INSTITUTO FED. DE EDUCAÇÃO, CIÊNC. E TEC. DE PERNAMBUCO
CURSO: TEC. EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS
DISCIPLINA: ALGORITMOS E ESTRUTURAS DE DADOS
PROFESSOR: RAMIDE DANTAS
ASSUNTO: C++: HERANÇA

Aluno (a):			
Matrícula:		**Data:**	

Prática 02

OBS: Essa prática faz uso de conhecimentos exercitados na Prática 1.

Parte 1: Preparação

Passo 1: Crie um novo projeto chamado Pratica2.

Passo 2: Crie um novo arquivo fonte nesse projeto chamado **pratica2.cpp**.

> Esse arquivo deve conter o método `main()` da aplicação. Faça as modificações necessárias para usar a entrada/saída padrão de C++.

Passo 3: Compile a rode a aplicação para se certificar que o projeto está corretamente configurado.

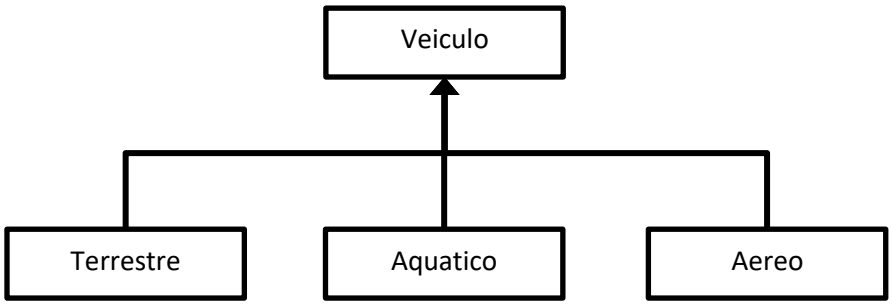
> Crie uma nova configuração de execução se preciso.

Parte 2: Criando uma hierarquia de classes simples

Passo 1: Crie um par de arquivos, como descrito a seguir:

> **veiculo.h** Contém as definições de classes usadas na prática.
>
> **veiculo.cpp** Contém as implementações de métodos dessas classes.

Passo 2: Em **veiculo.h**, codifique a seguinte hierarquia de classes:



> C++ permite declarar várias classes no mesmo arquivo. Use herança pública entre as classes, como no exemplo a seguir:

```
classe Subclasse : public Superclasse {
        ...
};
```

Passo 3: Inclua o arquivo **veiculo.h** em **pratica2.cpp**. Dentro do método `main()`, crie uma instância local de cada classe, usando a sintaxe a seguir:

```
...
int main() {
        Veiculo v1;
        ...
}
```

Passo 4: Compile e teste a aplicação.

Nesse ponto não deve haver problemas com a compilação e execução.

Passo 5: Adicione uma propriedade chamada `nome` com tipo `string` à classe `Veiculo` com nível de visibilidade `protected`.

Passo 6: Crie um construtor público no corpo da classe `Veiculo`, que receba um parâmetro do tipo `const char *` e o use para inicializar a propriedade `nome`.

Use o construtor para informar ao usuário via console que o objeto foi criado, dizendo o nome do objeto. Faça as modificações necessárias em **veiculo.h** para isso.

Passo 7: Compile e teste a aplicação.

Verifique que devem ocorrer erros de compilação causados pela falta de construtores padrão nas subclasses. Como o construtor padrão da superclasse (`Veiculo`) deixou de ser gerado implicitamente (devido ao passo 6), as subclasses também ficaram sem construtores padrão.

Passo 8: Adicione construtores públicos nas subclasses, os quais recebem também o nome do veículo como parâmetro.

Use a sintaxe a seguir para acionar o construtor da superclasse, repassando o nome do veículo:

```
classe Subclasse : public Superclasse {
public:
        Subclasse(<params>) : Superclasse (<params>) {
                ...
        }
};
```

Passo 9: No método `main()` de **pratica2.cpp**; coloque nomes apropriados na instanciação:

```
...
int main() {
        Veiculo v1("v1");
        ...
}
```

Passo 10: Compile e teste a aplicação novamente.

Nesse ponto não deve haver mais erros de compilação ou execução.

Parte 3: Especializando as subclasses

Passo 1: Adicione as seguintes propriedades às subclasses `Terrestre`, `Aquatico` e `Aereo`:

Classe	Nome	Tipo	Visibilidade	Default	Descrição
`Terrestre`	`cap_pass`	int	Privado	5	Número máximo de passageiros.
`Aquatico`	`carga_max`	float	Privado	10	Carga máxima em toneladas.
`Aereo`	`vel_max`	float	Privado	100	Velocidade máxima em km/h.

Inicialize cada propriedade no construtor de sua respectiva classe com o valor default.

Passo 2: Crie métodos especializados `get()` e `set()` para cada uma dessas propriedades.

Defina os métodos em **veiculo.h** e os implemente em **veiculo.cpp**.

Passo 3: Na função `main()` em **pratica2.cpp**, utilize o seguinte trecho de código:

```
Veiculo * terr, * aqua, * aereo;

terr = new Terrestre("VT1");
terr->setCapacidadeMax(45);

aqua = new Aquatico("VQ1");
aqua->setCargaMax(12.5);

aereo = new Aereo("VA1");
aereo->setVelocidadeMax(1040.5);
```

Passo 4: Compile e teste a aplicação.

Nesse ponto devem ocorrer erros de compilação na tentativa de acessar os métodos das subclasses (por exemplo, `setCapacidadeMax()`) a partir de ponteiros para a superclasse (`Veiculo * terr`). Para corrigir isso é preciso realizar um *cast*.

