembre-se de que um código que seus colegas avaliadores não conseguem entender é um código que eles não podem avaliar. Faça o possível para escrever um código limpo e legível.

Permitido/Proibido

Você não está mais codificando em C. É hora de usar C++! Portanto:

- Você tem permissão para usar quase tudo da biblioteca padrão. Portanto, em vez de se ater ao que já sabe, seria inteligente usar o máximo possível as versões em C++ das funções em C com as quais 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 as usar, sua nota será 0 e pronto.

- Observe que, a menos que explicitamente declarado de outra forma, o namespace de uso `<ns_name>` e palavras-chave amigas são proibidas. Caso contrário, sua nota será -42.
- **Você tem permissão para usar o STL somente nos Módulos 08 e 09.** Isso significa: nada de **Contêineres** (vetor/lista/mapa/e assim por diante) e nada de **Algoritmos** (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 no C++. Quando você aloca memória (usando o comando `new`), você deve evitar **vazamentos de memória**.
- Do Módulo 02 ao Módulo 09, suas aulas devem ser elaboradas na **Forma Canônica Ortodoxa, exceto quando explicitamente indicado 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 dos seus cabeçalhos independentemente dos outros. Portanto, eles devem incluir todas as dependências de que precisam. No entanto, você deve evitar o problema da inclusão dupla adicionando **proteções de inclusão**. Caso contrário, sua nota será 0.

Leia-me

- Você pode acrescentar alguns arquivos adicionais, se necessário (ou seja, para dividir seu código). Como essas tarefas não são verificadas por um programa, sinta-se à vontade para fazer isso, 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! De fato, faça isso.
- Por Odin, por Thor! Use seu cérebro!!!




Você terá de implementar muitas classes. Isso pode parecer entediante, a menos que você saiba programar seu editor de texto favorito.



Você tem certa liberdade para concluir os exercícios. Entretanto, siga as regras obrigatórias e não seja preguiçoso. Você perderia muitas informações úteis! Não hesite em ler sobre conceitos teóricos.

Capítulo III

Exercício 00: Polimorfismo

	Exercício : 00
Polimorfismo	
Diretório de entrada: <i>ex00/</i>	
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. Os 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**. Ela tem um atributo protegido:

- `std::string` type;

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 de acordo com 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, e não os do Animal.

```
int main()
{
    const Animal* meta = new Animal();
    const Animal* j = new Dog();
    const Animal* i = new Cat();

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


    retornar 0;
}
```

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

Implemente e entregue mais testes do que os fornecidos acima.

Capítulo IV

Exercício 01: Não quero que o mundo pegue fogo

	Exercício: 01
Não quero incendiar o mundo	
Diretório de entrada: <i>ex01/</i>	
Arquivos a serem entregues : Arquivos do exercício anterior + *.cpp, *.{h, hpp}	
Funções proibidas : Nenhuma	

Os construtores e destruidores de cada classe devem exibir mensagens específicas.

Implemente uma classe **Brain**. Ela contém uma matriz de 100 `std::string` chamada `ideas`.

Dessa forma, Dog e Cat terão um atributo `Brain*` privado.

Após a construção, o Cão e o Gato criarão seu Cérebro usando `new Brain()`;

Após a destruição, o Cão e o Gato excluirão seu Cérebro.

Em sua função principal, crie e preencha uma matriz de objetos **Animal**. Metade dela será de objetos **Dog (cachorro)** e a outra metade será de objetos **Cat (gato)**. No final da execução do programa, faça um loop nessa matriz e exclua todos os animais. Você deve excluir diretamente cães e gatos como `Animals`. Os destrutores 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 precisa testar se suas cópias são cópias profundas!



```
int main()
{
    const Animal* j = new Dog();
    const Animal* i = new Cat();

    delete j; // não deve criar um vazamento
    excluir i;
    ...
    retornar 0;
}
```

Implemente e entregue mais testes do que os fornecidos acima.

Capítulo V

Exercício 02: Classe abstrata

	Exercício: 02
Classe abstrata	
Diretório de entrada: <i>ex02/</i>	
Arquivos a serem entregues : Arquivos do exercício anterior + *.cpp, *.{h, hpp}	
Funções proibidas : Nenhuma	


Afinal, criar objetos animais não faz sentido. É verdade, eles não fazem 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 entrada: <i>ex03/</i>	
Arquivos a serem entregues : Makefile, main.cpp, *.cpp, *.{h, hpp}	
Funções proibidas : Nenhuma	

As interfaces não existem no C++98 (nem mesmo no C++20). Entretanto, as classes abstratas puras são comumente chamadas de interfaces. Portanto, neste último exercício, vamos tentar implementar interfaces para ter certeza de que você entendeu este módulo.

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

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

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

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

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


Implemente as classes concretas **Materia Ice** e **Cure**. Use seus nomes em letras minúsculas ("ice" para Ice, "cure" para Cure) para definir seus tipos. Obviamente, sua função membro `clone()` retornará uma nova instância do mesmo tipo (ou seja, se você clonar uma Matéria Ice, obterá uma nova Matéria Ice).

A função membro `use(ICharacter&)` será exibida:

- Gelo: `"* dispara um raio de gelo em <nome> *"`
- Cure: `"* cura as feridas de <name> *"`

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



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

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

```
classe ICharacter
{
    público:
        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 espaços, o que significa, no máximo, 4 **Materias**. O inventário está vazio na construção. Ele equipa as **Materias** no primeiro espaço vazio que encontrar. Isso significa, nesta ordem: do espaço 0 ao espaço 3. Caso eles tentem adicionar uma Matéria a um inventário cheio ou usar/desequipar uma Matéria inexistente, não faça nada (mas ainda assim, os bugs são proibidos). A função de membro `unequip()` NÃO deve excluir a Matéria!



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

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



O inventário de seu personagem poderá suportar qualquer tipo de AMateria.

Seu **caractere** deve ter um construtor que receba seu nome como parâmetro. Qualquer cópia (usando o construtor de cópia ou o 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. Obviamente, as Materias devem ser excluídas quando um Personagem for destruído.

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

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

- **learnMateria(AMateria*)**

Copia a Matéria passada como parâmetro e a armazena na memória para que possa ser clonada posteriormente. Assim como o Character, o **MateriaSource** pode conhecer no máximo 4 Materias. Elas não são necessariamente exclusivas.

- **createMateria(std::string const &)**

Retorna uma nova Matéria. Essa última é uma cópia da Matéria aprendida anteriormente pelo **MateriaSource**, cujo tipo é igual ao que foi passado como parâmetro. Retorna 0 se o tipo for desconhecido.

Em resumo, seu **MateriaSource** deve ser capaz de aprender "modelos" de Materias para criá-los quando necessário. Em seguida, você poderá gerar uma nova Matéria usando apenas uma string que identifique seu tipo.

Executando este código:

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

    ICharacter* me = new Character("me");

    AMateria* tmp;
    tmp = src->createMateria("ice");
    me->equip(tmp);
    tmp = src->createMateria("cure");
    me->equip(tmp);

    ICharacter* bob = new Character("bob");

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

    delete bob;
    delete me;
    delete src;

    return 0;
}
```

Deve produzir:

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

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



Você pode ser aprovado neste módulo sem fazer o exercício 03.