Exercício:

Expanda a classe Pacote desenvolvida no exercício do slide 386 sobrecarregando o operador de extração de fluxo para inicializar um objeto da classe, também sobrecarregue o operador de inserção de fluxo para imprimir o endereço do remetente e o endereço do destinatário, considerando o padrão utilizado pelos correios.

UNIVASF

Polimorfismo

Um detalhe muito importante que viabilizará o polimorfismo em tempo de execução é o fato de que um ponteiro para uma classe base pode apontar para uma classe derivada.

O exemplo a seguir se utiliza desta característica.

```
#include <iostream>
using namespace std;
class Base {
  public:
    void func(){
      cout << "Esta eh func() de base" << endl;
    }
};
```

```
class Derivada1: public Base{
  public:
    void func2(){
      cout << "Esta eh func2() de derivada1" << endl;
    }
};
class Derivada2: public Base{
  public:
    void func2(){
      cout << "Esta eh func2() de derivada2" << endl;
    }
};
int main(){
  Base *ptr;
  Derivada1 o1;
  Derivada2 o2;
  ptr = &o1;
  ptr->func();
  ptr = &o2;
  ptr->func2(); //Erro: no matching for call to ‘base::func2()’
  return 0;
}
```

Polimorfismo

Conforme vimos anteriormente, podemos sobrescrever (redefinir) uma função-membro de uma classe base em uma classe derivada. Porém, uma curiosidade ocorre quando apontamos para uma classe derivada que redefine uma função de sua classe base, com um ponteiro para sua classe base, e utilizamos o ponteiro para invocar a função-membro sobrescrita.

```
#include <iostream>
using namespace std;
class Base {
  public:
    void func(){
      cout << "Esta eh func() de base" << endl;
    }
};
```

```
class Derivada1: public Base{
  public:
    void func(){
      cout << "Esta eh func() de derivada1" << endl;
    }
};
class Derivada2: public Base{
  public:
    void func(){
      cout << "Esta eh func() de derivada2" << endl;
    }
};
int main(){
  Base *ptr;
  Derivada1 o1;
  Derivada2 o2;
  ptr = &o1;
  ptr->func();
  ptr = &o2;
  ptr->func();
  return 0;
}
```

Polimorfismo

Percebemos que a função-membro invocada é a da classe base.

A questão é que poderíamos desejar definir uma classe base que especificasse apenas uma interface comum às suas classes derivadas.

Em outra palavras, no exemplo a seguir, esperaríamos as seguintes saídas:

Esta eh func() de derivada1

Esta eh func() de derivada2

```
#include <iostream>
using namespace std;
class Base {
  public:
     void func() { cout << "Esta eh func() de base" << endl; } };
class Derivada1: public Base {
  public:
    void func() { cout << "Esta eh func() de derivada1" << endl; } };
class Derivada2: public Base {
  public:
    void func() { cout << "Esta eh func() de derivada2" << endl; } };
int main()
{
  Base *ptr;
  Derivada1 o1;
  Derivada2 o2;
  ptr = &o1;
  ptr->func();
  ptr = &o2;
  ptr->func();
  return 0;
}
```

Polimorfismo

A linguagem C++ possibilita esta manipulação, disponibilizando o conceito de função virtual.

Uma função-membro virtual é uma função-membro definida em uma classe base e redefinida em uma de suas classes derivadas e quando um ponteiro para a classe base apontar para a classe derivada, em questão, ao se invocar a função-membro será executada a função redefinida.

Para se especificar uma função-membro como virtual se utiliza a palavra reservada *virtual*, a qual precede o protótipo da função no arquivo .h da classe.

Vamos analisar um exemplo.