Passo 5: Corrija o código acima realizando o *cast* do ponteiro para a subclasse correta:

```
Superclasse * ponteiro = new Subclasse();

((Subclasse *)ponteiro)->metodoSubclasse();
```

Passo 6: Compile e teste a aplicação, verificando se os métodos funcionam corretamente.

Tipos de *cast* em C++

O sintaxe tradicional de *cast* `x = (tipo)y` deve ser evitada pois é ambígua. C++ introduziu novos tipos de *cast* com operadores próprios e comportamentos bem definidos:

- `x = static_cast<tipo>(y)`: realiza o *cast* com conversão implícita de tipos **sem** verificação dinâmica (similar ao *cast* tradicional);
- `x = dynamic_cast<tipo>(y)`: verifica dinamicamente se `y` pode ser convertido para `tipo`, do contrário retorna 0;
- `x = const_cast<tipo>(y)`: usado para anular o efeito de `const` nas propriedades de `y`; usar esporadicamente.
- `x = reinterpret_cast<tipo>(y)`: *cast* bit-a-bit; usado em conversões de ponteiros, por exemplo; não seguro, só use se souber o que está fazendo.

Parte 4: Trabalhando com métodos virtuais

Passo 1: Adicione o método público `mover()` a todas as classes (incluindo `Veiculo`) usando a seguinte assinatura:

```
void mover();
```

Defina o método em **veiculo.h** e seu corpo em **veiculo.cpp**. No corpo o método deve informar ao usuário que o veiculo foi movido, dizendo o tipo de veículo por exemplo, "Veiculo terrestre <*nome*> moveu".

Passo 2: Chame método `mover()` para todos os objetos criados no passo 3 da parte 4, sem realizar *cast*. Por exemplo:

```
terr->mover();
```

Passo 3: Compile e teste a aplicação.

Verifique pela saída gerada que o método `mover()` chamado depende do tipo estático do ponteiro (no caso `Veiculo`), e não do tipo real do objeto.

Passo 4: Adicione a palavra reservada `virtual` as definições e implementações de `mover()`:

```
virtual void mover();
```

Passo 5: Compile e teste a aplicação.

Verifique que agora os métodos corretos de cada classe são chamados.

Passo 6: Destrua os objetos criados na função `main()` usando `delete`.

Verifique que ocorreu o mesmo fenômeno que antes, agora com o destrutor. Isso pode causar problemas pois o objeto não será desalocado corretamente. Por exemplo, um subobjeto criado dinamicamente na subclasse (como o *array* da prática 1) pode deixar de ser desalocado, causando vazamento de memória.

Passo 7: Crie destrutores virtuais em cada classe, sinalizando ao usuário a destruição:

```
class Objeto {
public:
         virtual ~Objeto() { ... }
};
```

Passo 8: Compile e teste a aplicação.

Verifique que agora foram chamados os destrutores corretos das subclasses. O destrutor da superclasse também é chamado em seguida.

Passo 9: Torne a classe `Veiculo` abstrata: `mover()` não deve ter implementação:

```
virtual void mover() = 0;
```

Passo 10: Faça os ajustes necessários na função `main()`, compile e teste a aplicação.

Parte 5: Herança múltipla

Passo 1: Crie uma classe chamada `Anfibio`, a qual herda publicamente de `Terrestre` e `Aquatico`.

Passo 2: Modifique as classes `Terrestre` e `Aquatico` de forma que a herança delas seja pública e virtual, como na sintaxe a seguir:

```
classe Subclasse : public virtual Superclasse { ... };
```

A herança virtual evita a duplicação de subobjetos `Veiculo` dentro de `Anfibio`, do contrário o compilador criaria duas cópias e não saberia qual usar.

Passo 3: Crie construtores protegidos sem parâmetros nas classes `Terrestre` e `Aquatico`.

Por serem protegidos, esses construtores só serão acessíveis de dentro das subclasses (no caso, `Anfibio`). Nesses novos construtores, chame o construtor de `Veiculo` com um parâmetro qualquer; nos construtores das subclasses essa chamada será ignorada (isso é uma fonte potencial de bugs, por isso escondemos os construtores).

Passo 4: Na classe `Anfibio`, adicione o construtor a seguir:

```
Anfibio (const char * nome) : Veiculo(nome), Terrestre(), Aquatico() {}
```

Veja que esse construtor chama explicitamente o construtor da superclasse geral `Veiculo` para inicializar o seu subobjeto virtual. Em seguida chama os construtores protegidos de `Terrestre` e `Aquatico` para inicializar suas respectivas propriedades.

Passo 5: Sobrescreva o método `mover()` em `Anfibio`, fazendo com que sejam chamados os métodos `mover()` de `Terrestre` e `Aquatico` nessa ordem.

Use `Terrestre::mover()` e `Aquatico::mover()`. Se esse método não for criado o compilador lançará um erro. (Não é necessário chamar algum dos métodos `mover()`)

Passo 6: Na função `main()` em **pratica2.cpp**, substitua os *casts* usados anteriormente por:

```
dynamic_cast<NovoTipo>(var)->metodo();
```

Devido ao uso de herança virtual, o compilador não permite *casts* estáticos da superclasse `Veiculo` para as subclasses virtuais `Terrestre` e `Aquatico`. Note que como não mexemos na classe `Aereo`, o compilador não aponta erro nesse *cast*.

Passo 7: Crie o ponteiro `Veiculo * anfi` na função `main()` de **pratica2.cpp**.

Inicialize esse ponteiro com uma instância de `Anfibio` usando `new`. Chame o método `mover()` dessa instância em seguida.

Passo 8: Compile e teste a aplicação, verificando a saída gerada.