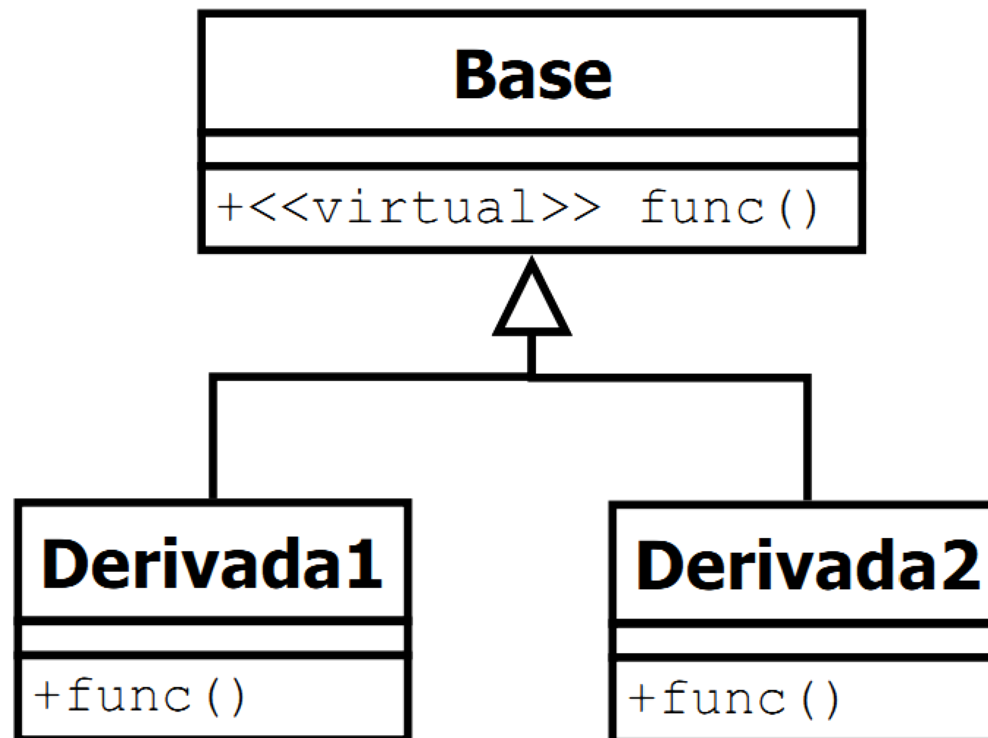
UNIVASF

```
#include <iostream>
using namespace std;
class Base {
  public:
     virtual void func() { cout << "Esta eh func() de base" << endl; } };
class Derivada1: public Base {
  public:
    void func() { cout << "Esta eh func() de derivada1" << endl; } };
class Derivada2: public Base {
  public:
    void func() { cout << "Esta eh func() de derivada2" << endl; } };
int main()
{
  Base *ptr;
  Derivada1 o1;
  Derivada2 o2;
  ptr = &o1;
  ptr->func();
  ptr = &o2;
  ptr->func();
  return 0;
}
```


UNIVASF

Polimorfismo

Exemplo de uma função-membro virtual em um diagrama de classes UML.



Polimorfismo

Podemos aplicar este conceito utilizando alocação dinâmica de memória e gerarmos esta solução:

```
#include <iostream>
using namespace std;
class Base {
  public:
    virtual void func(){
      cout << "Esta eh func() de base" << endl;
    }
};
class Derivada1: public Base{
  public:
    void func(){
      cout << "Esta eh func() de derivada1" << endl;
    }
};
```

Linguagem de Programação C++

```
class Derivada2: public Base{
  public:
    void func(){
      cout << "Esta eh func() de derivada2" << endl;
    }
};
int main(){
  Base *ptr;
  ptr = new Base;
  ptr->func();
  delete ptr;
  ptr = new Derivada1;
  ptr->func();
  delete ptr;
  ptr = new Derivada2;
  ptr->func();
  delete ptr;
  return 0;
}
```


UNIVASF

Linguagem de Programação C++

Exercício:

Com base no que vimos, use a hierarquia de herança Pacote criada no exercício do slide 386 e aperfeiçoada no exercício do slide 400 para criar um programa que exibe as informações de endereço e calcule os custos de entrega de vários pacotes. O programa deve conter um vetor de ponteiros pacotes para objetos das classes PacoteDoisDias e PacoteNoite e manipulá-lo através das operações disponibilizadas pelo seguinte menu.

Digite:

1 – Inserir um pacote para entrega em dois dias;

2 – Inserir um pacote para entrega a noite;

3 – Imprimir os endereços para postagem e custos da postagem;

4 – Imprimir o custo total das postagens.

Para simplificar as manipulações considere que no máximo o vetor possuirá 100 ponteiros para pacotes.

UNIVASF